Verify SCR requirements using XSPIN model checking to Elevator Case Study

Kenneth K.C. Cheung
Department of Computer Science,
University of Toronto

February 21, 2001

Abstract

Model checking analysis can improve the correctness of tabular software specification. Software Specification defines what the system does. This documentation used in each stage of software engineering. The design of the software will base on what software specification state. Code development and test evaluate according to the software specification. These usages show the importance of software specification. The correctness of the specification can affect the quality of the software directly. Formal method has been introduced to improve the correctness of the software specification. But, the requirement of most of the formal method technique requires mathematical training or theorem proving skills becomes the barrier towards practical use. Tabular notation is introduced to break this barrier. It tries to abstract the property of what the system does rather than formalize all the information. This notation requires less mathematical training or theorem proving skills, but the power of formality decrease. SCR (Software Cost Reduction) is a tabular notation, which have a tool call SCRTool. This tool has a feature called verifier, which allow the tabular specification covert to formal specification for spin to check safety critical and liveness property. Thus, the power of formality maintained and practical usability increased. In this project, a case study for a two-floor elevator is used to try out this feature.

1. Introduction

SCR (Software Cost Reduction) is a formal notion that uses to specify event-driven systems. It was originally developed from Naval Research Lab (NRL) researchers for document the requirements of the operational flight program of the US Navy's A-7 aircraft [1]. A wide range of practical systems uses the SCR notation to specify the requirements afterward. These systems mainly use the notation to specify functional properties of the system. It also has been used in other systems such as security, safety, real-time, and fault-tolerance [2]. SCR notation describes the system as a black box, which only show what the system do, without the detail of how the system built. The specification generate by SCR notation can be used to describe the required behavior of a system or component of the system precisely. Any hardware or software can be built by following the specification. (The specification can be use to build any hardware or software system.)

SCR used state machine idea to specify the state of the system, where a state represents the situation of the software or system and transactions between different states are trigger by events. But, SCR calls them a little different. A state is called a mode and event

identifies as change of state of a variable. For example, button, a boolean variable that represent a on/off light switch, change from false to true is a event. A transition table is used to store these information.

Four Variable Model has adapted by SCR for specifying the required system behavior. The input and output interaction of the system formalized using monitor variable as input from environment, control variable as output to control the environment, and term variable for internal control. These variables relate to each other using different tables, such as condition table, event table. Further analysis, such as consistence and completeness check, can be perform with the specification

Naval Research Lab developed SCRTool to support SCR notation and provide different tools to analysis the specification. Specification Editor used to view and edit specification that use SCR notation. Different analysis tools, such as type check, syntax check, consistent check, etc, can run from CC Checker to analysis the specification. Dependency Graph Browser will show the dependency relationship between different variables and modes. Simulator provides features for setting up and running simulation with different scenario to verify the specification. But, evaluate by running simulation is not confidence enough. Model checking can be use in the next step to increase the confidence of correctness. A feature call Verifier within SCRTool can translate SCR notation to Promela such that it can be run on SPIN. In this project, a two-floor elevator will be model using SCR notation and using Verifier in SCRTool to convert it to Promela. Then, XSPIN is used to verify different safety critical and liveness properties. More detail about SCR and SCR notation will be explain in section 2. And, the case study will be shown in section 3.

2. Software Cost Reduction (SCR)

2.1 Parnas Four Variable Model

Four Variable Model is one of the main formal frameworks for SCR notation [3]. It describes the required system behavior as a set of logical relations on four sets of variables. The four sets of variables are monitored and controlled variable and input and out data items. Monitor variables are quantities in the environment that influence system behavior. On the other hand, Controlled variable are quantities in the environment that the system control. Input data items represent the values that the input device read. And, Output data items represent the values that write to output device. For example, in the case study, the sensor that monitors the elevator door is a monitor variable and the mechanism that control the elevator door to be open or close is a controlled variable. The value open and close are input and output data items.

A set of relations is used to link between these four sets of variables. NAT and REQ are two relations describe on monitored and control quantities. NAT describes the constraints imposed by physical laws and system environment. REQ describes the relation between the monitored and controlled quantities that the system must enforce to produce required behavior. IN relation describe the mapping between the monitored quantities to the input

data items. And, OUT relation describe the mapping between the data items to the controlled quantities. More information about Four Variable Model can be found in Parnas Four Variable Model [4].

2.2 SCR Notation

To apply Four Variable Model practically and concisely, SCR introduced mode classes, terms, conditions, and events. A mode classes is like a state machine, whose values are modes. Each mode is a system mode, which represents the current state of the entire system. In complex systems, several modes classes can be define in SCR such that they can operate in parallel. A term is an internal variable that used in the computation. A condition is a predicate defined on one or more state variable, such as monitored or controlled variable, mode class or term, at some point in time. For example, when elevator is on floor one and the door is open, the condition of ready is true. SCR show this condition as " efloor = 1 and edoor = open". An event occur when state variable change its value. For example, Floor one button change from off to on is an event. There are different types of event define in SCR. A primitive event is a change in state variable. If the state variable happen to be a monitor variable, it will become an input event. A condition event is a primitive event, which occurs only when the condition is satisfied. In SCR, event denoted by the notation "@T", "@F", and "@C". "@T(variable)" means the variable will become true in the next state. "@F(variable)" means the variable will become false in the next state and "@C(variable)" means the variable will change value in the next state. A more formal definition of primitive event is "@T(a) = not a and a" and conditional event is "@T(a) WHEN b = not a and a and b". A floor one button becomes true only when a person present "@T(fb1) WHEN person = true" is a conditional event example.

The functionality of the system can be defined by expressing the values of controlled variables as a function that take monitored variables as input, and used mode classes and terms for computation. The question "how all variables, mode classes, terms, conditions and events be link together" still needed to answer to complete the REQ relation between monitored and controlled variable. SCR solving this question by using tables. A SCR specification consists of a set of directories and a set of tables defining the system behavior. Firstly, variables will be define in one of the directories. A directory defines what the variables are and each type of variable has its own directory, such as type, constant, variable and mode class directory. Each variable belongs to only one directory. After define the variables in the directories, each class mode, term, and controlled variable need to setup its own table. A mode transition table is used to specify the transition between states in a mode class. Transitions between the states have to trigger by events. Without any event, the system will not change its states. This is very similar to start a car. The driver has to turn the key (an event) before the engine start (idle mode to ready mode). In each transition, the mode transition table requires original and destination mode and the event to specify a full transition. And, all transitions in the table have to be deterministic. The value of each term and controlled variable can be specify on either condition table or event table.

A condition table defines all possible values of a variable. When a transition happens, the value of the variable will change according to the condition table. Thus, the condition table has to include all the possible value of the variable in different condition. The condition table can be specify into mode or modeless table. A mode condition table, rows represent modes and columns represent the possible values of the variable. Each cell is a condition for a particular mode. When the condition is true in a particular mode, the value, which locates at the bottom of the same column, is selected. In a modeless condition table, the layout is the same as mode condition table except mode is not exist in modeless condition table. Each condition is a global condition. As soon as the condition is true, the value will be selected regardless which mode the system is in. No matter which type of condition type is used. Completeness of the table has to fulfil.

An event table is a little different. It describes the change of value of the variable by event. The value of the variable will not change its value even the condition is true. The variable only changes its value when events that specify in the event table happens. The selected value represents the value for next state not the current state. There are mode and modeless event table as well. The format of mode event table and condition table basically is the same. The only difference is the cell. In mode event table, a cell represents a event expression instead. Modeless event table and modeless condition table has the same difference as well. Event table is not as strict as condition. Specify all value of the variable is not necessary. It only requires to specify events that cause it to change value. But, both of the tables have to be disjoint. Otherwise, the specification will become non-deterministic.

Environment assumption is the last topic that is going to discuss about SCR notation. It is similar to the NAT relation from four variable model that discussed in pervious section. These assumptions are constraints from the environment that will hold in all states. For example, the elevator can only be locate in either floor one or floor two. It can not be locate in different floor at the same time. In SCR notation, this kind of environment assumptions can be specify in environment assertion directory. More detail examples of the variables, table, mode class that just discuss can be found in the elevator case study in section 3.

2.3 More about SCRTool

As pervious section mention, SCRTool is a tool that developed by NRL to support SCR notation. It general feature has been introduced in introduction section. In this section, Verification feature of SCRTool is the focus. Since, this is the motivation of this case study. SCRTool provide different features for analysis the specification. Different checks, such as syntax, type, disjointness, names and variables checks, etc, can be perform from CC Checker, one of the main component of SCRTool. These checks will help to catch error in describing the specification in SCR notation. This is only the static property of the specification. The dynamic property of the specification has not been check yet. In this case study, two analysis tool has been use to check the dynamic property of the specification. They are Simulator in SCRTool and XSPIN model checker. These are two different type of analysis tool.

Simulator is a tool within SCRTool that capable to setup, edit, and view simulation. Simulation is a property-based analysis. It is able to detect errors in property-based specification, which describe the required system properties. Different scenarios are run within the simulation to verify the required system properties has been specify correctly. And, XSPIN is a model checker that can verify properties of the specification, too. It is an operational (model-based) tool, which exams every state of the specification. This is a stronger verification than simulation. If the specification can pass in both tests, it is pretty confidence to say the properties satisfy. But, these tests require different specification, property and operational based.

SCR used a dual-language approach [9], which allows a specification covert between property and operational based specification. SCRTool has a component called Verifier. It can convert a SCR specification into Promela with out losing any semantic or logical information. This feature allows the same specification to be test and verify with two analysis techniques. A better specification will able to generate. The case study in section 3 will show how this process can be done.

3. Case Study: elevator model

In this case study, a simple two-floor elevator is going to be model. The model has to make sure the following properties satisfy.

1) If there is a request to a particular floor, the elevator will eventually service it.
2) Elevator never moves with its door open.

The first step to achieve these goals is to specify the system precisely using SCR notation. Then, apply the check features in CC Checker to make sure the static property of the specification is correct. Dynamic property of the specification will be check next by simulate the system behavior using Simulator follow by verify the system properties using XSPIN model checker.

3.1 The SCR specification

In this elevator model, nine variables and one mode class have been used. In nine variables, there are five monitored variables, two terms, and two controlled variables. These variables generate by the following four steps:

1) Identify and describe the controlled variables.
2) Identify and describe the monitored variables.
3) Identify and describe the mode classes.
4) Specify the relation between monitored and controlled variable.

