Project Report

Software Testing and Validation

Prof. João Dias Pereira

Group 01

Alexandra Rodrigues	**Eduardo Miranda**	**Rodrigo Pinto**
95528	95569	95666

Instituto Superior Técnico - Alameda

MEIC-A

2022/2023

Contents

1 Method-Scope Tests

1.1 `sendSMS` method

A non-busy turned-on terminal can send an SMS to another turned-on terminal. The return value of this method indicates whether the SMS was successfully delivered or not. If the length of the SMS is valid, and the destination terminal is turned on, the method returns true. If the length of the SMS is valid but the destination terminal is invalid, the method returns false. Finally, if the length of the SMS is invalid, then this method throws the `IllegalArgumentException`.

1.1.1 Test Pattern – Category-Partition Test

1.1.2 Functions

- **Primary Function**

  Send an SMS from a non-busy turned-on terminal to another turned-on terminal. This method should return true, successfully delivered an SMS, under these conditions:

  - if the length of the SMS is valid;
  - and the destination terminal is turned on.

  This method should return false, under these conditions:

  - if the length of the SMS is valid;
  - and the destination terminal is invalid.

- **Secondary Function**

  Throw `IllegalArgumentException` if conditions aren't met:

  - msg length $\notin [10, 200]$

  Throw `InvalidInvocationException` if conditions aren't met:

  - the source terminal is not turned on;

  - or the source terminal is busy.

Store the SMS message if sent successfully.

1.1.3 Input/Output Parameters

- **Input**
  - Message `msg` to be sent (`msg.length()`);
  - Destination terminal `to` (`to.getMode()`);
  - Terminal `from` in cause (`from.getMode()`);
- **Output**
  - A boolean corresponding to the success or failure of the delivery of the message;
  - The updated list `list` of SMS sent by the source terminal;
  - Exception (`IllegalArgumentException` , `IllegalInvocationException`).

1.1.4 Categories & Choices

Variable	Category	Choices
msg.length()	Valid (msg.length() in [10, 200])	msg.length() = 10, msg.length() = 200, msg.length() = some x in]10, 200[
	Invalid1 (msg.length() <10) [error]	msg.length() = 9, msg.length() = 3
	Invalid2 (msg.length() >200) [error]	msg.length() = 201, msg.length() = 300
to	Not defined [error]	null
	Defined	any Terminal
to.getMode()	Not Off	to.getMode() = some x in {idle, silence, busy}
	Off	to.getMode() = off
from.getMode()	Idle	from.getMode() = idle
	Silence	from.getMode() = silence
	Busy [error]	from.getMode() = busy
	Off1 [error]	from.getMode() = off

Table 1: Set of `sendSMS` method input parameters broken into categories and test case choices

1.1.5 Constraints

- If `msg.length()` is invalid there is no need to test all possible states because the `IllegalArgumentException` should always be thrown.
- If `to` is not defined there is no need to test all possible states because the `InvalidInvocationException` should always be thrown.
- If `from.getMode()` is busy or off there is no need to test all possible states because the `InvalidInvocationException` should always be thrown.

1.1.6 Test Cases

	Input				Expected Output		
Test Case	msg.length()	to	to.getMode()	from.getMode()	Returned	Exception	list
1	10	to	idle	idle	True	-	list $\cup$ {SMS}
2	10	to	off	idle	False	-	list
3	10	to	idle	silence	True	-	list $\cup$ {SMS}
4	10	to	off	silence	False	-	list
5	200	to	silence	idle	True	-	list $\cup$ {SMS}
6	200	to	off	idle	False	-	list
7	200	to	idle	silence	True	-	list $\cup$ {SMS}
8	200	to	off	silence	False	-	list
9	95	to	idle	idle	True	-	list $\cup$ {SMS}
10	95	to	off	idle	False	-	list
11	95	to	silence	silence	True	-	list $\cup$ {SMS}
12	95	to	off	silence	False	-	list
13	9	to	busy	idle	-	IllegalArgumentException	list
14	3	to	silence	silence	-	IllegalArgumentException	list
15	201	to	idle	idle	-	IllegalArgumentException	list
16	300	to	busy	silence	-	IllegalArgumentException	list
17	95	null	idle	silence	-	InvalidInvocationException	list
18	95	to	silence	busy	-	InvalidInvocationException	list
19	95	to	idle	off	-	InvalidInvocationException	list

Table 2: Set of test cases for the `sendSMS` method after constraints are applied

Description of the test cases

- In total we have 19 test cases;
- The `to` parameter can be null, so it has a Not defined category;
- The expected result for each test case indicates the output of the MUT for that possible combination;
- Every combination of choices is tested and each exception is thrown at least once.

1.2 `computeCallUnitCost` method

The responsibility of `computeCallUnitCost` method is to determine the unit cost of voice communications (cost per minute) taking into account customer level, number of terminals, calls and SMS made by the customer.

