The Scientific Journal of Cihan University – Sulaimaniya **PP: 82-100**
Volume (6), Issue (1), June 2022
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)

A Proposed Formal Verification Model for Card-controlled Doors Using PROMELA with SPIN Model Checker

Bryar Ahmad Hassan[1], Shko Muhammed Qader[2], Hawkar Saeed Ezat[3], Hawkar Omar Ahmed [4], Hozan Khalid Hamarashid[5]

[1,3] Department of Information Technology, Kurdistan Institution for Strategic Studies and Scientific Research, Sulaimani, Iraq
[2,4] Department of Information Technology, University College of Goizha, Sulaimani, Iraq
[2,5] Information Technology Department, Computer Science Institute, Sulaimani Polytechnic University, Sulaymaniya, Iraq
[4] Department of Information Technology, College of Commerce, University of Sulaimani, Sulaimaniya, Iraq

Email: bryar.hassan@kissr.edu.krd[1], shko.qader@spu.edu.iq [2], hawkarsaeed@kissr.edu.krd[3] hawkar.omar@univsul.edu.iq[4], hozan.khalid@spu.edu.iq[5]

Abstract:

One of the problems with the security building system is the use of card-controlled doors. When users are classified, card-controlled models provide them different access rights depending on their classification, allowing them to use just certain doors and not always in both directions. A formal verification of this system is needed in order to determine its validity. This is the process of demonstrating or disproving the correctness of a model in terms of a particular formal specification or feature. On the basis of this assumption, this study used the Protocol or Process Meta Language (PROMELA) in combination with SPIN to verify the features of the card-controlled model. Following an introduction to the model's requirements, this article provides a basic explanation of the model's assumptions as well as the definition of global variables. Following that, the variables are set to their original values. It is then explained in detail, along with a SPIN simulation, how case study 1 of the model is carried out, which includes the building of an underground structure fitted with a card-operated access system. Finally, job 2 is defined as a description and validation of the model's accuracy features, with the latter being the primary focus. Since the suggested model fulfils the requirements and can be applied to a wide range of building topologies, it has been deemed to be an optimal, generic, and parametric model for the management of building access.

Keywords: Card-controlled doors, Formal verification, System verification, PROMELA, SPIN model checker.

الملخص:

إحدى مشاكل نظام الأمن للمباني هي إستخدام الأبواب التي يتم التحكم فيها بالبطاقة الذكية عندما يتم تصنيف المستخدمين فأن نماذج بطاقة التحكم توفر لهم طرق وصول مختلفة إعتمادا على تصنيفهم مما يسمح لهم بإستخدام أبواب معينة فقط و ليس دائما في كلا الإتجاهين حيث هناك حاجة إلى التحقق الرسمي من هذا النظام من أجل تحديد صلاحيته .على أساس هذا الإفتراض إستخدمت هذه الدراسة البروتوكول أو عملية لغة (Meta) بالأشتراك مع (SPIN)للتحقق من ميزات نموذج التحكم بالبطاقة . بعد مقدمة لمتطلبات النموذج تقدم هذه المقالة شرحا أساسيا لإفتراضات النموذج بالإضافة إلى تعريف المتغيرات العالمية , بعد ذلك يتم تحديد

DOI: http://dx.doi.org/10.25098/6.1.30

المتغيرات حسب قيمتها الأصلية .بإستخدام تقنية (SPIN) أخذنا نموذجين و الذي سوف نجرب عليه النظام المقترح ,الأول الطابق الأرضي للمبنى حيث من خلال هذا النظام سوف يتم التحكم بالأبواب و التجربة الثانية يعمل على خصائص النظام من حيث السرعة و تنفيذ الأنشطة الدقيقة للنظام .في النهاية نستطيع أن نقول أن هذا النظام يمكن إستخدامه في المباني و المواقع الذين يستعملون النظام الذكي للتحكم بالأبواب من خلال فحصه و مراقبته لأن النظام المقترح له القابلية على تنفيذ الأوامر المبرمجة المختلفة و القابلية على التأقلم مع متطلبات الموقع المستخدم.

الكلمات الدالة: أبواب يتم التحكم فيها بالبطاقة ، التحقق الرسمي ، التحقق من النظام ، بروميلا ، مدقق نموذج SPIN

پوخته:

یەکێک لە گرنگترین کێشەی باڵەخانەکان و ئەو شوێنانەی کە سیستەمی زیرەکی ئەلەکترۆنییان هەیە، بەکارهێنانی کارتی زیرەکە بۆ کۆنترۆڵکردنی دەرگاکانیان لە ڕووی چاودێریکردن و سێکورتییەوە. گرنگی ئەم کارەش لەوەدایە کە بە هۆی کۆنترۆڵکردنی بەکارهێنەرەکانیانەوە، دەتوانین بەکارهێنەران بکەین بە گروپ بۆ چەند گروپێکی جیاواز، بەشێوەیەك هەر گروپەو دەسەڵاتی جیاوازی خۆی هەبێت لە سیستەمە زیرەکە ئەلەکترۆنیەکاندا. یەکێکی دیکە لە تایبەتمەندییەکانی سیستمی زیرەکی ئەلیکترۆنی ئەوەیە کە ئەم سیستەمە توانای خۆ گونجاندنی هەیە بە پێی هەر پێشنیارو پێویستییەکانی بەکارهێنەرانی لە لایەن بەڕێوەبەری سیستەمەوە کە جێبەجێکردنەکەی بە سەر چەند پرۆسێسێکی جیاوازدا دەروات. لەم توێژینەوەدا تەکنەلۆجیاکانی (پرۆتۆکۆڵ) و (زمانی وەسفکردنی پرۆتۆکۆل) بەیەکەوە لەگەڵ (سپان) کە تایبەتمەندن بە بەکارهێنانی مۆدێلی کۆنترۆڵکردنی دەرگاکان بەکارهاتووە. هەروەها لە دوای پێناسەکردنی پێداویستییەکانی مۆدیلی سیستمەکە و ڕوونکردنەوەی تەواوی بەشەکانی و بەکارهێنانی پارامیتەری تایبەت بۆ چۆنیەتی ئیشپێکردنی سیستەمەکە بە تەواوی ڕوونکراوەتەوە .هەر لە درێژەی ڕوونکردنەوەکەماندا لەم توێژینەوەدا، لە گەڵ بەکارهێنانی تەکنەلۆجیای (سپان)، دوو نموونەمان وەرگرتووە کە سیستمی پێشنیارکراوی لەسەر تاقی ئەکەینەوە. ئەوانیش قاتی خوارەوەی باڵەخانەیەکە کە بەهۆی ئەم سیستەمەوە کۆنترۆلی دەرگاکانی ئەکرێت. وە تاقیکردنەوەی دووەم کار لە سەر تایبەتمەندییەکانی سیستەمەکە ئەکات لە ڕووی خێرایی و جێبەجێکردنی چالاکییە وردەکانی سیستەمەکە. لەکۆتاییدا ئەم سیستەمە ئەتوانرێت بەکار بهێنرێت لە باڵەخانەکان و ئەو شوێنانەی کە سیستمی زیرەک بە کارئەهێنن بۆ کۆنترۆڵکردنی دەرگاکانیان لە ڕووی چاودێریکردن و سێکوێرتییەوە، لەبەر ئەوەی سیستمی پێشنیارکراو توانای جێبەجێکردنی فەرمانە پرۆگرامینە جیاوازەکانی هەیە و ئەتوانێت خۆی بگونجێنێت لەگەڵ پێویستییەکانی شوێنی بەکار هێنراو.

کلیله وشەکان: کارتی کۆنترۆڵکردن دەرگاکان، سەلماندی سیستەم، پشتگیریکردنی سیستەم، پرۆمیلا، چێکردنەوەی مۆدێلەکان بەهۆی سپانەوە.

1. INTRODUCTION

This article includes a PROMELA model of a security building system, which is called card-controlled doors, and uses SPIN to verify some of its properties. The card-controlled model has different access rights are assigned to users in different categories, allowing them to use only some of the doors and not always in both directions[1]. This paper starts with presenting the requirements of the model with a general description to explain model's assumptions and define global variables. Next, an initialisation of the variables is shown. Then, a description of case study 1 of the model with its SPIN simulation is presented, which is about constructing a building fitted with a card operated access system. Finally, case study 2 as a description and verification of the model's correctness properties is indicated.

