



UNIVERSIDAD
POLITECNICA
DE VALENCIA



UNIVERSITÀ
DEGLI STUDI
DI UDINE

Test cases generation for functional tests of user interfaces

Master Thesis in Software Engineering,
Formal Methods and Information Systems.
(SEFMIS)

Università degli studi di Udine.
Universidad Politécnica de Valencia (UPV).

September 2012

Francisco Gramuntell Desco

Directors:

*Dr. Giorgio Brajnik
Dr. Emilio Insfran*

Test cases generation for functional tests of user interfaces.

*Test cases generation for
functional tests of user interfaces.*

© Francisco Gramuntell Desco.

Developed in Università degli Studi di Udine (Italia, Udine) and Universidad Politécnica de Valencia (UPV).
Printed in Valencia, Spain.

September 2012.

Test cases generation for functional tests of user interfaces.

*Grateful
to the directors and co-directors of this thesis
of the departments of computer science at the Universities of Udine (ITA) and Valencia (SPA),
as well as to the creator of W and Wp method,
who has kindly lent his help since the University of Sheffield (UK) to solve my doubts.*

Abstract

Applications software and Web applications are getting bigger and need more resources, thus the creation of these requires a long time for its development. Apart from the Engineering Requirements, different testing tools can help us to streamline and to guide the development of an application to a good end, so as to check it again seems to be complete through test sequences to ensure that a protocol implementation conforms to its specification. Automatic test data generation helps testers to validate software against user requirements more easily and verify whether the software is working properly following software requirements. It accounts more or less the 50% of software life cycle. The most thorough approach is to test all possible combinations with the objective of discover some error, but this is quite often impossible due to the large number of tests which far exceeds the time and resources available to execute them, therefore, the crucial part of software testing is to select the test data for testing software. Testers have to decide which data test they should use, and a heuristic technique is needed to solve this problem automatically to reduce the number of combinatorial tests while maintaining the fault-detection capability of combinatorial testing. Research in Software testing in recent years has focused on modifying existing methods, including the most important and where more work is based on, the W method, from which other approaches have been developed as HIS, Wp, Mp and UIOv.

This thesis has analyzed the behavior of existing techniques such as W and Wp method in our own statechart looking for limitations and improvements. As an alternative to existing methods, we have proposed an algorithm based on W and Wp method, own ideas, and collected ideas on different ways of working with statechart transitions. We apply the techniques of grey-box testing which combines black-box and white-box testing. Black-box testing is a software testing techniques in which functionality of the software under test (SUT) is tested without looking at the internal code structure (testers prepare test input and expected output) while white-box testing is a verification technique software that engineers can use to examine if their code works as expected (tests internal structures). The concept of grey-box testing is simple, is based in performing black box testing based on test cases performed by people who know the program inside.

Some of the improvement is that the method works incrementally to reduce the length of generated test sequence so our new method always starts from the same starting state of the given FSM. This overcomes the problem that an extra leading sequenced may have to be added in the case that the test sequence generated started from a state different from the starting state of the given FSM. One of the biggest problems that we found was falling into infinite loops when we apply our formula by the appearance of them in our statechart, which we have solved by adding a finite number of iterations to the algorithm so that it does not end in an infinite loop. Another improvement is that we have modified the initial statechart to identify each state separately, assuming that test cases can be identified and do not need to apply discrimination set for this. Our goal is to develop an algorithm that reduces significantly the length of the test sequences required for conformance testing while maintaining the same fault detection capability. To prove it, we will work on the same example, with various modifications, where we will apply different testing methods apart from that one we have developed.

Keywords: Software testing, test data generation, statecharts, specification-based testing, formal methods, finite-state machines (FSM), grey-black-white box testing, discrimination set.

Resumen

Tanto las aplicaciones software como las web se están haciendo cada vez más y más grandes necesitando así una mayor cantidad de recursos, por lo tanto la creación de estos requiere mucho tiempo para su desarrollo. Además de los requisitos de ingeniería, las diferentes herramientas de pruebas nos pueden ayudar a racionalizar y orientar el desarrollo de una aplicación a un buen final, con el objetivo de comprobar de nuevo si parece estar completa a través de secuencias de prueba para asegurarse de que la implementación del protocolo se ajusta a su especificación. La generación de datos a partir de tests automáticos ayuda a los testers a validar el software contra los requerimientos del usuario con mayor facilidad y verificar si el software está funcionando adecuadamente siguiendo los requisitos software. Este proceso representa más o menos el 50% del ciclo de vida del software. El enfoque más exhaustivo es poner a prueba todas las combinaciones posibles con el objetivo de descubrir algún error, pero esto con frecuencia es imposible debido a la gran cantidad de pruebas que supera con creces el tiempo y los recursos disponibles para ejecutarlos, por lo tanto, la parte crucial de las pruebas software consiste en seleccionar los datos que serán posteriormente analizados en dichas pruebas software. Los testers tienen que decidir qué datos de prueba deben utilizar, y que técnica heurística se necesita para resolver un problema dado automáticamente para reducir el número de pruebas combinatorias mientras se mantiene la capacidad de detección de fallos. La investigación en el campo de pruebas software en los últimos años se ha centrado en la modificación de métodos existentes, incluyendo entre ellos el más importante y en el que se ha basado más trabajo, el método W, a partir del cual se han desarrollado otros métodos como HIS, Wp, Mp y UIOv.

En esta tesis se ha analizado el comportamiento de las técnicas existentes, tanto del método W como del Wp en nuestro propio statechart en busca de las limitaciones y mejoras. Como una alternativa a los métodos existentes, hemos propuesto un algoritmo basado en los métodos W y Wp, ideas propias e ideas recogidas de las distintas formas de trabajar con transiciones de un statechart dado. Aplicamos las técnicas de pruebas de caja gris (*grey-box testing*) que combina las pruebas de caja negra y blanca (*black-box and white-box testing*). Las pruebas de caja negra son una técnica software de testing en las que la funcionalidad del software se pone a prueba sin tener en cuenta la estructura del código interno (los testers prepararan los datos de entrada y los resultados esperados), mientras que las pruebas de caja blanca son una técnica de verificación de software que los ingenieros pueden utilizar para examinar si su código funciona como se esperaba (pruebas de estructura internas). El concepto de las pruebas de caja gris es simple, se basa en la realización de pruebas de caja negra basadas en casos de prueba realizados por personas que conocen el código del programa.

Una de las mejoras que aporta nuestro método es que funciona de forma incremental reduciendo así la longitud de la secuencia de prueba generada por lo que este siempre empieza desde el mismo estado inicial de un FSM dado. Uno de los mayores problemas que encontramos es la aparición de bucles infinitos cuando aplicamos nuestra fórmula, que hemos resuelto mediante la adición de un número finito de iteraciones para el algoritmo de manera que no entre en un bucle infinito. Otra mejora es que se ha modificado el statechart inicial para identificar cada estado por separado, suponiendo que los casos de prueba pueden ser identificados y no es necesario aplicar la discriminación por conjuntos para ello. Nuestro objetivo es desarrollar un algoritmo que reduzca significativamente la duración de las secuencias de prueba necesarias para las pruebas de conformidad, manteniendo la misma capacidad de detección de fallos. Para demostrarlo, vamos a trabajar en el mismo ejemplo, con diversas modificaciones, donde se aplican diferentes métodos de prueba, aparte del que hemos desarrollado.

Palabras clave: Pruebas software, generación de casos de prueba, statecharts, pruebas basadas en la especificación, métodos formales, máquinas de estados finitos (FSM), pruebas de caja gris-negra-blanca, discriminación por conjuntos.

Resum

Tant les aplicacions software com les web s'estan fent cada vegada més i més grans necessitant així una major quantitat de recursos, per tant la creació d'aquests requereix molt de temps per al seu desenvolupament. A més dels requisits d'enginyeria, les diferents eines de proves ens poden ajudar a racionalitzar i orientar el desenvolupament d'una aplicació a un bona fi, amb l'objectiu de comprovar de nou si sembla estar completa a través de seqüències de prova per assegurar-se que la implementació del protocol s'ajusta a la seva especificació. La generació de dades a partir de tests automàtics ajuda als testers a validar el software contra els requeriments de l'usuari amb més facilitat i verificar si el software està funcionant adequadament seguint els requisits. Aquest procés representa més o menys el 50% del cicle de vida del software. L'enfocament més exhaustiu és posar a prova totes les combinacions possibles amb l'objectiu de descobrir algun error, però això a sovint és impossible a causa de la gran quantitat de proves que supera amb creixes el temps i els recursos disponibles per executar-los, per tant, la part crucial de les proves software consisteix a seleccionar les dades que seran posteriorment analitzades en aquestes proves. Els testers han de decidir quines dades de prova han d'utilitzar, i que tècnica heurística es necessita per resoldre un problema donat automàticament per reduir el nombre de proves combinatòries mentre es manté la capacitat de detecció d'errors. La investigació en el camp de proves software en els últims anys s'ha centrat en la modificació de mètodes existents, incloent-hi el més important i en el qual s'ha basat més treball, el mètode W, a partir del qual s'han desenvolupat altres mètodes com HIS, Wp, Mp i UIOv.

En aquesta tesi s'ha analitzat el comportament de les tècniques existents, tant del mètode W com del Wp en el nostre propi statechart a la recerca de les limitacions i millores. Com una alternativa als mètodes existents, hem proposat un algoritme basat en els mètodes W i Wp, idees pròpies i idees recollides de les diferents formes de treballar amb transicions d'un statechart donat. Apliquem les tècniques de proves de caixa gris (grey-box testing) que combina les proves de caixa negra i blanca (black-box and white-box testing). Les proves de caixa negra són una tècnica software de testing on la funcionalitat del software es posa a prova sense tenir en compte l'estructura del codi intern (els testers prepararan les dades d'entrada i els resultats esperats), mentre que les proves de caixa blanca són una tècnica de verificació de software que els enginyers poden utilitzar per examinar si el seu codi funciona com s'esperava (proves d'estructura internes). El concepte de les proves de caixa gris és simple, es basa en la realització de proves de caixa negra basades en casos de prova realitzats per persones que coneixen el codi del programa.

Una de les millores que aporta el nostre mètode és que funciona de manera incremental reduint així la longitud de la seqüència de prova generada de manera que aquest sempre comença des del mateix estat inicial d'un FSM donat. Un dels majors problemes que trobem és l'aparició de bucles infinits quan apliquem la nostra fórmula, que hem resolt mitjançant l'addició d'un nombre finit d'iteracions per l'algoritme de manera que no entri en un bucle infinit. Una altra millora és que s'ha modificat el statechart inicial per identificar cada estat per separat, suposant que els casos de prova poden ser identificats i no cal aplicar la discriminació per conjunts per a això. El nostre objectiu és desenvolupar un algoritme que redueixi significativament la durada de les seqüències de prova necessàries per a les proves de conformitat, mantenint la mateixa capacitat de detecció d'errors. Per demostrar-ho, treballarem amb el mateix exemple, amb diverses modificacions, on s'apliquen diferents mètodes de testing, a part del que hem desenvolupat.

Paraules clau: Proves software, generació de casos de prova, statecharts, proves basades en la especificació, mètodes formals, màquines de estats finits (FSM), proves de caixa gris-negra-blanca, discriminació per conjunts.

Test cases generation for functional tests of user interfaces.

Content

Index of Figures	18
Index of Tables	21
Index of Equations	23
Acronyms	25
Chapter 1.	
1. Introduction.	27
1.1 Motivation of the work	29
1.2 Goals	29
1.3 Research method	30
1.4 Document organization	31
1.5 Structure of the thesis	34
Chapter 2.	
2. Basic concepts.	38
2.1 Concept of testing	40
2.1.1 Test planning	41
2.2 Different ways to create test cases	42
2.2.1 Randomly	42
2.2.2 Generating usual paths	43
2.2.3 Generating all possible paths	43
2.3 Automated test concepts	44
2.3.1 Implantation of automated tests	44
2.4 Finite state machines (FSM)	45
2.5 Statechart diagram	46
2.6 Differences between FSM and statechart	46
Chapter 3.	
3. State of the Art.	50
3.1 Evolution of methodologies	52
3.1.1 First stage. Since the industrial revolution to 1930	53
3.1.2 Second stage. 1930-1949	54
3.1.3 Third stage. 1950-1979	54
3.1.4 Quarter stage. Decade of 80	54
3.1.5 Fifth stage. From 1990 to date	54
3.2 Basic concepts and tools to run performance tests	54
3.2.1 Performance test	54
3.2.2 Performance testing types	54
3.2.3 Performance testing by Ian Molyneaux	55
3.2.4 Object-Oriented Methodology for Life Cycle Completed	56
3.2.6 Types of performance tests	59
3.3 Tools to conduct performance tests	60
3.3.1 JMeter	60
3.3.2 Jcrawler	61
3.3.3 WAPT	61
3.3.4 Netsparker	61

3.3.5 OpenSTA	62
3.3.6 TestMaker	62
3.4 Comparative analysis	63

Chapter 4.

4. Literature Review.	66
4.1 Analysis of W and Wp methods.	68
4.1.1 Describing the statechart	68
4.1.2 Flattened statechart of the MAIN without hierarchy	69
4.1.3 State hierarchy	69
4.1.4 Configurations	70
4.1.5 Test generation for statecharts with W method	70
4.1.6 Applying the formulas for all the statechart	73
4.1.7 Final results for W-Method	74
4.1.8 Test case generation for statecharts with Wp method	74
4.1.9 Conclusions of W & Wp Methods	77
4.2 Satisfy All-Configurations-Transitions on Statecharts	77
4.3 A Bipartite Graph Approach	77

Chapter 5.

5. Adapt Bogdanov's theory (W) to state machines used to model UIs.	80
5.1 Describing our TV1 statechart	82
5.2 Solving the problem of loops	82
5.2.1 Create the first version of the flattened statechart of TV1	84
5.2.2 Positive, negative and redundant test cases	85
5.3 Apply W method to TV1 without state hierarchy and without concurrent region	87
5.3.1 Simplest model change transitions "+" and "-" for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.	88
5.3.2 Simplest model change transitions "+" and "-" for numbers only up direction between channels.	93
5.3.3 Simplest model with transitions "+" and "-" with a broken transition "-" between ch1 and ch4.	98
5.3.4 Simplest model with transitions "+" and "-".	103
5.3.5 Simplest model change transitions "+" and "-" for numbers (direction up and down between channels).	104
5.3.6 Simplest model with transitions "+" and "-" with a broken forward and a backward loop around ch1, ch2, ch3, ch4.	
5.4 Obtained results and conclusions of W Method.	107
5.5 Applying Wp-method to TV1 without concurrent region and without state hierarchy	109
5.5.1 Simplest model with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.	109
5.5.2 Simplest model change transitions "+" and "-" for numbers only up direction between channels.	157
5.5.3 Simplest model change transitions "+" and "-" for numbers only up direction between channels.	113
5.5.4 Simplest model with transitions "+" and "-".	123
5.6 Obtained results and conclusions of Wp Method.	124
5.7 Comparison of the results obtained with W and Wp methods	125

Chapter 6.	
<hr/>	
6. Definition and analysis of the new algorithm U-Method.	129
6.1 Creation of the new testing algorithm U-method	131
6.2 Adapt U-method to state machines used to model UIs	133
6.2.1 Apply U-method to TV1 original flattened model without customization.	134
6.2.2 Apply U-method to TV1 to model 3 of chapter 5	137
6.3 Obtained results and conclusions for the U-Method.	140
6.4 Comparison of the results obtained with W, Wp and U methods	141
Chapter 7.	
<hr/>	
7. Applying the methods to the statechart with Bugs.	145
7.1 Errors on Model 1 obtained with W Method.	147
7.1.1 Damage state Ch1.	147
7.1.2 Create new state Ch5.	149
7.1.3 Create a new transition “on”.	150
7.1.4 Create a new transition “x”.	151
7.1.5 Create a new transition “on[ch1]”.	152
7.1.6 Damage transition between states Ch2 and Ch1.	153
7.1.7 Change transition “off” by “standby”.	155
Chapter 8.	
<hr/>	
8. Conclusions.	158
8.1 Contributions	160
8.2 Alternatives to testing	161
8.3 Conclusions and future work	161
Bibliography	165
Annexes	168

Test cases generation for functional tests of user interfaces.

Index of figures

Figure 1.3.1 Research method.	30
Figure 1.4.1 Summary Schedule of tasks.	31
Figure 1.4.2 Calendar of open tasks.	34
Figure 2.1.1 Synonyms of test.	40
Figure 2.1.1.1 Software Testing Life Cycle (STLC).	41
Figure 2.2.1.1 Randomly test cases generation	42
Figure 2.2.2.1 Generating test cases by usual paths.	43
Figure 2.2.3.1. Generating all possible paths by algorithms.	43
Figure 2.2.3.2. Represent the User Interface and their equivalent statechart to apply the algorithms.	44
Figure 2.4.1 Representation of a simple finite state machine.	45
Figure 2.5.1 Example of simple statechart.	46
Figure 2.6.1 Finite state machine.	46
Figure 2.6.2 Statechart diagram.	47
Figure 3.1.1 Methodologies for software development.	52
Figure 3.1.2 Basic ideas of quality.	53
Figure 3.1.1.1 Example of automation in the vehicle industry	53
Figure 3.2.4.1 Quality Assurance.	57
Figure 3.2.4.2 Test Techniques.	58
Figure 3.2.5.1 Types of tests.	59
Figure 3.2.1 JMeter graph result.	60
Figure 3.3.1 WAPT Report graph.	61
Figure 3.3.4.1 Graphical interface of Netsparker.	62
Figure 3.3.6.1 Interface of the tool TestMaker.	62
Figure 4.1.1.1 The taperecorder statechart.	68
Figure 4.1.2.1 The flattened statechart of the MAIN state	69
Figure 4.1.3.1 The state tree of the tape recorder.	70
Figure 4.1.4.1 Types of configurations (compound transitions).	70
Figure 4.1.5.1 MAIN without state hierarchy	71
Figure 4.1.5.2 Calculating state hierarchy.	72
Figure 4.1.5.3 Concurrent part omitted.	72
Figure 4.1.5.4 State hierarchy part.	72
Figure 4.1.5.5 Calculating for concurrency	73
Figure 4.1.8.1 Calculating small w sets	75
Figure 4.1.8.2 Erroneous transition	75
Figure 5.1.1 TV1 Statechart.	82
Figure 5.2.1 Example of statement coverage.	84
Figure 5.2.1.1. Original statechart.	84
Figure 5.2.1.2. First version of Flattened statechart.	84
Figure 5.2.2.1 Example of a positive path test case.	85
Figure 5.2.2.2. Example of a negative path test case.	86
Figure 5.2.3. Example of a redundant path test case.	85
Figure 5.3.1 Original Statechart of TV1.	87
Figure 5.3.2 Flattened statechart.	87
Figure 5.3.1.1. Changed model 1.	88
Figure 5.3.1.2. Covered transitions.	91
Figure 5.3.1.3 Types of test cases model 1.	93
Figure 5.3.2.1 Changed model 2.	93
Figure 5.3.2.2. Types of test cases on model 2.	98

Figure 5.3.3.1. Changed model 3.	99
Figure 5.4.2.2. Types of test cases on model 3.	103
Figure 5.3.4.1 Changed model 4.	103
Figure 5.3.5.1. Changed Model 5.	104
Figure 5.3.6.1. Changed Model 6.	106
Figure 5.4.1 Test cases for the different models of TV1 statechart.	107
Figure 5.5.1 Original Statechart of TV1.	108
Figure 5.5.2 Flattened statechart.	109
Figure 5.5.1.1 Model 1.	110
Figure 5.5.1.2. Types of test cases on model 1.	113
Figure 5.5.2.1 Customized model 2.	113
Figure 5.5.2.2. Types of test cases on model 2.	117
Figure 5.5.3.1 Customized model 3.	117
Figure 5.5.3.2. Types of test cases on model 3.	122
Figure 5.5.4.1 Customized model 4.	123
Figure 5.5.4.2 Not supported models 5 and 6.	124
Figure 5.6.1 Test cases for the different models of TV1 statechart with Wp method.	124
Figure 5.7.1 Simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4	125
Figure 5.7.2 Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4	126
Figure 5.7.3 Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4	127
Figure 5.7.4 Statechart and Diagram of the best model after testing with W & Wp methods.	127
Figure 6.1.1. Content of the mail to create the algorithm.	131
Figure 6.1.1. TV1 statechart to extract new variables.	132
Figure 6.2.1 Original Statechart of TV1.	133
Figure 6.2.2 Flattened statechart.	133
Figure 6.2.1.1 Original flattened statechart.	134
Figure 6.2.1.2. Obtained test cases for the original flattened statechart of TV1.	137
Figure 6.2.2.1 Simplest model with transitions "+" and "-" with a broken transition "-" between ch1 and ch4.	137
Figure 6.2.2.2 Obtained test cases for the simplest model with transitions "+" and "-" with a broken transition "-" between ch1 and ch4.	140
Figure 6.3.1 Test cases for the different models of TV1 statechart with Wp method.	141
Figure 6.4.1. Obtained results for the original flattened statechart.	142
Figure 6.4.2. Obtained results for the simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4. (Model 3 in chapter 5)	143
Figure 6.4.3 Statechart and Diagram of the best model after testing with W & Wp methods.	143
Figure 7.1.1. Model 1 of chapter 5 to apply bugs.	147
Figure 7.1.1.1 Bug 1.	147
Figure 7.1.1.2 Detection of the error.	148
Figure 7.1.1.3 Part of code with Dreamweaver.	148
Figure 7.1.2.1 Bug 2.	149
Figure 7.1.2.2 Validation with Selenium.	150
Figure 7.1.3.1 Bug 3.	150
Figure 7.1.3.2 Detection of the error.	151

Figure 7.1.3.3 Part of code with Dreamweaver.	151
Figure 7.1.4.1 Bug 4.	151
Figure 7.1.5.1 Bug 5.	152
Figure 7.1.5.2 Detection of the error with Selenium.	152
Figure 7.1.5.3 Part of code with Dreamweaver.	153
Figure 7.1.6.1 Bug 6.	153
Figure 7.1.6.2 Detection of the error with Selenium.	154
Figure 7.1.6.3 Part of code in Dreamweaver.	154
Figure 7.1.7.1 Bug 7.	155
Figure 7.1.7.2 Detection of the error with Selenium.	155
Figure 7.1.7.3 Part of code with Dreamweaver.	156
Figure 8.3.1. Divide the statechart in two parts.	162

Index of tables

Table 1.2.1 Typical Software Quality Factors.	30
Table 2.1.1 Verification and Validation: Definition, Differences, Details.	41
Table 2.1.2.1 Activities, deliverables and Necessities of each phase of STLC.	42
Table 3.4.1 Comparative Analysis of Commercial Applications.	64
Table 5.3.1.1. Transition matrix A.	87
Table 5.3.1.2. Adjacency matrix B.	88
Table 5.3.1.3. Adjacency matrix $B*B$.	89
Table 5.3.2.1 Transition matrix A.	94
Table 5.3.2.2. Adjacency matrix B.	94
Table 5.3.2.3. Adjacency matrix $B*B$.	94
Table 5.3.3.1 Transition matrix A.	99
Table 5.3.3.2. Adjacency matrix B.	99
Table 5.3.3.3. Adjacency matrix $B*B$.	100
Table 5.3.4.1 Transition matrix A.	104
Table 5.3.5.1. Transition matrix A.	105
Table 5.3.6.1. Transition matrix A.	106
Table 5.5.1.1 Test cases of phase 2.	111
Table 5.5.2.1 Test cases of phase 2.	116
Table 5.5.3.1 Test cases of phase 2.	121
Table 6.1.1 Definition of new variables for U-Method.	132

Index of equations

Equation 4.1.5.1 Non hierarchical.	72
Equation 4.1.7.1. Results for the W-Method	74
Equation 4.1.8.1 Wp method.	74
Equation 4.1.8.1 Phase 2 of Wp-Method.	75
Equation 5.3.1.1 W Method.	89
Equation 5.3.2.1 W method.	95
Equation 5.3.3.1 W method.	100
Equation 5.3.4.1. W Method.	104
Equation 5.3.5.1. W Method.	105
Equation 5.3.6.1. W Method.	106
Equation 5.5.1.1 Set of test cases for the first phase of Wp method.	110
Equation 5.5.1.2 Transitions that will be explored in phase 2.	110
Equation 5.5.1.3 Developed Transitions that will be explored in phase 2.	110
Equation 5.5.1.4 Test cases of phase 2.	110
Equation 5.5.1.5 Test cases of phase 2.	110
Equation 5.5.1.6 Expected output of every test case.	111
Equation 5.5.2.1 Set of test cases for the first phase of Wp method.	114
Equation 5.5.2.2 Transitions that will be explored in phase 2.	115
Equation 5.5.2.3 Developed Transitions that will be explored in phase 2.	115
Equation 5.5.2.4 Test cases of phase 2.	115
Equation 5.5.2.5 Test cases of phase 2.	115
Equation 5.5.2.6 Expected output of every test case.	116
Equation 5.5.3.1 Set of test cases for the first phase of Wp method.	119
Equation 5.5.3.2 Transitions that will be explored in phase 2.	119
Equation 5.5.3.3 Developed Transitions that will be explored in phase 2.	119
Equation 5.5.3.4 Test cases of phase 2.	120
Equation 5.5.3.5 Test cases of phase 2.	120
Equation 5.5.2.6 Expected output of every test case.	120
Equation 6.1.1. U-Method	131
Equation 6.1.2. Simplified equation of U-Method.	131

Acronyms

QA	Quality assurance.
RE	Requirements engineering.
SRS	Software requirements specification.
FSM	Finite state machines.
SUT	System under test.
MDE	Model Driven Engineering.
SQC	Statistical Quality Control
CQC	Total Quality Control.
FLOOT	Full Life-Cycle Object-Oriented Testing.
SLA	Service Level Agreement.
CAST	Computer Aided Software Testing.
QoS	Optimal levels of Service Quality.
CVU	Concurrent Virtual Users.
MRT	Maximum Response Time.
STLC	Software Testing Life Cycle.
XM	X-machine.
DNF	Disjunctive Normal Form.
ATSP	Asymmetric Travelling Salesman Problem.
IOTS	Input/Output Transition System.
ATSP	Asymmetric Travelling Salesman Problem.
IUT	Implementation Under Test.

Chapter 1

Introduction

This chapter focuses on introducing the main motivations that have carried to the completion of this Master's thesis and its objectives. The chapter is structured in four points.

First point concerns the motivation that has led us to choose this work. Secondly speaks of work goals. Thirdly mentioned the different algorithms which work on the thesis and the final point talks about the organization and dates of the distribution of the thesis.

1.1 Motivation of the work

One of the biggest reasons that led us to investigate about generating test cases from user interfaces was that it is an area in currently booming and is increasingly demanded by different companies since these are beginning to integrate testing into their production lines. Therefore is interesting to research in this area since it is increasingly demanded by companies and this knowledge is interesting for get more possibilities to find a job the day of tomorrow.

Another reason that made us to choose this line of research is that we found several weaknesses in generating test cases with existing methods, so that we have proposed solve them and even improve the number of test cases that these methods get developing a new testing method about which we will discuss in Chapter 6 of this work.

We need to differentiate the terms of test cases and testing. Test cases are every one of the possible actions that one user can execute on a user interface and there are several ways to calculate of which we'll talk in chapter two of this thesis. On the other hand the process of software testing is the activity that verify or not if an application runs as his specification indicates, in other words, software testing is a process in the life-cycle of a software project that verifies that the product or service meets quality expectations and validates that software meets the requirements specification (SRS) identifying these as the test cases. In the process of software testing are involved "*test cases*" as inputs for the testing tool, "*generated outputs*" by the test cases and "*expected outputs*" determined by the analyst. If the "*generated outputs*" coincide with the "*expected outputs*" we can confirm that don't exist errors in the implementation. We can do this kind of work by Testing tools as Selenim. The purpose of testing can be quality assurance, verification and validation, or reliability estimation.

Today there are several testing algorithms (HIS, W, Wp, Mp, UIOv ...), we analyze the most important looking for strengths and weaknesses of each in the case study of chapter 5 with the existing methods W & Wp developed in [7] looking for limitations and then in chapter 6 apply the method that we have developed showing that eliminates the limitations of previous methods and improves the results obtained.

With this process of testing what is being sought is to create an algorithm capable of analyzing any type of application before being exposed to the end user, ensuring that complies with all software requirements that can be sell a safe and quality software.

1.2 Goals

This work aims to analyze different methods of testing on the same statechart, so as to obtain conclusions about the quality attributes. Based on these conclusions obtained about the existing methods discussed we have decided to develop a new method by solving these deficiencies and improving the obtained results. We can say that we are in the final stage of the life cycle of a software product line, since only take out the activities of testing once the software is ready, or seems to be, for commercialization or market exhibition. The principal objectives of our research are to "*improve quality*" and "*for Verification & Validation (V&V)*".

Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works

under certain situations, or it does not work. We can also compare the quality among different products under the same specification, based on results from the same test. We cannot test quality directly, but we can test related factors to make quality visible. Quality has three sets of factors functionality, engineering, and adaptability. These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. Table 1.2.1 illustrates some of the most frequently cited quality considerations.

Functionality(exterior quality)	Engineering (interior quality)	Adaptability (future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Table 1.2.1 Typical Software Quality Factors

Thus it is clear the purpose of testing of applications as the process of execution of a program with the intention of verifying the correctness of the requirements, identify differences between actual and expected behavior, measure quality, provide confidence and errors.

1.3 Research method

The first thing that has been done before starting to write this thesis is to clarify the points that were analyzed. From the information that has been discussed previously to the conclusions that were obtained after analyzing the different testing methods shown in the case study outlined in section 7. The research method is shown in Figure 1.3.1.

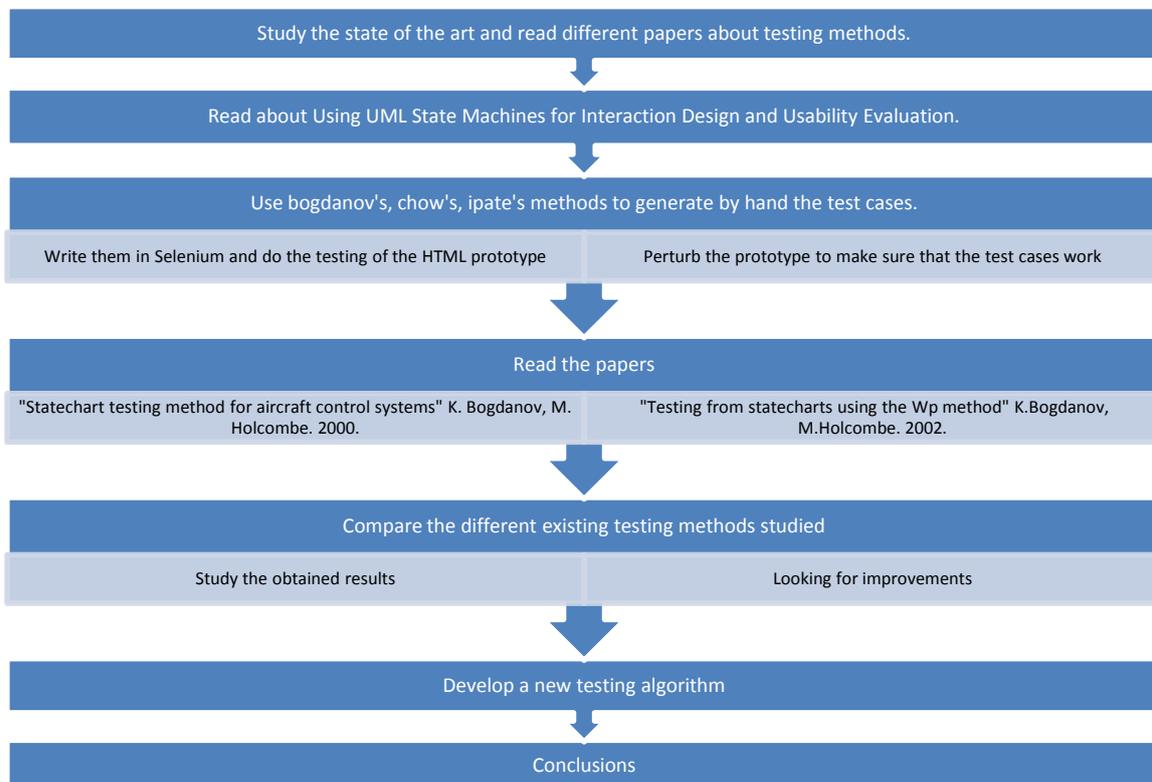


Figure 1.3.1 Research method.

For the realization of this thesis we have read several documents about using UML state machines for interaction design and usability evaluation before focusing this thesis on the comparison and analysis of different methods of testing and creating a new one. After that, we have drawn conclusions from them working on a statechart created by us previously and drawing conclusions on how to act so each of the different methods on our case study.

1.4 Document organization

Below shows the planning of the tasks that make this work. Additionally show the duration in weeks of each task and the start and end dates.

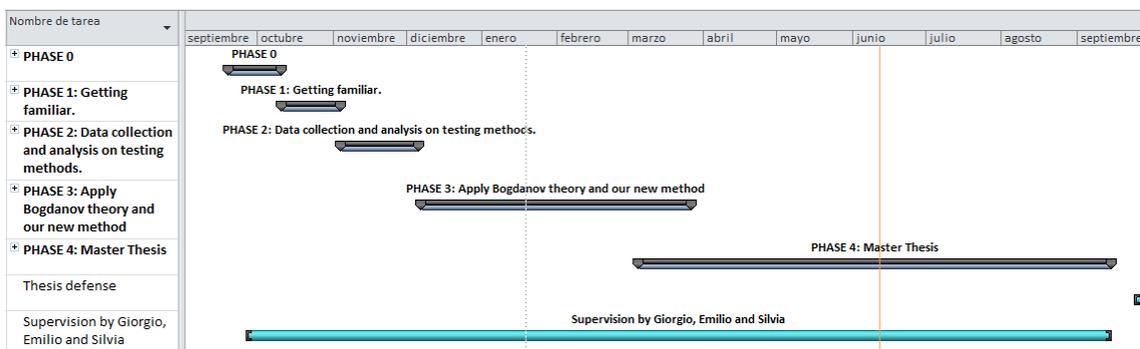
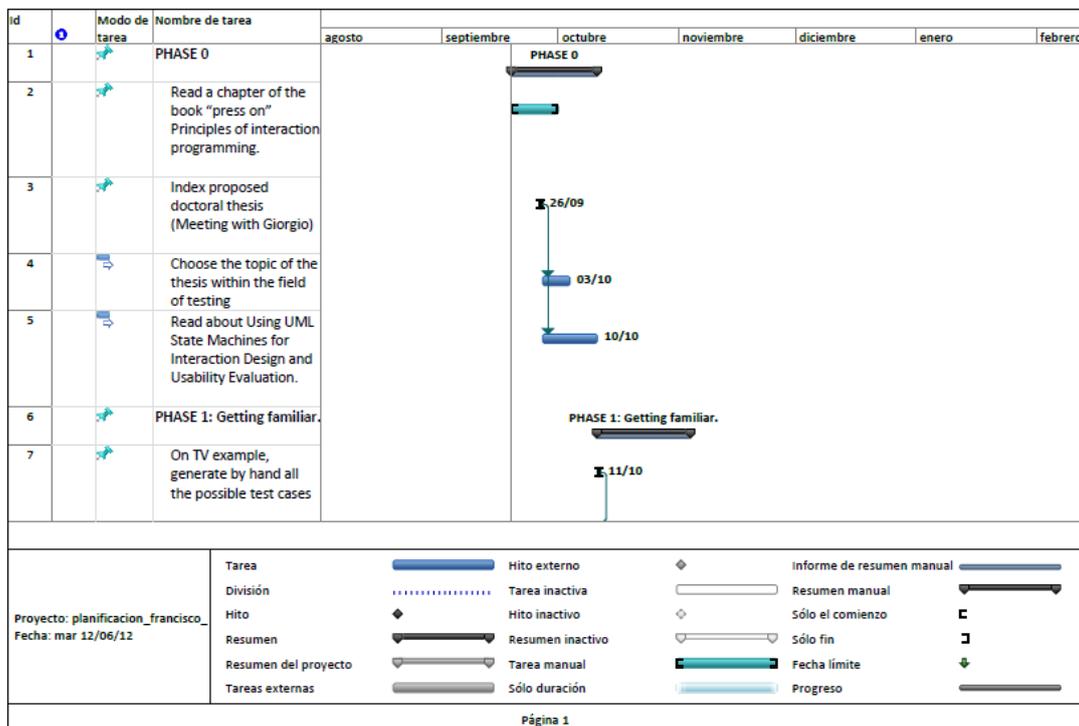
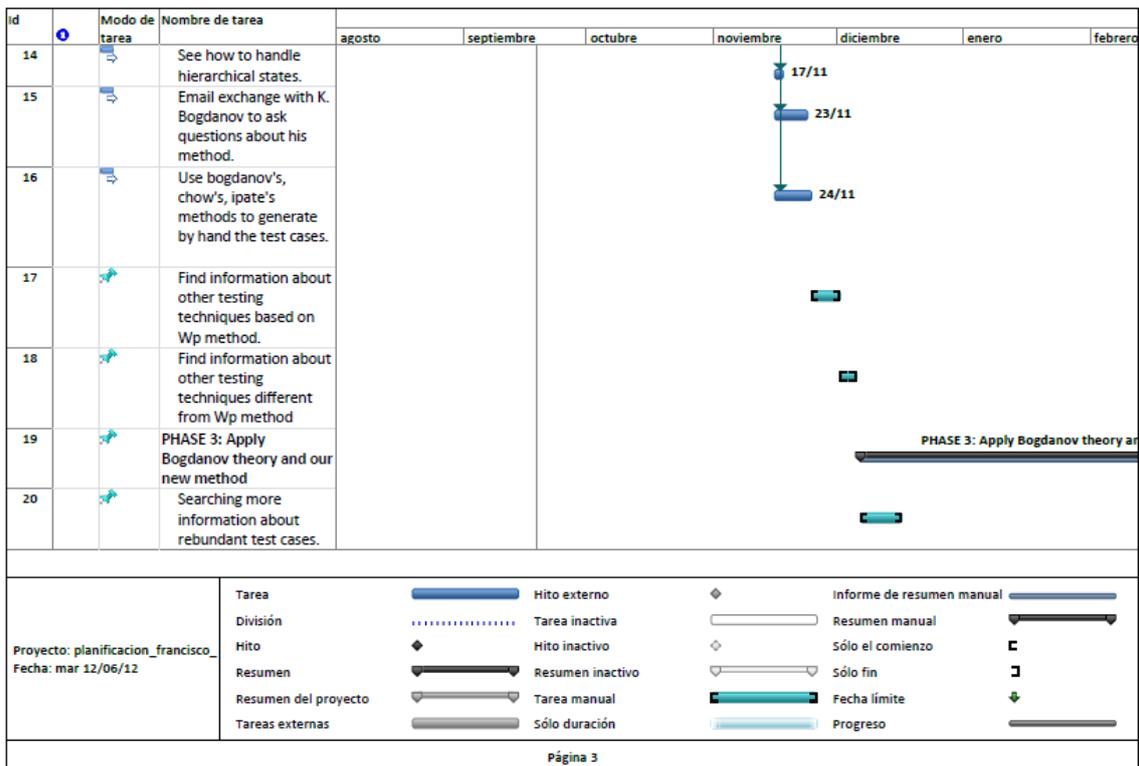
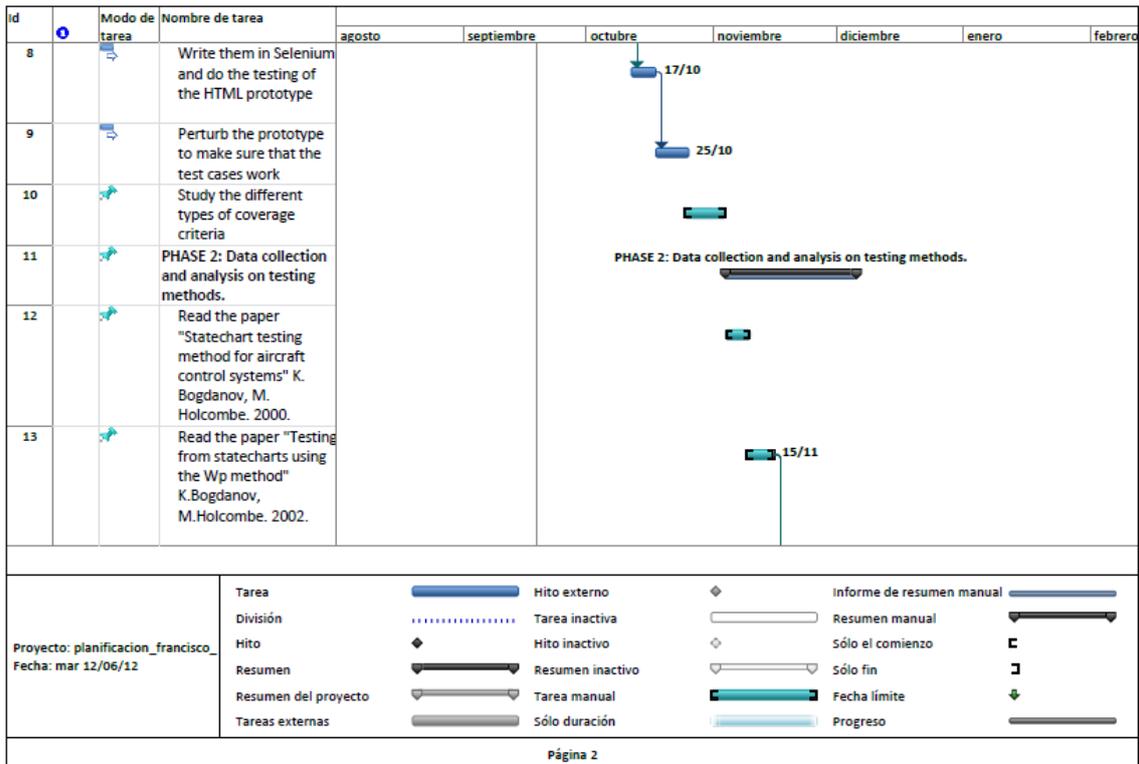
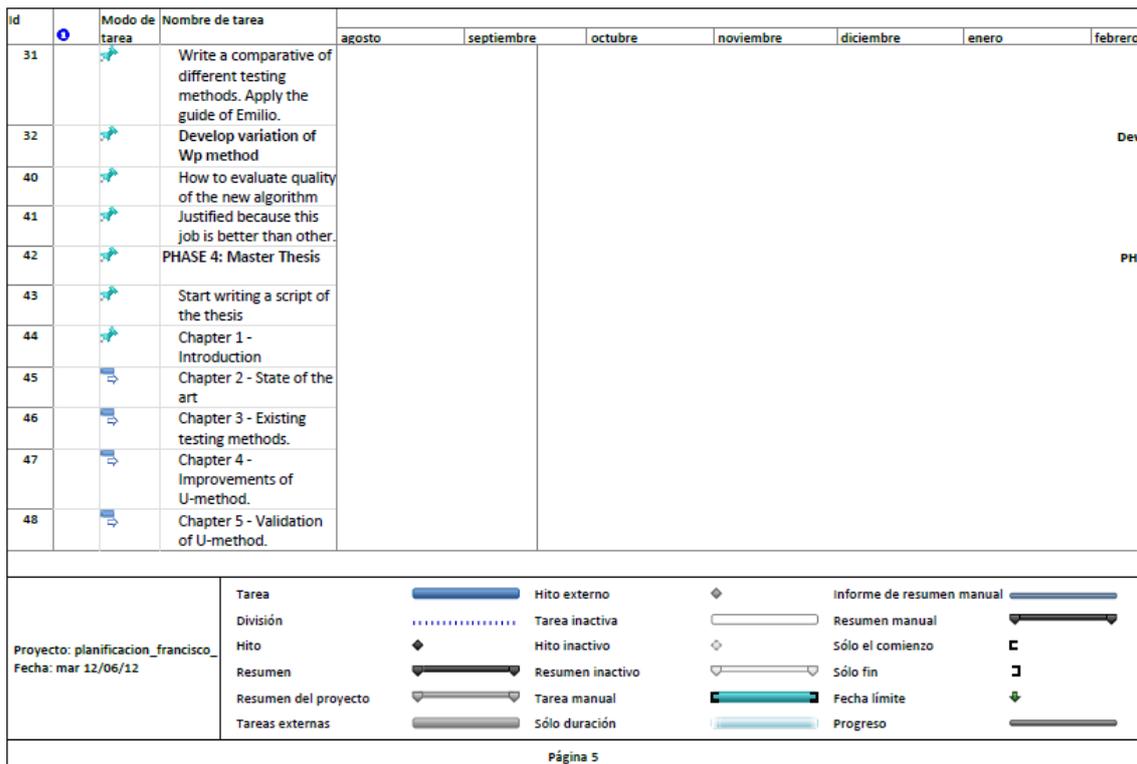
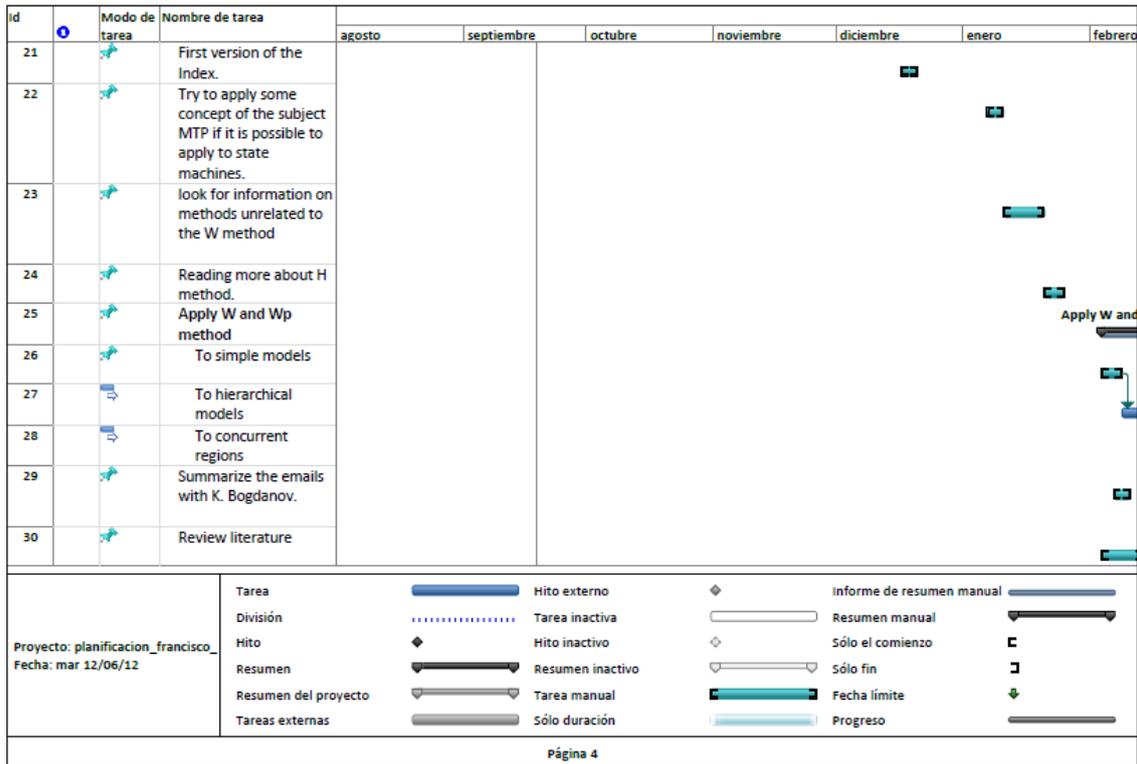


Figure 1.4.1 Summary Schedule of tasks.







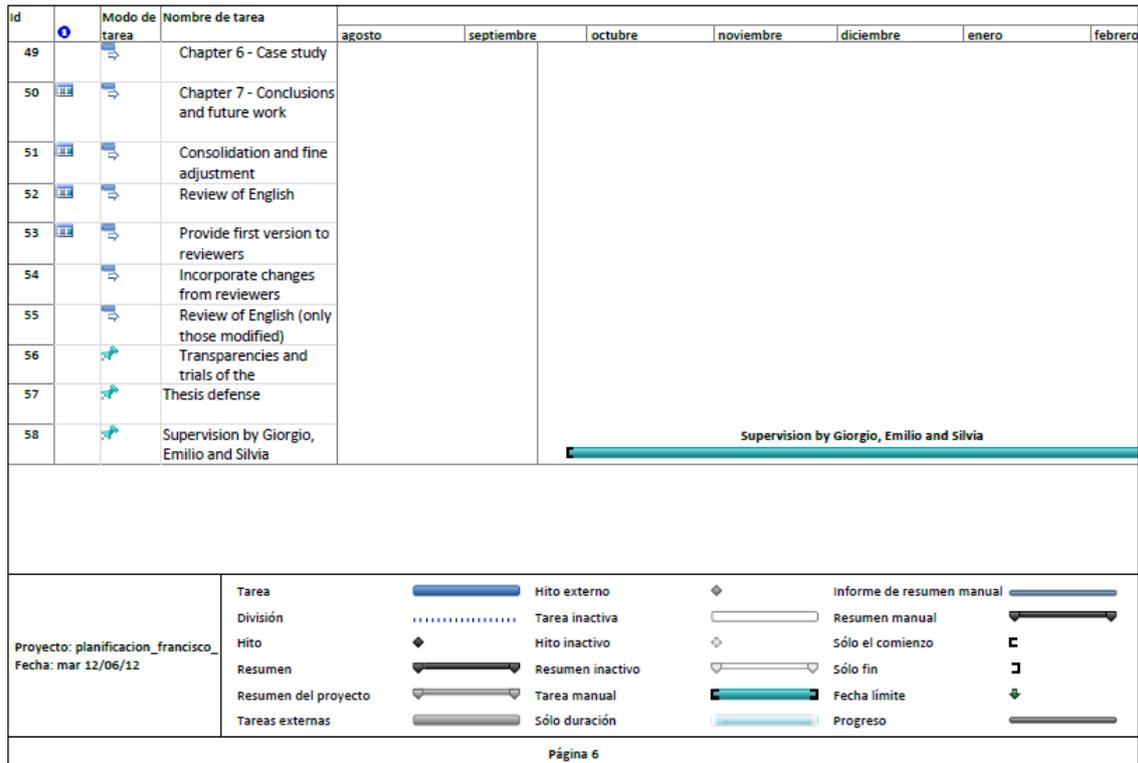


Figure 1.4.2. Calendar of open tasks.

1.5 Structure of the thesis

The memory of the thesis is organized as follows:

Chapter 2. Previous concepts.

- This chapter will introduce the concepts necessary to understand the context in which this proposal is developed. It will be necessary to define general concepts of testing and other concepts as “Software Testing Life Cycle (STLC)” developed in [39]. After we will discuss about the three ways that there are to generate test cases. Randomly, generating usual paths and generating all possible paths, introducing the concept of automated testing. And finally we will talk about the differences between “Finite State Machines” comprehensively explained in [45] and Statecharts also comprehensively explained in [2] by “D. Harel and A. Naamad”, which are important because in the case of study of the chapters 5 and 6 we work with Statecharts.

Chapter 3. State of the Art.

- In the first part of this chapter we will talk about the evolution of methodologies for software development and the historical stages of quality developed in [46] by “Juan Oliver”. In the second part describes basic concepts of testing, more explicit methodologies, and positions with respect to performance testing proposed by “Ian Molyneaux” in [27]. and mentioned various tools to run performance tests. Finally, there is a comparative table of the most used free tools in the market. With all this demonstrates the importance of software testing in the development of such projects

and how they affect the achievement of software with features more in line with new working methods applied to the development of software products.

Chapter 4. Literature Review.

- In the first part of this chapter, at section 4.1 we will analyze the paper “*Testing from statecharts using the Wp method*” [7] (which includes the W method (section 4.1.8) created by Kirill Bogdanov with whom we maintained contact through several emails to clarify some of the more complicated points of his method in their articles published and after in section 4.1.11 we will analyze the Wp method, which is an improvement of the said W created by the same author. At the end of the chapter we will discuss other methods which do not have relationship with W and Wp finding alternative ways for creating testing algorithms. In the following sections of this chapter are discussed the papers of whom principal authors are “Siamak Haschemi” [32] (section 4.2) and “Jun Wang” [33] (section 4.3).

Chapter 5. Adapt Bogdanov's theory (W) to state machines used to model UIs.

- In this chapter we have adapted “Bogdanov's theory (W and Wp)” developed in [4] and [7] respectively to state machines used to model UIs, without having to care about regions and loops. In section 5.1 shows the statechart that we work from now (TV1 Statechart). Below in 5.2 discusses the problems associated with loops and how to avoid them support us by articles published by “Beizer” in [21] and [22] explaining how we have created the flattened statechart to implement W and Wp methods with which we will work. Working with the algorithms in section 5.3 we focus on applying W method in the different versions of our statechart while in section 5.4 we show a graphic with the obtained results for everyone of the variations of the statechart. The same we will do in section 5.5 and 5.6 respectively with Wp method. Finally at section 5.7 of this chapter we will check the results obtained between W and Wp methods comparing the number of test cases in each of the tested models of our TV1 statechart.

