

# CA 548 - Secure Software Development, Assignment #1

Sebastian Wolfgarten, 55135811

April 8, 2006

## **Abstract**

In the first assignment of the course CA548 (Secure Software Development) the students were asked to formally specify an access control system and to verify its implementation as well as safety and liveness properties. Therefore the students were given a specification which they had to implement in Promela (PROcess MEta LAnguage), a high level language to specify system descriptions.

# Contents

<b>1</b>	<b>Design of my specification</b>	<b>3</b>
<b>2</b>	<b>Safety and liveness properties</b>	<b>3</b>
<b>3</b>	<b>Verification of the safety and liveness properties</b>	<b>4</b>
3.1	Test cases . . . . .	4
3.2	Formal verification with Spin . . . . .	11
<b>A</b>	<b>Assignment text</b>	<b>14</b>
<b>B</b>	<b>Declaration</b>	<b>14</b>

# 1 Design of my specification

Due to the fact that we were asked to use Spin and Promela in this assignment I have used these languages for my implementation. Spin<sup>1</sup> is an open-source verification tool for distributed systems that was developed in the 1980's at the Bell Labs. It uses Promela for specifying system descriptions in a high level language and is written in C which makes it available for various platforms including but not limited to Linux, MacOS, Sun Solaris and Microsoft Windows. Based on the specification of the access control system we were given (see appendix) I decided to split my implementation into three different parts:

- Central server/system
- Keypad(s)
- User process

The first part is the central server which contains most of the actual application logic and which processes the user input. Therefore according to the specification the server maintains a list of upto 10 valid user codes and sends a `valid_user` signal to the keypad in case a user entered a valid code. If an invalid code is entered, the server will send a `invalid_user` signal. Additionally if the same invalid code is entered three times in a row or if 10 invalid codes are entered sequentially at the same keypad, the central system will generate an alert. If a user enters a code which is one more than a valid code, the system will accept the code and generate a `duress_alert` signal. Finally the central system contains a built-in administrator code which can be used to add or remove codes or to cancel system or fire alarms.

Secondly there is a keypads beside each door that is connected to the central system and in which a user will enter keys that then will be send to the server for further processing. If the central system then validates a user code, the keypad will open a door for five seconds. If a code is not valid, the keypad will beep. Furthermore if the keypad cannot establish a connection to the central system it will automatically disengage the electronic lock on the door it is monitoring. Additionally each keypad has a connection to the fire alarm system. In case of a fire the keypad will immediately open the door. In my implementation only one keypad is supported. However this can easily be extended to support N keypads by starting N keypad processes.

The last part of my implementation are a number of user processes that are used to emulate user input. These processes will later on be used to verify the systems' safety and liveness properties. Finally overall development time of the solution outlined in this document was approx. 60 hours. It was developed on a Gentoo Linux system using only an editor (KWrite) and the command-line version of Spin.

# 2 Safety and liveness properties

Safety properties are defined<sup>2</sup> as "conditions that must always hold" and "nothing bad will happen". A typical example of a safety property would be that the

---

<sup>1</sup><http://www.spinroot.com>

<sup>2</sup>Dr. David Sinclair, "Introduction to temporal logic", URL: <http://www.computing.dcu.ie/~davids/courses/CA548/Temporal.Logic.pdf>

system must not lock the door in case of a fire alarm. Hence given the specification of the system I was able to identify the following safety properties:

1. In case of a fire alarm, the door will be opened immediately.
2. If the connection to the central system is not available, the door will be opened.
3. A user entering the same invalid code three times in a row, will trigger an alert.
4. 10 sequentially entered invalid codes will also trigger an alert.
5. A duress signal is generated, when a user enters a code that is one more than a valid code.

Furthermore the system has liveness properties that are defined<sup>3</sup> as "conditions that will eventually hold" and "something good will eventually happen". I do believe these are reactions/responses of the system to user input and provide the user with some kind of interaction. Based on the specification and my implementation I have identified the following liveness properties:

1. Only a valid code will open a door.
2. If a valid code is entered, the door will be opened for five seconds.
3. Entering an invalid code won't open a door and will cause the keypad to beep.
4. The user input must be verified/rejected directly after being entered.
5. If a user fails to complete entering a code within 10 seconds, the keypad will reset itself.

Let's now try to verify these properties.