2. Problem Statement

Card-controlled doors model includes two case studies. The first case study is to construct a PROMELA model [2] to represent a building fitted with a card operated access system and simulate it using SPIN simulation. The second case study is to construct an appropriate description of four properties, and then use SPIN's verification facilities to establish whether the model satisfies each of them.

3. Model Description

This section is about a general and introductory description of the model that includes the model assumptions and defining the global variables.

1. This model can be applied on every topology after changing the global variables. For the sake of ease, this model and its report focus on the below building topology.

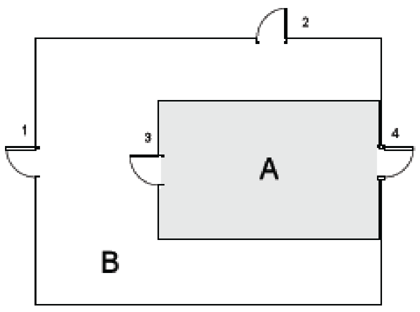


Figure 1: building topology sample

2. For the current building topology, User locations can be classified into three types that they are defined as global variables in this model as follows:

Table 1: Zones of model

Location	**Identifier**
Outside	0
ZoneA	1
ZoneB	2

In this model, these locations are defined by global variables named Outside, ZoneA, and ZoneB.

3. There are four doors for the current building topology. For this model, they are classified and identified follows:

Table 2: doors classification

Door number	**Identifier**
Door 1	0
Door 2	1
Door 3	2
Door 4	3

DOI: http://dx.doi.org/10.25098/6.1.30

The number of doors is identified by a defined global constant named NofDoors that holds values from 0 to 3. Likewise, CoomonDoor is initialised to 2 as a common door between ZoneA and ZoneB.

4. Based on the model requirements, there are four groups of users and these groups are classified and identified for this model as follows:

Table 3: classification of users

User type	**Identifier**
UsersAB	0
UsersA	1
UsersB	2
Others	3

5. Doors, their locations (rooms), and their direction (inside and outside) are connected via a *typedef* user-defined data structure called named DoorTopology as shown below. Because there are four doors, an array of size four is declared based on this *DoorTopology* called *doorTop*.

```
typedef DoorTopology {
short     doorLoc;
byte doorInside;
byte doorOutside
};
```

6. The four types of users and their authentication level to access the doors for both entering and exiting directions are defined by using *typedef* user-defined data type as global, which is named *AccessTopology*. By using this user-defined datatype, *access* as an array of size four is declared.

```
typedef AccessTopology {
byte      userLocation;
bit doorIn [4];
bit doorOut [4]
};
```

7. To communicate between the users and doors, a synchronous channel is defined to communicate between the users and doors, which is called *swipe* [3]. This channel has three values for user, door, and user location.

```
chan swipe = [0] of {byte  , byte, byte};
```

8. In this model, it is assumed that each user can enter or exit through a door in a different time.

9. It is assumed that when a user presents their card to a reader by a door, if the door opens, the user passes through and the door is locked behind them; people do not swipe their cards and walk away, not do they open for others, and "tailgating" does not happen.

DOI: http://dx.doi.org/10.25098/6.1.30

4. The proposed Model Installation

All of the global variables are initialised in the initialisation process as the initial system state [4]. To run the processes for all the users, the user () process is run four times for each group of the users and process door () runs one.

125 atomic { run user(0); run user(1); run user(2); run user(3); run door(); }

5. Case Study 1

In this case study, a PROMELA model is presented to construct a building fitted with a card operated access system. In order for the model to be more generic and parametric in the building topology and the user right access[5], all information, such as the connection between rooms and doors and the rights to access specific doors[6] spending on the user type is defined as global variables[7].

5.1 Model Description

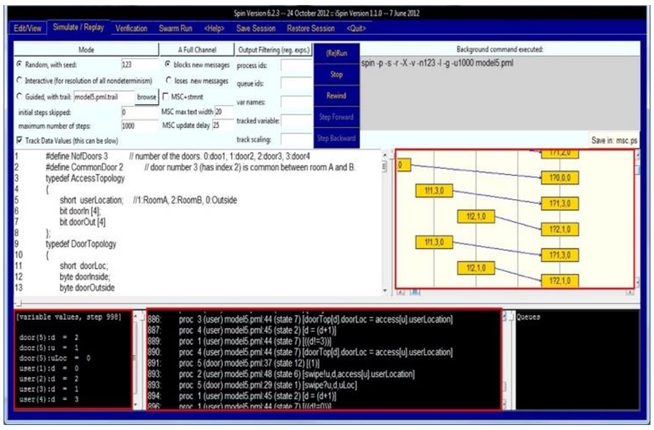
The case study 1 of this model is done by defining two processes, which are *user (byte u)* and *door ()*. Both processes can communicate on each other via a synchronous channel[8]. The former process presents the way of swiping the users' card into the doors via the *swipe* channel. The parameter *byte u* in *user (byte u)* process represents user. Doors are generated by random from 0 to 3. Users can swipe their card into the doors that are in their own zone. For instance, usersAB can swipe their card into door 1, 2, or 4 if their location is Outside. Similarly, users in ZoneA and ZoneB can swipe their cards into the surrounded doors and the common door between ZoneA and ZoneB can be swiped by users in either location. Thus, they can swipe their cards to the authorised doors[9], [10], and then enter from a location into another. The latter process describes the receiving requests from the users to access their permitted doors. After receiving a request from the users to enter or exit from the doors, the *door ()* process checks for authentication for both directions of the doors whether the user can access the door or not. *access[u].doorIn[d]* is used to check the inside direction of the doors' authentication and *access[u].doorOut[d]* is used to check the outside direction of the doors' authentication. In coincident with doors' authentication, the process checks each user's current location to be able to change and represent user's location after entering or exiting a door.

5.2 Simulation Output

In this section, a SPIN simulation[11] of this model presents to gain early feedback on it.

1. Before simulating the model, the display mode parameters can be set up. Message Sequence Chart (MSC) Panel, Time Sequence Panel, Data Values Panel are presented by default on main windows SPIN simulator. Figure 2 depicts this display mode.

DOI: http://dx.doi.org/10.25098/6.1.30



Mode
- (•) Random, with seed: 123
- ( ) Interactive (for resolution of all nondeterminism)
- ( ) Guided, with trail: model5.pml.trail browse
- initial steps skipped: 0
- maximum number of steps: 1000
- [✓] Track Data Values (this can be slow)

A Full Channel
- (•) blocks new messages
- ( ) loses new messages
- [ ] MSC+stmnt
- MSC max text width 20
- MSC update delay 25

Output Filtering (reg. exps.)
- process ids:
- queue ids:
- var names:
- tracked variable:
- track scaling:

Background command executed:

```
spin -p -s -r -X -v -n123 -l -g -u1000 model5.pml
```

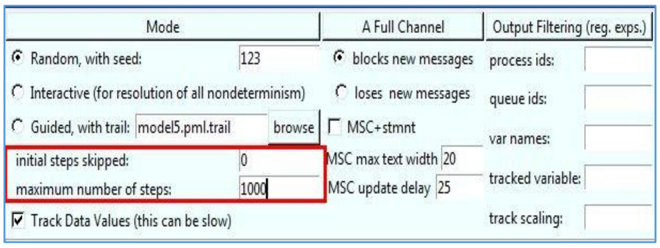
```
#define NofDoors 3        // number of the doors. 0:doo1, 1:door2, 2:door3, 3:door4
#define CommonDoor 2        // door number 3 (has index 2) is common between room A and B.
typedef AccessTopology
{
    short userLocation;    //1:RoomA, 2:RoomB, 0:Outside
    bit doorIn [4];
    bit doorOut [4]
};
typedef DoorTopology
{
    short doorLoc;
    byte doorInside;
    byte doorOutside
```

```
[variable values, step 998]

 door(5):d  =  2
 door(5):u  =  1
 door(5):uLoc  =  0
 user(1):d  =  0
 user(2):d  =  2
 user(3):d  =  1
 user(4):d  =  3
```

```
886:    proc  3 (user) model5.pml:44 (state 7) [doorTop[d].doorLoc = access[u].userLocation]
887:    proc  4 (user) model5.pml:45 (state 2) [d = (d+1)]
889:    proc  1 (user) model5.pml:44 (state 7) [((d!=3))]
890:    proc  4 (user) model5.pml:44 (state 7) [doorTop[d].doorLoc = access[u].userLocation]
891:    proc  5 (door) model5.pml:37 (state 12) [(1)]
893:    proc  2 (user) model5.pml:48 (state 6) [swipe!u,d,access[u].userLocation]
893:    proc  5 (door) model5.pml:29 (state 1) [swipe?u,d,uLoc]
894:    proc  1 (user) model5.pml:45 (state 2) [d = (d+1)]
896:    proc  1 (user) model5.pml:44 (state 7) [((d!=0))]
```

Figure 2: SPIN’s display mode parameter

Style parameters can be set up to Random (default)[12], Guided, and Interactive. Moreover, on setting the parameters, the number of initial steps to skip as well as the maximum number of steps can be specified in SPIN[13]. As it is shown in Figure 3.