3.1.1 Identify controlled variables

Controlled variables are environment quantities that the elevator can control. Elevators usually able to control which floor they locate and open or close the elevator door. These controllable environment quantities represent by cDoor and cFloor. They are listed in

Figure 1 along with their types, initial values and descriptions. cDoor is used to specify the elevator door is going to open or close. It is assigned to yDoor type. All the types are listed in Figure 2. yDoor type is a user define type which has value open or close only. Thus, cDoor can indicate the elevator controller is going to close or open the door. cFloor is used to specify which floor the elevator are moving to. It is assigned to yFloor user define type. Look at yFloor type. It is a range of Integer from zero to one. This range is expandable depend how many floor the elevator support. One of the reasons why yFloor type is introduced rather than using Integer type is because of the expandability just mention. Another reason is the retune cause by the translation from SCR notation to Promela. This problem can refer to section 3.4 for more detail. Variables, cFloor and cDoor, are part of the environment, but, they are controlled by the system.

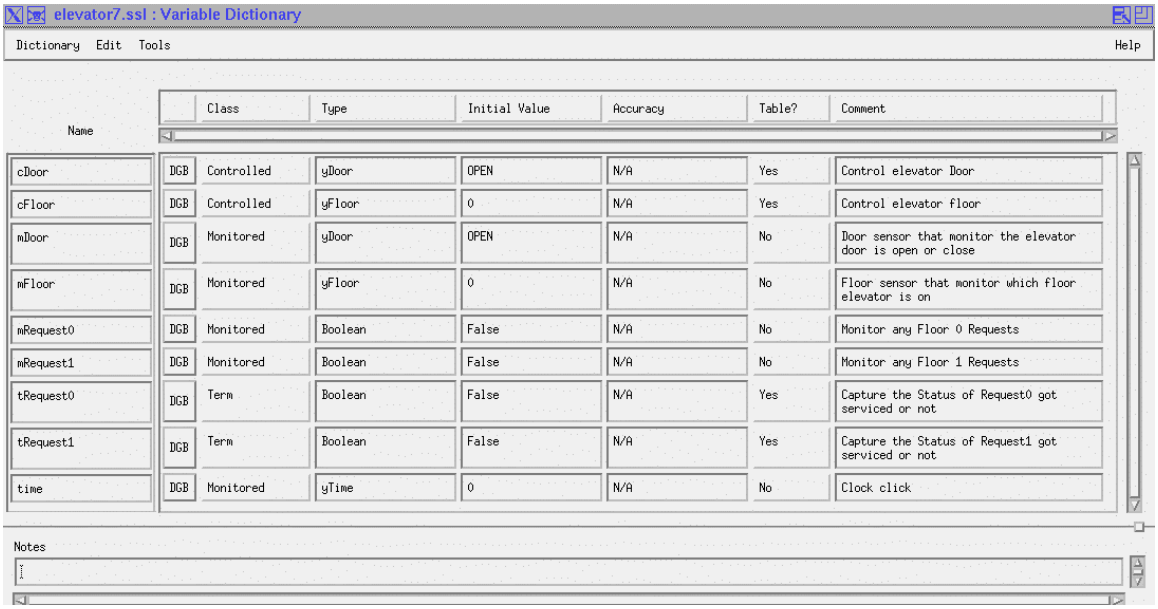


Figure 1: Variable Directory for elevator model

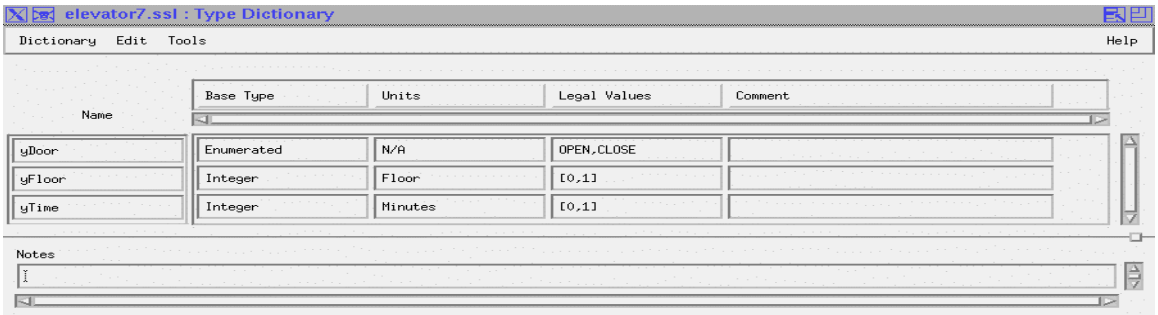


Figure 2: Type Directory for elevator model

3.1.2 Identify monitored variables

The environment quantities that the system monitors are going to be identify next. The key to select monitor variables is to think them as the input of the system. Imagine the

human skin. It monitors the environment temperate all the time. Human brain will take this information as input and react to it by contract the holes on the skin if the temperate is low. Thus, The system will monitor the change of these variables and react according to the specification. Elevator needs information about the current status of its door, location, and service request to determine what to do. These concepts captured by monitor variables, mRequest0, mRequest1, mFloor, mDoor, and time. They are listed in Figure 1. mRequest0 and mRequest1 monitor elevator's button. If the button, which request elevator to floor zero, is on, mRequest0 will be true. Same situation happen in mRequest1. They provide service request information to the elevator. mDoor is used to monitor the status of the elevator door and mFloor is used to monitor which floor the elevate locate. At last, the monitor variable, time, represent the time click that is happening in the real world.

3.1.3 Identify mode classes

Now, the input and output of the elevator system has been identified. The general framework of the elevator is done. The internal specification of the system is going to start from identifying mode classes. This is the engine of the elevator controller. As section 2 mention, a mode class is very similar to a state machine. Thus, the design of the state machine for the elevator controller is the starting point at this stage. When the design is done, it can translate into mode class directory and mode transition table. A state in the machine will become a mode in SCR and a transition will record as event expression in SCR. Then, summarize all the mode information into mode class directory and all the transition information into mode transition table. The mode class directory of the elevator model is shown in Figure 3 and mode transition table is shown in Figure 4. Terms may introduce in this stage, too.

The elevator model used one mode class only. As a reminder, SCR allow more than one mode class run in parallel. Status is the name of the mode class and modes are other possible states in the state machine. The initial state of this machine is Idle0. The transitions between these modes are specified in the mode transition table. In the first row of mode transition table, it shows the transition move from Idle0 mode to EClose0 mode when request to floor one event occur or at timeout request to floor one was requested. More detail about the model can be found in Figure 3 and 4.

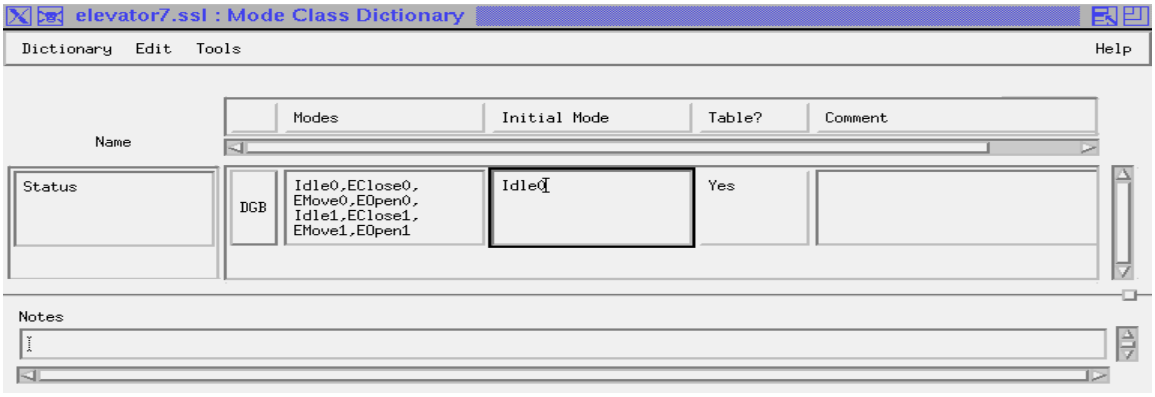


Figure 3: Mode Class Directory for elevator model

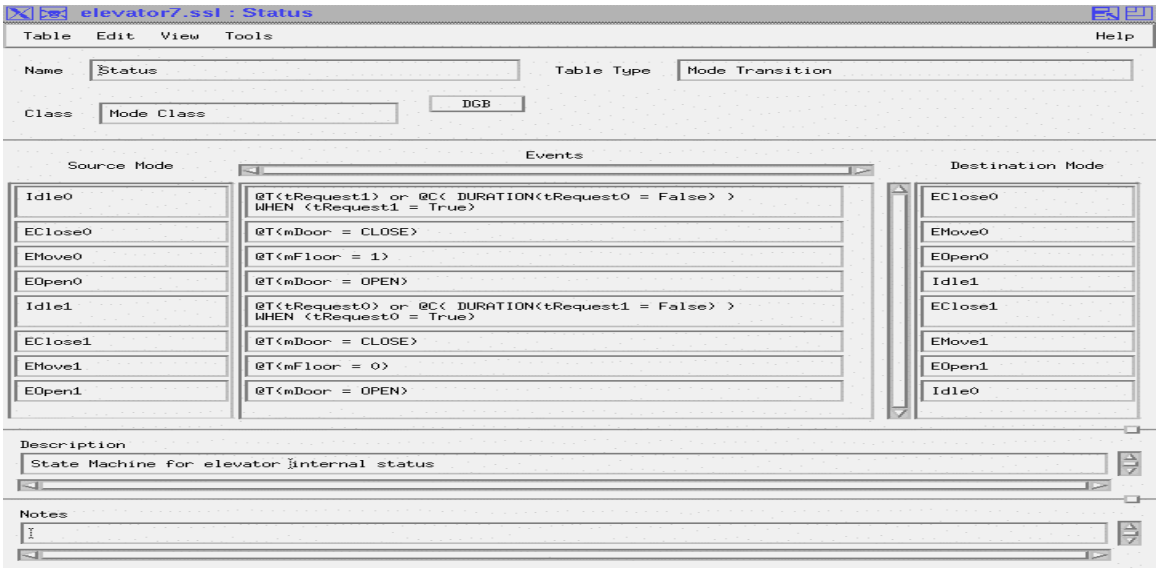


Figure 4: Mode Transition Table for elevator model

3.1.4 Specify the REQ relation

After the mode class is designed. The link between monitored variable and controlled variable still need to establish. They are going to linked by condition table and event table. Each controlled variable and terms require one condition or event table for the specification to be complete. Sometime the decision to choose the type of table for a particular controlled variable or term could be hard. A very clear understanding about the model that is modeling and SCR notation may require. Two condition table and two event table has been used in the elevator specification. Controlled variables, cFloor and cDoor, are specified using control table. And, terms, tRequest0 and tRequest1, are specified using event table. tRequest0 and tRequest1 choose to use event table because of the natural dependent property of these variable. The computation of these variables depend on monitored variables, mRequest0 and mRequest1, who is the event trigger the transition of state. cFloor and cDoor choose condition table because their result are depend on which mode they are in. Thus, condition table is the best fit. See Figure 5 and 6 for some example.