## 3 Verification of the safety and liveness properties

### 3.1 Test cases

As described above, each user is supposed to enter a 4 digit code which is then send to the central server by the keypad. Therefore in order to test my implementation I have created a number of test cases whereas each test case targets a different situation. By examing the response of the system to these test cases, I am able to verify its behaviour and safety as well as liveness properties. My test cases are as follows:

1. Sending a valid user code
2. Sending an invalid user code

---

<sup>3</sup>Dr. David Sinclair, "Introduction to temporal logic", URL: [http://www.computing.dcu.ie/~davids/courses/CA548/Temporal\\_Logic.pdf](http://www.computing.dcu.ie/~davids/courses/CA548/Temporal_Logic.pdf)

3. Causing a duress alert signal
4. Sending an admin code
5. A user timeout occurs while entering a code
6. Sending the same invalid code 3 times in a row
7. Triggering a fire alarm
8. Sending 10 invalid/random codes in a row
9. Keypad loses connection to server

If the system handles these test cases correctly, this would be a good indicator of a correct implementation.

In the first test case a valid user code (7361) is entered at a given keypad. The keypad sends this code to the server which validates the code and as soon as this done, the door is opened. The following output illustrates this process (called "test\_case1" in the Promela code):

User process:

```
-----
Running 1st test case: Sending a valid user code.
The user entered: 7-3-6-1.
```

Keypad:

Server:

```
-----
Keypad retr: 7
Keypad retr: 3
Keypad retr: 6
Keypad retr: 1
Sending keys to server...
```

```
Server retr: 7-3-6-1
Verifying user input...
Code 7361 verified.
```

```
Valid code. Opening door.
Will close door in 5 seconds...
5 second(s).
4 second(s).
3 second(s).
2 second(s).
1 second(s).
Closing door!
```

The second test case is very similar and emulates an invalid code being send to the central system. The central server responds with an invalid\_user signal and thus an error message is generated ("test\_case2"):

User process:

-----  
Running 2nd test case: Sending an invalid user code.  
The user entered: 3-8-9-4.

Keypad:

Server:

-----  
Keypad retr: 3  
Keypad retr: 8  
Keypad retr: 9  
Keypad retr: 4  
Sending keys to server...

Server retr: 3-8-9-4  
Verifying user input..  
Code 3894 NOT verified.  
Please try again.

Unknown/invalid code.  
Please try again.

In the next test case a duress alert signal is generated by sending a code which is one more than a valid code (7362 rather than 7361). As a consequence of a duress signal, an alert is triggered and the door is opened ("test\_case3" in the Promela code):

User process:

-----  
Running 3rd test case: Sending a duress user code.  
The user entered: 7-3-6-2.

Keypad:

Server:

-----  
Keypad retr: 7  
Keypad retr: 3  
Keypad retr: 6  
Keypad retr: 2  
Sending keys to server...

Server retr: 7-3-6-2  
Verifying user input...  
Code 7361 verified.  
ALERT: Duress code entered!  
Sending alert signal...

Valid code. Opening door.  
Will close door in 5 seconds...  
5 second(s).  
4 second(s).

```
3 second(s).
2 second(s).
1 second(s).
Closing door!
```

According to the specification the system also has a fixed administrator code which can be used to add or remove codes and to cancel system or fire alarms. Hence the next test case sends the fixed administrator code (7781) to the central system ("test\_case4"):

User process:

```
-----
Running 4th test case: Sending an admin code.
The user entered: 7-7-8-1.
```

Keypad:

Server:

```
-----
Keypad retr: 7
Keypad retr: 7
Keypad retr: 8
Keypad retr: 1
Sending keys to server...
```

```
Server retr: 7-7-8-1
Verifying user input...
Code 7781 NOT verified.
Please try again.
```

Admin code entered!

```
::: ADMIN MENU :::
(1) Add new user.
(2) Delete user.
(3) Modify existing user.
```

Unknown/invalid code.  
Please try again.

The fifth test case is a timeout that occurs while a user enters a code. According to the specification, in such a case the keypad should terminate and not query the central server at all. For the actual implementation I have decided to let the user enter "99" as their last input which is then treated by the keypad as a timeout. Due to the fact the central system is not queried at all, there is no output being generated by the server (Promela code: "test\_case5"):

User process:

```
-----
Running 5th test case: A timeout occurs while entering a code.
The user entered: 1-4-6-99.
```

Keypad:

Server:

-----  
Keypad retr: 1  
Keypad retr: 4  
Keypad retr: 6  
Keypad retr: 99  
A timeout occurred while  
entering the keys.

In the next test case the same invalid code (1461) is entered three times in a row.  
As defined by the specification this will lead to an alarm signal being generated  
("test\_case6"):

User process:

-----  
Running 6th test case: Same invalid code is entered 3 times in a row.  
The user entered: 1-4-6-1.