Mode
- (•) Random, with seed: 123
- ( ) Interactive (for resolution of all nondeterminism)
- ( ) Guided, with trail: model5.pml.trail
- initial steps skipped: 0
- maximum number of steps: 1000
- [✓] Track Data Values (this can be slow)

A Full Channel
- (•) blocks new messages
- ( ) loses new messages
- [ ] MSC+stmnt
- MSC max text width 20
- MSC update delay 25

Output Filtering (reg. exps.)
- process ids:
- queue ids:
- var names:
- tracked variable:
- track scaling:

Figure 3: setting Initial Steps Skipped and Maximum Number of Steps

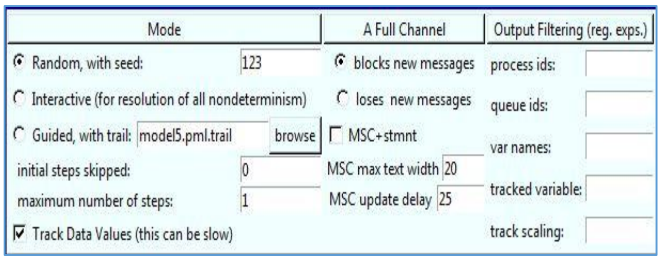
The Initial Steps Skipped is set to zero and the Maximum Number of Steps is set to 1000 in this simulation. Table 4 illustrated the most important parameters in the SPIN simulation mode with their default values.

Table 4: SPIN simulation mode – setting parameters

Parameter	**Value**
Mode	Random, with seed (default); interactive; guided, with trail
A full channel	Blocks new messages (default); loses new messages
Initial steps skipped	Default value is 0
Maximum number of steps	Default value is 10000
MSC max text width	Default value is 20
MSC update delay	Default value is 25

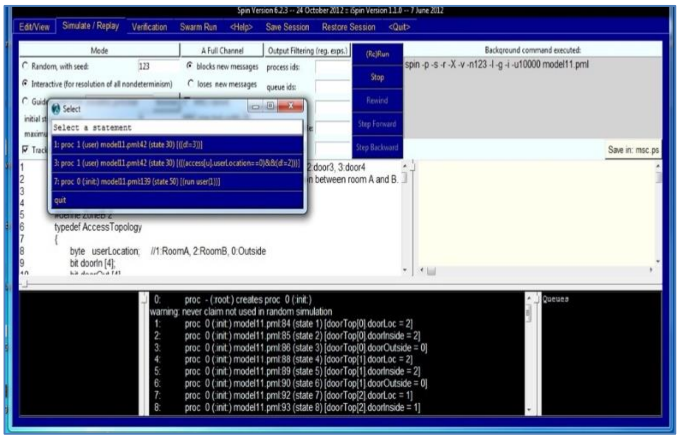
Figure 4 presents the SPIN’s setting parameters in simulation mode and also Figure 5 gives a screenshot of interactive parameter mode.

DOI: http://dx.doi.org/10.25098/6.1.30



[✓ Random, with seed:] 123
Interactive (for resolution of all nondeterminism)
Guided, with trail: model5.pml.trail
initial steps skipped: 0
maximum number of steps: 1
[✓ Track Data Values (this can be slow)]
[✓ blocks new messages]
loses new messages
MSC+stmnt
MSC max text width 20
MSC update delay 25
process ids:
queue ids:
var names:
tracked variable:
track scaling:

Figure 4: SPIN's display mode parameter



Random, with seed: 123
[✓ Interactive (for resolution of all nondeterminism)]
[✓ blocks new messages]
loses new messages
process ids:
queue ids:

```
spin -p -s -r -X -v -n123 -l -g -i -u10000 model11.pml
```

```
Select a statement
1: proc 1 (user) model11.pml:42 (state 30) [((d!=3))]
3: proc 1 (user) model11.pml:42 (state 30) [(((access[u].userLocation==0)&&(d!=2)))]
7: proc 0 (:init:) model11.pml:139 (state 50) [(run user(1))]
quit
```

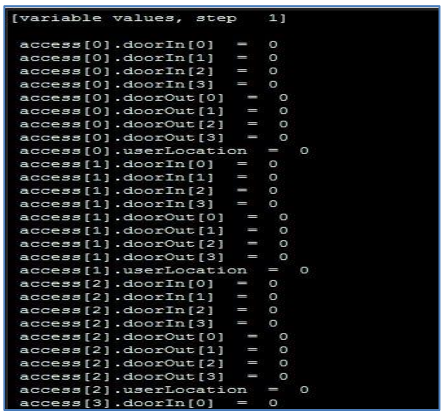
```
        #define ZoneB 2
        typedef AccessTopology
        {
            byte   userLocation;     //1:RoomA, 2:RoomB, 0:Outside
            bit doorIn [4];
```

```
2:door3, 3:door4
n between room A and B.
```

```
  0:        proc  - (:root:) creates proc  0 (:init:)
warning: never claim not used in random simulation
  1:        proc  0 (:init:) model11.pml:84 (state 1) [doorTop[0].doorLoc = 2]
  2:        proc  0 (:init:) model11.pml:85 (state 2) [doorTop[0].doorInside = 2]
  3:        proc  0 (:init:) model11.pml:86 (state 3) [doorTop[0].doorOutside = 0]
  4:        proc  0 (:init:) model11.pml:88 (state 4) [doorTop[1].doorLoc = 2]
  5:        proc  0 (:init:) model11.pml:89 (state 5) [doorTop[1].doorInside = 2]
  6:        proc  0 (:init:) model11.pml:90 (state 6) [doorTop[1].doorOutside = 0]
  7:        proc  0 (:init:) model11.pml:92 (state 7) [doorTop[2].doorLoc = 1]
  8:        proc  0 (:init:) model11.pml:93 (state 8) [doorTop[2].doorInside = 1]
```

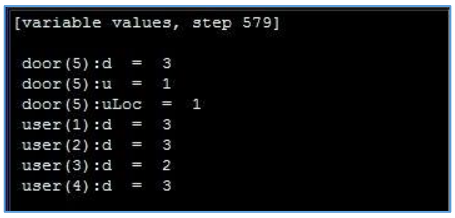
Figure 5: SPIN's interactive mode for resolution of all non-determinism

2. Data Values Panel presents data values across time that includes buffered channels, and global and local variables. The values of all variables as shown in Figure 6 are initialised on the Data Values Panel in the first step of simulation. This Figure is the first screenshot of this panel that shown the values of buffered channel, global and local variables in the initial step. Similarly, Figure 7 is an example of capturing variable values of step 579 in the execution steps.



```
[variable values, step    1]

 access[0].doorIn[0]   =   0
 access[0].doorIn[1]   =   0
 access[0].doorIn[2]   =   0
 access[0].doorIn[3]   =   0
 access[0].doorOut[0]   =   0
 access[0].doorOut[1]   =   0
 access[0].doorOut[2]   =   0
 access[0].doorOut[3]   =   0
 access[0].userLocation   =   0
 access[1].doorIn[0]   =   0
 access[1].doorIn[1]   =   0
 access[1].doorIn[2]   =   0
 access[1].doorIn[3]   =   0
 access[1].doorOut[0]   =   0
 access[1].doorOut[1]   =   0
 access[1].doorOut[2]   =   0
 access[1].doorOut[3]   =   0
 access[1].userLocation   =   0
 access[2].doorIn[0]   =   0
 access[2].doorIn[1]   =   0
 access[2].doorIn[2]   =   0
 access[2].doorIn[3]   =   0
 access[2].doorOut[0]   =   0
 access[2].doorOut[1]   =   0
 access[2].doorOut[2]   =   0
 access[2].doorOut[3]   =   0
 access[2].userLocation   =   0
 access[3].doorIn[0]   =   0
```