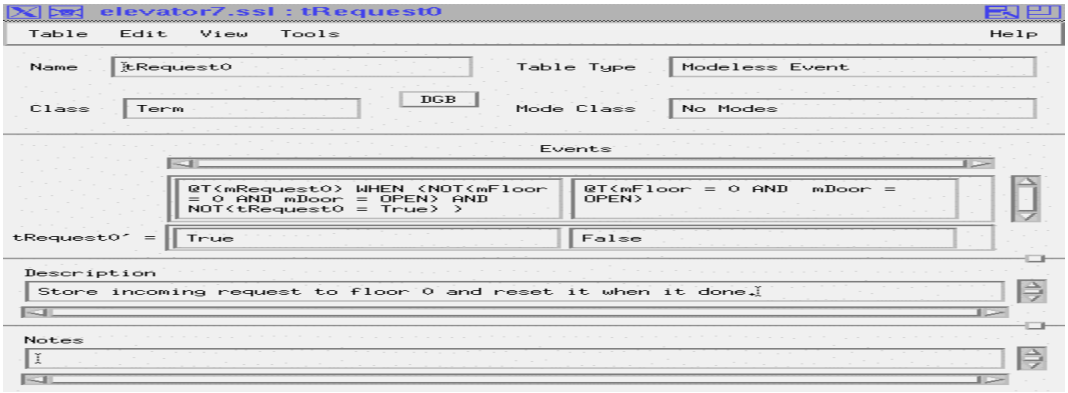


Figure 5: Event Table for tReques0

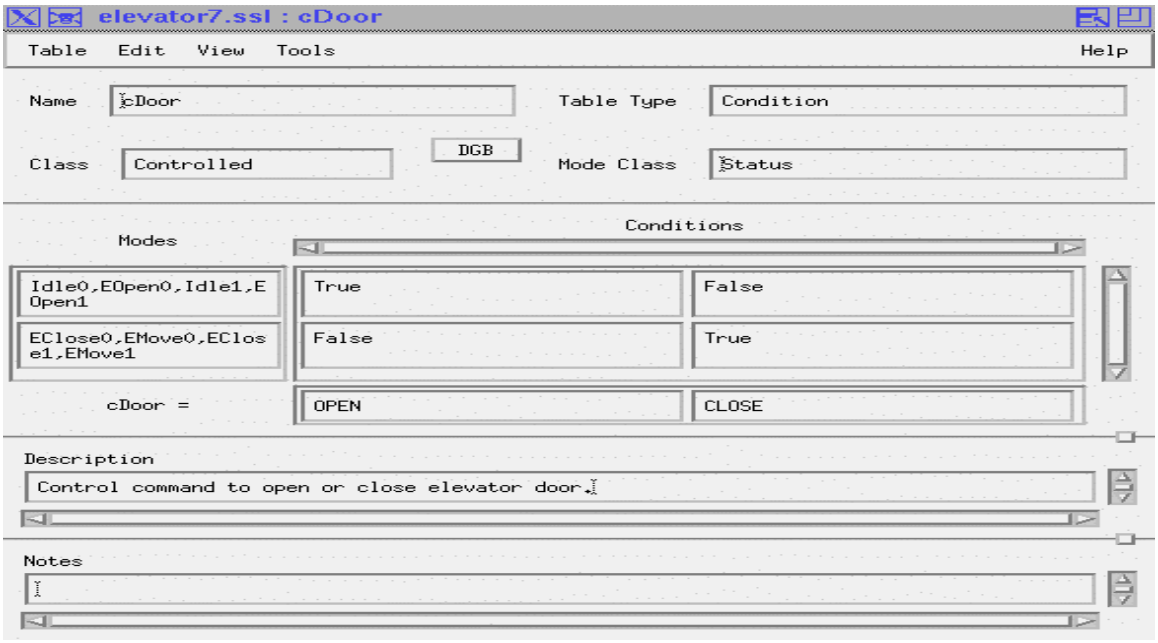


Figure 6: Condition Table for cDoor

3.1.5 Specify the environment assumptions

The elevator specification hasn't complete yet. To complete the elevator specification, addition of environment assumptions, the NAT relation, is require. The environment assumptions are constraints or physical laws in the environment. In the elevator model, a couple of simple assumptions have applied. The assumption of the next state of monitored variable, mDoor, is equal to the controlled variable, cDoor, is the first assumption. This assumption is shown as "mDoor' = cDoor" in environment assertion directory on Figure 7. The symbol ' means the next state in SCR. The logic behind this assumption is the natural relation of the elevator controller and the elevator door. Usually, the elevator controller commands the elevator door to close. Then, the elevator door control itself to close the door. There must be some input to the sensor telling the elevator controller that the door is already close. If this relation is invalid, the elevator must have a hardware failure.

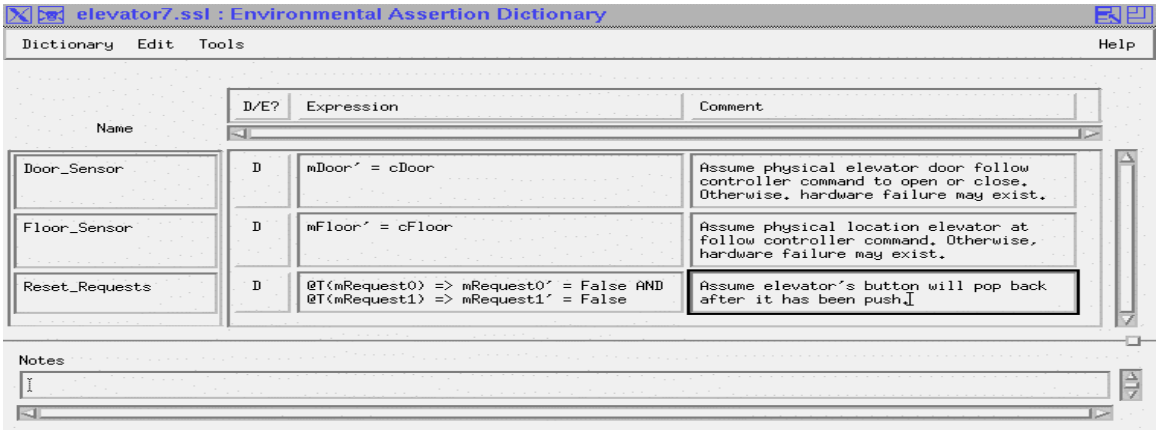


Figure 7: Environmental Assertion Directory

The second assumption has the similar reason. There must be an input to the sensor, mFloor, from the controller, cFloor, to tell the elevator controller that the elevator arrived certain floor. The last assumption is from the physical design of a button. When a button is push, a signal is generated. Then, the human hand leaves the button. The button will pop up automatically without generate any meaningful signal. Since, there is no unset feature to floor request button usually. But, SCR requires to consider both cases separately. To guarantee this natural relation exist. Thus, the third assumption is added. At this stage, the initial elevator specification is completed. The full SCR elevator specification locates in appendix A.

3.2 CC Checker Analysis

CC Checker can improve the correctness of the initial elevator specification by providing different checks. Some of the errors found by the CC checker are easy to solve, such as syntax error or unique name error. The next level of errors is logical errors, such as coverage and consistency errors. They are a little harder to solve. In the process of finding the solution, review the logics in the specification demand a lot of work and thinking. Sometime, the solution may only affect one variable. But, indeed most of time, the solution involves changes in a couple of variables. The hardest errors found from CC Checker are dependency errors. The cause of this error is form the cycle dependency relationship between variables and mode classes. One of the example from this elevator case study is the design of serial event generate from one external event. Some serial internal events from the elevator controller want to trigger from a floor request event. CC Checker found cycle dependency between a variable and the mode class. Errors like these are hard to find solution sometime. In worst case, the solution may require to change a large portion in the specification. The Dependency Graph Browser in SCRTool will able to help analyzing this problem by showing a dependency graph for all variables and mode class. See Figure 8. The static analysis of the elevator specification is done.

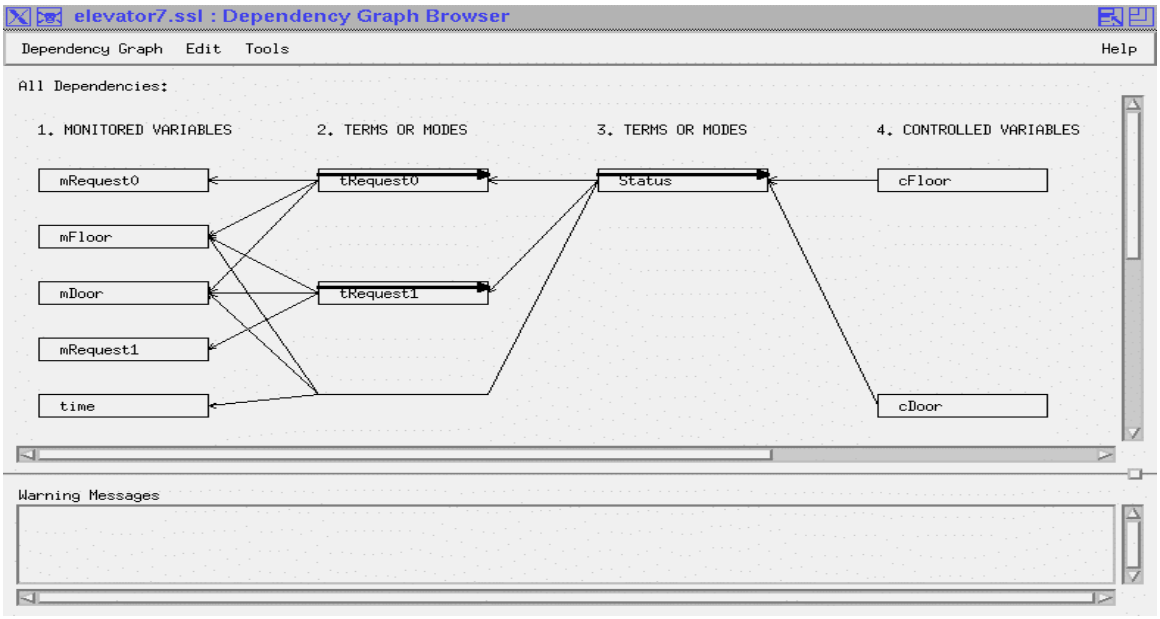


Figure 8: Dependency Graph

3.3 Running simulation

Next, the dynamic behavior of the specification is going to be check. Simulation is going to test the elevator specification first. Then. Model Checker is going to be use in the next section to verify some properties of the specification. In this test, some typical and special scenario has been planned to challenge the behavior of the elevator specification. One of the typical scenarios is the elevator at floor zero waiting for request and floor request to another floor or same floor event happen. This scenario can test the behavior of the elevator in a normal situation. One of the special scenarios setup a floor zero request event when the elevator is in the process servicing floor one request. The special scenarios are quite helpful. It found some design errors in the mode class where the elevator stuck on floor one after it serviced floor one request rather than carry on the service queue to serve floor zero request. Another problem found from the simulation is the button push and pop situation. It is fixed by add an environment assertion.