Keypad:

Server:

-----  
Keypad retr: 1  
Keypad retr: 4  
Keypad retr: 6  
Keypad retr: 1  
Sending keys to server...

Server retr: 1-4-6-1  
Verifying user input...  
Code 1461 NOT verified.  
Please try again.

Unknown/invalid code.  
Please try again.

Keypad retr: 1  
Keypad retr: 4  
Keypad retr: 6  
Keypad retr: 1  
Sending keys to server...

Server retr: 1-4-6-1  
Verifying user input...  
Code 1461 NOT verified.  
Please try again.

Unknown/invalid code.  
Please try again.

Keypad retr: 1  
Keypad retr: 4  
Keypad retr: 6  
Keypad retr: 1



Keypad retr: 1  
Keypad retr: 1  
Keypad retr: 2  
Sending keys to server...

Server retr: 1-1-1-2  
Verifying user input...  
Code 1112 NOT verified.  
Please try again.

Unknown/invalid code. Please try again.

[ ... ]

Keypad retr: 3  
Keypad retr: 3  
Keypad retr: 3  
Keypad retr: 3  
Sending keys to server...

Server retr: 3-3-3-3  
Verifying user input...  
Code 3333 NOT verified.  
Please try again.  
ALARM: 10 invalid codes were  
entered in sequence!

Unknown/invalid code. Please try again.

Finally in the last test case the keypad is unable to establish a connection with the central server and therefore opens the door it is monitoring ("test\_case9"):

User process:

-----  
Running 9th test case: Keypad lost connection to server.

Keypad:

Server:

-----  
Server is shutting down...

timeout  
The keypad was unable to  
connect to the central server!  
Door will be opened.

As the output above indicates, the system successfully handles all of my test cases correctly. I will now try to verify my implementation formally:

## 3.2 Formal verification with Spin

I verified my solution using the Spin<sup>4</sup> utility by firstly performing a syntax check:

```
$ ./spin -a ca548-wolfgarten-assignment1.pml
```

Next I ran a simulation run with columnated output:

```
$ ./spin -c ca548-wolfgarten-assignment1.pml
```

```
proc 0 = :init:
proc 1 = central_server
proc 2 = test_case1
proc 3 = test_case2
proc 4 = test_case3
proc 5 = test_case4
proc 6 = test_case5
proc 7 = test_case6
proc 8 = test_case7
proc 9 = test_case8
proc 10 = test_case9
q\p  0  1  2  3  4  5  6  7  8  9  10
  1  test_sequencer!case1
  1  . . test_sequencer?case1
      Running 1st test case: Sending a valid user code.
      Central system up and running...

      Activating keypad.
proc 11 = keypad
                                          Keypad number 0 started.

q\p  0  1  2  3  4  5  6  7  8  9  10  11
  2  . . . . . . . . . . . keypad_to_user!keypad_ready
  2  . . keypad_to_user?keypad_ready
  4  . . user_to_keypad!7
  4  . . user_to_keypad!3
  4  . . user_to_keypad!6
  4  . . . . . . . . . . . user_to_keypad?7
  4  . . user_to_keypad!1
      The user entered: 7-3-6-1.
  1  . . test_sequencer!case2
  4  . . . . . . . . . . . user_to_keypad?3
  1  . . . test_sequencer?case2
  4  . . . . . . . . . . . user_to_keypad?6
      Running 2nd test case: Sending an invalid user code.
  4  . . . . . . . . . . . user_to_keypad?1
      Keypad retr: 7
      Keypad retr: 3
```

---

<sup>4</sup>Unknown author, "BUILDING AND VERIFYING Spin MODELS", URL: <http://spinroot.com/spin/Man/Roadmap.html>

```

Keypad retr: 6
Keypad retr: 1
Sending keys to server...
3 . . . . . keypad_to_server!7,3,6,1
3 . keypad_to_server?7,3,6,1
Server retr: 7-3-6-1
Verifying user input...
Code 7361 verified.

[ ... ]

4 . . . . . user_to_keypad?9
4 . . . . . user_to_keypad?9
4 . . . . . user_to_keypad?9
Keypad retr: 9
Keypad retr: 9
Keypad retr: 9
Keypad retr: 9
Sending keys to server...
3 . . . . . keypad_to_server!9,9,9,9
3 . keypad_to_server?9,9,9,9
Server retr: 9-9-9-9
Verifying user input...
Server is shutting down...
timeout

The keypad was unable to connect
to the central server!
Door will be opened.
2 . . . . . keypad_to_user!keypad_ready
1 . . . . . test_sequencer?keypad_done
1 . . . . . test_sequencer!server_done