Chapter 6. Definition and Analysis of the new algorithm U-Method.

- In this chapter we will explain the steps that we followed for the creation of a new testing algorithm, which we have called U-Method. The following sections of this chapter (6.2.1 and 6.2.2) shall apply this new method on TV1 statechart as was done in Chapter 5 with “W and Wp methods” from [4] and [7]. Next in section 6.3 we will make a comparison between the number of test cases and the relevance of them between the different versions of the statechart for the U-method and finally in 6.4 we will do the same but for the different methods with which we have tested our case of study in the thesis to check which is the best of all for this statechart.

Chapter 7. Bugs.

- In this chapter we will modify the customization 1 of the statechart used in chapter 5 to compare that the positive test cases obtained with W-method developed in [4] are useful and can be detect the different bugs. There are seven types of bugs, since eliminate the transition 1 since channel 2 to channel 1 to create a new channel 5 or create a new transition "x" to see the behavior of the algorithm amongst others. Every

one of them are checked and has sought in each case the test case that identifies the created bug.

Chapter 8. Conclusions.

- The conclusions of the study are shown in this chapter. Are remembered proposed at the beginning objectives of the work and checks whether there is compliance for the different algorithms studied. We discuss possible alternatives to the testing of applications for those who are not yet convinced the idea of issue this type of techniques. Finally, current research and future work are exposed.

Test cases generation for functional tests of user interfaces.

Chapter 2

Basic concepts

This chapter will introduce the concepts necessary to understand the context in which this proposal is developed. It will be necessary to define general concepts of testing and other concepts as “Software Testing Life Cycle (STLC)” developed in [39].

After we will discuss about the three ways that there are to generate test cases. Randomly, generating usual paths and generating all possible paths, introducing the concept of automated testing.

And finally we will talk about the differences between “Finite State Machines” comprehensively explained in [45] and Statecharts also comprehensively explained in [2] by “D. Harel and A. Naamad”, which are important because in the case of study of the chapters 5 and 6 we work with Statecharts.

Test cases generation for functional tests of user interfaces.

2.1 Concept of testing

Testing is the process to show that an application has no errors and does what it should do. It provides quality throughout the process, decreased costs and risk reduction. About 85% of the defects occur when an application start of the stage of development. It can also be used to generate communication designed to alter consumer attitudes toward existing products. First to start to analyze the different types of testing, we need to clarify some basic concepts about testing.

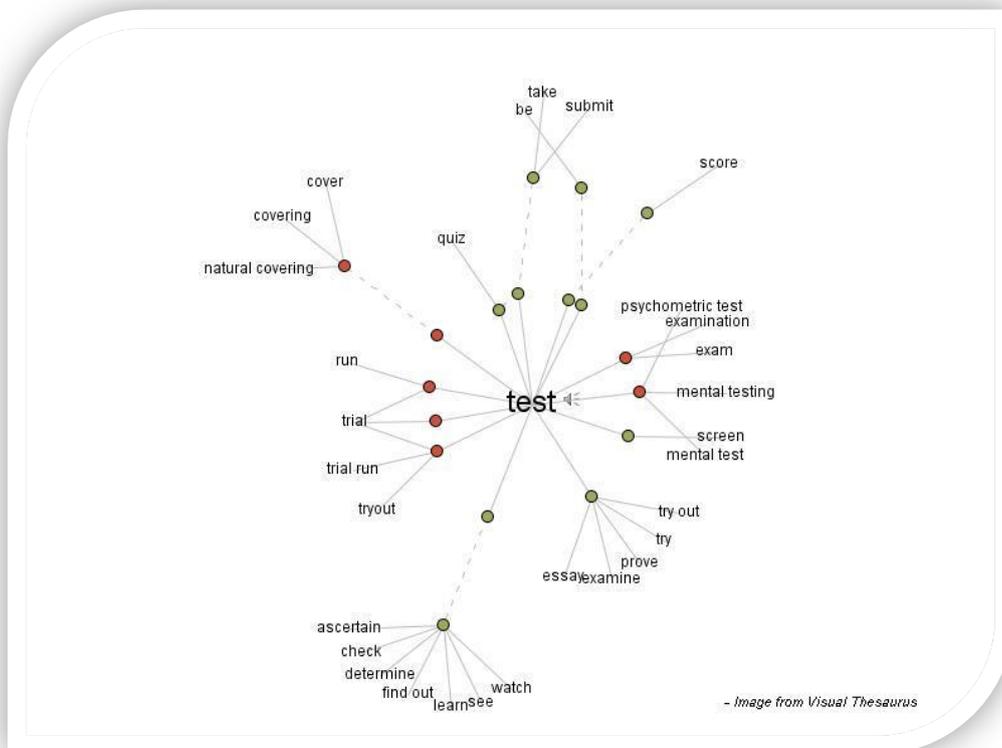


Figure 2.1.1 Synonyms of test.

Software testing is one of the “verification and validation,” or V&V, software practices. Some other V&V practices, such as inspections and pair programming, will be discussed in [14] edited by “J.D. Meier and Scott Barber”. Verification activities include testing and reviews. Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. At the end of development validation activities are used to evaluate whether the features that have been built into the software satisfy the customer requirements and are traceable to customer requirements. “Boehm” in [15] has informally defined verification and validation in the table below:

Criteria	Verification	Validation
Definition	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.
Objective	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs, and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.
Question	Are we building the product right?	Are we building the right product?
Evaluation Items	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.
Activities	<ul style="list-style-type: none"> • Reviews • Walkthroughs • Inspections 	<ul style="list-style-type: none"> • Testing

Table 2.1.1 Verification and Validation: Definition, Differences, Details.

2.1.1 Test planning

Test planning as mentioned in [38] involves scheduling and estimating the system testing process, establishing process standards and describing the tests that should be carried out. A test plan is a document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency plans as described in [14]. An important component of the test plan is the individual test cases. A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. So the idea is to propose a test plan as soon as possible in the development cycle as we can see in fig. 2.1.1.1 from when things are generally still going pretty and not when things start to go wrong as we can see in table 2.1.1.1 which comes from "Software Testing Life Cycle (STLC)" commented in [39], thus preventing alarm states in the development process.



Figure 2.1.1.1 Software Testing Life Cycle (STLC).

Phase	Activity	Deliverables	Necessity
Requirements/Design Review	You review the software requirements/design	<ul style="list-style-type: none"> • Review • Defect • Reports 	Curiosity
Test Planning	Once you have gathered a general idea of what needs to be tested, you 'plan' for the tests.	<ul style="list-style-type: none"> • Test Plan • Test Estimation • Test Schedule 	Farsightedness
Test Designing	You design/detail your tests on the basis of detailed requirements/design of the software	<ul style="list-style-type: none"> • Test Cases / Test Scripts /Test Data • Requirements Traceability Matrix 	Creativity
Test Environment Setup	You setup the test environment with the goal of replicating the end-users' environment.	<ul style="list-style-type: none"> • Test Environment 	Rich company
Test Execution	You execute your Test Cases/Scripts in the Test Environment to see whether they pass.	<ul style="list-style-type: none"> • Test Results (Incremental) • Defect Reports 	Patience
Test Reporting	You prepare various reports for various stakeholders.	<ul style="list-style-type: none"> • Test Results (Final) • Test Metrics • Test Closure Report 	Diplomacy

Table 2.1.2.1 Activities, deliverables and Necessities of each phase of STLC.

2.2 Different ways to generate test cases

We are going to talk about the three ways that there are to generate test cases. Randomly, generating usual paths and generating all possible paths.

2.2.1 Randomly

This type of generation of test cases is generated daily on the web that is, it would be a common user browsing a web user interface and finding an error by chance. We could say that every web user would be a “randomly generated test case” that can find a bug in the implementation of some interface that are running.



Figure 2.2.1.1 Randomly test cases generation.

2.2.2 Generating usual paths

This kind of test cases will be obtained through tools which capture all the actions that the user executes on a application generating scripts which would be the test cases for a certain tool. For example if we capture all the possible user actions commented previously in point 2.2.1 by a tool, this tool internally converts each one of the captured user actions in a test case. With this process we will obtain a set of test cases representing the usual paths in certain application. We are talking about usual paths because we are working with common user actions on an application, if we collect the different actions of a thousand users, usually execute repetitive and basic actions.

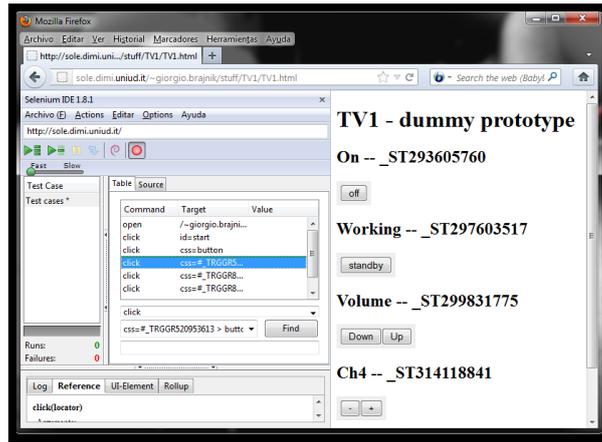


Figure 2.2.2.1 Generating test cases by usual paths.

As well as we have talked about capturing the common user actions, by a tool it could capture the user actions of a tester, who will be patient, observant, speculative, creative, innovative, open-minded, resourceful and skillful. This is a laborious activity that requires the tester to possess this set of qualities.

The generated test cases this time don't will be only the usual paths, because now is a tester not a common user who is executing actions over the application, but it is impossible to confirm that a tester will explore all the possible paths of one application. It is possible if the application is small and doesn't have a lot of actions, but if we are talking about a commercial application with a lot of states and transitions, it is impossible to run all of them by a tester. Manual testing can be replaced by test automation covering all the possible paths in an application and we are going to see it in next step 2.2.3.

2.2.3 Generating all possible paths

We generate this type of test cases by algorithms defining abstract user representations on an interface through statecharts.

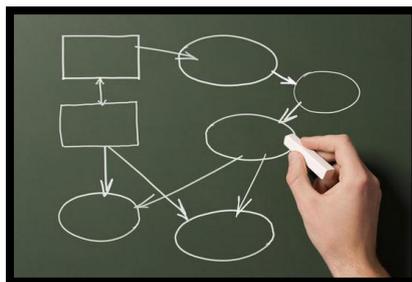


Figure 2.2.3.1. Generating all possible paths by algorithms.

To apply the algorithms the first thing that we need to do is to convert the application that we want test in a model with which we can apply the formula of the commented algorithm. For example for a user interface, we can generate an statechart that represent the same functionality in terms of states, transitions and condition transitions. We can see an example in Figure 2.2.3.2 in which we present an user interface and the equivalent statechart in terms of functionality.

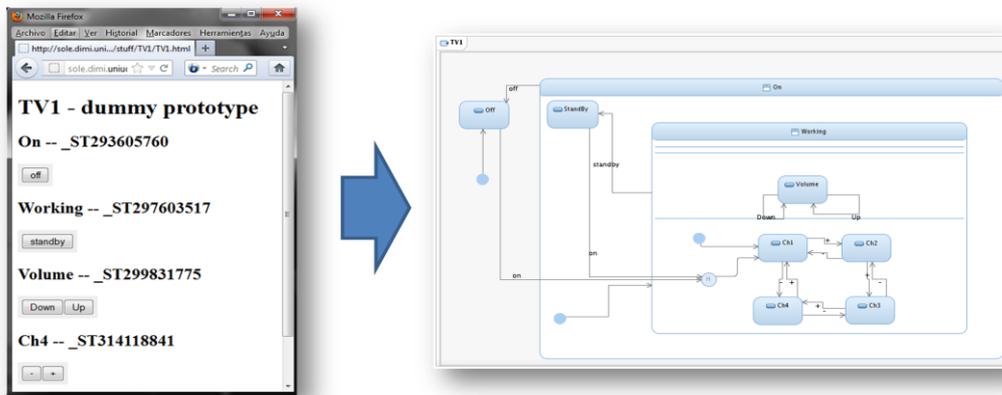


Figure 2.2.3.2. Represent the User Interface and their equivalent statechart to apply the algorithms.

Now we can apply several methods of automated generation test cases (as for example “W & Wp Methods” developed in [7] and commented in chapter 5 or our developed U-Method commented in chapter 6 of this thesis) on the generated statechart. If a method accept the statechart, we can assure that the obtained test cases cover all the possible paths in the User Interface as we will see in chapters 6 and 7 with the case study.

2.3 Automated test concepts

As mentioned in the previous section 2.2.3 as an introduction to the test automation, this process is the use of software to control;

- The execution of tests,
- The comparison of actual outcomes to predicted outcomes,
- The setting up of test preconditions,
- and other test control and test reporting functions.

2.3.1 Implantation of automated tests

As we have commented in chapter 1, there are some companies that are recognizing the importance of automating the work of testers and including the auto-test as part of the regular build process. The results of the automatic test are seen as a measure of the current quality of the software.

Here are some of the advantages, collected from “Automated Test Concepts” in [40], of having automated test scripts which can be run after each new build of the application:

- *Low Running Cost:* running an automated test script before each release of a new version, patch or bug fix is a lot cheaper than a manual test.

- *Better Quality*: especially for individual developers and small companies who would not employ a tester and will perform all testing themselves.
- *Consistency*: the test script will perform the same checks every time it is run. A manual test will be affected by human error and it will tend to skip certain areas believed to be stable.
- *Speed*: a script will execute many times faster than a manual test, giving us a full report on the quality of your product in a few minutes.
- *Formal*: A code coverage tool can tell us how much code is tested. The test scripts can then tell us if our test runs fine. The result is the exact percentage of the code which is guaranteed to work fine.
- *Compactness*: we can perform a full compatibility check by simply copying the application together with the test scripts on all the platforms where you believe it should work. It can give us the confirmation that all functionality works indeed as expected.

“Automated Test Concepts” in [40] are not meant to completely replace manual testing because they cannot be used on small components during development process. For more information about automated test concepts we can consult the point “Types of test” inside the sections of [40] of our bibliography.

2.4 Finite state machines (FSM)

A finite-state machine (FSM) is a mathematical model used to design computer programs and digital logic circuits. “Finite-state machine” in [44] is conceived as an abstract machine that can be in one of a finite number of states.

- The machine is in only one state at a time.
- The state it is in at any given time is called the current state.
- It can change from one state to another when initiated by a triggering event or condition, this is called a transition.
- A particular FSM is defined by a list of the possible transition states from each current state, and the triggering condition for each transition.

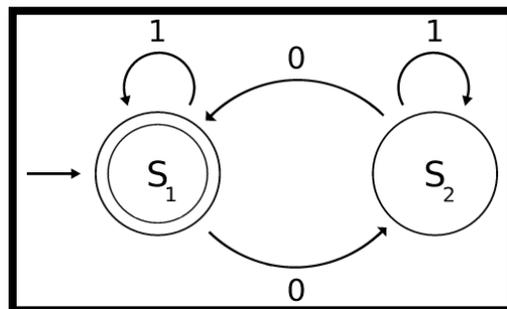


Figure 2.4.1 Representation of a simple finite state machine.

“Finite-state machines” commented in [45] can model a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems, and in linguistics they can be used to describe the grammars of natural languages.

2.5 Statechart diagram

The “statechart diagram” in [42] were invented by “Dr. Harel” in [1], and are sometimes called Harel Statecharts. He defined a pretty broad extension to typical state machines, with the goal of making state machines more useful for actual work with complicated systems. A variant of Statecharts are build into Matlab now, as “*stateflow*”, which is an extension of simulink.

Statechart diagram is one of the five UML diagrams used to model dynamic nature of a system. The basic elements of an statechart are;

- There are an initial state A.
- The states (represented by boxes).
- Transitions (represented by arrows and go from one state to another in one direction or another).
- Transition condition (for each one of the transitions we have one condition).
- When a state call a condition transition, the system go since this state to the state that receive as input the transition called by his condition.
- Statecharts can have composite states.

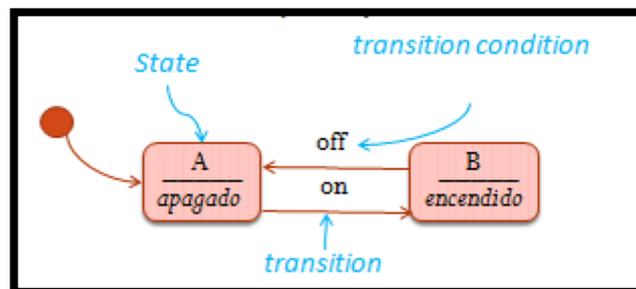


Figure 2.5.1 Example of simple statechart.

2.6 Differences between finite state machine and statechart

Before to see the differences between FSM and Statecharts we want to remember that when we apply the methods analyzed in chapter 4 (Literature Review) in our case study in chapters 5 and 6, we will work with state diagrams (statecharts) for this reason is important to clarify the differences between them. A state machine (FSM) is an abstract machine for parsing strings of input in a formal language, while a state diagrams are used to give an abstract description of the behavior of a system.

So “UML Statechart” in [43] describes a state machine. Now to clarify it, state machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events. To see the differences clearly we are going to show two examples, the first with FSM and then with an statechart.

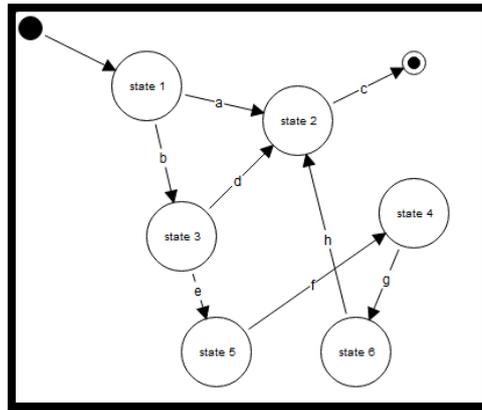


Figure 2.6.1. Finite state machine.

In the above FSM, the state machine would successfully parse the string "ac" but would not parse the string "befd" (because there is no path from the starting state to a final state that successively picks off those symbols in that order). They consist of states and arrows between the states where certain actions can trigger an transition along an arrow. Moore and Mealy machines are the two main variants, which indicate whether the output is derived from the transitions or the states themselves.

In the other hand with the below example of the statechart says that some system may be in one of three main states (TV1, ON or WORKING). We can see that the main state TV1 is a composite state by another composite state ON. At the same time the composite state ON contain the main state WORKING with four states. The main state WORKING inherits the transitions of the composite states that belongs (TV1 and ON). At the same time the main state ON inherits the transitions of TV1.

- It assumes that state Off is the initial state
- State Off can execute the transition on from state Off to state Ch1, because Ch1 is the initial state in the main state WORKING.
- Since Ch1 we can move up or down between the different states of the channels (Ch1, Ch2, Ch3, Ch4) by the transitions + and -.
- Since the main state WORKING we can execute the transition standby to go to main state ON or transition off to go to main state TV1.
- State Standby can back again to main state WORKING by the transition on or go to main state TV1 with transition off.

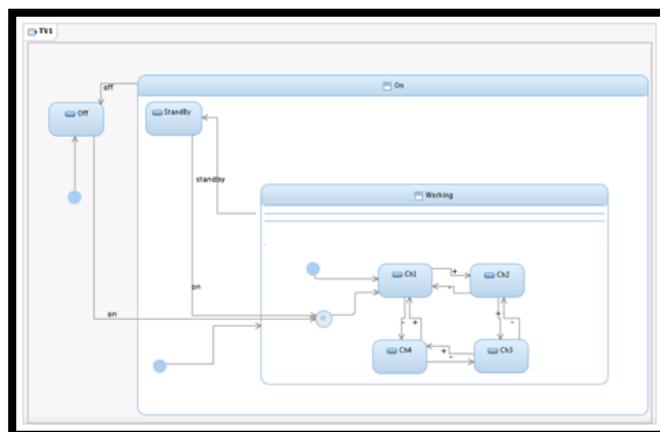


Figure 2.6.2 Statechart diagram.

So while they may appear visually similar, they are different tools from different toolboxes used for different purposes; one's from computational theory, the other is from a design description formalism.

Test cases generation for functional tests of user interfaces.

Chapter 3

State of the Art

In the first part of this chapter we talk about the evolution of methodologies for software development and the historical stages of quality developed in [46] by “Juan Oliver”.

In the second part describes basic concepts of testing, more explicit methodologies, and positions with respect to performance testing proposed by “Ian Molyneaux” in [27] and mentioned various tools to run performance tests.

Finally, there is a comparative table of the most used tools in the market. With all this demonstrates the importance of software testing in the development of such projects and how they affect the achievement of software with features more in line with new working methods applied to the development of software products.

Test cases generation for functional tests of user interfaces.

3.1 Evolution of methodologies

Before looking at the evolution of the methodologies of testing, we will briefly review the emergence of quality in the integration of software products, as this quality required in each and every one of the products coming to market is the factor that has originated and expanded testing, which we could say that they are the last step in the chain when the end product.

Software development for many people is still something that seems like magic and believes that developing software is easy and only involves pulling in front of a computer code and burn a CD. But the reality is different, since the beginning of the programming languages users have wanted to do more with computer programs, while many of those users have made the task of creating software that may solve their specific problems but eventually fail to meet many important factors for the use of software over a long time. We can see the methodologies for software development in Figure 3.1.1.

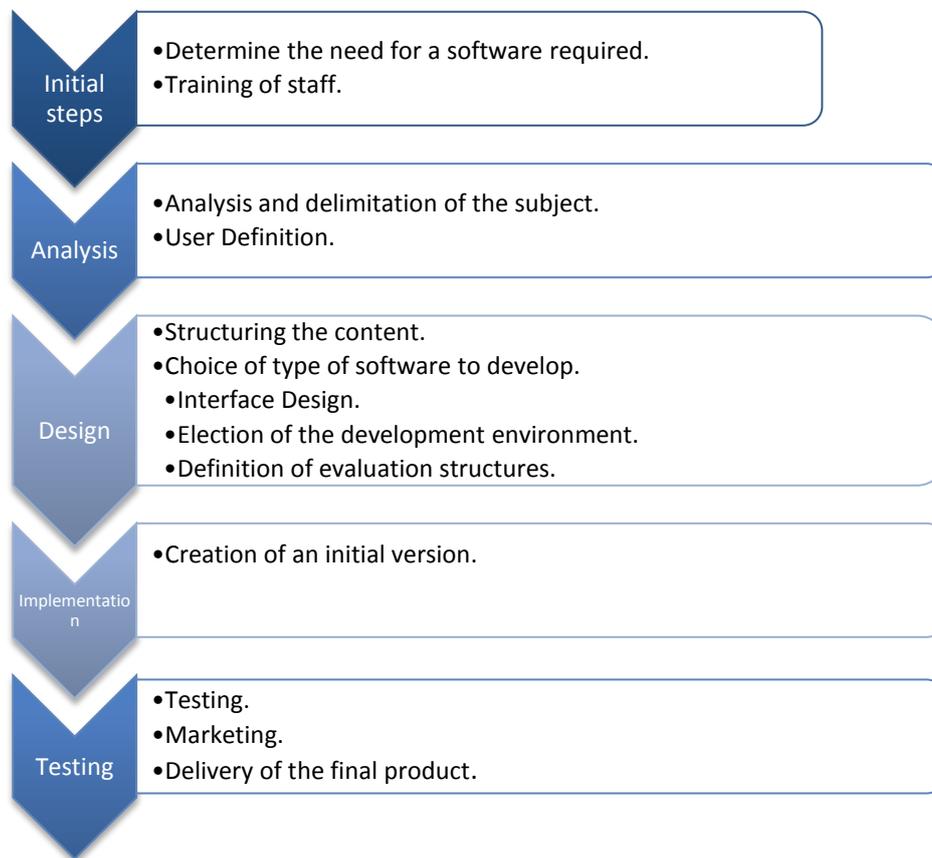


Figure 3.1.1 Methodologies for software development.

Years ago the software development tasks were done only by the developer or programmer, this was responsible for all the tasks necessary to make a program and are often limited to deliver the application with the features they asked for and did not care much if what really gave met customer expectations and thereafter be kept worrying about the software as commented in [46] by “Juan Oliver”. The absence of documentation and a defined methodology for product development made the common people distrusted software products and preferred to do their work manually. Seeing that other industries were implemented new ways to make consumer products the professional information systems began to improve the way software is developed to provide products that comply with the features that ensure that their product and their company have a reputation in the market,

thus ensuring that the delivered software product quality characteristics had standards could ensure improvement. In this chapter we will see how is done to ensure that a software product meets the necessary features to ensure quality considering the testing.

Many others have emerged with concepts and ideas arising in particular from its experience, but simultaneously all agree on a set of ideas that are basic to quality has a total character, they are shown in Figure 3.1.2.

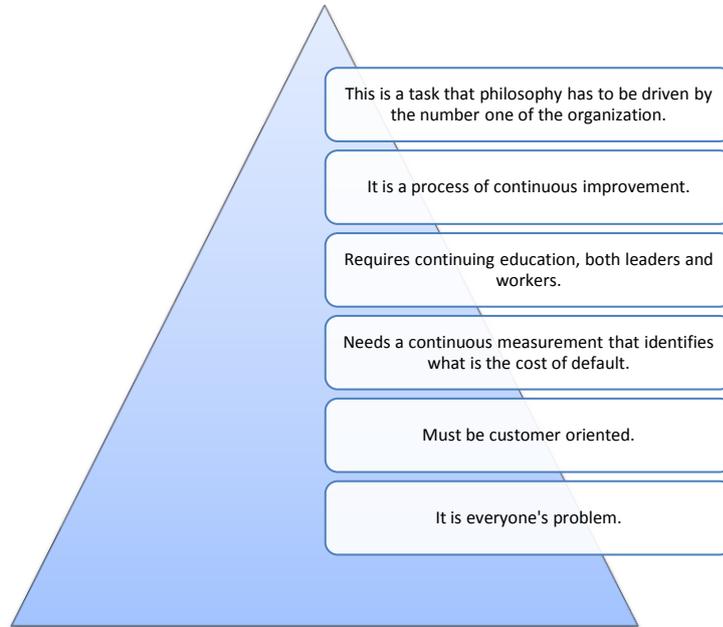


Figure 3.1.2 Basic ideas of quality.

To see how the quality has evolved during this century we can see it through analysis of its key features including all five major stages of development as we will see the first stage in point 3.2.1. We only make a brief description of each phase, for more information see [46] by “Juan Oliver”.

3.1.1 First stage. Since the industrial revolution to 1930

The Industrial Revolution, from the point of view of production, represented the transformation of manual labor by mechanized work. We can see it in Figure 3.2.1.

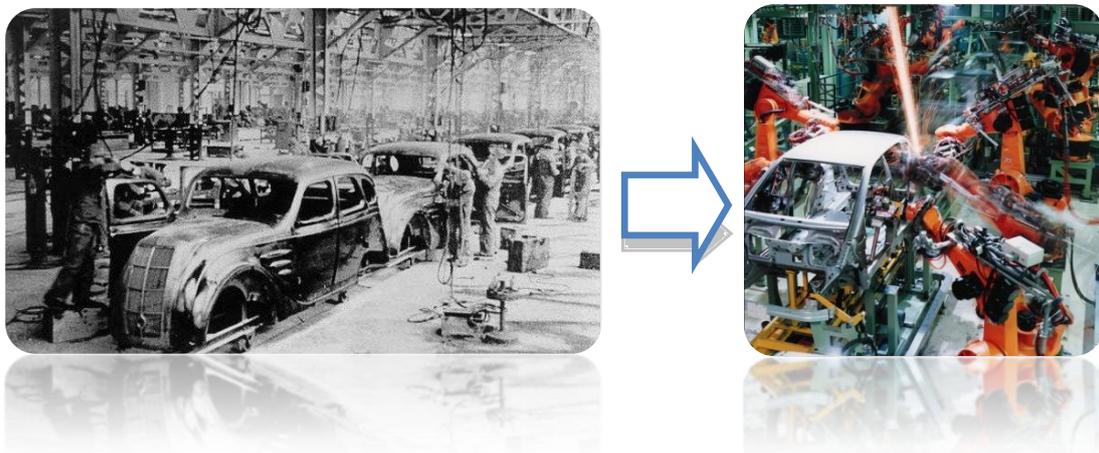


Figure 3.1.1.1 Example of automation in the vehicle industry

3.1.2 Second stage. 1930-1949

The main interest of this period is characterized by the control that ensures not only understand and choose the product malfunctions or failures, but also taking corrective action on technological processes. Quality inspectors remained a key factor in the outcome of the company, but this time not only had the responsibility for final product inspection they were distributed throughout the production process. One could say that at that time the orientation and focus on the quality went from being inspected as to quality that is controlled.

3.1.3 Third stage. 1950-1979

Summarizing the information obtained by “Juan Oliver” [46] the above stages were focused on increasing production in order to sell more, here is passed to produce higher quality in order to sell the best, considering the needs of consumers and producing according to the market. Programs begin to appear and develop quality systems for the areas of quality companies, where in addition to the measurement, incorporates quality planning, considering its orientation and approach as the quality is built from the inside.

3.1.4 Quarter stage. Decade of 80

Responsibility for quality is the first senior management, which should lead it and should involve all members of the organization. At this stage, the quality was seen as a competitive opportunity with what companies were beginning to worry about this.

3.1.5 Fifth stage. From 1990 to date

Completing this review of the last century of the recopilated information of “Juan Oliver” [46] the main characteristic of this stage is that the old distinction between product and service loses meaning. What exists is the total value for the customer. This stage is known as Total Quality Service. The customer of the 90s only willing to pay for what value means to him. That's why the quality is appreciated by the client from two points of view, perceived quality and quality factual.

3.2 Basic concepts and tools to run performance tests

According to the IEEE “Std 610.12-1990” in [11] performance testing is the degree a system or component that performs its designated functions within given limitations, such as speed, accuracy or the memory use. Performance tests were originally seen as the way to break or break existing applications for errors. Eventually understood that the idea is not to destroy an application, but rather, finding defects that make the application does not comply with all aspects expected of it, and thus promote their improvement. We will see in chapter 7 “Bugs” of this thesis an example of finding errors in an application.

3.2.1 Performance test

“J.D. Meier” and “Scott Barber” in [14] what they are looking to run performance testing is to discover and address one or more risks, timing, cost and company reputation.

These same authors in [14] suggested, for a successful testing project, which project tasks should be relevant to the project context. If this does not give, you make mistakes in which the testers tend to focus and take on aspects of the project that are not really the most important for testing, all this leads to the generation of conflict, frustration and wasted time.

The project context can be understood as important aspects of the development project to be achieved and evaluate the evidence, these aspects may be;

- Project scope.
- The project life cycle.
- Performance criteria for successful implementation.

3.2.2 Performance testing types

In software engineering, “J.D. Meier” and “Scott Barber” talk in [14] about performance testing indicating that it is in general testing performed to determine how a system performs in terms of responsiveness and stability under a particular workload. It can also serve to investigate, measure, validate or verify other quality attributes of the system, such as scalability, reliability and resource usage. Performance testing is a subset of performance engineering, an emerging computer science practice which strives to build performance into the design and architecture of a system, prior to the onset of actual coding effort. We make a brief description of some of the different types of performance testing by “J.D. Meier” and “Scott Barber”, for more information see [14].

- *Load testing* is the simplest form of performance testing. A load test is usually conducted to understand the behaviour of the system under a specific expected load.
- *Stress testing* is normally used to understand the upper limits of capacity within the system.
- *Endurance testing*: It essentially involves applying a significant load to a system for an extended, significant period of time. The goal is to discover how the system behaves under sustained use.
- *Spike testing* is done by suddenly increasing the number of, or load generated by, users by a very large amount and observing the behaviour of the system.
- *Configuration testing* rather than testing for performance from the perspective of load, tests are created to determine the effects of configuration changes to the system's components on the system's performance and behavior
- *Isolation testing* is not unique to performance testing but a term used to describe repeating a test execution that resulted in a system problem.

3.2.3 Performance testing by Ian Molyneaux

“Ian Molyneaux” propose in [27] contrary to what was proposed by “J.D. Meier” and “Scott Barber” in [14], look at the performance of software from the point of view and analyze end-user perception when it performs several tasks simultaneously in the application and verifies the delays in the execution of simultaneous tasks.

While “J.D. Meier” and “Scott Barber” focus on how to conduct performance tests and they talk about it in “Performance Testing Guidance for Web Applications” [14], “Molyneaux” is inclined to how to measure and analyze the results of those tests who talk about it in “The Art of Application Performance Testing” [27]. For this establishes key performance indicators that must be taken into account when analyzing the test results. These indicators divides them into two groups: service-oriented and efficiency oriented.

The indicators are designed to service availability and response time, this measure if an application provides a service to end users. As we did with the types of performance testing for “J.D. Meier” and “Scott Barber” [14] in previous section 3.2.2, below we are going to discuss briefly some of the indicators proposed by “Ian Molyneaux” in [27].

- *Availability*: The amount of time an application is available for the end user. The lack of availability is important because many applications will have a major cost of business, even for a small power outage. As for the performance tests, this would mean the complete inability of an end user to make effective use of the application.
- *Response Time*: The amount of time it takes for the application to respond to a user request. For performance testing, usually measured response time of the system, which is the time between when the user requests a response and that response time reaches the workstation user has requested.

The efficiency indicators are performance oriented and use, these indicators measure whether the application uses the scenario being used.

- *Performance*: The speed at which the application-oriented events occur. A good example would be the number of hits on a website within a specified period of time
- *Use*: The percentage of the theoretical capacity of a resource that is being used. Examples include the amount of network bandwidth, is being consumed by the application traffic and the amount of memory used in a server when a thousand visitors are active.

Taken together, these indicators can give an accurate idea of how the performance of an application and its impact, in terms of capacity, in the application environment. As “Ian Molyneaux” says in [27] that these methodologies rely heavily on the application that we are testing and what we want to test within the application. We can find the object-oriented methodology for the entire life cycle and performance management methodology, so we are going to talk about it in next point 3.2.4.

3.2.4 Object-Oriented Methodology for Life Cycle Completed

The methodology of "Full Life-Cycle Testing Object-Oriented" developed in [47] by “Scott W. Ambler” is a collection of techniques to verify and validate object-oriented software. The methodology FLOOT indicates a wide variety of techniques (described in Figure 3.2.4) that are available in all aspects of software development. The list is not exhaustive techniques instead aims to make explicit the fact that there is a wide range of options available.

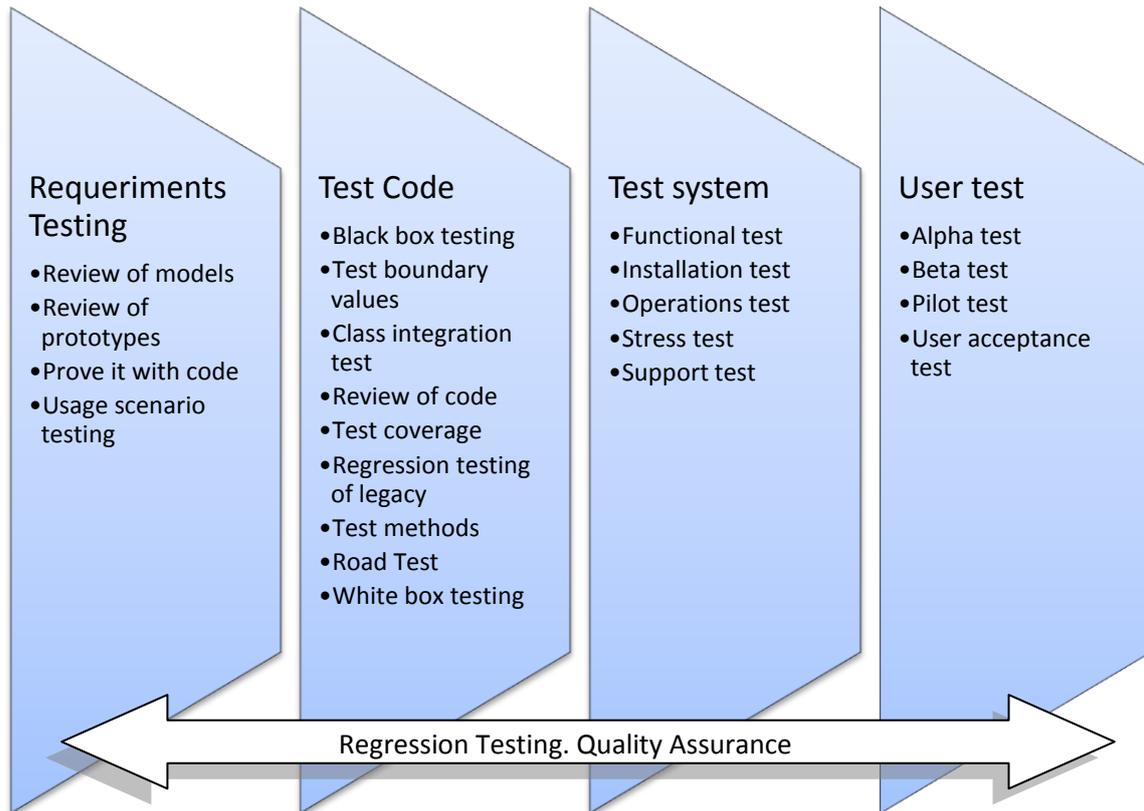


Figure 3.2.4.1 Quality Assurance.

It is important to understand that although the method FLOOT is presented as collection sequential phases need not be: FLOOT techniques can be applied also to processes agile/evolutionary. The reason that FLOOT was presented in a traditional way is to return explicit the fact that we can actually test all aspects of software development not only during encoding. The figure 3.2.4.2 below show the main techniques FLOOP which are elaborated in [47].

Black-box testing	•The test verifies that the item being tested, when given the appropriate inputs, produces the expected results
Test-Border Securities	•It is proof of extreme or unusual situations that the item should be able to handle
Class Test	•The act of ensuring that a class and all instances meet the defined behavior
Class Integration Test	•The act of ensuring that the classes and their instances, comprise a software that meets the defined behavior.
Revision Code	•A technical review form in which the deliverable is reviewed in the source code.
Component Test	•The act of validating that a component works as defined.

Covering test	<ul style="list-style-type: none">• The act of ensuring that every line of code is exercised at least once.
Design Review	<ul style="list-style-type: none">• A revision technique in which inspects a design model.
Heritage regression testing	<ul style="list-style-type: none">• It is the act of running test cases of the super classes, both directly and indirectly, in a specific subclass
Integration Testing	<ul style="list-style-type: none">• Is to perform tests to verify that a large set of pieces of software work together.
Test Method	<ul style="list-style-type: none">• Is to perform tests to verify that a method (member function) works as defined.
Review of Models	<ul style="list-style-type: none">• One type of inspection, which can range from a formal technical review to a casual tour, conducted by persons other than those who were directly involved in model development.
Road Test	<ul style="list-style-type: none">• The act of ensuring that all logical paths in the code are exercised at least once.
Prototype Review	<ul style="list-style-type: none">• It is a process by which users work through a collection of use cases, using a prototype like the real system.
Demonstrate with code	<ul style="list-style-type: none">• The best way to determine if a model reflejalo really needed, or what to build, is to build software based on the model to show that the model is good.
Regression test	<ul style="list-style-type: none">• The act of ensuring that previously tested behaviors still work as expected after changes were made to the application.
Stress Test	<ul style="list-style-type: none">• The act of ensuring that the system works as expected under large transaction volumes, users, and freight.
Technical Review	<ul style="list-style-type: none">• A technique for quality assurance in which the design of your application is reviewed extensively by a group of your peers.
Usage Scenarios Test	<ul style="list-style-type: none">• A testing technique in which one or more validated a model following the logic of usage scenarios.
User Interface test	<ul style="list-style-type: none">• Is to test the user interface to ensure it meets the standards and requirements defined.
White-box testing	<ul style="list-style-type: none">• Is to perform tests to verify that specific lines of code work as defined. Also known as test-transparent box

Figure 3.2.4.2 Test Techniques.

3.2.5 Types of performance tests

Commonly performance tests are conducted to Web applications, since these are the ones with more orders for execution by the environment in which they run. The different types of performance tests that can find and apply to most applications we can see in Figure 3.2.5.1.



Figure 3.2.5.1 Types of tests.

3.3 Tools to conduct performance tests

As we have talked in chapter 2, concretely in section 2.2 and his subpoints, when we try to generate test cases we can calculate them manually or automated. We can apply the same theory to software testing existing manually and automated tools.

Manual testing: Manual tests are the oldest type of evidence that exists, these are that a person (tester) manually run system operations without the aid of automation tools. In these tests the tester must be patient, observant, creative, among other qualities. Manual tests are focused on product functionality, usability and graphical user interface.

Automated testing: The test automation is the cycle where a software quality automation application is used to control the execution of the tests, compare results, create preconditions, etc... Available on the market there are a lot of automated testing tools and frameworks work. These tools are known as Computer Aided Software Testing (CAST) explained in [53] and come in free and commercial.

All the tools that we are going to describe collecting information from their official websites are free and we have download and tested every one of them to get more knowledge about the different tools for testing. These and other tools are compared in Section 3.4.

3.3.1 JMeter

It is necessary to highlight that we have downloaded and tested these free applications and we have interacted with them testing the feature's functionality. The Apache JMeter desktop application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions. If we need more information about this tool we can see [48].

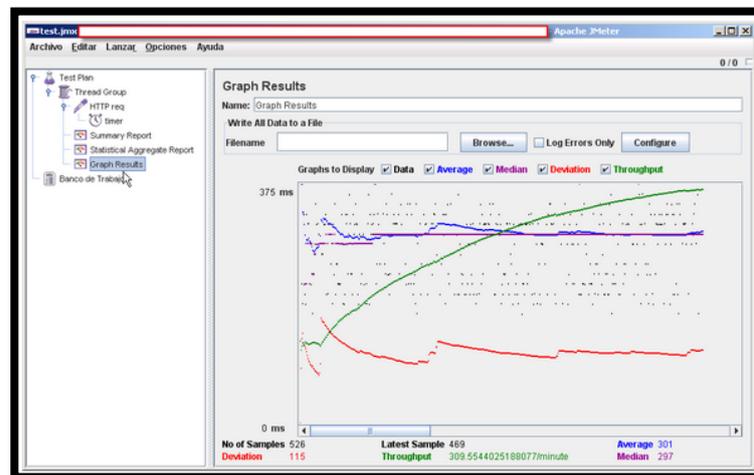


Figure 3.3.1.1 JMeter graph result.

Apache JMeter may be used to test performance both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers and more). It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types. You can use it to make a graphical analysis of performance or to test your server/script/object behavior under heavy concurrent load.

3.3.2 Jcrawler

Note that this tool like the previous (JMeter) we have downloaded and tested looking for limitations or advantages of it.



Jcrawler was created as an open-source (under the CPL) Stress-Testing Tool for web-applications. It comes with the crawling/exploratory feature. You can give Jcrawler a set of starting URLs and it will begin crawling from that point onwards, going through any URLs it can find on its way and generating load on the web application. The load parameters (hits/sec) are configurable. If we need more information about this tool we can see his official website [49].

Jcrawler is under redevelopment to create with even more features.

3.3.3 WAPT

WAPT, which information we have collected from the official WAPT page [41], is a load and stress testing tool that provides you with an easy-to-use and cost-effective way to test any web site: from a privately used business application to a distributed web portal consisting of load balancers, web servers, application servers, database storages, etc. For more information about the tool we can consult his official page [41] of our bibliography.

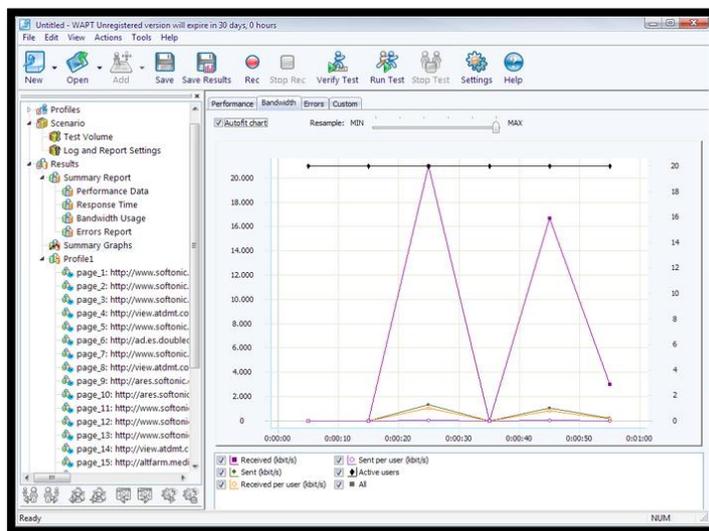


Figure 3.3.1 WAPT Report graph.

3.3.4 Netsparker

Among its features is announced that is free to report false positives, or whatever it is, identify vulnerabilities requests as they really are not. Although this feature is tricky, since it cannot guarantee the label as vulnerable as "possible", so that later the auditor to make the necessary verifications. The engines support the detection of the most common risks: SQL Injection, XSS, including local and remote files, command injection, CRLF, obsolete files, source

code, hidden resources, directory listing, configuration vulnerabilities of different web servers, etc.

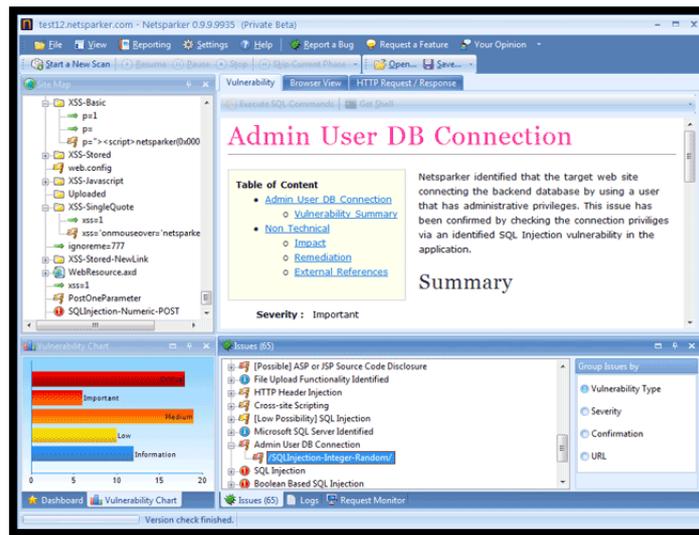


Figure 3.3.4.1 Graphical interface of Netsparker.

But undoubtedly the most noteworthy and works best is the SQL injection, which also allows execution of commands and statements once it detects a vulnerable parameter. To get more information about this tool we can see [50].

3.3.5 OpenSTA

A test software designed around CORBA, originally developed to be marketed by CYRANO. OpenSTA is a set of tools has the ability to test through scripts and heavy load tests with performance measurements from Win32 platforms. To get more information about this tool we can see [51].

3.3.6 TestMaker

This application is capable of performing functional testing, regression, load and performance. It offers two versions: Community and Enterprise, both free but support is provided by the Enterprise. To get more information about this tool we can see [52].

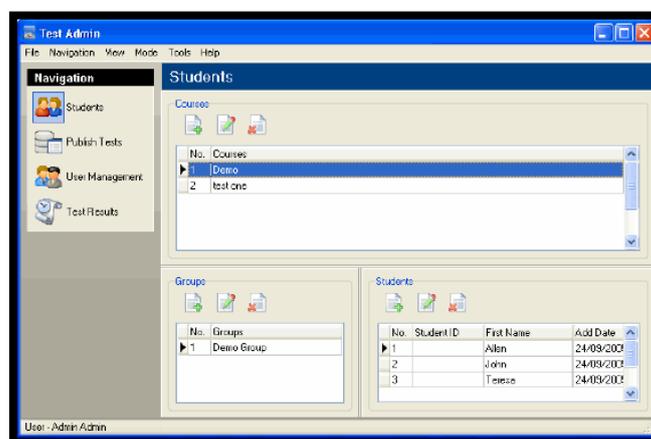


Figure 3.3.6.1 Interface of the tool TestMaker.

3.4 Comparative analysis

In this section we compare 5 software tools of the six previously commented (since the section 3.3.1 to 3.3.6) for testing applications based on Internet protocols and client-server software architectures. The comparative study will be aimed at the most representative characteristics and comparison tools to analyze the trend of their development in terms of functionality based on the use that we have made of them and the information that we have collected through the web and other users experiences.

	Name	OpenSTA	Web Load	TestMark er	vPerfor mer	WAPT	Netsparker
i.	Url address	http://opensta.sourceforge.net	www.gomez.com	www.pushtotest.com	www.verisium.com	www.loadtestingtool.com/	www.mavitusnasecurity.com
ii.	License type	FREWARE OPEN SOURCE	PAYMENT	FREEWAR E OPEN SOURCE	PAYME NT	PAYMENT	FREWARE (standard version) PAYMENT (proof. version)
iii.	Platform	WINDOWS	MULTIPLAT FORM	MULTIPLA TFORM	WINDO WS	WINDOWS	WINDOWS
iv.	Manag ement	Yes	Yes	Yes	Yes	Yes	Not specified
v.	Support IP	No	No	No	No	Yes	Yes
vi.	Proxy http	Yes	Yes	Yes	Yes	Yes	Yes
vii.	Protocols	HTTP 1.0/1.1 HTTPS	HTTP HTTPS	HTTP 1.0/1.1 HTTPS	HTTP	HTTP HTTPS	HTTP HTTPS
viii	Monit oring	No	No	No	Yes	Yes	Yes

ix.	Manager Cookies	Yes	Yes	Yes	Yes	Yes	Yes
x.	User Scalability	Yes	Not specified	Not specified	Yes	Yes	Not specified

Table 3.4.1. Comparative Analysis of Commercial Applications.

Predicting the behavior of an application under specified conditions as a large number of users, resources, restricted or limited bandwidth, among others, becomes a task of utmost importance to users and the company that owns the application and that is why there are these tools on the market.

Test cases generation for functional tests of user interfaces.

Chapter 4

Literature Review

In the first part of this chapter we will analyze the W method created by “Kirill Bogdanov” in [4] with whom we maintained contact through several emails to clarify some of the more complicated points of his method in their articles published. After that we will analyze the W_p method also developed by “Kirill Bogdanov” in [7], which is an improvement of the said W created.

At the end of the chapter will discuss other methods which do not have relationship with W and W_p finding alternative ways for creating testing algorithms like the papers of whom principal authors are “Siamak Haschemi” [32] (section 4.2) and “Jun Wang” [33] (section 4.3).

Test cases generation for functional tests of user interfaces.

4.1 Analysis of W & Wp methods.

An existing testing method for statecharts with hierarchy and concurrency is based on what is known as the “Chow’s W method” developed initially in [4] by “Kirill Bogdanov”. In the review of the document “Testing from statecharts using the Wp method” [7] it’s presented W method again and an extension of this testing method, the Wp method. Today there are many methods to calculate the approximate number of test cases of an application, but this was the first who showed convincingly and more robust method, for this reason we have decided to work with these methods in the case study of chapter 5 and improving them with the development of a new algorithm in chapter 6 of this thesis.

Before to see how it works we are going to describe in next section 4.1.1 the statechart used by “Kirill Bogdanov” in [7]. After that we will see the different variables and formulas gathering information of [4] and [7] trying to explain briefly how they apply these method and the meaning of each variable used in both research papers.

4.1.1 Describing the statechart

Statechart extracted from [7] is a specification and design language derived from finite-state machines described by “T. Chow” in [6] by extending them with arbitrarily complex functions on transitions, state hierarchy and concurrency. Below, in figure 4.1.3.1 shows the statechart which has been used in [7], consider a simple tape recorder capable of playback, rewinding, fast forwarding and recording as well as changing a side of a tape when the button play is pressed during playback or when a tape ends. The inputs to this controller are events play, stop, rec, rew, ff and tape-end.

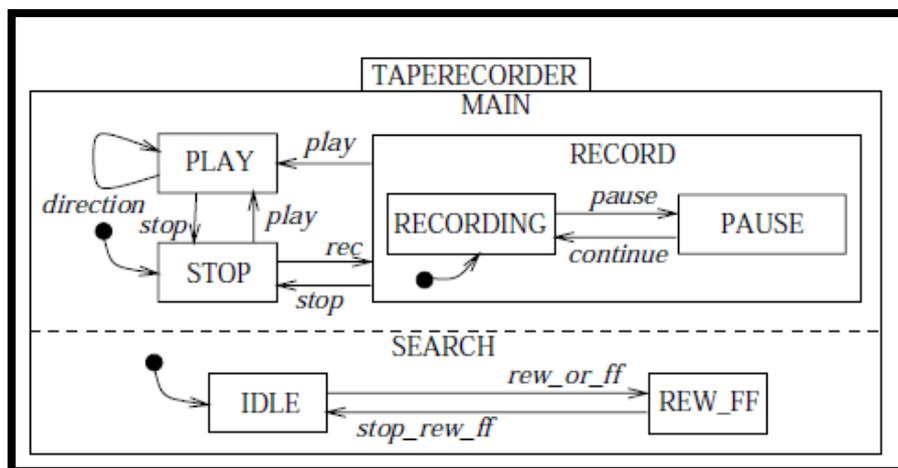


Figure 4.1.1.1 The taperecorder statechart.