3.4 Verify the elevator specification with XSPIN

The last test in this case study used XSPIN Model Checker to verify the validity of described properties. Although a lot of thought test has been ran on the initial elevator specification already, but these testes are not able to test all case. Test the specification using model checker allow all possible case to be tested such that the achievement of required properties can be confirm confidently.

Verifier, one of the components of SCRTool, converts the elevator specification from SCR notation to Promela. This tool covert most of the specification correctly. But, some modification to the Promela still required before verifying the properties. The modified Promela can be found in Appendix B. There are couple main modifications. Environment assertion (NAT relation) has not covert into Promela. They have to add manually. Pick one of the environment assertions as an example. Monitored variable, mDoor, next state value should equal to the value of controlled variable, cDoor. The statement “mDoor_NEW = cDoor_NEW” is added to conserve the assertion. Another environment assertion has been indirectly converted because of the design of value picking for monitored variables, which can be found in the guard of selecting value of monitored variables, mRequest0 and mRequest1.

The second main modification found in converting integer. Verifier covert any integer type variables into a very large test range. This problem causes the cases that model checker need to check grow rapidly. The XSPIN verify results are out of memory. This problem happened in two areas in this case study. The first problem found in the variables, mFloor and cFloor. It has been solve by adding a user define type, yFloor, to limit the range of an integer. The same situation found in the monitored variable, time. It has been solve using similar technique. The time variable only has two acceptable values, 0 and 1, to represent the time change. The test results show this solution successful removed a large number of states.

Another problem area is in the event generation section. Verifier will convert all monitored variable into the event generate section. The convertion ignore the information from the assertion specify. For example, the monitored variable, mDoor, and controlled variable, cDoor, problem that mention previously. The value mDoor should come from cDoor. So, these events do not necessary require to generate. This causes a lot of work in debugging the counter example. Another problem found in this area is the fairness of generating possible value for monitored variables. Thus, the event generation section needs to be analysis carefully. There is no systematic way to check these errors. These errors really depend on the case. A good understanding in XSPIN may require solving these problems.

Although a lot of Promela modification may need in the translation. But the model checker verification process has successfully found some missing mistake. For example, when verifying the first property, "If there is a request to a particular floor, the elevator will eventually service it", there are couple errors found that has not been detected before. The result show a weakness in the event tables of tRequest0 and tRequest1, where they may change to true even the request is in the same floor that the elevate locate. This problem causes some logical problem on resetting them back to false. Another problem found by the model checker is very similar to the one that found by simulation in the special scenario. The elevator stuck on floor one after serving floor one request and it is not capable to server the floor zero request. Different approaches have been tried, such as record pervious history and generate internal event. Unlucky these approach have not successful solve the problem. Thus, time is introduced as a solution. It bypass the problem with the system define method "DURATION(expression)", which record the time for the expression specify in the parameter. This is not the best way to solve this problem. Serial event should be a better solution for this kind of problem, where a event can easily invoke another event. RSML, a tabular notation, provide some support in this idea [22].

After all these thought testes, the final specification is very confident to announce the required properties has been achieve. The detail about the properties test and the result can be found in Appendix B.

4. Discussions and Conclusions

This elevator case study successfully shown the approach is practically possible. The use of tabular SCR notation require less mathematic and theorem proving training and maintain correctness level using model checking technique. The SCR elevator specification provides a clear understanding of what the system is and the result from model checking support the elevator satisfy the required properties. But, there are some concerns need to notice in this approach.

In the process of designing the initial specification, it is easily overstate or understate the initial specification. For example, in the case study, the first initial specification has controlled variables, dOpen, dClose, up and down. It was overstated because it is not necessary need to specify each possible elevator action as a Boolean variable. The actions

can easily simply into two controlled variables, cFloor and cDoor. The original specification has nothing wrong. But, a complicated specification may generate. This could cause difficulty in analyzing and specifying REQ relation.

The skill level requirement for mathematical and theorem proving may decrease in the initial specification. But, eventually the require skill level become the same. The translation from SCR notation to Promela requires a good understanding in Promela. To learn Promela, advance mathematical and theorem proving training is require. Thus, it is not necessary decreasing the require skill overall.

The simplicity of SCR notation allows it to specify large system. On the other hand, there is a limitation from model checking. The number of possible state needs to test by model checker increase rapidly for large system. Without the model checker, simulation in SCR is not strong enough to test properties. This dependency relationship may limit the use of this technique in a system that is not too big.

The elevator case study brings out some limitation of SCR notation. One of the limitations of SCR notation is the expressiveness. SCR does not support parallel event. In the elevator case study, floor request and time click are parallel events. The natural relationship is lost during the specification process. Another helpful support is the use of serial events. SCR seems a little weak in this use. Difficult situation have been found in setting up serial event in the elevator case study. More support on these concepts and pervious points may improve the popularity of use of this technique.