Figure 6: initial step of data value panel

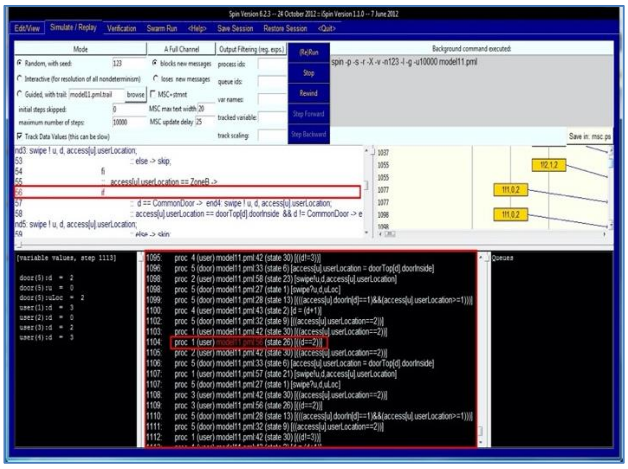


```
[variable values, step 579]

 door(5):d   =   3
 door(5):u   =   1
 door(5):uLoc   =   1
 user(1):d   =   3
 user(2):d   =   3
 user(3):d   =   2
 user(4):d   =   3
```

Figure 7: values of variable for a step

DOI: http://dx.doi.org/10.25098/6.1.30

3. By looking at the Time Sequence Panel that provides a graphical presentation of the process's execution over time, it can be seen that multiple perspectives are supported, such as execution steps interleaved, one window per process, and one execution trace per process. Figure 8 shows a screenshot of this panel. By pointing out a process in this panel, the execution line can be seen by red font.



Mode: (●) Random, with seed: 123 ( ) Interactive (for resolution of all nondeterminism) ( ) Guided, with trail: model11.pml.trail browse
initial steps skipped: 0
maximum number of steps: 10000
☑ Track Data Values (this can be slow)

A Full Channel: (●) blocks new messages ( ) loses new messages
☐ MSC+stmnt
MSC max text width 20
MSC update delay 25

Output Filtering (reg. exps.): process ids: queue ids: var names: tracked variable: track scaling:

Background command executed:

```
spin -p -s -r -X -v -n123 -l -g -u10000 model11.pml
```

```
nd3: swipe ! u, d, access[u].userLocation;
53                                 :: else -> skip;
54                        fi
55                        ::   access[u].userLocation == ZoneB ->
56                        if
57                                 ::  d == CommonDoor ->  end4: swipe ! u, d, access[u].userLocation;
58                                 :: access[u].userLocation == doorTop[d].doorInside  && d != CommonDoor -> e
nd5: swipe ! u, d, access[u].userLocation;
59                                 :: else -> skip;
```

```
[variable values, step 1113]

 door(5):d  =  2
 door(5):u  =  0
 door(5):uLoc  =  2
 user(1):d  =  3
 user(2):d  =  0
 user(3):d  =  2
 user(4):d  =  3
```

```
1095:   proc  4 (user) model11.pml:42 (state 30) [((d!=3))]
1096:   proc  5 (door) model11.pml:33 (state 6) [access[u].userLocation = doorTop[d].doorInside]
1098:   proc  2 (user) model11.pml:58 (state 23) [swipe!u,d,access[u].userLocation]
1098:   proc  5 (door) model11.pml:27 (state 1) [swipe?u,d,uLoc]
1099:   proc  5 (door) model11.pml:28 (state 13) [(((access[u].doorIn[d]==1)&&(access[u].userLocation>=1)))]
1100:   proc  4 (user) model11.pml:43 (state 2) [d = (d+1)]
1102:   proc  5 (door) model11.pml:32 (state 9) [((access[u].userLocation==2))]
1103:   proc  1 (user) model11.pml:42 (state 30) [((access[u].userLocation==2))]
1104:   proc  1 (user) model11.pml:56 (state 26) [((d==2))]
1105:   proc  2 (user) model11.pml:42 (state 30) [((access[u].userLocation==2))]
1106:   proc  5 (door) model11.pml:33 (state 6) [access[u].userLocation = doorTop[d].doorInside]
1107:   proc  1 (user) model11.pml:57 (state 21) [swipe!u,d,access[u].userLocation]
1107:   proc  5 (door) model11.pml:27 (state 1) [swipe?u,d,uLoc]
1108:   proc  3 (user) model11.pml:42 (state 30) [((access[u].userLocation==2))]
1109:   proc  3 (user) model11.pml:56 (state 26) [((d==2))]
1110:   proc  5 (door) model11.pml:28 (state 13) [(((access[u].doorIn[d]==1)&&(access[u].userLocation>=1)))]
1111:   proc  5 (door) model11.pml:32 (state 9) [((access[u].userLocation==2))]
1112:   proc  1 (user) model11.pml:42 (state 30) [((d!=3))]
```

Figure 8: Time Sequence Panel

4. By looking at Message Sequence Chart (MSC), each of the four users swipes its card to their accessed doors randomly through the channel. Users send requests and the doors receive them. If a group of users is authenticated to pass through a door, the door will open directly in order to the user pass through it. In addition, each group of users has its own communication line to the doors. For example, Figure 9 depicts the movement of usersAB. The red lines show the movement of usersAB through the rooms, and paying particular attention to how the location of this user is updated and represented in the chart.

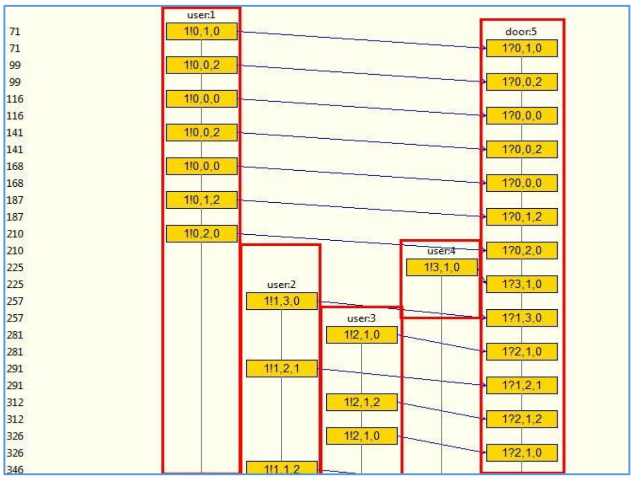


Figure 9: Message Sequence Chart

Similarly, the movement of the other users can be traced as it is shown on Table 5 based upon the Message Sequence Chart.

Table 5: MSC simulation tracing

UsersA (user:2)	Doors (door:5)	Location	Comment
1!1,3,0	1?1,3,0	Outside	Swiping and entering to ZoneA via door 4
1!1,3,1	1?1,3,1	ZoneA	Swiping via door 4, but has not authorised to exit
1!1,3,1	1?1,3,1	ZoneA	Swiping via door 4, but has not authorised to exit
1!1,2,1	1?1,2,1	ZoneA	Swiping and exiting from ZoneA via door 3
1!1,2,2	1?1,2,2	ZoneB	Swiping and exiting ZoneA via door 3

In regard with the communication between the users and doors, Figure 10 presents the way of swiping the users' cards to their authorised doors. This communication is done by the *swipe* channel between the doors and users. Via clicking each of the communication channels between the users and doors on MSC, the fragment of code(s) in relation with this communication can be spotted by red line on the model. In relation with the MSC and the executed code, Figure 11 presents the executed line of code when each communication between users and doors is spotted.