1.2.1 Test Pattern – Combinational Function Test

1.2.2 Decision Tree

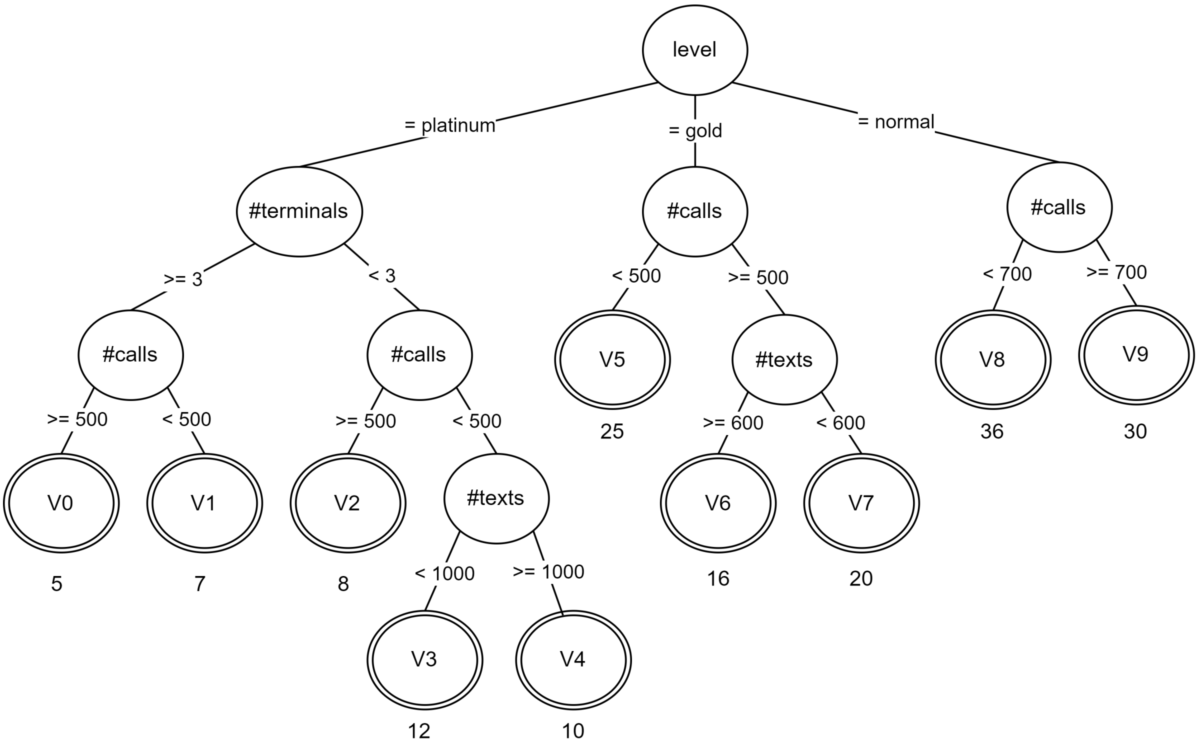


Figure 1: Decision tree describing the output given by `computeCallUnitCost` based on the level, number of terminals, SMS's sent and calls made by the client.

Boundary conditions for each variant

- V_0 : level = platinum $\wedge$ #terminals $>=$ 3 $\wedge$ #calls $>=$ 500
- V_1 : level = platinum $\wedge$ #terminals $>=$ 3 $\wedge$ #calls $<$ 500
- V_2 : level = platinum $\wedge$ #terminals $<$ 3 $\wedge$ #calls $>=$ 500
- V_3 : level = platinum $\wedge$ #terminals $<$ 3 $\wedge$ #calls $<$ 500 $\wedge$ #texts $<$ 1000
- V_4 : level = platinum $\wedge$ #terminals $<$ 3 $\wedge$ #calls $<$ 500 $\wedge$ #texts $>=$ 1000
- V_5 : level = gold $\wedge$ #calls $<$ 500
- V_6 : level = gold $\wedge$ #calls $>=$ 500 $\wedge$ #texts $>=$ 600
- V_7 : level = gold $\wedge$ #calls $>=$ 500 $\wedge$ #texts $<$ 600
- V_8 : level = normal $\wedge$ #calls $<$ 700
- V_9 : level = normal $\wedge$ #calls $>=$ 700

1.2.3 Domain Matrices for Variants

V_0			**Test Cases**					
Variable	**Condition**	**Type**	1	-	2	-	3	-
level	= platinum	ON	platinum					
		OFF		normal				
	Typical	IN			platinum	platinum	platinum	platinum
#terminals	$>=$ 3	ON			3			
		OFF				2		
	Typical	IN	4	8			5	6
#calls	$>=$ 500	ON					500	
		OFF						499
	Typical	IN	502	510	520	530		
#texts	Typical	IN	50	60	70	80	90	100
Expected Result			5	V_8	5	V_2	5	V_1