Appendix A: Full SCR two-floors elevator model

Table 1: Contents of the Specification

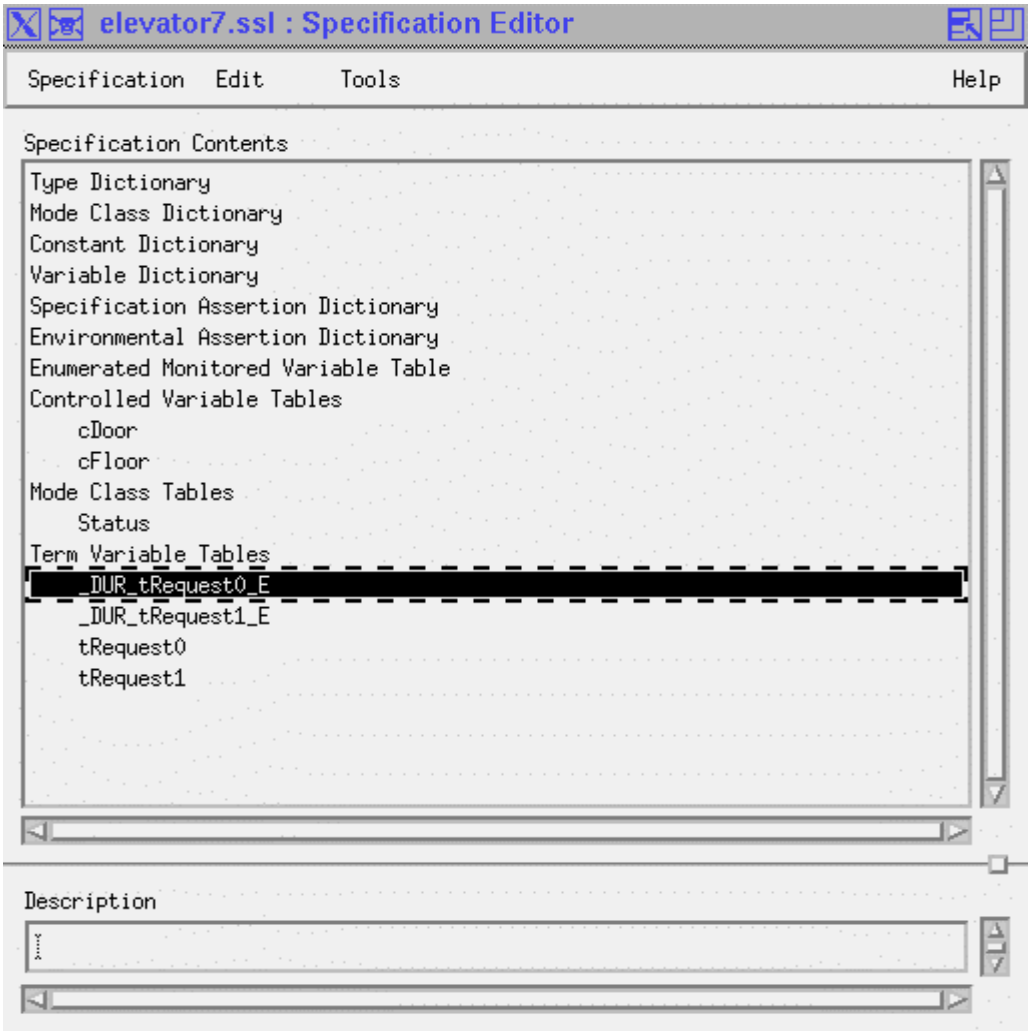


Table 2: Type Directory

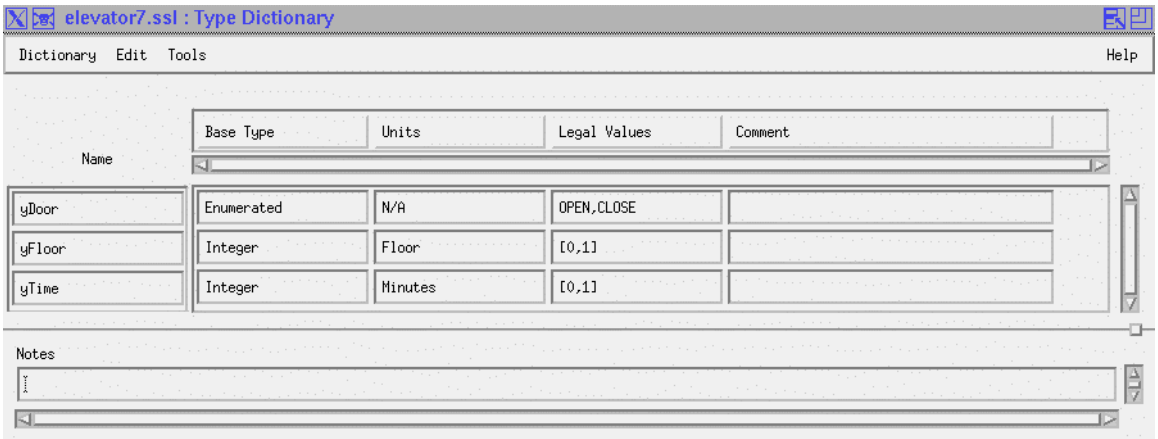


Table 3: Variable Directory

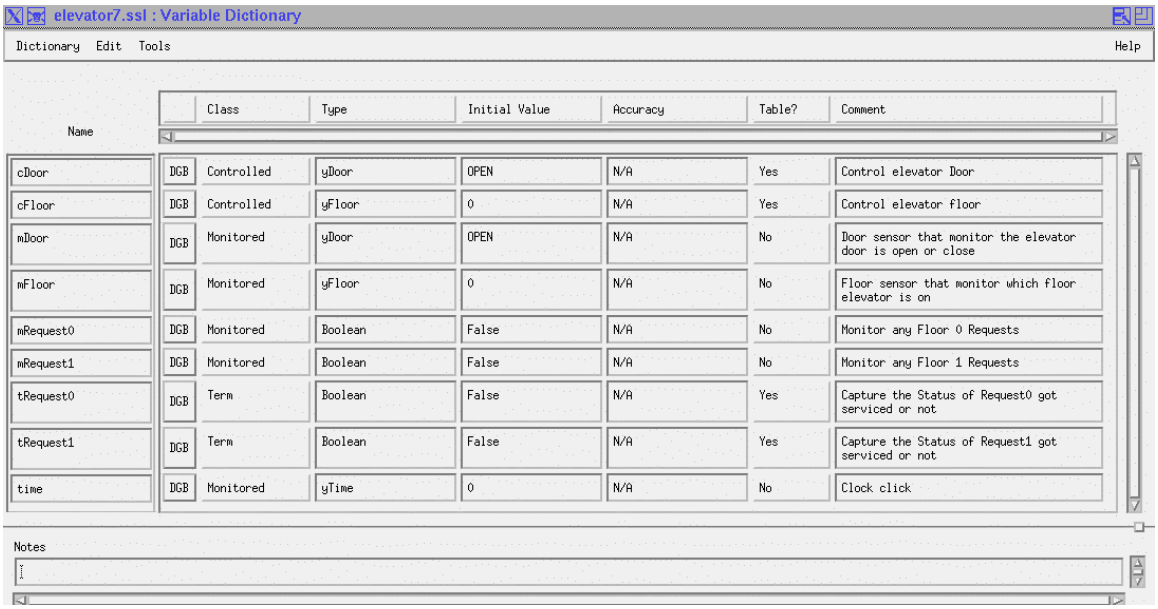


Table 4: Mode Transition Table

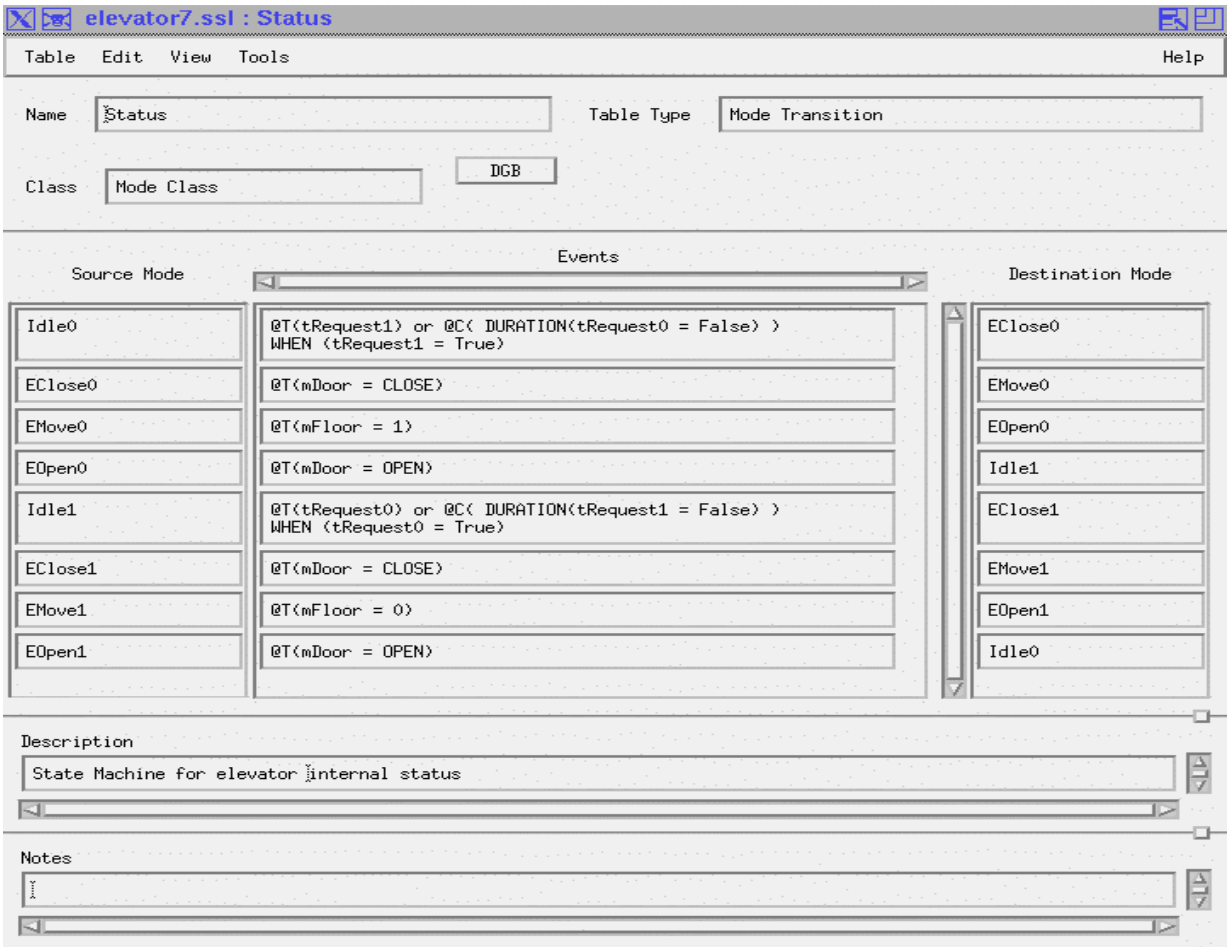


Table 5: Mode Class Directory