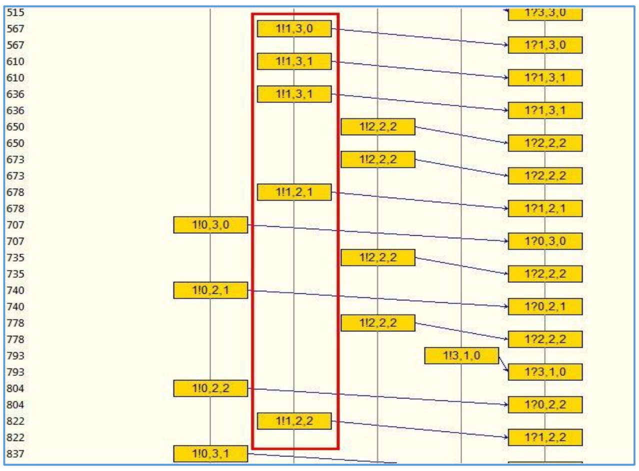


Figure 10: tracing movement of usersAB

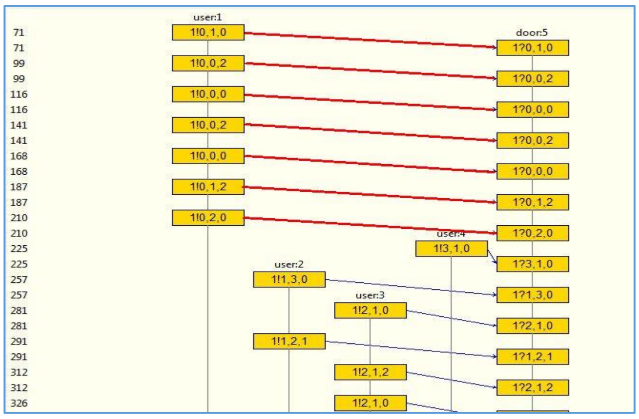
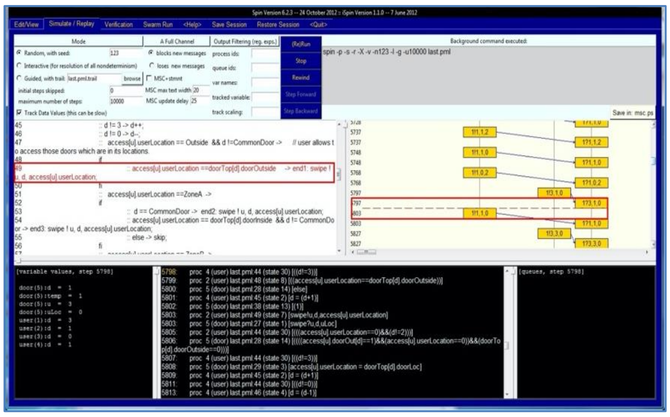


Figure 11: channel communication between users and doors (send and receive values)

DOI: http://dx.doi.org/10.25098/6.1.30



```
Mode
(•) Random, with seed: 123
( ) Interactive (for resolution of all nondeterminism)
( ) Guided, with trail: last.pml.trail   browse
initial steps skipped: 0
maximum number of steps: 10000
[✓] Track Data Values (this can be slow)

A Full Channel
(•) blocks new messages
( ) loses new messages
[ ] MSC+stmnt
MSC max text width 20
MSC update delay 25

Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:

Background command executed:
spin -p -s -r -X -v -n123 -l -g -u10000 last.pml
```

```
                        :: d != 3 -> d++;
                        :: d != 0 -> d--;
                        ::  access[u].userLocation == Outside  && d !=CommonDoor ->     // user allows t
o access those doors which are in its locations.
                        if
                                :: access[u].userLocation ==doorTop[d].doorOutside    -> end1: swipe !
u, d, access[u].userLocation;
                        fi
                        ::  access[u].userLocation ==ZoneA  ->
                        if
                                :: d == CommonDoor ->  end2: swipe ! u, d, access[u].userLocation;
                                :: access[u].userLocation == doorTop[d].doorInside  && d != CommonDo
or -> end3: swipe ! u, d, access[u].userLocation;
                                :: else -> skip;
                        fi
```

```
5728
5737   1!1,1,2        1?1,1,2
5737
5748   1!1,1,0        1?1,1,0
5748
5768   1!1,0,2        1?1,0,2
5768
5797          1!3,1,0
5797                  1?3,1,0
5803   1!1,1,0
5803                  1?1,1,0
5827          1!3,3,0
5827                  1?3,3,0
```

```
[variable values, step 5798]

 door(5):d   =  1
 door(5):temp  =  1
 door(5):u   =  3
 door(5):uLoc  =  0
 user(1):d   =  3
 user(2):d   =  1
 user(3):d   =  0
 user(4):d   =  1
```

```
5798:   proc  4 (user) last.pml:44 (state 30) [((d!=3))]
5799:   proc  2 (user) last.pml:48 (state 8) [((access[u].userLocation==doorTop[d].doorOutside))]
5800:   proc  5 (door) last.pml:28 (state 14) [else]
5801:   proc  4 (user) last.pml:45 (state 2) [d = (d+1)]
5802:   proc  5 (door) last.pml:38 (state 13) [(1)]
5803:   proc  2 (user) last.pml:49 (state 7) [swipe!u,d,access[u].userLocation]
5803:   proc  5 (door) last.pml:27 (state 1) [swipe?u,d,uLoc]
5805:   proc  2 (user) last.pml:44 (state 30) [((((access[u].userLocation==0)&&(d!=2)))]
5806:   proc  5 (door) last.pml:28 (state 14) [(((((access[u].doorOut[d]==1)&&(access[u].userLocation==0))&&(doorTo
p[d].doorOutside==0))))]
5807:   proc  4 (user) last.pml:44 (state 30) [((d!=3))]
5808:   proc  5 (door) last.pml:29 (state 3) [access[u].userLocation = doorTop[d].doorLoc]
5809:   proc  4 (user) last.pml:45 (state 2) [d = (d+1)]
5811:   proc  4 (user) last.pml:44 (state 30) [((d!=0))]
5813:   proc  4 (user) last.pml:46 (state 4) [d = (d-1)]
```

```
[queues, step 5798]
```

Figure 12: spotting communication channel with executed code

6. Case Study 2

In case study 2, a description and verification of the model's correctness properties are identified.

6.1 Verification Description

In this section, an appropriate description of each of the following properties is constructed.

1. To verify the model whether it does or does not deadlock[14], end-state can be used to judge whether a process is in a deadlock state or an accepTable waiting state in a non-terminating model. End labels as one of the types of meta-labels can be used to verify for absence of reachable deadlock states in this model. End labels have names that begin with end. The only valid end states are where every PROMELA process has reached the end of its code, whereas any invalid end state is a deadlock state. By using a valid end-state, every instantiated process has either terminated or is blocked at a statement that is labelled as an end-state. The below fragment code is about using the end-state in the model.

```
::        access[u].userLocation == Outside  && d !=CommonDoor ->
        if
        :: access[u].userLocation ==doorTop[d].doorOutside    ->
       end1: swipe ! u, d, access[u].userLocation;
       fi
```

2. To verify a user classed as "Others" is never able to open any door, never claim can be deployed because it is typically used to specify a behaviour that never happens. To do so, a never-claim is defined in this model to check the location of "Others" user as it should never be changed as indication that unable to open any door and enter. The below fragment code is the never claim process to verify this property.

```
       never{
      do
      :: (access[3].userLocation != 0) -> break
     :: else -> skip
    od
    }
```

Never claim checks for invariance of this property *(access[3].userLocation != 0)*. This property is a boolean expression. While the boolean expression remains true, the claim process remains in its initial state. As soon as the property is found to be false, the never-claim process ends and indicates an error. Furthermore, the *else->skip* statement can be replaced with *else*, but this replacement might miss some violations of the invariant.

3. To verify UsersA are able to enter zoneA through door 3 and able to leave the same way via door 4, LTL formula for verification can be used to specify that this property must hold at certain points in execution. To do so, a formula is defined in this model, which is labelled as *p1*. This claim evaluates the condition as a side-effect free, if it will be false, an error is reported. This claim can be run by setting its name. If the claim is false, execution will be stopped with an error message.

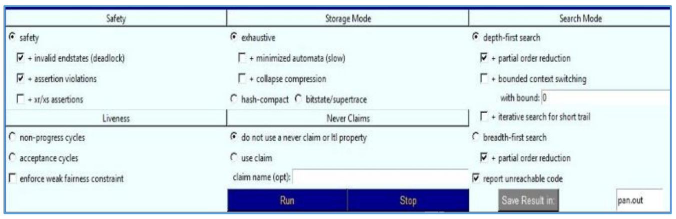
22 ltl p1 { (access[1].userLocation == ZoneB -> access[1].userLocation == ZoneA) }

4. To verify if there is any circumstance where one of the several authorised users may be unable to pass through a door, LTL formulas can be used. Users can pass those doors who are allowed to do so, whereas they cannot pass through any unauthorised doors. For example, users labelled as "Others" cannot access any door and users labelled as "UsersA" can pass through all of the doors. However, there might be a circumstance that a user may not be enabled infinity often to pass through a door and it may be scheduled to be enabled eventually. This situation is called weak fairness. SPIN only supports weak fairness assumptions, but strong fairness can be enforced by using LTL formulas. For example, usersA can be enforced to enter from ZoneB to RoomA by using the below LTL formula:

23 ltl p2 { (access[2].userLocation == ZoneB -> access[2].userLocation == Outside) }

6.2 Verification Output

In this section, the usage of SPIN's verification facilitates to establish whether the model satisfies each of following correctness properties. To start with verifying the four correctness properties in SPIN, the following SPIN verification parameters in Table 6 should be set up based on the needs of case study 2 to represent. Figure 14 and Figure 13 depict the SPIN's basic verification parameters and advanced verification parameters.



