

Ana Cristina Ramada Paiva Pimenta

Automated Specification-Based Testing of Graphical User Interfaces



Universidade do Porto

Faculdade de Engenharia

FEUP

Department of Electrical and Computer Engineering

November 2006

Ana Cristina Ramada Paiva Pimenta

Automated Specification-Based Testing of Graphical User Interfaces



Universidade do Porto

Faculdade de Engenharia

FEUP

Dissertação apresentada para a obtenção do grau de Doutor em Engenharia
Electrotécnica e de Computadores

Dissertação realizada sob a orientação científica de

Doutor Raul Fernando de Almeida Moreira Vidal

Professor Associado do Departamento de Engenharia Electrotécnica e de
Computadores, FEUP,

e co-orientação partilhada de

Doutor João Carlos Pascoal de Faria

Professor Auxiliar do Departamento de Engenharia Electrotécnica e de
Computadores, FEUP, e

Doutor José Nuno Fonseca de Oliveira

Professor Associado do Departamento de Informática, Escola de Engenharia,
Universidade do Minho

Novembro 2006

Abstract

Today's software systems usually feature Graphical User Interfaces (GUIs). GUIs have become an important and accepted way of interacting with today's software. They can be a crucial point in the users' decisions to use or not use the system. However, GUI testing is difficult, extremely time-consuming, and costly, with very few tools and techniques available to aid in the testing process.

This dissertation addresses the GUI testing problem. The goal is to introduce more systematization and automation into the GUI testing process by applying specification-based testing methods. The use of formal specifications allows the automatic generation of test cases containing not only the input data but also the outcomes expected. Specification-based testing methods have been applied for API testing but are insufficiently developed for GUI testing. Some of the specific challenges posed by GUI testing are addressed in this research work.

The starting phase of the GUI testing process proposed is the construction of the GUI model. Then test cases are generated from the model and are executed on the GUI implementation. The results obtained from the GUI are compared with the results derived from the specification (test oracle). Whenever there is a conformance error it is reported.

A set of guidelines are proposed for GUI modelling. For scalability and reusability reasons, GUI models are organized as a set of modules or classes. Besides modelling the atomic user actions and their effect on the GUI state, it is also possible to model composite actions (sequences of atomic actions), views (e.g., navigation map), and use case scenarios.

Test cases are automatically generated in a two-step process: a FSM is built by a bounded exploration of the GUI model first; secondly, test sequences are generated from the FSM according to some coverage criteria (e.g., full transition coverage). The exploration process calculates the set of methods available in each state (those whose pre-condition holds) and calls them with parameter values taken from domains supplied by the tester. Test cases are sequences of operations that model user actions interleaved with operations to check the outcomes of those actions.

The quality/adequacy of the generated FSM is assessed according to the degree of coverage of the model elements (actions, scenarios and views) as well as additional test conditions supplied by the tester. In order to reduce the number of test cases, it was developed an algorithm to reduce the FSM by removing redundant states and transitions with respect to the coverage goals defined.

Conceptually, during test execution test cases are run in both levels, specification and implementation, in a "lock-step" mode and their results are compared after each step. This requires the definition of a mapping between abstract actions defined in the specification and concrete actions on concrete GUI objects in the implementation. To automate this process it was developed a GUI Mapping Tool that allows the tester to interactively relate the abstract actions with concrete GUI

objects. The tool also generates automatically the code of a set of methods that simulate the concrete user actions on the GUI, and binds such methods to the abstract actions for test execution.

The approach proposed in this research is illustrated and validated by two case studies performed on two software applications: the Notepad application that ships with the Microsoft Windows operating system, and the Address Book example application freely available with the open-source Eclipse platform. In spite of being used for several years, two errors were found in the Notepad application related to uncommon sequences of user actions. Since the source code of the Address Book application is available, a mutation testing technique was applied to assess the defect detection capability of the test cases generated automatically. All defects injected were detected.

Overall, the approach proposed represents a significant improvement over the current GUI testing approaches based on Capture/Replay tools, since they only automate the execution and recording of the test cases.

Resumé

Les systèmes logiciels d'aujourd'hui comportent habituellement les interfaces utilisateur graphiques (GUIs). Les GUIs sont devenus une forme importante et admise d'agir l'un sur l'autre avec le logiciel d'aujourd'hui. Ils peuvent être un point crucial dans les décisions des utilisateurs pour employer ou ne pas employer le système. Cependant, l'essai de GUI est difficile, extrêmement long, et coûteux, avec très peu d'outils et techniques disponibles à l'aide dans le processus d'essai.

Ce travail adresse le problème d'essai de GUI. Le but est de présenter plus de systématisation et d'automatisation dans le processus de essai de GUI en appliquant des méthodes d'essai spécification-basées. L'utilisation des caractéristiques formelles permet génération automatique des cas d'espèce contenant non seulement les données d'entrée mais également les résultats prévus. Des méthodes d'essai spécification-basées ont été appliquées pour l'api examinant mais sont insuffisamment développées pour l'essai des GUI. Certains défis spécifiques posés par l'essai de GUI sont adressés dans ce travail.

La phase initiale du processus d'essai des GUI proposé est la construction du modèle des GUI. Après ça les cas d'espèce sont produits du modèle et sont exécutés sur l'exécution de GUI. Les résultats obtenus à partir du GUI sont comparés aux résultats dérivés des spécifications (oracle d'essai). Toutes les fois qu'il y a une erreur de conformité on la rapporte.

On propose un ensemble de directives pour modeler des GUI. Pour des raisons de balance et de réutilisation, des modèles de GUI sont organisés comme ensemble de modules ou de classes. Il est tant possible de modeler les actions atomiques d'utilisateur et leur effet sur l'état des GUI comme de modeler les actions composées (ordres des actions atomiques), les vues (par exemple, carte de navigation), et les scénarios de cas d'utilisation.

Des cas d'espèce sont automatiquement produits dans un processus en deux étapes: premièrement, un FSM est construit par une exploration liée du modèle des GUI; deuxièmement, les ordres d'essai sont produit du FSM selon quelques critères d'assurance (par exemple, pleine assurance de transition). Le procédé d'exploration calcule l'ensemble de méthodes disponibles dans chaque état (ceux dont les prises de condition préalable) et les appelle avec des valeurs de paramètre prises des domaines a fourni par l'appareil de contrôle. Les cas d'espèce sont des ordres des opérations que les actions modèles d'utilisateur ont intercalé avec des opérations pour vérifier les résultats de ces actions.

La qualité/adéquation du FSM produit est évaluée selon le degré d'assurance des éléments modèles (actions, scénarios et vues) aussi bien que conditions d'essai additionnelles fournies par l'essayeur. Afin de réduire le nombre de cas d'espèce, il

a été développé un algorithme pour réduire le FSM par l'enlèvement états et transitions superflus en ce qui concerne les buts d'assurance définis.

Conceptuellement, pendant l'exécution d'essai des cas d'espèce sont courus dans les deux niveaux, spécifications et l'exécution, en mode de «serrure-étape» et leurs résultats sont comparées après chaque étape. Ceci exige la définition de tracer entre les actions abstraites définies dans les spécifications et les actions concrètes sur les objets concrets de GUI dans l'exécution. Pour automatiser ce processus c'a été développé un GUI traçant l'outil qui permet à l'essayeur de rapporter interactivement les actions abstraites avec les objets concrets de GUI. L'outil produit également automatiquement du code d'un ensemble de méthodes qui simulent les actions concrètes d'utilisateur sur le GUI, et lie de telles méthodes aux actions abstraites pour l'exécution d'essai.

L'approche proposée dans ce travail est illustrée et validée par deux études de cas réalisées sur deux applications de logiciel: l'application de bloc-notes qui se transporte avec le logiciel d'exploitation de Microsoft Windows, et l'exemple de carnet d'adresses application librement disponible avec la plateforme d'éclipse d'ouvrir-source. Malgré être employé pendant plusieurs années, deux erreurs ont été trouvées dans l'application de bloc-notes liée aux ordres rares des actions d'utilisateur. Depuis le code source de l'application de carnet d'adresses est disponible, une méthode d'essai de mutation a été appliquée pour évaluer les possibilités de détection de défaut des cas d'espèce produits automatiquement. Tous les défauts injectés ont été détectés.

D'une façon générale, l'approche proposée représente une amélioration significative au-dessus des approches de essai courantes des GUI basées sur la Captation/Rejoue des outils, puisqu'ils automatisent seulement l'exécution et l'enregistrement des cas d'espèce.

Resumo

Os sistemas de software possuem normalmente uma interface gráfica com o utilizador. Este tipo de interface tornou-se a forma mais comum e importante de interagir com o software e a sua qualidade é um factor determinante na decisão de o usar. O teste de interfaces gráficas com o utilizador é difícil, moroso, dispendioso e dispõe de poucas ferramentas e técnicas.

Esta dissertação trata o problema do teste de interfaces gráficas com o utilizador. Tem por objectivo introduzir uma maior sistematização e automação no processo de teste de interfaces gráficas com o utilizador aplicando métodos de teste baseados em especificações formais. Os métodos baseados em especificações formais possibilitam a geração automática de casos de teste, com os dados de entrada e também os resultados esperados, e têm sido aplicados ao teste de software através de APIs. No entanto, estes métodos ainda não estão suficientemente desenvolvidos para testar software através da interface gráfica com o utilizador.

Na fase inicial do processo de teste de interfaces gráficas com o utilizador constrói-se o modelo e, em seguida, os casos de teste são gerados a partir do modelo e executados na implementação. Os resultados obtidos a partir da interface gráfica são comparados com os resultados derivados da especificação. Todos os erros de conformidade detectados são documentados.

A abordagem apresentada nesta dissertação propõe um conjunto de orientações para modelar interfaces gráficas com o utilizador. As interfaces gráficas com o utilizador são representadas por conjuntos de módulos ou classes por razões relacionadas com a escalabilidade e a reutilização do código. Além de se modelarem as acções atómicas do utilizador e o seu efeito na interface a testar, ainda é possível modelar acções compostas (sequências de acções atómicas), vistas (ex.: mapa de navegação) e cenários de utilização.

Os casos de teste são sequências de operações que modelam as acções do utilizador intercaladas com operações que verificam os resultados dessas acções e são gerados automaticamente em dois passos. No primeiro passo, constrói-se uma máquina de estados finita, por um processo de exploração do modelo da interface gráfica com o utilizador, e no passo seguinte, geram-se as sequências de teste, a partir da máquina de estados de finita, de acordo com determinados critérios de cobertura (por ex. a cobertura total de transições). O processo de exploração calcula o conjunto de métodos disponíveis em cada estado (pré-condição verdadeira) e invoca-os com valores apropriados dos parâmetros retirados dos domínios fornecidos pelo utilizador (aquele que está a testar).

A qualidade/adequação da máquina de estados finita gerada é avaliada de acordo com o grau de cobertura dos elementos do modelo (acções, cenários e vistas) e condições de teste adicionais fornecidas pelo utilizador (aquele que está a testar). De modo a reduzir o número de casos de teste, desenvolveu-se um algoritmo para

reduzir a máquina de estados finita removendo estados e transições considerados redundantes relativamente aos objectivos de cobertura de teste definidos.

Conceptualmente, o teste baseado em especificações executa os casos de teste nos dois níveis, especificação e implementação, e compara os resultados obtidos. Para isso, é necessário relacionar acções abstractas definidas na especificação com acções concretas em objectos concretos da interface gráfica com o utilizador. Para automatizar este processo, desenvolveu-se uma ferramenta "GUI Mapping Tool" que permite relacionar interactivamente as acções abstractas com objectos concretos da interface gráfica com o utilizador. A ferramenta também gera automaticamente o código dos métodos que simulam as acções do utilizador sobre a interface e relaciona esses métodos com as acções abstractas para execução dos testes.

A abordagem proposta nesta dissertação é ilustrada e validada por dois casos de estudo sobre duas aplicações de software distintas: o editor de texto Notepad, disponível em conjunto com o sistema operativo Microsoft Windows, e a aplicação Address Book que está disponível dentro da plataforma Eclipse. Apesar de ser usada já há vários anos, foram detectados dois erros na aplicação Notepad relacionados com sequências não comuns de acções do utilizador. Uma vez que o código da aplicação Address Book está acessível, aplicou-se uma técnica de teste baseada em mutações para avaliar a capacidade de detecção de erros dos testes gerados automaticamente. Todos os erros injectados foram detectados.

Em conclusão, a abordagem proposta representa uma melhoria significativa sobre as abordagens correntes de teste de interfaces com o utilizador baseadas em ferramentas "Capture/Replay", uma vez que estas só automatizam a execução e gravação dos casos de teste.