- Main States: TAPERECORDER, MAIN, RECORD and SEARCH.
 - Concurrent states: the main state TAPERECORDER include concurrent composite states like MAIN (who contains RECORD) and SEARCH. System can be simultaneously in the composite states MAIN (who contains RECORD) and SEARCH.
 - States: PLAY, STOP, RECORDING, PAUSE, IDLE and REW_FF.
- Transitions: direction, stoy, play, re, pause, continue, rew_or_ff and stop_rew_ff.

- Labels: The inability of finite-state machines to represent data without a state explosion can be solved by using functions on transitions, which can access and modify global data.
- Action: An operation carried out by a label on a transition when that transition executes is called an *action*.

4.1.2 Flattened statechart of the MAIN without hierarchy

As “Kirill Bogdanov” explain in [4] and [7] the most simple approach to testing state hierarchy is to flatten a statechart, i.e. turn it into a equivalent one without substates (AND / OR states). To apply the different algorithms (the existing as W & Wp methods and the new U-Method develop by us) to our case study we will see on next chapter 5 and 6 that it is necessary to obtain a flattened statechart of our case study.

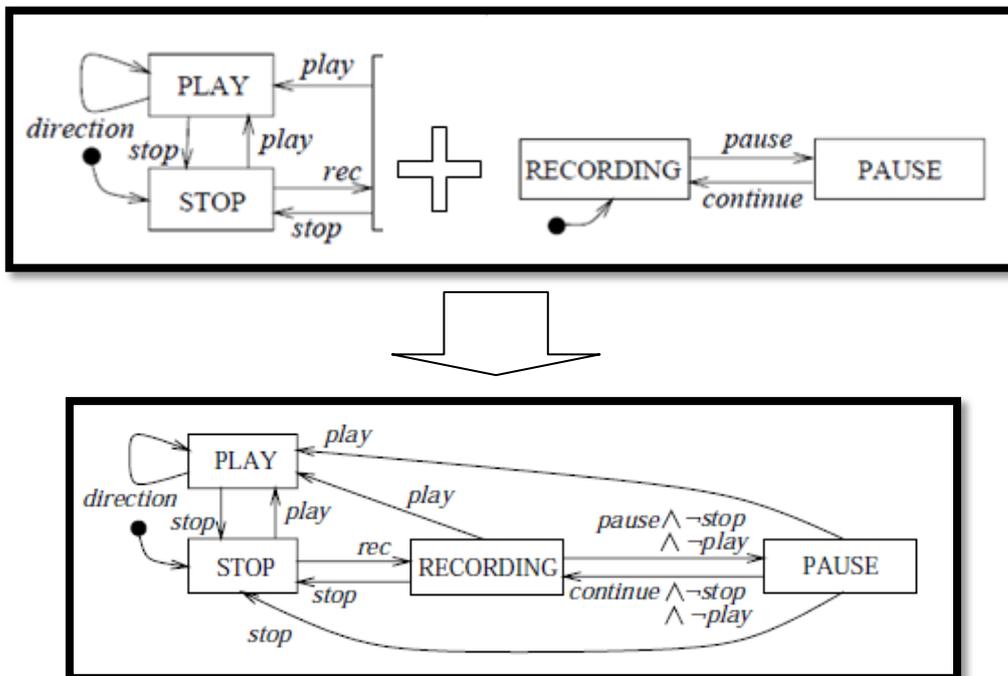


Figure 4.1.2.1 The flattened statechart of the MAIN state

The idea is to eliminate any hierarchy in the initial statechart. However as discussed in [7] don't work with this model, they will work for parts, gradually eliminating the hierarchy of the initial statechart

- Test case generation for statecharts without state hierarchy.
- Test case generation for state hierarchy.
- Test case generation for concurrency.

4.1.3 State hierarchy

State hierarchy of a statechart can be viewed as a tree. The root state is the implicit top-level state; it was introduced because TAPERECORDER is an AND-state and statecharts require the top-level state to be an OR one.

- TAPERECORDER is an AND-state because it has concurrent parts separated by a dashed line.

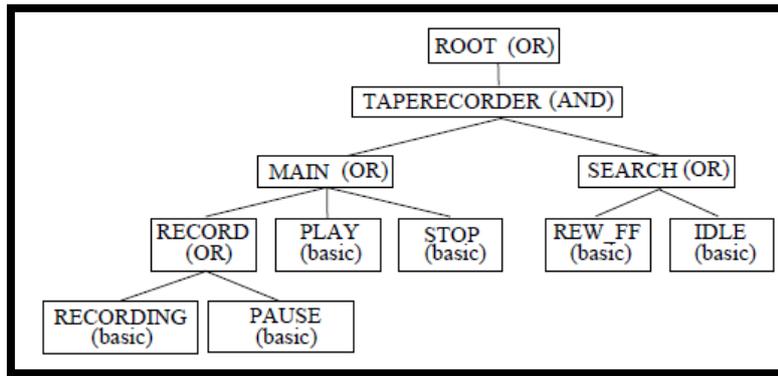


Figure 4.1.3.1 The state tree of the tape recorder.

- The p provides the set of only basic substates of a given state.
- The opposite to p is *parent*.
- The **scope** of a transition is the lowest-level OR-state above all source and target states of it.

4.1.4 Configurations

Sets of states which are left and entered by full compound transitions are called configurations as defined by “K. Bogdanov and M. Holcombe” in [5] and consist of states a statechart can be in simultaneously. Every substate of an entered AND-state has to be entered, so that a possible configuration in Figure 4.1.3.1 is: {root, TAPERECORDER, MAIN, SEARCH, RECORD, PAUSE, REW_FF}.

A configuration is uniquely determined by a set of basic states in it -> {PAUSE, REW_FF}. Every state in a flattened statechart corresponds to a configuration in the original one.

Static reactions	Interlevel transitions	Paths
<ul style="list-style-type: none"> •Are a special case of transitions which may occur within a state, without leaving it or entering it again (thus no states are left and no default transitions fire when static reactions are taken). 	<ul style="list-style-type: none"> •Are transitions which cross levels of hierarchy. For instance, if the controller had a transition from PAUSE to STOP, it would be interlevel. Interlevel transitions are not considered in this paper. 	<ul style="list-style-type: none"> •Sequences of labels of transitions (not necessarily those which could be taken) are called paths in this document.

Figure 4.1.4.1 Types of configurations (compound transitions).

4.1.5 Test generation for statecharts with W method

As we have commented in setion 4.1.2 we can divide the statechart in three parts and apply test case generation methods for each one of them; test case generation for statecharts without state hierarchy, test case generation for state hierarchy and test case generation for concurrency. We are going to see the three phases to generate the test cases.

➤ Test case generation for statecharts without state hierarchy

With some restrictions exposed in “Statechart testing method for aircraft control systems” [4], statecharts which do not contain state hierarchy or concurrency are behaviourally equivalent and it isn’t necessary to descompund in new states. For a systematic construction of a set of test cases, auxiliary sets have to be built and we don’t need the information of the interior of record.

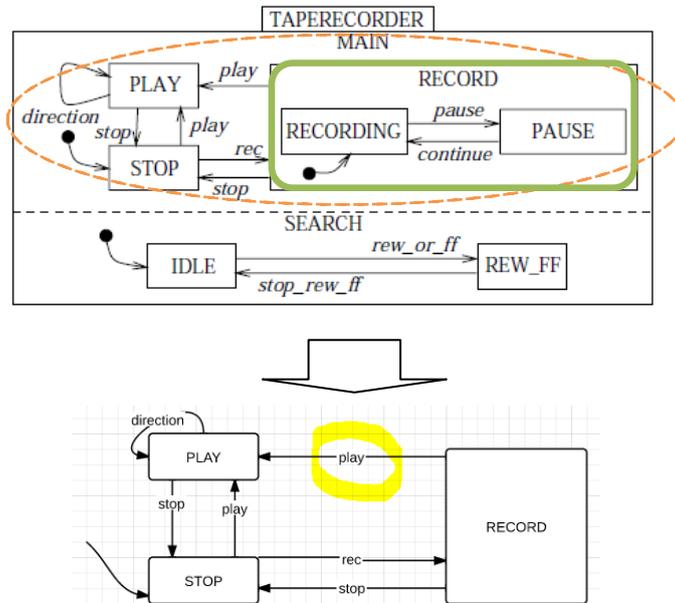


Figure 4.1.5.1 MAIN without state hierarchy

The method is founded on the “Chow’s W method” in [6] and relies on a separation of function and transition diagram testing. The method concentrates on testing of the transition diagram, behaviour of the labels of transitions is assumed to have been tested in advance. The approach to testing of a transition diagram is very similar to testing of labelled-transition systems. The main difference is the reliance of this work on an input/output behaviour of transitions rather than on deadlocks to tell a tester whether a transition with a given label exists from a particular state in an implementation or not. For a systematic construction of a set of test cases, auxiliary sets have to be built. We use MAIN without state hierarchy show in fig. 4.1.5.1.

There are three variables to elaborate the Equation of “W-Method” explained in [4]. We are going to do a brief description of them, to see more information we can see the articles “Statechart testing method for aircraft control systems” [4], “Automated test set generation for statecharts” [5] and “Testing from statecharts using the Wp method” [7] in which the principal author is “Kirill Bogdanov”. The set of transition labels (denoted by Φ) is the set of labels of a statechart. If we calculate the Φ of the above statechart we obtain; $\Phi = \{\text{stop, play, rec, direction}\}$. Any desired state starting from the initial one (denoted by C) is; $C = \{1, \text{play, rec}\}$. Here 1 denotes an empty sequence. Finally the most complicated variable to understanding is W also developed in [5] in which it is explained that W allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. Such paths for every pair of states comprise a characterisation set. So $W = \{\text{stop, play}\}$. Each element of this particular W is a sequence consisting of a single label. In Equation 4.1.5.1 we can see the formula for the W-Method to the non-hierarchical part.

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 4.1.5.1. Non hierarchical

➤ Test case generation for state hierarchy

As mentioned in section 4.1.2 and also developed in [4] and [7] by “K. Bogdanov” the most simple approach to testing state hierarchy is to flatten a statechart.

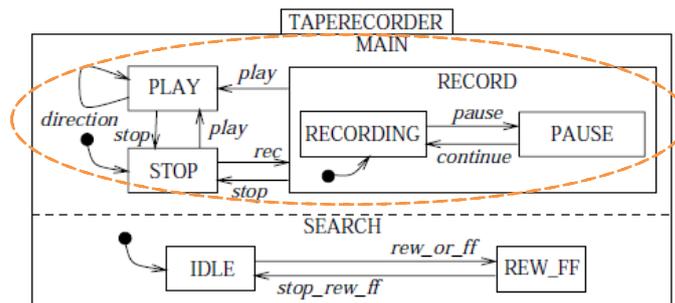


Figure 4.1.5.2 Calculating state hierarchy.

Moreover, if certain parts of a statechart are implemented separately and do not share any labels, it isn't necessary to test for faults where labels from one part are used in another one and vice-versa, significantly reducing the size of a test set.



Figure 4.1.5.3 Concurrent part omitted.

Now we need to calculate the test case generation begins with the construction of a tuple (Φ, C, W) called TCB for every non-basic state considering all its substates as basic ones. As indicated in [4] by “K. Bogdanov” to calculate the three variables to form the Equation 4.1.5.1 of hierarchical part of the statechart are the same that we have applied in previous step but now we will do it over the statechart of the non hierarchical part which we can see in the below figure 4.1.5.4.

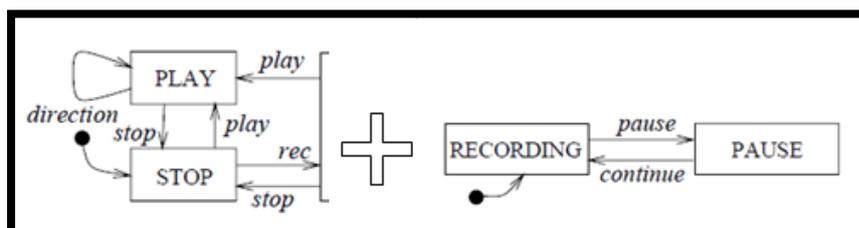


Figure 4.1.5.4 State hierarchy part

➤ Test case generation for concurrency

Finally the concurrent part follows the same approach as testing of state hierarchy, except that multiple transitions are attempted.

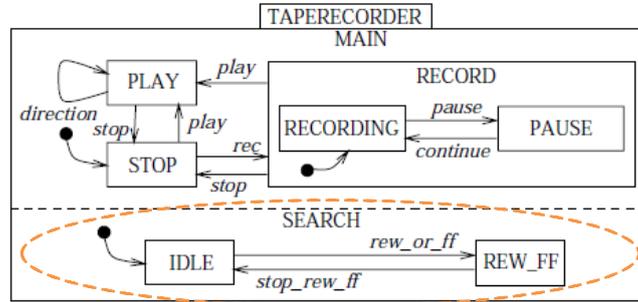


Figure 4.1.5.5 Calculating for concurrency

The results calculated in “Testing from statecharts using the Wp method” [7] for the concurrent part are:

$$\begin{aligned}\Phi_{SEARCH}^M &= \{rew_or_ff, stop_rew_ff\} \\ C_{SEARCH}^M &= \{1, rew_or_ff\} \\ W_{SEARCH}^M &= \{rew_or_ff\}\end{aligned}$$

At this point we have calculated the hierchycal, non hierchycal and concurrent part of the statechart. In next point 4.1.6. we are going to calculate the test cases for TAPERECORDER and after we apply the equation 1 to all the model in setion 4.1.7.

4.1.6 Applying the formulas for all the statechart

With the formulas of figure 4.1.6.1 calculated in [7] by “K. Bogdanov” we can build the complete tuple $(\Phi_{TAPERECORDER}^M, C_{TAPERECORDER}^M, W_{TAPERECORDER}^M)$, with which we can calculate the number of test cases to be implemented on the statechart being analyzed. For more information we can see the document [7] of our bibliography.

$$\begin{aligned}\Phi_{TAPERECORDER}^M &= ((\{I\} \cup \Phi_{MAIN}^M) * (\{I\} \cup \Phi_{SEARCH}^M) \setminus \{I\}) = \{play, stop, direction, rec, \\ &pause, continue, rew_or_ff, stop_rew_ff, play_rew_or_ff, stop_rew_or_ff, \\ &direction_rew_or_ff, rec_rew_or_ff, pause_rew_or_ff, continue_rew_or_ff, \\ &play_stop_rew_ff, stop_stop_rew_ff, direction_stop_rew_ff, \\ &rec_stop_rew_ff, pause_stop_rew_ff, continue_stop_rew_ff\}, \\ C_{TAPERECORDER}^M &= C_{MAIN}^M * C_{SEARCH}^M = \{1, play, rec, rec\ pause, rew_or_ff, play_rew_or_ff, \\ &rec_rew_or_ff, rec_rew_or_ff\ pause\}, \\ W_{TAPERECORDER}^M &= W_{MAIN}^M \cup W_{SEARCH}^M = \{stop, play, pause, rew_or_ff\}.\end{aligned}$$

Figure 4.1.6.1 Equations to calculate number of test cases.

4.1.7 Final results for W-Method

For the tape recorder under the assumption of an implementation containing no more states than the specification, test case generation produces 672 sequences calculated in [7] and corroborating the results by us, because each one of the methods that we are describing in this chapter and extracted from [4], [5] and [7] of which “K. Bogdanov” developed the W & Wp methods we have been calculated on the paper from the beginning, making the whole process from the beginning and checking that the results obtained after applying the formulas are the same as reported by “K. Bogdanov” in [7]. It is later contrasted with the size obtained using the Wp method, which we will do a brief description of how calculate test cases for Wp-Method.

$$T_{TAPERECORDER}^M = C_{TAPERECORDER}^M \cup (C_{TAPERECORDER}^M * \Phi_{TAPERECORDER}^M) * W_{TAPERECORDER}^M = 672$$

Equation 4.1.7.1. Results for the W-Method

At the final of this point 4.1. we will analyze the result obtained with W (analyzed in 4.1.5) and Wp which we are going to analyze below in point 4.1.8.

4.1.8 Test case generation for statecharts with Wp method

The Wp method implemented in [7] by “K. Bogdanov” is an improvement of the W one, targeted at the reduction of a number of test sequences. We are going to do a brief description of this method but to get more information about the method we can consult [7] of our bibliography.

$$\bigcup_{i=1}^n P_i \cdot W_i. \text{ Since } |\bigcup_{i=1}^n P_i \cdot W_i| \ll n \times m \times w.$$

Equation 4.1.8.1 Wp method.

➤ Two-phase approach

Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly. To cope with this, a two-phase approach is proposed in [7] by “K. Bogdanov” where the first stage tests a part of a statechart and checks whether the small sets identify while the second phase check all remaining transitions of the implementation for correct output and ending state as defined by the specification.

➤ First phase of the Wp method

In one of the emails exchanged with “K. Bogdanov”, he says “The purpose of the first phase is to ensure that small w sets (explained below) are capable of identifying states in an implementation”. For this purpose, every state of an implementation is visited and W set is applied in that state. Let’s see an example of the application of W_{conf}^{root} .

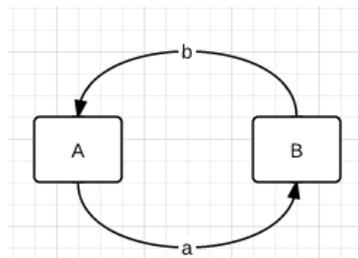


Figure 4.1.8.1 Calculating small sets

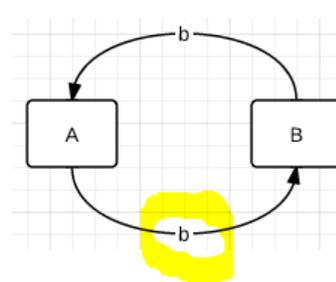


Figure 4.1.8.2 Erroneous transition

To clarify this Small sets for these states could be $w_A = \{a\}$ and $w_B = \{b\}$. To understand this point we had to exchange several emails with K. Bogdanov, who summarized his explanations and include in the analysis of this document.

If an implementation has an erroneous transition "b" from A, this state look as both A and B in this implementation. Hence if there is an erroneous transition leading to state A rather than state B somewhere in this implementation, the defect will not be found if only small sets are used, because we'll check the target state with w_B and the erroneous transition "b" from the A state will make us think that this is B rather than A.

Summarizing, the results obtained for the first phase obtained in [7] by "K. Bogdanov" for the Wp-Method if $T_1 = C^M * W = 8 * 4 = 32$ test cases in phase 1.

➤ Second phase, apply small w sets

At the second phase calculated in [7] all transitions which were left out in the first phase are tested, using small sets w_{conf}^{root} to identify configurations and therefore create less test cases compared to the W method while still providing the same level of confidence in the result of testing. In the emails that we exchanged with "K. Bogdanov" (who is the creator of both the W and the Wp method as we have commented several times in this thesis), he clarified to us the second phase of the Wp method Because this phase is so complicated. K. Bogdanov say to us in the email that "Construction of elements of w sets is similar to that for the full W method (recursively bottom-up). CE is a function which computes a configuration entered by a statechart when a sequence of operations is attempted." Let the initial configuration of a statechart be denoted by $conf_{init} = \{STOP, IDLE\}$ (only basic states in these two configurations are shown because the initial state is PAUSE and IDLE and all the parents of these two states but we only need to show the basic states this is the full $conf_{init}$); $conf_{init} = \{STOP, IDLE, MAIN, TAPERECODER, ROOT, SEARCH\}$).

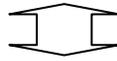
With the clarifications of K. bogdanov we were able to develop ourselves formula of Wp-Method from [7] showing the test cases obtained below.

$$TS = C^M * \Phi^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n})$$

Equation 4.1.8.1 Phase 2 of Wp-Method.

Developing the formula:

TS= {1,play, rec, rec pause, rew_or_ff, play-rew_or_ff, rec-rew_or_ff, rec-rew_or_ff pause} *
 {play, stop, direction, rec, pause, continue, rew_or_ff, stop_rew_ff, play-rew_or_ff; stop-
 rew_or_ff, direction-rew_or_ff, rec-rew_or_ff, pause-rew_or_ff, continue-rew_or_ff, play-
 stop_rew_ff, stop-stop_rew_ff, direction-stop_rew_ff, rec-stop_rew_ff, pause-stop_rew_ff,
 continue-stop_rew_ff }



TS= { play, stop, direction, rec, pause, continue, rew_or_ff, stop_rew_ff, play-rew_or_ff, stop-
 rew_or_ff, direction-rew_or_ff, rec-rew_or_ff, pause-rew_or_ff, continue-rew_or_ff, play-
 stop_rew_ff, stop-stop_rew_ff, direction-stop_rew_ff, rec-stop_rew_ff, pause-stop_rew_ff,
 continue-stop_rew_ff, play play, play stop, play direction, play rec, play pause, play continue, *
 play rew_or_ff [=play, rew_or_ff], play stop_rew_ff, play play-rew_or_ff, play stop-rew_or_ff,
 play direction-rew_or_ff, play rec-rew_or_ff, play pause-rew_or_ff, play continue-rew_or_ff,
 play play-stop_rew_ff, play stop-stop_rew_ff, play direction-stop_rew_ff , play rec-
 stop_rew_ff, play pause-stop_rew_ff, play continue-stop_rew_ff, rec play, rec stop, rec
 direction, rec rec, rec pause [=rec, pause], rec continue, rec rew_or_ff, rec stop_rew_ff, rec
 play-rew_or_ff; rec stop-rew_or_ff, rec direction-rew_or_ff, rec rec-rew_or_ff, rec pause-
 rew_or_ff, rec continue-rew_or_ff, rec play-stop_rew_ff, rec stop-stop_rew_ff, rec direction-
 stop_rew_ff, rec rec-stop_rew_ff, rec pause-stop_rew_ff, rec continue-stop_rew_ff, (rec
 pause) * play [=rec,play,pause,play], (rec pause) * stop, (rec pause) * direction, (rec pause) *
 rec, (rec pause) * pause, (rec pause) * continue, (rec pause) * rew_or_ff, (rec pause) *
 stop_rew_ff, (rec pause) * play-rew_or_ff, (rec pause) * stop-rew_or_ff, (rec pause) *
 direction-rew_or_ff, (rec pause) * rec-rew_or_ff, (rec pause) * pause-rew_or_ff, (rec pause) *
 continue-rew_or_ff, (rec pause) * play-stop_rew_ff, (rec pause) * stop-stop_rew_ff, (rec
 pause) * direction-stop_rew_ff, (rec pause) * rec-stop_rew_ff, (rec pause) * pause-
 stop_rew_ff, (rec pause) * continue-stop_rew_ff, rew_or_ff play, rew_or_ff stop, rew_or_ff
 direction, rew_or_ff rec, rew_or_ff pause, rew_or_ff continue, rew_or_ff rew_or_ff, rew_or_ff
 stop_rew_ff, rew_or_ff play-rew_or_ff, rew_or_ff stop-rew_or_ff, rew_or_ff direction-
 rew_or_ff, rew_or_ff rec-rew_or_ff, rew_or_ff pause-rew_or_ff, rew_or_ff continue-
 rew_or_ff, rew_or_ff play-stop_rew_ff, rew_or_ff stop-stop_rew_ff, rew_or_ff direction-
 stop_rew_ff, rew_or_ff rec-stop_rew_ff, rew_or_ff pause-stop_rew_ff, rew_or_ff continue-
 stop_rew_ff, play-rew_or_ff play, play-rew_or_ff stop, play-rew_or_ff direction, play-
 rew_or_ff rec, play-rew_or_ff pause, play-rew_or_ff continue, play-rew_or_ff rew_or_ff, play-
 rew_or_ff stop_rew_ff, play-rew_or_ff play-rew_or_ff, play-rew_or_ff stop-rew_or_ff, play-
 rew_or_ff direction-rew_or_ff, play-rew_or_ff rec-rew_or_ff, play-rew_or_ff pause-rew_or_ff,
 play-rew_or_ff continue-rew_or_ff, play-rew_or_ff play-stop_rew_ff, play-rew_or_ff stop-
 stop_rew_ff, play-rew_or_ff direction-stop_rew_ff, play-rew_or_ff rec-stop_rew_ff, play-
 rew_or_ff pause-stop_rew_ff, play-rew_or_ff continue-stop_rew_ff, rec-rew_or_ff play, rec-
 rew_or_ff stop, rec-rew_or_ff direction, rec-rew_or_ff rec, rec-rew_or_ff pause, rec-rew_or_ff
 continue, rec-rew_or_ff rew_or_ff, rec-rew_or_ff stop_rew_ff, rec-rew_or_ff play-rew_or_ff,
 rec-rew_or_ff stop-rew_or_ff, rec-rew_or_ff direction-rew_or_ff, rec-rew_or_ff rec-rew_or_ff,
 rec-rew_or_ff pause-rew_or_ff, rec-rew_or_ff continue-rew_or_ff, rec-rew_or_ff play-
 stop_rew_ff, rec-rew_or_ff stop-stop_rew_ff, rec-rew_or_ff direction-stop_rew_ff, rec-
 rew_or_ff rec-stop_rew_ff, rec-rew_or_ff pause-stop_rew_ff, rec-rew_or_ff continue-
 stop_rew_ff, (rec-rew_or_ff pause) * play [=rec-rew_or_ff play pause play =rec-
 rew_or_ff,play,pause,play], (rec-rew_or_ff pause) * stop, (rec-rew_or_ff pause) * direction,
 (rec-rew_or_ff pause) * rec, (rec-rew_or_ff pause) * pause, (rec-rew_or_ff pause) * continue,
 (rec-rew_or_ff pause) * rew_or_ff, stop_rew_ff, (rec-rew_or_ff pause) * play-rew_or_ff, (rec-
 rew_or_ff pause) * stop-rew_or_ff, (rec-rew_or_ff pause) * direction-rew_or_ff, (rec-
 rew_or_ff pause) * rec-rew_or_ff, (rec-rew_or_ff pause) * pause-rew_or_ff, (rec-rew_or_ff

pause) * continue-rew_or_ff, (rec-rew_or_ff pause) * play-stop_rew_ff, (rec-rew_or_ff pause) * stop-stop_rew_ff, (rec-rew_or_ff pause) * direction-stop_rew_ff, (rec-rew_or_ff pause) * rec-stop_rew_ff, (rec-rew_or_ff pause) * pause-stop_rew_ff, (rec-rew_or_ff pause) * continue-stop_rew_ff} = $8*20+146= 306$ test cases in phase 2

Sumarizing, the size of the set of test cases for the first phase of the Wp method is 32 and the second one 306, resulting in 338 sequences which is a half of the set provided in section 4.1.7 of this chapter with W-Method for the same statechart that Wp-Method. These results are calculated in [7] by “K.Bogdanov” and corroborated by us in this chapter.

4.1.9 Conclusions

We have analyzed W Method in our case study over different versions of the statechart and we can ensure that it is a high robust method. The Wp method which also we have analized in chapter 5, can be applied to statecharts in a similar way to the W one and her objective is to reduce the number of total test cases and remove all the redundant test cases generated with W method to be a more efficient method. As “K. Bodganov” explain in [7] one of the most important advantages is that unlike other methods, the W method and Wp method can be applied to all protocols, and can guarantee the detection of any output and transfer faults under certain conditions.

4.2 Satisfy All-Configurations-Transitions on Statecharts

This paper was very important for us, because the existing algorithms are based on “state-cover”, this menas that for example before to apply the formulas of two existing methods as W or Wp developed in [7] by “K. Bogdanov” he want to ensure that every one of the states are identifiable on the basis of the transitions that each state run. With the this paper of “Siamak Haschemi” based on “Model Transformations to Satisfy All-Configurations-Transitions on Statecharts” developed in [32] we try to apply with our new test case generation method the technique of “transition cover”, summarizing it means that we identify each one of the transition and not each one of the states as the technique “state cover” applied to existing methods.

We will see in chapter 5 the results with the technique “state cover” in our case study running W & Wp Methods and after in chapter 6 we will apply “transition cover” with U-Method (our new created method for test cases generation) to analyze the differences and see which of the techniques is more efficient.

4.3 A Bipartite Graph Approach

We are going to do a brief description of “A Bipartite Graph Approach to Generate Optimal Test Sequences for Protocol Conformance Testing using the Wp-method” developed in [33]. In it is proposed a bipartite graph approach to generate optimal test sequences for protocol conformance testing. To get more information about the method we can consult [33] of the bibliography. Resuming this approach significantly reduces the length of the test sequences required for conformance testing while maintaining the same fault detection capability but after analyze this method we have tested that Wp method are more reliable and get best results.

The “W and Wp Methos” developed in [7] are applicable if the FSM is completely specified, strongly connected and minimal. Many approaches, such as T, D and UIO commented in [33], have been proposed to solve the conformance testing problem without actually get. Wp-method developed in [7] considers the W set as a union of all W_i sets. If an element in P ends at state s_i , the Wp-method only needs to concatenate the element with those elements in W_i .

Chapter 5

Adapt Bogdanov's theory (W and Wp) to state machines used to model UIs.

In this chapter 5 we have adapted “Bogdanov's theory (W and Wp)” developed in [4] and [7] respectively to state machines used to model UIs, without having to care about regions and loops. In section 5.1 shows the statechart that we work from now (TV1 Statechart). Below in 5.2 discusses the problems associated with loops and how to avoid them support us by articles published by “Beizer” in [21] and [22] explaining how we have created the flattened statechart to implement W and Wp methods with which we will work.

Working with the algorithms in section 5.3 we focus on applying W method in the different versions of our statechart while in section 5.4 we show a graphic with the obtained results for everyone of the variations of the statechart. The same we will do in section 5.5 and 5.6 respectively with Wp method.

Finally at section 5.7 of this chapter we will check the results obtained between W and Wp methods comparing the number of test cases in each of the tested models of our TV1 statechart.

Test cases generation for functional tests of user interfaces.

5.1 Describing our TV1 statechart

We are going to describe all the features of our statechart, which we can see in Figure 5.1.1. The statechart is composed of a hierarchy of three workspaces, "TV1" which encompasses everything, "On" which is composed by the functions that the device can run once it is running as the name suggests and finally the set of states composing "Working", which are the 4 channels available to the device. The statechart with which we will work in this chapter 5 has no concurrent states and as discussed in Section 5.2.1. we will remove the hierarchy to facilitate implementation of the different algorithm methods with which we will work in this chapter. As we can see we have three initial states represented by a circle, this means that by default if we have not yet turned on the device for the first time we are in "off" state. When we fire for the first time it will take us directly to the state "channel 1", it is the default state inside the workspace "Working" where all the channels are. Once we change the channel to resume it at any of the 4 at its disposal, since it can remember, so if we are on channel 3 and turn off the device, when we turn it on again it return to the "channel 3" and not to "channel 1" as happened the first time that we turned on the device.

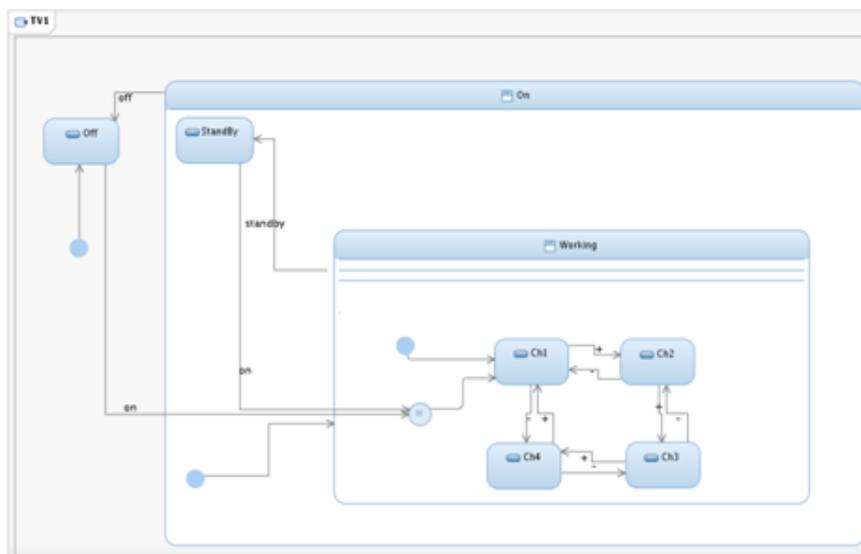


Figure 5.1.1 TV1 Statechart

From workspace "Working" channels where we vary, we can access both states "standby" and "Off" which as shown in the figure are of higher hierarchies but are connected directly to the workspace "Working". This means that while we are changing the channels we can leave the device in the states "Standby" or "Off" in both cases directly and saving in memory the channel where we were. From space "Working" if we select the state "Standby" the device stay waiting for new instructions. At this point we will be able to select the states "Off" which would lead us to the top hierarchy or "On" back to the selection of channels. The selection of channels within the workspace "Working" can be up or down, also from the state of "channel 1" can go to "channel 4" and vice versa.

5.2 Solving the problem of loops

To measure how well the program is exercised by a test suite, one or more coverage criteria are used. There are a number of coverage criteria, the main ones being:

- State Coverage: Cover every state in every state chart for basic test generation
- Condition Coverage: Cover both "true" and "false" case of if's and similar conditional constructs for basic test generation

- Switch Coverage: Cover every combination of the entry and exit transitions of all states for extended test generation
- Atomic Condition Coverage: for Boolean connectives, cover all combinations of left and right truth values for extended test generation
- Boundary Value Analysis: for comparisons of integer values, cover boundary conditions for extended test generation
- Method Coverage: Cover every method declared for extra structural traceability
- Statement Coverage: Cover every statement for extra structural traceability
- Transition All Paths: Cover all arbitrarily long distinct paths through transitions for exhaustive test generation
- Control Flow All Paths: Cover all arbitrarily long control flow paths for exhaustive test generation

Some of the coverage criteria above are connected. For instance, path coverage implies condition, statement and entry/exit coverage. Statement coverage does not imply condition coverage, as the code (in the C programming language) below shows:

```
void foo(int bar)
{
    printf("This is ");
    if (bar <= 0)
    {
        printf("not ");
    }
    printf("a positive integer.\n");
    return;
}
```

If the function `foo` were called with variable `bar` set to `-1`, statement coverage would be achieved. Condition coverage, however, would not. Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of n decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem). Techniques for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

Loop testing commented in [21], [22] and [23] is a typical spot for a semantic bug in most programming languages are the loops. They make path testing difficult due to the significantly increased number of possible paths, and they often contain bugs within the loop condition which are hard to find. A even bigger danger conceals within nested loops. It might seem likely that most loops can be tested with two checks, but a lot of bugs are not found this way. The condition of a loop has to be checked at three different times; when the loop is entered, during its execution and when the loop is left. The two borders are of special interest.

Statement coverage identifies which statements in a method or class have been executed. It is a simple metric to calculate, and a number of open source products exist that measure this level of coverage. Ultimately, the benefit of statement coverage is its ability to identify which blocks of code have not been executed. The problem with statement coverage, however, is that it does not identify bugs that arise from the control flow constructs in your source code, such as compound conditions or consecutive switch labels. This means that you easily can get

100 percent coverage and still have glaring, uncaught bugs. The following example demonstrates this. Here, the returnInput() method is made up of seven statements and has a simple requirement: its output should equal its input.

```

package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                           boolean condition2,
                           boolean condition3) {

        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x*x;
        }
        return x;
    }
}
    
```

Figure 5.2.1 Example of statement coverage.

There's an obvious bug in returnInput(). If the first or second decision evaluates true and the other evaluates false, the return value will not equal the method's input. An astute software developer will notice this right away, but the statement coverage report shows 100 percent coverage. If a manager sees 100 percent coverage, he or she may get a false sense of security, decide that testing is complete, and release the buggy code into production. Recognizing that statement coverage may not fit the bill, the developer decides to move on to a better testing technique: branch coverage.

5.2.1 Create the first version of the flattened statechart of TV1

With all the details discussed in section 5.2 on the loops we will see how to transform the original statechart in another statechart without hierarchy, with no more than one initial state and avoiding some loops with the elimination of some transitions as the highlighted "off". We will call it from now *flattened statechart* as we can see in figure 5.2.1.2. to eliminate any errors that had existed with loops avoiding the feared infinite loop which could fall to perform the tasks of testing.

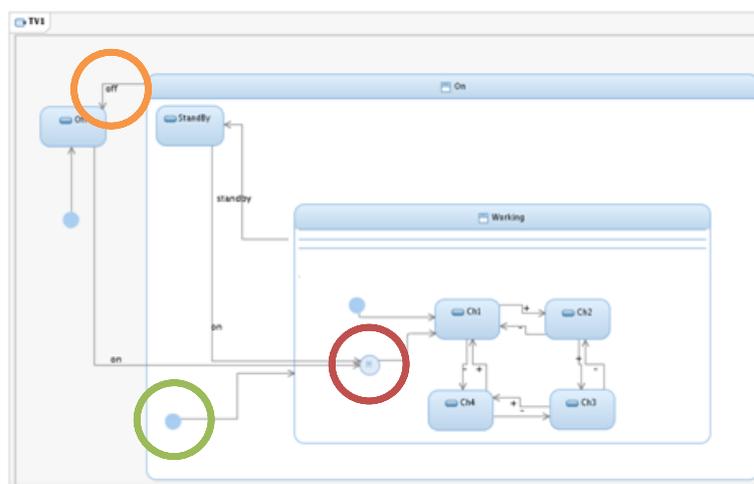


Figure 5.2.1.1. Original statechart.

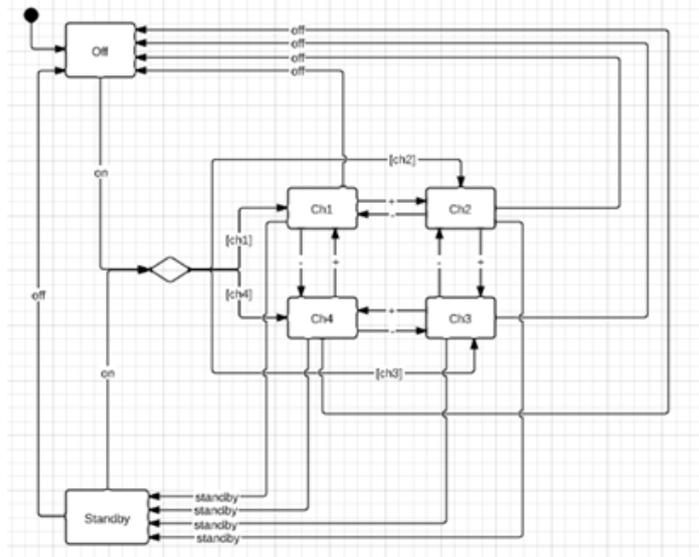


Figure 5.2.1.2. First version of Flattened statechart.

5.2.2 Positive, negative and redundant test cases

There are three types of test cases, positive, negative and redundant test cases. We are going to see an example of every one of them applied to our TV1 statechart to help us to understand W and Wp methods discussed in the following points 5.3 and 5.5. If the test case references one possible path, in the statechart which we are working, as for example “on[ch1] off” we denominated it as a positive test case, we can see it marked in green in the figure 5.2.2.1.

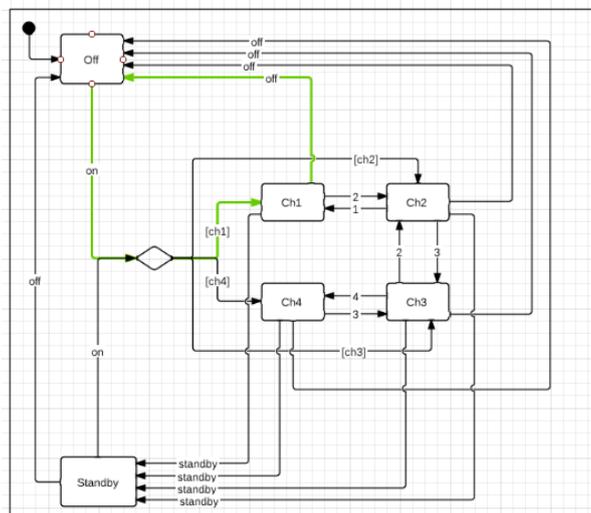


Figure 5.2.2.1 Example of a positive path test case.

When a test case cannot be identified with a path in the statechart, we call it as negative. We can see it with the negative test case “on[ch1] 1” represented in the below figure. If we try to follow the commented path “on[ch1] 1” divided in two steps, first we do “on[ch1]” and all is correct but when we try to do “1” since our new position in state Ch1 we can’t run this action and we obtain the mentioned negative test case.

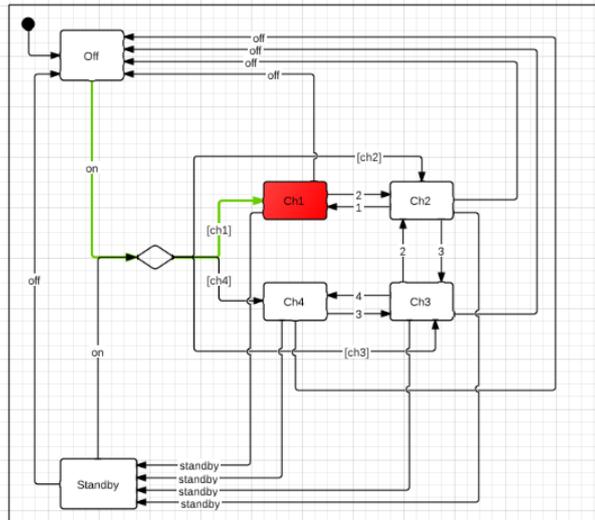


Figure 5.2.2.2. Example of a negative path test case.

Finally we obtain a redundant test case when we extend a negative test case. For example if a path p is negative, and path q is such that $q = \text{concat}(p, s)$ and s is not π_i then q is also negative, and we call q redundant. The test case “on[ch1] 1 off” is an example of redundant test case in the first version of the statechart (*Simplest model changing transitions “+” and “-” for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4*) that we will calculate in section 5.3.1. For example if $p = \text{on}[\text{ch1}] 1$ and $s = \text{off}$, the resulting $q = \text{on}[\text{ch1}] 1 \text{ off}$, this gives us a redundant test case.

As we will shown in chapter 7 of bugs, a negative test even as a test case wrong we can get to be helpful to detect errors and at times. When exist an error on the specification of the program that we are testing, is possible that a negative test case becomes positive, ie a path that should not be executed has become possible to its execution. The same applies to the redundant test cases but something more convoluted simply because it is more difficult to be in this situation. We can see it in the figure below with the test case “on[ch1] 1 off”.

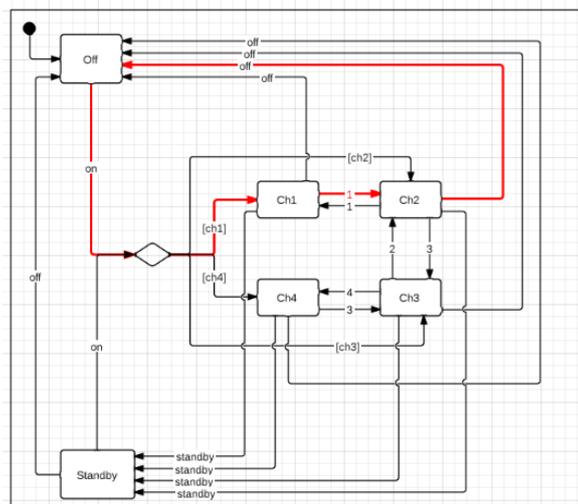


Figure 5.2.2.3. Example of a redundant path test case.

The only difference between negative and redundant test cases with the above wrong example of Figure 5.2.2.3 that has the *transition “1”* labeled in red, is that with the first (“on[ch1] 1”) we arrive to the state Ch2 and with the second (“on[ch1] 1 off”) to the state Off. In both cases the only way to detect that the program specification contains the error that

causes that the state Ch1 arrives to the state Ch2 by the *transition "1"* is through the detection that a negative or redundant test case has become positive.

5.3 Apply W method to TV1 without state hierarchy and without concurrent region

We will apply W method in subpoints of this section (since 5.3.1 to 5.3.5), after we will apply Wp (5.5) and finally in section 5.7 we will compare the results obtained for both methods. The first thing that we need to do, it's to eliminate all the substates of the original model. After that we can work with the simplest model of the statechart applying different changes, in section 5.3.1. we are going to work with the simplest model change transitions "+" and "-" for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4. After that we will work with other versions in points (since 5.3.2 to 5.3.6) checking the results obtained in section (5.4).

Step 0: modifications of the initial statechart: create a flattened statechart.

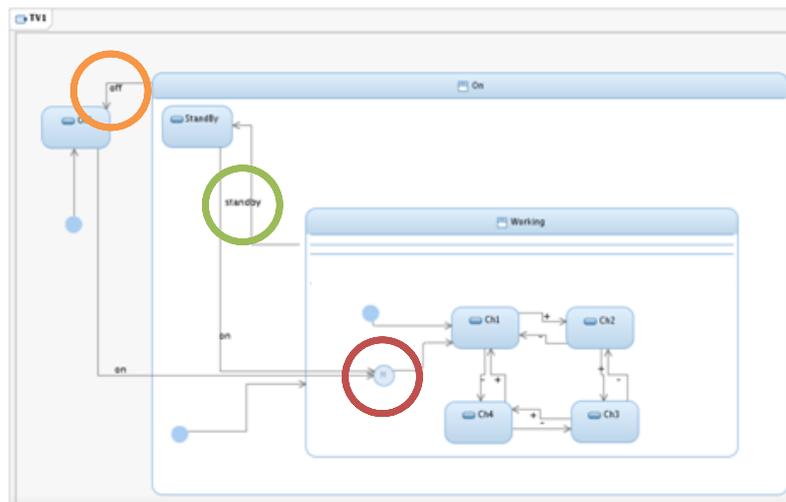


Figure 5.3.1 Original Statechart of TV1.

We will apply modifications on flattened version to calculate the different number of test cases on them. The important changes suffered by the model are the creation of every one of the transitions ("off", "standby") between the states of the channels (Ch1, Ch2, Ch3 and Ch4) and the states Off and Standby. With these transitions we managed to remove every substate of the statechart.

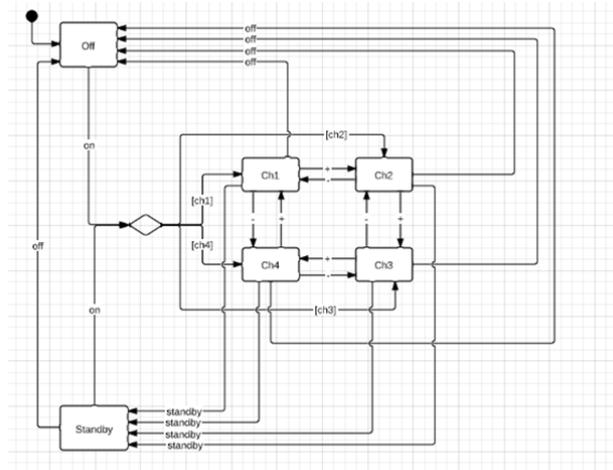


Figure 5.3.2 Flattened statechart.

5.3.1 Simplest model change transitions "+" and "-" for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.

Now we are going to calculate the test case basis "TCB" for the flattened model of TV1 with changes between the states of the channels to break the strong loop between them. Additionally we have changed transitions "+" and "-" by numbers to clarify the action of every event as we can see on Fig. 5.3.1.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$.

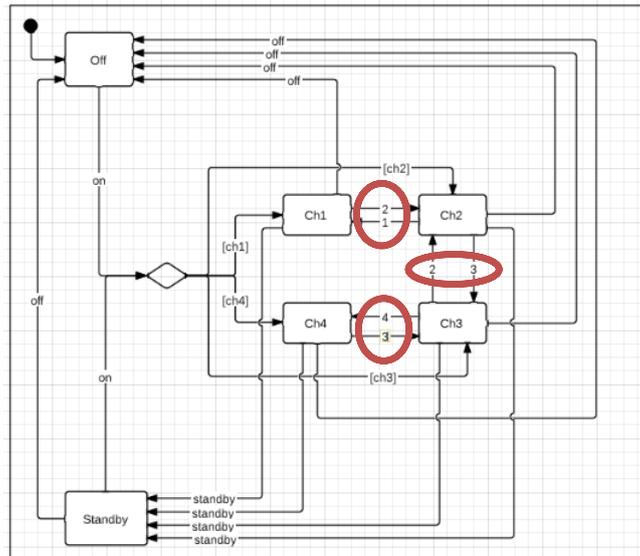


Figure 5.3.1.1. Changed model 1.

And finally we need to obtain the characterisation set (W), but this needs to be explained in several steps. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label and a set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only identify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4], off
Ch1	off	standby		2			off, standby, 2
Ch2	off	standby	1		3		off, standby, 1, 3
Ch3	off	standby		2		4	off, standby, 2, 4
Ch4	off	standby			3		off, standby, 3

Table 5.3.1.1. Transition matrix A.

The matrix that tells us how many transitions are there between pairs of states. The marginal sum by row gives the outdegree of a state, while the marginal sum by column gives its indegree (for example, outdegree(off)=4, indegree(off)=5).

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	0	0	1	1	1	1
Standby	1	0	1	1	1	1
Ch1	1	1	0	1	0	0
Ch2	1	1	1	0	1	0
Ch3	1	1	0	1	0	1
Ch4	1	1	0	0	1	0

Table 5.3.1.2. Adjacency matrix B.

If there is any diagonal of B (namely 0,0,1,0,0,0) it shows that there is a loop from Ch1 to itself of length 1. But in this case, we haven't any loop of length 1. The diagonal of B is (0, 0, 0, 0, 0, 0) so we haven't loops. To compute the number of paths with length 2 we simply compute the matrix product $B^2 = B * B$. For example there are 4 paths since Off to itself of length 2 (i.e., on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off). Variable B^2 also says that with paths of length 2 we can reach any state from any other state.

$$\text{Input} \quad \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\text{Output} \quad \begin{pmatrix} 4 & 4 & 1 & 2 & 2 & 1 \\ 4 & 4 & 2 & 3 & 3 & 2 \\ 2 & 1 & 3 & 2 & 2 & 2 \\ 3 & 2 & 2 & 4 & 3 & 3 \\ 3 & 2 & 3 & 2 & 3 & 2 \\ 2 & 1 & 2 & 3 & 3 & 3 \end{pmatrix}$$

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	4	4	1	2	2	1
Standby	4	4	2	3	3	2
Ch1	2	1	3	2	2	2
Ch2	3	2	2	4	3	3
Ch3	3	2	3	2	3	2
Ch4	2	1	2	3	3	3

Table 5.3.1.3. Adjacency matrix B*B.

The diagonal of B is (4, 4, 3, 4, 3, 3), which tell us that there are 21 loops of length 2, 4 of which start and end at Off, 4 at Standby, 3 at Ch1, etc. In the same way we can compute B^3 and B^4 and so forth. The resulting W it's, $W = \{off, standby, 1, 2, 4\}$. Now we can calculate the test cases associated at this statechart with the original formula (Equ. 1) of the W method. $T = \{ \lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby \} * \{on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby \} * \{off, standby, 1, 2, 4\} = 6 * 10 * 5 = 300$ test cases

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 5.3.1.1 W Method.

Now we can get the test cases and identify the redundant and negative test cases. As we have talked in the point 5.2.2 of this chapter an example of negative test case can be {off} and an example of redundant test case {off standby}.