Table 3: V_0 domain matrix

V_1			**Test Cases**					
Variable	**Condition**	**Type**	4	-	5	-	-	6
level	= platinum	ON	platinum					
		OFF		normal				
	Typical	IN			platinum	platinum	platinum	platinum
#terminals	>= 3	ON			3			
		OFF				2		
	Typical	IN	6	7			4	5
#calls	<500	ON					500	
		OFF						499
	Typical	IN	400	300	200	100		
#texts	Typical	IN	50	60	70	80	90	110
Expected Result			7	V_8	7	V_3	V_0	7

Table 4: V_1 domain matrix

V_2			**Test Cases**					
Variable	**Condition**	**Type**	7	-	-	8	9	-
level	= platinum	ON	platinum					
		OFF		normal				
	Typical	IN			platinum	platinum	platinum	platinum
#terminals	<3	ON			3			
		OFF				2		
	Typical	IN	1	1			1	1
#calls	>= 500	ON					500	
		OFF						499
	Typical	IN	510	520	530	540		
#texts	Typical	IN	50	60	70	80	90	100
Expected Result			8	V_8	V_0	8	8	V_3

Table 5: V_2 domain matrix

V_3			Test Cases							
Variable	**Condition**	**Type**	10	-	-	11	-	12	-	13
level	= platinum	ON	platinum							
		OFF		gold						
	Typical	IN			platinum	platinum	platinum	platinum	platinum	platinum
#terminals	<3	ON			3					
		OFF				2				
	Typical	IN	1	1			1	1	1	1
#calls	<500	ON					500			
		OFF						499		
	Typical	IN	50	80	100	200			300	400
#texts	<1000	ON							1000	
		OFF								999
	Typical	IN	400	450	600	700	800	900		
Expected Result			12	V_5	V_1	12	V_2	12	V_4	12

Table 6: V_3 domain matrix

V_4			Test Cases							
Variable	**Condition**	**Type**	14	-	-	15	-	16	17	-
level	= platinum	ON	platinum							
		OFF		gold						
	Typical	IN			platinum	platinum	platinum	platinum	platinum	platinum
#terminals	<3	ON			3					
		OFF				2				
	Typical	IN	1	1			1	1	1	1
#calls	<500	ON					500			
		OFF						499		
	Typical	IN	50	80	100	200			300	400
#texts	$>= 1000$	ON							1000	
		OFF								999
	Typical	IN	1100	1200	1300	1400	1500	1600		
Expected Result			10	V_5	V_1	10	V_2	10	10	V_3

Table 7: V_4 domain matrix

V_5			Test Cases			
Variable	**Condition**	**Type**	18	-	-	19
level	= gold	ON	gold			
		OFF		normal		
	Typical	IN			gold	gold
#calls	<500	ON			500	
		OFF				499
	Typical	IN	300	400		
#texts	Typical	IN	525	780	1200	1400
#terminals	Typical	IN	1	3	6	8
Expected Result			25	V_8	V_6	25

Table 8: V_5 domain matrix

V_6			Test Cases					
Variable	**Condition**	**Type**	20	-	21	-	22	-
level	= gold	ON	gold					
		OFF		normal				
	Typical	IN			gold	gold	gold	gold
#calls	>= 500	ON			500			
		OFF				499		
	Typical	IN	910	960			800	900
#texts	>= 600	ON					600	
		OFF						599
	Typical	IN	700	800	850	900		
#terminals	Typical	IN	1	2	3	4	5	6
Expected Result			16	V_9	16	V_5	16	V_7

Table 9: V_6 domain matrix

V_7			Test Cases					
Variable	**Condition**	**Type**	23	-	24	-	-	25
level	= gold	ON	gold					
		OFF		normal				
	Typical	IN			gold	gold	gold	gold
#calls	>= 500	ON			500			
		OFF				499		
	Typical	IN	910	960			800	900
#texts	<600	ON					600	
		OFF						599
	Typical	IN	100	200	300	400		
#terminals	Typical	IN	1	2	3	4	5	6
Expected Result			20	V_9	20	V_5	V_6	20

Table 10: V_7 domain matrix

V_8			**Test Cases**			
Variable	**Condition**	**Type**	26	-	-	27
level	= normal	ON	normal			
		OFF		gold		
	Typical	IN			normal	normal
#calls	$<$700	ON			700	
		OFF				699
	Typical	IN	200	400		
#texts	Typical	IN	150	400	600	725
#terminals	Typical	IN	1	2	5	8
Expected Result			36	V_5	V_9	36

Table 11: V_8 domain matrix

V_9			**Test Cases**			
Variable	**Condition**	**Type**	28	-	29	-
level	= normal	ON	normal			
		OFF		gold		
	Typical	IN			normal	normal
#calls	$>=$ 700	ON			700	
		OFF				699
	Typical	IN	800	900		
#texts	Typical	IN	150	400	600	725
#terminals	Typical	IN	1	2	5	8
Expected Result			30	V_7	30	V_8

Table 12: V_9 domain matrix

Description of the test cases

- In total we have 29 test cases;
- We made a domain matrix for each variant in order to exercise all the branches in the graph. In the matrix, each row represents a set of input values and each column a valid or invalid combination of instance variables;
- For all conditions, relational conditions and nonscalar type, we have one On point and one OFF point;
- The expected results marked with a variant number are test cases that belong to another variant, so we don't need to repeat them.