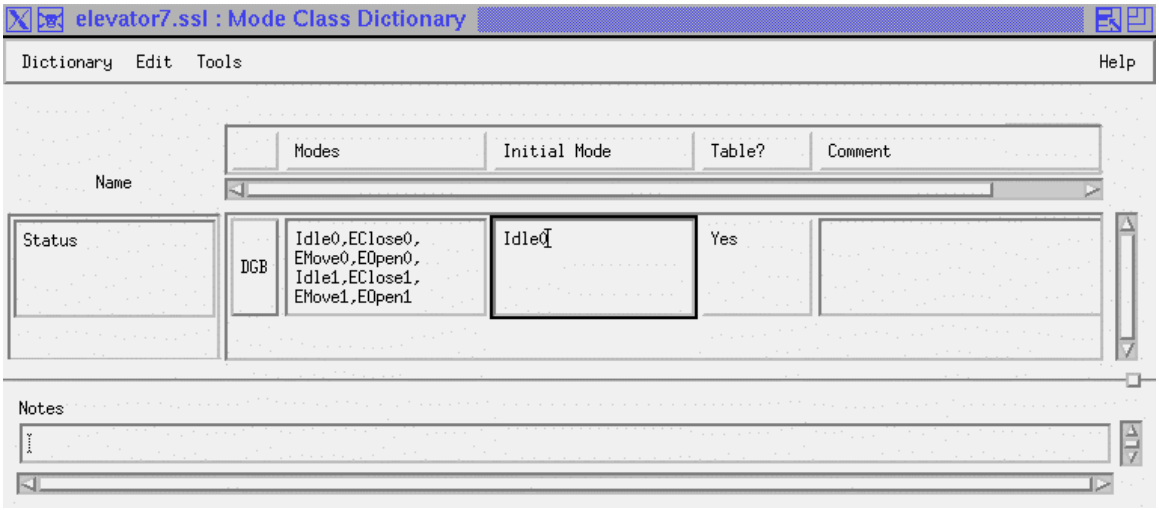


Table 6: Event Table for tRequest0

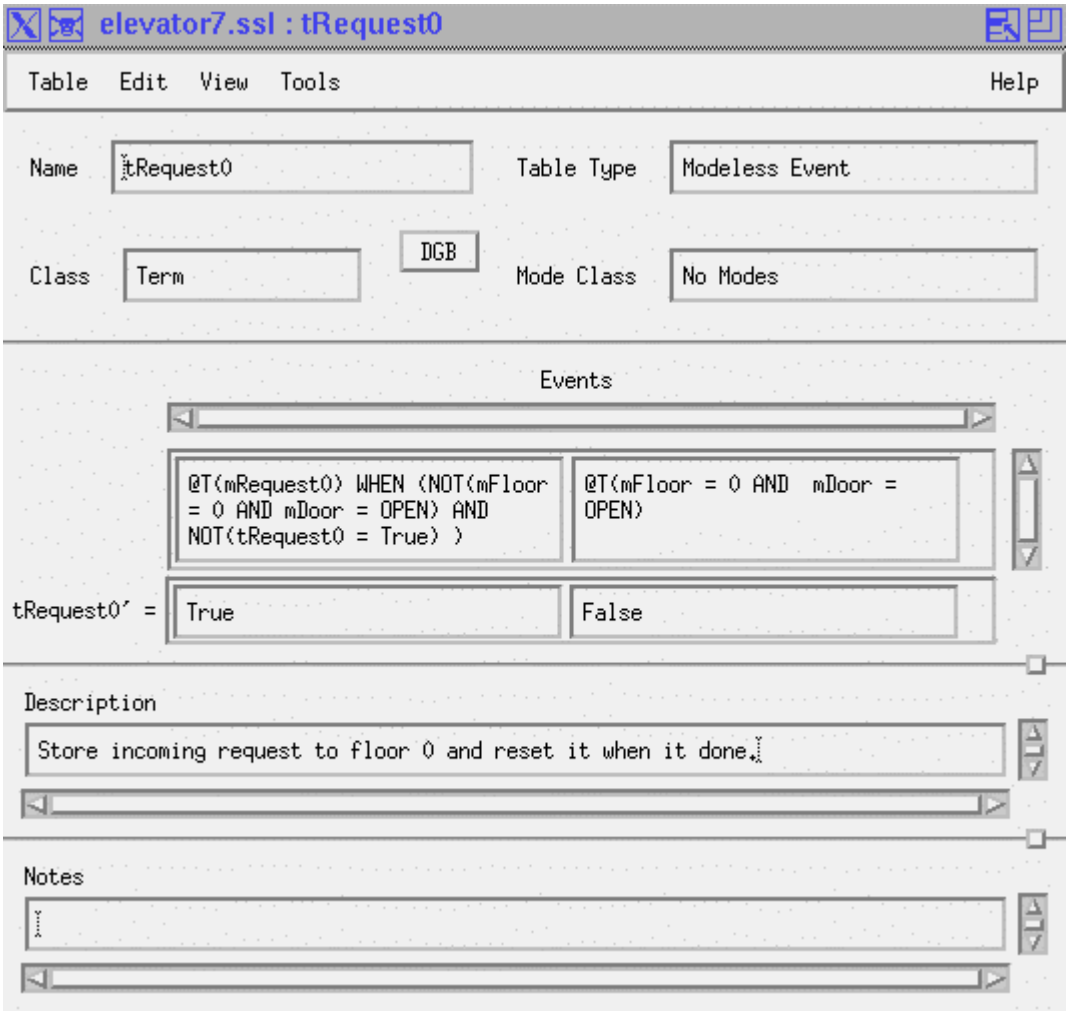


Table 7: Event Table for tRequest1

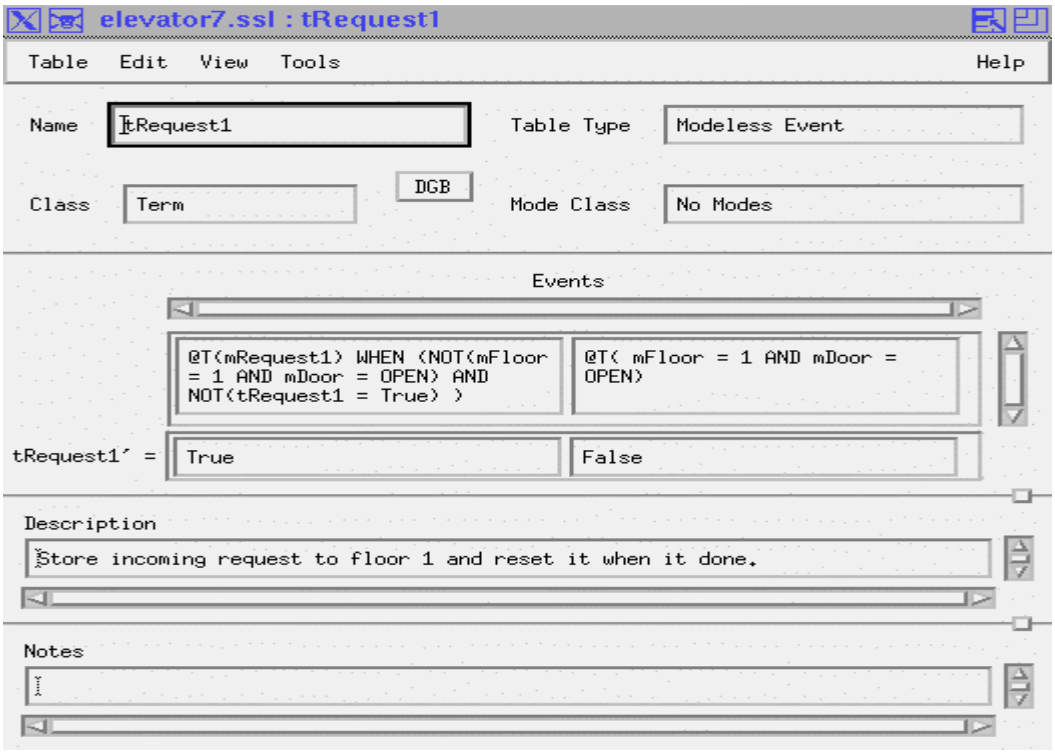


Table 8: Condition Table for cDoor

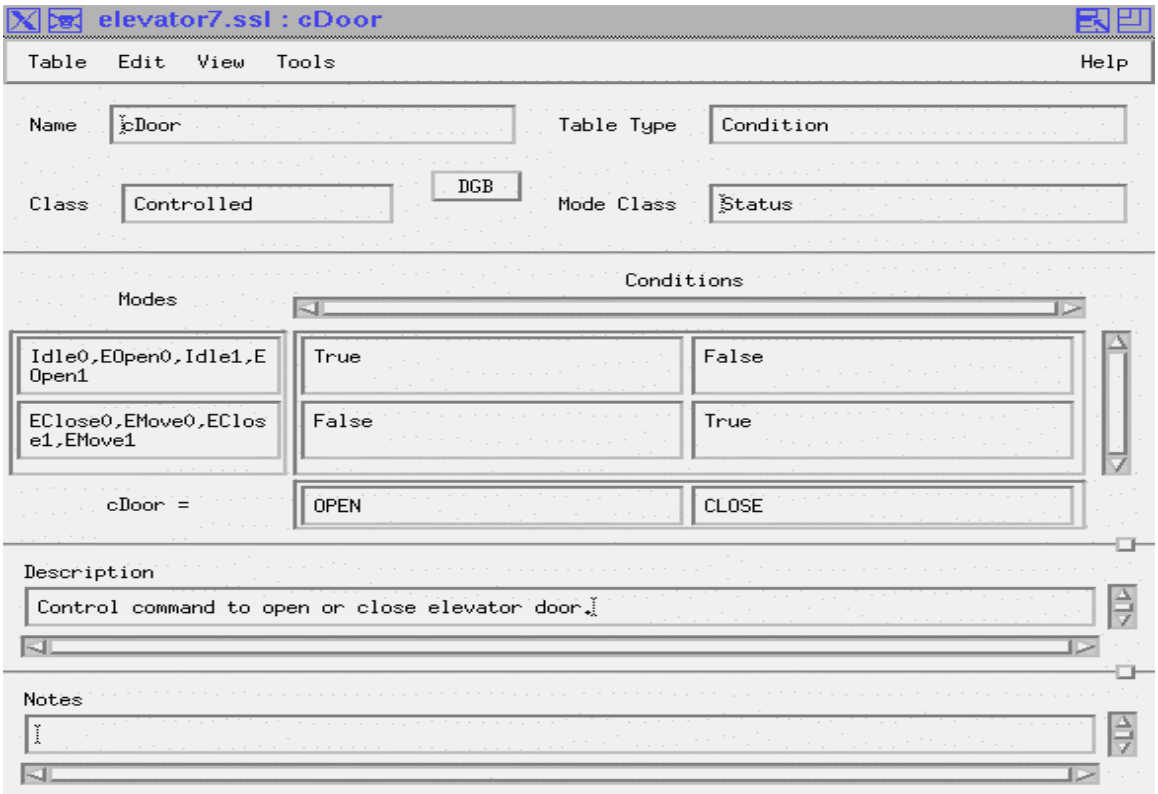


Table 9: Condition Table for cFloor

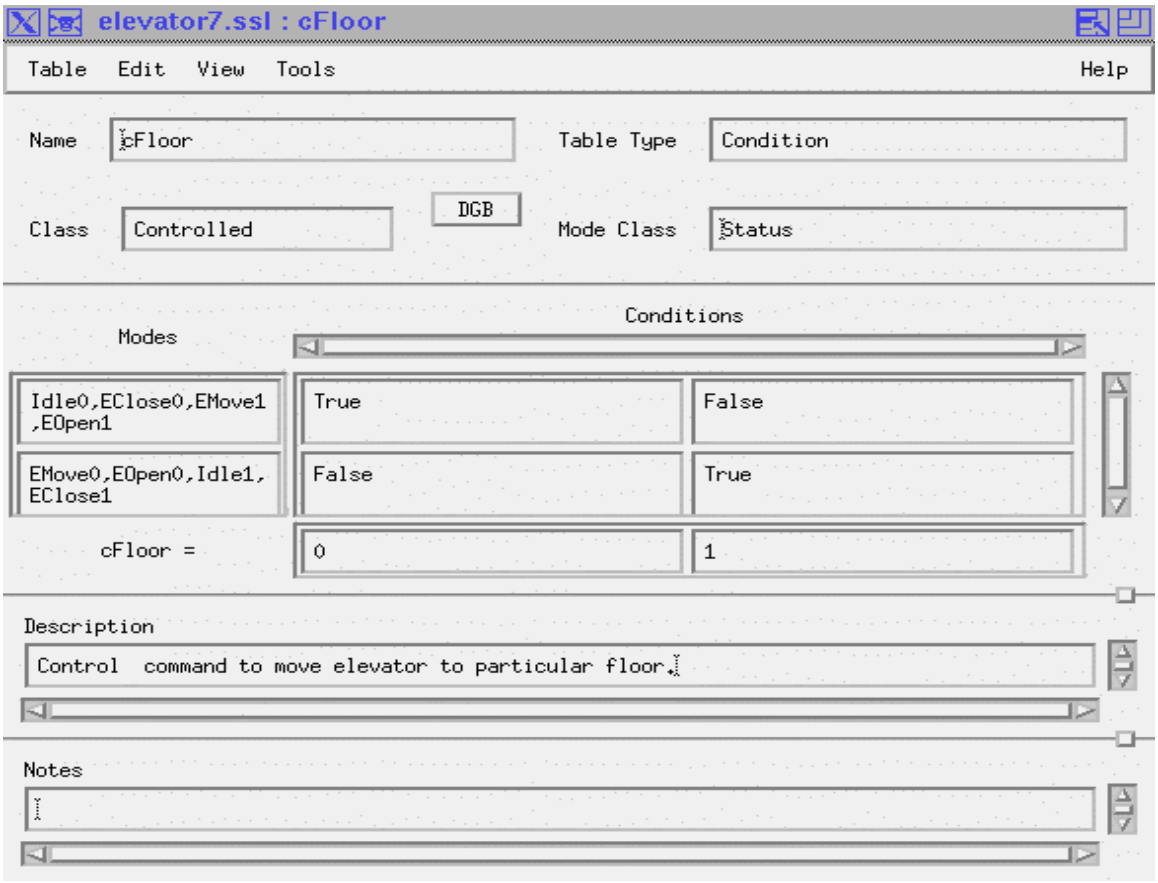
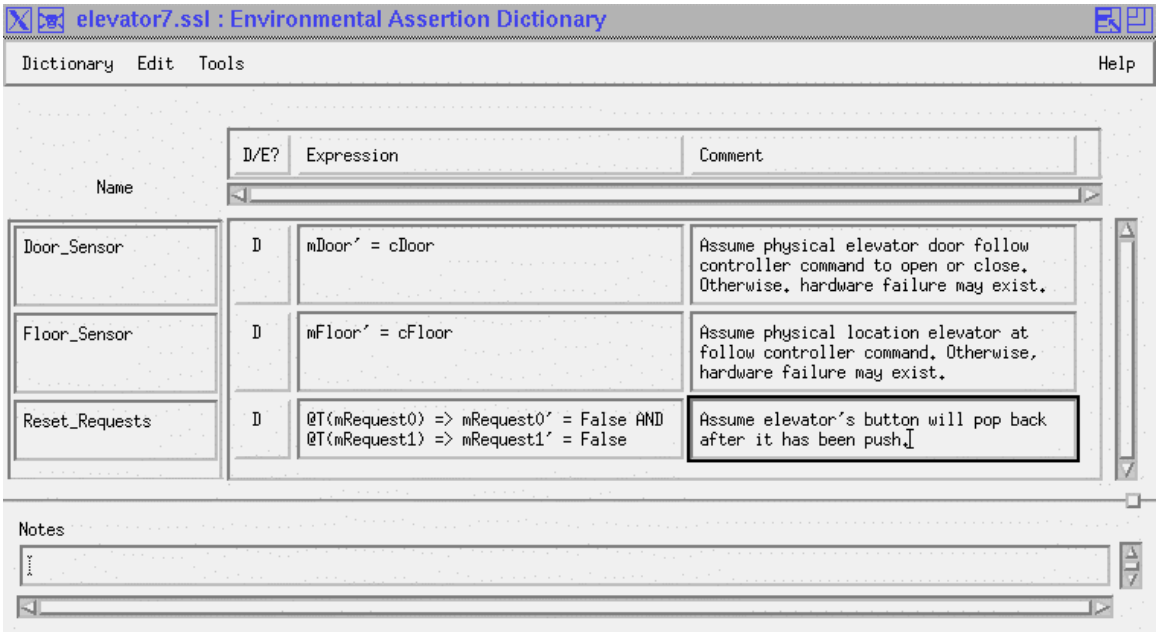