Acronyms

ACP – Algebra for Communicating Processes
API – Application Program Interface
ASM – Abstract State Machines
AsmL – Abstract State Machines Language
AUT – Application Under Test
BNF – Backus-Naur Form
CCS – Calculus of Communicating Systems
CIO – Concrete Interaction Objects
CIS – Complete Interaction Sequences
CSP – Communicating Sequential Processes
CTL – Computation Tree Logic
CTT – ConcurTaskTrees
DFA – Deterministic Finite state machines Automata
DNF – Disjunctive Normal Form
DTD – Document Type Definition
FSM – Finite State Machine
GUITAR – GUI Testing Framework
GUI – Graphical User Interface
HCI – Human Computer Interaction
HFSM – Hierarchical Finite State Machines
HyTech – The Hybrid TECHnology Tool
IDATG – Integrated Design and Automated Test Case
Generation
IDE – Integrated Development Environment
LTL – Linear Temporal Logic
MC/DC – Modified Condition/Decision Coverage
MVC – Model-View-Controller
NFA – Nondeterministic Finite state machines Automata
ObCS – Object Control Structure
OCR – Optical Character Recognition
OSU – Oregon Speedcode Universe
PAC – Presentation-Abstraction-Controller
RAISE – Rigorous Approach to Industrial Software
Engineering
RSL – RAISE Specification Language
SYNGRAPH – SYNtax directed GRAPHics
SMV – Symbolic Model Verifier
SWT – Standard Widget Toolkit
TAG – Task-Action Grammar's
TCTL – Timed CTL
UI – User Interface
UIMS – User Interface Management System
VDM – Vienna Development Method

VEG – Visual Event Grammar
VFSM – Variable Finite State Machine
WYSIWYG – What You See Is What You Get
XIML – eXtensible Interface Markup Language
XML – eXtensible Markup Language
XSL – eXtensible Stylesheet Language

Contents

ABSTRACT	V
RESUMÉ.....	VII
RESUMO	IX
ACRONYMS	XI
CONTENTS.....	XIII
LIST OF FIGURES	XVII
ACKNOWLEDGMENTS.....	XXI
CHAPTER I.....	1
INTRODUCTION.....	1
1.1. THE CHALLENGE.....	2
1.1.1. <i>Formal Methods</i>	3
1.1.2. <i>Specification-based testing</i>	5
1.1.3. <i>Specification-based GUI testing</i>	6
1.2. RESEARCH GOAL.....	8
1.3. METHODOLOGY	8
1.4. CONTRIBUTIONS	10
1.5. OVERVIEW OF THE DISSERTATION.....	11
CHAPTER II.....	15
GUI DEVELOPMENT AND TESTING.....	15
2.1. TYPES OF USER INTERFACES	16
2.2. DESIRED QUALITIES AND COMMON DEFECTS IN UIs.....	18
2.3. GUI CONCEPTUAL ARCHITECTURES	20
2.4. GUI DEVELOPMENT PROCESSES AND TOOLS	22
2.4.1. <i>Non model-based tools</i>	22
2.4.2. <i>Model-based tools</i>	24
2.5. GUI V&V	27
2.5.1. <i>Manual GUI testing</i>	29
2.5.2. <i>Static analysis</i>	30
2.5.3. <i>Automated GUI testing approaches</i>	39
2.6. CONCLUSIONS.....	48
CHAPTER III.....	51
SPECIFICATION-BASED GUI TESTING	51
3.1. GUI TEST AUTOMATION CHALLENGES	52
3.2. FORMAL GUI SPECIFICATION	54
3.2.1. <i>Grammars</i>	54
3.2.2. <i>Finite state machines</i>	58
3.2.3. <i>Model-based specifications</i>	60
3.2.4. <i>Property-based</i>	62
3.2.5. <i>Behaviour-based</i>	63
3.2.6. <i>Hybrid approaches</i>	67

3.3.	SPECIFICATION-BASED TEST CASE GENERATION	68
3.3.1.	Test data generation.....	69
3.3.2.	Generation of expected test results	70
3.3.3.	Coverage analysis.....	71
3.3.4.	Test generation from grammars.....	73
3.3.5.	Test generation from FSMs.....	73
3.3.6.	Test generation from model-based specifications	75
3.3.7.	Test generation from property-based specifications	76
3.3.8.	Test generation from behaviour-based specifications.....	76
3.3.9.	Test case generation from GUI models.....	77
3.4.	CONFORMITY CHECK.....	78
3.5.	CONCLUSIONS	80
CHAPTER IV		85
SPECIFICATION-BASED GUI TEST AUTOMATION		85
4.1.	GUI TESTING PROCESS.....	86
4.1.1.	Spec# System.....	89
4.1.2.	Automated model-based testing with Spec Explorer.....	90
4.2.	GUI MODELLING WITH SPEC# AND SPEC EXPLORER	93
4.2.1.	Modelling GUI structure and behaviour.....	94
4.2.2.	Modelling scenarios.....	99
4.2.3.	State machine views	101
4.2.4.	Obtain complete models from navigation maps and dialog views.....	107
4.2.5.	Independent dialogs	113
4.3.	TEST CASE GENERATION.....	115
4.3.1.	Overview of test case generation with Spec Explorer.....	115
4.3.2.	Domain definition	117
4.3.3.	Test coverage and adequacy criteria on the FSM.....	118
4.3.4.	FSM reduction.....	122
4.4.	GUI MAPPING TOOL.....	124
4.5.	CONCLUSIONS	132
CHAPTER V		135
CASE STUDIES.....		135
5.1.	NOTEPAD APPLICATION	136
5.1.1.	Model	136
5.1.2.	Scenarios.....	142
5.1.3.	Testing goals	145
5.1.4.	Choosing domain values for adequate testing	145
5.1.5.	State filtering.....	149
5.1.6.	FSM generation and reduction	150
5.1.7.	FSM validation.....	150
5.1.8.	Test case generation and execution.....	158
5.1.9.	Test results	158
5.1.10.	Metrics	160
5.2.	ADDRESS BOOK APPLICATION.....	161
5.2.1.	Model	161
5.2.2.	Scenarios.....	164
5.2.3.	Testing goals	167
5.2.4.	Choosing domain values for adequate testing	167
5.2.5.	State filtering.....	170
5.2.6.	FSM generation and reduction	170
5.2.7.	FSM validation.....	170
5.2.8.	Test case generation and execution.....	178
5.2.9.	Capacity of detecting errors.....	178

5.2.10. <i>Metrics</i>	179
5.3. CONCLUSIONS.....	180
CHAPTER VI.....	183
CONCLUSIONS AND FUTURE WORK.....	183
6.1. SUMMARY OF CONTRIBUTIONS.....	183
6.2. SUMMARY OF EXPERIMENTAL RESULTS	185
6.3. FUTURE WORK.....	185
BIBLIOGRAPHY	189
APPENDIX A	205
A.1. NOTEPAD SPECIFICATION	205
A.2. ADDRESS BOOK SPECIFICATION	217
A.3. WINDOW MANAGER AND FILE MANAGER	227

List of Figures

Figure 1: The morphism of abstraction.	6
Figure 2: Form Master/Detail.....	16
Figure 3: Seeheim architecture.....	21
Figure 4: Arch model	21
Figure 5: MVC model	21
Figure 6: PAC model.....	22
Figure 7: Model Checking.....	31
Figure 8: a) linear time; b) branching time.....	32
Figure 9: York Interactor.....	33
Figure 10: Models PiE and RED-PiE.....	36
Figure 11: Relation between windowed data and scroll bar (taken from [37]).....	37
Figure 12: Model-based testing process.....	43
Figure 13: Visual test development environment (taken from [147]).....	44
Figure 14: IDATG test process (taken from www.qualityscope.com/28.html)	45
Figure 15: GUITAR process (taken from www.cs.umd.edu/~atif/GUITARWeb/guitar_process.htm).....	46
Figure 16: Event-Flow Graph for WordPad --> Connect to Printer (taken from www.cs.umd.edu/~atif/GUITARWeb).....	46
Figure 17: Integration Tree for WordPad (taken from www.cs.umd.edu/~atif/GUITARWeb)	47
Figure 18: Petri net.....	63
Figure 19: ObCS notation (taken from [16]).....	64
Figure 20: Symbolic execution tree example	70
Figure 21: Testing flow (taken from [10])	74
Figure 22: Conformity tests model.....	79
Figure 23: Overview of the GUI modelling and testing process.....	87
Figure 24: Spec# system	89
Figure 25: Boogie static verifier.....	90
Figure 26: State variables of a textbox	95
Figure 27: Find Next pre-condition.....	95
Figure 28: Find dialog inside Notepad software application.....	95

Figure 29: Probe action example extracted from the Notepad's GUI model	96
Figure 30: Window manager	97
Figure 31: Message box of acknowledge	97
Figure 32: Message box with different possible answers	98
Figure 33: Open file scenario within the Notepad application	100
Figure 34: Navigation map obtained from focus property of the windows.....	103
Figure 35: Navigation map obtained from the enabled windows' property.....	104
Figure 36: Navigation map obtained from opened windows abstracting away the message boxes	104
Figure 37: Open dialog view obtained from the projection onto the interactive object with the focus in each moment.....	105
Figure 38: Open dialog view obtained from the projection onto the manipulated variables	106
Figure 39: Changes in the set of enabled actions inside Find dialog.....	107
Figure 40: State machine of an application with dialogs D1 (action A1) and D2 (actions A3 to A6).....	109
Figure 41: State machines of dialogs D1 and D2 projected from the FSM depicted in Figure 40. Dotted lines represent test cases	110
Figure 42: HFSM with three levels.....	111
Figure 43: Dependent dialogs	115
Figure 44: Test case generation	116
Figure 45: Open scenario view	120
Figure 46: Coverage analysis of a special case condition	121
Figure 47: GUI modelling and testing process	125
Figure 48: Architecture of the GUI Mapping Tool.....	126
Figure 49: Front-end of the GUI Mapping Tool.....	126
Figure 50: Selection of menu options.....	127
Figure 51: Examples of methods implemented in the GUI test library ..	129
Figure 52: Excerpt of the code generated automatically for the Notepad example	130
Figure 53: Test execution	132
Figure 54: Notepad main window	136
Figure 55: Open dialog	139
Figure 56: File not found message box.....	139
Figure 57: File manager module.....	140

Figure 58: Find dialog	141
Figure 59: Find scenario within Notepad application	142
Figure 60: Replace scenario within Notepad application.....	143
Figure 61: Open file scenario within the Notepad application.....	144
Figure 62: Save scenario within Notepad application.....	144
Figure 63: Navigation map obtained from focus property of the windows	151
Figure 64: Open dialog view	152
Figure 65: Find dialog view	153
Figure 66: Navigation map obtained from the enabled windows' property	154
Figure 67: Open dialog view obtained from the projection onto the manipulated variables.....	155
Figure 68: Save scenario view.....	156
Figure 69: Find scenario view	156
Figure 70: Coverage analysis of a functional dependency	157
Figure 71: Coverage analysis of a special case situation "several occurrences overlapping each other".....	158
Figure 72: Address book main window	161
Figure 73: Contact dialog of the Address Book.....	163
Figure 74: Find dialog of the Address Book	164
Figure 75: Navigation map view of the Address Book software application	171
Figure 76: Open dialog view	172
Figure 77: Save dialog view.....	172
Figure 78: Contact dialog view	173
Figure 79: Find dialog view	175
Figure 80: Close scenario view	176
Figure 81: Find scenario view	176
Figure 82: Open scenario view.....	177
Figure 83: Save scenario view.....	177
Figure 84: GUI Mapping Tool relating model action of the Address Book application with interactive controls.....	178

Acknowledgments

I would like to thank my supervisor, Professor Raul Fernando de Almeida Moreira Vidal, from Engineering Faculty of Porto University, for his guidance, determined search of resources, unforgettable mentoring and encouragement that made this dissertation possible.

A special thank is due to my co-supervisor Professor João Carlos Pascoal de Faria, also from Engineering Faculty of Porto University, for his inputs, enthusiasm and his invaluable perceptiveness in the discussions we had that enriched my perspective.

It was a privilege to have the co-supervision of Professor José Nuno Oliveira, from Minho University, to whom I would like to express my earnest thankfulness for being actively interested in my work.

I am indebted to Eng.º Vitor Santos from Microsoft Portugal for introducing me to the Foundations of Software Engineering group within Microsoft Research in Redmond, USA.

I am also grateful to the Department of Electrical and Computer Engineering, in the person of Professor Silva Matos, and to the Informatics Section, in the person of Professor Eugénio Oliveira, for having financially supported the airplane travel of my first visit to the Microsoft Research in Redmond in which I established contacts and planed collaborations that were undoubtedly important for my research work as a guide to the real problems felt by GUI testers.

My overwhelming thanks goes to the coordinator of the Foundations of Software Engineering group in Microsoft Research in Redmond, Wolfram Schulte, for the interest on my work, for supporting my stay in the first visit to Redmond, and for inviting me and supporting all the expenses of my second visit to Microsoft. I also want to thank Wolfram Schulte and Microsoft for the unconditional financial support to this research work that will foster future collaborations.

I owe special thanks to Nikolai Tillmann, researcher of Microsoft in Redmond, for the suggestions, feedback, the time we spent working together, and the talk he gave here in Engineering Faculty of Porto University. In particular, I want to thank him for his help in structuring the presentations of the research papers in the conferences ASM'05 and ICFEM'05.

During my visits to Microsoft, in Redmond, I had the privilege to meet many researchers to whom I wish to thank for their disinterested comments on my research. Among others, to Wolfgang Grieskamp, Margus Veanes, Lev Nachmanson, Colin Campbell, and Yuri Gurevich for being so kind and gentle to me.

I would like to thank Isidro Ramos Salavert, from "Departamento de Sistemas Informáticos y Computación of the Universitat Politècnica de València", and

Pedro J. Molina for being so kind with me during my visit to the University of València.

I wish to express my gratitude to my parents, Silvério Paiva and Albertina Ramada, for all their support, comprehension, and love. Thank you.

Finally, I thank my husband, João, for his encouragement, patience, support, and love, and my dear son, Rui, for being so sweet and for giving me only moments of joy.

*to my parents
Albertina e Silvério*

*to my husband João and
my dear son Rui*

Chapter I

Introduction

This chapter gives a general introduction to the main subjects of this dissertation: formal methods in software engineering, the application of formal methods to software testing, and, more precisely, the specification-based testing of graphical user interfaces (GUIs). The problems with current practices in GUI testing and how formal methods applied to software testing can help to overcome those problems are briefly pointed out. It also describes the objectives of the research work and the methodology used, presents the main scientific contributions, and gives an overview of the dissertation structure.

Our society is becoming more and more dependent on software systems. They are present in virtually all parts of modern society: airplanes and cars have computer boards, we do payments electronically, our identity information is registered on databases, we do shopping on the Internet, among others. This growing implantation of software systems makes our daily life more dependent on their functioning without errors. The correct functioning depends on the exact, unambiguous and complete capture of the customer requirements. It is well known that problems resulting from a misunderstanding of the customer requirements are the most expensive to correct, and there is a need to validate requirements as early as possible with the customer.

One of the most widespread activities to increase the confidence in the correctness of software systems is testing. Testing a software system involves executing that system with a set of inputs and evaluate whether the outputs obtained match the ones expected. There are different kinds of tests: white-box (also called structural

testing) and black-box tests (also called functional testing). In white-box testing, the knowledge of the source code is used to derive a set of test cases that cover the source code to a specific degree (all statements, all decisions, etc.). In black-box testing techniques, the software system is seen as a closed box that receives inputs and produces outputs. Test cases are derived from requirements or models of varying degrees of formality (implicit, explicit but informal, explicit and formal). When system models are used to derive test cases, the technique is called (black-box) model-based testing. Although semi-formal models (e.g., based on UML diagrams) can be used to derive test cases, in this dissertation "specification-based testing" will always mean that formal models are used. In addition, "model-based testing" will refer to testing techniques that use models which are not necessarily formal.

The use of formal models enables a rigorous approach to software developing and testing and a higher degree of test automation.