2 Class-Scope Tests

2.1 `TerminalNetwork` class

A terminal network has a maximum number of clients that cannot exceed 50000, and the name of each client is a unique identifier. A terminal network has a name, and the number of characters in the name must be greater than or equal to 3 and less than 10.

2.1.1 Test Pattern – Non-Modal Class Test

2.1.2 Class Invariant

`TerminalNetwork` Variables	
Variable	**Type**
`name`	`String`
`maxClients`	`int`
`terminals`	`List<Terminal>`
`clients`	`List<Client>`

Table 13: `TerminalNetwork` class' variables and their respective types

Domain restrictions

- A terminal network has a name, and the number of characters in the name must be greater than or equal to 3 and less than 10: `3 <= name.length() < 10`
- A terminal network has a maximum number of clients: `maxClients <= 50000 && clients.size() <= maxClients`
- The name of each client is a unique identifier within the context of the terminal network: $\forall_{c_1,c_2 \in clients} \; c_1.getName() = c_2.getName() \implies c_1 = c_2$ (condition 3)

The logical conjunction of all of these restrictions makes up the Class Invariant.

2.1.3 On and Off points

On and Off points for the **`TerminalNetwork`** invariant		
Boundary Condition	**On point**	**Off point**
`name.length()` >= 3	3	2
`name.length()` < 10	10	9
`maxClients` <= 50000	50000	50001
`clients.size() <= maxClients`	30000 (`maxClients` = 30000)	30001 (`maxClients` = 30000)
condition 3	T	F

Table 14: On and Off points for the `TerminalNetwork` class' invariant boundaries

2.1.4 Domain Matrix

Boundary			Test Cases									
Variable	Condition	Type	1	2	3	4	5	6	7	8	9	10
name.length()	>= 3	ON	3									
		OFF		2								
	<10	ON			10							
		OFF				9						
	Typical	IN					4	5	6	7	8	5
maxClients	<= 50000	ON					50000					
		OFF						50001				
	Typical	IN	50	240	1780	10000			30000	30000	25000	35000
client.size()	<= maxClients	ON							30000			
		OFF								30001		
	Typical	IN	30	220	1700	9900	15000	20000			22000	28000
c.getName()	condition 3	ON									T	
		OFF										F
	Typical	IN	T	T	T	T	T	T	T	T		
Expected Result			✓	✗	✗	✓	✓	✗	✓	✗	✓	✗

Table 15: `TerminalNetwork` class domain matrix

Description of the test cases

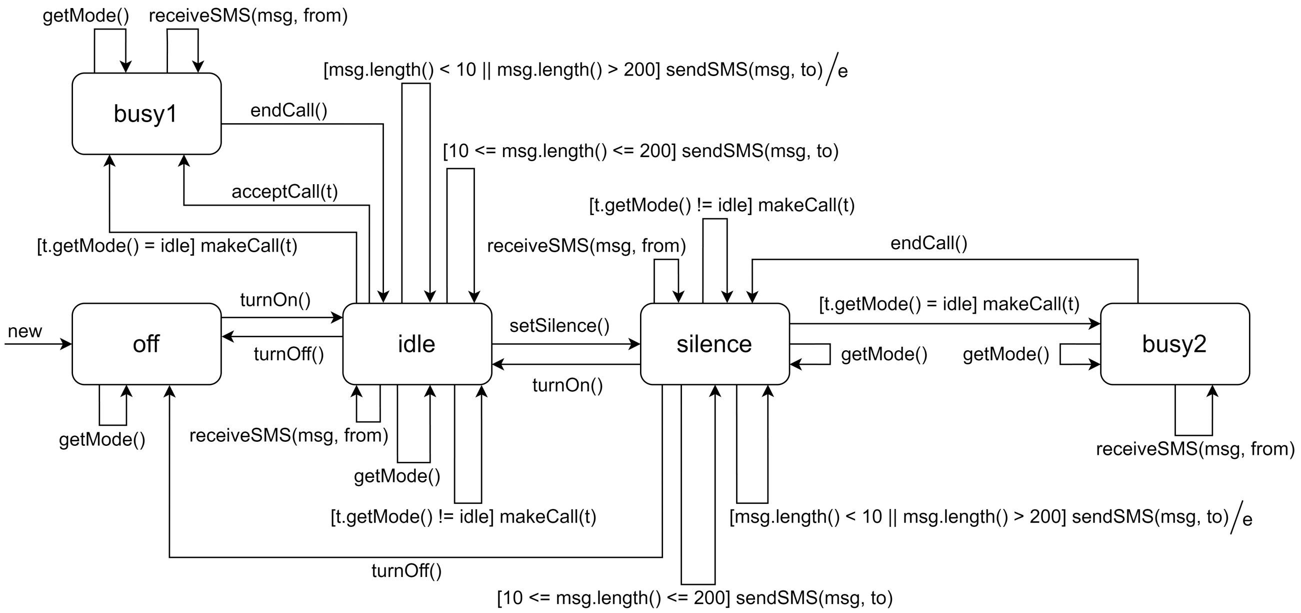
- In total we have 10 test cases;
- In the matrix, each row represents a set of input values and each column a valid or invalid combination of instance variables (✓- accepted, ✗- rejected).