Table 10: Environment Assertion Directory



Appendix B:
Modified Promela elevator model and test result from XSPIN

Promela:

```
/* This file contains the PROMELA/spin version of an SCRTool
specification. */
/* It is created by SCRTool and automatically fed to Xspin. */
/* However, this file was left in the file elevator7.ssl.spin */
/* for you to use, look at, etc. */


/* When executed, the program outputs a line begining with 'MVC ' */
/* (monitored variable change) each time a monitored variable is */
/* updated, indicating the name of that variable and its new value. */


/*****************************/
/*     numeric constants     */
/*****************************/
#define TRUE 1
#define FALSE 0
#define Idle0 0
#define EClose0 1
#define EMove0 2
#define EOpen0 3
#define Idle1 4
#define EClose1 5
#define EMove1 6
#define EOpen1 7
#define OPEN 0
#define CLOSE 1
#define WTime 0

/*******************************/
/*     variable declarations     */
/*******************************/
int _DUR_tRequest0_E_OLD = 0;
int _DUR_tRequest0_E_NEW = 0;
int _DUR_tRequest1_E_OLD = 0;
int _DUR_tRequest1_E_NEW = 0;
byte cDoor_NEW = OPEN;
int cFloor_NEW = 0;
byte mDoor_OLD = OPEN;
byte mDoor_NEW = OPEN;
int mFloor_OLD = 0;
int mFloor_NEW = 0;
bool mRequest0_OLD = FALSE;
bool mRequest0_NEW = FALSE;
bool mRequest1_OLD = FALSE;
bool mRequest1_NEW = FALSE;
bool tRequest0_OLD = FALSE;
bool tRequest0_NEW = FALSE;
bool tRequest1_OLD = FALSE;
bool tRequest1_NEW = FALSE;
```

```
int time_OLD = 0;
int time_NEW = 0;
byte Status_OLD = Idle0;
byte Status_NEW = Idle0;

/**********************/
/*    init function    */
/**********************/
init {


/*************************************************************************/
    /*     update each variable that has no specified initial value
*/

/*************************************************************************/
  /*****************************/
  /*    main processing loop    */
  /*****************************/
  do

  ::

    /*******************************************/
    /*     "any state" specification asserts     */
    /*******************************************/


/*************************************************************************/
    /*     update each variable and mode class for this state change
*/

/*************************************************************************/
    d_step { /* update in one "step" */
      _DUR_tRequest0_E_OLD = _DUR_tRequest0_E_NEW;
      _DUR_tRequest1_E_OLD = _DUR_tRequest1_E_NEW;
      mDoor_OLD = mDoor_NEW;
      mFloor_OLD = mFloor_NEW;
      mRequest0_OLD = mRequest0_NEW;
      mRequest1_OLD = mRequest1_NEW;
      tRequest0_OLD = tRequest0_NEW;
      tRequest1_OLD = tRequest1_NEW;
      time_OLD = time_NEW;
      Status_OLD = Status_NEW;
    } /* close state update d_step */

    /*******************************************/
    /*    simulate monitored variable changes    */
    /*******************************************/
    if
    ::if
      /* toggle the current value */
      :: (mRequest0_OLD) -> mRequest0_NEW = FALSE
      :: (!mRequest0_OLD) -> mRequest0_NEW = TRUE
      fi;
      d_step { /* print new value, no state effect */
        if
```

```
        :: (mRequest0_NEW) -> printf("MVC mRequest0 TRUE\n")
        :: (!mRequest0_NEW) -> printf("MVC mRequest0 FALSE\n")
        fi
      }
    ::if
      /* toggle the current value */
      :: (mRequest1_OLD) -> mRequest1_NEW = FALSE
      :: (!mRequest1_OLD) -> mRequest1_NEW = TRUE
      fi;
      d_step { /* print new value, no state effect */
        if
        :: (mRequest1_NEW) -> printf("MVC mRequest1 TRUE\n")
        :: (!mRequest1_NEW) -> printf("MVC mRequest1 FALSE\n")
        fi
      }
    fi;

    if
      /* randomly jump to any value within the legal range of the
variable */
      :: ((time_OLD + 1) <= 1) -> time_NEW = time_OLD + 1
      :: ((time_OLD - 1) >= 0) -> time_NEW = time_OLD - 1
    fi;
    d_step { /* print new value, no state effect */
      printf("MVC time %d\n", time_NEW)
    }


    /*********************************************************/
    /*     executions of the functions in dependency order     */
    /*********************************************************/

    d_step { /* calculate in one "step" */

      /*Environment Assumption*/
      mDoor_NEW = cDoor_NEW;
      mFloor_NEW = cFloor_NEW;

      /* the PROMELA version of the tRequest0 function */
      if
      /* event: @T(mRequest0) WHEN (NOT(mFloor = 0 AND mDoor = OPEN) AND
                NOT(tRequest0 = True) ) */
      :: (mRequest0_NEW && (!mRequest0_OLD)) && ((!((mFloor_OLD == 0) &&
(mDoor_OLD == OPEN))) && (!(tRequest0_OLD == TRUE)))
             -> tRequest0_NEW = TRUE;
      /* event: @T(mFloor = 0 AND  mDoor = OPEN) */
      :: ((mFloor_NEW == 0) && (mDoor_NEW == OPEN)) && (!((mFloor_OLD ==
0) && (mDoor_OLD == OPEN)))
             -> tRequest0_NEW = FALSE;
      :: else skip;
      fi;

      /* the PROMELA version of the tRequest1 function */
      if
      /* event: @T(mRequest1) WHEN (NOT(mFloor = 1 AND mDoor = OPEN) AND
                NOT(tRequest1 = True) ) */
```

```
      :: (mRequest1_NEW && (!mRequest1_OLD)) && ((!((mFloor_OLD == 1) &&
(mDoor_OLD == OPEN))) && (!(tRequest1_OLD == TRUE)))
             -> tRequest1_NEW = TRUE;
      /* event: @T( mFloor = 1 AND mDoor = OPEN) */
      :: ((mFloor_NEW == 1) && (mDoor_NEW == OPEN)) && (!((mFloor_OLD ==
1) && (mDoor_OLD == OPEN)))
             -> tRequest1_NEW = FALSE;
      :: else skip;
      fi;

      /* the PROMELA version of the _DUR_tRequest0_E function */
      if
      /* event: @F(tRequest0 = FALSE) */
      :: (tRequest0_OLD == FALSE) && (!(tRequest0_NEW == FALSE))
             -> _DUR_tRequest0_E_NEW = 0;
      /* event: @C(time)
                WHEN (tRequest0 = FALSE)
                AND (tRequest0 = FALSE)' */
      :: ((!(time_NEW == time_OLD)) && (tRequest0_OLD == FALSE)) &&
(tRequest0_NEW == FALSE)
             -> _DUR_tRequest0_E_NEW = (_DUR_tRequest0_E_OLD + (time_NEW
- time_OLD));
      :: else skip;
      fi;

      /* the PROMELA version of the _DUR_tRequest1_E function */
      if
      /* event: @F(tRequest1 = FALSE) */
      :: (tRequest1_OLD == FALSE) && (!(tRequest1_NEW == FALSE))
             -> _DUR_tRequest1_E_NEW = 0;
      /* event: @C(time)
                WHEN (tRequest1 = FALSE)
                AND (tRequest1 = FALSE)' */
      :: ((!(time_NEW == time_OLD)) && (tRequest1_OLD == FALSE)) &&
(tRequest1_NEW == FALSE)
             -> _DUR_tRequest1_E_NEW = (_DUR_tRequest1_E_OLD + (time_NEW
- time_OLD));
      :: else skip;
      fi;

      /* the PROMELA version of the Status function */
      if
      /* modes: Idle0 */
      /* event: @T(tRequest1) or @C( DURATION(tRequest0 = False) )
                WHEN (tRequest1 = True) */
      :: ((tRequest1_NEW && (!tRequest1_OLD)) ||
((!(_DUR_tRequest0_E_NEW == _DUR_tRequest0_E_OLD)) && (tRequest1_OLD ==
TRUE))) && (Status_OLD == Idle0)
             -> Status_NEW = EClose0;
      /* modes: EClose0 */
      /* event: @T(mDoor = CLOSE) */
      :: ((mDoor_NEW == CLOSE) && (!(mDoor_OLD == CLOSE))) &&
(Status_OLD == EClose0)
             -> Status_NEW = EMove0;
      /* modes: EMove0 */
      /* event: @T(mFloor = 1) */
```

```
      :: ((mFloor_NEW == 1) && (!(mFloor_OLD == 1))) && (Status_OLD ==
EMove0)
            -> Status_NEW = EOpen0;
      /* modes: EOpen0 */
      /* event: @T(mDoor = OPEN) */
      :: ((mDoor_NEW == OPEN) && (!(mDoor_OLD == OPEN))) && (Status_OLD
== EOpen0)
            -> Status_NEW = Idle1;
      /* modes: Idle1 */
      /* event: @T(tRequest0) or @C( DURATION(tRequest1 = False) )
               WHEN (tRequest0 = True) */
      :: ((tRequest0_NEW && (!tRequest0_OLD)) ||
((!(_DUR_tRequest1_E_NEW == _DUR_tRequest1_E_OLD)) && (tRequest0_OLD ==
TRUE))) && (Status_OLD == Idle1)
            -> Status_NEW = EClose1;
      /* modes: EClose1 */
      /* event: @T(mDoor = CLOSE) */
      :: ((mDoor_NEW == CLOSE) && (!(mDoor_OLD == CLOSE))) &&
(Status_OLD == EClose1)
            -> Status_NEW = EMove1;
      /* modes: EMove1 */
      /* event: @T(mFloor = 0) */
      :: ((mFloor_NEW == 0) && (!(mFloor_OLD == 0))) && (Status_OLD ==
EMove1)
            -> Status_NEW = EOpen1;
      /* modes: EOpen1 */
      /* event: @T(mDoor = OPEN) */
      :: ((mDoor_NEW == OPEN) && (!(mDoor_OLD == OPEN))) && (Status_OLD
== EOpen1)
            -> Status_NEW = Idle0;
      :: else skip;
      fi;

      /* the PROMELA version of the cFloor function */
      if
      /* modes:     Idle0,EClose0,EMove1,EOpen1 */
      /* condition: True */
      :: ((((Status_NEW == EClose0) || (Status_NEW == EMove1)) ||
(Status_NEW == EOpen1)) || (Status_NEW == Idle0))
            -> cFloor_NEW = 0;
      /* modes:     EMove0,EOpen0,Idle1,EClose1 */
      /* condition: True */
      :: ((((Status_NEW == Idle1) || (Status_NEW == EClose1)) ||
(Status_NEW == EMove0)) || (Status_NEW == EOpen0))
            -> cFloor_NEW = 1;
      fi;

      /* the PROMELA version of the cDoor function */
      if
      /* modes:     Idle0,EOpen0,Idle1,EOpen1 */
      /* condition: True */
      :: ((((Status_NEW == Idle1) || (Status_NEW == EOpen1)) ||
(Status_NEW == Idle0)) || (Status_NEW == EOpen0))
            -> cDoor_NEW = OPEN;
      /* modes:     EClose0,EMove0,EClose1,EMove1 */
      /* condition: True */
```

```
      :: ((((Status_NEW == EClose0) || (Status_NEW == EMove1)) ||
(Status_NEW == EClose1)) || (Status_NEW == EMove0))
              -> cDoor_NEW = CLOSE;
      fi;

    } /* close calculation d_step */

    /***************************************************/
    /*   "post-initial state" specification asserts   */
    /***************************************************/

    /* The assertion will make sure the elevator never moves with its
door open*/
    assert ((mFloor_OLD == 0 && mFloor_NEW == 1 && mDoor_OLD == CLOSE &&
mDoor_NEW == CLOSE) || (mFloor_OLD == 1 && mFloor_NEW == 0 && mDoor_OLD
== CLOSE && mDoor_NEW == CLOSE) || (mFloor_OLD == mFloor_NEW))

  od /* end of main processing loop */

}
```