1.1. The Challenge

Today's software systems usually feature Graphical User Interfaces (GUIs). GUIs have become an important and accepted means of interacting with today's software. They can be a crucial point in the users' decisions to either use or not use the system.

However, GUI testing is difficult, extremely time-consuming, and costs a lot of money, with very few tools and techniques available to aid in the testing process.

Currently used GUI testing methods are almost ad hoc and require the test designer to manually develop test cases, identify the conditions to check during test execution, determine when to check these conditions, and evaluate whether the GUI software is suitably tested. There is no guarantee of adequate coverage according to some criteria, and the evaluation decision whether the GUI is properly tested is taken based on the developer's experience without theoretical justification. Applications are becoming bigger and more complex and manual testing of GUIs is becoming an even more difficult activity. When the GUI is modified, the developer needs to redefine the test suite and run the tests again.

There have been efforts to automate the GUI testing process. Some tools, called Capture/Replay tools (www.testingfaqs.org/t-gui.html), are commercially available. They can be used to record user interactions and replay them later. Among other problems [199], these tools still require too much manual effort and postpone the testing activity to the end of the development process when the GUI is already constructed.

1.1.1. Formal Methods

Formal Methods are "mathematically based techniques for describing system properties" [197]. They can be seen as the applied mathematics of software engineering, providing the notations, theories, models and analytical techniques that can be used to control and analyse software designs. Formal Methods can be helpful to increase the confidence in the correctness of software by proof, refinement and testing (both at the specification and at the implementation levels) [115]. Proof, sometimes called formal verification, involves a rigorous demonstration (usually involving deductive logic) that an implementation matches its specification. Refinement is the development of implementations that are correct by construction (a specification is rigorously transformed to derive an efficient implementation). An introduction to the subject can be found in [69,139]. Testing at the specification level involves executing (animating) the specification to verify (i.e., detect internal inconsistencies and problems) and validate (i.e., assure that customer requirements are correctly captured) the specification. Testing at the implementation level involves executing an implementation with some input and comparing the actual results to the ones expected. In the case of specification-based testing or conformance testing, the results expected are obtained from the specification, thus reducing the effort required to prepare them.

There are two different ways of performing formal verification: theorem proving and model checking.

Theorem proving may be supported by interactive reasoning tools based on proof systems with a set of axioms and inference rules, like simplification, rewriting, and induction. The proofs are constructed in a traditional mathematical way as a sequence of steps. The implementation and the specification are expressed through the same formal language and the goal is to verify that the implementation performs the specification. The logical implication or equivalence relation between the implementation (I) and the specification (S) is written as a theorem ($I \rightarrow S$ or $I \equiv S$) that has to be proved.

Model checking is a technique intended to prove automatically that a logical property, P , holds of a system behaviour, S , specified as a finite state machine. Properties are expressed in temporal logic that allows reasoning over the possible execution paths. In order to verify that the property holds, $S \models P$, the entire state space of the finite state machine may be exhaustively analysed. State space explosion is the main drawback of the model checking technique. SPIN (spinroot.com), and SMV (Symbolic Model Verifier) (www.cs.cmu.edu/~modelcheck/smv.html) are examples of model checking tools. The main advantage of these techniques is the fact that the proof is automatically evaluated. The main drawback of model checking techniques is the incapability to deal with infinite state spaces.

A formal specification allows capturing the customer requirements in an exact, unambiguous and complete manner. The high level of abstraction frees us from thinking about implementation and platform details focusing the attention on the real problem. Formal methods can be generically classified as model-based,

property-based, and behaviour-based. Model-based specifications describe the states of the system explicitly by using mathematical constructions like sets, lists, maps, etc. Examples of model-based specification languages are VDM [158], and Z [179]. In property-based specifications, the data types are modelled implicitly and the behaviour of the system is modelled as a set of properties. Examples of property-based specification languages are OBJ [76] and Larch [82]. Behaviour-based specifications describe systems as a sequence of possible states and are normally used to model concurrent and distributed systems. Examples of behaviour-based specification languages are Petri nets, Calculus of Communicating Systems (CCS), and Communicating Sequential Processes (CSP) [94].

Formal methods are rigorous and systematic. Nevertheless, the use of formal methods in the industry is still quite limited. Some of the reasons for such difficulty are:

- **Limited tool support:** Existing tools usually cover only specific tasks and aspects, and the integration of different tools is difficult due to different notational rules.
- **Lack of integration with other methods,** like IDEs (Integrated Development Environments), with a higher degree of acceptance in industrial environments.
- **Complexity and unfamiliarity with formal notations:** Formal notations are based on simple mathematical concepts, but some of them may seem unfriendly to software engineers.
- **Incomplete life-cycle coverage:** There is a lack of models and notations that support all the activities of software development (specification, implementation, verification and validation).
- **Limited application of Formal Methods to the development of graphical user interfaces (GUI):** Nowadays, a considerable part of the time spent in application development is consumed by the user interface. Formal specification of user interfaces is important to find errors and inconsistencies during the initial phases of development and to prove desired properties. In spite of the research work in applying Formal Methods to user interfaces, this area is not yet a common area of application.

Although formal methods are not widespread in common industry environments, it is possible to find some examples of companies that use formal methods to develop their projects all over the Europe : ATX Software in Portugal (www.atxsoftware.com); B-Core in the UK (www.b-core.com); Cinderella in Denmark (www.cinderella.dk); Clearsy in France (www.clearsy.com/html/clearsy.htm); Escher Technologies in the UK (www.eschertech.com); IFAD in Denmark (www.ifad.dk); Praxis Critical Systems in the UK (www.praxis-his.com); Prover Technology in Sweden (www.prover.com); Sidereus in Portugal (www.sidereus.pt); Telelogic in Sweden (www.telelogic.com); and Trusted Logic in France (www.trusted-logic.com).

In addition, it is also possible to find a considerable number of successful industrial experiences on the application of formal methods to real projects [45]. One example, very well known in Portugal, was the application of formal methods to solve the problem of assigning teachers to available places in high-schools. In 2004, the Portuguese government contracted a software company to develop a software system to solve the teacher's assignment problem. The problem was that the software system was not able to construct a correct solution for the problem: teachers with low priority were assigned to places that should be occupied by teachers with a higher priority. The software company that developed the software couldn't fix the problem and high-school lectures didn't start on time. Then, another company, ATX Software, developed an efficient algorithm that could solve the problem. This company used formal method to prove that the algorithm developed by them was able to finish and produce a correct solution for the problem.

The interest on formal methods from academic environments is far from ending. A list of conferences on this topic can be found in vl.fimnet.info/meetings and can easily illustrate the academic interest on this subject.

Because of its inherent rigor, formal methods have been well accepted when applied to critical systems. The same cannot be said about formal methods being applied to common systems. In this case, formal methods were the subject of severe critics. One of the critics were related to the fact that formal methods were too far away from the software development methods used in industry. It is known that testing is the most widespread activity to increase the confidence in the correctness of software systems in industries. But, testing and formal methods were traditionally totally apart activities. Today, these two methods can be seen together in software projects, complementing each other.

1.1.2. Specification-based testing

Software testing is laborious, cost intensive, and almost empirical. Formal methods, on the other hand, are systematic and have always been concerned with the formal correctness of software. They introduce system models early in the software development process with inherent advantages. Traditionally, formal methods and testing were completely separated activities. Formal methods are rigorous but not common in industrial environments. Testing activity lacks systematization but is very common in the development of software systems. By using formal methods and testing together, it is possible to systematize and automate more the testing process [26].

Specification-based testing checks if a software system's implementation conforms to the specification of the same system. Formal specifications can be used as input to generate test cases that fulfil a given criteria, to generate input data, and as an oracle to calculate the expected results. If requirements change along the software project, the specification can be modified and the test cases generated again.

The characteristics of the specification language used will influence the techniques used to generate test-cases within the specification-based testing process. Formal specification can be executable or not. When the goal is to automate the testing process, the latter can turn that goal into a more difficult one to reach.

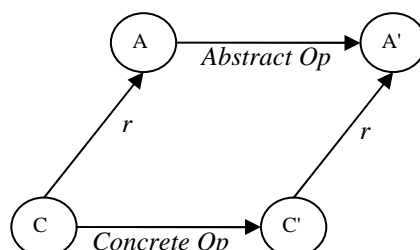


Figure 1: The morphism of abstraction.

Conceptually speaking, specification-based testing runs related operations on both levels, implementation and specification, and compares the results obtained from both in each run step. The specification operations are abstract so, to automate the conformance checking, a map (r) needs to be defined between concrete operations and states of the implementation and abstract operations and states of the specification [4] (Figure 1). An error is reported every time the concrete and abstract states or results after executing each step do not match.

One of the main problems with specification-based testing is that software systems are typically infinite or have a huge unmanageable number of states. This problem is known as **state space explosion problem** and it usually happens because the set of possible values for a particular type is boundless. For instance, the domain of possible values for an integer is only limited by the hardware constraints. In a system of 64 bits, an integer can get values from 0 to $2^{64}-1$. The challenge is to reduce the state space of the system to a manageable size and still describe the system in a level of abstraction without losing relevant behaviour from the tester perspective. There are several techniques that can be used to reduce the state space of the systems. One of those techniques restricts the domain of possible values for the variables. Even so, this technique may not be sufficient.

1.1.3. Specification-based GUI testing

It is known that nowadays a considerable part of the time spent in application development is consumed by the user interface [141] and that the user interface can be a determinant point in the decision of the users to use or not use the system. So, it is important to develop a systematic process to help reaching higher quality user interfaces.

Specification-based testing has been applied to the testing of software applications through their API (Application Program Interface), but it is not so commonly applied to the testing of software applications through their GUI. To perform GUI

specification-based testing, a GUI model has to be constructed. The GUI model can vary on the level of abstraction either modelling atomic user actions (like clicking on a button); or composed actions constructed as a sequence of atomic actions (like "drag and drop" which is the sequence of pressing the mouse button in the origin point, dragging the mouse to the destination point and releasing the mouse button); or modelling high level properties of the GUI (like GUI navigation map); or modelling scenarios that describe how the user should interact with the GUI to achieve a specific goal. The level of abstraction of the GUI model should be the best suited for the testing goals.

The construction of the GUI model may be quite laborious. However, GUIs are constructed by reusing interactive components or entire dialogs, so the GUI model should promote the reuse of already modelled behaviour.

Depending on the nature of the GUI models, different techniques can be used to generate test cases [19,53,159] from them. As soon as test cases are generated, they can be executed on the GUI in order to verify the conformity between the implementation and the specification. Test cases are sequences of operations to manipulate the GUI interleaved with operations to read and verify the results obtained after each operation performed on that GUI.

In order to automate the conformity check, a map needs to be defined between methods and states of the GUI and its specification. This can be relatively easy when the source code (or an API) of the GUI under test is available and structured as a set of operations that correspond to the actions that can be performed by a user on the GUI. However, sometimes, the only interface available is the GUI itself. In this case, some intermediate code needs to be constructed to interact with the GUI simulating the user. This intermediate code will be mapped to operations of the specification to be run in steps and results compared after each step. This intermediate code can be built based on available libraries (e.g., Win32 API, Abbot (abbot.sourceforge.net), and Jemmy (jemmy.netbeans.org), etc.) that allow to simulate user actions interacting with the GUI. However, the manual construction of this intermediate code may involve too much work which can compromise the application of GUI specification-based testing techniques.

The state space explosion problem is even more challenging when talking about GUIs. GUIs increase even further the number of possible states because there are several different modes of interacting with a GUI, like a mouse and a keyboard, different ways to achieve one goal, and there is no restriction on the sequence according to which parameters can be given.

The process of writing a specification can also be useful to find user interface errors and inconsistencies during initial phases of development and to prove desired properties that can result in time and money savings. Examples of these properties are: absence of deadlock, predictability of a command, ability to reinstate, availability of a command, succession of commands, exclusion of commands, bound of state variable and integrity constraints [152]. Also, the construction of models enables the analysis of alternative designs without having to code them.

In spite of investigation about Formal Methods applied to user interfaces, this area is not yet a common area of application.

1.2. Research goal

The goal of this research work is to improve current GUI testing methods and tools, taking advantage of formal behavioural models to enable the automatic generation of test cases and the automatic conformity checking of the implementation with respect to the specification, and hence, contribute to the construction of higher quality graphical user interfaces.

As a side effect, one wants to stimulate the use of formal methods in industrial environments. Also, with the construction of formal specifications of graphical user interfaces, we give a contribution to the construction of unambiguous documentation that can be used for other purposes besides testing. Formal specifications can be used, for instance, by code generators in such a way as to transfer legacy systems to more recent technologies.

1.3. Methodology

According to Zelkowitz and Wallace, in [200], research methodologies can be classified into scientific, engineering, empirical, and analytical. These research methodologies can vary on the type of problem they try to solve and on the type of solution they propose to solve the identified problem. A scientific method identifies a phenomenon without a scientific explanation and tries to develop a theory to explain it. An engineering method formulates a hypothesis and tries to develop and test a proposed solution. An empirical method uses statistical methods as a means to validate a given hypothesis. An analytical method develops a formal theory. The characteristics of the problem research at hand, the research question, and the solution proposed (e.g., method, methodology, theory, or tool) influence the research approach and the techniques used to validate and evaluate the approach.

The above mentioned research methodologies can be applied to science in general, but software engineering research may require specific methodology combining diverse research approaches from different research fields due to the fact that software engineering may combine several different issues, such human, organizational, and referring to computer science, so the borderline between software engineering and its scientific base is not clear defined. In [200], Zelkowitz et al. present a list of twelve software engineering validation models that are classified into three categories: observational, historical, and controlled. Observational methods gather information considered relevant during the development of the project, e.g., project monitoring, case study, assertion, and

field study. Historical methods gather existing information about projects that have already been concluded, e.g., literature search, legacy data, lessons learned, and static analysis. Controlled methods are the classical methods of experimental design used in other scientific disciplines and they gather information from different instances of an observation for statistical validity of the results, e.g., replicated experiment, synthetic environment experiments, dynamic analysis, and simulation. In addition, validation methods can also be classified according to another dimension which results in a separation between quantitative, qualitative, and hybrid evaluations [111]. Quantitative methods measure some property (or properties) of the software product or system that is expected to change as a result of the use of the approach to evaluate. Qualitative methods use "feature analysis" to describe a qualitative evaluation. Hybrid approaches combine features of the previous methods.