2.2 `Terminal` class

2.2.1 Test Pattern – Modal Class Test

2.2.2 Finite State Machine

We started by designing the state machine diagram that represents the all states of the `Terminal` class with their respective transitions.



getMode()
receiveSMS(msg, from)
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to)/e
busy1
endCall()
[10 <= msg.length() <= 200] sendSMS(msg, to)
acceptCall(t)
[t.getMode() != idle] makeCall(t)
[t.getMode() = idle] makeCall(t)
receiveSMS(msg, from)
endCall()
turnOn()
[t.getMode() = idle] makeCall(t)
new
off
turnOff()
idle
setSilence()
silence
getMode()
getMode()
busy2
turnOn()
getMode()
receiveSMS(msg, from)
receiveSMS(msg, from)
getMode()
[t.getMode() != idle] makeCall(t)
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to)/e
turnOff()
[10 <= msg.length() <= 200] sendSMS(msg, to)

Figure 2: `Terminal` class state machine, representing the class' states and transitions between them

2.2.3 Truth table

State	Message	Condition	Next State
idle	`sendSMS(msg,to)`	10 <= `msg.length()` <= 200	idle
idle	`sendSMS(msg,to)`	`msg.length()` < 10 || `msg.length()` > 200 /e	idle
idle	`makeCall(t)`	`t.getMode()` = idle	busy1
idle	`makeCall(t)`	`t.getMode()` != idle	idle
silence	`sendSMS(msg,to)`	10 <= `msg.length()` <= 200	silence
silence	`sendSMS(msg,to)`	`msg.length()` < 10 || `msg.length()` > 200 /e	silence
silence	`makeCall(t)`	`t.getMode()` = idle	busy2
silence	`makeCall(t)`	`t.getMode()` != idle	silence

Table 16: Full expansion of conditional transition variants

As we can see from the truth table and the state diagram, all conditional transitions of the CUT are already displayed in the state diagram, so we can assume that the state diagram is complete.

2.2.4 Transition Tree

After that, we generated an initial transition tree based on the state diagram.