Safety
- [✓] safety
- [✓] + invalid endstates (deadlock)
- [✓] + assertion violations
- [ ] + xr/xs assertions

Liveness
- [ ] non-progress cycles
- [ ] acceptance cycles
- [ ] enforce weak fairness constraint

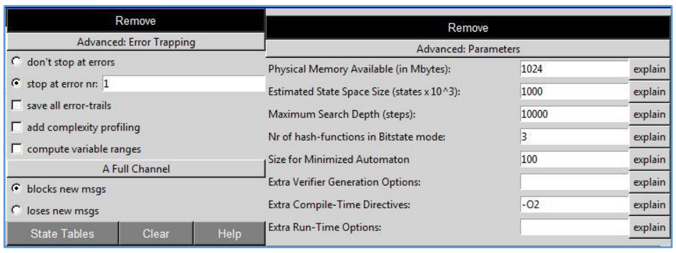
Storage Mode
- [✓] exhaustive
- [ ] + minimized automata (slow)
- [ ] + collapse compression
- [ ] hash-compact [ ] bitstate/supertrace

Never Claims
- [✓] do not use a never claim or ltl property
- [ ] use claim
- claim name (opt):

Search Mode
- [✓] depth-first search
- [✓] + partial order reduction
- [ ] + bounded context switching
- with bound: 0
- [ ] + iterative search for short trail
- [ ] breadth-first search
- [✓] + partial order reduction
- [✓] report unreachable code

Figure 13: SPIN verification parameters

DOI: http://dx.doi.org/10.25098/6.1.30



Remove
Advanced: Error Trapping
( ) don't stop at errors
(●) stop at error nr: 1
☐ save all error-trails
☐ add complexity profiling
☐ compute variable ranges
A Full Channel
(●) blocks new msgs
( ) loses new msgs
State Tables | Clear | Help

Remove
Advanced: Parameters

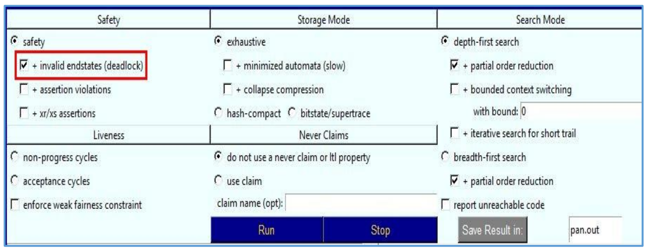
Physical Memory Available (in Mbytes):	1024	explain
Estimated State Space Size (states x 10^3):	1000	explain
Maximum Search Depth (steps):	10000	explain
Nr of hash-functions in Bitstate mode:	3	explain
Size for Minimized Automaton	100	explain
Extra Verifier Generation Options:		explain
Extra Compile-Time Directives:	-O2	explain
Extra Run-Time Options:		explain

Figure 14: advanced SPIN verification parameters

Table 6: SPIN verification parameters

Parameter	**Value**
Safety	Deadlocks; assertion; violations
Liveness	Non-progress cycles; acceptance cycles; enforce weak fairness
Never claims	Do not use never claim or ltl propery; use claim or claim name (opt)
Storage mode	Exhaustive; hash-compact; bitstate search
Search mode	Depth-first - partial order reduction; iterative search (shortest trail); breadth first; report unreachable code

1. The model does not deadlock due to the use of meta-label in the model. Figure 13 and Figure 15 present setting the deadlock verification parameter and a deadlock result of the model. Based on the verification result on the latter Figure, the model does not deadlock.



Safety
(●) safety
☑ + invalid endstates (deadlock)
☐ + assertion violations
☐ + xr/xs assertions

Storage Mode
(●) exhaustive
☐ + minimized automata (slow)
☐ + collapse compression
( ) hash-compact ( ) bitstate/supertrace

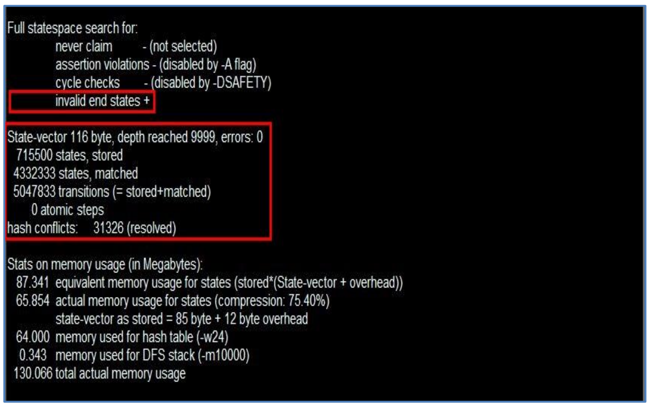
Search Mode
(●) depth-first search
☑ + partial order reduction
☐ + bounded context switching
with bound: 0
☐ + iterative search for short trail
( ) breadth-first search
☑ + partial order reduction
☐ report unreachable code

Liveness
( ) non-progress cycles
( ) acceptance cycles
☐ enforce weak fairness constraint

Never Claims
(●) do not use a never claim or ltl property
( ) use claim
claim name (opt):

Run | Stop | Save Result in: pan.out

Figure 15: deadlock verification parameter

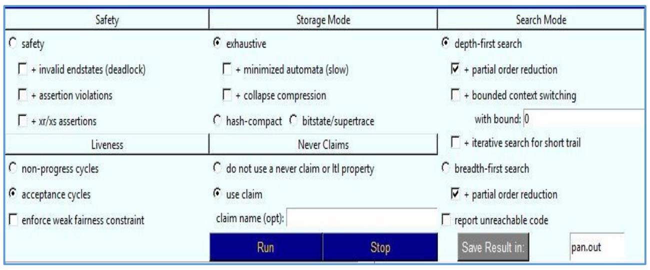


```
Full statespace search for:
        never claim         - (not selected)
        assertion violations - (disabled by -A flag)
        cycle checks        - (disabled by -DSAFETY)
        invalid end states +

State-vector 116 byte, depth reached 9999, errors: 0
  715500 states, stored
 4332333 states, matched
 5047833 transitions (= stored+matched)
       0 atomic steps
hash conflicts:    31326 (resolved)

Stats on memory usage (in Megabytes):
  87.341  equivalent memory usage for states (stored*(State-vector + overhead))
  65.854  actual memory usage for states (compression: 75.40%)
          state-vector as stored = 85 byte + 12 byte overhead
  64.000  memory used for hash table (-w24)
   0.343  memory used for DFS stack (-m10000)
 130.066 total actual memory usage
```

Figure 16: deadlock verification result

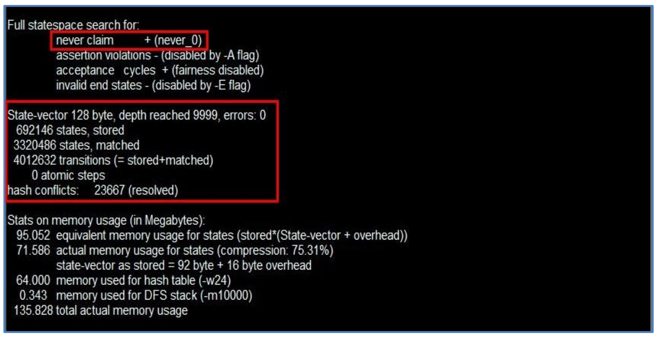
2. The verification result of the never claim has shown that if a user classed as “Others”, it never be able to enter any door. Figure 17 and Figure 18 depict the details of setting never claim parameter and its verification result.