The research methodological difficulties of software engineering research have not (yet) been solved so the researcher has to choose a research approach which is suitable for his problem at hand [166].

The scientific area of this research work is software engineering. The research process consisted of four phases: information gathering; hypothesis definition; approach development; approach evaluation. Throughout these phases, different methods, the ones considered the best adapted, were used.

Information gathering

The information gathering was the initial phase of the research work. Collect, study, and synthesize information on the main topics for the problem defined considered relevant were the main activities involved at this stage. The goals were (1) to gain knowledge about the theory related to the research area, (2) to identify the remaining open issues, and (3) to indicate the direction for research. The information gathered was structured in an easy access database and consisted mainly of scientific papers published in magazines, journals, and conference proceeding, books, and websites. The main topics subject of investigation were: currently used approaches for developing and testing GUIs; formal methods and more precisely formal specification of GUIs; and specification-based testing. The research methods used in this phase were essentially historical methods as it is the case of literature search.

Hypothesis definition

After having gathered, studied, and synthesized the information the hypothesis was formulated:

"The use of GUI formal behavioural models enables improving GUI testing process in terms of higher degrees of automation and systematization".

Higher degrees of automation can be achieved by generating test cases automatically from formal models, and executing those tests automatically

checking conformity between specification and the GUI under test. By executing test cases automatically, it is possible to run more tests more often.

By using formal methods and testing together it is possible to increase the systematization of the testing process. Formal methods introduce models early in the software development process from which conditions to check during test case execution as well as the moment when these conditions should be checked may be inferred. In addition, formal models can be used to evaluate if the GUI software is adequately tested.

Approach development

After formulating the hypothesis, the approach was developed. The approach entails the development of a method to specify GUIs using a model-based specification language, called Spec#; the constructing of an algorithm to reduce the state space and the size of the test cases; and the construction of a tool to reduce the manual work required to perform conformity tests between a specification of a GUI and its implementation. The research methods used in this phase were observational ones.

Interaction with Microsoft researchers and testers was crucial in this phase of the research process to understand the real needs and problems of the GUI testers and as a way to discuss and exchange ideas.

Approach evaluation

Once constructed the proposed solution for the identified problem, observational methods, like case study, and controlled methods, like replicated experiments, were conducted to validate the solution.

The results obtained during the research work were presented and discussed in international conferences after being approved in its reviewing processes.

1.4. Contributions

The main contributions of the research work spread over the three identified GUI testing problems:

1. GUI modelling problem
 - A GUI modelling approach that provides a set of guidelines for modelling GUIs for testability and reusability (GUI components are specified as reusable classes or modules).
2. State space explosion problem
 - An algorithm to reduce the state space of the GUI model and consequently the size of the test suite based on a hierarchical structure of the GUI model [151].

3. Model-to-implementation mapping problem

- A tool to automatically construct the code needed to interact with a GUI simulating the user [149]. The main goals of this tool are:
 - to reduce the manual work required to test an application through its GUI;
 - to bridge the gap between a model written in a high-level modelling language and the simulation of user events;
 - to test GUI applications even if their source code isn't available.

These contributions (1-3) were described in the following papers presented in international conferences after being approved by a reviewing process:

- (1) (3) – "A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing" presented at the 7th International Conference on Formal Engineering Methods (ICFEM'05), 2005.
- (1) (2) – "Modelling and Testing Hierarchical GUIs" presented at the 12th International Workshop on Abstract State Machines, 2005.
- (1) – "Automated Specification-based Testing of Interactive Components with AsmL" presented at the 5th edition of the international conference QUATIC (Quality: the bridge to the future in ICT), 2004.
- (1) – "Specification-based Testing of User Interfaces" presented at 10th DSV-IS Workshop - Design, Specification and Verification of Interactive Systems, 2003
- (1) – "Métodos Formais na Especificação de Interfaces com o utilizador: a linguagem VDM++ e o tratamento de eventos" presented at the "3^a Conferência da Associação Portuguesa de Sistemas de Informação", 2002.

Each paper is preceded with numbers within round brackets that identify the contributions described in each one of them.

1.5. Overview of the dissertation

This dissertation is structured into three main logical sections. The first one presents a review of the several approaches to develop and test GUIs. It is spread over Chapters I, II, and III. Chapter II presents techniques and tools to develop and test GUIs without the support of formal methods. Chapter III presents techniques for specification-based testing of GUIs. The second section presents the approach proposed in this dissertation in Chapter IV, which is validated and evaluated in Chapter V. The third part presents conclusions and future work.

Chapter I

This chapter gives a general introduction to the main subjects of this dissertation: formal methods in software engineering, the application of formal methods to software testing, and the specification-based testing of graphical user interfaces (GUIs). The problems with current practices in GUI testing and how formal methods used in combination with software testing can help to overcome those problems are briefly pointed out. It also describes the objectives of the research work and the methodology used, and presents the main scientific contributions.

Chapter II

This chapter begins by classifying the different kinds of user interfaces and their desired qualities and common defects. After that it gives an overview of the current practices in the GUI development process and presents their main problems. An overview of the current practices for testing GUIs is presented and compared with other approaches to promote the quality of GUIs. The main drawbacks of each described approach are then pointed out.

Chapter III

This chapter opens with the presentation of the main challenges of Graphical User Interface (GUI) testing either when compared to Application Programming Interface (API) testing or when one wishes to automate the test process. After that it presents a survey on the work related with GUI specification-based testing. It begins by describing different ways of modelling GUI using different kinds of formal specification languages and then presents different techniques used to generate test cases from those different formal specifications. At the end, different strategies of performing automatic verification of the test results (conformity check) influenced by the kind and style of the specification used are presented.

Chapter IV

This chapter presents a new approach to model and test GUIs. The model is written in Spec# and structured in a hierarchy. The methodology followed and the decisions taken to model GUIs are explained in detail. The hierarchical structure of the model is used by an algorithm to reduce the number of states of the model and contribute to diminish the state space explosion problem. At the end of the chapter, a tool prototype to support the specification-based GUI testing is described. This tool is an extension of the specification-based testing tool, Spec Explorer, developed at Microsoft Research that already supports automatic generation and execution of test cases for API testing, but still requires too much work to test software applications through their GUI.

Chapter V

This chapter presents and discusses the results of the case studies used to evaluate and validate the specification-based testing approach proposed in this dissertation.

Chapter VI

This chapter presents the main achievements of the research work described in this dissertation and points out topics which deserve future attention.

Chapter II

GUI development and testing

This chapter gives an overview of the current practices in the GUI development. It starts by classifying the different kinds of user interfaces and their desired qualities and common defects. It then offers an overview of the current practices for GUI testing. Other approaches to promote the quality of GUIs are presented, compared, and their main drawbacks pointed out.

User interfaces (UIs) are mediators between users and systems. Users interact with user interfaces to perform tasks. A UI is a crucial part of an interactive system in the sense that it determines how system is. It can then be a determinant point in the decision on whether to use or not to use it. A UI provides ways of controlling the system through inputs and ways to observe the system through outputs. There are different modalities in which inputs and outputs can be sensed, for instance, vision and audition. Different modalities can be combined in the same system and for the same task there can be a multiplicity of different modalities available. For instance, a user can see outputs of the system in a computer monitor and send inputs to the system through sensors or devices like keyboard, mouse, and touch screens. The ways in which these modalities are implemented give origin to different interaction styles.

2.1. Types of User Interfaces

There are two main user interface styles: command-line and graphical user interfacing (GUIs).

Command-line interfaces (CLI) are examples of synchronous and sequential user interfaces. The dialog between the system and the user is established as a sequence of questions and answers. At each execution step, the system waits for the user command, processes it, writes the output, and moves on to another execution step. An example of this type of interfaces is the Unix Shell.

Graphical user interfaces (GUIs) are richer CLIs in the sense that they can support other kinds of interaction-styles like form fill-in, menu selection, and direct manipulation. A GUI may have multiple windows on screen with interactive objects, like menus, and buttons, mixed with text in a graphical display which creates a more pleasant environment than text-only terminals. Windows allow users to switch among multiple tasks, or multiple parts of a single task. Typically, the user can resort to the mouse as a pointing device to select a command from the menu, rather than type the equivalent command in a command language, click on a button, select an item, or drag and drop an item.

When interacting through GUIs, the order in which tasks are performed is arbitrary. In particular, users can interrupt one task to interact with another window/dialog, e.g., to get information from a database, and then return to complete the first task, e.g., by using the information previously read from the database. The concept of "multi-threaded dialog" is used for this kind of interaction [175].

It is common to let information exchange among sub-dialogs of the same application and among related data. One example for the latter case is a dialog that shows information gathered by two different tables, *A* and *B*, of a database associated by an one-to-many relation (Figure 2). Typically these dialogs have a *master/detail* structure that allows one to select a particular object of the first table (*A*), as *master*, and shows the *detailed* information of that particularly master gathered from table *B*. Every time the user changes the selection, the detailed information should be updated accordingly.

Department:	Marketing
Employees:	
	John Smith
	Carl Cooper
	Peter Dix

Figure 2: Form Master/Detail

Another particularity of GUIs is semantic feedback. Semantic feedback refers to outputs made visible to the user that are application-specific. For instance, a graphical editor of entity relationship diagrams may display a message warning users whenever trying to connect two relationship objects, which is a meaningful and not allowed operation [175].

There are different kinds of GUIs: hypertext, web-based, form-based, direct-manipulation, rich client, multi-modal, and virtual reality.

Hypertext is a non-linear way of presenting information to the user. The information is structured as a network of nodes and links in which readers are free to navigate and create their own reading order. This kind of user interfaces does not support the drag and drop interaction style.

Web-based user interfaces provide a way to access infra-structures and applications from remote computers using internet or intranets. They accept input and provide output by generating web pages which are transported via the internet and viewed by the user using a web browser program.

A **form-based** interface is an independent graphical window, with a set of embedded controls. It can be seen as electronic version of a paper form that common public services ask clients to fill-in. Form-based user interfaces allow for typing information and pointing with the mouse.

With a **direct manipulation** interface, the user seems to operate directly on the objects visible on a graphical display using actions more similar to the actions in the physical world. Examples of direct-manipulation are window resizing or changing the directory of a file by dragging and dropping the icon that represents it on the new location.

Rich client or **smart clients** are software applications that can work online or offline whether connected or disconnected from the internet. Microsoft Outlook is an example of this kind of software applications. It can only check for new mail messages when connected to the internet but it allows reading previously received messages even when disconnected from the internet.

Multi-modal systems are a sophistication of standard GUIs. The goal of these systems is to make communication with machines easier. They intent to extract meaning from the different possible ways of communication among humans, like speech, gestures, and visual recognition, and use such modal inputs as inputs to the system.

Virtual reality user interfaces are examples of concurrent and real-time interfaces. They use computer-generated graphics to simulate a real or imagined environment with three dimensions of width, height and depth for the user to enter, explore and interact with. The user can manipulate more than one device at a time to achieve a goal which may vary from common devices like keyboard and mouse to more sophisticated ones like data gloves and head-mounted displays. Computer games are examples of this kind of user interfaces.

The focus of this research work is on form-based, direct manipulation, and rich client GUIs.

In the sequel, GUI means form-based, direct manipulation, and rich client user interfaces. When there is a need to mention other kinds of GUIs, they will be individually cited.

2.2. Desired qualities and common defects in UIs

Regardless of its type, the quality of a given UIs can be evaluated from two different perspectives: the user's perspective (external), and the software engineering's perspective (internal) [77].

External perspective

The user's perspective is more concerned with the so called usability properties of the system. These properties, which are indicators of how easy it is to use the UI, can be classified as follows:

- **Satisfaction** – this is related to the user's subjective view of the systems, e.g., how pleasant, comfortable, intuitive, consistent it is.
- **Reliability** – from the user's perspective, this refers to the errors a user can do when using the system. This property is closely related to the degree of flexibility of the system. A flexible system gives more freedom to the user and more opportunities to fail while a rigid system gives less freedom to the user but less opportunities to fail.
- **Learnability** – this refers to the time users take to learn how to work with the system and how much the users recall when redoing a task.
- **Efficiency of use** – this refers to how efficient the user can be when performing a task using the system. This can be measured by the time taken and/or the number of actions needed to perform a task. An inefficient UI can be usefulness.

Internal perspective

From the software engineer perspective, UI quality is judged in a way similar to other parts of the system, as follows:

- **Code** – assess its readability, logical structure; easiness of maintenance, style, etc.
- **Architecture** – represents systems in terms of abstract components with external visible properties and relationships. The architecture of a system can influence the degree of manageability and scalability.
- **Run time efficiency** – time needed to underlying the execution is closely related to the complexity of the algorithms.

- **Correctness** – There are different ways of defining software correctness [160]: the software is correct if it meets its specification, also known as verification, and either specification or software is correct if it meets the requirements of its users, also known as validation. Breaks in the contract established with the user are detected during validation process. Errors or discrepancies between the calculated/computed and expected values are detected during the verification process. Both processes are important to increase the confidence in the correctness of the UI.

GUI errors

A GUI can increase the number of errors or failures of the underlying application of a software system. The different types of GUI errors/failures can be classified as follows:

Usability errors or failures are related to the difficulties the user has to overcome when using the system. They can be due to problems of communication between application and users, confusing command structure and entry, and feedback missing [108]. There are errors in the communication between the application and the user whenever the user is expecting information which is missing to continue his task or, for instance, when messages shown to the user are not clear or have spelling errors. The user can easily get lost when the command structure and entry is confusing like when there are inconsistencies with names, menu positions and command entry style. System feedback should be complete and understandable to the user in order to make them easier to use. However, there may exist problems in the output of certain data, it can be impossible to redirect output, and it may be difficult to control output layout (e.g., colours, font, scaling graphs, etc).

Usually, to detect errors from an external perspective, the system is tried out by real users in controlled environments. Information is gathered by asking the users to fill in forms, or by gathering information about the time spent to achieve a given goal, or time spent in redoing a previously performed task, or the number of steps needed to perform a task. The information gathered is then analysed and the system is improved accordingly.

Functionality errors or failures are related to the tasks or functionality the system should support. Problems or errors are detected when the system behaves unexpectedly; or performs in an awkward or incomplete manner; or even when it does more than it was expected to. Functionality may be missing because there are commands missing or existing commands are either not available or do not work. The system does not do what was expected when, for instance, it does not ensure data validation, provides incorrect field defaults, when it does not enforce mandatory fields, when wrong fields or wrong number of rows are retrieved by queries, when windows have incorrect modality, when derived values are not updated or wrongly calculated, and so on.