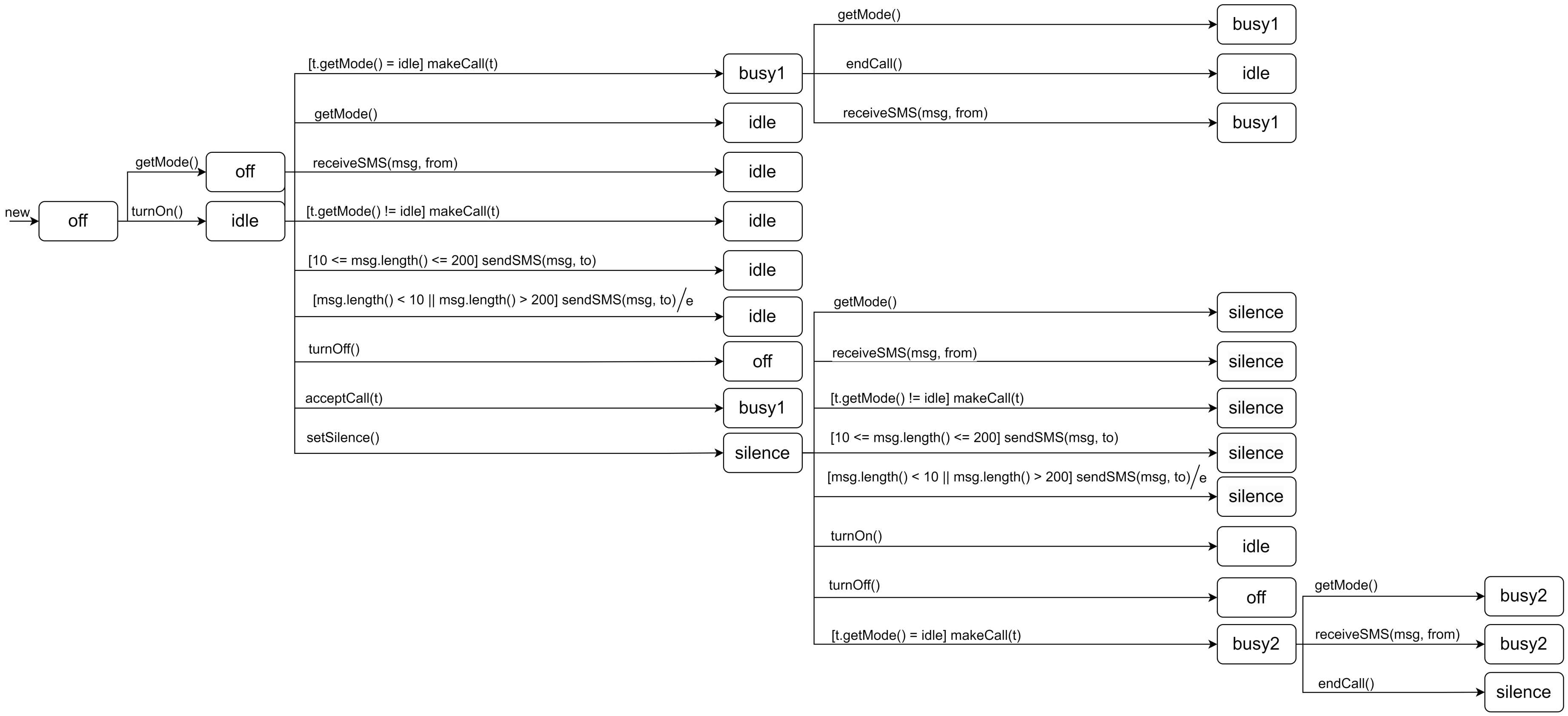

new
off
getMode()
off
turnOn()
idle
[t.getMode() = idle] makeCall(t)
busy1
getMode()
idle
receiveSMS(msg, from)
idle
[t.getMode() != idle] makeCall(t)
idle
[10 <= msg.length() <= 200] sendSMS(msg, to)
idle
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to)/e
idle
turnOff()
off
acceptCall(t)
busy1
setSilence()
silence
getMode()
busy1
endCall()
idle
receiveSMS(msg, from)
busy1
getMode()
silence
receiveSMS(msg, from)
silence
[t.getMode() != idle] makeCall(t)
silence
[10 <= msg.length() <= 200] sendSMS(msg, to)
silence
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to)/e
silence
turnOn()
idle
turnOff()
off
[t.getMode() = idle] makeCall(t)
busy2
getMode()
busy2
receiveSMS(msg, from)
busy2
endCall()
silence

Figure 3: `Terminal` class transition tree (sneak paths are not represented)

2.2.5 Conformance Test Suite

Next, we generated a conformance test suit based on the transition tree above where each row it’s a different possible path.

Run	Test Run/Event Path					Expected Terminal State	Exception
	Level 1	Level 2	Level 3	Level 4	Level 5		
1	new					off	No
2	new	getMode()				off	No
3	new	turnOn()				idle	No
4	new	turnOn()	getMode()			idle	No
5	new	turnOn()	receiveSMS(msg, from)			idle	No
6	new	turnOn()	[t.getMode() != idle] makeCall(t)			idle	No
7	new	turnOn()	[10 <= msg.length() <= 200] sendSMS(msg, to)			idle	No
8	new	turnOn()	[msg.length() <10 ∥ msg.length() >200] sendSMS(msg, to)			idle	Yes
9	new	turnOn()	[t.getMode() = idle] makeCall(t)			busy1	No
10	new	turnOn()	turnOff()			off	No
11	new	turnOn()	acceptCall(t)			busy1	No
12	new	turnOn()	setSilence()			silence	No
13	new	turnOn()	[t.getMode() = idle] makeCall(t)	endCall()		idle	No
14	new	turnOn()	[t.getMode() = idle] makeCall(t)	getMode()		busy1	No
15	new	turnOn()	[t.getMode() = idle] makeCall(t)	receiveSMS(msg, from)		busy1	No
16	new	turnOn()	setSilence()	getMode()		silence	No
17	new	turnOn()	setSilence()	receiveSMS(msg, from)		silence	No
18	new	turnOn()	setSilence()	[t.getMode() != idle] makeCall(t)		silence	No
19	new	turnOn()	setSilence()	[10 <= msg.length() <= 200] sendSMS(msg, to)		silence	No
20	new	turnOn()	setSilence()	[msg.length() <10 ∥ msg.length() >200] sendSMS(msg, to)		silence	Yes
21	new	turnOn()	setSilence()	turnOn()		idle	No
22	new	turnOn()	setSilence()	turnOff()		off	No
23	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)		busy2	No
24	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	getMode()	busy2	No
25	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	receiveSMS(msg, from)	busy2	No
26	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	endCall()	silence	No