Volume (6), Issue (1), June 2022
ISSN 2520-7377 (Online), ISSN 2520-5102 (Print)



Safety	Storage Mode	Search Mode
( ) safety	(•) exhaustive	(•) depth-first search
[ ] + invalid endstates (deadlock)	[ ] + minimized automata (slow)	[✓] + partial order reduction
[ ] + assertion violations	[ ] + collapse compression	[ ] + bounded context switching
[ ] + xr/xs assertions	( ) hash-compact ( ) bitstate/supertrace	with bound: 0
Liveness	Never Claims	[ ] + iterative search for short trail
( ) non-progress cycles	( ) do not use a never claim or ltl property	( ) breadth-first search
(•) acceptance cycles	(•) use claim	[✓] + partial order reduction
[ ] enforce weak fairness constraint	claim name (opt):	[ ] report unreachable code
	Run Stop	Save Result in: pan.out

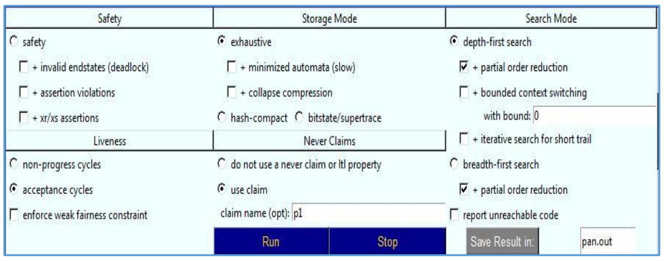
Figure 17: never claim parameter



```
Full statespace search for:
        never claim         + (never_0)
        assertion violations - (disabled by -A flag)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by -E flag)

State-vector 128 byte, depth reached 9999, errors: 0
   692146 states, stored
  3320486 states, matched
  4012632 transitions (= stored+matched)
        0 atomic steps
hash conflicts:     23667 (resolved)

Stats on memory usage (in Megabytes):
   95.052  equivalent memory usage for states (stored*(State-vector + overhead))
   71.586  actual memory usage for states (compression: 75.31%)
           state-vector as stored = 92 byte + 16 byte overhead
   64.000  memory used for hash table (-w24)
    0.343  memory used for DFS stack (-m10000)
  135.828 total actual memory usage
```

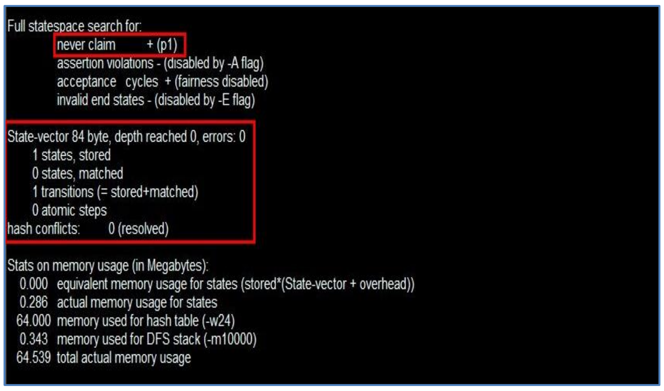
Figure 18: never claim verification result

3. To verify the third property, the claim has given the result that UsersA are able to enter ZoneA through door 3 and able to leave the same way or via door 4. Both Figure 19 and Figure 20 present the *p1* claim parameter setting and its verification result.



Safety	Storage Mode	Search Mode
( ) safety	(•) exhaustive	(•) depth-first search
[ ] + invalid endstates (deadlock)	[ ] + minimized automata (slow)	[✓] + partial order reduction
[ ] + assertion violations	[ ] + collapse compression	[ ] + bounded context switching
[ ] + xr/xs assertions	( ) hash-compact ( ) bitstate/supertrace	with bound: 0
Liveness	Never Claims	[ ] + iterative search for short trail
( ) non-progress cycles	( ) do not use a never claim or ltl property	( ) breadth-first search
(•) acceptance cycles	(•) use claim	[✓] + partial order reduction
[ ] enforce weak fairness constraint	claim name (opt): p1	[ ] report unreachable code
	Run Stop	Save Result in: pan.out

Figure 19: assertion verification parameter

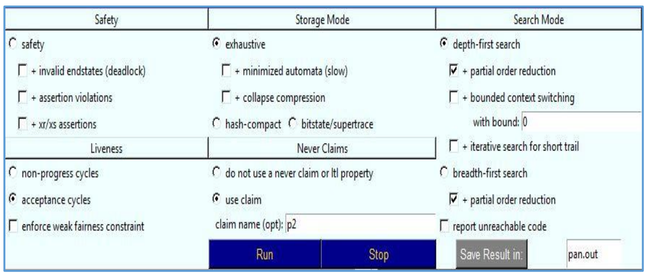


```
Full statespace search for:
        never claim         + (p1)
        assertion violations - (disabled by -A flag)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by -E flag)

State-vector 84 byte, depth reached 0, errors: 0
        1 states, stored
        0 states, matched
        1 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.000  equivalent memory usage for states (stored*(State-vector + overhead))
    0.286  actual memory usage for states
   64.000  memory used for hash table (-w24)
    0.343  memory used for DFS stack (-m10000)
   64.539  total actual memory usage
```

Figure 20: assertion verification result

4. It has verified an instance of weak verification that can be enforced by using LTL. Figure 21 and Figure 22 give the screenshots of setting the LTL parameter and their verification result.

DOI: http://dx.doi.org/10.25098/6.1.30



Safety
- ( ) safety
- [ ] + invalid endstates (deadlock)
- [ ] + assertion violations
- [ ] + xr/xs assertions

Storage Mode
- (•) exhaustive
- [ ] + minimized automata (slow)
- [ ] + collapse compression
- ( ) hash-compact ( ) bitstate/supertrace

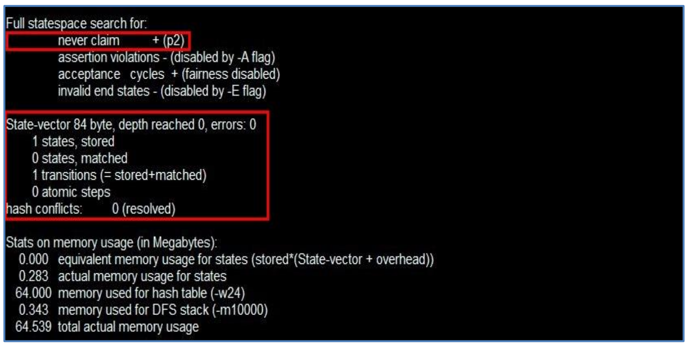
Search Mode
- (•) depth-first search
- [✓] + partial order reduction
- [ ] + bounded context switching
- with bound: 0
- [ ] + iterative search for short trail
- ( ) breadth-first search
- [✓] + partial order reduction
- [ ] report unreachable code

Liveness
- ( ) non-progress cycles
- (•) acceptance cycles
- [ ] enforce weak fairness constraint

Never Claims
- ( ) do not use a never claim or ltl property
- (•) use claim
- claim name (opt): p2

Save Result in: pan.out

Figure 21: LTL verification parameter



```
Full statespace search for:
        never claim         + (p2)
        assertion violations - (disabled by -A flag)
        acceptance  cycles  + (fairness disabled)
        invalid end states - (disabled by -E flag)

State-vector 84 byte, depth reached 0, errors: 0
    1 states, stored
    0 states, matched
    1 transitions (= stored+matched)
    0 atomic steps
hash conflicts:        0 (resolved)

Stats on memory usage (in Megabytes):
   0.000   equivalent memory usage for states (stored*(State-vector + overhead))
   0.283   actual memory usage for states
  64.000   memory used for hash table (-w24)
   0.343   memory used for DFS stack (-m10000)
  64.539   total actual memory usage
```

Figure 22: LTL verification result

7. Conclusion

In this article, a model of the card-controlled doors has presented by using PROMELA to build the model and SPIN to verify some of its properties. Firstly, the requirements of the model with an outline description have presented. Secondly, an initialisation of the variables has been shown. Thirdly, a description of case study 1 of the model with its SPIN simulation has presented. Finally, case study 2 as a description and verification of the model's correctness properties has indicated. Therefore, the suggested model is a relatively optimal, generic, and parametric model for building access control because it meets the requirements and it could be applied for different building topologies.

8. References

[1] R. Nardone *et al.*, "Modeling railway control systems in Promela," in *International Workshop on Formal Techniques for Safety-Critical Systems*, 2015, pp. 121–136.

[2] S. Löffler and A. Serhrouchni, "Creating implementations from PROMELA models," 1996.

[3] D. J. C. A. P. Monteiro, "Coverage-based validation of embedded systems," 2015.

[4] H. E. H. Santoso, H. Saputra, A. Shofyan, K. Anam, and E. Supraptono, "The Use of RFID Sensors for Automatic Doorstop Application".