To detect functionality errors, one need to know before hand what the intended functionality of the system is, that is, what its expected behaviour is. This can be kept in the developer's head only, in an informal requirements' document or in a

formal specification. The way in which requirements are kept bears a strong impact on which technique to use to find errors.

Performance errors/failures are non-functional errors. These errors are related to the efficiency of the system. They can be measured in terms of the time taken to perform a task, or the amount of resources consumed. Errors are detected when the system takes too much time to perform a given task, like, for instance, the time taken to show a message to the user or the time taken to move the cursor to the end of a text file.

Sometimes, performance testing is combined with stress testing to check what happens when a load bound is exceeded. Usually, performance errors are detected with the help of tools that are capable of measuring how performance varies, for example, with the load number of users vs. response time. An example of this kind of tools is Compuware Corporation's QACenter Performance Edition (www.compuware.com/products/qacenter).

A main concern of this work is on finding functionality errors or failures. By testing a software application through its GUI it is possible to detect defects related to the underlying application and also related to the GUI itself.

2.3. GUI conceptual architectures

The GUI model-based development process comprises requirements such as capture, design, implementation, verification/testing, and maintenance. However, tools that support GUI development process present deficiencies on the modelling and verification phases. Typically, models only exist in the programmer's head and the verification phase is restricted to the realization of manual tests without systematization concerns.

Among the first attempts to make UI development more systematic, we find UIMS (User Interface Management Systems), which are based on conceptual architectures that make a clear distinction between the presentation and the application. The goal was to increase the portability (degree of independence between the presentation level and the underlying application) and adaptability (the capacity of the systems to deal with changes, e.g., requirements' changes, system improvements and correction of errors) of the systems. These architectures can present a layered or an object oriented structure. Examples of these architecture models are Seeheim (Figure 3), Arch (Figure 4), MVC (Figure 5), and PAC (Figure 6) models.

The Seeheim model was inspired in linguistic systems. It splits the system into lexical, syntactic, and semantic aspects that correspond to presentation (*P*), dialog (*D*), and application interface (*AI*) respectively (Figure 3).

The box at the bottom is a controller. It receives messages from *AI* and *D*, and sends messages to *D* and *P*.

The presentation layer describes the interactive objects and the data presented by them. The dialog layer gets input data and determines how they should be treated. The application interface describes the services available to the user.

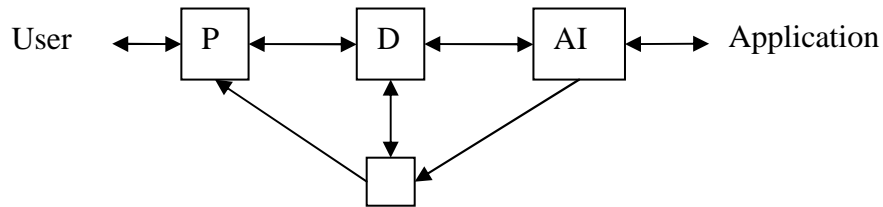


Figure 3: Seeheim architecture

The Arch model adds more structure to the Seeheim model by refining the Seeheim presentation component into interaction toolkit and presentation adapter component, and refining the Seeheim application interface into domain-specific component and domain adapter component. The adaptors contribute for the improvement of the code reusability, portability, and modifiability [51].

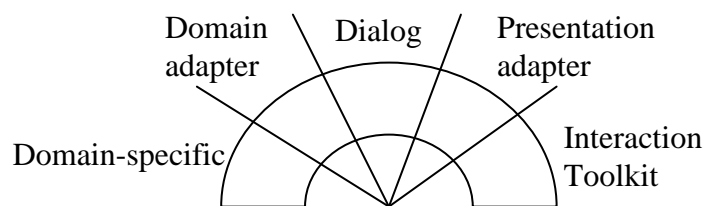


Figure 4: Arch model

With agent-based models, the interactive systems are structured as a collection of agents. These active agents, also called interactors because they communicate directly with the user, are capable of producing and reacting to events.

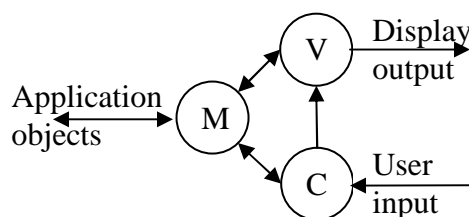


Figure 5: MVC model

The MVC (Model-View-Controller) model splits the system into a model (*M*) of the objects of the domain, a view (*V*) for making instances of the objects visible to the user, and a controller (*C*) to deal with the data received from the user. The view is notified whenever the information kept by the model changes.

The PAC (Presentation-Abstraction-Controller) model splits the interactive system [51] into presentation for implementing the perceivable behaviour of the agent (interactor appearance), abstraction for the competence of the agent (functional core), and control for linking the abstraction part of the agents to its presentation and maintenance of the relationship of the agent with other agents.

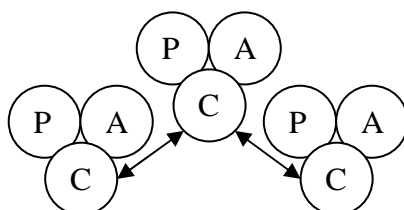


Figure 6: PAC model

The main differences between MVC and PAC models are the way in which synchronisation of related interactors is achieved, and the location of input and output responsibilities [96].

These architectures split the interactive software systems into the application and the user interface. Although this separation has its merits, it also leads to serious adaptability problems when functionality of the software system has both application and user interface aspects that cross the application-interface boundary [63].

2.4. GUI development processes and tools

Myers and Rosson [141] estimate that an average of 48% of the application code and 50% of the time spent with implementation are dedicated to the user interface. To increase the productivity of user interface (UI) development teams, some tools have been developed to aid the construction of user interfaces. These tools can be classified into two major groups: non model-based tools, and model-based tools. Interface builders with toolkits on top of window managers, IDEs (Integrated Development Environments), and markup languages are included in the former case. In the latter group, it is possible to include tools that automatically generate the final GUI from the model (MB-UIDE – Model-based User Interface Development Environments, Pattern-based, and CASE – Computer-aided Software Engineering tools), tools that generate automatically the model of an existing GUI by a reverse engineering process, and prototyping tools.

2.4.1. Non model-based tools

These tools are widespread in industrial environments. They are characterized by not requiring an explicit GUI model and for being used to build the interface itself

and nothing else. Examples of these tools are interface builders, IDEs (Integrated Development Environments), and scripting languages like Tcl/tk, XML and XSL.

User interface builders

User interface builders provide UI components or widgets from a toolkit that the developer can use as building blocks in the construction of new graphical user interfaces. The developer can manipulate those widgets in an interactive environment to graphically construct the layout of the screens and automatically generate part of the interface code. These tools work on top of window managers and are WYSIWYG (*What You See Is What You Get*) oriented. Whenever GUIs change dynamically, these tools are useless. Dynamic changes must be programmed manually. Even so, there are studies saying that these tools can reduce to half the time spent with GUI development [140]. Examples of these tools are Nextstep [122], and Visual Basic [132].

One of the problems with interface builders is that they do not support modelling and verification phases. In addition, they entail an early commitment to the concrete interaction objects (CIO), physical properties and details of the display.

IDEs

An Integrated Development Environment (IDE) seems like a single tool where all the development is done. Typically these environments integrate a source code editor, a compiler and/or interpreter, build-automation tool, and a debugger. Examples are: Microsoft Visual Studio [133] and Eclipse (www.eclipse.org). On behalf of using interface builders, these tools are also limited to handle only the static parts of the interface. The dynamic behaviour has to be programmed manually.

Visual programming environments are a special case of IDE where the software application is constructed graphically as building blocks of code.

Although IDEs are widespread over industries, they lack on their support for the UI development process. They also have a weak support for the modelling and verification phases.

Markup languages

Nowadays, the growing diversity of devices makes it more and more common to find the development of software applications for multiple-platforms. However, it is difficult to construct those applications without duplicating the development effort.

XML (eXtensible Markup Language) technology makes a good separation between content and presentation aspects of a user interface. They import concepts from conceptual architectures described in section 2.3. XSL (eXtensible Stylesheet Language) is concerned with the style and layout while XML is concerned with data. The same XML file can be associated with different XSL

producing different web UIs or HTML files for different devices, languages, or connections. Even so, XML and XSL do not capture the essence of user interfaces like user interface description languages try to do [8,178,187]. These languages describe the user interface at different levels of abstraction trying to address different purposes such as device, platform, modality, and context independency. Examples are AAIML (Alternate Abstract Interface Markup Language), AUIML (Abstract User Interface Markup Language), XIIML (eXtensible Interface Markup Language), XUL (eXtensible User Interface Language), XAML (Microsoft eXtensible Application Markup Language), UIML (OASIS User Interface Markup Language), UsiXML (USer Interface eXtensible Markup Language), etc. Most of them can be found in (xml.coverpages.org/userInterfaceXML.html).

2.4.2. Model-based tools

Different from the previous tools, these are characterized by requiring a GUI model in which aspects of the user interface design are represented. The aim of these tools is to support the systematic and efficient development of user interfaces providing the developer with better methods for constructing UIs.

MB-UIDE – Model-Based User Interface Development Environment

MB-UIDE (Model-Based User Interface Development Environments) appeared as an improvement of the user interface management systems (UIMS) driven by the goal of executing UIs from declarative models [171]. The focus of the first model-based generation tools was on automatic generation of preliminary user interfaces from declarative models while the second generation of tools focused on supporting user interface design by the involvement of the users in the development process [186].

Typically, the kind of models used by the first generation of tools was based on domain models with weak expressive power. From these models, it was possible to generate form-based user interfaces with a simple menu. This kind of UI could work for restricted situations, like data driven applications, or form-based user interfaces, where tasks are mainly related to data maintenance such as create, retrieve, update, and delete (CRUD), but not for the wide spectrum of UIs. Examples of this kind of tools are: UIDE [17]; MECANO [161] (predecessor of MOBI-D [163]); AME [118]; and JANUS [13]. Other systems increase the expressive power of their models which allow them to generate richer user interfaces, e.g., ITS [195,196], and generate additional features like help and redo/undo sub-systems, as it is the case of the HUMANOID tool [182].

The second generation of model-based tools was targetting at getting input from the users in order to improve the usability and usefulness of the systems. This is called the user-centred design paradigm (UCD) which places the user at the centre of the UI development process. The design is driven by an iterative prototyping process based on a "trial-and-error" evolution [50]. The focus is on cognitive issues such as perception, memory, learning, problem-solving, etc. In addition,

some of these tools, such as TRIDENT [25] and DON [110], also evaluate models for various qualities. Other examples of tools from the second generation are: ADEPT [117]; MASTERMIND [183] (which is a continuation of the previous work on HUMANOID and UIDE); TADEUS; GENIUS [100]; FUSE [113]; MOBI-D [163] (successor of MECANO); Teallach [80,81]; and DRIVE [134].

Typically, the interface model used by the second generation of tools is structured into many declarative models, like domain, user, task, dialog, and presentation models [167]. The most crucial model in supporting a user-centred design philosophy is the user-task model [162]. The user model describes the characteristics and abilities of the users. The task model describes the significant tasks that the users have to accomplish. These descriptions are then used to determine which tasks the system should support.

There are different notations in the literature to describe task models [193]. In particular, most of the grammar-based models are descriptions of the user's tasks. UAN (User Action Notation) [90] and ETAG (Extended Task Action Grammar) [84] are examples of these notations. Other examples are CTT (ConcurTaskTrees) [154] and TKS (Task Knowledge Structures) [104].

Pattern-based

A pattern can be defined as a reusable solution for a recurring problem that occurs in a certain context of use [172]. Patterns enclose a significant amount of reusable knowledge and can be an effective way to transmit experience about recurrent problems in UI development domain [172]. Sinnig et al. describe a model-based framework with models constructed from a generic notation of patterns and tools to integrate those patterns into the development framework [172]. Patterns are defined dynamically with variables that are replaced by concrete values for a particular context of use during the pattern adaptation process [173]. Tasks can be grouped in dialog views. Dialog views and transitions from them can be saved in XIML. A first abstract prototype can be generated from the dialog description.

Molina, in [135,136,137], describes a set of conceptual patterns for business applications' user interfaces. The concepts and patterns are then used to model object-oriented user interfaces in a graphical notation where interaction units are represented as boxes and navigation between units as directed arrows. The pattern language (which is independent from implementation technology) is precise, and non-ambiguous. The models built in this language can be read by code generators for several different target implementation languages.

CASE – Computer-Aided Software Engineering

The aim of Computer-Aided Software Engineering (CASE) tools is to automate, manage, and simplify the software development process. Examples of these tools are Oracle Designer (www.oracle.com) and Rational Rose (www.ibm.com). However, these tools present a very long learning curve.

Considering only the domain of applications supported by CASE tools, and after overcoming the initial learning effort to work with them, the construction of new applications can be very fast. These tools can also be useful to construct prototypes. The problem is that many users do not overcome the initial obstacles.

Reverse engineering

The world is full of legacy systems. The technology is in constant change and some companies need to update their old systems. Reverse engineering tools can be used to build the model of existing applications that can be used by UIMSs to generate new GUIs with the same functionality of the older ones, but implemented in more recent technologies, or to be accessed from other computer platforms with specific characteristics.

One common example is the migration of legacy user interfaces to web-accessible platforms in order to support e-commerce activities. Stroulia et al. describe the CelLEST system within which a new process for migrating legacy systems for the Web was developed [180,181].

Vanderdonckt et al. describe a reverse engineering process of Web user interfaces [190]. The goal is to extract models of Web applications that were not constructed using a model-based approach and then use those models to generate UIs for other computer platforms, like palms, pocket computers, and mobile phones, without losing the effort deployed in the construction of the initial application.

Prototyping tools

Prototypes are visual representations which may or may not be animated. The animated prototype must be capable of generating an interactive environment which accurately emulates the intended system operation.

Prototypes can be discarded after implementation of the final product (throwaway prototype) or used in an evolutionary scenario where it suffers changes until it becomes a final product (evolutionary prototype).

Throwaway prototypes can be developed manually using paper and pencil or with the help of a tool (like HyperCard and Director [157]). Prototyping tools, like HyperCard and SuperCard (*supercard.us*), provide an interface builder with which it is possible to drag and drop widgets (abstracted as cards) onto a black window where the author can manipulate them. However, to do something, the authors must leave the interface builder shell and write code in a sample script language, like Hypertalk.