T={on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, off off, 1 off, 2 off, 3 off, 4 off, standby off,

on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, off standby, 1 standby, 2 standby, 3 standby, 4 standby, standby standby, on[ch1] 1, on[ch2] 1, on[ch3] 1, on[ch4] 1, off 1, 1 1, 2 1, 3 1, 4 1, standby 1, on[ch1] 2, on[ch2] 2, on[ch3] 2, on[ch4] 2, off 2, 1 2, 2 2, 3 2, 4 2, standby 2, on[ch1] 4, on[ch2] 4, on[ch3] 4, on[ch4] 4, off 4, 1 4, 2 4, 3 4, 4 4, standby 4, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] off off, on[ch1] 1 off, on[ch1] 2 off, on[ch1] 3 off, on[ch1] 4 off, on[ch1] standby off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] off standby, on[ch1] 1 standby, on[ch1] 2 standby, on[ch1] 3 standby, on[ch1] 4 standby, on[ch1] standby standby, on[ch1] on[ch1] 1, on[ch1] on[ch2] 1, on[ch1] on[ch3] 1, on[ch1] on[ch4] 1, on[ch1] off 1, on[ch1] 1 1, on[ch1] 2 1, on[ch1] 3 1, on[ch1] 4 1, on[ch1] standby 1, on[ch1] on[ch1] 2, on[ch1] on[ch2] 2, on[ch1] on[ch3] 2, on[ch1] on[ch4] 2, on[ch1] off 2, on[ch1] 1 2, on[ch1] 2 2, on[ch1] 3 2, on[ch1] 4 2, on[ch1] standby 2, on[ch1] on[ch1] 4, on[ch1] on[ch2] 4, on[ch1] on[ch3] 4, on[ch1] on[ch4] 4, on[ch1] off 4, on[ch1] 1 4, on[ch1] 2 4, on[ch1] 3 4, on[ch1] 4 4, on[ch1] standby 4, on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] 1 off, on[ch2] 2 off, on[ch2] 3 off, on[ch2] 4 off, on[ch2] standby off, on[ch2] on[ch1] standby, on[ch2] on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] off standby, on[ch2] 1 standby, on[ch2] 2 standby, on[ch2] 3 standby, on[ch2] 4 standby, on[ch2] standby standby, on[ch2] on[ch1] 1, on[ch2] on[ch2] 1, on[ch2] on[ch3] 1, on[ch2] on[ch4] 1, on[ch2] off 1, on[ch2] 1 1, on[ch2] 2 1, on[ch2] 3 1, on[ch2] 4 1, on[ch2] standby 1, on[ch2] on[ch1] 2, on[ch2] on[ch2] 2, on[ch2] on[ch3] 2, on[ch2] on[ch4] 2, on[ch2] off 2, on[ch2] 1 2, on[ch2] 2 2, on[ch2] 3 2, on[ch2] 4 2, on[ch2] standby 2, on[ch2] on[ch1] 4, on[ch2] on[ch2] 4, on[ch2] on[ch3] 4, on[ch2] on[ch4] 4, on[ch2] off 4, on[ch2] 1 4, on[ch2] 2 4, on[ch2] 3 4, on[ch2] 4 4, on[ch2] standby 4, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] off off, on[ch3] 1 off, on[ch3] 2 off, on[ch3] 3 off, on[ch3] 4 off, on[ch3] standby off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] off standby, on[ch3] 1 standby, on[ch3] 2 standby, on[ch3] 3 standby, on[ch3] 4 standby, on[ch3] standby standby, on[ch3] on[ch1] 1, on[ch3] on[ch2] 1, on[ch3] on[ch3] 1, on[ch3] on[ch4] 1, on[ch3] off 1, on[ch3] 1 1, on[ch3] 2 1, on[ch3] 3 1, on[ch3] 4 1, on[ch3] standby 1, on[ch3] on[ch1] 2, on[ch3] on[ch2] 2, on[ch3] on[ch3] 2, on[ch3] on[ch4] 2, on[ch3] off 2, on[ch3] 1 2, on[ch3] 2 2, on[ch3] 3 2, on[ch3] 4 2, on[ch3] standby 2, on[ch3] on[ch1] 4, on[ch3] on[ch2] 4, on[ch3] on[ch3] 4, on[ch3] on[ch4] 4, on[ch3] off 4, on[ch3] 1 4, on[ch3] 2 4, on[ch3] 3 4, on[ch3] 4 4, on[ch3] standby 4, on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] off off, on[ch4] 1 off, on[ch4] 2 off, on[ch4] 3 off, on[ch4] 4 off, on[ch4] standby off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] off standby, on[ch4] 1 standby, on[ch4] 2 standby, on[ch4] 3 standby, on[ch4] 4 standby, on[ch4] standby standby, on[ch4] on[ch1] 1, on[ch4] on[ch2] 1, on[ch4] on[ch3] 1, on[ch4] on[ch4] 1, on[ch4] off 1, on[ch4] 1 1, on[ch4] 2 1, on[ch4] 3 1, on[ch4] 4 1, on[ch4] standby 1, on[ch4] on[ch1] 2, on[ch4] on[ch2] 2, on[ch4] on[ch3] 2, on[ch4] on[ch4] 2, on[ch4] off 2, on[ch4] 1 2, on[ch4] 2 2, on[ch4] 3 2, on[ch4] 4 2, on[ch4] standby 2, on[ch4] on[ch1] 4, on[ch4] on[ch2] 4, on[ch4] on[ch3] 4, on[ch4] on[ch4] 4, on[ch4] off 4, on[ch4] 1 4, on[ch4] 2 4, on[ch4] 3 4, on[ch4] 4 4, on[ch4] standby 4, on[ch1] standby on[ch1] off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby off off, on[ch1] standby 1 off, on[ch1] standby 2 off, on[ch1] standby 3 off, on[ch1] standby 4 off, on[ch1] standby standby off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby off standby, on[ch1] standby 1 standby, on[ch1] standby 2 standby, on[ch1] standby 3 standby, on[ch1] standby 4 standby, on[ch1] standby standby standby, on[ch1] standby on[ch1] 1, on[ch1] standby on[ch2] 1, on[ch1] standby on[ch3] 1, on[ch1] standby on[ch4] 1, on[ch1] standby off 1, on[ch1] standby 1 1, on[ch1] standby 2 1, on[ch1] standby 3 1, on[ch1] standby 4 1, on[ch1] standby standby 1, on[ch1] standby on[ch1] 2, on[ch1] standby on[ch2] 2, on[ch1] standby on[ch3] 2, on[ch1] standby

on[ch4] 2, on[ch1] standby off 2, on[ch1] standby 1 2, on[ch1] standby 2 2, on[ch1] standby 3 2, on[ch1] standby 4 2, on[ch1] standby standby 2, on[ch1] standby on[ch1] 4, on[ch1] standby on[ch2] 4, on[ch1] standby on[ch3] 4, on[ch1] standby on[ch4] 4, on[ch1] standby off 4, on[ch1] standby 1 4, on[ch1] standby 2 4, on[ch1] standby 3 4, on[ch1] standby 4 4, on[ch1] standby standby 4}.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that q = concat(p,s) and s is not π then q is also negative, and we call q redundant. $T_{positive} =$ on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, on[ch2] 1, on[ch3] 2, on[ch3] 4, on[ch1] 2 off, on[ch1] standby off, on[ch1] 2 standby, on[ch1] 2 1, on[ch2] 1 off, on[ch2] 3 off, on[ch2] standby off, on[ch2] 1 standby, on[ch2] 3 standby, on[ch2] 1 2, on[ch2] 3 2, on[ch2] 3 4, on[ch3] 2 off, on[ch3] 4 off, on[ch3] standby off, on[ch3] 2 standby, on[ch3] 4 standby, on[ch3] 2 1, on[ch4] 3 off, on[ch4] 3 standby, on[ch4] 3 2, on[ch4] 3 4, on[ch1] standby on[ch1] off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby on[ch2] 1, on[ch1] standby on[ch1] 2, on[ch1] standby on[ch3] 2, on[ch1] standby on[ch3] 4 = 45 positive test cases

We can see that all the paths are traversed:

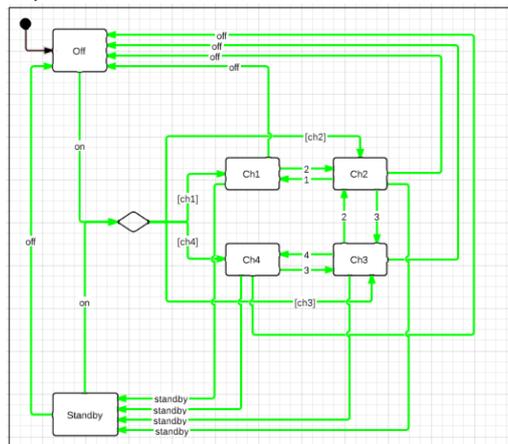


Figure 5.3.1.2. Covered transitions.

$T_{negative} =$ off off, 1 off, 2 off, 3 off, 4 off, standby off, off standby, 1 standby, 2 standby, 3 standby, 4 standby, standby standby, on[ch1] 1, on[ch3] 1, on[ch4] 1, off 1, 1 1, 2 1, 3 1, 4 1, standby 1, on[ch2] 2, on[ch4] 2, off 2, 1 2, 2 2, 3 2, 4 2, standby 2, on[ch1] 4, on[ch2] 4, on[ch4] 4, off 4, 1 4, 2 4, 3 4, 4 4, standby 4, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] off off, on[ch1] 1 off, on[ch1] 3 off, on[ch1] 4 off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] off standby, on[ch1] 1 standby, on[ch1] 3 standby, on[ch1] 4 standby, on[ch1] standby standby, on[ch1] on[ch1] 1, on[ch1] on[ch2] 1, on[ch1] on[ch3] 1, on[ch1] on[ch4] 1, on[ch1] off 1, on[ch1] 1 1, on[ch1] 3 1, on[ch1] 4 1, on[ch1] standby 1, on[ch1] on[ch1] 2, on[ch1] on[ch2] 2, on[ch1] on[ch3] 2, on[ch1] on[ch4] 2, on[ch1] off 2, on[ch1] 1 2, on[ch1] 2 2, on[ch1] 3 2, on[ch1] 4 2, on[ch1] standby 2, on[ch1] on[ch1] 4, on[ch1] on[ch2] 4, on[ch1] on[ch3] 4, on[ch1] on[ch4] 4, on[ch1] off 4, on[ch1] 1 4, on[ch1] 2 4, on[ch1] 3 4, on[ch1] 4 4, on[ch1] standby 4, on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] 2 off, on[ch2] 4 off, on[ch2] on[ch1] standby, on[ch2]

on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] off standby, on[ch2] 2 standby, on[ch2] 4 standby, on[ch2] standby standby, on[ch2] on[ch1] 1, on[ch2] on[ch2] 1, on[ch2] on[ch3] 1, on[ch2] on[ch4] 1, on[ch2] off 1, on[ch2] 1 1, on[ch2] 2 1, on[ch2] 3 1, on[ch2] 4 1, on[ch2] standby 1, on[ch2] on[ch1] 2, on[ch2] on[ch2] 2, on[ch2] on[ch3] 2, on[ch2] on[ch4] 2, on[ch2] off 2, on[ch2] 2 2, on[ch2] 4 2, on[ch2] standby 2, on[ch2] on[ch1] 4, on[ch2] on[ch2] 4, on[ch2] on[ch3] 4, on[ch2] on[ch4] 4, on[ch2] off 4, on[ch2] 1 4, on[ch2] 2 4, on[ch2] 4 4, on[ch2] standby 4, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] off off, on[ch3] 1 off, on[ch3] 3 off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] off standby, on[ch3] 1 standby, on[ch3] 3 standby, on[ch3] standby standby, on[ch3] on[ch1] 1, on[ch3] on[ch2] 1, on[ch3] on[ch3] 1, on[ch3] on[ch4] 1, on[ch3] off 1, on[ch3] 1 1, on[ch3] 3 1, on[ch3] 4 1, on[ch3] standby 1, on[ch3] on[ch1] 2, on[ch3] on[ch2] 2, on[ch3] on[ch3] 2, on[ch3] on[ch4] 2, on[ch3] off 2, on[ch3] 1 2, on[ch3] 2 2, on[ch3] 3 2, on[ch3] 4 2, on[ch3] standby 2, on[ch3] on[ch1] 4, on[ch3] on[ch2] 4, on[ch3] on[ch3] 4, on[ch3] on[ch4] 4, on[ch3] off 4, on[ch3] 1 4, on[ch3] 2 4, on[ch3] 3 4, on[ch3] 4 4, on[ch3] standby 4, on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] off off, on[ch4] 1 off, on[ch4] 2 off, on[ch4] 4 off, on[ch4] standby off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] off standby, on[ch4] 1 standby, on[ch4] 2 standby, on[ch4] 4 standby, on[ch4] standby standby, on[ch4] on[ch1] 1, on[ch4] on[ch2] 1, on[ch4] on[ch3] 1, on[ch4] on[ch4] 1, on[ch4] off 1, on[ch4] 1 1, on[ch4] 2 1, on[ch4] 3 1, on[ch4] 4 1, on[ch4] standby 1, on[ch4] on[ch1] 2, on[ch4] on[ch2] 2, on[ch4] on[ch3] 2, on[ch4] on[ch4] 2, on[ch4] off 2, on[ch4] 1 2, on[ch4] 2 2, on[ch4] 4 2, on[ch4] standby 2, on[ch4] on[ch1] 4, on[ch4] on[ch2] 4, on[ch4] on[ch3] 4, on[ch4] on[ch4] 4, on[ch4] off 4, on[ch4] 1 4, on[ch4] 2 4, on[ch4] 4 4, on[ch4] standby 4, on[ch1] standby off off, on[ch1] standby 1 off, on[ch1] standby 2 off, on[ch1] standby 3 off, on[ch1] standby 4 off, on[ch1] standby standby off, on[ch1] standby off standby, on[ch1] standby 1 standby, on[ch1] standby 2 standby, on[ch1] standby 3 standby, on[ch1] standby 4 standby, on[ch1] standby standby standby, on[ch1] standby on[ch1] 1, on[ch1] standby on[ch3] 1, on[ch1] standby on[ch4] 1, on[ch1] standby off 1, on[ch1] standby 1 1, on[ch1] standby 2 1, on[ch1] standby 3 1, on[ch1] standby 4 1, on[ch1] standby standby 1, on[ch1] standby on[ch2] 2, on[ch1] standby on[ch4] 2, on[ch1] standby off 2, on[ch1] standby 1 2, on[ch1] standby 2 2, on[ch1] standby 3 2, on[ch1] standby 4 2, on[ch1] standby standby 2, on[ch1] standby on[ch1] 4, on[ch1] standby on[ch2] 4, on[ch1] standby on[ch4] 4, on[ch1] standby off 4, on[ch1] standby 1 4, on[ch1] standby 2 4, on[ch1] standby 3 4, on[ch1] standby 4 4, on[ch1] standby standby 4 = 300 – 45(positive test cases) – 33(redundant test cases, calculated below) = 222 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that q = concat (p,s) and s is not λ then q is also negative, and we call q redundant. We can see an example:

Negative test case; p = on[ch1] 4

Redundant test case; q = on[ch1] 4 off

- q = concat (p, s) = “on[ch1] 4” + “off” = results in a redundant test case.

$T_{redundant} = \{on[ch1] 1 off, on[ch1] 1 standby, on[ch1] 1 1, on[ch1] 1 2, on[ch1] 1 4, on[ch1] 4 off, on[ch1] 4 standby, on[ch2] 2 off, on[ch2] 4 off, on[ch2] 4 standby, on[ch2] 4 1, on[ch2] 4 2, on[ch2] 4 4, on[ch3] 1 off, on[ch3] 1 standby, on[ch3] 1 1, on[ch3] 1 2, on[ch3] 1 4, on[ch4] 1 off, on[ch4] 2 off, on[ch4] 4 off, on[ch4] 1 standby, on[ch4] 2 standby, on[ch4] 4 standby, on[ch4] 1 1, on[ch4] 2 1, on[ch4] 4 1, on[ch4] 1 2, on[ch4] 2 2, on[ch4] 4 2, on[ch4] 1 4, on[ch4] 2 4, on[ch4] 4 4 = 33 redundant test cases originating of the negative test cases “on[ch1] 1, on[ch3] 1, on[ch4] 1, on[ch2] 2, on[ch4] 2, on[ch1] 4, on[ch2] 4, on[ch4] 4} = 33 redundant test cases.$

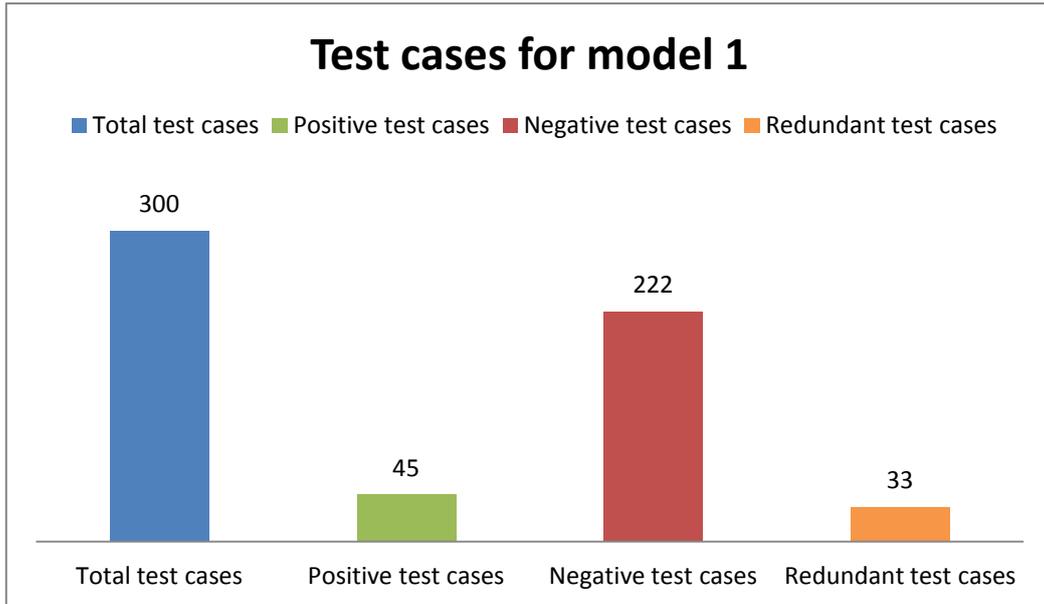


Figure 5.3.1.3 Types of test cases model 1.

5.3.2 Simplest model change transitions "+" and "-" for numbers only up direction between channels.

Again we have changed transitions "+" and "-" by numbers to clarify the action of every event as we can see on Fig. 5.3.2.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{\text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby}\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}], \text{standby}\}$.

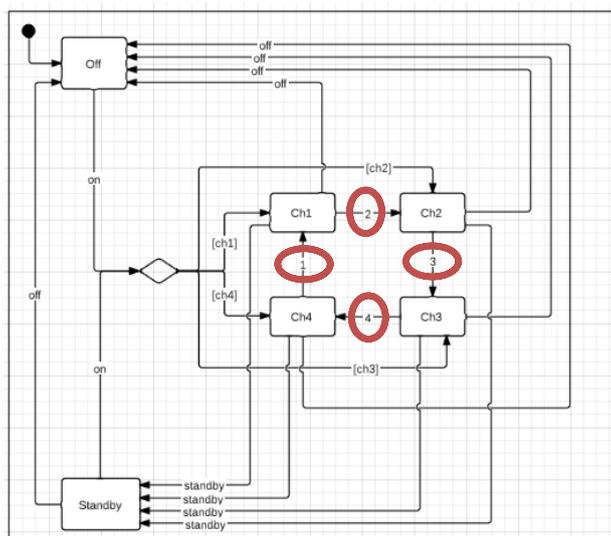


Figure 5.3.2.1 Changed model 2.

And finally we need to obtain the characterisation set (W), but this needs to be explained in several steps. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label and a set (denoted by W) allows a tester to check the state arrived

at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only identify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4], off
Ch1	off	standby		2			off, standby, 2
Ch2	off	standby			3		off, standby, 3
Ch3	off	standby				4	off, standby, 4
Ch4	off	standby	1				off, standby, 1

Table 5.3.2.1 Transition matrix A.

The matrix that tells us how many transitions are there between pairs of states. The marginal sum by row gives the outdegree of a state, while the marginal sum by column gives its indegree.

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	0	0	1	1	1	1
Standby	1	0	1	1	1	1
Ch1	1	1	0	1	0	0
Ch2	1	1	0	0	1	0
Ch3	1	1	0	0	0	1
Ch4	1	1	1	0	0	0

Table 5.3.2.2. Adjacency matrix B.

To compute the number of paths with length 2 we simply compute the matrix product $B^2 = B * B$. B^2 also says that with paths of length 2 we can reach any state from any other state.

$$\text{Input } \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{Output } \begin{pmatrix} 4 & 4 & 1 & 1 & 1 & 1 \\ 4 & 4 & 2 & 2 & 2 & 2 \\ 2 & 1 & 2 & 2 & 3 & 2 \\ 2 & 1 & 2 & 2 & 2 & 3 \\ 2 & 1 & 3 & 2 & 2 & 2 \\ 2 & 1 & 2 & 3 & 2 & 2 \end{pmatrix}$$

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	4	4	1	1	1	1
Standby	4	4	2	2	2	2
Ch1	2	1	2	2	3	2
Ch2	2	1	2	2	2	3
Ch3	2	1	3	2	2	2
Ch4	2	1	2	3	2	2

Table 5.3.2.3. Adjacency matrix B*B.

The diagonal of B is (4, 4, 2, 2, 2, 2), which tell us that there are 16 loops of length 2, 4 of which start and end at Off, 4 at Standby, 2 at Ch1, etc. In the same way we can compute B^3 and B^4 and so forth. The resulting W it's, $W = \{\text{off, standby, 1, 2, 3, 4}\}$. Now we can calculate the test cases associated at this statechart with the original formula (Equ. 5.3.2.) of the W method. $T = \{\lambda, \text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{on[ch1] standby}\} * \{\text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{off, 1, 2, 3, 4, standby}\} * \{\text{off, standby, 1, 2, 3, 4}\} = 6 * 10 * 6 = 360$ test cases.

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 5.3.2.1 W method.

Now we can get the test cases and identify the redundant and negative test cases. As we have talked in the point 5.2.2. of this chapter an example of negative test case can be: {off} and an example of redundant test case; {off standby}.

$$T = \{\lambda, \text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{on[ch1] standby}\} * \{\text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{off, 1, 2, 3, 4, standby}\} * \{\text{off, standby, 1, 2, 3, 4}\}$$

T={ on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, off off, 1 off, 2 off, 3 off, 4 off, standby off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, off standby, 1 standby, 2 standby, 3 standby, 4 standby, standby standby, on[ch1] 1, on[ch2] 1, on[ch3] 1, on[ch4] 1, off 1, 1 1, 2 1, 3 1, 4 1, standby 1, on[ch1] 2, on[ch2] 2, on[ch3] 2, on[ch4] 2, off 2, 1 2, 2 2, 3 2, 4 2, standby 2, on[ch1] 3, on[ch2] 3, on[ch3] 3, on[ch4] 3, off 3, 1 3, 2 3, 3 3, 4 3, standby 3, on[ch1] 4, on[ch2] 4, on[ch3] 4, on[ch4] 4, off 4, 1 4, 2 4, 3 4, 4 4, standby 4, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] off off, on[ch1] 1 off, on[ch1] 2 off, on[ch1] 3 off, on[ch1] 4 off, on[ch1] standby off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] off standby, on[ch1] 1 standby, on[ch1] 2 standby, on[ch1] 3 standby, on[ch1] 4 standby, on[ch1] standby standby, on[ch1] on[ch1] 1, on[ch1] on[ch2] 1, on[ch1] on[ch3] 1, on[ch1] on[ch4] 1, on[ch1] off 1, on[ch1] 1 1, on[ch1] 2 1, on[ch1] 3 1, on[ch1] 4 1, on[ch1] standby 1, on[ch1] on[ch1] 2, on[ch1] on[ch2] 2, on[ch1] on[ch3] 2, on[ch1] on[ch4] 2, on[ch1] off 2, on[ch1] 1 2, on[ch1] 2 2, on[ch1] 3 2, on[ch1] 4 2, on[ch1] standby 2, on[ch1] on[ch1] 3, on[ch1] on[ch2] 3, on[ch1] on[ch3] 3, on[ch1] on[ch4] 3, on[ch1] off 3, on[ch1] 1 3, on[ch1] 2 3, on[ch1] 3 3, on[ch1] 4 3, on[ch1] standby 3, on[ch1] on[ch1] 4, on[ch1] on[ch2] 4, on[ch1] on[ch3] 4, on[ch1] on[ch4] 4, on[ch1] off 4, on[ch1] 1 4, on[ch1] 2 4, on[ch1] 3 4, on[ch1] 4 4, on[ch1] standby 4, on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] 1 off, on[ch2] 2 off, on[ch2] 3 off, on[ch2] 4 off, on[ch2] standby off, on[ch2] on[ch1] standby, on[ch2] on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] off standby, on[ch2] 1 standby, on[ch2] 2 standby, on[ch2] 3 standby, on[ch2] 4 standby, on[ch2] standby standby, on[ch2] on[ch1] 1, on[ch2] on[ch2] 1, on[ch2] on[ch3] 1, on[ch2] on[ch4] 1, on[ch2] off 1, on[ch2] 1 1, on[ch2] 2 1, on[ch2] 3 1, on[ch2] 4 1, on[ch2] standby 1, on[ch2] on[ch1] 2, on[ch2] on[ch2] 2, on[ch2] on[ch3] 2, on[ch2] on[ch4] 2, on[ch2] off 2, on[ch2] 1 2, on[ch2] 2 2, on[ch2] 3 2, on[ch2] 4 2, on[ch2] standby 2, on[ch2] on[ch1] 3, on[ch2] on[ch2] 3, on[ch2] on[ch3] 3, on[ch2] on[ch4] 3, on[ch2] off 3, on[ch2] 1 3, on[ch2] 2 3, on[ch2] 3 3, on[ch2] 4 3, on[ch2] standby 3, on[ch2] on[ch1] 4, on[ch2] on[ch2] 4, on[ch2] on[ch3] 4, on[ch2] on[ch4] 4, on[ch2] off 4, on[ch2] 1 4, on[ch2] 2 4, on[ch2] 3 4, on[ch2] 4 4, on[ch2] standby 4, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] off off, on[ch3] 1 off, on[ch3] 2 off, on[ch3] 3 off, on[ch3] 4 off, on[ch3] standby off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] off standby, on[ch3] 1 standby, on[ch3] 2 standby, on[ch3] 3 standby, on[ch3] 4 standby, on[ch3] standby standby, on[ch3] on[ch1] 1, on[ch3] on[ch2] 1, on[ch3] on[ch3] 1, on[ch3] on[ch4] 1, on[ch3] off 1, on[ch3] 1 1, on[ch3] 2 1,

on[ch3] 3 1, on[ch3] 4 1, on[ch3] standby 1, on[ch3] on[ch1] 2, on[ch3] on[ch2] 2, on[ch3] on[ch3] 2, on[ch3] on[ch4] 2, on[ch3] off 2, on[ch3] 1 2, on[ch3] 2 2, on[ch3] 3 2, on[ch3] 4 2, on[ch3] standby 2, on[ch3] on[ch1] 3, on[ch3] on[ch2] 3, on[ch3] on[ch3] 3, on[ch3] on[ch4] 3, on[ch3] off 3, on[ch3] 1 3, on[ch3] 2 3, on[ch3] 3 3, on[ch3] 4 3, on[ch3] standby 3, on[ch3] on[ch1] 4, on[ch3] on[ch2] 4, on[ch3] on[ch3] 4, on[ch3] on[ch4] 4, on[ch3] off 4, on[ch3] 1 4, on[ch3] 2 4, on[ch3] 3 4, on[ch3] 4 4, on[ch3] standby 4, on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] off off, on[ch4] 1 off, on[ch4] 2 off, on[ch4] 3 off, on[ch4] 4 off, on[ch4] standby off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] off standby, on[ch4] 1 standby, on[ch4] 2 standby, on[ch4] 3 standby, on[ch4] 4 standby, on[ch4] standby standby, on[ch4] on[ch1] 1, on[ch4] on[ch2] 1, on[ch4] on[ch3] 1, on[ch4] on[ch4] 1, on[ch4] off 1, on[ch4] 1 1, on[ch4] 2 1, on[ch4] 3 1, on[ch4] 4 1, on[ch4] standby 1, on[ch4] on[ch1] 2, on[ch4] on[ch2] 2, on[ch4] on[ch3] 2, on[ch4] on[ch4] 2, on[ch4] off 2, on[ch4] 1 2, on[ch4] 2 2, on[ch4] 3 2, on[ch4] 4 2, on[ch4] standby 2, on[ch4] on[ch1] 3, on[ch4] on[ch2] 3, on[ch4] on[ch3] 3, on[ch4] on[ch4] 3, on[ch4] off 3, on[ch4] 1 3, on[ch4] 2 3, on[ch4] 3 3, on[ch4] 4 3, on[ch4] standby 3, on[ch4] on[ch1] 4, on[ch4] on[ch2] 4, on[ch4] on[ch3] 4, on[ch4] on[ch4] 4, on[ch4] off 4, on[ch4] 1 4, on[ch4] 2 4, on[ch4] 3 4, on[ch4] 4 4, on[ch4] standby 4, on[ch1] standby on[ch1] off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby off off, on[ch1] standby 1 off, on[ch1] standby 2 off, on[ch1] standby 3 off, on[ch1] standby 4 off, on[ch1] standby standby off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby off standby, on[ch1] standby 1 standby, on[ch1] standby 2 standby, on[ch1] standby 3 standby, on[ch1] standby 4 standby, on[ch1] standby standby standby, on[ch1] standby on[ch1] 1, on[ch1] standby on[ch2] 1, on[ch1] standby on[ch3] 1, on[ch1] standby on[ch4] 1, on[ch1] standby off 1, on[ch1] standby 1 1, on[ch1] standby 2 1, on[ch1] standby 3 1, on[ch1] standby 4 1, on[ch1] standby standby 1, on[ch1] standby on[ch1] 2, on[ch1] standby on[ch2] 2, on[ch1] standby on[ch3] 2, on[ch1] standby on[ch4] 2, on[ch1] standby off 2, on[ch1] standby 1 2, on[ch1] standby 2 2, on[ch1] standby 3 2, on[ch1] standby 4 2, on[ch1] standby standby 2, on[ch1] standby on[ch1] 3, on[ch1] standby on[ch2] 3, on[ch1] standby on[ch3] 3, on[ch1] standby on[ch4] 3, on[ch1] standby off 3, on[ch1] standby 1 3, on[ch1] standby 2 3, on[ch1] standby 3 3, on[ch1] standby 4 3, on[ch1] standby standby 3, on[ch1] standby on[ch1] 4, on[ch1] standby on[ch2] 4, on[ch1] standby on[ch3] 4, on[ch1] standby on[ch4] 4, on[ch1] standby off 4, on[ch1] standby 1 4, on[ch1] standby 2 4, on[ch1] standby 3 4, on[ch1] standby 4 4, on[ch1] standby standby 4}.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not π then q is also negative, and we call q redundant. The set of positive test cases are $T_{\text{positive}} = \{\text{on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, on[ch4] 1, on[ch1] 2, on[ch2] 3, on[ch3] 4, on[ch1] 2 off, on[ch1] standby off, on[ch1] 2 standby, on[ch1] 2 3, on[ch2] 3 off, on[ch2] standby off, on[ch2] 3 standby, on[ch2] 3 4, on[ch3] 4 off, on[ch3] standby off, on[ch3] 4 standby, on[ch3] 4 1, on[ch4] 1 off, on[ch4] standby off, on[ch4] 1 standby, on[ch4] 1 2, on[ch1] standby on[ch1] off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby on[ch4] 1, on[ch1] standby on[ch1] 2, on[ch1] standby on[ch2] 3, on[ch1] standby on[ch3] 4\} = 40$ positive test cases.

$T_{negative} = \{on[ch1] 1, on[ch2] 1, on[ch3] 1, on[ch2] 2, on[ch3] 2, on[ch4] 2, on[ch1] 3, on[ch3] 3, on[ch4] 3, on[ch1] 4, on[ch2] 4, on[ch4] 4, on[ch1] off standby, on[ch1] off 1, on[ch1] 2 1, on[ch1] standby 1, on[ch1] off 2, on[ch1] 2 2, on[ch1] standby 2, on[ch1] off 3, on[ch1] standby 3, on[ch1] off 4, on[ch1] 2 4, on[ch1] standby 4, on[ch2] off standby, on[ch2] off 1, on[ch2] 3 1, on[ch2] standby 1, on[ch2] off 2, on[ch2] 3 2, on[ch2] standby 2, on[ch2] off 3, on[ch2] 3 3, on[ch2] standby 3, on[ch2] off 4, on[ch2] standby 4, on[ch3] off standby, on[ch3] off 1, on[ch3] standby 1, on[ch3] off 2, on[ch3] 4 2, on[ch3] standby 2, on[ch3] off 3, on[ch3] 4 3, on[ch3] standby 3, on[ch3] off 4, on[ch3] 4 4, on[ch3] standby 4, on[ch4] off standby, on[ch4] off 1, on[ch4] 1 1, on[ch4] standby 1, on[ch4] off 2, on[ch4] standby 2, on[ch4] off 3, on[ch4] 1 3, on[ch4] standby 3, on[ch4] off 4, on[ch4] 1 4, on[ch4] standby 4, on[ch1] standby on[ch1] 1, on[ch1] standby on[ch2] 1, on[ch1] standby on[ch3] 1, on[ch1] standby off 1, on[ch1] standby on[ch2] 2, on[ch1] standby on[ch3] 2, on[ch1] standby on[ch4] 2, on[ch1] standby off 2, on[ch1] standby 1 2, on[ch1] standby 2 2, on[ch1] standby on[ch1] 3, on[ch1] standby on[ch3] 3, on[ch1] standby on[ch4] 3, on[ch1] standby off 3, on[ch1] standby on[ch1] 4, on[ch1] standby on[ch2] 4, on[ch1] standby on[ch4] 4, on[ch1] standby off 4\} = 360 calculated test cases – 40 (positive test cases) – 242 (redundant test cases, calculated below) = 78 negative test cases.$

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that $q=concat(p,s)$ and s is not λ then q is also negative, and we call q redundant. We can see an example:

Negative test case; p = on[ch1] 4

Redundant test case; q = on[ch1] 4 off

- o q = concat (p, s) = “on[ch1] 4” + “off” = results in a redundant test case.

$T_{redundant} = \{off off, 1 off, 2 off, 3 off, 4 off, standby off, off standby, 1 standby, 2 standby, 3 standby, 4 standby, standby standby, off 1, 1 1, 2 1, 3 1, 4 1, standby 1, off 2, 1 2, 2 2, 3 2, 4 2, standby 2, off 3, 1 3, 2 3, 3 3, 4 3, standby 3, off 4, 1 4, 2 4, 3 4, 4 4, standby 4, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] off off, on[ch1] 1 off, on[ch1] 3 off, on[ch1] 4 off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] 1 standby, on[ch1] 3 standby, on[ch1] 4 standby, on[ch1] standby standby, on[ch1] on[ch1] 1, on[ch1] on[ch2] 1, on[ch1] on[ch3] 1, on[ch1] on[ch4] 1, on[ch1] 1 1, , on[ch1] 3 1, on[ch1] 4 1, on[ch1] on[ch1] 2, on[ch1] on[ch2] 2, on[ch1] on[ch3] 2, on[ch1] on[ch4] 2, on[ch1] 1 2, on[ch1] 3 2, on[ch1] 4 2, on[ch1] on[ch1] 3, on[ch1] on[ch2] 3, on[ch1] on[ch3] 3, on[ch1] on[ch4] 3, on[ch1] 1 3, on[ch1] 3 3, on[ch1] 4 3, on[ch1] on[ch1] 4, on[ch1] on[ch2] 4, on[ch1] on[ch3] 4, on[ch1] on[ch4] 4, on[ch1] 1 4, on[ch1] 3 4, on[ch1] 4 4, on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] 1 off, on[ch2] 2 off, on[ch2] 4 off, on[ch2] on[ch1] standby, on[ch2] on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] 1 standby, on[ch2] 2 standby, on[ch2] 4 standby, on[ch2] standby standby, on[ch2] on[ch1] 1, on[ch2] on[ch2] 1, on[ch2] on[ch3] 1, on[ch2] on[ch4] 1, on[ch2] 1 1, on[ch2] 2 1, on[ch2] 4 1, on[ch2] on[ch1] 2, on[ch2] on[ch2] 2, on[ch2] on[ch3] 2, on[ch2] on[ch4] 2, on[ch2] 1 2, on[ch2] 2 2, on[ch2] 4 2, on[ch2] on[ch1] 3, on[ch2] on[ch2] 3, on[ch2] on[ch3] 3, on[ch2] on[ch4] 3, on[ch2] 1 3, on[ch2] 2 3, on[ch2] 4 3, on[ch2] on[ch1] 4, on[ch2] on[ch2] 4, on[ch2] on[ch3] 4, on[ch2] on[ch4] 4, on[ch2] 1 4, on[ch2] 2 4, on[ch2] 4 4, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] off off, on[ch3] 1 off, on[ch3] 2 off, on[ch3] 3 off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] 1 standby, on[ch3] 2 standby, on[ch3] 3 standby, on[ch3] standby standby, on[ch3] on[ch1] 1, on[ch3] on[ch2] 1, on[ch3] on[ch3] 1, on[ch3] on[ch4] 1, on[ch3] 1 1, on[ch3] 2 1, on[ch3] 3 1, on[ch3] on[ch1] 2, on[ch3] on[ch2] 2, on[ch3] on[ch3] 2, on[ch3] on[ch4] 2, on[ch3] 1 2, on[ch3] 2 2, on[ch3] 3 2, on[ch3] on[ch1] 3, on[ch3] on[ch2] 3,$

on[ch3] on[ch3] 3, on[ch3] on[ch4] 3, on[ch3] 1 3, on[ch3] 2 3, on[ch3] 3 3, on[ch3] on[ch1] 4, on[ch3] on[ch2] 4, on[ch3] on[ch3] 4, on[ch3] on[ch4] 4, on[ch3] 1 4, on[ch3] 2 4, on[ch3] 3 4, on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] off off, on[ch4] 2 off, on[ch4] 3 off, on[ch4] 4 off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] 2 standby, on[ch4] 3 standby, on[ch4] 4 standby, on[ch4] standby standby, on[ch4] on[ch1] 1, on[ch4] on[ch2] 1, on[ch4] on[ch3] 1, on[ch4] on[ch4] 1, on[ch4] 2 1, on[ch4] 3 1, on[ch4] 4 1, on[ch4] on[ch1] 2, on[ch4] on[ch2] 2, on[ch4] on[ch3] 2, on[ch4] on[ch4] 2, on[ch4] 2 2, on[ch4] 3 2, on[ch4] 4 2, , on[ch4] on[ch1] 3, on[ch4] on[ch2] 3, on[ch4] on[ch3] 3, on[ch4] on[ch4] 3, on[ch4] 2 3, on[ch4] 3 3, on[ch4] 4 3, on[ch4] on[ch1] 4, on[ch4] on[ch2] 4, on[ch4] on[ch3] 4, on[ch4] on[ch4] 4, on[ch4] 2 4, on[ch4] 3 4, on[ch4] 4 4, on[ch1] standby off off, on[ch1] standby 1 off, on[ch1] standby 2 off, on[ch1] standby 3 off, on[ch1] standby 4 off, on[ch1] standby standby off, on[ch1] standby off standby, on[ch1] standby 1 standby, on[ch1] standby 2 standby, on[ch1] standby 3 standby, on[ch1] standby 4 standby, on[ch1] standby standby standby, on[ch1] standby 1 1, on[ch1] standby 2 1, on[ch1] standby 3 1, on[ch1] standby 4 1, on[ch1] standby standby 1, on[ch1] standby 3 2, on[ch1] standby 4 2, on[ch1] standby standby 2, on[ch1] standby 1 3, on[ch1] standby 2 3, on[ch1] standby 3 3, on[ch1] standby 4 3, on[ch1] standby standby 3, on[ch1] standby 1 4, on[ch1] standby 2 4, on[ch1] standby 3 4, on[ch1] standby 4 4, on[ch1] standby standby 4} = 242 redundant test cases originating of the negative test cases “on[ch1] 1, on[ch3] 1, on[ch4] 1, on[ch2] 2, on[ch4] 2, on[ch1] 4, on[ch2] 4, on[ch4] 4.”

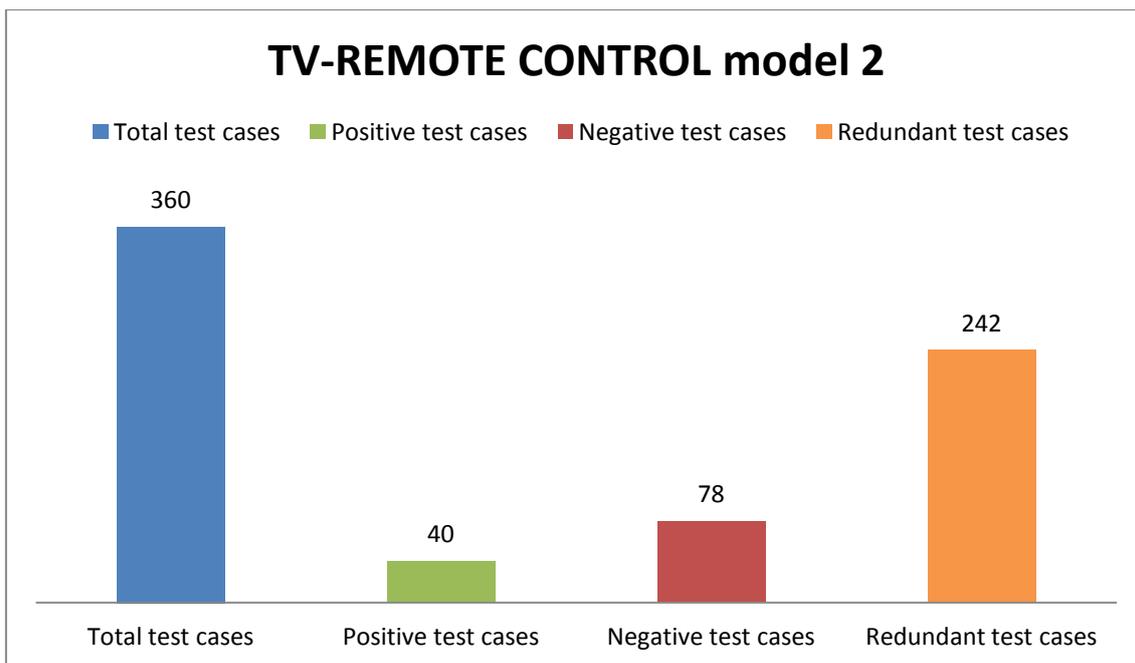


Figure 5.3.2.2. Types of test cases on model 2.

5.3.3 Simplest model with transitions "+" and "-" with a broken transition "-" between ch1 and ch4.

With this model we maintain the transitions "+" and "-" but we have removed the transition "-" since channel 1 to channel 4 to break the loop. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{\text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{off}, +, -, \text{standby}\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, \text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{on[ch1] standby}\}$.

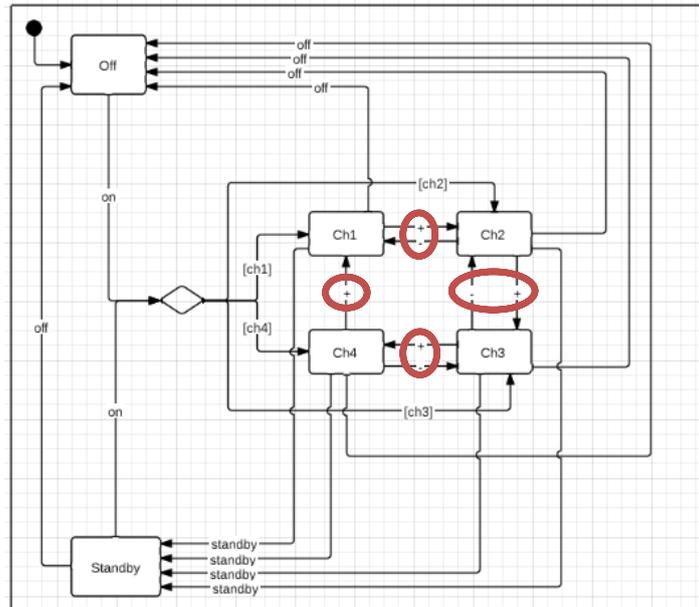


Figure 5.3.3.1. Changed model 3.

And finally we need to obtain the characterisation set (W), but this needs to be explained in several steps. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label and a set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only identify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4],off
Ch1	off	standby		+			off, standby, +*
Ch2	off	standby	-		+		off, standby, +*, -
Ch3	off	standby		-		+	off, standby, +*, -, - -
Ch4	off	standby	+		-		off, standby, +*, -, - -, - - -

Table 5.3.3.1 Transition matrix A.

The matrix that tells us how many transitions are there between pairs of states. The marginal sum by row gives the outdegree of a state, while the marginal sum by column gives its indegree.

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	0	0	1	1	1	1
Standby	1	0	1	1	1	1
Ch1	1	1	0	1	0	0
Ch2	1	1	1	0	1	0
Ch3	1	1	0	1	0	1
Ch4	1	1	1	0	1	0

Table 5.3.3.2. Adjacency matrix B.

To compute the number of paths with length 2 we simply compute the matrix product $B^2 = B * B$. B^2 also says that with paths of length 2 we can reach any state from any other state.

$$\begin{matrix} \text{Input} & \left(\begin{matrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{matrix} \right) & \left(\begin{matrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{matrix} \right) \\ \text{Output} & \left(\begin{matrix} 4 & 4 & 2 & 2 & 2 & 1 \\ 4 & 4 & 3 & 3 & 3 & 2 \\ 2 & 1 & 3 & 2 & 3 & 2 \\ 3 & 2 & 2 & 4 & 2 & 3 \\ 3 & 2 & 4 & 2 & 4 & 2 \\ 3 & 2 & 2 & 4 & 2 & 3 \end{matrix} \right) & \end{matrix}$$

	Off	Standby	Ch1	Ch2	Ch3	Ch4
Off	4	4	2	2	2	1
Standby	4	4	3	3	3	2
Ch1	2	1	3	2	3	2
Ch2	3	2	2	4	2	3
Ch3	3	2	4	2	4	2
Ch4	3	2	2	4	2	3

Table 5.3.3.3. Adjacency matrix B^2 .

The diagonal of B is (4, 4, 3, 4, 4, 3), which tell us that there are 22 loops of length 2, 4 of which start and end at Off, 4 at Standby, 3 at Ch1, etc. In the same way we can compute B^3 and B^4 and so forth. The resulting W it's, $W = \{ \text{off, standby, -, -, -} \}$. Now we can calculate the test cases associated at this statechart with the original formula (Equ. 5.3.3.) of the W method. $T = \{ \lambda, \text{on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby} \} * \{ \text{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby} \} * \{ \text{off, standby, -, -, -} \} = 6 * 8 * 5 = 240$ test cases.

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 5.3.3.1 W method.

Now we can get the test cases and identify the redundant and negative test cases. As we have talked in the point 5.2.2. of this chapter an example of negative test case can be: {off} and an example of redundant test case; {off standby}.

$$T = \{ \lambda, \text{on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby} \} * \{ \text{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby} \} * \{ \text{off, standby, -, -, -} \}$$

T={ on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, off off, + off, - off, standby off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, off standby, + standby, - standby, standby standby, on[ch1] -, on[ch2] -, on[ch3] -, on[ch4] -, off -, + -, -, standby -, on[ch1] --, on[ch2] --, on[ch3] --, on[ch4] --, off --, + --, --, standby --, on[ch1] ---, on[ch2] ---, on[ch3] ---, on[ch4] ---, off ---, + ---, ---, standby ---, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] off off, on[ch1] + off, on[ch1] - off, on[ch1] standby off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] off standby, on[ch1] + standby, on[ch1] - standby, on[ch1] standby standby, on[ch1] on[ch1] -, on[ch1] on[ch2] -, on[ch1] on[ch3] -, on[ch1] on[ch4] -, on[ch1] off -, on[ch1] + -, on[ch1] --, on[ch1] standby -, on[ch1] on[ch1] --, on[ch1] on[ch2] --, on[ch1] on[ch3] --, on[ch1] on[ch4] --, on[ch1] off --, on[ch1] + --, on[ch1] ---, on[ch1] standby ---, on[ch1] on[ch1] ---, on[ch1] on[ch2] ---, on[ch1] on[ch3] ---, on[ch1] on[ch4] ---}

on[ch4] - - -, on[ch1] off - - -, on[ch1] + - - -, on[ch1] - - - -, on[ch1] standby - - - , on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] + off, on[ch2] - off, on[ch2] standby off, on[ch2] on[ch1] standby, on[ch2] on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] off standby, on[ch2] + standby, on[ch2] - standby, on[ch2] standby standby, on[ch2] on[ch1] -, on[ch2] on[ch2] -, on[ch2] on[ch3] -, on[ch2] on[ch4] -, on[ch2] off -, on[ch2] + -, on[ch2] - -, on[ch2] standby -, on[ch2] on[ch1] - -, on[ch2] on[ch2] - -, on[ch2] on[ch3] - -, on[ch2] on[ch4] - -, on[ch2] off - -, on[ch2] + - -, on[ch2] - - -, on[ch2] standby - - -, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] off off, on[ch3] + off, on[ch3] - off, on[ch3] standby off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] off standby, on[ch3] + standby, on[ch3] - standby, on[ch3] standby standby, on[ch3] on[ch1] -, on[ch3] on[ch2] -, on[ch3] on[ch3] -, on[ch3] on[ch4] -, on[ch3] off -, on[ch3] + -, on[ch3] - -, on[ch3] standby -, on[ch3] on[ch1] - -, on[ch3] on[ch2] - -, on[ch3] on[ch3] - -, on[ch3] on[ch4] - -, on[ch3] off - -, on[ch3] + - -, on[ch3] - - -, on[ch3] standby - -, on[ch3] on[ch1] - - -, on[ch3] on[ch2] - - -, on[ch3] on[ch3] - - -, on[ch3] on[ch4] - - -, on[ch3] off - - -, on[ch3] + - - -, on[ch3] - - - -, on[ch3] standby - - - , on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] off off, on[ch4] + off, on[ch4] - off, on[ch4] standby off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] off standby, on[ch4] + standby, on[ch4] - standby, on[ch4] standby standby, on[ch4] on[ch1] -, on[ch4] on[ch2] -, on[ch4] on[ch3] -, on[ch4] on[ch4] -, on[ch4] off -, on[ch4] + -, on[ch4] - -, on[ch4] standby -, on[ch4] on[ch1] - -, on[ch4] on[ch2] - -, on[ch4] on[ch3] - -, on[ch4] on[ch4] - -, on[ch4] off - -, on[ch4] + - -, on[ch4] - - -, on[ch4] standby - - , on[ch4] on[ch1] - - -, on[ch4] on[ch2] - - -, on[ch4] on[ch3] - - -, on[ch4] on[ch4] - - -, on[ch4] off - - -, on[ch4] + - - -, on[ch4] - - - -, on[ch4] standby - - - , on[ch1] standby on[ch1] standby off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby off off, on[ch1] standby + off, on[ch1] standby - off, on[ch1] standby standby off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby off standby, on[ch1] standby + standby, on[ch1] standby - standby, on[ch1] standby standby standby, on[ch1] standby on[ch1] -, on[ch1] standby on[ch2] -, on[ch1] standby on[ch3] -, on[ch1] standby on[ch4] -, on[ch1] standby off -, on[ch1] standby + -, on[ch1] standby - -, on[ch1] standby standby -, on[ch1] standby on[ch1] - -, on[ch1] standby on[ch2] - -, on[ch1] standby on[ch3] - -, on[ch1] standby on[ch4] - -, on[ch1] standby off - -, on[ch1] standby + - -, on[ch1] standby - - -, on[ch1] standby standby - - , on[ch1] standby on[ch1] - - -, on[ch1] standby on[ch2] - - -, on[ch1] standby on[ch3] - - -, on[ch1] standby on[ch4] - - -, on[ch1] standby off - - -, on[ch1] standby + - - -, on[ch1] standby - - - -, on[ch1] standby standby - - - }.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not π then q is also negative, and we call q redundant. The set of positive test cases are $T_{positive} = \{$ on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, on[ch2] -, on[ch3] -, on[ch4] -, on[ch3] - -, on[ch4] - -, on[ch4] - - -, on[ch1] + off, on[ch1] standby off, on[ch1] + standby, on[ch1] + -, on[ch2] + off, on[ch2] - off, on[ch2] standby off, on[ch2] + standby, on[ch2] - standby, on[ch2] + -, on[ch2] + - -, on[ch3] + off, on[ch3] - off, on[ch3] standby off, on[ch3] + standby, on[ch3] - standby, on[ch3] + -, on[ch3] - -, on[ch3] + - -, on[ch3] + - - -, on[ch4] + off, on[ch4] - off, on[ch4] standby off, on[ch4] + standby, on[ch4] - standby, on[ch4] - -, on[ch4] - - -, on[ch4] - - - -, on[ch1] standby on[ch1] standby off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby

on[ch4] off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby on[ch2] -, on[ch1] standby on[ch3] -, on[ch1] standby on[ch4] -, on[ch1] standby on[ch3] - -, on[ch1] standby on[ch4] - -, on[ch1] standby on[ch4] - - - } = 56 positive test cases.