Table 17: `Terminal` class conformance test suite

2.2.6 Test data

Then we developed test data for each path with a boundary condition.

makeCall in idle		
Condition	**ON**	**OFF**
[t.getMode(t) != idle]	idle	silence
[t.getMode(t) = idle]	idle	silence

Table 18: Test data for `makeCall` in idle

sendSMS in idle			
Condition		**ON**	**OFF**
[10 <= msg.length() <= 200]	msg.length() >= 10	10	9
	msg.length() <= 200	200	201
[msg.length() <10 || msg.length() >200]	msg.length() <10	10	9
	msg.length() >200	200	201

Table 19: Test data for `sendSMS` in idle

makeCall in silence		
Condition	**ON**	**OFF**
[t.getMode(t) != idle]	idle	silence
[t.getMode(t) = idle]	idle	silence

Table 20: Test data for `makeCall` in silence

sendSMS in silence			
Condition		**ON**	**OFF**
[10 <= msg.length() <= 200]	msg.length() >= 10	10	9
	msg.length() <= 200	200	201
[msg.length() <10 || msg.length() >200]	msg.length() <10	10	9
	msg.length() >200	200	201

Table 21: Test data for `sendSMS` in silence

2.2.7 Sneak Path Test Suite

Finally, we developed the Sneak Path Test Suite by building a Transition table.

Events	**States**				
	off	idle	silence	busy1	busy2
`turnOn`	✓	PSP	✓	PSP	PSP
`turnOff`	PSP	✓	✓	PSP	PSP
`getMode`	✓	✓	✓	✓	✓
`receiveSMS`	PSP	✓	✓	✓	✓
`sendSMS`	PSP	?	?	PSP	PSP
`makeCall`	PSP	?	?	PSP	PSP
`acceptCall`	PSP	✓	PSP	PSP	PSP
`setSilence`	PSP	✓	PSP	PSP	PSP
`endCall`	PSP	PSP	PSP	✓	✓

Table 22: `Terminal` class transition table

And we added each PSP from the previous table to the transition tree and completed the conformance test suite.