Property one and its result:

Property: If there is a request to a particular floor, the elevator will eventually service it. (liveness)

Note: There are two tests. Part one test Floor zero request and part two test Floor one request.

Part one specification and result:

```
#define p ( tRequest0_NEW == TRUE )
#define q ( mDoor_NEW == OPEN && mFloor_NEW == 0 )

      /*
       * Formula As Typed: []  (p -> <>  q)
       * The Never Claim Below Corresponds
       * To The Negated Formula !([]  (p -> <>  q))
       * (formalizing violations of the original)
       */

warning: for p.o. reduction to be valid the never claim must be stutter-
closed
(never claims generated from LTL formulae are stutter-closed)
(Spin Version 3.3.3 -- 21 July 1999)
      + Partial Order Reduction

Full statespace search for:
      never-claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness enabled)
      invalid endstates - (disabled by never-claim)

State-vector 60 byte, depth reached 801, errors: 0
    2217 states, stored (3113 visited)
    1027 states, matched
    4140 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 241 (resolved)
(max size 2^19 states)

2.644       memory usage (Mbyte)

unreached in proctype :init:
      (0 of 118 states)

real        0.1
user        0.0
sys         0.0

#endif
```

Part two specification and result:

```
#define p ( tRequest1_NEW == TRUE )
#define q ( mDoor_NEW == OPEN && mFloor_NEW == 1 )

      /*
       * Formula As Typed: []  (p -> <>  q)
       * The Never Claim Below Corresponds
       * To The Negated Formula !([]  (p -> <>  q))
       * (formalizing violations of the original)
       */

warning: for p.o. reduction to be valid the never claim must be stutter-
closed
(never claims generated from LTL formulae are stutter-closed)
(Spin Version 3.3.3 -- 21 July 1999)
      + Partial Order Reduction

Full statespace search for:
      never-claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness enabled)
      invalid endstates - (disabled by never-claim)

State-vector 60 byte, depth reached 801, errors: 0
    2217 states, stored (3113 visited)
    1027 states, matched
    4140 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 134 (resolved)
(max size 2^19 states)

2.644       memory usage (Mbyte)

unreached in proctype :init:
      (0 of 118 states)

real        0.1
user        0.0
sys         0.0

#endif
```

Property two and its result:

Property: The elevator never moves with its doors open (safety)

Note:

A assert statement is added to check the property. It is added at the end of the while-do loop to make sure each transition maintain the validity of the assertion. The assertion is "mDoor_NEW == CLOSE) || (mFloor_OLD == 1 && mFloor_NEW == 0 && mDoor_OLD == CLOSE && mDoor_NEW == CLOSE) || (mFloor_OLD == mFloor_NEW))". The semantic of the assertion show as follow. If the elevator maintain in the same floor, the status of the elevtor door is ignore. If the elevator change its floor, the door has to be close.

Result:

```
warning: for p.o. reduction to be valid the never claim must be stutter-
closed
(never claims generated from LTL formulae are stutter-closed)
(Spin Version 3.4.3 -- 21 December 2000)
      + Partial Order Reduction

Full statespace search for:
      never-claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness enabled)
      invalid endstates - (disabled by never-claim)

State-vector 60 byte, depth reached 901, errors: 0
    2003 states, stored (3489 visited)
    1147 states, matched
    4636 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 16 (resolved)
(max size 2^19 states)

2.644       memory usage (Mbyte)

unreached in proctype :init:
      line 262, state 119, "-end-"
      (22 of 119 states)

#endif
```

References:

[1] C. Heitmeyer, R. Bharadwaj. "Applying the SCR Requirements Method to the Light Control Case Study". Journal of Universal Computer Science (JUCS), August 2000

[2] R. Bharadwaj, C. Heitmeyer. "Developing high assurance avionics systems with the SCR requirements method". In Proc. 19th Digital Avionics Systems Conference, 7-13 October 2000, Philadelphia, PA.

[3] C. Heitmeyer, R. Jeffords, B. Labaw. "Automated Consistency Checking of Requirements Specifications". To appear in ACM Transition on Software Engineering and Methodology 5, 3, July 1996, 231-261

[4] D. Parnas, J. Madey. "Functional Documentation for Computer Systems". Science of Computer Programming, vol. 25, no. 1, pp. 41-61, OCT 1995

[5] C. Heitmeyer, J. Kirby, B. Labaw. "Tools for Formal Specification, Verification, and Validation of Requirements". To appear in Proc. COMPASS '97

[6] Constance Heitmeyer 's Web site.
http://chacs.nrl.navy.mil/personnel/heitmeyer.html

[7] V. Wiels, S. Easterbrook. "Formal Modeling of Space Shuttle Software Change Requests using SCR". To appear in the Proceedings of the Fourth International Symposium on Requirements Engineering (RE'99), Limerick, Ireland, June 7-11, 1999

[8] T. Sreemani, J. Atlee. "Feasibility of Model Checking Software Requirements: A Case Study". Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[9] C. Heitmeyer, J. Kirby, Jr., B. Labaw, M. Archer, R. Bharadwaj. "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications". IEEE Transactions on Software Engineering, Vol 24, No 11, Novemeber 1998.

[10] SCR* Toolset: The User Guide for SCRTool Version 1.7 from Naval Research Lab

[11] M. Chechik. "SC(R)3: Towars Usability of Formal Methods". University of Toronto, Department of Computer Science, Toronto. Ontario, Canada.

[12] R. Bharadwaj, C. Heitmeyer. "Model Checking Complete Requirements Specifications Using Abstraction". Automated Software Engineering, 6, 37-68, 1999, Kluwer Academic Publishers, Boston.

[13] R. Bharadwaj, C. Heitmeyer. “Hardware/Software Co-Design and Co-Validation Using the SCR Method”. In Proc. IEEE Int’l High Design Validation and Test Workshop (HLDVT’99), Nov 1999.

[14] M. Chechik, B. Devereux, S. Easterbrook. “Implementing a Multi-Valued Symbolic Model Checker”. Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

[15] R. Bharadwaj, C. Heitmeyer. “Applying the SCR Requirements Specification Method to Practical Systems: A Case Study”. Presented at The 21st Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt MD, USA Dec, 1996

[16] R. Bharadwaj, C. Heitmeyer. “Verifying SCR Requirements Specifications Using State Exploration”. In Proc. First ACM SIGPLAN Workshop on Automatic Analysis of Software, Jan 1997.

[17] C. Heitmeyer, J. Kirby, B. Labaw. “Applying the SCR Requirements Method to a Weapons Control Panel: An Experience Report”. In Proc., Formal Methods in Software Practice ’98.

[18] C. Heitmeyer. “SCR: A Practical Method for Requirements Specification”. Naval Research Laboratory, Washington, DC.

[19] C. Heitmeyer. “Using the SCR Toolset to Specify Software Requirements”. Naval Research Laboratory, Washington, DC.

[20] C. Heitmeyer, J. Kirby, B. Labaw, R. Bharadwaj. “SCR: A Toolset for Specifying and Analyzing Software Requirements”. Naval Research Laboratory, Washington, DC.

[21] A. Gargantini, C.Heitmeyer. “Using Model Checking to Generate Tests from Requirements Specifications”. Politecnico di Milano, Milano, Italy. Naval Research Laboratory, Washington, DC.

[22] M. Heimdahl, N. Leveson. “Completeness and Consistency in Hierarchical State-Based Requirements”. IEEE Transaction on Software Engineering, Vol 22, No. 6, June 1996