Evolutionary prototypes can be constructed using tools such as Visual Basic [132] from Microsoft and PowerBuilder (www.retrossoftware.com/12016.html) from PowerSoft (www.powersoft.it).

Another approach in rapid prototyping is the so called Abstract Prototype from Larry Constantine [49]. Abstract prototypes can represent the contents of a user interface without showing how it looks like. The goal is to abstract from

implementation details and detect usability problems during the modelling phase. This approach is based on usage-centred design. Usage-centred design is focused on the "usage" as opposed to "user" on which user-centred design is focused. It is based on three abstract models: a role model (describes the users' roles in relation to the system), a task model (describes the structure of the users' work), and a content model (describes the content and organization of the user interface needed to support the identified tasks).

Problems with GUI development processes and tools

The tools described so far were very useful to increase the productivity of UI development teams. However, they have some problems due to the fact that they do not support all the activities of the UI development process, namely:

- Interface development tools were developed to reduce the time spent with UI development but they have no concerns with systematization of the process. They do not support modelling, verification, and maintenance phases.
- Model-based tools make UI development more systematic but they are also subject of critics:
 - Poor UIs generated based on standardized interface elements.
 - Suited for specific UIs but useless for not directly supported interfaces.
 - Most of them do not take dynamic semantics of the application under consideration.
 - The developers are not given enough control over interface details.
 - It is difficult to relate characteristics of the model with final UIs generated from the model and there is little control over the look and feel of the final UIs.
 - Developers have to learn one more language: the specific language of the tool.
 - Verification and evaluation phases are not supported.

2.5. GUI V&V

The verification and validation (V&V) phase of the software life cycle may consume around 50% of the total time of the project [20,26,164]. It can be performed by static or dynamic analysis. In the former case, instead of executing the application under test, methods like code review and formal analysis like model checking and formal proofs are used. In the latter case, the analysis is

performed by executing the application under test, e.g., specification-based testing and beta-testing.

Although there have been improvements in static V&V techniques such as model checking and theorem proving, testing is still the most widely used technique to evaluate the quality and increase the confidence on software systems. It can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence [56].

One of the problems with testing is the lack of systematization. Most often, tests are performed manually without coverage criteria, based only on the good sense and sensibility of the tester. However, the complexity of software systems is growing and having to deal with several different input values and different possible outcomes manually is becoming an unmanageable activity. In addition, without defining coverage criteria, determining when to stop testing and evaluating the tests performed is almost ad hoc, once again, based only on the experience and sensibility of the tester.

Every time the software suffers changes, tests have to be run again. Tests performed in the software after being changed are referred to as regression tests. The goal of these tests is to assure that:

- the source code added or modified didn't introduce new errors;
- the program still acts in accordance with requirements; and
- the unchanged code was not affected by the modification.

Another problem with these tests is that they are delayed to the last phases of the software development process. This happens with so-called white-box testing techniques which need knowledge of the programming code to select test data. At this point in time, when the code is already constructed, the errors detected are the most costly to correct which can have impact on the estimated conclusion date of the project.

There is another kind of testing techniques in which the software is regarded as a black-box. The only thing needed by these techniques to construct test cases is the specification of the program describing the expected outputs for different inputs. In this case, the test cases can be constructed sooner than with white-box testing techniques. When the specification is formal, the construction and execution of the test cases can be automated and the overall process becomes more systematic.

In general, testing strategies applicable to API testing can also be applied to GUI testing. However, GUIs testing raises specific challenges due to time constraints, test case explosion problems, the need to combine testing techniques, and test automation difficulties. This will be explained in more detail later on in section 3.1.

Very few tools and techniques are available to aid the GUI testing process. By contrast, GUIs are getting more and more established in our daily lives which make us more dependent on their correct functioning. GUIs are becoming more and more complex, which makes manual GUI testing unpractical. Like in API

testing, the GUI testing process can be automated although, current practices in GUI testing is still a manual activity.

2.5.1. Manual GUI testing

Manual GUI tests are useful in exploratory/initial testing. Also, manual tests are especially well adapted for being performed by real users. Beta releases are tested by real users for a couple of weeks in order to find errors. This approach, also known as random human testing, lacks systematization and offers no guaranties of covering all the functionalities of the application.

There are other kinds of manual tests, more systematic, in order to find GUI problems. Whenever they are performed by trained specialists it is possible to find more bugs per test case executed and bugs found can provide hints to find other bugs, i.e., the tests can be adapted to look for bugs similar to the ones found (adaptability). These can be classified into inspection, inquiry, and usability tests.

Inspection

A group of specialists examines the user interface regarding a set of guidelines. Those guidelines can vary from detailed characteristics about physical properties of the UI to board principles based on usability studies for making interfaces more intuitive, learnable, and consistent, e.g., how to organize the display and the menu structure. Examples of inspection methods are heuristic and cognitive walkthrough [74].

Heuristic Methods

A group of specialists studies the interface in order to find usability problems. These problems are detected when the elements of the user interface do not follow the usability heuristics used to guide the evaluator through the inspection process. Whenever problems are detected, they are written down and classified in order of severity.

Cognitive walkthrough

The developers walk through the interface in the context of core tasks a typical user (not an expert) will need to accomplish. The actions and the feedback of the interface are compared to the user's goals and knowledge, and discrepancies between user's expectations and the steps required by the interface are noted.

Inquiry

The users have opportunity to experiment the software system and then answer questionnaires about their experience. Questions can vary from subjective to

objective, for instance, "do you think the system is nice?" or "what would you change in the system?" or questions about screen features and system information provided, like error messages.

Usability tests

The interface is studied under real-world or controlled conditions (real users), with evaluators gathering data on problems that arise during its use [101]. The interaction characteristics are measured and weaknesses are identified for correction. The data gathered has information about time (the time a user takes to complete a task), accuracy (the number of mistakes the user makes), recollection (how much the user needs to recall when redoing a task), and emotional response (how does the user feel after completing a task).

Manual test disadvantages

The results/errors found by manual tests are very dependent on the capabilities of the tester. Manual testing is monotonous, frustrating, and affected by human errors. Too much effort is required to construct, execute, and analyse the results of the test cases. Manual tests may be difficult to reproduce/repeat and when software is updated the test cases need to be run again given that there is no support for regression testing. The errors found by manual tests are dependent on the expertise of the specialists, who are difficult to find. Also, manual testing is based on weak coverage criteria.

Manual tests are appropriate for finding usability problems and making general assessments about usability but not for predicting usability measures [99]. For that, software engineering practices like model-based development and simulation are more appropriate. Examples of models used to predict usability are ETIT (external/internal task mapping), TAG (task-action grammar), GOMS, PUM, CLG (command language grammar), ETAG (extended task-action grammar). They are classified according to the different aspects they are able to predict in [99]. Simulation methods simulate the user's interaction with the interface reporting performance measures and interface operations [99].

2.5.2. Static analysis

Static methods analyse the code or specification of a software system in order to find constructs that break certain correctness criteria. These methods do not involve the execution of the software under test.

Static analysis performed on code, code inspection, can provide feedback to the developer, for instance, when common errors are found, when guidelines are not followed, and when UI components are not used appropriately, e.g., a button without a Click event handler.

Static analysis performed on a formal specification is called formal static analysis. Model checking and theorem proving are the basic types of formal verification. They both have advantages and drawbacks as we will see next.

Model checking

Model Checking is a formal verification technique that has been successfully applied to hardware, communication protocols, and also reactive systems.

The system is modelled as a finite state machine (FSM) and properties that the model should obey are written in temporal logic (see Figure 7). Model checkers are then used to prove automatically by exhaustive analysis of the entire state space of the system that those properties hold in the model of the system. This can be expressed mathematically as: $S \models P$, meaning that property P holds in the system S (specified as a finite state machine).

The result obtained by a model checker can be either true (the properties hold) or false in which case a counter-example may be provided. The counter-example is a path, sequence of states, within the transition system that shows the property failing.

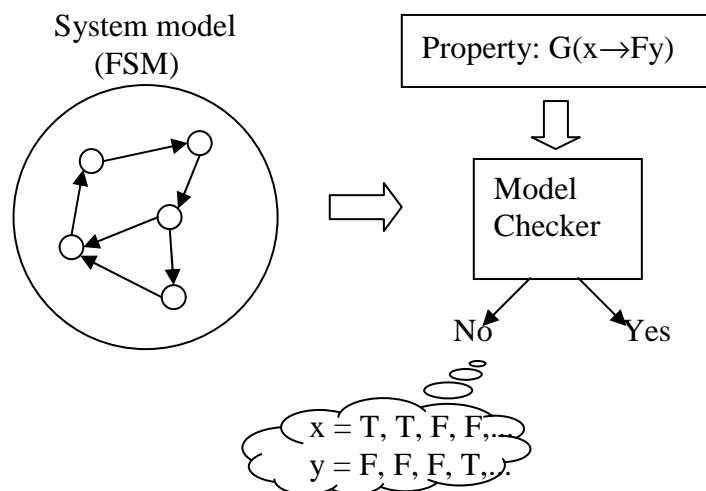


Figure 7: Model Checking

Examples of model checking tools are: Spin (spinroot.com/spin), SMV – Symbolic Model Verifier (www.cs.cmu.edu/~modelcheck/smv.html), HyTech – The Hybrid TECHnology Tool (embedded.eecs.berkeley.edu/research/hytech/), Kronos (www-verimag.imag.fr/TEMPORISE/kronos/), and UPPAAL (www.uppaal.com).

Temporal logic is a class of modal logic. It extends propositional logic to incorporate time operators, in the sense that formulas can evaluate to different truth values over time.

The use of temporal logic to model systems is straightforward. Each state corresponds to a possible state of the program and moving from one state on to the next corresponds to the execution of one step of the program. This representation of the system corresponds to a transition system in which temporal formulas can be tested.

There are different types of temporal logic that correspond to different views of time (branching vs. linear, discrete vs. continuous, past vs. future). With a linear time model (Figure 8a), each instant has only one successor. With branching time (Figure 8b), each instant can have one or more instants as successors. Examples of temporal logic formal languages are Linear Temporal Logic (LTL), and Computational Tree Logic (CTL) [7].

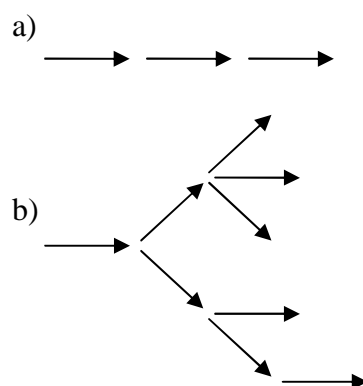


Figure 8: a) linear time; b) branching time.

In linear temporal logic (LTL) it is possible to express properties about one state, about a sequence of states (path), about the past, and about the future. The standard LTL set of operators is: \square (always in the future); \blacksquare (always in the past); \diamond (eventually in the future); \blacklozenge (eventually in the past); pUq (p until q); pSq (p since q); \circ (next time); and \bullet (previous time).

In branching-time logic the temporal operators quantify over the paths that are possible from a given state. It adds two operators to the linear set of operators which are E (for some path) and A (for all paths).

Temporal logic can be a powerful tool to express safety, liveness, and fairness properties about a system. Safety properties state that "something bad does never happen". Liveness and fairness properties state that "something good will eventually happen". Fairness can be seen as a special case of a liveness property and can be used to express, for instance, that a scheduler does never ignore a process.

While a violation of a safety property can be detected by a finite sequence of executions steps in the system, a violation of a liveness property may be detected only by an infinite execution of the system.

The main drawback of Model Checking has to do with the *state explosion* problem. The size of the finite state machine needed to specify a given system may be so huge that analyzing the entire state space becomes unpractical. There are some techniques available to diminish this problem:

- **Abstraction** – The model of the system is replaced by a simpler one in which irrelevant low level details are removed [198].
- **Bounding the state space** – the domains of the state variables are bound to a certain number of possible values [39].
- **Partial Order Reduction (POR)** – POR is based on the fact that the order in which concurrent transitions are executed does not influence the result, so just one of the possible execution sequences is considered and the other ones ignored [47].
- **Symbolic model checking** – use of symbols implicit representations of potentially infinite states and transitions that model the system [198].
- **Binary Decision Diagrams (BDD)** – A special case of symbolic model checking techniques where the implicit representation of the states and transitions is based on Boolean formulas [198].

There are some examples in the literature of applying model checking techniques to the verification of properties of interactive systems.

The *interactor* concept [60] is the basis for the specification of interactive systems used by Campos [41]. The *interactor* model was developed at York and applies general purpose specification languages to interactive systems.

Interactors describe the interactive system as a composition of independent entities. These unitary abstractions can be thought of as a software architectural abstraction similar to objects in object-oriented programming. Each interactor consists of an internal state which is reflected through a rendering relation (*rho*) onto some perceivable representation (*P*) (Figure 9).

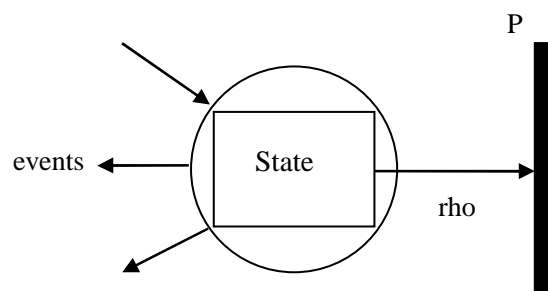


Figure 9: York Interactor

Campos, in [41], adapts (deontic) modal logic to specify interactors which are composed of state, behaviour, and rendering. Modal logic is a branch of logic in which sentences are quantified by modalities. He adds two deontic operands to reason about permission (*per*) and obligation (*obl*):

- $per(ac)$ means that action ac can happen next, and
- $obl(ac)$ means that action ac is obliged to happen in the future.

These operands work as quantifiers over the actions in a given state. An interactor's specification has attributes to model the state, and actions and axioms to model the behaviour. Attributes and actions can be prefixed with quantifier vis meaning that they are visibly perceivable.

The `i2smv` tool [41] translates interactor's specifications to the SMV input language. The properties are described by computation tree logic (CTL) formulas and checked automatically by the SMV model checker.