$T_{negative} = \{ \text{on[ch1] -, on[ch2] - -, on[ch3] - - -, on[ch1] off off, on[ch1] off standby, on[ch1] standby standby, on[ch1] off -, on[ch1] standby -, on[ch1] + - -, on[ch2] off standby, on[ch2] off -, on[ch2] - -, on[ch2] standby -, on[ch2] + - - -, on[ch3] off off, on[ch3] off standby, on[ch3] standby standby, on[ch3] off -, on[ch3] standby -, on[ch3] - - -, on[ch4] off off, on[ch4] off standby, on[ch4] standby standby, on[ch4] off -, on[ch4] + -, on[ch4] standby -, on[ch1] standby off standby, on[ch1] standby on[ch1] -, on[ch1] standby off -, on[ch1] standby on[ch2] - -, on[ch1] standby on[ch3] - - - } = 240 \text{ calculated test cases} - 56 \text{ (positive test cases)} - 153 \text{ (redundant test cases, calculated below)} = 31 \text{ negative test cases.}$

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p, s)$ and s is not λ then q is also negative, and we call q redundant. We can see an example:

Negative test case; p = on[ch1] 4

Redundant test case; q = on[ch1] 4 off

- q = concat (p, s) = “on[ch1] 4” + “off” = results in a redundant test case.

$T_{redundant} = \{ \text{off off, + off, - off, standby off, off standby, + standby, - standby, standby standby, off -, + -, - -, standby -, on[ch1] - -, off - -, + - -, - - -, standby - -, on[ch1] - - -, on[ch2] - - -, off - - -, + - - -, - - - -, standby - - -, on[ch1] on[ch1] off, on[ch1] on[ch2] off, on[ch1] on[ch3] off, on[ch1] on[ch4] off, on[ch1] - off, on[ch1] on[ch1] standby, on[ch1] on[ch2] standby, on[ch1] on[ch3] standby, on[ch1] on[ch4] standby, on[ch1] - standby, on[ch1] on[ch1] -, on[ch1] on[ch2] -, on[ch1] on[ch3] -, on[ch1] on[ch4] -, on[ch1] - -, on[ch1] on[ch1] - -, on[ch1] on[ch2] - -, on[ch1] on[ch3] - -, on[ch1] on[ch4] - -, on[ch1] off - -, on[ch1] - - -, on[ch1] standby - -, on[ch1] on[ch1] - - -, on[ch1] on[ch2] - - -, on[ch1] on[ch3] - - -, on[ch1] on[ch4] - - -, on[ch1] off - - -, on[ch1] + - - -, on[ch1] - - - -, on[ch1] standby - - -, on[ch2] on[ch1] off, on[ch2] on[ch2] off, on[ch2] on[ch3] off, on[ch2] on[ch4] off, on[ch2] off off, on[ch2] on[ch1] standby, on[ch2] on[ch2] standby, on[ch2] on[ch3] standby, on[ch2] on[ch4] standby, on[ch2] standby standby, on[ch2] on[ch1] -, on[ch2] on[ch2] -, on[ch2] on[ch3] -, on[ch2] on[ch4] -, on[ch2] on[ch1] - -, on[ch2] on[ch2] - -, on[ch2] on[ch3] - -, on[ch2] on[ch4] - -, on[ch2] off - -, on[ch2] - - -, on[ch2] standby - -, on[ch2] on[ch1] - - -, on[ch2] on[ch2] - - -, on[ch2] on[ch3] - - -, on[ch2] on[ch4] - - -, on[ch2] off - - -, on[ch2] - - - -, on[ch2] standby - - -, on[ch3] on[ch1] off, on[ch3] on[ch2] off, on[ch3] on[ch3] off, on[ch3] on[ch4] off, on[ch3] on[ch1] standby, on[ch3] on[ch2] standby, on[ch3] on[ch3] standby, on[ch3] on[ch4] standby, on[ch3] on[ch1] -, on[ch3] on[ch2] -, on[ch3] on[ch3] -, on[ch3] on[ch4] -, on[ch3] on[ch1] - -, on[ch3] on[ch2] - -, on[ch3] on[ch3] - -, on[ch3] on[ch4] - -, on[ch3] off - -, on[ch3] standby - -, on[ch3] on[ch1] - - -, on[ch3] on[ch2] - - -, on[ch3] on[ch3] - - -, on[ch3] on[ch4] - - -, on[ch3] off - - -, on[ch3] - - - -, on[ch3] standby - - -, on[ch4] on[ch1] off, on[ch4] on[ch2] off, on[ch4] on[ch3] off, on[ch4] on[ch4] off, on[ch4] on[ch1] standby, on[ch4] on[ch2] standby, on[ch4] on[ch3] standby, on[ch4] on[ch4] standby, on[ch4] on[ch1] -, on[ch4] on[ch2] -, on[ch4] on[ch3] -, on[ch4] on[ch4] -, on[ch4] on[ch1] - -, on[ch4] on[ch2] - -, on[ch4] on[ch3] - -, on[ch4] on[ch4] - -, on[ch4] off - -, on[ch4] + - -, on[ch4] standby - -, on[ch4] on[ch1] - - -, on[ch4] on[ch2] - - -, on[ch4] on[ch3] - - -, on[ch4] on[ch4] - - -, on[ch4] off - - -, on[ch4] + - - -, on[ch4] standby - - -, on[ch1] standby off off, on[ch1] standby + off, on[ch1] standby - off, on[ch1] standby standby off, on[ch1] standby + standby, on[ch1] standby - standby, on[ch1] standby standby standby, on[ch1] standby + -, on[ch1] standby - -, on[ch1] standby standby -, on[ch1] standby on[ch1] -, on[ch1] standby off - -, on[ch1]$

standby + - -, on[ch1] standby - - -, on[ch1] standby standby - - -, on[ch1] standby on[ch1] - - -, on[ch1] standby on[ch2] - - -, on[ch1] standby off - - -, on[ch1] standby + - - -, on[ch1] standby - - -, on[ch1] standby standby - - - } = 153 redundant test cases.

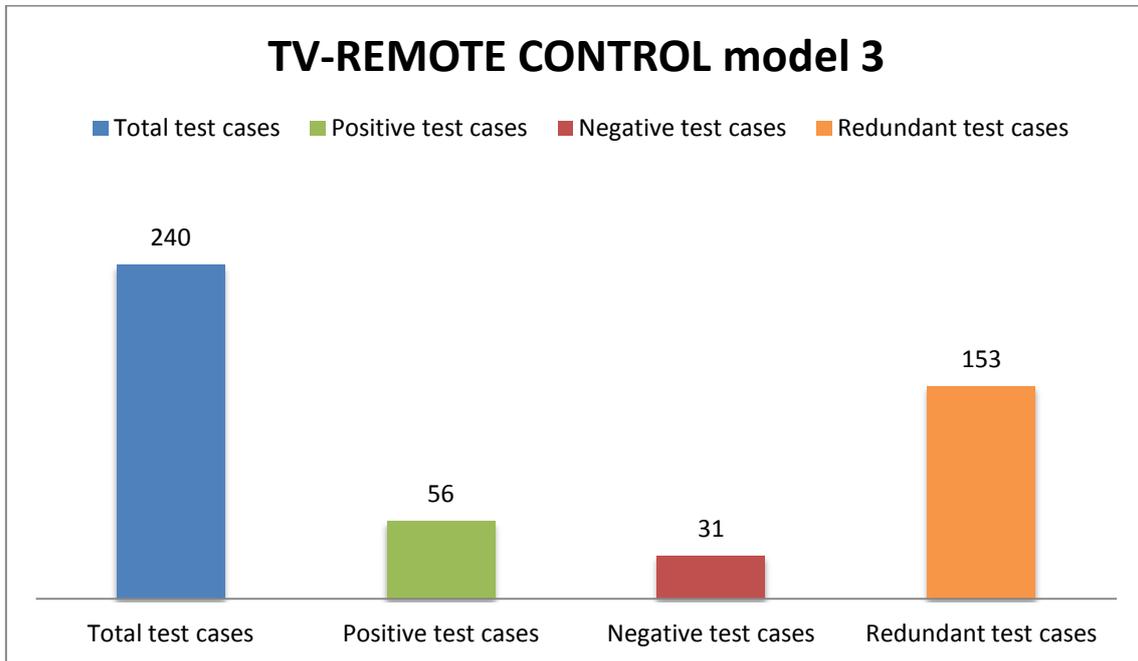


Figure 5.4.2.2. Types of test cases on model 3.

5.3.4 Simplest model with transitions "+" and "-".

The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$. Below we will see that it's impossible to calculate characterization set W.

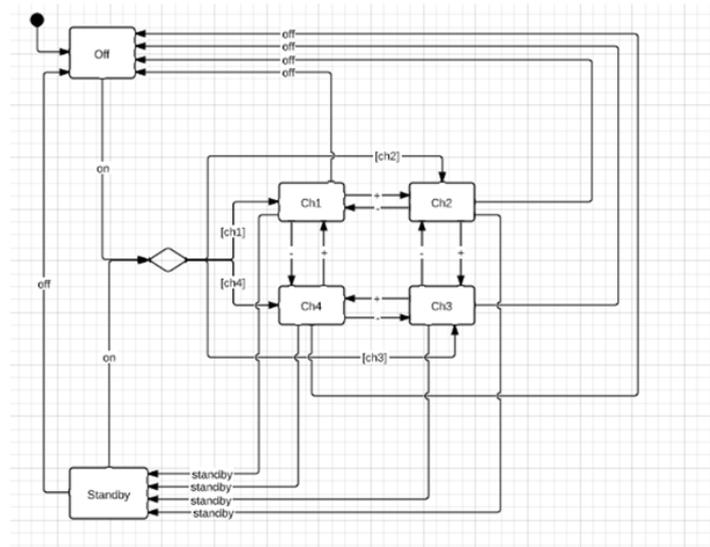


Figure 5.3.4.1 Changed model 4.

And finally we need to obtain the characterisation set (W), but this needs to be explained in several steps and we are going to see that for this model it is impossible to calculate. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label

and a set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only identify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4], off
Ch1	off	standby		+		-	off, standby, +*, -*
Ch2	off	standby	-		+		off, standby, +*, -*
Ch3	off	standby		-		+	off, standby, +*, -*
Ch4	off	standby	+		-		off, standby, +*, -*

Table 5.3.4.1 Transition matrix A.

As K. Bogdanov say in one of the emails we have exchanged the W set for the Working state cannot be generated, because different channels are not distinguishable. Each of the states can do both '+' and '-'. They said that we could make one channel, the initial one, such that there is no 'previous' channel. This will make them distinguishable. The W set for that state has to be built using '+' and '-'. So with this model it's impossible to calculate the characterisation set.

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 5.3.4.1. W Method.

T = { λ, on[ch1] , on[ch2] , on[ch3] , on[ch4] , on[ch1] standby} * {on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby} * (?), T = 6 * 8 * ? = Not supported

5.3.5 Simplest model change transitions "+" and "-" for numbers (direction up and down between channels).

Again we have changed transitions "+" and "-" by numbers to clarify the action of every event as we can see on Fig. 5.3.5.1.

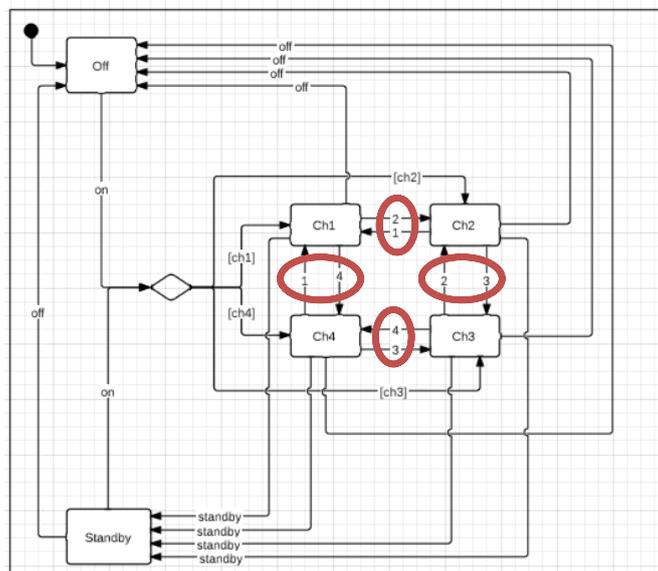


Figure 5.3.5.1. Changed Model 5.

The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$. Below we will see that it's impossible to calculate characterization set W.

And finally we need to obtain the characterisation set (W), but this needs to be explained in several steps and we are going to see as in section 5.3.4. that it is impossible to calculate. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label and a set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only indentify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4], off
Ch1	off	standby		2		4	off, standby, 2, 4
Ch2	off	standby	1		3		off, standby, 1, 3
Ch3	off	standby		2		4	off, standby, 2, 4
Ch4	off	standby	1		3		off, standby, 1, 3

Table 5.3.5.1. Transition matrix A.

$W = \{off, standby, 1, 2, 3, 4\}$. We can't differentiate the states Ch1 of Ch3 and states Ch2 of Ch4.

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

Equation 5.3.5.1. W Method.

$T = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * \{on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby\} * (?)$. $T = 6 * 10 * ? =$ Not supported

5.3.6 Simplest model with transitions "+" and "-" with a broken forward and a backward loop around ch1, ch2, ch3, ch4.

The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{ \lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$. Below as in the sections 5.3.4 and 5.3.5 we will see that it's impossible to calculate characterization set W.

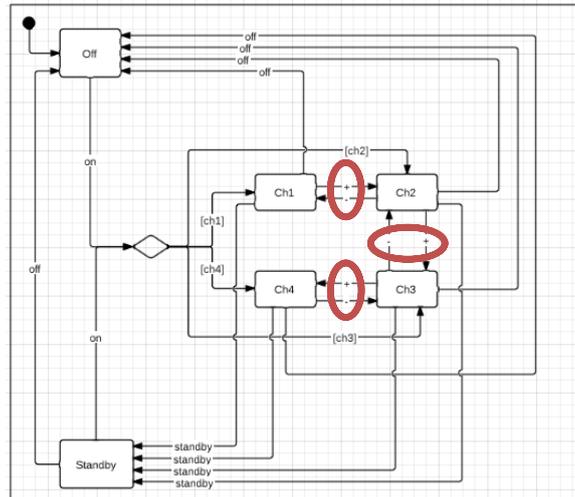


Figure 5.3.6.1. Changed Model 6.

And finally we need to obtain the characterisation set (W), but with this model it is impossible to calculate. The most important things to calculate W are; such paths for every pair of states comprise a characterisation set, each element of this particular W is a sequence consisting of a single label and a set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. In the table below (transition matrix A). We only identify all the transitions of every state.

	Off	Standby	Ch1	Ch2	Ch3	Ch4	Set of events
Off			on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4]
Standby	off		on[ch1]	on[ch2]	on[ch3]	on[ch4]	on[ch1],on[ch2], on[ch3],on[ch4], off
Ch1	off	standby		+			off, standby, +, ++, +++
Ch2	off	standby	-		+		off, standby, +, -, ++
Ch3	off	standby		-		+	off, standby, +, -, --
Ch4	off	standby			-		off, standby, -, --, ---

Table 5.3.6.1. Transition matrix A.

We may calculate the resulting W set because as we shown in figure 5.3.6.1 we can distinguish different states based on the sets of transitions which each one run but we consider that model 3 of this chapter is more efficient and it isn't necessary to calculate the test cases for this model, we only want to show that it would be possible to calculate them.

5.4 Obtained results and conclusions of W Method.

As shown in figure 5.4.1, the number of test cases obtained for different customizations of the flattened statechart (Figure 5.3.2) are 300 for “simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4”, 360 for “simplest model change transitions ‘+’ and ‘-’ for numbers only up direction between channels” and 240 for “simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4” while for the customizations of “simplest model change transitions ‘+’ and ‘-’ for numbers” and “simplest model transitions ‘+’ and ‘-’ without loops” cannot be supported by the W method because it is impossible to calculate the variable W as we have explained in section 5.3.5 and 5.3.6 for each of the two variations of the flattened unsupported statechart respectively. For the first model we have obtained 15% of positive test cases, 74% of negative test cases and 11% of redundant test cases. On the other hand in the second valid model we have obtained 11.1% of positive test cases, 21.6% and 67.3% of negative and redundant test cases and finally in the third model we have obtained 23.3% positive test cases, 12.9% negative test cases and 63.8% redundant test cases.

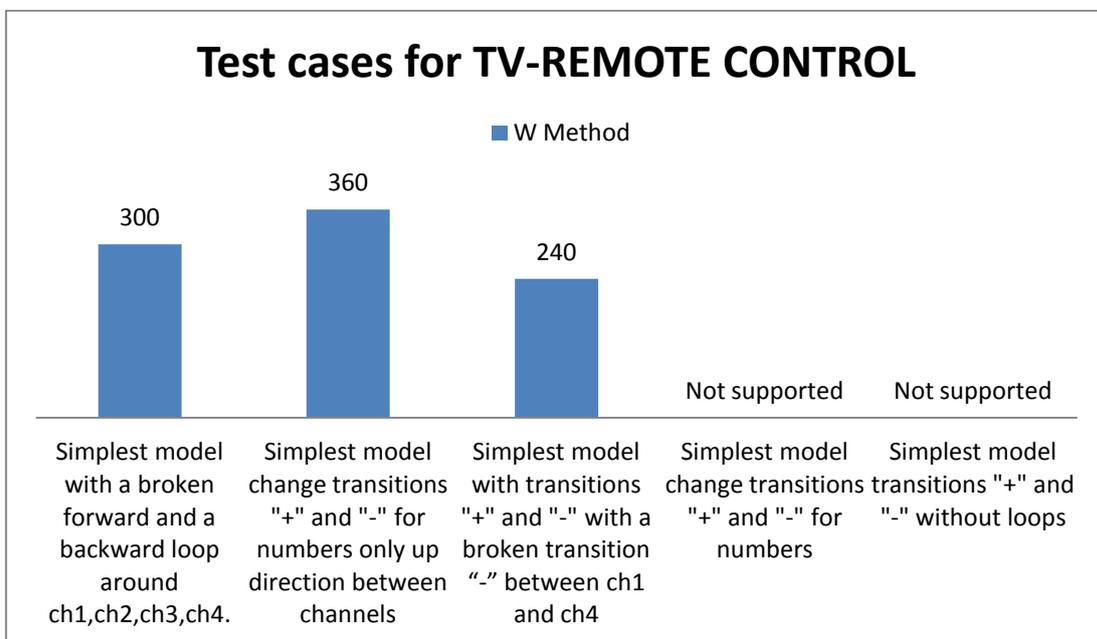


Figure 5.4.1 Test cases for the different models of TV1 statechart.

So in the third model (Figure 5.3.3.1) we obtain a larger number of positive test cases than models 1 (Figure 5.3.1.1) and 2 (Figure 5.3.2.1) which can help us most to confirm that a given statechart works properly or not, the basic rule would that it is not the same confirm that a statechart is properly functional with a single test case rather than check it whit several, dozens or even hundreds of test cases and the best percentage of positive test cases is for the third customization of the statechart, “simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4” model 3. The other aspect is rating which of the different customizations of the statechart (model 1, 2 or 3) seems more efficient in regard to design as in model 1 “simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4” there is no possibility of changing since channel 4 to 1 while in model 3 we can do it, so we could assume that this action increases the number of negative or redundant test cases, but on the contrary, we get a better percentage of positive test cases with model 3, so with the W method this is the most efficient of all customizations of the flattened statechart with we have worked.

5.5 Applying Wp-method to TV1 without concurrent region and without state hierarchy

We have applied W method in section 5.3 and now we will apply Wp method in subpoints of this section (since point 5.5.1 to 5.5.4), after in (5.5) we will discuss the results obtained for this method and we will talk about the conclusions obtained. Finally in section 5.7 we will compare the results obtained for W and Wp method in our case of study. The first thing that we need to do just like we did with W method, it's to eliminate all the substates of the original model. After that we can work with the simplest model of the statechart applying different changes, in section 5.5.1. We are going to work with the simplest model change transitions "+" and "-" for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4. After that we will work with other versions in points (since 5.5.2 to 5.5.4) checking the results obtained in section (5.6).

Step 0: modifications of the initial statechart: create a flattened statechart.

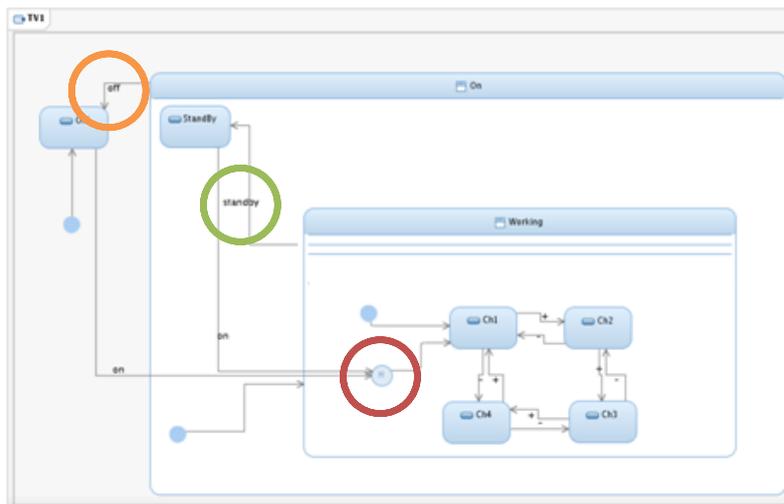


Figure 5.5.1 Original Statechart of TV1.

We will apply modifications on flattened version to calculate the different number of test cases on them. The important changes suffered by the model are the creation of every one of the transitions ("off", "standby") between the states of the channels (Ch1, Ch2, Ch3 and Ch4) and the states Off and Standby. With these transitions we managed to remove every substate of the statechart.

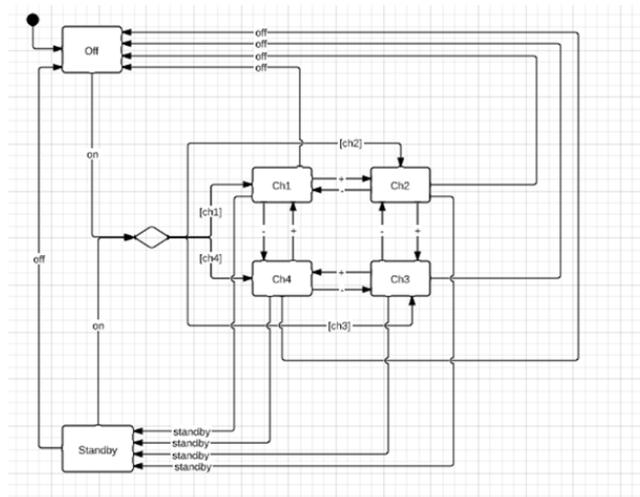


Figure 5.5.2 Flattened statechart.

5.7.2 Simplest model with a broken forward and a backward loop around ch1 , ch2, ch3 and ch4.

Now we are going to calculate the test case basis “TCB” for the flattened model of TV1 with changes between the states of the channels to break the strong loop between them. Additionally we have changed transitions “+” and “-” by numbers to clarify the action of every event as we can see in Fig. 5.5.1.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{ \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby} \}$. Any desired state starting from the initial one (denoted by C). $C = \{ \lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{ standby} \}$. The set denoted by W it's the same that we have calculated in section 5.3.1 of this chapter for the W method; $W = \{ \text{off}, \text{standby}, 1, 2, 4 \}$.

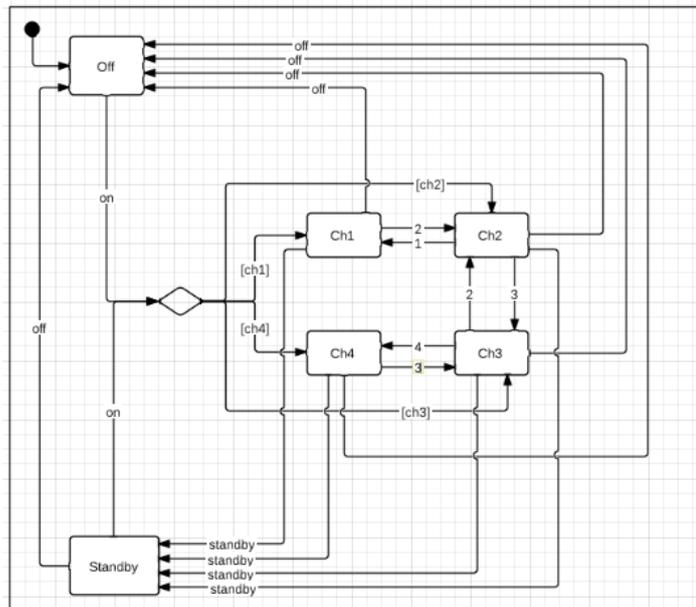


Figure 5.5.1.1 Model 1.

Note that in this statechart ϕ , C and W are equals at Φ^M , C^M and W^M because we have not any composite-state. We have explained the meaning and how to calculate them (Φ^M , C^M and W^M) in section 4.1.11 literature review of this thesis “Testing from statecharts using the W_p method”. From here varies how to calculate the number of test cases for this method. It consists of two phases and for the first one we need to clarify the Concept of “ $Conf^p$ ”; for a configuration $conf$, an identification set W_{conf}^{root} is a set allowing one to distinguish between $conf$ and all other configurations in a statechart. The Concept of Configuration comprises sets of states which are left and entered by full compound transitions are called configurations and consist of states a statechart can be in simultaneously. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. The most important part of this first phase it is to calculate the small w sets. These are a set allowing one to distinguish between $conf$ and all other configurations in a statechart and we use small w sets to identify states in an implementation applying the same formula W as we use in W method in each state. Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly.

The w sets obtained for this first model are $W_{OFF}^{root} = \{off\}$, $W_{STANDBY}^{root} = \{off, standby\}$, $W_{CH1}^{root} = \{2, off, standby\}$, $W_{CH2}^{root} = \{1, 3, off, standby\}$, $W_{CH3}^{root} = \{2, 4, off, standby\}$, $W_{CH4}^{root} = \{3, off, standby\}$. Combining all small w sets to obtain full W, usually it's the same that W method before to apply the formula of phase 1. The full $W = \bigcup_{conf} W_{conf}^{root}$ and we can develop the formula with the w sets in $W = w_{OFF}^{root} \cup w_{STANDBY}^{root} \cup w_{CH1}^{root} \cup w_{CH2}^{root} \cup w_{CH3}^{root} \cup w_{CH4}^{root} = \{off, standby, 2, 3\}$. The advantage of working with Wp method is that we can reduce the W chain because with the W method the W set was $\{off, standby, 1, 2, 4\}$ and with Wp method we have reduced it to $\{off, standby, 2, 3\}$. Now we can apply the formula of phase 1, but first we are going to calculate the transitions that we will check with this formula; $T_{explored_in_phase1} = C^*(\lambda \cup \Phi \cup \dots \cup \Phi^{m-n})$. Therefore if we apply this first phase of the formula to our case study we get the number of transitions that are covering when doing testing with this method on our statechart; $T_{explored_in_phase1} = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} = 6$ transitions explored in phase 1, the rest of transition will be explored in phase 2. After comprove the number of covered transitions we will go to apply the formula to calculate the set of test cases used in the first phase.

$$T_1 = C^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n}) * W$$

Equation 5.5.1.1 Set of test cases for the first phase of Wp method.

We have calculated the set $C^M = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$ and $W = \{off, standby, 2, 3\}$. Now we can calculate the set of test cases for the first phase of Wp method with $T_1 = C^M * W = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * \{off, standby, 2, 3\} = \{off, standby, 2, 3, on[ch1] off, on[ch1] standby, on[ch1] 2, on[ch1] 3, on[ch2] off, on[ch2] standby, on[ch2] 2, on[ch2] 3, on[ch3] off, on[ch3] standby, on[ch3] 2, on[ch3] 3, on[ch4] off, on[ch4] standby, on[ch4] 2, on[ch4] 3, on[ch1] standby off, on[ch1] standby standby, on[ch1] standby 2, on[ch1] standby 3\} = 6 * 4 = 24$ test cases in phase 1. Additionally we can compute the transitions that are out of the phase 1 and we will check in the phase 2 as we have commented previously.

$$T_{explored_in_phase2} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - T_{explored_in_phase1}$$

Equation 5.5.1.2 Transitions that will be explored in phase 2.

$$T_{explored_in_phase2} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n})$$

Equation 5.5.1.3 Developed Transitions that will be explored in phase 2.

The set of transitions that will be explored in phase 2 are; $T_{explored_in_phase2} = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * (\{\lambda\} \cup \{on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby\}) - \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} = 6 * 10 - 6 = 54$ transitions out off the phase 1 and will be explored in phase 2. To calculate the set of test cases for the second phase of the Wp method we need to calculate the variable $conf_{init} = \{Off\}$ which is the initial state of the statechart before to do any action. In Equation 5.5.1.4 is showed the formula to compute the number of test cases in phase 2.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE}^{root}(path, conf_{init})$$

Equation 5.5.1.4 Test cases of phase 2.

The equation 5.5.1.4 can be expressed in two parts as defined by “K. Bogdanov” in [7]; First we can calculate TS with the equation 5.5.1.5. Once calculated TS we can calculate CE (Equation 5.5.1.6) and give as input parameter each of the paths we have obtained in TS to obtain the expected output of every test case shown in table 5.5.1.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

$$TS = C^M * \Phi^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n})$$

$$C^M * (\{1\} \cup \Phi^M \cup (\Phi^M)^{m-n+1})$$

$$C^M * \Phi^M$$

Equation 5.5.1.5 Test cases of phase 2.

TS = { λ, on[ch1] , on[ch2] , on[ch3] , on[ch4] , on[ch1] standby} * {on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby} = {on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby , on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] off, on[ch1] 1, on[ch1] 2, on[ch1] 3, on[ch1] 4, on[ch1] standby, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] off, on[ch2] 1, on[ch2] 2, on[ch1] 3, on[ch2] 4, on[ch2] standby, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] off, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch3] 4, on[ch3] standby, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] off, on[ch4] 1, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1] standby standby} = 6 * 10 = 60 test cases in phase 2.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

Equation 5.5.1.6 Expected output of every test case.

$CE_{(ON(CH1), conf_{\{OFF\}})} = \{CH1\} = w_{CE(ON(CH1), conf_{init})}^{root} = w_{CH1}^{root}$
$CE_{(ON(CH2), conf_{\{OFF\}})} = \{CH2\} = w_{CE(ON(CH2), conf_{init})}^{root} = w_{CH2}^{root}$
$CE_{(ON(CH3), conf_{\{OFF\}})} = \{CH3\} = w_{CE(ON(CH3), conf_{init})}^{root} = w_{CH3}^{root}$
$CE_{(ON(CH4), conf_{\{OFF\}})} = \{CH4\} = w_{CE(ON(CH4), conf_{init})}^{root} = w_{CH4}^{root}$
$CE_{(OFF, conf_{\{OFF\}})} = \{OFF\} = w_{CE(OFF, conf_{init})}^{root} = w_{OFF}^{root}$
$CE_{(STANDBY, conf_{\{OFF\}})} = \{OFF\} = w_{CE(STANDBY, conf_{init})}^{root} = w_{OFF}^{root}$
$CE_{(ON(CH1)_ON(CH1), conf_{\{OFF\}})} = \{ON(CH1)\} = w_{CE(ON(CH1), conf_{init})}^{root} = w_{CH1}^{root}$
$CE_{(ON(CH1)_STANDBY, conf_{\{OFF\}})} = \{STANDBY\} = w_{CE(STANDBY, conf_{init})}^{root} = w_{STANDBY}^{root}$
...

Table 5.5.1.1 Test cases of phase 2.

Applying this to the TV1, the size of the set of test cases for the first phase of the Wp method is 24 and the second one 60, resulting in 84 sequences which is less than that obtained with the W method (300 test cases) for the same customization of the original statechart (section 5.3.1 of this chapter “*model 1*”). Finally we are going to calculate the positive, negative and redundant test cases for this model as we did with the W method to see which one is more effective in section 5.7 of this chapter.

Combining the set of test cases obtained in the two phases of the Wp method we obtain the set { off, standby, 2, 3, on[ch1] off , on[ch1] standby, on[ch1] 2, on[ch1] 3, on[ch2] off , on[ch2] standby, on[ch2] 2, on[ch2] 3, on[ch3] off , on[ch3] standby, on[ch3] 2, on[ch3] 3, on[ch4] off , on[ch4] standby, on[ch4] 2, on[ch4] 3, on[ch1] standby off , on[ch1] standby standby, on[ch1] standby 2, on[ch1] standby 3, on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby , on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] off, on[ch1] 1, on[ch1] 2, on[ch1] 3, on[ch1] 4, on[ch1] standby, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] off, on[ch2] 1, on[ch2] 2, on[ch1] 3, on[ch2] 4, on[ch2] standby, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] off, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch3] 4, on[ch3] standby, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] off, on[ch4] 1, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1] standby standby} = 84 test cases for the Wp method.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that q = concat (p,s) and s is not π then q is also negative, and we call q redundant. The set of positive test cases are $T_{positive} = \{$ on[ch1] off , on[ch1] standby, on[ch1] 2, on[ch2] off , on[ch2] standby, on[ch2] 3, on[ch3] off, on[ch3] standby, on[ch3] 2, on[ch4] off , on[ch4] standby, on[ch4] 3, on[ch1] standby off, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] off, on[ch1] 2, on[ch1] standby, on[ch2] off, on[ch2] 1, on[ch2] 3, on[ch2] standby, on[ch3] off, on[ch3] 2, on[ch3] 4, on[ch3] standby, on[ch4] off, on[ch4] 3, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off} = 36 positive test cases.

$T_{negative} = \{$ off, standby, 2, 3, on[ch1] 3, on[ch2] 2, on[ch3] 3, on[ch4] 2, on[ch1] standby standby, on[ch1] standby 2, on[ch1] standby 3, off, 1, 2, 3, 4, standby , on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] 1, on[ch1] 3, on[ch1] 4, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] 2, on[ch2] 4, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] 1, on[ch3] 3, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] 1, on[ch4] 2, on[ch4] 4, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1] standby standby} = 84 calculated test cases – 44 (positive test cases) – 0 (redundant test cases) = 48 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that q=concat(p,s) and s is not λ then q is also negative, and we call q redundant. We can see an example: Negative test case; p = on[ch1] 4 and redundant test case; q = on[ch1] 4 off because “q = concat (p, s)” = “on[ch1] 4” + “off” = results in a redundant

test case. As we can see there aren't any redundant test case from the negative test cases calculated above.

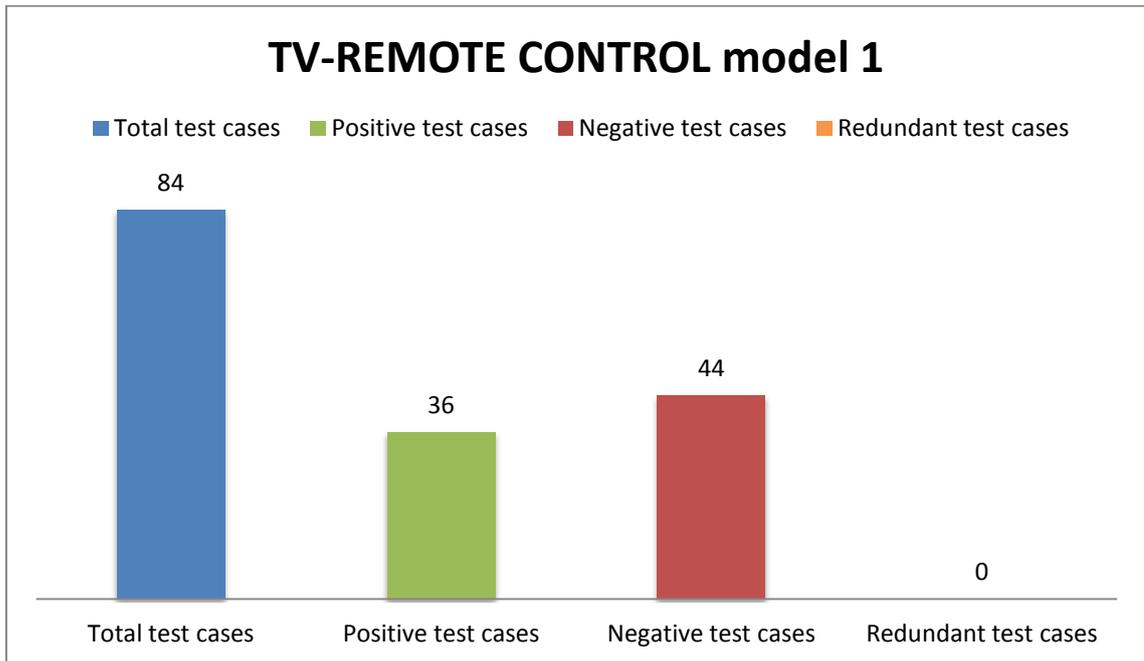


Figure 5.5.1.2. Types of test cases on model 1.

5.7.3 Simplest model change transitions "+" and "-" for numbers only up direction between channels.

Now we are going to calculate the test case basis "TCB" for the flattened model of TV1 with changes between the states of the channels to break the strong loop between them. Again we have changed transitions "+" and "-" by numbers to clarify the action of every event as we can see on Fig. 5.5.2.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{\text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby}\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}], \text{standby}\}$.

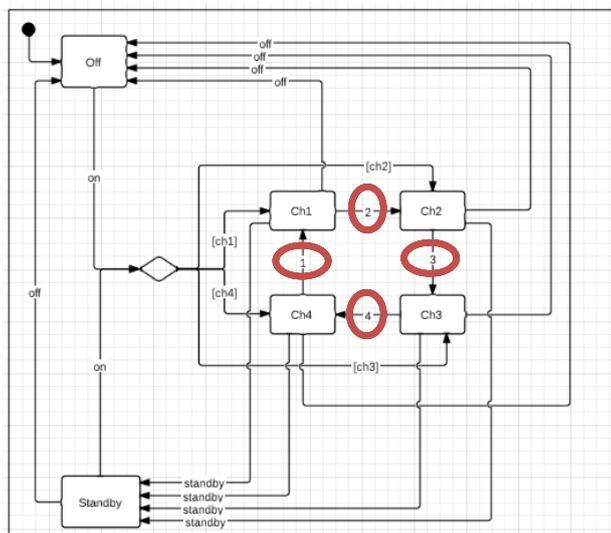


Figure 5.5.2.1 Customized model 2.

Note that in this statechart ϕ , C and W are equals at Φ^M , C^M and W^M because we have not any composite-state. We have explained the meaning and how to calculate them (Φ^M , C^M and W^M) in section 4.1.11 literature review of this thesis "Testing from statecharts using the Wp method". From here varies how to calculate the number of test cases for this method. It consists of two phases and for the first one we need to clarify the Concept of "Conf"; for a configuration conf, an identification set W_{conf}^{root} is a set allowing one to distinguish between conf and all other configurations in a statechart. The concept of configuration comprises sets of states which are left and entered by full compound transitions are called configurations and consist of states a statechart can be in simultaneously. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. The most important part of this first phase it is to calculate the small w sets. These are a set allowing one to distinguish between conf and all other configurations in a statechart and we use small w sets to identify states in an implementation applying the same W formula as we use in W method in each state. Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly.

The w sets obtained for this first model are $W_{OFF}^{root} = \{off\}$, $W_{STANDBY}^{root} = \{off, standby\}$, $W_{CH1}^{root} = \{2, off, standby\}$, $W_{CH2}^{root} = \{3, off, standby\}$, $W_{CH3}^{root} = \{4, off, standby\}$, $W_{CH4}^{root} = \{1, off, standby\}$. Combining all small w sets to obtain full W, usually it's the same that W method before to apply the formula of phase 1. The full $W = \bigcup_{conf} W_{conf}^{root}$ and we can develop the formula with the w sets in $W = W_{OFF}^{root} \cup W_{STANDBY}^{root} \cup W_{CH1}^{root} \cup W_{CH2}^{root} \cup W_{CH3}^{root} \cup W_{CH4}^{root} = \{off, standby, 1, 2, 3, 4\}$. In this case it is impossible to reduce the size of the working string W that obtained with the same method with the W method. Now we can apply the formula of phase 1, but first we are going to calculate the transitions that we will check with this formula; $T_{explored_in_phase1} = C * (\lambda \cup \Phi \cup \dots \cup \Phi^{m-n})$. Therefore if we apply this first phase of the formula to our case study we get the number of transitions that are covering when doing testing with this method on our statechart; $T_{explored_in_phase1} = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} = 6$ transitions explored in phase 1, the rest of transition will be explored in phase 2. After compare the number of covered transitions we will go to apply the formula to calculate the set of test cases used in the first phase.

$$T_1 = C^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n}) * W$$

Equation 5.5.2.1 Set of test cases for the first phase of Wp method.

We have calculated the set $C^M = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$ and $W = \{off, standby, 1, 2, 3, 4\}$. Now we can calculate the set of test cases for the first phase of Wp method with $T_1 = C^M * W = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * \{off, standby, 1, 2, 3, 4\} = \{off, standby, 1, 2, 3, 4, on[ch1] off, on[ch1] standby, on[ch1] 2, on[ch1] 1, on[ch1] 2, on[ch1] 3, on[ch1] 4, on[ch2] off, on[ch2] standby, on[ch2] 1, on[ch2] 2, on[ch2] 3, on[ch2] 4, on[ch3] off, on[ch3] standby, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch3] 4, on[ch4] off, on[ch4] standby, on[ch4] 1, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch1] standby off, on[ch1] standby standby, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4\} = 6 * 6 = 36$ test cases in phase 1.

Additionally we can compute the transitions that are out of the phase 1 and we will check in the phase 2 as we have commented previously.

$$T_{\text{explored_in_phase2}} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - T_{\text{explored_in_phase1}}$$

Equation 5.5.2.2 Transitions that will be explored in phase 2.

$$T_{\text{explored_in_phase2}} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n})$$

Equation 5.5.2.3 Developed Transitions that will be explored in phase 2.

The set of transitions that will be explored in phase 2 are; $T_{\text{explored_in_phase2}} = \{\lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{standby}\} * (\{\lambda\} \cup \{\text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby}\}) - \{\lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{standby}\} = 6 * 10 - 6 = 54$ transitions out off the phase 1 and will be explored in phase 2. To calculate the set of test cases for the second phase of the Wp method we need to calculate the variable $conf_{\text{init}} = \{\text{Off}\}$ which is the initial state of the statechart before to do any action. In Equation 5.5.1.4 is showed the formula to compute the number of test cases in phase 2.

$$T_2 = \bigcup_{\text{path} \in TS} \{\text{path}\} * w_{CE(\text{path}, \text{conf}_{\text{init}})}^{\text{root}}$$

Equation 5.5.2.4 Test cases of phase 2.

The equation 5.5.1.4 can be expressed in two parts; First we can calculate TS with the equation 5.5.1.5. Once calculated TS we can calculate CE (Equation 5.5.1.6) and give as input parameter each of the paths we have obtained in TS to obtain the expected output of every test case shown in table 5.5.2 (*CE stands for Configuration Entered by TS*).

$$T_2 = \bigcup_{\text{path} \in TS} \{\text{path}\} * w_{CE(\text{path}, \text{conf}_{\text{init}})}^{\text{root}}$$

$$TS = C^M * \Phi^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n})$$

$$\downarrow$$

$$C^M * (\{1\} \cup \Phi^M \cup (\Phi^M)^{m-n+1})$$

$$\downarrow$$

$$C^M * \Phi^M$$

Equation 5.5.2.5 Test cases of phase 2.

$TS = \{\lambda, \text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{standby}\} * \{\text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby}\} = \{\text{on}[\text{ch1}], \text{on}[\text{ch2}], \text{on}[\text{ch3}], \text{on}[\text{ch4}], \text{off}, 1, 2, 3, 4, \text{standby}, \text{on}[\text{ch1}] \text{on}[\text{ch1}], \text{on}[\text{ch1}] \text{on}[\text{ch2}], \text{on}[\text{ch1}] \text{on}[\text{ch3}], \text{on}[\text{ch1}] \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{off}, \text{on}[\text{ch1}] 1, \text{on}[\text{ch1}] 2, \text{on}[\text{ch1}] 3, \text{on}[\text{ch1}] 4, \text{on}[\text{ch1}] \text{standby}, \text{on}[\text{ch2}] \text{on}[\text{ch1}], \text{on}[\text{ch2}] \text{on}[\text{ch2}], \text{on}[\text{ch2}] \text{on}[\text{ch3}], \text{on}[\text{ch2}] \text{on}[\text{ch4}], \text{on}[\text{ch2}] \text{off}, \text{on}[\text{ch2}] 1, \text{on}[\text{ch2}] 2, \text{on}[\text{ch1}] 3, \text{on}[\text{ch2}] 4, \text{on}[\text{ch2}] \text{standby}, \text{on}[\text{ch3}] \text{on}[\text{ch1}], \text{on}[\text{ch3}] \text{on}[\text{ch2}], \text{on}[\text{ch3}] \text{on}[\text{ch3}], \text{on}[\text{ch3}] \text{on}[\text{ch4}], \text{on}[\text{ch3}] \text{off}, \text{on}[\text{ch3}] 1, \text{on}[\text{ch3}] 2, \text{on}[\text{ch3}] 3, \text{on}[\text{ch3}] 4, \text{on}[\text{ch3}] \text{standby}, \text{on}[\text{ch4}] \text{on}[\text{ch1}], \text{on}[\text{ch4}] \text{on}[\text{ch2}], \text{on}[\text{ch4}] \text{on}[\text{ch3}], \text{on}[\text{ch4}] \text{on}[\text{ch4}], \text{on}[\text{ch4}] \text{off}, \text{on}[\text{ch4}] 1, \text{on}[\text{ch4}] 2, \text{on}[\text{ch4}] 3, \text{on}[\text{ch4}] 4, \text{on}[\text{ch4}] \text{standby}, \text{on}[\text{ch1}] \text{standby} \text{on}[\text{ch1}], \text{on}[\text{ch1}] \text{standby} \text{on}[\text{ch2}], \text{on}[\text{ch1}] \text{standby} \text{on}[\text{ch3}], \text{on}[\text{ch1}] \text{standby} \text{on}[\text{ch4}], \text{on}[\text{ch1}] \text{standby} \text{off}, \text{on}[\text{ch1}] \text{standby} 1, \text{on}[\text{ch1}] \text{standby} 2, \text{on}[\text{ch1}] \text{standby} 3, \text{on}[\text{ch1}] \text{standby} 4, \text{on}[\text{ch1}] \text{standby} \text{standby}\} = 6 * 10 = 60$ test cases in phase 2.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

Equation 5.5.2.6 Expected output of every test case.

$CE_{(ON(CH1), conf_{\{OFF\}})} = \{CH1\} = w_{CE(ON(CH1), conf_{init})}^{root} = W_{CH1}^{root}$
$CE_{(ON(CH2), conf_{\{OFF\}})} = \{CH2\} = w_{CE(ON(CH2), conf_{init})}^{root} = W_{CH2}^{root}$
$CE_{(ON(CH3), conf_{\{OFF\}})} = \{CH3\} = w_{CE(ON(CH3), conf_{init})}^{root} = W_{CH3}^{root}$
$CE_{(ON(CH4), conf_{\{OFF\}})} = \{CH4\} = w_{CE(ON(CH4), conf_{init})}^{root} = W_{CH4}^{root}$
$CE_{(OFF, conf_{\{OFF\}})} = \{OFF\} = w_{CE(OFF, conf_{init})}^{root} = W_{OFF}^{root}$
$CE_{(STANDBY, conf_{\{OFF\}})} = \{OFF\} = w_{CE(STANDBY, conf_{init})}^{root} = W_{OFF}^{root}$
$CE_{(ON(CH1)_{-}ON(CH1), conf_{\{OFF\}})} = \{ON(CH1)\} = w_{CE(ON(CH1), conf_{init})}^{root} = W_{CH1}^{root}$
$CE_{(ON(CH1)_{-}STANDBY, conf_{\{OFF\}})} = \{STANDBY\} = w_{CE(STANDBY, conf_{init})}^{root} = W_{STANDBY}^{root}$
...

Table 5.5.2.1 Test cases of phase 2.

Applying this to the TV1, the size of the set of test cases for the first phase of the Wp method is 36 and the second one 60, resulting in 96 sequences which is less than that obtained with the W method (360 test cases) for the same customization of the original statechart (section 5.3.2 of this chapter “model 2”). Finally we are going to calculate the positive, negative and redundant test cases for this model as we did with the W method to see which one is more effective in section 5.7 of this chapter.

Combining the set of test cases obtained in the two phases of the Wp method we obtain the set { off, standby, 1, 2, 3, 4, on[ch1] off , on[ch1] standby, on[ch1] 1, on[ch1] 2, on[ch1] 3, on[ch1] 4, on[ch2] off , on[ch2] standby, on[ch2] 1, on[ch2] 2, on[ch2] 3, on[ch2] 4, on[ch3] off, on[ch3] standby, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch3] 4, on[ch4] off, on[ch4] standby, on[ch4] 1, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch1] standby off , on[ch1] standby standby, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1], on[ch2], on[ch3], on[ch4], off, 1, 2, 3, 4, standby , on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] off, on[ch1] 1, on[ch1] 2, on[ch1] 3, on[ch1] 4, on[ch1] standby, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] off, on[ch2] 1, on[ch2] 2, on[ch1] 3, on[ch2] 4, on[ch2] standby, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] off, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch3] 4, on[ch3] standby, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] off, on[ch4] 1, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1] standby standby} = 96 test cases for the Wp method.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next

step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not π then q is also negative, and we call q redundant. The set of positive test cases are $T_{positive} = \{ \text{off, standby, on[ch1] off, on[ch1] standby, on[ch1] 2, on[ch2] off, on[ch2] standby, on[ch2] 3, on[ch3] off, on[ch3] standby, on[ch3] 4, on[ch4] off, on[ch4] standby, on[ch4] 1, on[ch1] standby off, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] off, on[ch1] 2, on[ch1] standby, on[ch2] off, on[ch2] 1, on[ch2] 3, on[ch2] standby, on[ch3] off, on[ch3] 2, on[ch3] 4, on[ch3] standby, on[ch4] off, on[ch4] 3, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off} = 38$ positive test cases.

$T_{negative} = \{ 1, 2, 3, 4, \text{on[ch1] 1, on[ch1] 3, on[ch1] 4, on[ch2] 1, on[ch2] 2, on[ch2] 4, on[ch3] 1, on[ch3] 2, on[ch3] 3, on[ch4] 2, on[ch4] 3, on[ch4] 4, on[ch1] standby standby, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, off, 1, 2, 3, 4, standby, on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] 1, on[ch1] 3, on[ch1] 4, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] 2, on[ch2] 4, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] 1, on[ch3] 3, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] 1, on[ch4] 2, on[ch4] 4, on[ch1] standby 1, on[ch1] standby 2, on[ch1] standby 3, on[ch1] standby 4, on[ch1] standby standby} = 96$ calculated test cases – 38 (positive test cases) – 0 (redundant test cases) = 58 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not λ then q is also negative, and we call q redundant. We can see an example: Negative test case; p = on[ch1] 4 and redundant test case; q = on[ch1] 4 off because “q = concat (p, s)” = “on[ch1] 4” + “off” = results in a redundant test case. As we can see there aren’t any redundant test case from the negative test cases calculated above.

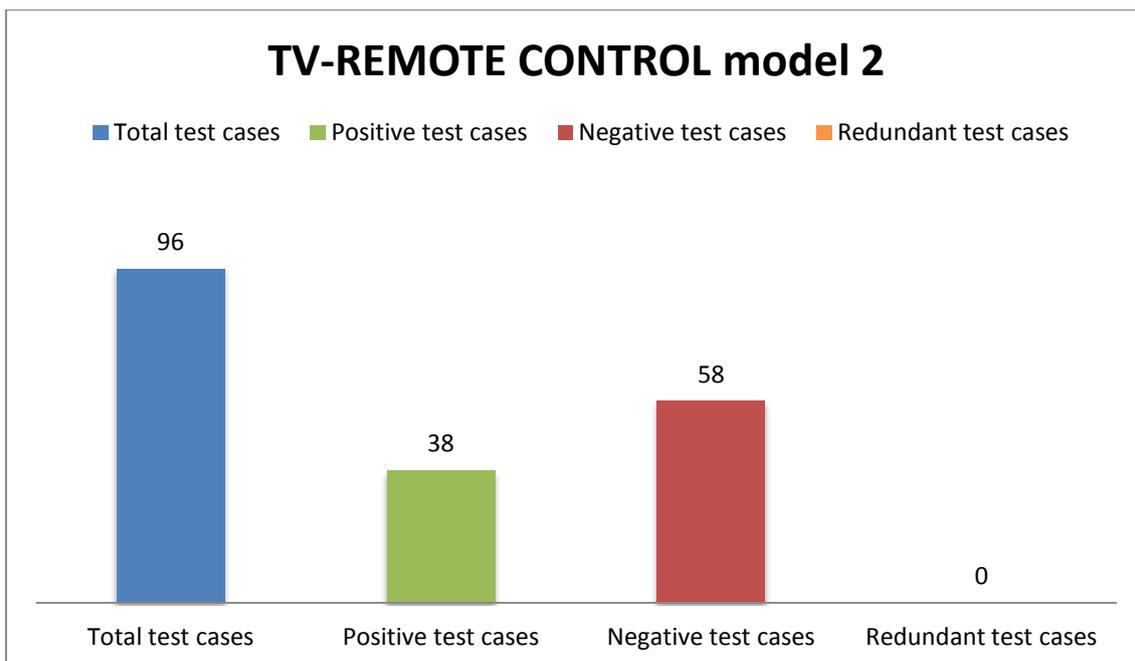


Figure 5.5.2.2. Types of test cases on model 2.