```

-----  
final state:  
-----

12 processes created

This provided with a nice overview of the states the model is in at any given time. Afterwards I generated a verifier as outlined in the Spin documentation:

```
$ ./spin -a ca548-wolfgarten-assignment1.pml
```

Furthermore I have compiled the protocol analyzer ("pan") as follows:

```
$ gcc -o pan pan.c
```

Finally I ran a full verification with  $2^{27}$  million states which is the maximum value I can use on my test system (P4, 3 GHz, 512 MB RAM, running KDE on Gentoo Linux):

```
$ time ./pan -w27
```

```
hint: this search is more efficient if pan.c is compiled -DSAFETY
(Spin Version 4.2.6 -- 27 October 2005)
```

+ Partial Order Reduction

Full statespace search for:

never claim - (none specified)  
assertion violations +  
acceptance cycles - (not selected)  
invalid end states +

State-vector 184 byte, depth reached 874, errors: 0

20879 states, stored  
4704 states, matched  
25583 transitions (= stored+matched)  
420 atomic steps

hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

4.009 equivalent memory usage for states (stored\*(State-vector + overhead))  
3.453 actual memory usage for states (compression: 86.13%)  
State-vector as stored = 157 byte + 8 byte overhead  
536.871 memory used for hash table (-w27)  
0.320 memory used for DFS stack (-m10000)  
0.172 other (proc and chan stacks)  
0.074 memory lost to fragmentation  
540.570 total actual memory usage

unreached in proctype :init:

(0 of 12 states)

unreached in proctype central\_server

(0 of 89 states)

unreached in proctype keypad

(0 of 87 states)

unreached in proctype test\_case1

(0 of 12 states)

unreached in proctype test\_case2

(0 of 12 states)

unreached in proctype test\_case3

(0 of 12 states)

unreached in proctype test\_case4

(0 of 12 states)

unreached in proctype test\_case5

(0 of 12 states)

unreached in proctype test\_case6

(0 of 28 states)

unreached in proctype test\_case7

(0 of 6 states)

unreached in proctype test\_case8

(0 of 120 states)

unreached in proctype test\_case9

(0 of 6 states)

```
real    0m33.226s
user    0m0.188s
sys     0m1.156s
```

As the output above indicates, the command used almost 540 MB of RAM and took 33 seconds to run. When compiling the protocol analyzer with the `-DSAFETY` option (`"gcc -o pan pan.c -DSAFETY"`) this search could be done in even two seconds less. Most importantly no errors, invalid or unreachable states were encountered during the analysis which indicates that my implementation is correct.

## A Assignment text

The following text is an excerpt from the assignment given to the students<sup>5</sup> by Dr. David Sinclair and specifies the system to be implemented:

SafeWithUs Ltd. designs and builds access control systems. A potential large corporate customer is interested in installing the access control system in all their offices but first need to be assured that the system will secure their premises. As part of this assurance process SafeWithUs Ltd. want you to formally specify their system and to prove safety and liveness properties about the system.

The access control system consists of a keypad beside each door. A user enters a 4 digit code on the keypad. The keypad communicates with a central system that maintains a list of valid user codes and if the user code is valid the central system sends a `valid_user` signal to the keypad. The keypad will disengage the electric lock on the door it is monitoring for 5 seconds. If the user code is not a valid user code the central system will send an `invalid_user` code signal to the keypad. The keypad will beep. If the same invalid code is entered 3 times in a row at the same keypad, or if 10 invalid codes are entered sequentially at the same keypad, the central system generates an alarm signal. If a user fails to complete entering a 4 digit code in 10 seconds the code entry process is terminated without querying the central system. If a user enters a user code that is one more than a valid user code the system will accept the user code as valid but will generate a special `duress_alert` signal. If the keypad cannot query the central system it will also disengage the electric lock on its door. Each keypad also has an input from the fire alarm system. In the event of a fire all keypads are immediately disengage the electric locks on their doors. The central system can hold upto 10 different user codes. This system also has a special fixed administrator code that can add or remove user codes from the system, and cancel any system alarms or alerts.

## B Declaration

The given Promela code performs as required by the problem specification. I herewith confirm that the code is fully based on my own work. I have not received any assistance beyond what is normal, and I have cited any sources from

---

<sup>5</sup>[http://www.computing.dcu.ie/~davids/courses/CA548/CA548\\_assign.html](http://www.computing.dcu.ie/~davids/courses/CA548/CA548_assign.html)

which I have borrowed. I have not given a copy of my work, or part of my work, to anyone. I am aware that copying or giving a copy may have serious consequences, including failing the module or course.

Sebastian Wolfgarten (11. April 2006)