Paternó et al. [155] use ConcurTaskTrees (CTT) specifications to formalize task models structured in a hierarchical way where the lower levels refine the upper ones. These specifications use a semantic extension to LOTOS [27] in order to define temporal relations between tasks:

- $T1 \parallel T2$ – interleaving tasks;
- $T1 \parallel []$ – synchronized tasks;
- $T1 \gg T2$ – the end of task $T1$ enables task $T2$;
- $T1 [] \gg T2$ – task $T1$ enables task $T2$ and passes information on to it;
- $T1 [> T2$ – task $T2$ deactivates task $T1$;
- $T1 *$ – iteration of task $T1$;
- $T1(n)$ – finite iteration of task $T1$ (n steps);
- $[T1]$ – optional task.

CTT specifications are translated to LOTOS which can be accepted as input language by model checking tools, e.g., CADP [155].

Berstel [21] translates his VEG (Visual Event Grammar) formalism into the Promela language of the Spin model checker (spinroot.com/spin).

Abowd et al. [3] use Propositional Production Systems for specifying user interfaces. The specification is then translated into the SMV input language and analysed using CTL (Computation Tree Logic) formulas.

Dwyer et al. [61] describes several abstractions that can be used to reduce the state space of GUI models in order to make the application of model checking techniques feasible to verifying system requirements expressed as properties in computation tree logic. The model checker used is SMV. The problem with this technique is the lack of guidance in choosing which abstraction to use and the possibility of obtaining false results due to the abstraction. False results can be obtained when abstraction is weakly preserved for model checking i.e., when every property that holds on the concrete system also holds on the abstract one but properties that hold on the abstracted system may not hold on the concrete system. This may be due to three different reasons: a fault in the system; a mistake in the specification; an imprecision due to abstraction (e.g., excessive abstraction).

The main advantage of model checking is automation. Even so, it exhibits problems when applied to HCI mainly due to the construction of the model and the formalization of the properties to check. The model must be meaningful while abstracting from many low level details as possible. Properties are often difficult to formalize in modal logic. In addition, the kind of errors/faults that model checking is adapted to check are somehow related to sequences of states like the ones mentioned by Palanque in [152]: absence of deadlocks; predictability of a command; reinitiability; availability of a command; succession of commands; exclusion of commands. Other kinds of problems/errors may need a different technique to be detected.

Model checking techniques can also support the generation of test cases [9] as will be explained in the following chapter.

Theorem proving

Theorem proving is a well established formal verification technique applicable to verifying if a given implementation (I) conforms to its specification (S). This can be expressed mathematically either by an implication ($I \rightarrow S$) or by an equivalence relation ($I \equiv S$) between I and S as a theorem that has to be proved. Both specification (S) and implementation (I) are expressed in the same formal language. The formal proof is rigorously constructed as a sequence of steps based on a set of axioms and inference rules, like simplification, rewriting, and induction.

Unlike model checking, theorem proving can deal with infinite state spaces. Induction proof techniques apply to proving properties about infinite domains. The structure of the proof is split in two sub-proofs: the property is verified for the initial state ($n=1$) first, followed by the induction step, which verifies the property for every subsequent state ($n+1$).

There are also other proof techniques like deductive proof and proof by contradiction. A deductive proof is straightforward. Given that a certain hypothesis is true, a sequence of steps based on axioms and inference rules, is constructed in order to a conclusion. In a proof by contradiction, the starting point is the negation of the hypothesis ($\neg P$) to be proved. Then, a deductive proof is constructed. If the conclusion contradicts (\perp) the starting point then the original hypothesis (P) is proven to be true.

There has been research in applying theorem proving techniques to GUI verification [33,34,35,36]. Bellow we stress on experiments based on the abstract models PiE and RED-PiE (Figure 10).

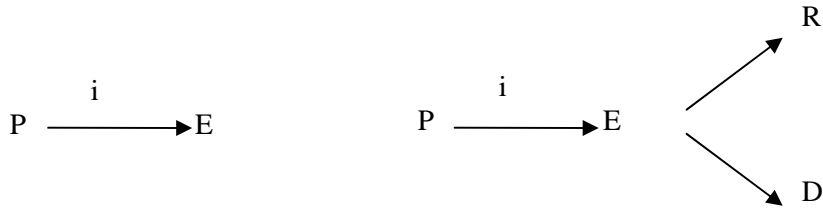


Figure 10: Models PiE and RED-PiE.

The PiE model and its successor RED-PiE [57] are abstract representations of user interfaces. The goal of abstract models is not to lead directly to an implementation of an application but rather to provide guidelines for future implementation attempts. An interactive system receives a sequence of input commands, P , that produce an effect, E , by applying the interpretation function, i , from P to E . With this abstract model, it is possible to express general properties of the systems like monotony or predictability

$$\forall p, q, r \in P : i(p) = i(q) \Rightarrow i(p.r) = i(q.r)$$

and reachability

$$\forall p, q \in P : (\exists r \in P : i(p.r) = i(q))$$

Essentially, the predictability property states that the effect produced by sending a command r to two systems with an equivalent current effect is the same. Reachability means that it is possible to reach any state from any other.

However, the PiE model has some limitations when one wants to express the effects produced in terms of rendering and output values. This limitation is overcome by the RED-PiE model by adding a projection function from the effect into representations of the results (R) and display (D). Other extensions to these models can be found in [57]. They are used to express exception conditions and undoing errors.

Bumbulies et al. [34] use HOL (Higher Order Logic Theorem Prover) to verify properties about user interface specifications.

Butterword et al. [35] provides proof of usability properties about interactive systems. They discover a problem with their system that could also be detected using model checking techniques. However, they claim that with model checking techniques they would not understand why the problem existed while with the construction of the proof they can understand it.

Butterworth and Cooke [37] use the Temporal Logic of Actions (TLA) to specify reactive and interactive systems. TLA is an extension of temporal logic in the sense that assertions about a single state, S , are generalized to assertions about

actions (assertions about pairs of states, $S \times S$). Actions specify allowed state transitions. An action A allows a transitions $s \rightarrow t$ from state s to t iff $\llbracket A \rrbracket (s, t)$ equals true. A state transition allowed by A is called an A -transition.

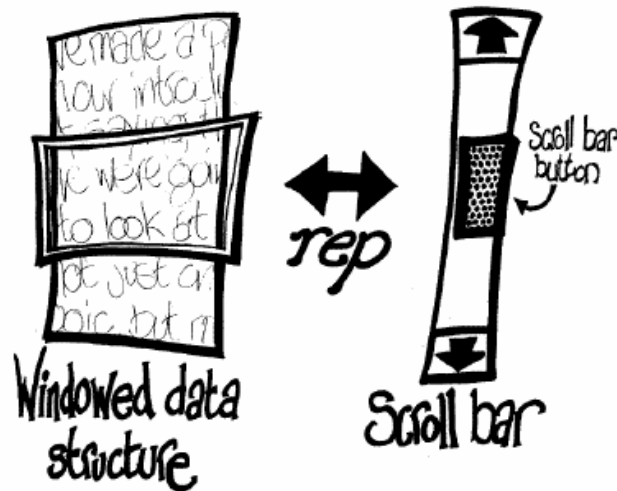


Figure 11: Relation between windowed data and scroll bar (taken from [37])

The case study of Butterword and Cooke [37] is the scroll bar interaction object and the related data structure within a window (Figure 11). At each instant, there is a relation, rep , between the position of the scroll bar button and the windowed data shown to the user. This can be expressed in temporal logic by

$$\text{StrictReq} \triangleq \Box(\text{rep}(\text{win}, \text{sbar}))$$

The user can perform two actions on the system, either altering the scroll bar (alterBar) or altering the window position (alterWindow).

$$\begin{aligned} \text{alterBar} \triangleq & \text{enable: true} \\ & \text{sbar} \neq \text{sbar}' \wedge \text{barAltered}' \end{aligned}$$

$$\begin{aligned} \text{alterWindow} \triangleq & \text{enable: true} \\ & \text{win} \neq \text{win}' \wedge \text{windowAltered}' \end{aligned}$$

Each action has an enabling condition identifying the system states where the action can occur. Variable names without dash refer to the start state before action, while variable names with dash refer to the end state of the action.

After describing the user actions, the behaviour of the user interacting with the system can be described by

$$\text{user} \triangleq \text{alterBar} \vee \text{alterWindow}$$

meaning that the user can perform either of these actions.

Similarly, the kernel actions reacting to the user actions are

$$\begin{aligned} \text{updateBar} &\triangleq \mathbf{enable: windowAltered} \\ &\quad \text{rep(win, sbar')} \wedge \neg \text{windowAltered}' \\ \\ \text{updateWindow} &\triangleq \mathbf{enable: barAltered} \\ &\quad \text{rep(win', sbar)} \wedge \neg \text{barAltered}' \end{aligned}$$

The overall kernel actions are

$$\text{kernel} \triangleq \text{updateBar} \vee \text{updateWindow}$$

The specification of the entire system is described by

$$\begin{aligned} \text{specInit} &\triangleq \text{rep(win, sbar)} \wedge \\ &\quad \neg \text{barAltered} \wedge \neg \text{windowAltered} \\ \\ \text{spec} &\triangleq \text{specInit} \wedge \square[\text{kernel}] \wedge \square\langle \text{user} \rangle \end{aligned}$$

where $\langle ac \rangle$ mean that action ac is permitted and $[ac]$ mean that action ac is obligatory.

Doherty [58] provides another example of applying theorem proving techniques to analyse properties about interactive systems. He starts with a VDM [158] specification which is translated into PVS (*pvs.csl.sri.com*) notation for that purpose.

Atif-Ameur [6] uses B to specify and prove properties about interactive systems.

Disadvantages of static analysis

The application of theorem proving techniques to an entire software system may involve so much work and resources that it may be unfeasible to apply them within software resource limits. Usually, this verification technique is performed on a small part of the entire system. The parts of the system to prove formally are selected either because they are critical parts of the system or because they are an implementation of a non trivial algorithm.

Theorem proving requires a formal model of the system at target. Proofs can be carried out with the help of a theorem prover. These tools can help in ensuring that the steps of the proof are correct but give no support for the conception of the proof.

With model checking it is possible to reach a higher degree of automation than with the other two static analysis techniques but some properties can be difficult to express in modal logic and therefore remain unverified.

Although static techniques can help in finding errors, there are errors that will be very difficult to detect with these techniques. This is the case of errors that rise only when the system is executed.

To verify GUIs with static analysis performed on specifications, one must build a formal model of the GUI on appropriate formal language. Some formal languages can be better adapted than others for that purpose. This will be the subject of Chapter III.

Another problem related with static analysis techniques is that they are far from current techniques used in industrial environments. Specification-based testing is a way to reduce this gap. It combines formal models with testing, leading to more systematic testing processes while automating most of the testing activities. This will be the subject of the following Chapter III.

2.5.3. Automated GUI testing approaches

Although many tools are available for developing GUI applications visually (e.g., user interface builders), they provide support neither for specifying or modelling GUIs including their functional behaviour at a higher abstraction level, nor for testing them in an effective way. Yet, testing GUIs represents a significant amount of the overall testing efforts at industrial level. To overcome this discrepancy, several kinds of testing tools have been developed. These tools vary from those that only support the automatic execution of test cases, to those that support test case execution, test case generation, and construction of the GUI model by a reverse engineering process.

Capture/Replay tools

In this kind of tools, test scripts are constructed by testers interacting with the GUI that records their actions, like mouse motions and keyboard inputs, in order to replay them later. These tools provide a record mode, in which every user action is saved in a test script, and a replay mode, in which test scripts are executed.

These tools often provide a scripting language that engineers can use for maintaining test scripts. They can, for instance, record a basic test script and modify it later manually to make it more effective. Test script execution is automatic and can be repeated several times. Test scripts can be constructed by interacting with the application under test (AUT) but capture/replay tools give no support for their design and coverage criteria analysis.

The problem with test scripts is their lack of structure which makes their maintenance difficult. This problem is softened by adoption of methodologies that entail more structure to the test scripts [105,107,199].

The **data-driven** automated testing methodology adds more structure to scripts by keeping separate input data and results from the testing procedure. This is accomplished by including variables in the test script that will get actual values from an external data source, file or database. This increases reusability, makes

the script more modular and easier to manage. New test cases can be constructed by adding new data to the data file without any changes to the original test script.

Keyword-driven testing increases reusability even further. The data file used in data-driven testing is expanded with an additional keyword describing what the test case does but not how it does it. The file constructed this way comprises the test script. It is more abstract than the one used in the data-driven testing approach. The detailed behaviour is described in an additional layer of scripts or library function. At run time, a test driver interprets the keyword and calls the corresponding detailed script/function in the function library.

Test scripts present a level of abstraction that does not impose knowledge about the scripting language used by the tool, so they can be developed by experts on the application domain who do not necessarily have knowledge about particular details of the tool.

These tools can vary as to the way they identify GUI objects. They can identify a GUI object by its position on screen or by capturing the object itself (object-aware). The first ones run into synchronization problems, for instance, clicking on buttons before they have appeared. Nevertheless, there are situations where there is not really an object on screen, just a bitmap, and an interaction based on screen point may be useful.

Advantages of Capture-Replay tools

These tools may have good observability capabilities, like optical character recognition (OCR) and image processing techniques, and may be helpful for regression testing and in other contexts such as: demonstrations; remote support; analysis of user behaviour; macro functionality; and educational scenarios. However, for testing purposes, they are still subject to severe critics [92].

Disadvantages of Capture-Replay tools

- Tools of this kind defer testing to the final phases of the software development process because they can only be used when the GUI, or part of it, is already available.
- If during test scripts construction, the tester makes a mistake, for instance, giving a wrong input field value, the test script must be constructed right from the beginning. The same happens if the tool gives an error. All that is being tested are things that already work [199].
- These tools don't provide any support to design test cases and to evaluate them according to coverage criteria.
- Changes to the implementation usually require the re-capturing of all affected test scripts.

- Scripts may contain hard-coded values, e.g., some of these tools store information at a low level of abstraction, capturing mouse positions. Representing the information at such a low level of abstraction makes these tools very dependent on the physical properties of the user interface. A small change on the layout of the user interface might invalidate all test cases.

Examples of these tools are WinRunner (www.mercury.com) and Rational Robot (www.ibm.com).

Random input testing

Random input testing is also referred to as stochastic testing or monkey testing [143]. The latter designation is used to give the idea of "someone" without a brain, or without knowing what he's doing, seated in front of a computer and interacting randomly with the keyboard or mouse.