5.5.3 Simplest model change transitions "+" and "-" for numbers only up direction between channels.

Now we are going to calculate the test case basis "TCB" for the flattened model of TV1 with changes between the states of the channels to break the strong loop between them. With this model we maintain the transitions "+" and "-" but we have removed the transition "-" since channel 1 to channel 4 to break the loop. as we can see on Fig. 5.5.3.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$.

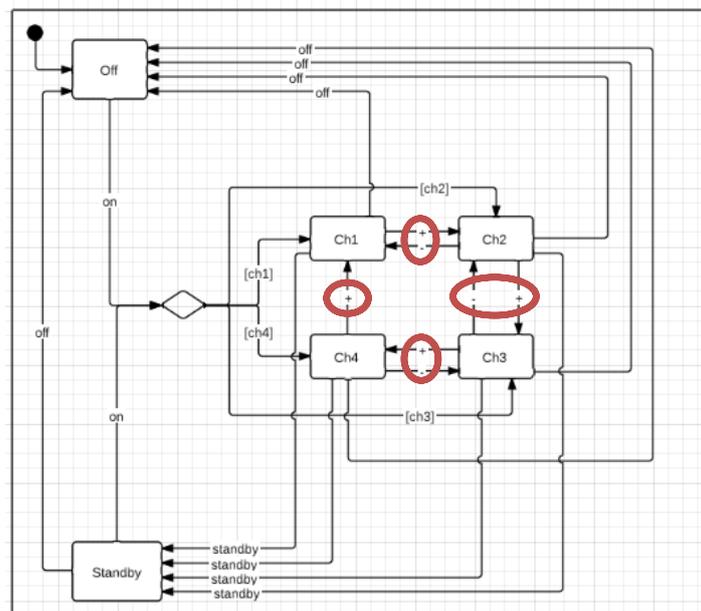


Figure 5.5.3.1 Customized model 3.

Note that in this statechart ϕ , C and W are equals at Φ^M , C^M and W^M because we have not any composite-state. We have explained the meaning and how to calculate them (Φ^M , C^M and W^M) in section 4.1.11 literature review of this thesis "Testing from statecharts using the Wp method". From here varies how to calculate the number of test cases for this method. It consists of two phases and for the first one we need to clarify the Concept of "Conf"; for a configuration conf, an identification set W_{conf}^{root} is a set allowing one to distinguish between conf and all other configurations in a statechart. The concept of configuration comprises sets of states which are left and entered by full compound transitions are called configurations and consist of states a statechart can be in simultaneously. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. The most important part of this first phase it is to calculate the small w sets. These are a set allowing one to distinguish between conf and all other configurations in a statechart and we use small w sets to identify states in an implementation applying the same W formula as we use in W method in each state. Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly.

The w sets obtained for this first model are $W_{OFF}^{root} = \{off\}$, $W_{STANDBY}^{root} = \{off, standby\}$, $W_{CH1}^{root} = \{+, off, standby\}$, $W_{CH2}^{root} = \{-, off, standby\}$, $W_{CH3}^{root} = \{-, -, off, standby\}$, $W_{CH4}^{root} = \{-, -, -, off, standby\}$. Combining all small w sets to obtain full W , usually it's the same that W method before to apply the formula of phase 1. The full $W = \bigcup_{conf} w_{conf}^{root}$ and we can develop the formula with the w sets in $W = W_{OFF}^{root} \cup W_{STANDBY}^{root} \cup W_{CH1}^{root} \cup W_{CH2}^{root} \cup W_{CH3}^{root} \cup W_{CH4}^{root} = \{off, standby, +, -, -, - -\}$. In this case it is impossible to reduce the size of the working string W that obtained with the same method with the W method. Now we can apply the formula of phase 1, but first we are going to calculate the transitions that we will check with this formula; $T_{explored_in_phase1} = C^*(\lambda \cup \Phi \cup \dots \cup \Phi^{m-n})$. Therefore if we apply this first phase of the formula to our case study we get the number of transitions that are covering when doing testing with this method on our statechart; $T_{explored_in_phase1} = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} = 6$ transitions explored in phase 1, the rest of transition will be explored in phase 2. After comprove the number of covered transitions we will go to apply the formula to calculate the set of test cases used in the first phase.

$$T_1 = C^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n}) * W$$

Equation 5.5.3.1 Set of test cases for the first phase of Wp method.

We have calculated the set $C^M = C = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$ and $W = \{off, standby, +, -, -, - -\}$. Now we can calculate the set of test cases for the first phase of Wp method with $T_1 = C^M * W = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * \{off, standby, +, -, -, - -\} = \{off, standby, +, -, -, - -, on[ch1] off, on[ch1] standby, on[ch1] +, on[ch1] -, on[ch1] - -, on[ch1] - - -, on[ch2] off, on[ch2] standby, on[ch2] +, on[ch2] -, on[ch2] - -, on[ch2] - - -, on[ch3] off, on[ch3] standby, on[ch3] +, on[ch3] -, on[ch3] - -, on[ch3] - - -, on[ch4] off, on[ch4] standby, on[ch4] +, on[ch4] -, on[ch4] - -, on[ch4] - - -, on[ch1] standby off, on[ch1] standby standby, on[ch1] standby +, on[ch1] standby -, on[ch1] standby - -, on[ch1] standby - - -\} = 6 * 6 = 36$ test cases in phase 1.

Additionally we can compute the transitions that are out of the phase 1 and we will check in the phase 2 as we have commented previously.

$$T_{explored_in_phase2} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - T_{explored_in_phase1}$$

Equation 5.5.3.2 Transitions that will be explored in phase 2.

$$T_{explored_in_phase2} = C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) - C^M * (\{\lambda\} \cup \Phi^M \cup (\Phi^M)^{m-n})$$

Equation 5.5.3.3 Developed Transitions that will be explored in phase 2.

The set of transitions that will be explored in phase 2 are; $T_{explored_in_phase2} = \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} * (\{\lambda\} \cup \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}) - \{\lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\} = 6 * 8 - 6 = 42$ transitions out off the phase 1 and will be explored in phase 2. To calculate the set of test cases for the second phase of the Wp method we need to calculate the variable $conf_{init} = \{Off\}$ which is the initial state of the statechart before to do any action. In Equation 5.5.3.4 is showed the formula to compute the number of test cases in phase 2.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

Equation 5.5.3.4 Test cases of phase 2.

The equation 5.5.3.4 can be expressed in two parts; First we can calculate TS with the equation 5.5.3.5. Once calculated TS we can calculate CE (Equation 5.5.3.6) and give as input parameter each of the paths we have obtained in TS to obtain the expected output of every test case shown in table 5.5.3 (CE stands for Configuration Entered by TS).

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

$$\begin{aligned}
 TS &= C^M * \Phi^M * (\{I\} \cup \Phi^M \cup (\Phi^M)^2 \cup \dots \cup (\Phi^M)^{m-n}) \\
 &\quad \downarrow \\
 &= C^M * (\{1\} \cup \Phi^M \cup (\Phi^M)^{m-n+1}) \\
 &\quad \downarrow \\
 &= C^M * \Phi^M
 \end{aligned}$$

Equation 5.5.3.5 Test cases of phase 2.

TS = { λ, on[ch1] , on[ch2] , on[ch3] , on[ch4] , on[ch1] standby} * {on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby} = {on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby, on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] off, on[ch1] +, on[ch1] -, on[ch1] standby, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] off, on[ch2] +, on[ch2] -, on[ch2] standby, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] off, on[ch3] +, on[ch3] -, on[ch3] standby, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] off, on[ch4] +, on[ch4] -, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] standby +, on[ch1] standby -, on[ch1] standby standby } = 6 * 8 = 48 test cases in phase 2.

$$T_2 = \bigcup_{path \in TS} \{path\} * w_{CE(path, conf_{init})}^{root}$$

Equation 5.5.2.6 Expected output of every test case.

$$CE_{(ON(CH1), conf_{\{OFF\}})} = \{CH1\} = w_{CE(ON(CH1), conf_{init})}^{root} = w_{CH1}^{root}$$

$$CE_{(ON(CH2), conf_{\{OFF\}})} = \{CH2\} = w_{CE(ON(CH2), conf_{init})}^{root} = w_{CH2}^{root}$$

$$CE_{(ON(CH3), conf_{\{OFF\}})} = \{CH3\} = w_{CE(ON(CH3), conf_{init})}^{root} = w_{CH3}^{root}$$

$$CE_{(ON(CH4), conf_{\{OFF\}})} = \{CH4\} = w_{CE(ON(CH4), conf_{init})}^{root} = w_{CH4}^{root}$$

$$CE_{(OFF, conf_{\{OFF\}})} = \{OFF\} = w_{CE(OFF, conf_{init})}^{root} = w_{OFF}^{root}$$

$$CE_{(+,conf_{\{OFF\}})} = \{OFF\} = W_{CE(+,conf_{init})}^{root} = W_{OFF}^{root}$$

$$CE_{(-,conf_{\{OFF\}})} = \{OFF\} = W_{CE(-,conf_{init})}^{root} = W_{OFF}^{root}$$

$$CE_{(STANDBY,conf_{\{OFF\}})} = \{OFF\} = W_{CE(STANDBY,conf_{init})}^{root} = W_{OFF}^{root}$$

$$CE_{(ON(CH1)_ON(CH1),conf_{\{OFF\}})} = \{ON(CH1)\} = W_{CE(ON(CH1),conf_{init})}^{root} = W_{CH1}^{root}$$

$$CE_{(ON(CH1)_STANDBY,conf_{\{OFF\}})} = \{STANDBY\} = W_{CE(STANDBY,conf_{init})}^{root} = W_{STANDBY}^{root}$$

...

Table 5.5.3.1 Test cases of phase 2.

Applying this to the TV1, the size of the set of test cases for the first phase of the Wp method is 36 and the second one 48, resulting in 84 sequences which is less than that obtained with the W method (240 test cases) for the same customization of the original statechart (section 5.3.3 of this chapter “model 3”). Finally we are going to calculate the positive, negative and redundant test cases for this model as we did with the W method to see which one is more effective in section 5.7 of this chapter.

Combining the set of test cases obtained in the two phases of the Wp method we obtain the set { off, standby, +, -, --, ---, on[ch1] off, on[ch1] standby, on[ch1] +, on[ch1] -, on[ch1] --, on[ch1] ---, on[ch2] off, on[ch2] standby, on[ch2] +, on[ch2] -, on[ch2] --, on[ch2] ---, on[ch3] off, on[ch3] standby, on[ch3] +, on[ch3] -, on[ch3] --, on[ch3] ---, on[ch4] off, on[ch4] standby, on[ch4] +, on[ch4] -, on[ch4] --, on[ch4] ---, on[ch1] standby off, on[ch1] standby standby, on[ch1] standby +, on[ch1] standby -, on[ch1] standby --, on[ch1] standby ---, on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby, on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] off, on[ch1] +, on[ch1] -, on[ch1] standby, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch2] off, on[ch2] +, on[ch2] -, on[ch2] standby, on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch3] off, on[ch3] +, on[ch3] -, on[ch3] standby, on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch4] off, on[ch4] +, on[ch4] -, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] standby +, on[ch1] standby -, on[ch1] standby standby } = 84 test cases for the Wp method.

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that q = concat (p,s) and s is not π then q is also negative, and we call q redundant. The set of positive test cases are $T_{positive} = \{$ on[ch1] off, on[ch1] standby, on[ch1] +, on[ch2] off, on[ch2] standby, on[ch2] +, on[ch2] -, on[ch3] off, on[ch3] standby, on[ch3] +, on[ch3] -, on[ch3] --, on[ch4] off, on[ch4] standby, on[ch4] +, on[ch4] -, on[ch4] --, on[ch4] ---, on[ch1] standby off, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] off, on[ch1] +, on[ch1] standby, on[ch2] off, on[ch2] +, on[ch2] -, on[ch2] standby, on[ch3] off, on[ch3] +, on[ch3] -, on[ch3] standby, on[ch4] off, on[ch4] +, on[ch4] -, on[ch4] standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off } = 43 positive test cases.

$T_{negative} = \{ \text{off, standby, +, -, --, ---, on[ch1] -, on[ch1] --, on[ch1] ---, on[ch2] --, on[ch2] ---, on[ch3] ---, on[ch1] standby standby, on[ch1] standby +, on[ch1] standby -, on[ch1] standby - -, on[ch1] standby - - -, off, +, -, standby, on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] -, on[ch2] on[ch1], on[ch2] on[ch2], on[ch2] on[ch3], on[ch2] on[ch4], on[ch3] on[ch1], on[ch3] on[ch2], on[ch3] on[ch3], on[ch3] on[ch4], on[ch4] on[ch1], on[ch4] on[ch2], on[ch4] on[ch3], on[ch4] on[ch4], on[ch1] standby +, on[ch1] standby -, on[ch1] standby standby } = 84$ calculated test cases – 43 (positive test cases) – 0 (redundant test cases) = 41 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not λ then q is also negative, and we call q redundant. We can see an example: Negative test case; $p = \text{on[ch1] 4}$ and redundant test case; $q = \text{on[ch1] 4 off}$ because “ $q = \text{concat}(p, s)$ ” = “ on[ch1] 4 ” + “ off ” = results in a redundant test case. As we can see there aren’t any redundant test case from the negative test cases calculated above.

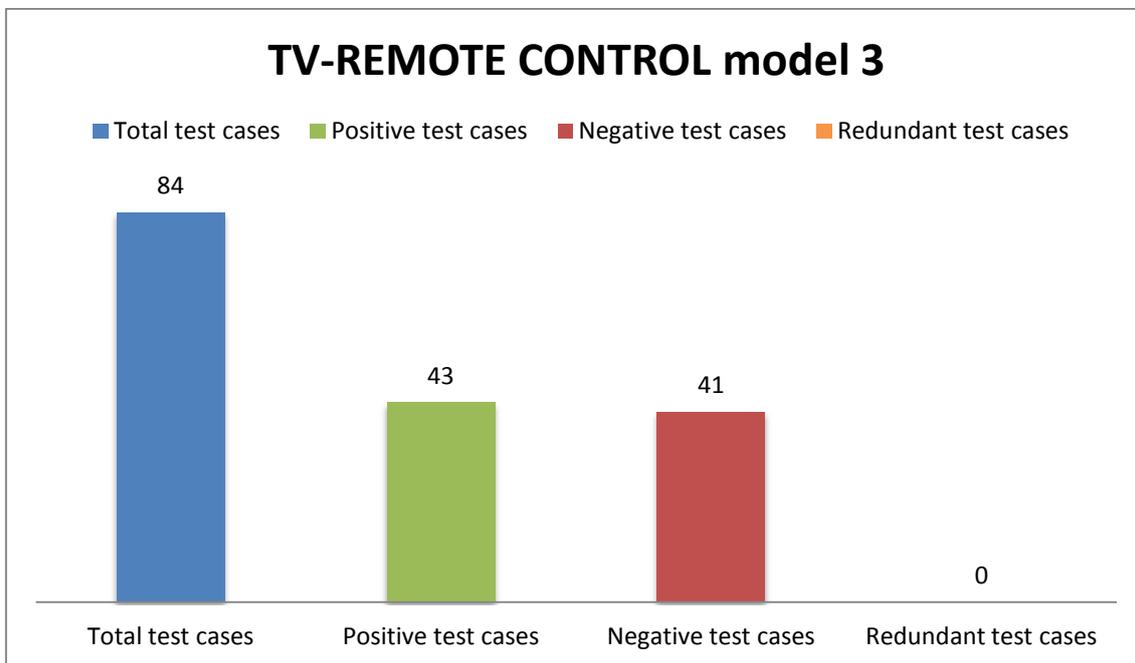


Figure 5.5.3.2. Types of test cases on model 3.

5.5.4 Simplest model with transitions "+" and "-".

Now we are going to calculate the test case basis "TCB" for the flattened model of TV1 without changes between the states of the channels to see what happens with the loop as we can see on Fig. 5.5.3.1. The set of transition labels (denoted by ϕ) is the set of labels of a statechart. $\phi = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. Any desired state starting from the initial one (denoted by C). $C = \{ \lambda, on[ch1], on[ch2], on[ch3], on[ch4], on[ch1] standby\}$. Below we will see that it's impossible to calculate characterization set W.

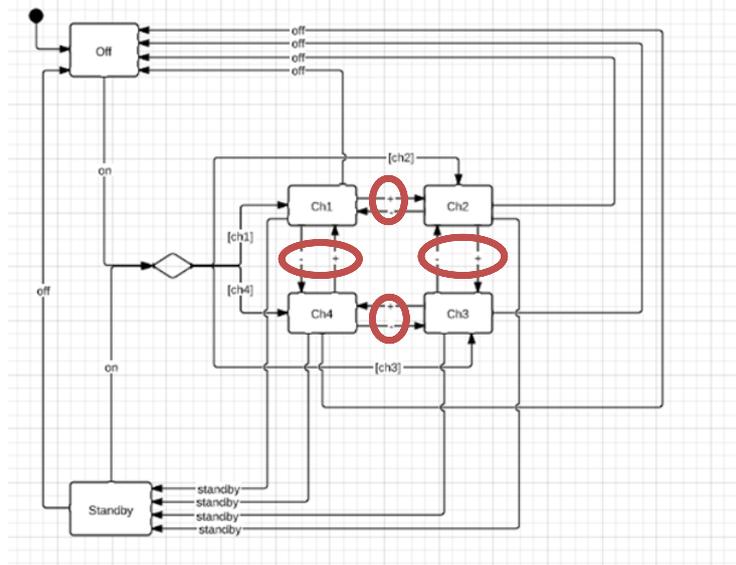


Figure 5.5.4.1 Customized model 4.

Note that in this statechart ϕ , C and W are equals at Φ^M , C^M and W^M because we have not any composite-state. We have explained the meaning and how to calculate them (Φ^M , C^M and W^M) in section 4.1.11 literature review of this thesis "Testing from statecharts using the Wp method" and in [7]. From here varies how to calculate the number of test cases for this method. It consists of two phases and for the first one we need to clarify the Concept of "Conf"; for a configuration conf, an identification set w_{conf}^{root} is a set allowing one to distinguish between conf and all other configurations in a statechart. The concept of configuration comprises sets of states which are left and entered by full compound transitions are called configurations and consist of states a statechart can be in simultaneously. A configuration is uniquely determined by a set of basic states in it. Every state in a flattened statechart corresponds to a configuration in the original one. The most important part of this first phase it is to calculate the small w sets. These are a set allowing one to distinguish between conf and all other configurations in a statechart and we use small w sets to identify states in an implementation applying the same W formula as we use in W method in each state. Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly. The w sets obtained for this first model are $W_{OFF}^{root} = \{on[ch1]\}$, $W_{STANDBY}^{root} = \{off, standby\}$, $W_{CH1}^{root} = \{+, off, standby\}$, $W_{CH2}^{root} = \{+, off, standby\}$, $W_{CH3}^{root} = \{+, off, standby\}$, $W_{CH4}^{root} = \{+, off, standby\}$. As we see we can not differentiate small w sets between different channels so we can not compute the identification set W as occurred when applying the W method with this model in section 3.5.4. The same applies to models discussed in points 5.3.5 and 5.3.6 and shown below in figure

5.5.4.2 in this chapter when applying Wp method. It is impossible to calculate the identification set so we will say that they are not supported by this method.

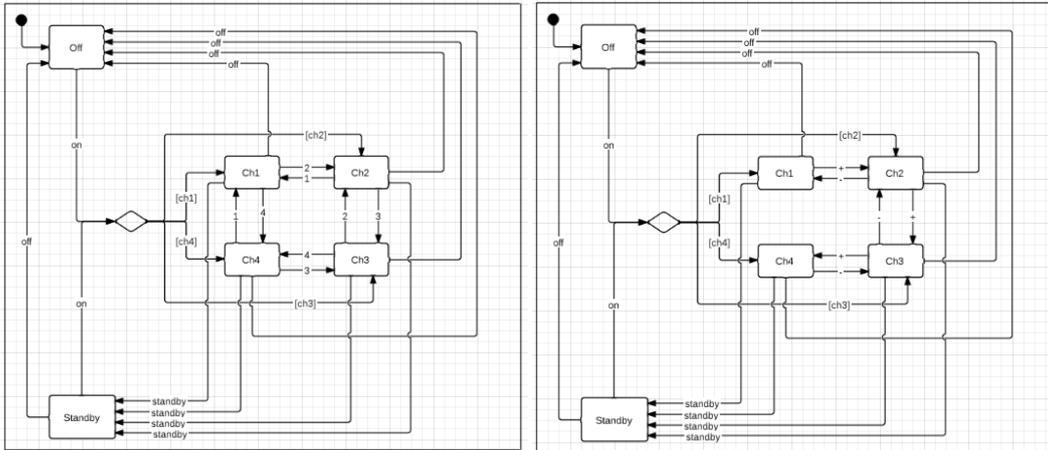


Figure 5.5.4.2 Not supported models 5 and 6.

5.6 Obtained results and conclusions of Wp Method.

As shown in figure 5.6.1, the number of test cases obtained for different customizations of the flattened statechart (Figure 5.5.2) with the Wp method are 84 for “simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4”, 96 for “simplest model change transitions ‘+’ and ‘-’ for numbers only up direction between channels” and 84 for “simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4” while for the customizations of “simplest model change transitions ‘+’ and ‘-’ for numbers” and “simplest model transitions ‘+’ and ‘-’ without loops” cannot be supported by the Wp method as occurred with W method because it is impossible to calculate the small W sets as we have explained in section 5.5.4 for each of the customizations of the flattened statechart. For the first model we have obtained 42.8% of positive test cases, 57.2% of negative test cases and 0% of redundant test cases. On the other hand in the second valid model we have obtained 39.6% of positive test cases, 60.4% and 0% of negative and redundant test cases and finally in the third model we have obtained 51.2% positive test cases, 48.8% negative test cases and 0% redundant test cases.

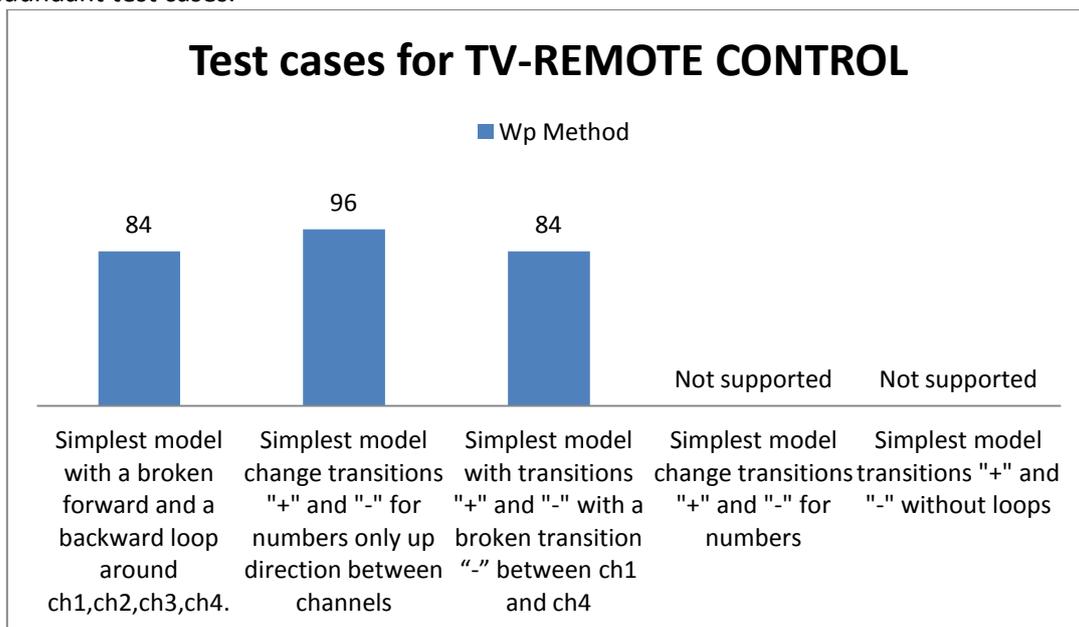


Figure 5.6.1 Test cases for the different models of TV1 statechart with Wp method.

So in the third model (Figure 5.5.3.1) we obtain a larger number of positive test cases than models 1 (Figure 5.5.1.1) and 2 (Figure 5.5.2.1) which can help us most to confirm that a given statechart works properly or not, the basic rule would that it is not the same confirm that a statechart is properly functional with a single test case rather than check it with several, dozens or even hundreds of test cases and the best percentage of positive test cases is for the third customization of the statechart, “simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4” model 3. The other aspect is rating which of the different customizations of the statechart (model 1, 2 or 3) seems more efficient in regard to design as in model 1 “simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4” there is no possibility of changing since channel 4 to 1 while in model 3 we can do it, so we could assume that this action increases the number of negative or redundant test cases, but on the contrary, we get a better percentage of positive test cases with model 3, so with the W method this is the most efficient of all customizations of the flattened statechart with we have worked.

5.7 Comparison of the results obtained with W and Wp methods

The following figures (since 5.7.1 to 5.7.3) show the different types of test cases (positive, negative and redundant) obtained for each of the models that we have worked in this chapter of the thesis. First in figure 5.7.1 shows the results obtained for the model 1 “Simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4”. As we can see with this first model is obtained a similar number of positive test cases with both W and Wp methods. Wp method still much better due to the large amount of negative test cases (222) which appear when we have applied W method to the first model compared to the 48 of the other method. So applying W method we get a greater number of total test cases but most of them become negative, plus get redundant test cases. This does not happen with Wp method with which we get less total number of test cases but almost half of them are positive specifically the 42.8% versus to the 15% of the W method and do not get any redundant.

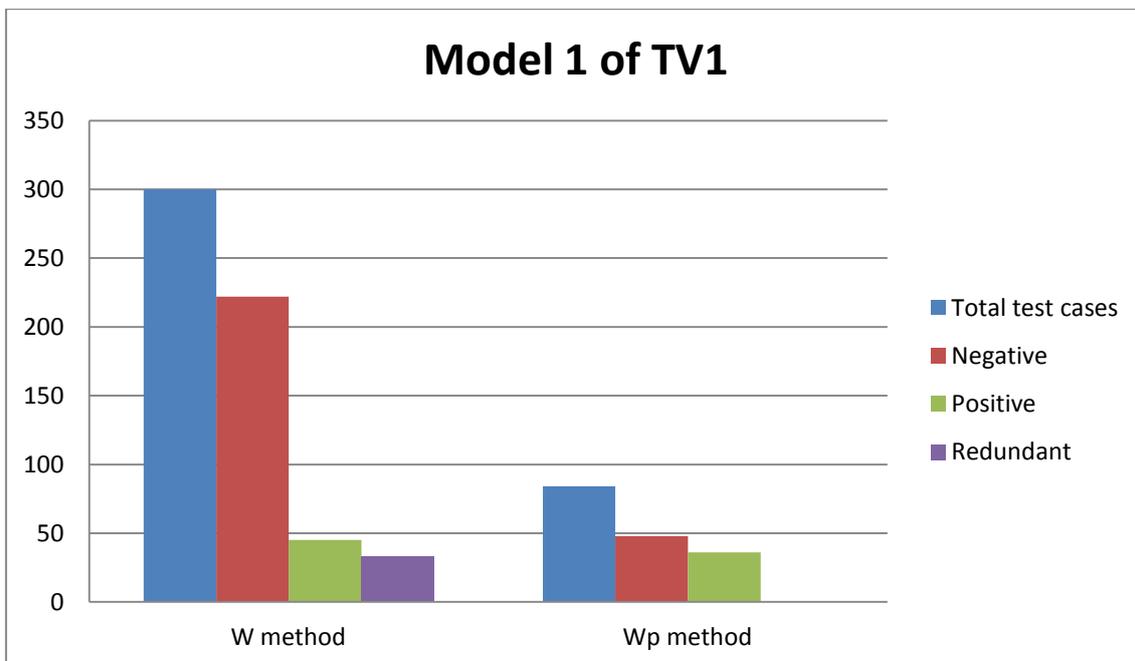


Figure 5.7.1 Simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4

Below (fig. 5.7.2) we shows the results obtained for the model 2 *"Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4"*. As we can see with this second model is obtained a similar number of positive and negative test cases with both W and Wp methods. As happened with model 1 (Figure 5.7.1) Wp method seems to perform better because we get a similar number of positive test cases as we have said above, based on a much smaller total number of test cases (360 of the W method versus the 96 test cases of the Wp).

This means that we get only 5% less of positive test cases from a 73.3% less of total test cases, which in terms of statistics and computation time is much more productive. Finally note that as was the case with model 1, neither redundant test cases are obtained when we apply the Wp method in this variation of our case of study.

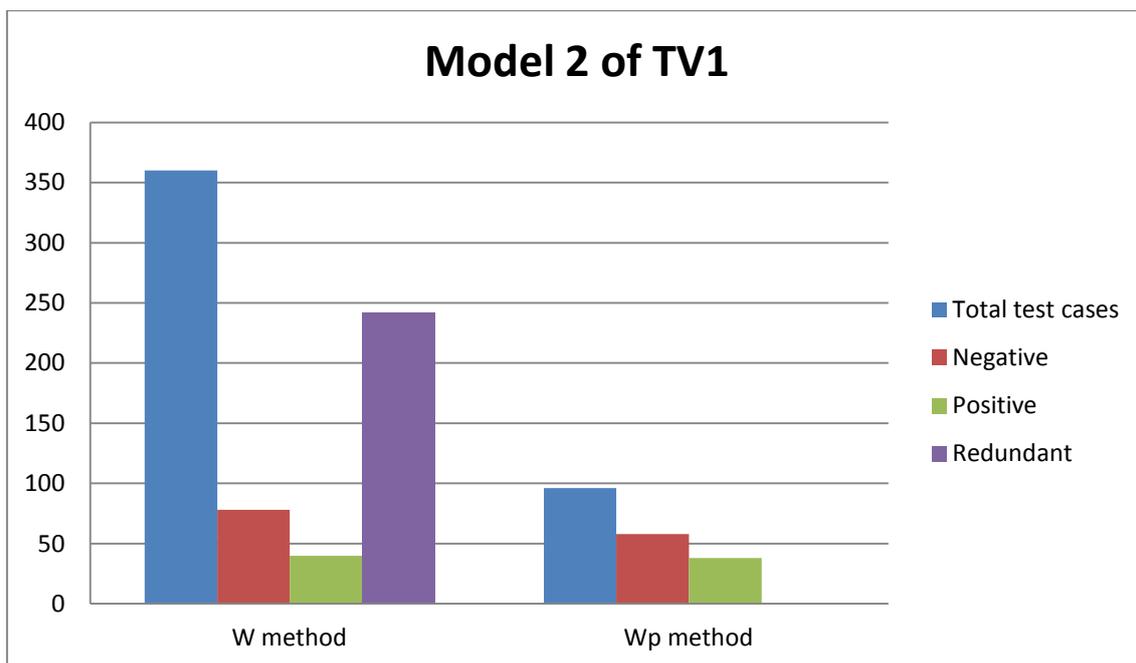


Figure 5.7.2 Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4

The following research shows the results obtained for the model 3 *"Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4"* in figure 5.7.3. This is the model in which the two methods (W & Wp) seem to be more balanced because in both we get a good number of positive test cases from the total number of tests. With W method we have obtained 56 of 240 (23.3%) of positive tests cases which remains low, but is the highest score achieved throughout the case study. The weak point of the W method is again the high number of redundant test cases as was the case with model 2 *"Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4"* because this method (W method) produces a much larger number of total test cases.

With the Wp method are obtained 43 of 84 positive test cases which indicates a percentage higher than 50% of the tests, namely a 51.2%. The rest of test cases for the Wp method are negative and again without redundant test cases as occurred with models 1 and 2 (Fig. 5.7.1 and 5.7.2).

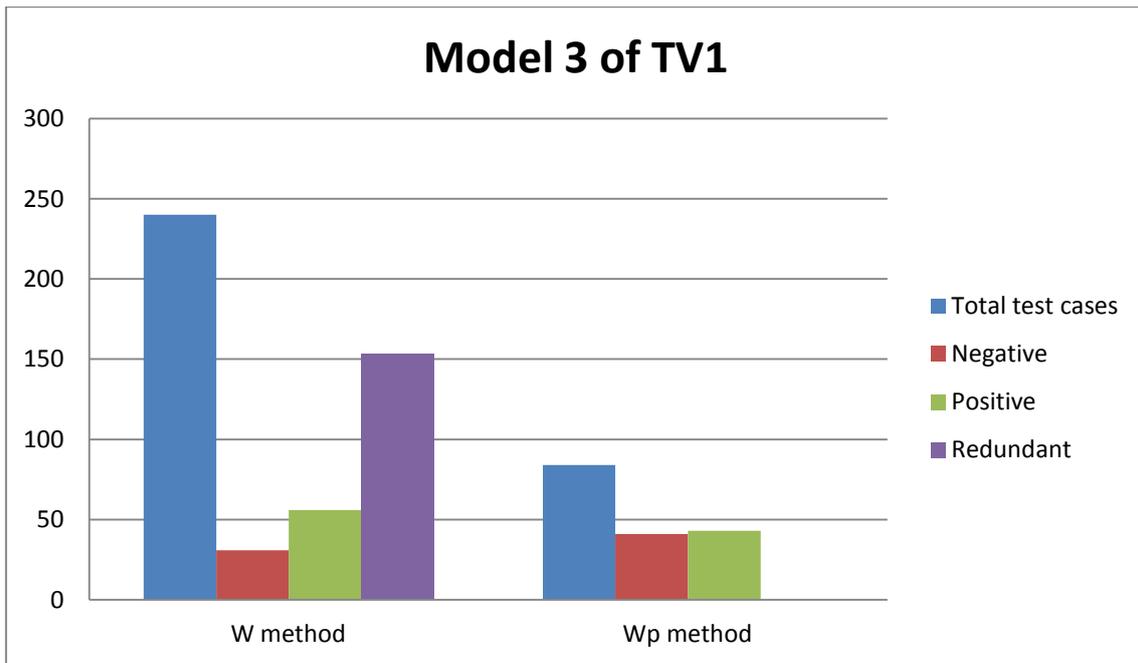


Figure 5.7.3 Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4

Finally as a conclusion of which is the best method and the most appropriate statechart we have shown that in the third model of the Wp method (Figure 5.5.3.1) we obtain a larger number of positive test cases than models 1 (Figure 5.5.1.1) and 2 (Figure 5.5.2.1) which can help us most to confirm that a given statechart works properly or not, the basic rule would that it is not the same confirm that a statechart is properly functional with a single test case rather than check it with several, dozens or even hundreds of test cases and the best percentage of positive test cases is for the third customization of the statechart, "simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4" model 3. Compared with the W method we have shown that this third model analyzed by the Wp method is better than all cases tested with the W method as explained before in this same point.

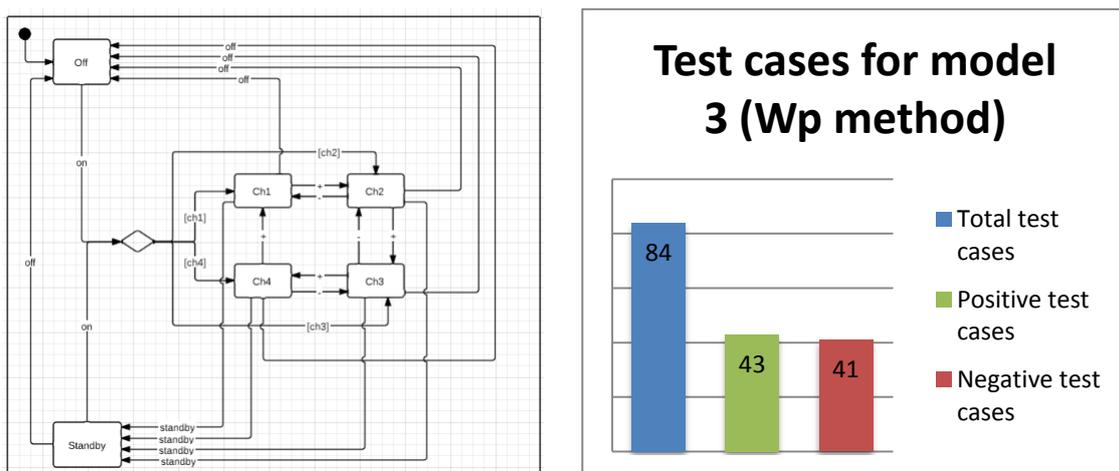


Figure 5.7.4 Statechart and Diagram of the best model after testing with W & Wp methods.

In chapter 6 we will study the statechart with the transition "-" since channel 1 to channel 4 with the dreaded loop with a new algorithm that we have developed and we will analyze the obtained results.

Chapter 6

Definition and analysis of the new algorithm U-method

In this chapter we will explain the steps that we followed for the creation of a new testing algorithm, which we have called U-Method. The following sections of this chapter (6.2.1 and 6.2.2) shall apply this new method on TV1 statechart as was done in Chapter 5 with “W and Wp methods” from [4] and [7] respectively.

Next in section 6.3 we will make a comparison between the number of test cases and the relevance of them between the different versions of the statechart for the U-method and finally in 6.4 we will do the same but for the different methods with which we have tested our case of study in the thesis to check which is the best of all for this statechart.

6.1 Creation of the new testing algorithm U-method

For creating the variables of the new algorithm we have chosen to reuse the variable phi that was used in “W and Wp methods” developed in [7] denominating it as “new phi”, which is all possible states that may occur in the given statechart, as in the previous methods discussed. So we decided to change the name to new phi but still has the same function. Unlike W and Wp algorithms we rely on a single more variable called Tau. This covers all possible transitions that exist in the analyzed statechart since the initial state using the technique transition cover. We decided to work with this technique because there isn’t any functional testing algorithm based on it. We can see the equation of the U-method below. Note that the subscript “n” indicates the number of repetitions that we want to do when we apply the formula, thereby controlling the number of times that we check or not a transition avoiding the dreaded loops.

$$S_n = \tilde{T} * (\{\lambda\} \cup \tilde{\Phi} \cup \tilde{\Phi}^2 \cup \dots \cup \tilde{\Phi}^n)$$

Equation 6.1.1. U-Method

$$S_n = \tilde{T} * (\{\lambda\} \cup \tilde{\Phi}^n)$$

Equation 6.1.2. Simplified equation of U-Method.

Here we make a brief description of the emails exchanged between us for the creation of the new algorithm; “I have two little doubts before to start to apply the formulas of new phi and tau. Which of the two phi think it would be better to use? ‘New Phi = { on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby}’ or ‘New Phi = { on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby, on}’ //I have not clear if adding the transition "on" or not. Which of the two tau think it would be better to use? ‘Tau = { 1, on[ch1], on[ch1] +, on[ch1] ++, on[ch1] +++ , on[ch1] ++++, ... }’ //I add "on[ch1] ++++" to prove the transition (+) since Ch4 to Ch1 or ‘Tau = { 1, on, on +, on ++, on +++ , on ++++, ... }’ //Other doubt it’s if I use only transition "on" or "on[chx]" to create Tau.

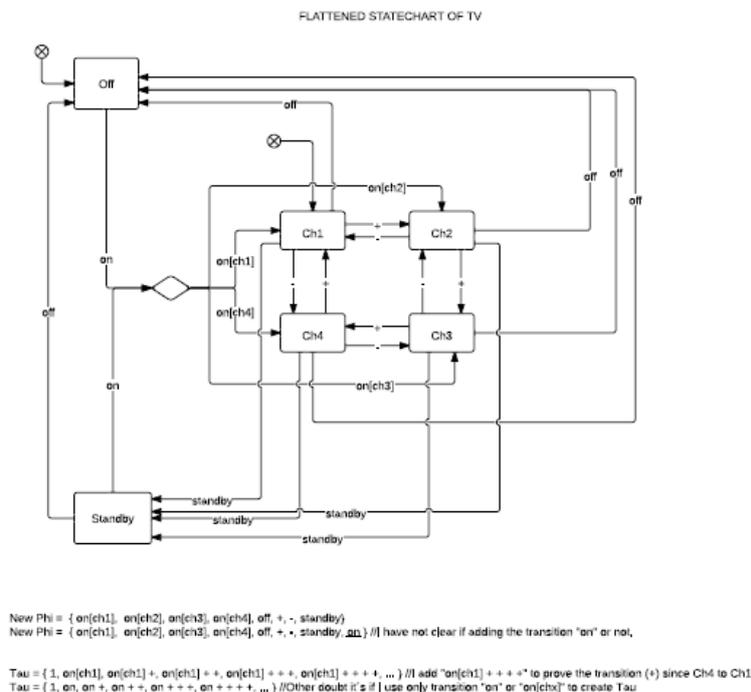


Figure 6.1.1. Content of the mail to create the algorithm.

Reply; "your diagram contains a small error: the transitions leaving a choice pseudostate should not have a trigger, the event, because that whole "compound transition" (for example from state "Off" to "Ch1") is triggered by event "on", which is correctly attached to the transition from "Off" to choice pseudostate. The transitions leaving the choice pseudostate should be labeled with the guards only, hence "[Ch1]", [Ch2], The reason is that when an event fires, a state machine can only move from one state to another, not from a state to a pseudostate. Therefore there should be no "on" element in your phi, only the combinations of the event and the subsequent guard, that is on[ch1], ..., on[ch4]. Same thing for tau. NB you should also remove the initial pseudostate that leads to Ch1: there can be only one such pseudostat."

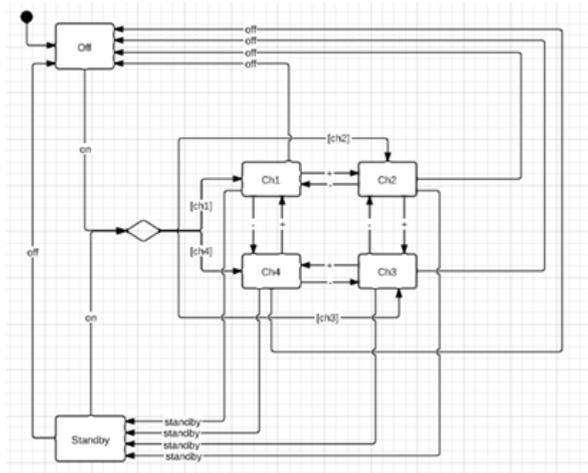


Figure 6.1.1. TV1 statechart to extract new variables.

Calculating the variables for our case of study of figure 6.1.1. we obtain, New Phi = {on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby} and Tau={1, on[ch1], on[ch1] + off on[ch2], on[ch1] ++ off on[ch3], on[ch1] - off on[ch4], on[ch1] + , on[ch1] ++ , on[ch1] +++ , on[ch1] ++++ , on[ch1] - , on[ch1] -- , on[ch1] --- , on[ch1] ---- , on[ch1] standby, on[ch1] - standby, on[ch1] + standby, on[ch1] ++ standby, on[ch1] off, on[ch1] + off, on[ch1] ++ off, on[ch1] - off, on[ch1] standby off, on[ch1] standby on[ch1]}

New Phi = { on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby}	
	Tau = { 1,
transition on[ch1] since Off	on[ch1],
transition on[ch2]	on[ch1] + off on[ch2],
transition on[ch3]	on[ch1] ++ off on[ch3],
transition on[ch4]	on[ch1] - off on[ch4],
transitions +	on[ch1] + ,
	on[ch1] ++ ,
	on[ch1] +++ ,
	on[ch1] ++++ ,
transitions -	on[ch1] - ,
	on[ch1] -- ,
	on[ch1] --- ,
	on[ch1] ---- ,
transitions standby	on[ch1] standby,
	on[ch1] - standby,
	on[ch1] + standby,
	on[ch1] ++ standby,
transitions off	on[ch1] off,
	on[ch1] + off,
	on[ch1] ++ off,
	on[ch1] - off,
	on[ch1] standby off,
transition on since Standby	on[ch1] standby on[ch1]}

Table 6.1.1 Definition of new variables for U-Method.

6.2 Adapt U-method to state machines used to model UIs

We have applied “W and Wp methods” developed in [7] in chapter 5 and now we will apply U-method in subpoints of this section (points 6.2.1 and 6.2.2), after in (6.3) we will discuss the results obtained for this method and we will talk about the conclusions obtained. Finally in section 6.4 we will compare the results obtained for Wp and U method in our case of study. The first thing that we need to do just like we did with W and Wp methods, it’s to eliminate all the substates of the original model. After that, we can work with the simplest model of the statechart applying different changes in it, in section 6.2.1. We are going to work with the original flattened model so U-method can support this model that was impossible to test with W and Wp methods. After that we will work with the version which best results we have obtained with W and Wp methods in section 6.2.2 checking the results obtained in section 6.3.

Step 0: modifications of the initial statechart: create a flattened statechart.

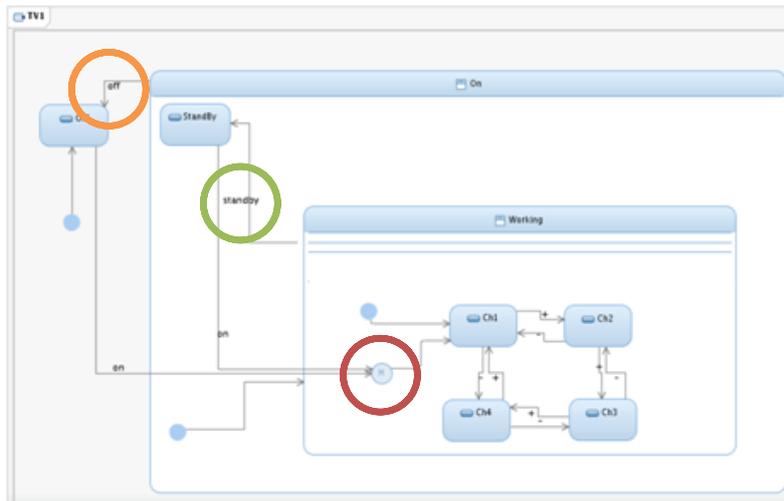


Figure 6.2.1 Original Statechart of TV1.

We will apply modifications on flattened version to calculate the different number of test cases on them. The important changes suffered by the model are the creation of every one of the transitions (“off”, “standby”) between the states of the channels (Ch1, Ch2, Ch3 and Ch4) and the states Off and Standby. With these transitions we managed to remove every substate of the statechart.

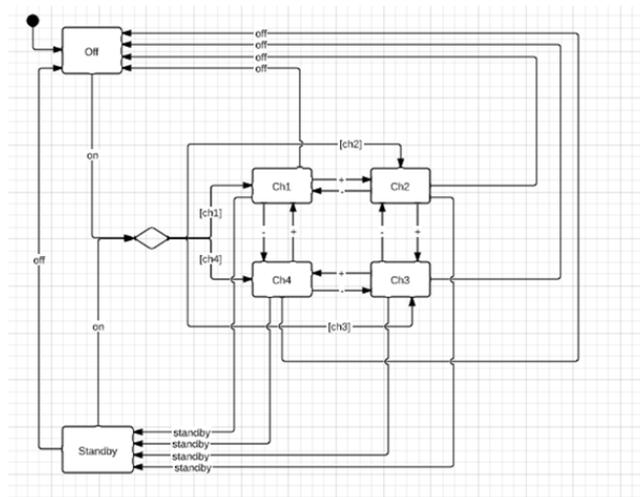


Figure 6.2.2 Flattened statechart.

As discussed in Section 6.2.1 and 6.2.2 we will apply this new method to the original statechart without any type of variation (Figure 6.2.1) because the new method that we have developed supports this original model because it doesn't use the W variable which prevented us to use this model with W and Wp methods. In figure 6.2.2 we will apply U-method to the customization of the statechart with which we have obtained the best results in section 5.5.3 of the previous chapter to check which method is more efficient if the Wp or U, so between W and Wp method we saw that the latter was more efficient. We will do a final check to see if it is also more efficient than the New U-method in section 6.4 of this chapter.

6.2.1 Apply U-method to TV1 original flattened model without customization.

We are going to work with the original flattened model as we can see on Fig. 6.2.1.1 because with our U-method we can work with the model with the feared loop that prevented to apply the W and Wp methods. The set of transition labels (denoted by $\tilde{\Phi}$) is the set of labels of a statechart. $\tilde{\Phi} = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. The other variable that we use in our algorithm is TAW and it is represented by \tilde{T} , it runs all the possible paths in our statechart since the initial state. $\tilde{T} = \{ \lambda, on[ch1], on[ch1] + off on[ch2], on[ch1] ++ off on[ch3], on[ch1] - off on[ch4], on[ch1] +, on[ch1] ++, on[ch1] + + +, on[ch1] + + + +, on[ch1] -, on[ch1] --, on[ch1] ---, on[ch1] ----, on[ch1] standby, on[ch1] - standby, on[ch1] + standby, on[ch1] + + standby, on[ch1] off, on[ch1] + off, on[ch1] + + off, on[ch1] - off, on[ch1] standby off, on[ch1] standby on[ch1] \}$

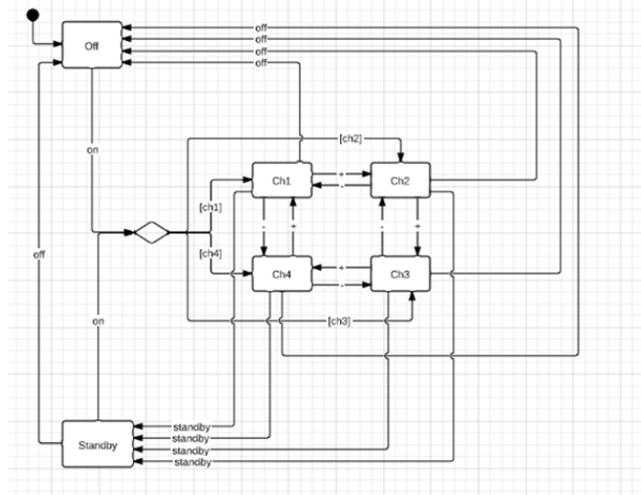


Figure 6.2.1.1 Original flattened statechart.

$$S_n = \tilde{T} * (\tilde{\Phi}^n) = \{ \lambda, on[ch1], on[ch1] + off on[ch2], on[ch1] ++ off on[ch3], on[ch1] - off on[ch4], on[ch1] +, on[ch1] ++, on[ch1] + + +, on[ch1] + + + +, on[ch1] -, on[ch1] --, on[ch1] ---, on[ch1] ----, on[ch1] standby, on[ch1] - standby, on[ch1] + standby, on[ch1] + + standby, on[ch1] off, on[ch1] + off, on[ch1] + + off, on[ch1] - off, on[ch1] standby off, on[ch1] standby on[ch1] \} * \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\} = 23 * 8^1 = 184 \text{ test cases.}$$

Now we can get the test cases and identify the redundant and negative test cases. As we have talked in the point 5.2.2 of the chapter 5 an example of negative test case can be $\{off\}$ and an example of redundant test case $\{off standby\}$.

on[ch2] +, on[ch1] + off on[ch2] -, on[ch1] + off on[ch2] standby, on[ch1] + + off on[ch3] off, on[ch1] + + off on[ch3] +, on[ch1] + + off on[ch3] -, on[ch1] + + off on[ch3] standby, on[ch1] - off on[ch4] off, on[ch1] - off on[ch4] +, on[ch1] - off on[ch4] -, on[ch1] - off on[ch4] standby, on[ch1] + off, on[ch1] + +, on[ch1] + -, on[ch1] + standby, on[ch1] + + off, on[ch1] + + +, on[ch1] + + -, on[ch1] + + standby, on[ch1] + + + off, on[ch1] + + + +, on[ch1] + + + -, on[ch1] + + + standby, on[ch1] - off, on[ch1] - +, on[ch1] - -, on[ch1] - standby, on[ch1] - - off, on[ch1] - - +, on[ch1] - - -, on[ch1] - - standby, on[ch1] - - - off, on[ch1] - - - +, on[ch1] - - - -, on[ch1] - - - standby, on[ch1] - - - - off, on[ch1] - - - - +, on[ch1] - - - - -, on[ch1] - - - - standby, on[ch1] standby on[ch1], on[ch1] standby on[ch2], on[ch1] standby on[ch3], on[ch1] standby on[ch4], on[ch1] standby off, on[ch1] - standby on[ch1], on[ch1] - standby on[ch2], on[ch1] - standby on[ch3], on[ch1] - standby on[ch4], on[ch1] - standby off, on[ch1] + standby on[ch1], on[ch1] + standby on[ch2], on[ch1] + standby on[ch3], on[ch1] + standby on[ch4], on[ch1] + standby off, on[ch1] + + standby on[ch1], on[ch1] + + standby on[ch2], on[ch1] + + standby on[ch3], on[ch1] + + standby on[ch4], on[ch1] + + standby off, on[ch1] off on[ch1], on[ch1] off on[ch2], on[ch1] off on[ch3], on[ch1] off on[ch4], on[ch1] + off on[ch1], on[ch1] + off on[ch2], on[ch1] + off on[ch3], on[ch1] + off on[ch4], on[ch1] + + off on[ch1], on[ch1] + + off on[ch2], on[ch1] + + off on[ch3], on[ch1] + + off on[ch4], on[ch1] - off on[ch1], on[ch1] - off on[ch2], on[ch1] - off on[ch3], on[ch1] - off on[ch4], on[ch1] standby off on[ch1], on[ch1] standby off on[ch2], on[ch1] standby off on[ch3], on[ch1] standby off on[ch4], on[ch1] standby on[ch1] off, on[ch1] standby on[ch1] +, on[ch1] standby on[ch1] -, on[ch1] standby on[ch1] standby } = 106 positive test cases.