[5] A. Yacoub, M. E.-A. Hamri, and C. Frydman, "Using DEv-PROMELA for modelling and verification of software," in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 2016, pp. 245–253.

[6] Y. Choi, "Automated validation of IoT device control programs through domain-specific model generation," in *International Conference on Software Engineering and Formal Methods*, 2018, pp. 254–268.

[7] V. S. Burenkov and A. S. Kamkin, “Checking parameterized Promela models of cache coherence protocols,” *Труды Института системного программирования РАН*, vol. 28, no. 4, 2016.

[8] N. Dilley and J. Lange, “Bounded verification of message-passing concurrency in Go using Promela and Spin,” *arXiv preprint arXiv:2004.01323*, 2020.

[9] M. I. Abbasi and L. M. Mackenzie, “A Flexible Approach for Modelling and Analysis of Feature Interactions in Service-Oriented Product Lines.,” *J. Softw.*, vol. 12, no. 10, pp. 823–830, 2017.

[10] A. S. Alghamdi, “Features Interaction Detection and Resolution in Smart home systems Using Agent-Based Negotiation Approach,” 2015.

[11] B. Vlaovič, A. Vreže, and Z. Brezočnik, “Applying automated model extraction for simulation and verification of real-life SDL specification with Spin,” *IEEE Access*, vol. 5, pp. 5046–5058, 2017.

[12] Z. Soufiane, E.-N. Abdeslam, and B. Slimane, “An SDL to Discrete-Time PROMELA Transformation of Home Area Network model,” in *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*, 2018, pp. 1–5.

[13] M. Dabaghchian and M. A. Azgomi, “Model checking the observational determinism security property using PROMELA and SPIN,” *Formal Aspects of Computing*, vol. 27, no. 5, pp. 789–804, 2015.

[14] A. de Lucia, V. Deufemia, C. Gravino, and M. Risi, “Detecting the behavior of design patterns through model checking and dynamic analysis,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 4, pp. 1–41, 2018.

The PROMELA Model

The complete source code of the model for both task 1 and task 2 is attached as a separate file and alternatively is shown below.

```
#define NofDoors 3                  // number of the doors. 0:doo1, 1:door2, 2:door3, 3:door4
#define CommonDoor    2             // door number 3 (has index 2) is common between room A and B.
#define Outside    0
#define ZoneA      1
#define ZoneB      2
// To define the four types of users and their authentication level to access the doors for both entering and exiting directions.
typedef AccessTopology
{
 byte        userLocation;    //1:RoomA, 2:RoomB, 0:Outside
 bit doorIn [4];
```

DOI: http://dx.doi.org/10.25098/6.1.30

```
 bit doorOut [4]
  };
// To connect doors, their locations (rooms), and their direction (inside and outside) together.
typedef DoorTopology
{
 short       doorLoc;
byte doorInside;
 byte doorOutside
};
AccessTopology access [4];
DoorTopology doorTop[4];
chan swipe = [0] of {byte  , byte, byte};   //to communicate between the users and doors

ltl p1 { (access[1].userLocation == ZoneB -> access[1].userLocation == ZoneA) }
//verification 3
ltl p2   {   ( access[2].userLocation == ZoneB ->   access[2].userLocation == Outside )  }
//verification 4
 proctype door(){
 byte u, d, uLoc;
 do
 :: swipe ?  u,d, uLoc ->     // If the doors recieve a swiping card request to access.
    if
    //Users authorisation check to enter the doors (checking outside directions of the doors).
   :: access[u].doorOut[d] == 1 &&  access[u].userLocation == Outside &&
   doorTop[d].doorOutside == 0 ->
   access[u].userLocation = doorTop[d].doorLoc;
  //Users authorisation check to exit the doors (checking inside directions of the doors).
  :: access[u].doorIn[d] ==1 &&  access[u].userLocation >= ZoneA  ->
    if
    ::  d == CommonDoor ->    //if a user tries to exit via door 3 (common door)
    byte temp = access[u].userLocation;
    access[u].userLocation = doorTop[temp].doorInside;
    //if a user tries to exit via the other doors.
    :: access[u].userLocation == doorTop[d].doorInside && d != CommonDoor ->
     access[u].userLocation = Outside;
     fi
    :: else ->   skip;
    fi
    od
     }
    proctype user(byte u){
    byte d = 0;
  do
  // counter for door numbers generator
  :: d != 3 -> d++;
  :: d != 0 -> d--;
  // users allow to access those doors which are in its locations.
  ::   access[u].userLocation == Outside  && d !=CommonDoor ->
 if
 :: access[u].userLocation ==doorTop[d].doorOutside     ->   // if the user tries to access one of the doors
end1: swipe ! u, d, access[u].userLocation;
 fi
 ::   access[u].userLocation ==ZoneA  ->  // if userlocation is in ZoneA
  if
 ::  d == CommonDoor ->  // if the user tries to exit from ZoneA and enter to ZoneB via door 3.
 end2: swipe ! u, d, access[u].userLocation;
// if the user tries to access the other doors
 :: access[u].userLocation == doorTop[d].doorInside  && d != CommonDoor ->
 end3: swipe ! u, d, access[u].userLocation;
:: else -> skip;
 fi
 // if the user tries to exit from ZoneA and enter to ZoneB via door 3.
 ::   access[u].userLocation == ZoneB ->
 if
 ::  d == CommonDoor ->  // if the user tries to exit from ZoneB and enter to ZoneA via door 3.
 end4: swipe ! u, d, access[u].userLocation;
  // if the user tries to access the other doors
 :: access[u].userLocation == doorTop[d].doorInside  && d != CommonDoor ->
 end5: swipe ! u, d, access[u].userLocation;
 :: else -> skip;
  fi
 :: else -> skip;
od
 }
 never{
 do
 :: (access[3].userLocation != Outside) -> break
 :: else -> skip
 od
 }
 init {
 // doorTop array inilialisation.
 doorTop[0].doorLoc = ZoneB;    //Door 1 in RoomB
 doorTop[0].doorInside = ZoneB;
```

DOI: http://dx.doi.org/10.25098/6.1.30

```
doorTop[0].doorOutside = Outside;

doorTop[1].doorLoc = ZoneB;    //Door2 in RoomB
doorTop[1].doorInside = ZoneB;
doorTop[1].doorOutside = Outside;

doorTop[2].doorLoc = ZoneA;      //Door3 in RoomA (and common with RoomB)
doorTop[2].doorInside = ZoneA;
doorTop[2].doorOutside = ZoneB;

doorTop[3].doorLoc = ZoneA;    //Door4 in RoomA
doorTop[3].doorInside = ZoneA;
doorTop[3].doorOutside = Outside;

//access array initialisation
access[0].doorIn[0] = 1;      // access[0] is UsersAB:
access[0].doorIn[1] = 1;
access[0].doorIn[2] = 1;
access[0].doorIn[3] = 1;
access[0].doorOut[0] = 1;
access[0].doorOut[1] = 1;
access[0].doorOut[2] = 1;
access[0].doorOut[3] = 1;
access[0].userLocation = Outside;

access[1].doorIn[0] = 1;    //access[1] is UsersA:
access[1].doorIn[1] = 1;
access[1].doorIn[2] = 1;
access[1].doorIn[3] = 0;
access[1].doorOut[0] = 1;
access[1].doorOut[1] = 1;
access[1].doorOut[2] = 1;
access[1].doorOut[3] = 1;
access[1].userLocation = Outside;

access[2].doorIn[0] = 1;    //access[2] is UsersB:
access[2].doorIn[1] = 1;
access[2].doorIn[2] = 0;
access[2].doorIn[3] = 0;
access[2].doorOut[0] = 1;
access[2].doorOut[1] = 1;
access[2].doorOut[2] = 0;
access[2].doorOut[3] = 0;
```

```
access[2].userLocation = Outside;

access[3].doorIn[0] = 0;   //access[3] is Others:
access[3].doorIn[1] = 0;
access[3].doorIn[2] = 0;
access[3].doorIn[3] = 0;
access[3].doorOut[0] = 0;
access[3].doorOut[1] = 0;
access[3].doorOut[2] = 0;
access[3].doorOut[3] = 0;
access[3].userLocation = Outside;

// running the processes:
atomic { run user(0); run user(1); run user(2); run user(3);  run door() };
}
```

DOI: http://dx.doi.org/10.25098/6.1.30