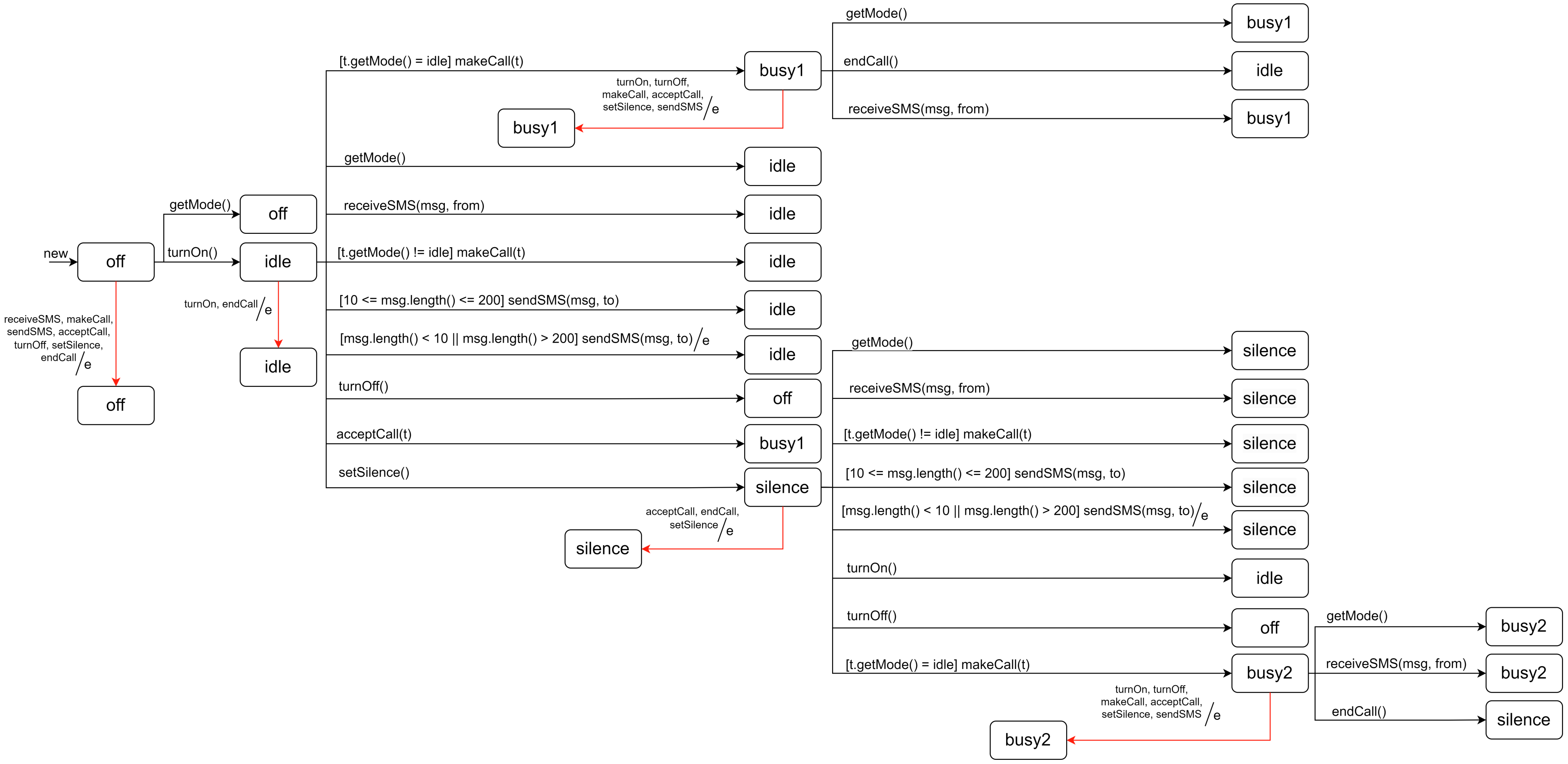

new
off
getMode()
off
turnOn()
idle
receiveSMS, makeCall, sendSMS, acceptCall, turnOff, setSilence, endCall / e
off
turnOn, endCall / e
idle
[t.getMode() = idle] makeCall(t)
busy1
turnOn, turnOff, makeCall, acceptCall, setSilence, sendSMS / e
busy1
getMode()
busy1
endCall()
idle
receiveSMS(msg, from)
busy1
getMode()
idle
receiveSMS(msg, from)
idle
[t.getMode() != idle] makeCall(t)
idle
[10 <= msg.length() <= 200] sendSMS(msg, to)
idle
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to) / e
idle
turnOff()
off
acceptCall(t)
busy1
setSilence()
silence
acceptCall, endCall, setSilence / e
silence
getMode()
silence
receiveSMS(msg, from)
silence
[t.getMode() != idle] makeCall(t)
silence
[10 <= msg.length() <= 200] sendSMS(msg, to)
silence
[msg.length() < 10 || msg.length() > 200] sendSMS(msg, to) / e
silence
turnOn()
idle
turnOff()
off
[t.getMode() = idle] makeCall(t)
busy2
turnOn, turnOff, makeCall, acceptCall, setSilence, sendSMS / e
busy2
getMode()
busy2
receiveSMS(msg, from)
busy2
endCall()
silence

Figure 4: `Terminal` class transition tree with PSP represented

Run	Test Run/Event Path					Expected Terminal State	Exception
	Level 1	Level 2	Level 3	Level 4	Level 5		
27	new	receiveSMS(msg, from)				off	Yes
28	new	makeCall(t)				off	Yes
29	new	sendSMS(msg, to)				off	Yes
30	new	acceptCall(t)				off	Yes
31	new	turnOff()				off	Yes
32	new	setSilence()				off	Yes
33	new	endCall()				off	Yes
34	new	turnOn()	turnOn()			idle	Yes
35	new	turnOn()	endCall()			idle	Yes
36	new	turnOn()	setSilence()	acceptCall(t)		silence	Yes
37	new	turnOn()	setSilence()	setSilence()		silence	Yes
38	new	turnOn()	setSilence()	endCall()		silence	Yes
39	new	turnOn()	[t.getMode() = idle] makeCall(t)	turnOn()		busy1	Yes
40	new	turnOn()	[t.getMode() = idle] makeCall(t)	turnOff()		busy1	Yes
41	new	turnOn()	[t.getMode() = idle] makeCall(t)	makeCall(t)		busy1	Yes
42	new	turnOn()	[t.getMode() = idle] makeCall(t)	acceptCall(t)		busy1	Yes
43	new	turnOn()	[t.getMode() = idle] makeCall(t)	setSilence()		busy1	Yes
44	new	turnOn()	[t.getMode() = idle] makeCall(t)	sendSMS(msg, to)		busy1	Yes
45	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	turnOn()	busy2	Yes
46	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	turnOff()	busy2	Yes
47	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	makeCall(t)	busy2	Yes
48	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	acceptCall(t)	busy2	Yes
49	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	setSilence()	busy2	Yes
50	new	turnOn()	setSilence()	[t.getMode() = idle] makeCall(t)	sendSMS(msg, to)	busy2	Yes

Table 23: Set of test cases able to detect possible sneak paths in the `Terminal` class

Description of the test cases

- In total we have 50 test cases to test the `Terminal` class and in the test cases where the `sendSMS` condition and `makeCall` condition is needed, it will use the test data defined above;
- Each row in the conformance test suit above represents a test case and by applying this test pattern we can test all possible transitions and states from the `Terminal` class;
- It's expected from test 1 to 7, 9 to 19 and 21 to 26 from the table to succeed in changing to another state and it's expected the remaining to throw an exception and remain in the same state.