$T_{negative} = \{ \text{off, +, -, standby, on[ch1] on[ch1], on[ch1] on[ch2], on[ch1] on[ch3], on[ch1] on[ch4], on[ch1] + off on[ch2] on[ch1], on[ch1] + off on[ch2] on[ch2], on[ch1] + off on[ch2] on[ch3], on[ch1] + off on[ch2] on[ch4], on[ch1] + + off on[ch3] on[ch1], on[ch1] + + off on[ch3] on[ch2], on[ch1] + + off on[ch3] on[ch3], on[ch1] + + off on[ch3] on[ch4], on[ch1] - off on[ch4] on[ch1], on[ch1] - off on[ch4] on[ch2], on[ch1] - off on[ch4] on[ch3], on[ch1] - off on[ch4] on[ch4], on[ch1] + on[ch1], on[ch1] + on[ch2], on[ch1] + on[ch3], on[ch1] + on[ch4], on[ch1] + + on[ch1], on[ch1] + + on[ch2], on[ch1] + + on[ch3], on[ch1] + + on[ch4], on[ch1] + + + on[ch1], on[ch1] + + + on[ch2], on[ch1] + + + on[ch3], on[ch1] + + + on[ch4], on[ch1] + + + + on[ch1], on[ch1] + + + + on[ch2], on[ch1] + + + + on[ch3], on[ch1] + + + + on[ch4], on[ch1] - on[ch1], on[ch1] - on[ch2], on[ch1] - on[ch3], on[ch1] - on[ch4], on[ch1] - - on[ch1], on[ch1] - - on[ch2], on[ch1] - - on[ch3], on[ch1] - - on[ch4], on[ch1] - - - on[ch1], on[ch1] - - - on[ch2], on[ch1] - - - on[ch3], on[ch1] - - - on[ch4], on[ch1] - - - - on[ch1], on[ch1] - - - - on[ch2], on[ch1] - - - - on[ch3], on[ch1] - - - - on[ch4], on[ch1] standby +, on[ch1] standby -, on[ch1] standby standby, on[ch1] - standby +, on[ch1] - standby -, on[ch1] - standby standby, on[ch1] + standby +, on[ch1] + standby -, on[ch1] + standby standby, on[ch1] + + standby +, on[ch1] + + standby -, on[ch1] + + standby standby, on[ch1] off off, on[ch1] off +, on[ch1] off -, on[ch1] off standby, on[ch1] + off off, on[ch1] + off +, on[ch1] + off -, on[ch1] + off standby, on[ch1] + + off off, on[ch1] + + off +, on[ch1] + + off -, on[ch1] + + off standby, on[ch1] - off off, on[ch1] - off +, on[ch1] - off -, on[ch1] - off standby, on[ch1] standby off off, on[ch1] standby off +, on[ch1] standby off -, on[ch1] standby off standby, on[ch1] standby on[ch1] on[ch1], on[ch1] standby on[ch1] on[ch2], on[ch1] standby on[ch1] on[ch3], on[ch1] standby on[ch1] on[ch4] \}$ = 184 – 106 (positive test cases) – 0 (redundant test cases) = 78 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that q=concat(p,s) and s is not λ then q is also negative, and we call q redundant. We can see an example: Negative test case; p = on[ch1] 4 and redundant test case; q = on[ch1] 4 off because “q = concat (p, s)” = “on[ch1] 4” + “off” = results in a redundant test case. As we can see there aren’t any redundant test case from the negative test cases calculated above.

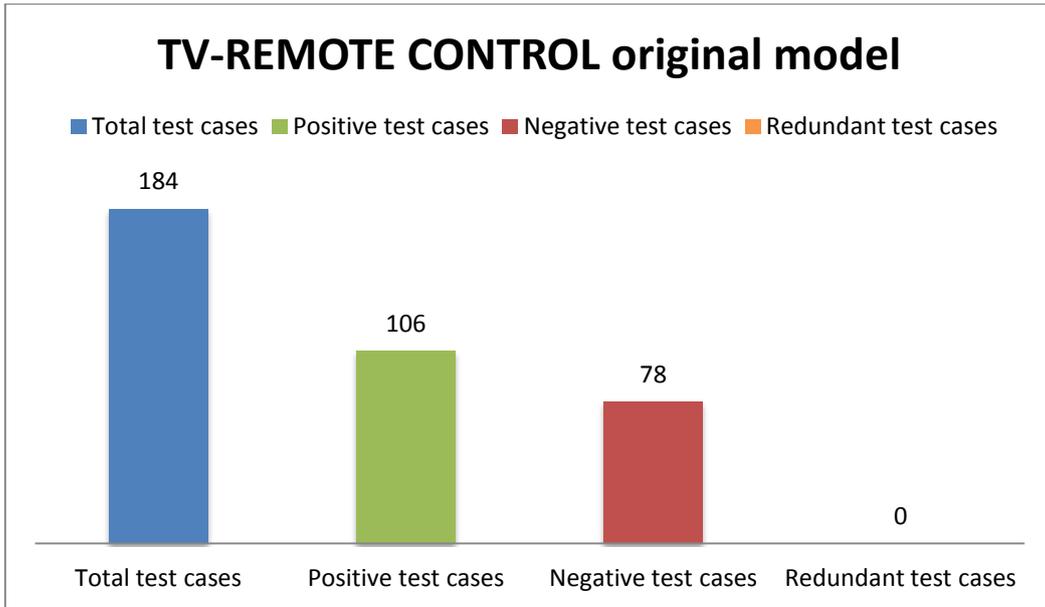


Figure 6.2.1.2. Obtained test cases for the original flattened statechart of TV1.

6.2.2 Apply U-method to TV1 to model 3 of chapter 5

In this point we are going to work with the third model of the original statechart as we can see on Fig. 6.2.2.1 to see the differences between U and Wp methods with the model which we have obtained the best results for the methods W and Wp. As these methods the set of transition labels (denoted by $\tilde{\Phi}$) is the set of labels of a statechart. $\tilde{\Phi} = \{on[ch1], on[ch2], on[ch3], on[ch4], off, +, -, standby\}$. The other variable that we use in our algorithm is TAW and it is represented by \tilde{T} , it runs all the possible paths in our statechart since the initial state. $\tilde{T} = \{\lambda, on[ch1], on[ch1] + off on[ch2], on[ch1] ++ off on[ch3], on[ch1] +++ off on[ch4], on[ch1] +, on[ch1] ++, on[ch1] +++, on[ch1] ++++, on[ch1] off on[ch4] -, on[ch1] off on[ch4] --, on[ch1] off on[ch4] ---, on[ch1] standby, on[ch1] + standby, on[ch1] ++ standby, on[ch1] +++ standby, on[ch1] off, on[ch1] + off, on[ch1] ++ off, on[ch1] +++ off, on[ch1] standby off, on[ch1] standby on[ch1]\}$

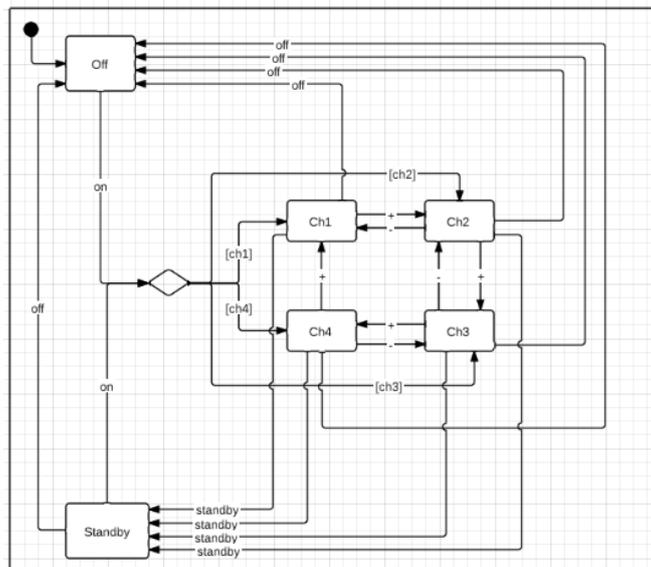


Figure 6.2.2.1 Simplest model with transitions "+" and "-" with a broken transition "-" between ch1 and ch4.

+++ off on[ch1], on[ch1]+++ off on[ch2], on[ch1]+++ off on[ch3], on[ch1]+++ off on[ch4], on[ch1]+++ off off, on[ch1]+++ off +, on[ch1]+++ off -, on[ch1]+++ off standby, on[ch1] standby off on[ch1], on[ch1] standby off on[ch2], on[ch1] standby off on[ch3], on[ch1] standby off on[ch4], on[ch1] standby off off, on[ch1] standby off +, on[ch1] standby off -, on[ch1] standby off standby, on[ch1] standby on[ch1] on[ch1], on[ch1] standby on[ch1] on[ch2], on[ch1] standby on[ch1] on[ch3], on[ch1] standby on[ch1] on[ch4], on[ch1] standby on[ch1] off, on[ch1] standby on[ch1] +, on[ch1] standby on[ch1] -, on[ch1] standby on[ch1] standby } = $S_1 = 22 * 8^1 = 176$ test cases

As we see when using an automated algorithm there are a lot of negative test cases. We have differentiated the set T of test cases that are positive from those that are negative. In next step, we will identify the redundant test cases among which are negative (on the basis of the length of a path: if path p is negative, and path q is such that $q = \text{concat}(p,s)$ and s is not π then q is also negative, and we call q redundant. $T_{positive} = \{ \text{on[ch1]}, \text{on[ch2]}, \text{on[ch3]}, \text{on[ch4]}, \text{on[ch1] off}, \text{on[ch1] +}, \text{on[ch1] standby}, \text{on[ch1] + off on[ch2] off}, \text{on[ch1] + off on[ch2] +}, \text{on[ch1] + off on[ch2] -}, \text{on[ch1] + off on[ch2] standby}, \text{on[ch1] ++ off on[ch3] off}, \text{on[ch1] ++ off on[ch3] +}, \text{on[ch1] ++ off on[ch3] -}, \text{on[ch1] ++ off on[ch3] standby}, \text{on[ch1] ++ + off on[ch4] off}, \text{on[ch1] ++ + off on[ch4] standby}, \text{on[ch1] + off}, \text{on[ch1] ++}, \text{on[ch1] + -}, \text{on[ch1] + standby}, \text{on[ch1] ++ off}, \text{on[ch1] +++}, \text{on[ch1] ++ -}, \text{on[ch1] ++ standby}, \text{on[ch1] ++ + off}, \text{on[ch1] ++ ++}, \text{on[ch1] ++ + -}, \text{on[ch1] ++ + standby}, \text{on[ch1] ++ + + off}, \text{on[ch1] ++ + + +}, \text{on[ch1] ++ + + standby}, \text{on[ch1] off on[ch4] - off}, \text{on[ch1] off on[ch4] - +}, \text{on[ch1] off on[ch4] - -}, \text{on[ch1] off on[ch4] - standby}, \text{on[ch1] off on[ch4] - - off}, \text{on[ch1] off on[ch4] - - +}, \text{on[ch1] off on[ch4] - - -}, \text{on[ch1] off on[ch4] - - standby}, \text{on[ch1] off on[ch4] - - - off}, \text{on[ch1] off on[ch4] - - - +}, \text{on[ch1] off on[ch4] - - - standby}, \text{on[ch1] standby on[ch1]}, \text{on[ch1] standby on[ch2]}, \text{on[ch1] standby on[ch3]}, \text{on[ch1] standby on[ch4]}, \text{on[ch1] + standby on[ch1]}, \text{on[ch1] + standby on[ch2]}, \text{on[ch1] + standby on[ch3]}, \text{on[ch1] + standby on[ch4]}, \text{on[ch1] + standby off}, \text{on[ch1] ++ standby on[ch1]}, \text{on[ch1] ++ standby on[ch2]}, \text{on[ch1] ++ standby on[ch3]}, \text{on[ch1] ++ standby on[ch4]}, \text{on[ch1] ++ + standby on[ch1]}, \text{on[ch1] ++ + standby on[ch2]}, \text{on[ch1] ++ + standby on[ch3]}, \text{on[ch1] ++ + standby on[ch4]}, \text{on[ch1] ++ + standby off}, \text{on[ch1] off on[ch1]}, \text{on[ch1] off on[ch2]}, \text{on[ch1] off on[ch3]}, \text{on[ch1] off on[ch4]}, \text{on[ch1] + off on[ch1]}, \text{on[ch1] + off on[ch2]}, \text{on[ch1] + off on[ch3]}, \text{on[ch1] + off on[ch4]}, \text{on[ch1] ++ off on[ch1]}, \text{on[ch1] ++ off on[ch2]}, \text{on[ch1] ++ off on[ch3]}, \text{on[ch1] ++ off on[ch4]}, \text{on[ch1] ++ + off on[ch1]}, \text{on[ch1] ++ + off on[ch2]}, \text{on[ch1] ++ + off on[ch3]}, \text{on[ch1] ++ + off on[ch4]}, \text{on[ch1] standby off on[ch1]}, \text{on[ch1] standby off on[ch2]}, \text{on[ch1] standby off on[ch3]}, \text{on[ch1] standby off on[ch4]}, \text{on[ch1] standby on[ch1] off}, \text{on[ch1] standby on[ch1] +}, \text{on[ch1] standby on[ch1] standby} \} = 82$ positive test cases

$T_{negative} = \{ \text{off}, +, -, \text{standby}, \text{on[ch1] on[ch1]}, \text{on[ch1] on[ch2]}, \text{on[ch1] on[ch3]}, \text{on[ch1] on[ch4]}, \text{on[ch1] -}, \text{on[ch1] + off on[ch2] on[ch1]}, \text{on[ch1] + off on[ch2] on[ch2]}, \text{on[ch1] + off on[ch2] on[ch3]}, \text{on[ch1] + off on[ch2] on[ch4]}, \text{on[ch1] ++ off on[ch3] on[ch1]}, \text{on[ch1] ++ off on[ch3] on[ch2]}, \text{on[ch1] ++ off on[ch3] on[ch3]}, \text{on[ch1] ++ off on[ch3] on[ch4]}, \text{on[ch1] ++ + off on[ch4] on[ch1]}, \text{on[ch1] ++ + off on[ch4] on[ch2]}, \text{on[ch1] ++ + off on[ch4] on[ch3]}, \text{on[ch1] ++ + off on[ch4] on[ch4]}, \text{on[ch1] ++ + off on[ch4] +}, \text{on[ch1] ++ + off on[ch4] -}, \text{on[ch1] + on[ch1]}, \text{on[ch1] + on[ch2]}, \text{on[ch1] + on[ch3]}, \text{on[ch1] + on[ch4]}, \text{on[ch1] + + on[ch1]}, \text{on[ch1] + + on[ch2]}, \text{on[ch1] + + on[ch3]}, \text{on[ch1] + + on[ch4]}, \text{on[ch1] + + + on[ch1]}, \text{on[ch1] + + + on[ch2]}, \text{on[ch1] + + + on[ch3]}, \text{on[ch1] + + + on[ch4]}, \text{on[ch1] + + + + on[ch1]}, \text{on[ch1] + + + + on[ch2]}, \text{on[ch1] + + + + on[ch3]}, \text{on[ch1] + + + + on[ch4]}, \text{on[ch1] + + + + -}, \text{on[ch1] off on[ch4] - on[ch1]}, \text{on[ch1] off on[ch4] - on[ch2]}, \text{on[ch1] off on[ch4] - on[ch3]}, \text{on[ch1] off on[ch4] - on[ch4]}, \text{on[ch1] off on[ch4] - - on[ch1]}, \text{on[ch1] off on[ch4] - - on[ch2]}, \text{on[ch1] off on[ch4] - - on[ch3]}, \text{on[ch1] off on[ch4] - - on[ch4]}, \text{on[ch1] off on[ch4] - - - on[ch1]}, \text{on[ch1] off on[ch4] - - - on[ch2]}, \text{on[ch1] off on[ch4] - - - on[ch3]}, \text{on[ch1] off on[ch4] - - - on[ch4]}, \text{on[ch1] off on[ch4] - - - -}, \text{on[ch1] standby +}, \text{on[ch1] standby -}, \text{on[ch1] standby} \}$

standby, on[ch1] + standby +, on[ch1] + standby -, on[ch1] + standby standby, on[ch1] + + standby +, on[ch1] + + standby -, on[ch1] + + standby standby, on[ch1] + + + standby +, on[ch1] + + + standby -, on[ch1] + + + standby standby, on[ch1] off off, on[ch1] off +, on[ch1] off -, on[ch1] off standby, on[ch1] + off off, on[ch1] + off +, on[ch1] + off -, on[ch1] + off standby, on[ch1] + + off off, on[ch1] + + off +, on[ch1] + + off -, on[ch1] + + off standby, on[ch1] + + + off off, on[ch1] + + + off +, on[ch1] + + + off -, on[ch1] + + + off standby, on[ch1] standby off off, on[ch1] standby off +, on[ch1] standby off -, on[ch1] standby off standby, on[ch1] standby on[ch1], on[ch1] standby on[ch1] on[ch2], on[ch1] standby on[ch1] on[ch3], on[ch1] standby on[ch1] on[ch4], on[ch1] standby on[ch1] - } = 176 - 82 (positive test cases) - 0(redundant test cases) = 94 negative test cases.

And also those that are redundant (on the basis of the length of a path: if path p is negative, and path q is such that q=concat(p,s) and s is not λ then q is also negative, and we call q redundant. We can see an example: Negative test case; p = on[ch1] 4 and redundant test case; q = on[ch1] 4 off because “q = concat (p, s)” = “on[ch1] 4” + “off” = results in a redundant test case. As we can see there aren’t any redundant test case from the negative test cases calculated above.

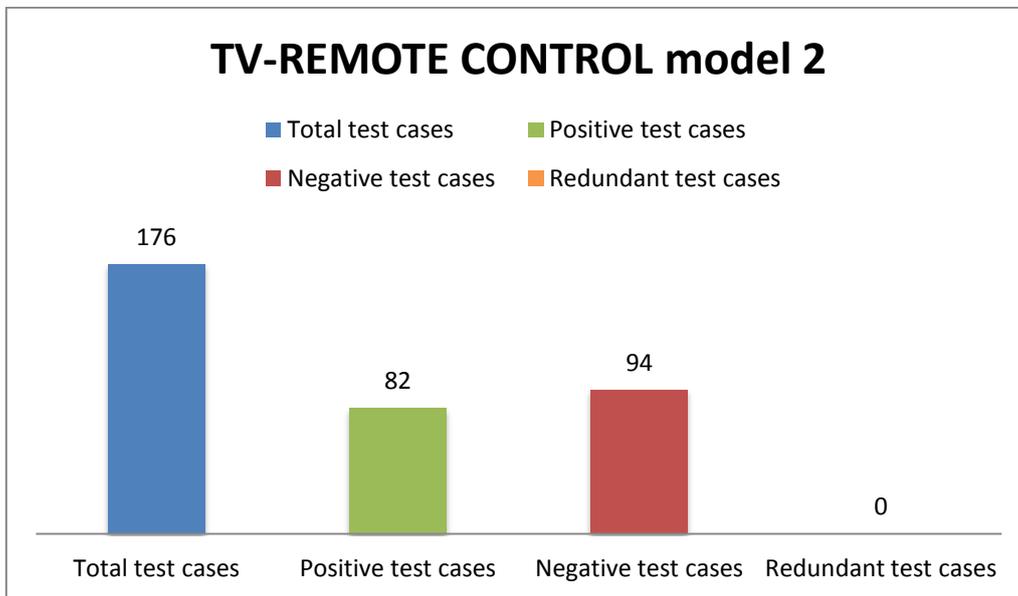


Figure 6.2.2.2 Obtained test cases for the simplest model with transitions "+" and "-" with a broken transition "- between ch1 and ch4.

6.3 Obtained results and conclusions for the U-Method.

As shown in figure 6.3.1, the number of test cases obtained for the two versions of the case of study analyzed in this chapter for the U-method are 184 for “original flattened statechart” and 176 for “simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4” while for the customizations of “simplest model change transitions ‘+’ and ‘-’ for numbers” and “simplest model transitions ‘+’ and ‘-’ without loops” (figures 5.2.4 and 5.2.5) can be supported by the U-method unlike with W and Wp algorithms because with our equation it is possible to calculate the number of test cases for them but we have omitted these studies to be less functional models that analyzed customizations in section 6.2. As with model one (fig. 5.2.1) “simplest model with a broken forward and a backward loop around ch1,ch2,ch3,ch4.” and model two (fig 5.2.2) “simplest model change transitions “+” and “-” for numbers only up direction between channels” there was possible to calculate them with W, Wp and U methods but we have omitted them for the same reason and because we want to

compare in the next section (6.4) the statechart with which we have obtained the best results with the method Wp to see the differences between this and the U method. For the first model we have obtained 57.6% of positive test cases, 42.4% of negative test cases and 0% of redundant test cases. With the second model we have obtained 46.6% of positive test cases, 53.4% and 0% of negative and redundant test cases.

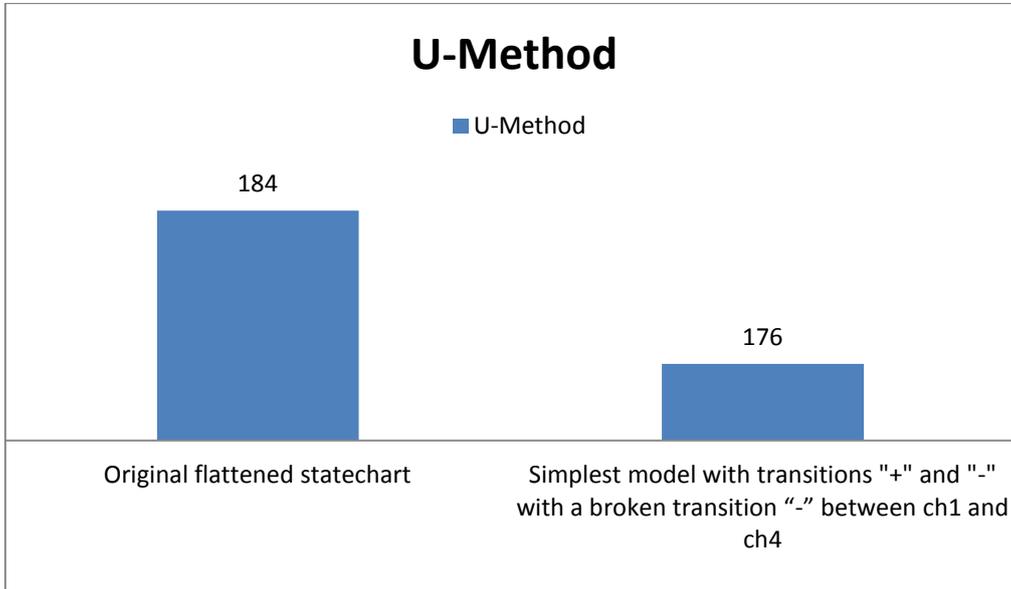


Figure 6.3.1 Test cases for the different models of TV1 statechart with Wp method.

So in the first model (Figure 6.2.1.1) we obtain a larger number of positive test cases than model 2 (Figure 6.2.2.1) which can help us most to confirm that a given statechart works properly or not, the basic rule would that it is not the same confirm that a statechart is properly functional with a single test case rather than check it whit several, dozens or even hundreds of test cases and the best percentage of positive test cases is for the first model of the statechart, *“original flattened statechart”*. The other aspect is rating which of the different customizations of the statechart (model 1, or 2) seems more efficient in regard to design as in model 2 *“simplest model with transitions ‘+’ and ‘-’ with a broken transition ‘-’ between ch1 and ch4”* there is no possibility of changing since channel 1 to 4 while in model 1 we can do it. So we can say that model 1 is the best model in all aspects as we have a greater number of positive test cases, that is what we seek when we testing an application, and also it work with all existing transitions in the statechart.

6.4 Comparison of the results obtained with W, Wp and U methods

The following figures (6.4.1 and 6.4.2) show the different types of test cases (positive, negative and redundant) obtained for each of the models that we have worked in this chapter of the thesis. First in figure 6.4.1 we show the results obtained for the model 1 “*original flattened statechart*”. As we can see with this first model we only obtain results for U-method because W and Wp methods cannot support this model for the problems with the loop created between the states of the different channel. Logically can not make any comparison between the different methods studied in this work, but we will analyze the obtained results. So applying U-method we 57.6% of positive, 42.4% of negative and any redundant test case. Compared to the other models studied in Chapters 5 and 6 of our case study this is the best of all the percentages obtained in positive test cases which indicates that this method is efficiently applied to model 1 “*original flattened statechart*”.

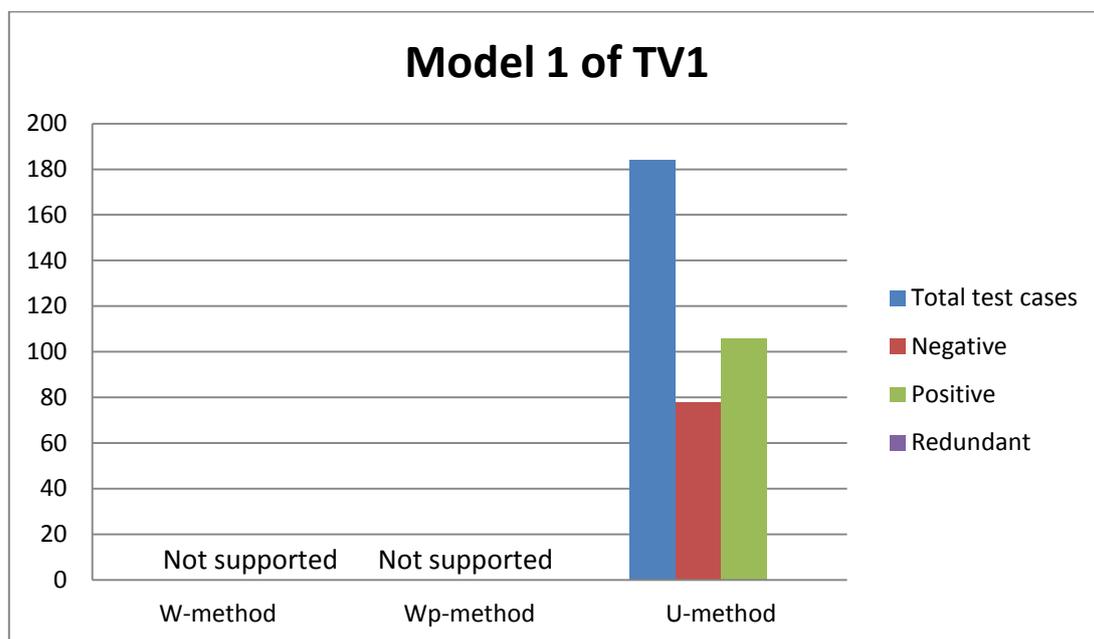


Figure 6.4.1. Obtained results for the original flattened statechart.

In the following figure (6.4.2) we show the different types of test cases (positive, negative and redundant) obtained for the model that we have worked in chapters 5 (section 5.2.3) and 6 (section 6.2.2) of the thesis (*simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4*) and with which we have obtained the best results for the W and Wp methods to see the differences between them and the results obtained with U method. This is the model in which W & Wp methods seem to be more balanced because in both we get a good number of positive test cases from the total number of tests.

As we saw in chapter 5 with W method we have obtained 56 of 240 (23.3%) of positive tests cases which remains low, but is the highest score achieved throughout the case study. The weak point of the W method is again the high number of redundant test cases as was the case with model 2 "*Simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4*" because this method (W method) produces a much larger number of total test cases. With the Wp method we have obtained 43 of 84 positive test cases which indicates a percentage higher than 50% of the tests, namely a 51.2%. The rest of test cases for the Wp method are negative and there aren't any redundant test case. Finally with U method we get a good percentage of positive test cases, namely 46.6%, but remember that we start from a greater number of test cases making it difficult to obtain a good percentage of positive test cases but our method produces good results.

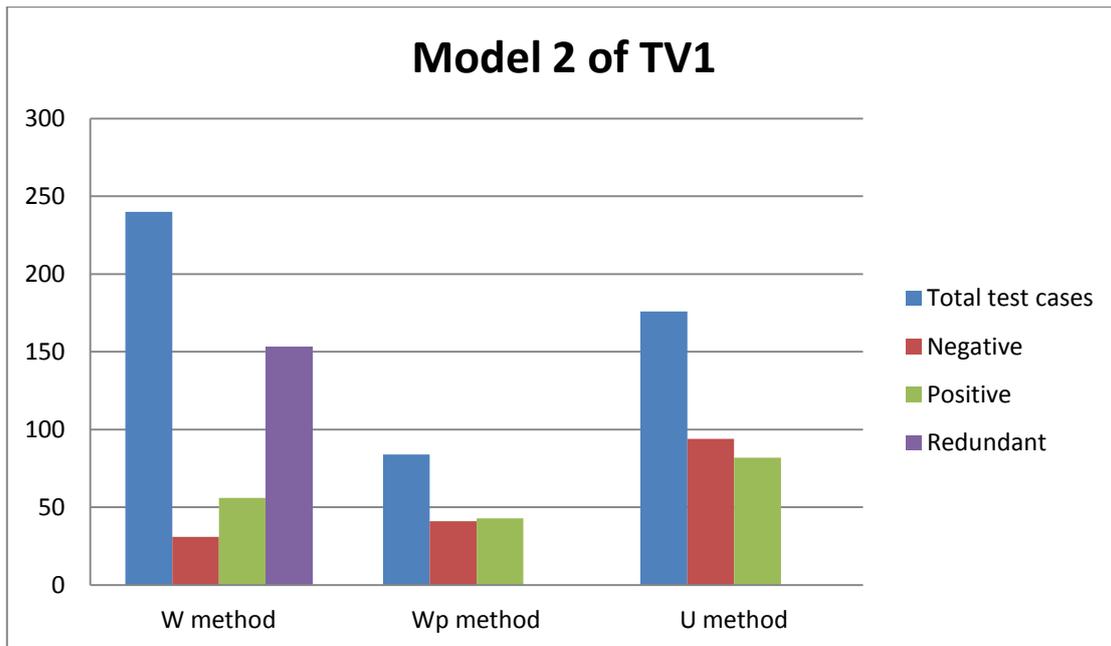


Figure 6.4.2. Obtained results for the simplest model with transitions '+' and '-' with a broken transition '-' between ch1 and ch4. (Model 3 in chapter 5)

As we saw in Chapter 5, for this model of our case of study, between W and Wp methods [7] the latter is the more efficient. Now we are going to discuss if it is more appropriate than U-method. Initially seems that W its more efficient than U method because it has more positive from less test cases, and this is the objective of the program testing, but U-method is able to get a high number of positives, namely 46.6%, from a much larger number of total test cases (176) and this is a good characteristic so we can see that with W-method we have also obtained a high number of total test cases but the percentage of positives is only 23.3%, it is a very poor result.

Therefore between Wp and U methods we could opt for either but the fact that U-method get more positive test cases makes us choose it as the most appropriate method because this method can help us most to confirm that a given statechart works properly or not, the basic rule would that it is not the same confirm that a statechart is properly functional with a single test case rather than check it whit several, dozens or even hundreds of test cases. In this case we do not get the best percentage of positive test cases as occurred in the comparisons performed in Chapter 5 but we obtain a greater number of positive with a good percentage of them.

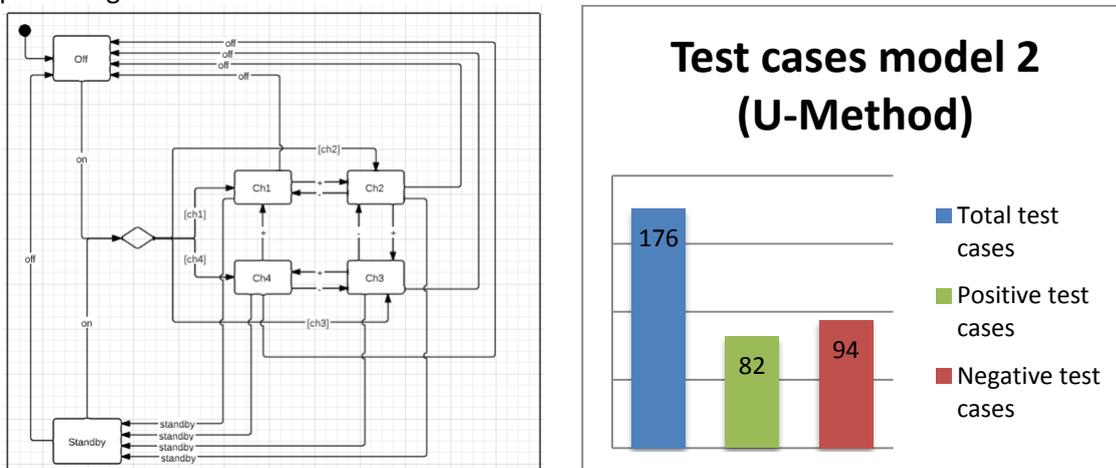


Figure 6.4.3 Statechart and Diagram of the best model after testing with W & Wp methods.

CHAPTER 7

Applying the methods to the statechart with Bugs.

In this chapter we will modify the customization 1 of the statechart used in chapter 5 to comprove that the positive test cases obtained with W-method developed in [4] are useful and can be detect the different bugs.

There are seven types of bugs, since eliminate the transition 1 since channel 2 to channel 1 to check that exist one or more positive test case that try to do the action of execute the eliminated transition. Another bug is to create a new channel 5 or create a new transition "x" to see the behavior of the algorithm amongst others. Every one of them are checked and has sough in each case the test case that identifies the created bug.

7. Changes in our case of study.

In this section we are going to modify one of the statecharts with which we have worked in the previous chapters to assess the value of the different test cases that we have obtained. We will apply changes to the model 1 for will apply different types of bugs to the statechart and show that the obtained test cases detect this bug.

7.1 Errors on Model 1 obtained with W Method.

First we are going to apply different changes on the customization 1 of the statechart analyzed in chapter 5, we can see it without bugs below in figure 7.1.1.

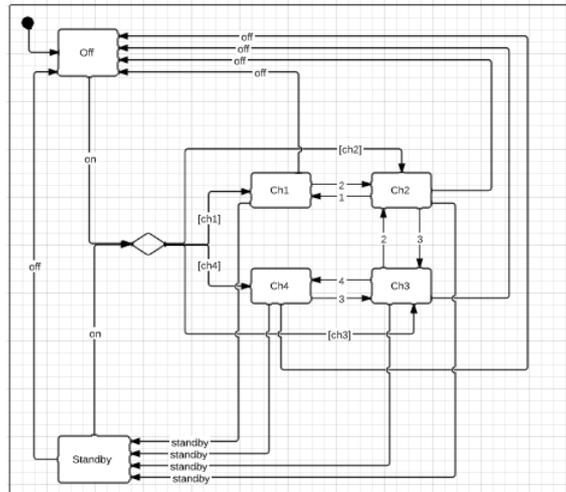


Figure 7.1.1. Model 1 of chapter 5 to apply bugs.

7.1.1 Damage state Ch1.

The first error that we apply to the model 1 shown in Sect. 7.1.1 “Simplest model change transitions “+” and “-” for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.” is to damage the state Ch1 and remove their transition “2” to go to the state Ch2. We maintain the transitions “off” and “standby” since Ch1 to states Off and Standby respectively. Also we have eliminated the transition “1” between states Ch2 and Ch1. We can see it in the figure below.

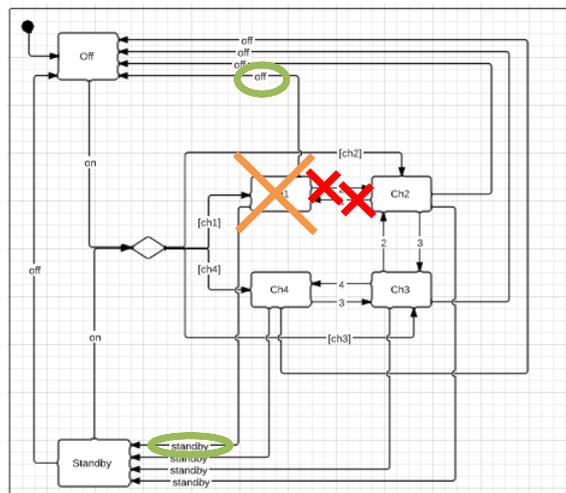


Figure 7.1.1.1 Bug 1.

To detect the error we only need to work with the 45 positive test cases obtained in section 7.1.1.1. We detect it concretely with the positive test case “on[ch2] 1”. Here we can see the set of positive test cases previously obtained. $T_{positive} = \{on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, on[ch2] 1, on[ch3] 2, on[ch3] 4, \dots, on[ch1] standby on[ch3] 4\}$. To detect it, we load on selenium the file html with the test cases and after we will check them in the statechart that we are interested with the browser Mozilla Firefox which we have integrated the extension selenium to run the test cases on every version of the statechart. The appearance of running selenium test cases is so:

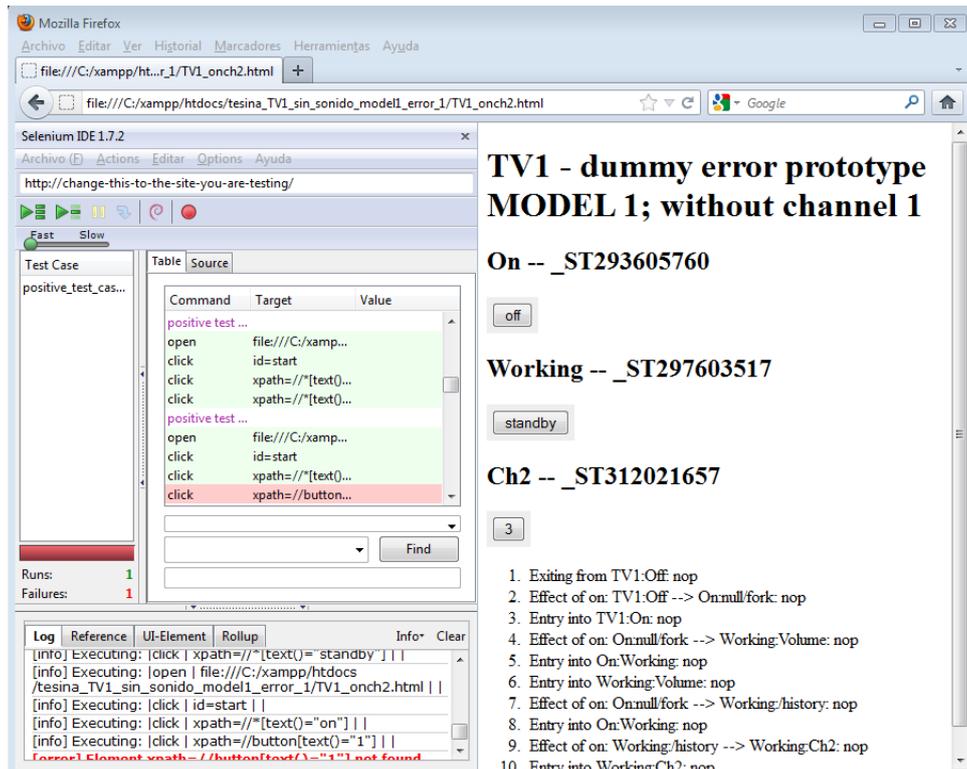


Figure 7.1.1.2 Detection of the error.

We can see the detection of the error in fig. 7.1.1.2, the moment in which selenium try to execute the transition “1” since the states Ch2 to Ch1. The code for this test case that has helped us to identify the error is as follows:

```

<!--positive test case-->
<tr>
  <td>open</td>
  <td>file:///C:/xampp/htdocs/tesina_TV1_sin_sonido_model1_error_1/TV1_onch2.html</td>
</tr>
<tr>
  <td>click</td>
  <td>id=start</td>
</tr>
<tr>
  <td>click</td>
  <td>xpath=//*[text()='on']</td>
</tr>
<tr>
  <td>click</td>
  <td>xpath=//button[text()='1']</td>
</tr>

```

Figure 7.1.1.3 Part of code with Dreamweaver.

7.1.2 Create new state Ch5.

The second error or modification that we apply to prove the usefulness of the obtained test cases is to create a new state Ch5 with their transition “4” to the state Ch4 and transition “5” since state Ch4 to the new state Ch5. We add to state Ch5 transitions “off” and “standby” to states Off and Standby respectively. The resulting statechart is shown in Fig. 7.1.2.1.

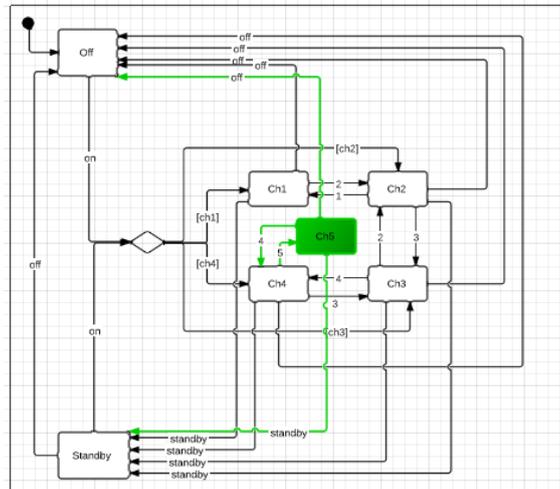


Figure 7.1.2.1 Bug 2.

In this case we can't detect it and analyze these new transitions with the generated test cases in section 7.1.1, we will need to add new test cases. The generation of these new test cases does not require a recalculation of the test cases generated as we have the advantage of working with a model where there is only one initial state *Off* and is not necessary to calculate all of them from the beginning, just add those that correspond to the state *Ch5*. after calculating the new test cases we can execute the following formula $T_{modified_model} - T_{initial_model}$, to reduce the test cases that we need to check, and do the subtraction with which we have obtained from the initial model. After that we only need to check the new test cases pertaining only to the state *Ch5*. The anterior test cases of the initial model still valid. As seen in Fig. 7.1.2.2 the execution of the test cases in Selenium obtained in Section 7.1.1 do not find any anomaly in this modified statechart.

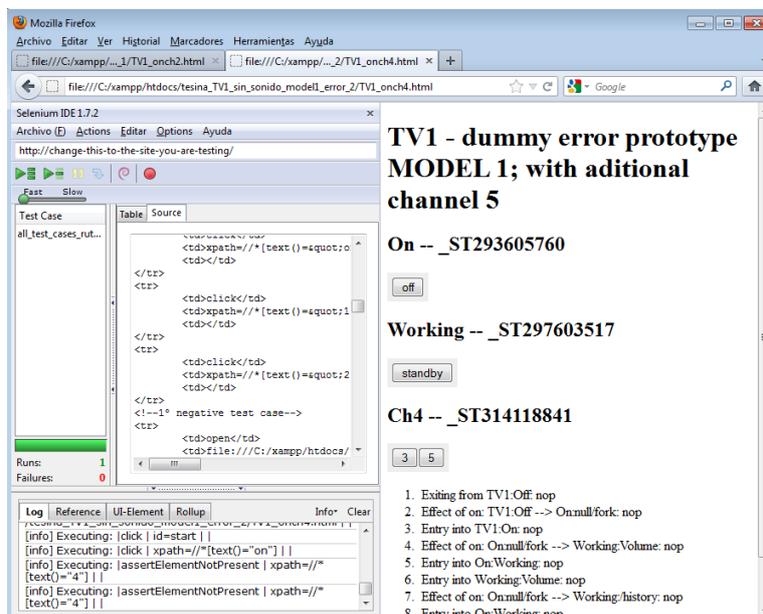


Figure 7.1.2.2 Validation with Selenium.

7.1.3 Create a new transition “on”.

The third modification that we apply to prove the usefulness of the obtained test cases is to create a new transition “on” since the state Ch3 to the state Off, maintaining the oldest transitions of the state Ch3. The resulting statechart is shown in Figure 7.1.3.1.

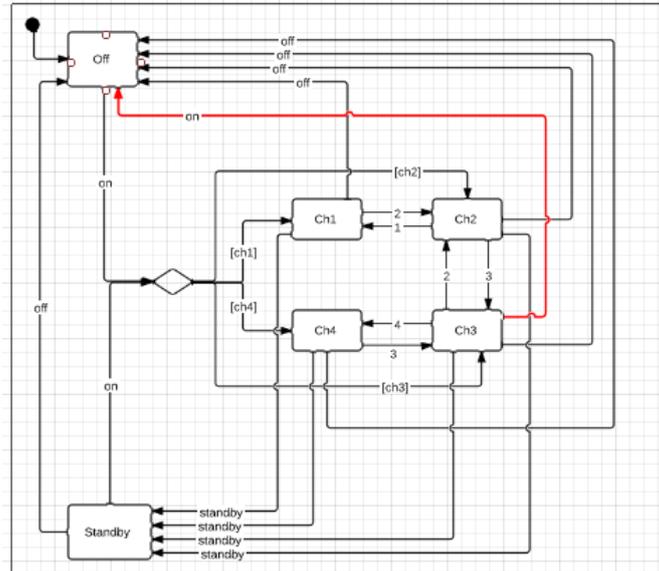


Figure 7.1.3.1 Bug 3.

To detect the error we only need to work with the negative test cases obtained in Sect. 7.1.1, there are 222. We detect it concretely with the negative test case “on[ch3] on[ch1] off”. Here we can see part of the negative test cases previously obtained. $T_{negative} = off\ off, 1\ off, 2\ off, 3\ off, 4\ off, standby\ off, off\ standby, 1\ standby, 2\ standby, 3\ standby, 4\ standby, standby\ standby, on[ch1]\ 1, on[ch3]\ 1, on[ch4]\ 1, off\ 1, 1\ 1, 2\ 1, 3\ 1, 4\ 1, standby\ 1, on[ch2]\ 2, on[ch4]\ 2, off\ 2, 1\ 2, 2\ 2, 3\ 2, 4\ 2, standby\ 2, on[ch1]\ 4, on[ch2]\ 4, on[ch4]\ 4, \dots, on[ch3]\ on[ch1]\ off, \dots$. To detect it, we load on selenium the file html with the test cases and after we will check them in the statechart that we are interested with the browser Mozilla Firefox which we have integrated the extension selenium to run the test cases on every version of the statechart. The appearance of running selenium test cases is so:

Command	Target	Value
negative test...		
open	file:///C:/xamp...	
click	id=start	
assertElement...	xpath=//*[@text()...	
assertElement...	xpath=//*[@text()...	
faltan casos ...		
open	file:///C:/xamp...	
click	id=start	
click	xpath=//*[@text()...	
assertElement...	xpath=//*[@text()...	
assertElementNotPresent	xpath=//*[@text()="on"]	Find

TV1 - dummy error 3 prototype MODEL 1; transition on between CH3 and Off

On -- _ST293605760

Working -- _ST297603517

Ch3 -- _ST313070249

Figure 7.1.3.2 Detection of the error.

We can see the detection of the error in fig. 7.1.3.2 in the moment in which selenium try to execute the transition "1" since the states Ch2 to Ch1. The code for this test case that has helped us to identify the error is as follows:

```

<tr><td>open</td>
<td>file:///C:/xampp/htdocs/tesina_TV1_sin_sonido_model1_error_3/TV1_onch3.html</td>
<td></td></tr>
<tr><td>click</td>
<td>id=start</td><td></td></tr>
<tr><td>click</td>
  <td>xpath=//*[@text()='on'];</td><td></td></tr>
<tr><td>assertElementNotPresent</td>
  <td>xpath=//*[@text()='on'];</td><td></td></tr>
<tr><td>click</td>
  <td>xpath=//*[@text()='off'];</td><td></td></tr>

```

Figure 7.1.3.3 Part of code with Dreamweaver.

7.1.4 Create a new transition "x".

This modification consist on add a new transition "x" that does not exist in our model to see what happens. We add to state Ch3 the transition "x" to state Off. Re-maintain old transitions that had the state Ch3. The resulting statechart is shown in figure 7.1.4.1.

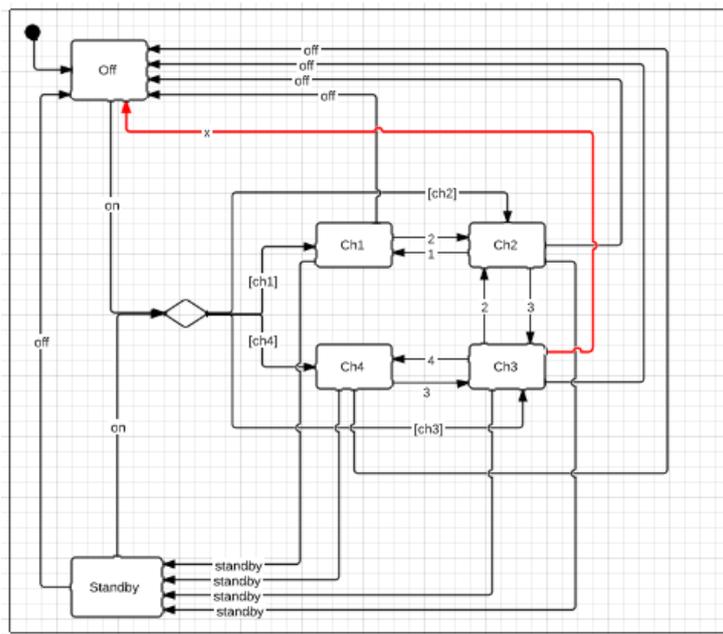


Figure 7.1.4.1 Bug 4.

If there is a new transition "x" which did not exist yet in the original statechart in which we had calculated the test cases, we need to recalculate the entire model applying again the formula of T. We have checked the positive, negative and redundant test cases in Selenium on this modification of the original statechart with the new transition "x" but it is impossible to find the bug of the discussed erroneous transition "x" that does not exist in the original model.

7.1.5 Create a new transition “on[ch1]”.

The fifth error that we apply to the model 1 shown in Sect. 7.1.1 “Simplest model change transitions “+” and “-” for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.” is to add a new transition “on[ch1]” from the states Ch1 to Standby. We maintain the transitions “off” and “standby” since Ch1 to states Off and Standby respectively. We can see it in the figure below.

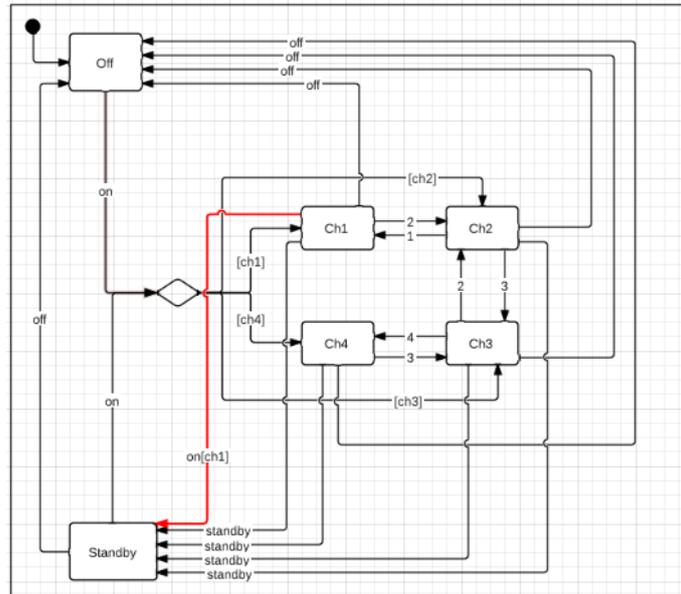


Figure 7.1.5.11 Bug 5.

To detect the error in this case any of the positive test cases obtained for the initial model is able to detect it because it is a negative test case. We need to work with the negative test cases obtained in Sect. 7.1.1, there are 222 test cases. We detect it concretely with the negative test case “on[ch1] on[ch1] off ”.

The screenshot shows Selenium IDE 1.7.2 running a test case. The test case table is as follows:

Command	Target	Value
negative test...		
open	file:///C:/xampp...	
click	id=start	
assertElement...	xpath=//*[@text()...	
assertElement...	xpath=//*[@text()...	
faltan casos ...		
open	file:///C:/xampp...	
click	id=start	
click	xpath=//*[@text()...	
assertElement...	xpath=//*[@text()...	
click	xpath=//*[@text()='1']	

The browser window displays the prototype for 'TV1 - dummy error 5' with the following text and buttons:

**TV1 - dummy error 5
prototype MODEL 1;
transition on between Ch1 and
Standby**

On -- _ST293605760

Working -- _ST297603517

Ch1 -- _ST310252158

Figure 7.1.5.2 Detection of the error with Selenium.

To detect it, we load on selenium the file html with the test cases and after we will check them in the statechart that we are interested with the browser Mozilla Firefox which we have integrated the extension selenium to run the test cases on every version of the statechart. We can see the appearance of running selenium test cases in figure 7.1.5.2. and the detection of the error in the moment in which selenium try to execute the transition “on” since the states Ch1. The code for this test case that has helped us to identify the error is as follows:

```

<tr><td>open</td>
<td>file:///C:/xampp/htdocs/tesina_TV1_sin_sonido_model1_error_5/TV1_onch1.html</td>
</tr>
<tr><td>click</td>
<td>id=start</td></td></tr><tr>
<td>click</td>
<td>xpath=//*[@text()='on']</td></tr>
</tr><tr><td>assertElementNotPresent</td>
<td>xpath=//*[@text()='on']</td></tr>
</tr><tr><td>click</td>
<td>xpath=//*[@text()='off']</td></tr>

```

Figure 7.1.5.32 Part of code with Dreamweaver.

7.1.6 Damage transition between states Ch2 and Ch1.

In this modification of the original statechart obtained in section 7.1.1. “Simplest model change transitions “+” and “-” for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.” we change the transition “1” from the states Ch2 to Ch1 by the transition “1”. We maintain the transitions “off” and “standby” since states Ch1 and Ch2 to states Off and Standby respectively. The old transition “2” between states Ch1 and Ch2 remains. We can see it in the figure below.

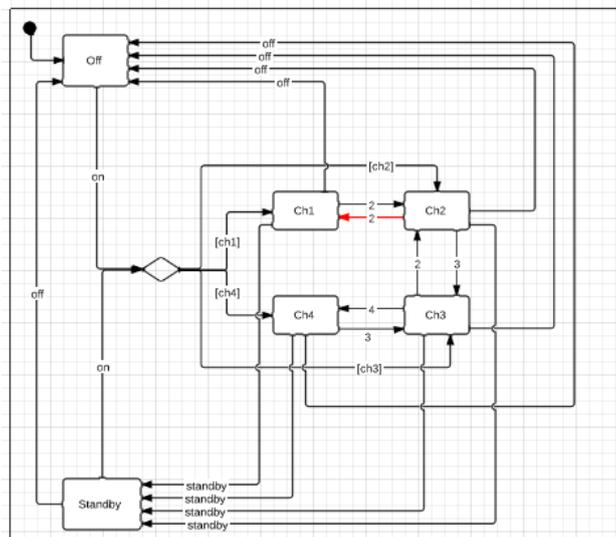


Figure 7.1.6.1 Bug 6.

To detect the error we only need to work with the 45 positive test cases obtained in Sect. 7.1.1. We detect it concretely with the positive test case “on[ch1] 2 1”. Here we can see the entire set of positive test cases previously obtained. $T_{positive} = on[ch1] off, on[ch2] off, on[ch3] off, on[ch4] off, on[ch1] standby, on[ch2] standby, on[ch3] standby, on[ch4] standby, on[ch2] 1, on[ch3] 2, on[ch3] 4, on[ch1] 2 off, on[ch1] standby off, on[ch1] 2 standby, on[ch1] 2 1, on[ch2] 1 off, on[ch2] 3 off, on[ch2] standby off, on[ch2] 1 standby, on[ch2] 3 standby, on[ch2] 1 2, on[ch2] 3 2, on[ch2] 3 2, on[ch2] 3 4, on[ch3] 2 off, on[ch3] 4 off, on[ch3] standby off, on[ch3]$

2 standby, on[ch3] 4 standby, on[ch3] 2 1, on[ch4] 3 off, on[ch4] 3 standby, on[ch4] 3 2, on[ch4] 3 4, on[ch1] standby on[ch1] off, on[ch1] standby on[ch2] off, on[ch1] standby on[ch3] off, on[ch1] standby on[ch4] off, on[ch1] standby on[ch1] standby, on[ch1] standby on[ch2] standby, on[ch1] standby on[ch3] standby, on[ch1] standby on[ch4] standby, on[ch1] standby on[ch2] 1, on[ch1] standby on[ch1] 2, on[ch1] standby on[ch3] 2, on[ch1] standby on[ch3] 4. To detect it, we load on selenium the file html with the test cases and after we will check them in the statechart that we are interested with the browser Mozilla Firefox which we have integrated the extension selenium to run the test cases on every version of the statechart. The appearance of running selenium test cases is so:

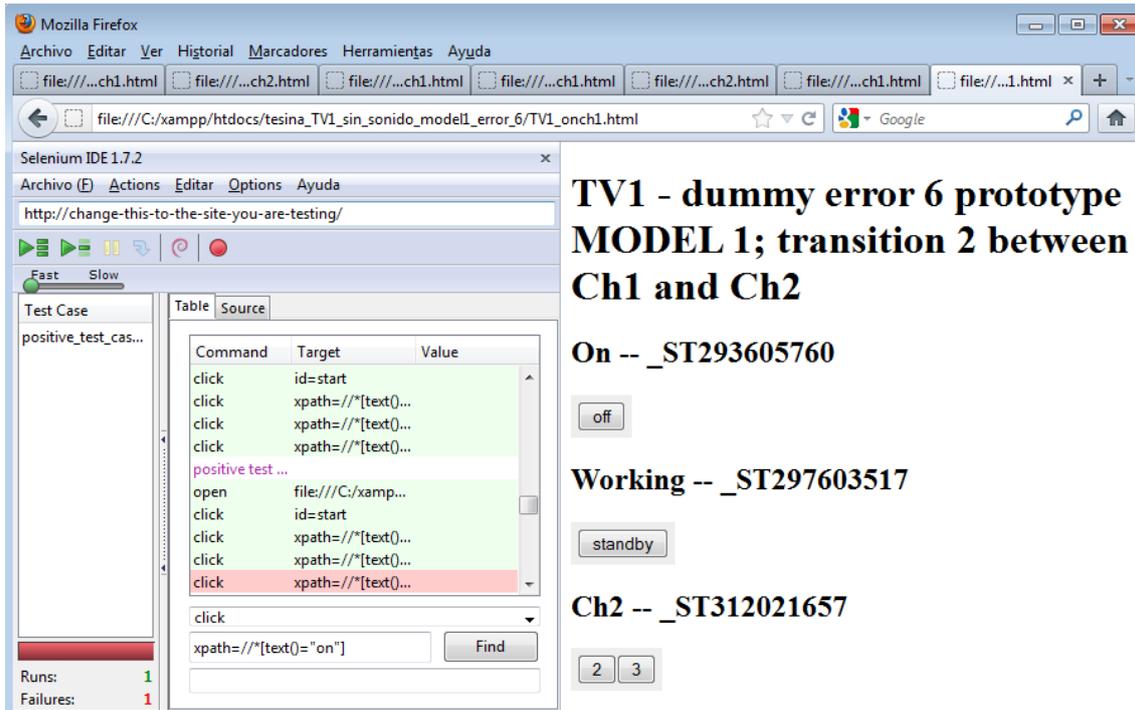


Figure 7.1.6.2 Detection of the error with Selenium.

We can see the detection of the error in fig. 7.1.6.2 in the moment in which selenium try to execute the transition “1” since the states Ch2 to Ch1. The code for this test case that has helped us to identify the error is as follows:

```

<tr><td>open</td>
<td>file:///C:/xampp/htdocs/tesina_TV1_sin_sonido_model1_error_6/TV1_onch1.html</td>
<td></td></tr><tr><td>click</td>
<td>id=start</td><td></td></tr><tr><td>click</td>
<td>xpath=//*[@text()='on';]</td><td></td></tr><tr><td>click</td>
<td>xpath=//*[@text()='2';]</td><td></td></tr><tr><td>click</td>
<td>xpath=//*[@text()='1';]</td><td></td></tr>

```

Figure 7.1.6.3 Part of code in Dreamweaver.

7.1.7 Change transition “off” by “standby”.

In the seventh modification of the original statechart obtained in section 7.1.1 “Simplest model change transitions “+” and “-” for numbers with a broken forward and a backward loop around ch1, ch2, ch3 and ch4.” we change the transition “off” from the states Ch1 to Off by the transition “standby”. We maintain all old transitions of the state Off, but we remove the transition “standby” since states Ch1 to Standby, the rest of old transitions of the state Ch1 remain. We can see it in the figure below.

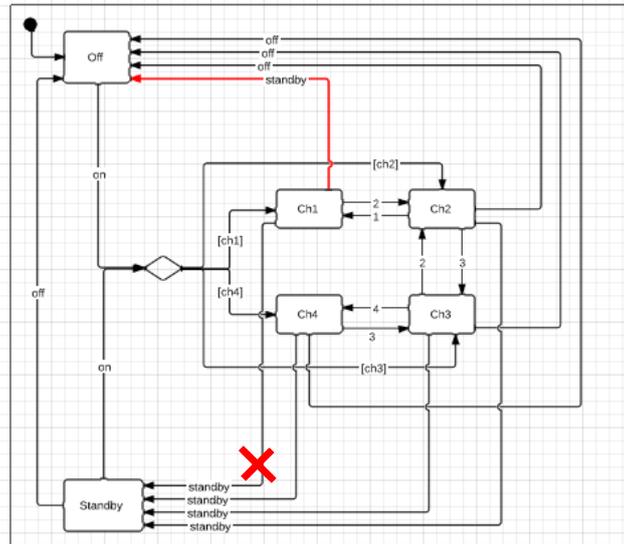
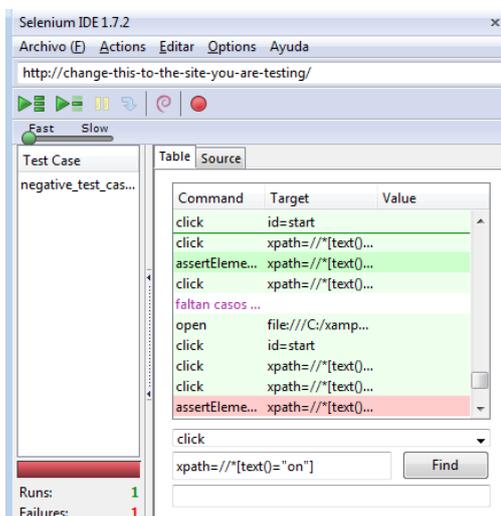


Figure 7.1.7.1 Bug 7.

To detect the error we only need to work with the negative test cases obtained in Sect. 7.1.1, there are 222. We detect it concretely with the negative test case “on[ch1] off off”. Here we can see part of the negative test cases previously obtained. $T_{negative} = \text{off off, 1 off, 2 off, 3 off, 4 off, standby off, off standby, 1 standby, 2 standby, 3 standby, 4 standby, standby standby, on[ch1] 1, on[ch3] 1, on[ch4] 1, off 1, 1 1, 2 1, 3 1, 4 1, standby 1, on[ch2] 2, on[ch4] 2, off 2, 1 2, 2 2, 3 2, 4 2, standby 2, on[ch1] 4, on[ch2] 4, on[ch4] 4, \dots, on[ch1] off off, \dots}$ To detect it, we load on selenium the file html with the test cases and after we will check them in the statechart that we are interested with the browser Mozilla Firefox which we have integrated the extension selenium to run the test cases on every version of the statechart. The appearance of running selenium test cases is so:



TV1 - dummy error 7 prototype MODEL 1; transition standby between Ch1 and Off

On -- _ST293605760



StandBy -- _ST295834018



1. Exiting from On:Working: nop
2. Effect of standby: On:Working --> On:StandBy: nop
3. Entry into On:StandBy: nop

Figure 7.1.7.23 Detection of the error with Selenium.

We can see the detection of the error in fig. 42 in the moment in which selenium try to execute the transition “1” since the states Ch2 to Ch1. The code for this test case that has helped us to identify the error is as follows:

```
<tr><td>open</td>
<td>file:///C:/xampp/htdocs/tesina_TV1_sin_sonido_model1_error_7/TV1_onch1.html</td>
<td></td></tr><tr><td>click</td><td>id=start</td><td></td>
</tr><tr><td>click</td>
  <td>xpath=//*[text()='on']</td><td></td></tr>
<tr><td>click</td>
  <td>xpath=//*[text()='off']</td><td></td></tr>
|<tr><td>assertElementNotPresent</td>
  <td>xpath=//*[text()='off']</td><td></td></tr>
```

Figure 7.1.7.3 Part of code with Dreamweaver.

Test cases generation for functional tests of user interfaces.

Chapter 8

Conclusions

The conclusions of the study are shown in this chapter. Are remembered proposed at the beginning objectives of the work and checks whether there is compliance for the different algorithms studied. We discuss possible alternatives to the testing of applications for those who are not yet convinced the idea of issue this type of techniques. Finally, current research and future work are exposed.

Test cases generation for functional tests of user interfaces.

8.1 Contributions

The objectives that have been raised in this study are:

- i. To perform a literature review with which to investigate current techniques, methods, and methodologies for software testing.
- ii. To analyze the behavior of existing techniques such as “W and Wp methods” [7] in our own case study looking for limitations and improvements showing which algorithm is more efficient.
- iii. To propose and develop an algorithm based on W and Wp method developed in [7], own ideas, and collected ideas on different ways of working with statechart transitions covering the deficiencies in the methods discussed, and follow the latest quality standards.
- iv. To propose a solution when the analyzed statechart contains the feared loops that make impossible the application of some testing algorithms.
- v. To show real examples where a bug is identified through the test cases obtained in our case study.

Regarding the first objective, with the starting experience, we perform a literature review on current techniques, methods, and methodologies for software testing. We have analyzed in chapter 4 the articles published by “Kirill Bogdanov” [4], [5] and [7] based on W & Wp methods and other papers about other methods which do not have relationship with these finding alternative ways for our proposed algorithm while we analyzed different testing techniques.

With the second objective we have shown in our case study that the Wp method is more efficient than the W and both are applicable to our statechart but not for all versions because one of the biggest problems that we found was falling into infinite loops when we try to apply these algorithms in our statechart (*remember that in Chapter 5 were created different versions of the original statechart because every time that the methods W & Wp discussed in this chapter came into an infinite loop was impossible to calculate the number of test cases categorizing them as unsupported methods*). With this we have shown some of the deficiencies of these methods. So this second point has been necessary to identify the advantages and disadvantages of each method according to the user needs and to discover existing gaps to address research efforts when developing a new algorithm, which was our next target. Simultaneously this analysis of existing methods was crucial to verify if the proposed algorithm is efficient and overcomes existing methods as discussed in the following paragraph.

About the third and fourth objectives our goal was to develop an algorithm that reduces significantly the length of the test sequences required for conformance testing while maintaining the same fault detection capability. We has shown that the algorithm we have created *U-Method* supports the original statechart avoiding problems with loops, so this is the only algorithm that can analyze our case study without modification. But when compared with the customization of statechart with which we have obtained the best results for Wp method this latter is somewhat more efficient as we saw in Chapter 6. So finally we have chosen U-Method as the most efficient algorithm as it works for all versions of statechart and obtain good results in the models created specifically for other algorithms. Some of the improvement is that the method works incrementally to reduce the length of generated test sequence so our new method (*U-Method*) always starts from the same starting state of the given FSM. This overcomes the problem that an extra leading sequenced may have to be added in the case that the test sequence generated started from a state different from the starting state of the given FSM. As we have commented in the previous paragraph one of the biggest problems that

we found was falling into infinite loops when we apply our formula by the appearance of them in our statechart, which we have solved by adding a finite number of iterations to the algorithm so that it does not end in an infinite loop.

As additional input about problems with infinite loops in section 8.3 of this final chapter we propose a new solution to the problem posing it as future work opening up the possibility of applying the “W and Wp Methods” developed in [7] when appear this type of problems that becomes unsupported methods considering the idea of divide the original statechart in two parts.

Finally with the final objective about bugs, we have altered the original statechart creating 7 different types of errors. With the test cases of chapter 5 obtained for the W Method we identified which one found the error verifying their usefulness.

8.2 Alternatives to testing

Software testing is an art. Most of the testing methods and practices are not very different from 20 years ago. It is nowhere near maturity, although there are many tools and techniques available to use. Good testing also requires a tester's creativity, experience and intuition, together with proper techniques. Testing is more than just debugging. Testing is not only used to locate defects and correct them. It is also used in validation, verification process, and reliability measurement.

In recent years enterprises worldwide have focused their efforts on providing software products with high quality according to standards and thus satisfy the end user, which ultimately is the most wins, of course, aside the money factor and the location of the developer in the competitive market. Testing is expensive. Automation is a good way to cut down cost and time. Complete testing is infeasible. Complexity is the root of the problem. At some point, software testing has to be stopped and product has to be shipped. The stopping time can be decided by the trade-off of time and budget. Or if the reliability estimate of the software product meets requirement.

Testing may not be the most effective method to improve software quality. Alternative methods, such as inspection, and clean-room engineering, may be even better.

Software testing is more and more considered a problematic method toward better quality. Using testing to locate and correct software defects can be an endless process. Bugs cannot be completely ruled out. Just as the complexity barrier indicates: chances are testing and fixing problems may not necessarily improve the quality and reliability of the software. In a narrower view, many testing techniques may have flaws. As early as in the publication [8] of “Myers and Glenford J.”, the so-called “human testing” including inspections, walkthroughs, reviews are suggested as possible alternatives to traditional testing methods. “Dick Hamlet” in [9] advocates inspection as a cost-effect alternative to unit testing. The experimental results in [10] “Victor R. Basili” suggests that code reading by stepwise abstraction is at least as effective as on-line functional and structural testing in terms of number and cost of faults observed.

8.3 Conclusions and future work

Software testing is an integrated part in software development. It is directly related to software quality. It has many subtle relations to the topics that software, software quality, software reliability and system reliability are involved. Software testing as part of plans for quality assurance, development companies offers the ability to detect and remove defects that arise during product development. Standards bodies offer different ways to implement testing processes, all based on maturity cycles that allow the measurement and optimization of them.

One of the biggest problems that we found with some statecharts is that we have problems to apply a testing technique to assure coverage when a loop appear in their specification. When we do testing in an application, we should put a probe on each link of the same specification. At a simplest, a probe consists of a counter which is incremented every time it is passed. With the part of the statechart without loops we haven't any problem to apply the probes, but if we apply it to the loops, we obtain an infinite number of probes and the testing algorithm would never end. There are some rules that can be applied to the loops; put a counter just after the loop-determining decision and put a counter just before the loop-back point.

At this point we have considered the idea of divide the statechart in two parts, on the one hand the part of the specification in which we can apply some of the automated existing testing technique witch which we have worked in this thesis that would result in the statechart we can see in the left part of fig. 8.3.1 and in the other hand we can consider the part that contain some loops and find a different technique to calculate the number of test cases that can contain the loops, resulting in a statechart as we see in the right part of fig. 8.3.1.

For example in the TV1 statechart of the fig. 5.3.1 we can divide it in these two parts as we defined in the previous paragraph. For the below figure we can compute x iterations to calculate the number of test cases, but for the other figure that corresponds to the part of the loop, the idea is that we can only execute a finite number of iterations to prevent the loop.

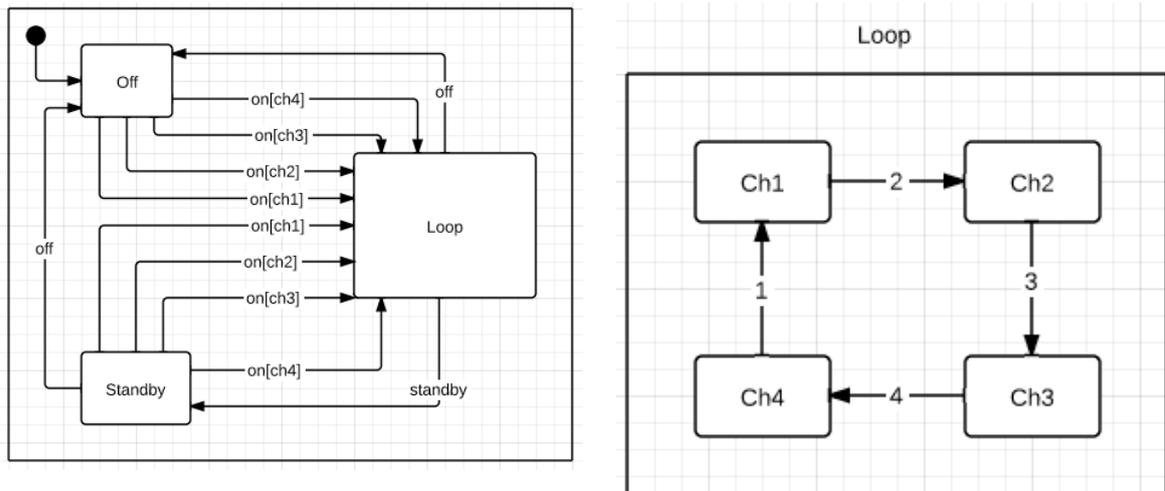


Figure 8.3.1. Divide the statechart in two parts to solve the problems with loops

After calculating the number of test cases for both models, we just have to make the union of both and get the number of final tests cases. This is just one possible way we could follow when analyzing loops.

Test cases generation for functional tests of user interfaces.

Bibliography

- [1] D. Harel, H. Lachover, A. Nammad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
- [2] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [3] K. Bogdanov. Automated testing of Harel’s statecharts. PhD thesis, The University of Sheffield, January 2000.
- [4] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software testing, verification and reliability*, 11:39–54, 2001.
- [5] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 1999.
- [6] T. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.
- [7] K. Bogdanov and M. Holcombe. Testing from statecharts using the Wp method, Department of Computer Science, The University of Sheffield Regent Court, 211 Portobello St., Sheffield S1 4DP, UK, 2002.
- [8] Myers, Glenford J., *The art of software testing*, Publication info: New York : Wiley, c1979. ISBN: 0471043281 Physical description: xi, 177 p. : ill. ; 24 cm.
- [9] Dick Hamlet; *Foundations of software testing: dependability theory*; Proceedings of the second ACM SIGSOFT symposium on Foundations of software engineering , 1994.
- [10] Victor R. Basili, Richard W. Selby, Jr. "Comparing the Effectiveness of Software Testing Strategies".
- [11] IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE Computer Soc., Dec. 10, 1990.
- [12] *Clever Algorithms: Nature-Inspired Programming Recipes* By Jason Brownlee PhD. First Edition, Lulu Enterprises, January 2011. ISBN: 978-1-4467-8506-5.
- [13] OMG, "Mda guide version 1.0.1," Object Management Group, Tech. Rep., 2003.
- [14] *Performance Testing Guidance for Web Applications* Microsoft patterns & practices -- by: J.D. Meier, Scott Barber, Carlos Farre, Prashant Bansode, and Dennis Rea
- [15] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: PrenticeHall, Inc., 1981.
- [16] J. Xiao, C. P. Lam, H. Li, and J. Wang, "Reformulation of the Generation of Conformance Testing Sequences to the Asymmetric Travelling Salesman Problem", *Proceedings of International Genetic and Evolutionary Computation Conference 2006*, Seattle, Washington, USA, July 8-12, 2006, ACM Press, pp.1933-1940.
- [17] W. H. Chen, "An Optimization Technique for Protocol Conformance Testing Based on the Wp Method", *International Journal of Applied Science and Engineering*, 1(1): 2003, pp. 45-54.
- [18] B. Korel, "Automated software test data generation" *Software Engineering, IEEE Transactions on*, vol. 16, pp. 870-879, 1990.
- [19] C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution" *Software Engineering, IEEE Transactions on*, vol. 27, pp. 1085-1110, 2001.
- [20] R. S. Pressman, *Software Engineering : a practitioner's approach*: McGraw-Hill, Inc., 2000.
- [21] Boris Beizer: *Software Testing Techniques*, Van Nostrand Reinhold, 1990.
- [22] Boris Beizer: *Black Box Testing*, Wiley, 1995.

- [23] Andreas F. Borchert: A Perl Crash Course, Script at the University of Ulm, 1996.
- [24] Marcel Alper: Professionelle Softwaretests (Professional Softwaretests), Vieweg, 1994.
- [25] Borland: dBASE 5.0 für DOS Handbuch (dBASE 5.0 for DOS Manual), Borland International, 1994.
- [26] Columbia University: The Concise Columbia Encyclopedia, Columbia University Press, 1995.
- [27] Ian Molyneaux. The Art of Application Performance Testing: Help for Programmers and Quality Assurance, 2009.
- [28] M. Grabert: DPV -- Ein Informationssystem zur Unterstützung von Qualitätsmonitoring, prospektiver Dokumentation und Routinearbeit bei der Behandlung von Typ 1 Diabetikern (A Diabetes Information System to support Quality Monitoring, Prospective Documentation, and Routine Work during the Medication of Type 1 Diabetics), University of Ulm, 1995.
- [29] Klaus Grimm: Systematisches Testen von Software -- Eine neue Methode und eine effektive Teststrategie (Systematic Testing of Software -- A new Method and an Effective Testing Strategy), Oldenbourg Verlag, 1995.
- [30] R. W. Holl: Benutzeranleitung Diabetessoftware zur Prospektiven Verlaufsdokumentation -- Version 3.0 DOS (Manual -- Diabetes Information System for a Prospective Documentation -- Version 3.0 DOS), University of Ulm, 1996.
- [31] Peter Hürter: Diabetes bei Kindern und Jugendlichen (Children and Juveniles having Diabetes), Springer Verlag, 1985.
- [32] Siamak Haschemi, Humboldt Universität zu Berlin, Berlin, Germany. "Model transformations to satisfy all-configurations-transitions on statecharts". MoDeVVA '09 Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation.
- [33] Jun Wang, Edith Cowan University. Jitian Xiao, Edith Cowan University. Chiou Peng Lam, Edith Cowan University. Huaizhong Li, Edith Cowan University. "A Bipartite Graph Approach to Generate Optimal Test Sequences for Protocol Conformance Testing using the Wp-method". Cowan University, 2005.
- [34] "A Match-based Approach to Optimize Conformance Test Sequence Generation using Mp-method", 2009.
- [35] "Covering Transitions of Concurrent Systems through Queues", 2005.
- [36] "Test Data Generation from UML State Machine Diagrams using Gas", 2007.
- [37] BCS Oxfordshire. Workshop of State transition testing presented by: Peter Quentin.
- [38] Test planning "<http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/Web/Testing/Planning.html>".
- [39] Software Testing Life Cycle (STLC) "<http://www.guru99.com/software-testing-life-cycle.html>"
- [40] Automated Test Concepts "http://www.lw-tech.com/q1/ug_concepts.htm"
- [41] Official page of WAPT "<http://www.loadtestingtool.com/>"
- [42] Statechart diagram "http://en.wikipedia.org/wiki/State_diagram"
- [43] UML Statechart "http://www.tutorialspoint.com/uml/uml_statechart_diagram.htm"
- [44] Finite-state machine "<http://www.objectmentor.com/resources/articles/umlfsm.pdf>"
- [45] "UML Tutorial: Finite State Machines." Robert C. Martin. Engineering Notebook Column. June 98.
- [46] "ESTADO DEL ARTE DE MÉTODOS, TIPOS DE TESTING Y HERRAMIENTAS PARA APLICAR PRUEBAS DERENDIMIENTO." Juan Oliver Navarro. FUNDACIÓN UNIVERSITARIA TECNOLÓGICO. 2010
- [47] Scott W. Ambler. The Full Life Cycle Object-Oriented Testing (FLOOT) Method. 2004-2010.
- [48] Free Tool JMeter "<http://jmeter.apache.org/>".
- [49] Free Tool Jcrawler "<http://jcrawler.sourceforge.net/>".

- [50] Free Tool Netsparker "<http://www.mavitunasecurity.com/netsparker/>"
- [51] Free Tool OpenSTA "<http://opensta.org/>".
- [52] Free Tool TestMaker "<http://www.easytestmaker.com/default.aspx>".
- [53] CAST "<http://www.elen.ktu.lt/~rsei/PT/Computer-Aided%20Software%20Test%20Tools%20for%20Unit%20Level%20Test%20-%20December%201994.htm>"

Annex

Test cases generation for functional tests of user interfaces.

A. Emails with Kirill Bogdanov

i. Day 22-11-2011

“Hello K.Bogdanov,

I’m a Spanish student in Italy, and I'm doing a master of computer science in software engineering. I’m reading your paper "Testing from statecharts using the Wp method" and would greatly appreciate you give me more information about the two formulas of the Wp method (the phase one and phase two). I have particular problems to solve the equation of the second phase.

Thank you very much.

Francisco Gramuntell Desco. Technical Engineer in Computer Science.”

ii. Day 23-11-2011

“Francisco,

Are you familiar with Wp method for FSM?

The paper you refer to describes how to incrementally build components from which Wp test set is subsequently constructed. Section 3 shows how to test statecharts using the full W method. Section 4 says that it is possible to adapt Wp method for testing statecharts in a similar way.

The elements that are different between the two methods are (1) merging rules for w sets and (2) the use of CE notation. Construction of elements of w sets is similar to that for the full W method (recursively bottom-up). CE is a function which computes a configuration entered by a statechart when a sequence of operations is attempted. Can you tell me please which of these is unclear? Do you understand what is happening in section 3? (you need to understand how the W method is adapted for statecharts before understanding how it works for the Wp one).

Dr. Kirill Bogdanov : K.Bogdanov@dcs.shef.ac.uk

<http://www.dcs.shef.ac.uk/~kirill>”

iii. Day 25-11-2011

“Thanks for answering Mr.Bogdanov

Yes, I’m very interested in W method for FSM. I understand very well the W method and all the section 3. I have problems in section 4, I understand the formula of the first phase of Wp but I don’t know what are the transitions that remain out in the first phase and after are tested in the second phase using small sets.

The other problem is with the use of CE notation because in the second phase of Wp in the formula T (sub two) I understand how to calculate TS but I don’t understand CE; you say that “CE is the configuration entered after taking a path path from a configuration conf” and you give an example with “conf (sub init)={STOP, IDLE}” but I don’t understand it.

Thank you very much.

Francisco Gramuntell Desco. Technical Engineer in Computer Science.”

iv. Day 26-11-11

"I understand the formula of the first phase of Wp but I don't know what are the transitions that remain out in the first phase and after are tested in the second phase using small sets.

I describe this below, as a part of my explanation of the Wp method. For FSM: The purpose of the first phase is to ensure that small w sets are capable of identifying states in an implementation. For this purpose, every state of an implementation is visited and W set is applied in that state. Here is why this phase is needed: Imagine two states in a specification, A with only the transition "a" and B with only "b". Small sets for these states could be $w_A=\{a\}$ and $w_B=\{b\}$. If an implementation has an erroneous transition "b" from A, this state look as both A and B in this implementation. Hence if there is an erroneous transition leading to state A rather than state B somewhere in this implementation, the defect will not be found if only small sets are used, because we'll check the target state with w_B and the erroneous transition "b" from the A state will make us think that this is B rather than A.

This example shows two things,

1. If I attempt all small sets from every state in an implementation (the first phase of the Wp method), this will verify if any pair of states may be confused. It will also test all transitions used to traverse an automaton when doing this "check for confusion", $\text{Explored} = C^*(1 \cup \Phi \cup \dots \cup \Phi^{m-n})$. Transitions from the rest of the full W set, namely anything in $C^*(1 \cup \Phi \cup \dots \cup \Phi^{m-n+1}) - \text{Explored}$ will not be tested by the first phase (note that +1 in the above set). These transitions are tested using the small sets in the second phase. For statecharts, Φ is a set of possible steps of a statechart. This includes all possible concurrent transitions and is the same one as described in section 3 of the paper.

2. The problem can be avoided if small sets are selected such that where I use path "b" to distinguish B from A, I should use the same path to distinguish A from B. In the above example, "b" is used to distinguish B from all other states, including A and thus should be used in a small set w for state A, making $w_A=\{a,b\}$. Such an arrangement makes it impossible to confuse states and thus it is not necessary to use the first phase of the Wp set. The testing method using this idea is called HSI or HIS and I think it can be adapted for statecharts in a similar way to the way the Wp method was adapted.

The other problem is with the use of CE notation because in the second phase of Wp in the formula T (sub two) I understand how to calculate TS but I don't understand CE; you say that ?CE is the configuration entered after taking a path path from a configuration conf .? And you give an example with ?conf (sub init)={STOP, IDLE}? but I don't understand it. For FSM, one can talk of a transition function which given a state and an input returns a target state. In a similar way for a statechart, there is a configuration and a set of transitions (step), leading to a target configuration. Given a series of inputs for FSM, one may determine a target state by applying the transition function a few times. For a statechart, the same applies where one chooses a series of steps and a starting configuration, the outcome is a final configuration. CE is a function computing this.

conf (sub init) is the initial configuration, that is {STOP, IDLE} (and in reality all parent states of these two for a full configuration, but a set of basic states can be used to uniquely identify a configuration).

Where I choose a step containing a single "play" transition, the next configuration is {PLAY, IDLE}, hence this will be the outcome returned by $\text{CE}(\text{play},\{\text{STOP}, \text{IDLE}\})=\{\text{PLAY}, \text{IDLE}\}$. If I take another valid step, rew_or_ff, then $\text{CE}(\text{rew_or_ff},\{\text{PLAY}, \text{IDLE}\})=\{\text{PLAY}, \text{REW_FF}\}$. It is also true

that $CE(\text{play rew_or_ff}, \{\text{STOP}, \text{IDLE}\}) = \{\text{PLAY}, \text{REW_FF}\}$ (the two steps taken consecutively from the initial configuration) and $CE(\{\text{play}, \text{rew_or_ff}\}, \{\text{STOP}, \text{IDLE}\}) = \{\text{PLAY}, \text{REW_FF}\}$ (here the two transitions are taken concurrently in the same step).

Let me know if this clarifies it.

Dr. Kirill Bogdanov : K.Bogdanov@dcs.shef.ac.uk
<http://www.dcs.shef.ac.uk/~kirill>

v. Day 29-11-2011

“Francisco,

Only one final doubt, this formula that you have written in your response it's good? $C^(1 \cup \Phi \cup \dots \cup \Phi^{m-n+1}) - \text{Explored}$ or you need to add $(* W)$ in the first part? $C^*(1 \cup \Phi \cup \dots \cup \Phi^{m-n+1}) * W - \text{Explored}$*

No, because I'm talking of transitions which are tested in the second part. W is a part of a test set, see below. Explored is the set of $_transitions_$ which happen to be visited and tested in the first phase of the W_p method. The test set for the first phase will certainly include W at the end to verify target states of those transitions. In a similar way, $C^*(1 \cup \Phi \cup \dots \cup \Phi^{m-n+1}) - \text{Explored}$ refers to $_transitions_$ which are considered in the second part of the W method. Target states entered by these transitions are verified using small w sets rather than the full W set.

I have been a great help. If you are interested I could send the summary I'm doing about your article.

Yes, I'm interested.

Dr. Kirill Bogdanov : K.Bogdanov@dcs.shef.ac.uk
<http://www.dcs.shef.ac.uk/~kirill>

vi. Day 28-01-12

“Hello K. Bogdanov,

I'm still working on my thesis about research on testing methods. Your articles on W and W_p method are very important to my research, in addition to integrate transition coverage and other features in them trying to find new solutions. As I said a few months ago, I attach in this mail a small part of my work. I applied on the statechart that we are working the W method to calculate the test cases. I'm currently working to implement the W_p method to it. I would like to ask you about the testing tools that you used to apply the W method and / or W_p method. If you can give me their names or some information about the tools that would help me.

Thank you very much. Regards.

Francisco Gramuntell Desco. Technical Engineer in Computer Science.”

➤ **Email content, without applying the changes commented by Bogdanov**

Apply W-method to TV without concurrent region (without volume)

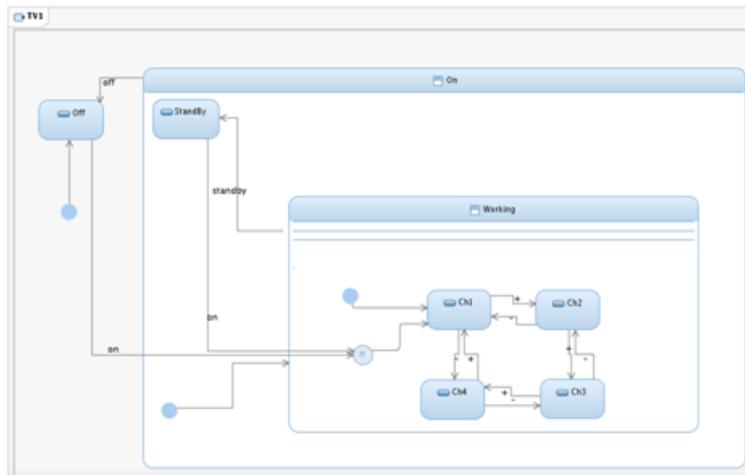


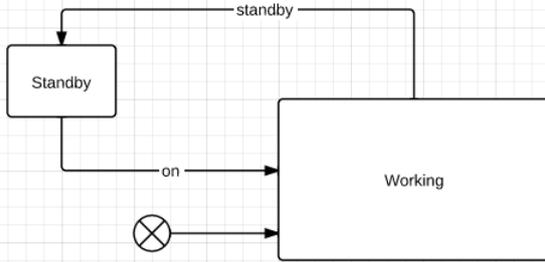
Figure Annexe.1. Original statechart without concurrency

TV1 (test case generation for states without hierarchy)

<p>Op. A</p>	<p>TCB for TV1</p> <p>$\Phi = \{on, off\}$ $C = \{1, on\}$ $W = \{on, off\}$</p>
<p>Op. B (WRONG)</p>	<p>$\Phi = \{off\}$ $C = \{1\}$ $W = \{off\}$</p>
<p>Op. C (WRONG)</p>	<p>$\Phi = \{on, off\}$ $C = \{1, on\}$ $W = \{on, off\}$</p>
<p>$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$ $T = \{1, on\} * \{on, off\} * \{on, off\} = 8 \text{ test cases}$</p>	

$$C_{TV1} != C_{TV1}^M$$

On (test case generation for states without hierarchy)



TCB for On

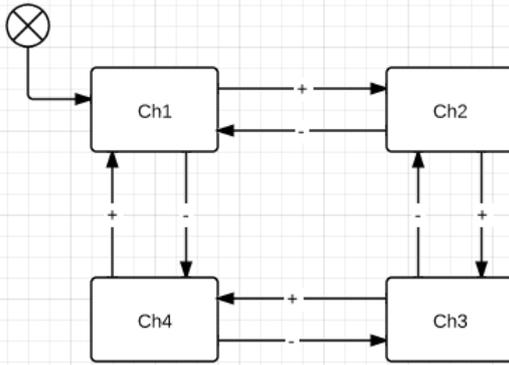
$$\begin{aligned} \Phi &= \{standby, on\} \\ C &= \{1, standby\} \\ W &= \{on\} \end{aligned}$$

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

$$T = \{1, standby\} * \{standby, on\} * \{on\} = 4$$

$$C_{On} != C_{On}^M$$

Working (test case generation for state hierarchy)



TCB for Working

$$\begin{aligned} \Phi &= \{+, -\} \\ C &= \{1, standby\} \\ W &= \{on\} \end{aligned}$$

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

$$T = \{1, standby\} * \{+, -\} * \{on\} = 4$$

$$C_{Working} = C_{Working}^M$$

Step 1

With the value of $C_{Working}^M$ we can start to develop the formulas for all the statechart.

First we are going to obtain $(C, \Phi, W)_{On}^M$

$$C_{On}^M = C_{On} \cup \{path_in_C_{On_to_enter_Working}\} * C_{Working}^M$$

$$W_{On}^M = W_{On} \cup W_{Working}^M$$

$$\Phi_{On}^M = \Phi_{On} \cup \Phi_{Working}^M$$

$$C_{On}^M = \{1, standby\} \cup \{1\} * \{1, standby\} = \{1, standby, standby, standby\}$$

$$W_{On}^M = \{on\} \cup \{on\} = \{on\}$$

$$\Phi_{On}^M = \{standby, on\} \cup \{+, -\} = \{standby, on, +, -\}$$

Step 2

Now we can calculate $(C, \Phi, W)_{TV1}^M$

Op.c. A

$$C_{TV1}^M = C_{TV1} \cup \{path_in_C_{TV1_to_enter_On}\} * C_{On}^M$$

$$W_{TV1}^M = W_{TV1} \cup W_{On}^M$$

$$\Phi_{TV1}^M = \Phi_{TV1} \cup \Phi_{On}^M$$

$$C_{TV1}^M = \{1, on\} \cup \{on\} * \{1, standby, standby, standby standby\} = \{1, standby, standby, standby standby, on, on standby, on standby, on standby standby\}$$

$$W_{TV1}^M = \{on, off\} \cup \{on\} = \{on, off\}$$

$$\Phi_{TV1}^M = \{on, off\} \cup \{standby, on, +, -\} = \{standby, on, off, +, -\}$$

Op.c. B

(WRONG)

$$C_{TV1}^M = C_{TV1} \cup \{path_in_C_{TV1_to_enter_On}\} * C_{On}^M \cup \{path_in_C_{TV1_to_enter_Working}\} * C_{Working}^M$$

$$W_{TV1}^M = W_{TV1} \cup W_{On}^M \cup W_{Working}^M$$

$$\Phi_{TV1}^M = \Phi_{TV1} \cup \Phi_{On}^M \cup \Phi_{Working}^M$$

Step 3

And finally we can apply the formula to calculate the test cases of ALL the statechart

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

$$T = C_{TV1}^M * \Phi_{TV1}^M * W_{TV1}^M$$

$$T = \{1, standby, standby, standby standby, on, on standby, on standby, on standby standby\} * \{standby, on, off, +, -\} * \{on, off\}$$

$$T = 8 * 5 * 2 = 80 \text{ test cases.}$$

vii. Day 2-02-2012

“Francisco,

Apologies for the delay, I start teaching next week and there is a lot of preparation what had to be done during January.

You used $W=\{on,off\}$ in the TCB for the TV1 part, you only need one of those, because absence of a transition is enough to distinguish On from Off. There is also a problem with the history connector, because you do not know in advance which state will be entered. If you can assume that (1) you will always enter Ch1 during testing of the TV, and (2) you follow this by a separate testing of the history connector, this will work. Without this assumption, you need to somehow ensure predictable behaviour of the history connector during testing. This will be the assumption which will influence test generation. I'm not sure I understand why you define W for an erroneous operation Op C, W is defined based on a correct model and is used to generate tests to identify all faulty implementations among a given class of faults.

Computation of a set of test cases is $T=C*W \cup C*\Phi*W$, you may be missing the first part. In order to obtain a set set, you need to follow all sequences you obtained in T, in your model up to and including the first element which is supposed to be missing in a correct implementation. In the process of following those sequences, you need to identify test input and corresponding test outputs.

For Working, standby should not be a part of the state cover, but sequences of '+' and '-' to enter all states should be. This is assuming that history connector is switched off. Hence you could use $C=\{1,+,+ +,+ + +\}$. The way you combine sets seems wrong, In order to obtain a combined state cover for On, $C=\{1, standby\} \cup \{1\}*\{1,+,+ +,+ + +\} = \{1, +,+ +,+ + +,standby\}$ Step 2 uses $C=\{1, on\} \cup \{on\} * \{1, +,+ +,+ + +,standby\} = \{1, on, on +,on + +,on + + +,on standby\}$.

About Tools: The implementation of the test method for statecharts that I developed many years ago was proprietary and I was told by DaimlerChrysler not to distribute it. I might be able to negotiate with them, but I think merging rules are sufficiently simple that you'll find it easy to implement. The current tool (Statechum) has implementation of TCB generation for FSM as well as test set generation, but there is no provision for hierarchy or concurrency in it. I think you will be able to use it for W set generation, but it is not clear how to integrate your work into it.

➤ **Content of the mail with the modifications proposed by Bogdanov**

Apply W-method to TV without concurrent region (without volume) original statechart

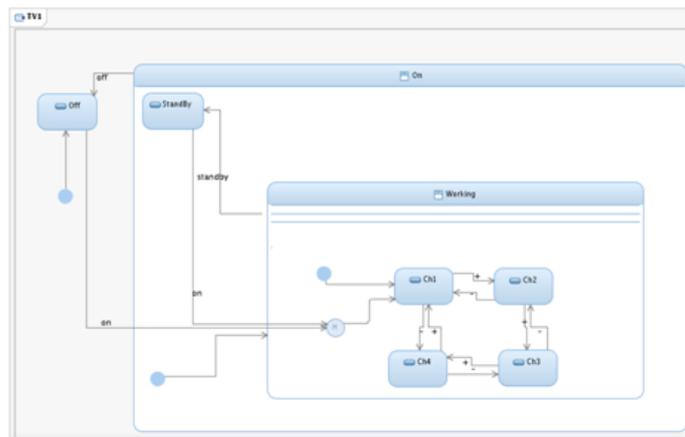
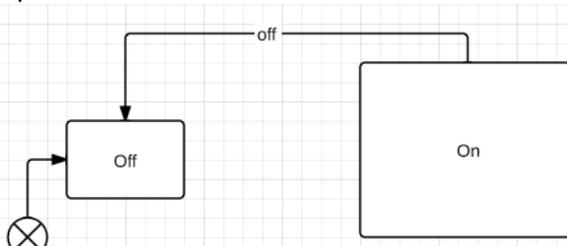
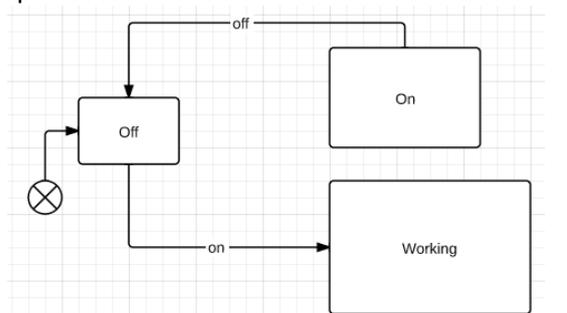


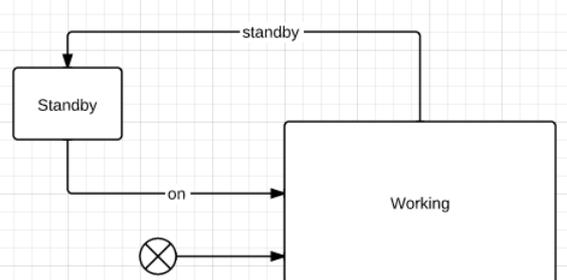
Figure Annexe.2. TV1 (test case generation for states without hierarchy)

I have worked with the option A but it may not be adequate and may be the B or C.

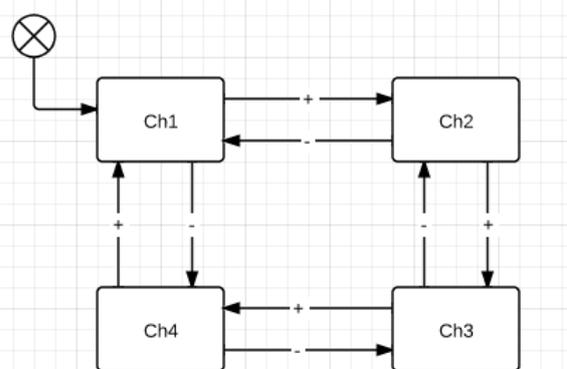
<p>Op. A</p>	<p><u>TCB for TV1</u></p> <p>$\Phi = \{on, off\}$ $C = \{1, on\}$ $W = \{on\}$</p>
--------------	---

<p>Op. B ????</p> 	<p>$\Phi = \{off\}$ $C = \{1\}$ $W = \{off\}$</p>
<p>Op. C ????</p> 	<p>$\Phi = \{on, off\}$ $C = \{1, on\}$ $W = \{on, off\}$</p>

$C_{TV1} \neq C_{TV1}^M$ therefore we continue decomposing the statechart

<p>On (test case generation for states without hierarchy)</p> 	<p><u>TCB for On</u></p> <p>$\Phi = \{standby, on\}$ $C = \{1, standby\}$ $W = \{on\}$</p>
---	---

$C_{On} \neq C_{On}^M$ therefore we continue decomposing the statechart

<p>Working (test case generation for state hierarchy)</p> 	<p><u>TCB for Working</u></p> <p>$\Phi = \{+, -\}$ $C = \{1, +, +, +, +\}$ $W = \{on\}$</p>
---	--

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

$$T = \{1, +, +, +, +\} * \{+, -\} * \{on\} = 8 \text{ test cases}$$

$C_{Working} = C_{Working}^M$ so we can start calculating the number of test cases

Step 1

With the value of $C_{Working}^M$ we can start to develop the formulas for all the statechart.

First we are going to obtain $(C, \Phi, W)_{On}^M$

$$C_{On}^M = C_{On} \cup \{path_in_C_{On_to_enter_Working}\} * C_{Working}^M$$

$$W_{On}^M = W_{On} \cup W_{Working}^M$$

$$\Phi_{On}^M = \Phi_{On} \cup \Phi_{Working}^M$$

$$C_{On}^M = \{1, standby\} \cup \{1\} * \{1, +, +, +, +\} = \{1, +, +, +, +, standby\}$$

$$W_{On}^M = \{on\} \cup \{on\} = \{on\}$$

$$\Phi_{On}^M = \{standby, on\} \cup \{+, -\} = \{standby, on, +, -\}$$

Step 2

Now we can calculate $(C, \Phi, W)_{TV1}^M$

$$C_{TV1}^M = C_{TV1} \cup \{path_in_C_{TV1_to_enter_On}\} * C_{On}^M$$

$$W_{TV1}^M = W_{TV1} \cup W_{On}^M$$

$$\Phi_{TV1}^M = \Phi_{TV1} \cup \Phi_{On}^M$$

$$C_{TV1}^M = \{1, on\} \cup \{on\} * \{1, +, +, +, +, standby\} = \{1, on, on +, on + +, on + + +, on standby\}$$

$$W_{TV1}^M = \{on\} \cup \{on\} = \{on\}$$

$$\Phi_{TV1}^M = \{on, off\} \cup \{standby, on, +, -\} = \{standby, on, off, +, -\}$$

Step 3

A)

And finally we can apply the formula to calculate the test cases of ALL the statechart

$$T = C * (\{I\} \cup \Phi \cup \Phi^2 \cup \dots \cup \Phi^{m-n+1}) * W.$$

$$T = C_{TV1}^M * \Phi_{TV1}^M * W_{TV1}^M$$

$$T = \{1, on, on +, on + +, on + + +, on standby\} * \{standby, on, off, +, -\} * \{on\}$$

$$T = 6 * 5 * 1 = 30 \text{ test cases.}$$

B) Now we are going to calculate it, doing the computation, but the result it's the same.

Computation of a set of test cases is $T=C*W \cup C*\Phi*W$

$$T = \{1, on, on +, on + +, on + + +, on standby\} * \{on\} \cup \{1, on, on +, on + +, on + + +, on standby\} * \{standby, on, off, +, -\} * \{on\} =$$

$$\{on, on on, on + on, on + + on, on + + + on, on standby on\} \cup$$

$\{1, \text{on}, \text{on} +, \text{on} + +, \text{on} + + +, \text{on standby}\} * \{\text{standby}, \text{on}, \text{off}, +, -\} * \{\text{on}\} =$

$\{\text{on}, \text{on on}, \text{on} + \text{on}, \text{on} + + \text{on}, \text{on} + + + \text{on}, \text{on standby on}\} \cup$

$\{\text{on}, \text{on on}, \text{on} + \text{on}, \text{on} + + \text{on}, \text{on} + + + \text{on}, \text{on standby on}\} * \{\text{standby}, \text{on}, \text{off}, +, -\}$

$T = 6 * 5 * 1 = 30$ test cases.

viii. Day 4-02-2012

“Hi K. Bogdanov,

I wish you luck in the preparation and teaching in your classes :). Don't worry about answering our emails, there is no need to hurry up. I added in the first step of TV1, the op.B and op.C as erroneous because I don't know how to do the hierarchical decomposition. I believe that the op.A is good but it seems appropriate op.C. By the moment I have worked with the op.C but it may not be adequate and may be the B or C.

On the other hand I again calculated the number of test cases by solving the mistakes that you told me. I have obtained a total of 30 test cases for this statechart using both the formula: $T = (C * W) \cup (C * \Phi * W)$ as well as: $T = C * \Phi * W$, since the result is the same.

Although I don't understand very well the difference between $T = (C * W) \cup (C * \Phi * W)$ and $T = C * \Phi * W$, because I thought that we only applied the second part of this ($T = C * \Phi * W$) once decomposed hierarchical states of the initial statechart. Enclosed in this mailing formulas applied to see what you think.

Thanks

Francisco Gramuntell Desco. Technical Engineer in Computer Science.”