Microsoft reported that 10-20% of the bugs in their software projects are found by monkey test tool [142].

Monkeys can vary in smartness. Ignorant or dumb monkeys don't know anything about the current state of the software application nor about legal or illegal input values. They generate test cases randomly and ignore any unexceptional outputs of the system. The main problem with such monkeys is that they cannot even recognize a software error, which is not very useful.

The goal of dumb monkeys is to crash the system under test. This category of monkeys is not well suited to find defects related to incorrect behaviour, but it is the most cost-effective for finding defects that crash the system. Rational's TestFactory detects application crashes without user intervention using dumb monkey method.

There is other another kind of semi-smart monkeys which allows them to recognize a bug when they see one.

Smart monkeys have some knowledge about the application they are testing. They have knowledge about states and know the legal steps to move forward in each state. They can also check if the reached state is the one expected.

Smart monkeys are more costly to develop because they need a model or state table. Dumb monkeys are easier to construct. For being "stupid", without particular knowledge about applications, dumb monkeys can be used to test a wide range of application types and they are independent of screen changes.

Smart monkeys can find more bugs but are more expensive to develop. They can be useful for load and stress testing, particularly at system level, for instance, using several monkeys interacting simultaneously with a multi-user software system.

One of the problems with random input testing is their weak code coverage. In every interaction, the tool has to choose an input value among the valid values in the domain. For a domain ranging from 1 to 100, each value has a 1/100

probability of being chosen. If somewhere in the code there is a if branch like "if ($n \neq 50$) ... else...", the else branch has 1/100 probability of being exercised.

Random human testing is performed by real users playing with some software which is made available by their owners for a couple of weeks, with the goal of catching errors. Although some errors can be found by this approach, it is rather arbitrary and does not provide reliable coverage criteria [52].

Unit testing frameworks

Another possible approach is to program the test cases. Frameworks like JUnit (www.junit.org) and NUnit (www.nunit.org) are of great help in organizing and executing test cases, particularly for API testing, but not in generating those tests. The test cases have to be constructed/programmed manually which gives a high level of flexibility to the tester.

A popular approach in GUI testing is to code the test cases "manually" in which unit testing frameworks can be helpful. Even so, the tester has a hard work to adequately test the GUI behaviour. In the case of GUI testing, many bugs can only be uncovered through particular sequences of actions, which might rise in the daily use of the GUI. Unit tests, however, are usually a few hand-written sequences of actions, which tend to be very short. Thus, there is a high probability to miss these kinds of errors.

With these tools, GUI testing is treated like API testing. The tester has to write code to simulate the user interacting with the GUI under test, while observing the output, and to check if the result obtained is the one expected. Even using a GUI library, like, for instance, Abbot (abbot.sourceforge.net/), or Jemmy (jemmy.netbeans.org) that provides methods to simulate user actions and observe the state of interaction objects, GUI testing using these tools requires a lot of extra programming effort to be effective.

Model-based testing

The model-based tools discussed in section 2.4.2 were generically concerned with automatic generation of user interfaces. Unless there is no trust on the code generators, it is expected that the set of user interfaces that can be generated by them is correct, so they do not provide support for the testing phase of the UI development process. However, these tools present some limitations as far as the type of user interfaces they are able to construct is concerned.

Model-based testing tools focus on the test automation process. They are used to test the conformity between an implementation and its model. A high level of test automation can be achieved with model-based testing tools given the fact that test case generation, test case execution, and the comparison of the expected results with actual results can all be automated.

A model-based testing process starts with the construction of the model of the application under test (AUT) (Figure 12). The model is then used as input to

generate test cases according to given coverage criteria. Test cases are executed over the AUT (application under test) and the results obtained and states reached are compared with the expected results and expected states described in the model.

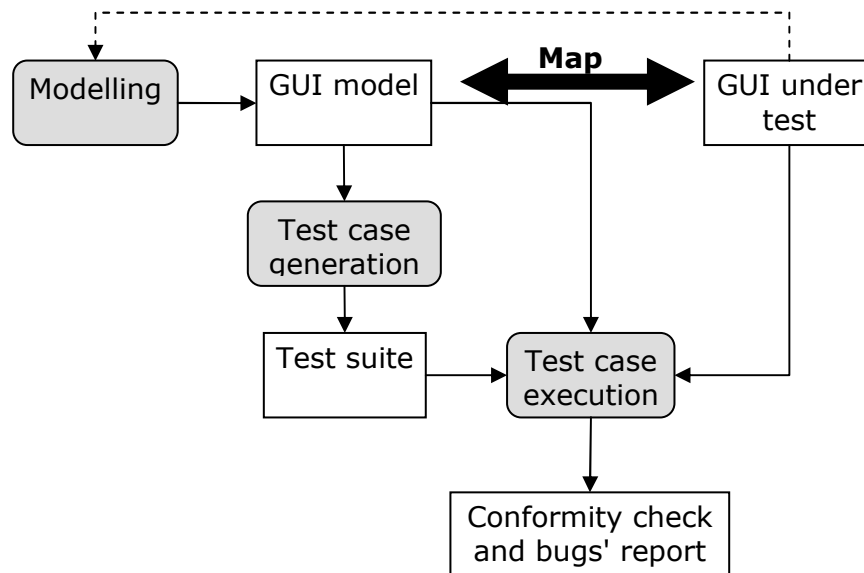


Figure 12: Model-based testing process.

The kind of model notations used can range from textual to graphical notations, can either be or not be executable, and can vary in their degree of formalization. Depending on the kind of the notation, different test case generation algorithms and different coverage criteria can be used.

Conformity between actual and expected states can be checked after each execution step in a "lock-step" mode, or at the end of the execution in which case intermediate results must be saved for comparison.

The model captures the requirements of the AUT. When they suffer changes, the model changes and the application must be tested to check if the new requirements are fulfilled. Some model-based testing tools provide support for regression testing by calculating the subset of the test suit that is affected by the requirements modification and calculating the modifications that it must suffer in order to test the new/changed functionality [123,177].

There are several examples of model-based testing tools for testing software applications through their API. Examples of these tools are: TGV (www-verimag.imag.fr/~async/TGV), AGEDIS [89] (www.agedis.de), Autofocus (autofocus.informatik.tu-muenchen.de), QuickCheck [46] (www.md.chalmers.se/~rjmh/QuickCheck), and Spec Explorer (research.microsoft.com/SpecExplorer). The literature, however, is scarce in model-based testing tools that test software applications through their GUI [18,125,147]. Unfortunately, only one of them is freely available. Nevertheless,

the characteristics of these tools are described next and their main pros and cons are pointed out.

Visual Test Development Environment

The work of Ostrand [147] combines capture/replay tools with model-based testing concepts. The capture functionality is used to construct a preliminary model of the GUI under test, which is converted automatically into a visual notation model for generalization. This generalization is obtained based on two main concepts: path variations and data variations. The former is used to model alternative sequences of actions and iterations. The latter replaces fixed values with variables that can take different values within a defined domain. The test scenarios constructed using these concepts may represent several test scripts. An independent test generation engine builds the set of test scripts represented by scenarios and translates them into the scripting language used by the capture/replay component of the test environment for being replayed and tested over the GUI under test.

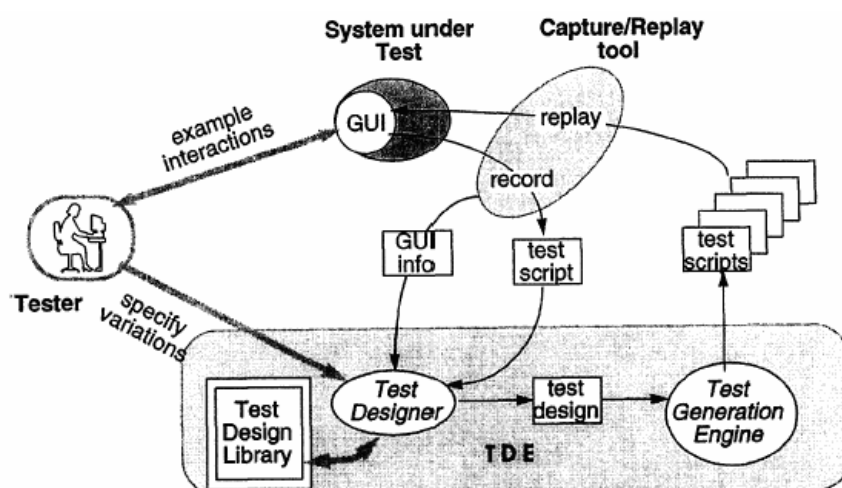


Figure 13: Visual test development environment (taken from [147])

Although this test development environment (Figure 13) overcomes some of the capture/replay problems, it does not generate test cases automatically. This is still done manually. The only help is the capture feature that enables the construction of a preliminary test case.

IDATG – Integrating Design and Automated Test Case Generation

IDATG is an integrated design and automated test case generation environment [18]. Test cases are generated from a model with three levels of abstraction: a requirements specification, a task flow model, and a low-level specification. The requirements specification is described in plain text. Then a task model is drawn as a task flow graph which describes typical usage scenarios. The third level

captures information about the real GUI objects and details about each task step like the expected result. Each task step is mapped to a real GUI object with a point and click.

This system does not require a complete specification of the application to generate test cases. They can be generated from part of the model covering all edges in the task flow graphs and can be regenerated when the specification changes. The test cases generated are stored in XML format and are displayed as flow diagrams which can be edited graphically. The XML files can be converted into other formats, like, for instance, WinRunner scripts, for replay.

The task flow graph can be structured into a hierarchy. Sub tasks can be reused which reduces the effort for test maintenance.

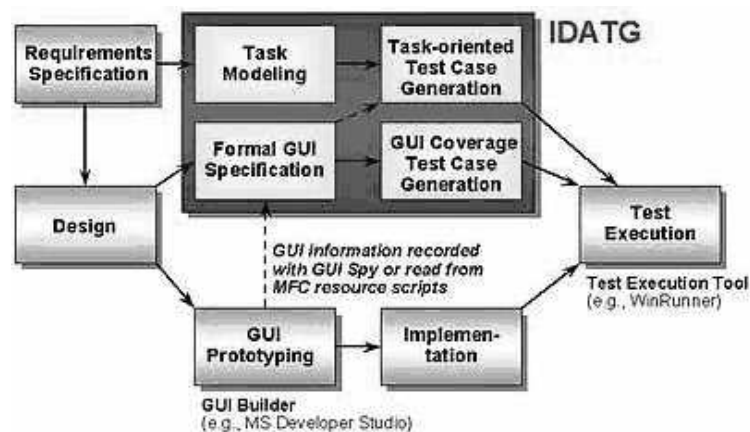


Figure 14: IDATG test process (taken from www.qualityscope.com/28.html)

The advantage of this tool (Figure 14), when compared with the previous one, is the support for test case generation. Even so, this environment does not include features for test case execution, which requires a change of environment for that purpose, for instance, using WinRunner. It is also not clear whether the tool integrates test input data or whether it leaves that for the test running tool used [23].

GUITAR – A GUI Testing framework

GUITAR (Figure 15) is another example of a GUI model-based testing tool. The GUI model from which test cases are generated is an event-flow graph and an integration tree [129]. The first one captures the flow of events within a component. It represents all possible interactions among events in a GUI component. The second identifies interactions between components.

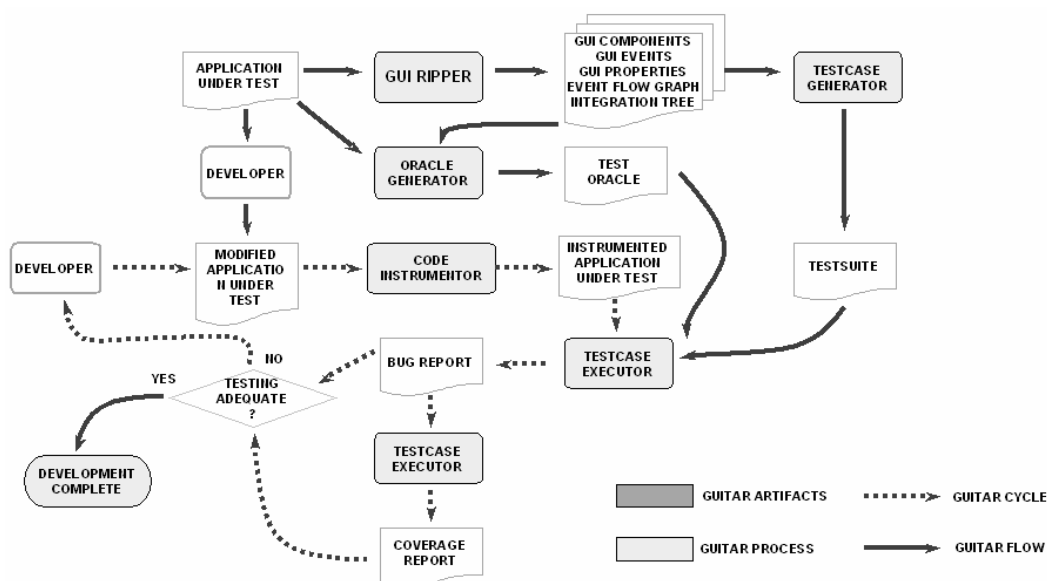


Figure 15: GUITAR process (taken from www.cs.umd.edu/~atif/GUITARWeb/guitar_process.htm)

The event-flow graph for a GUI component has a set of vertices, V , and a set of directed edges between vertices. An edge from v_1 to v_2 means that the event v_2 may occur immediately after v_1 . This usually gives rise to a strongly connected graph as illustrated in Figure 16.

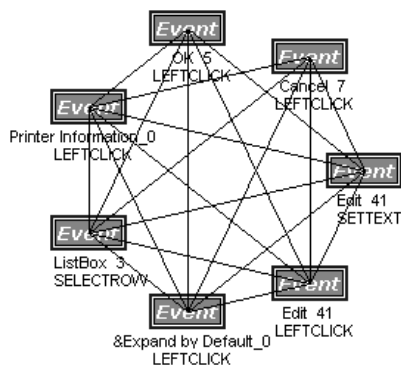


Figure 16: Event-Flow Graph for WordPad --> Connect to Printer (taken from www.cs.umd.edu/~atif/GUITARWeb)

The integration tree describes how GUI components are put together to form a complete GUI. The model has a set of components represented as nodes and a set of directed edges. An edge from c_1 to c_2 means the c_1 invokes c_2 (Figure 17).

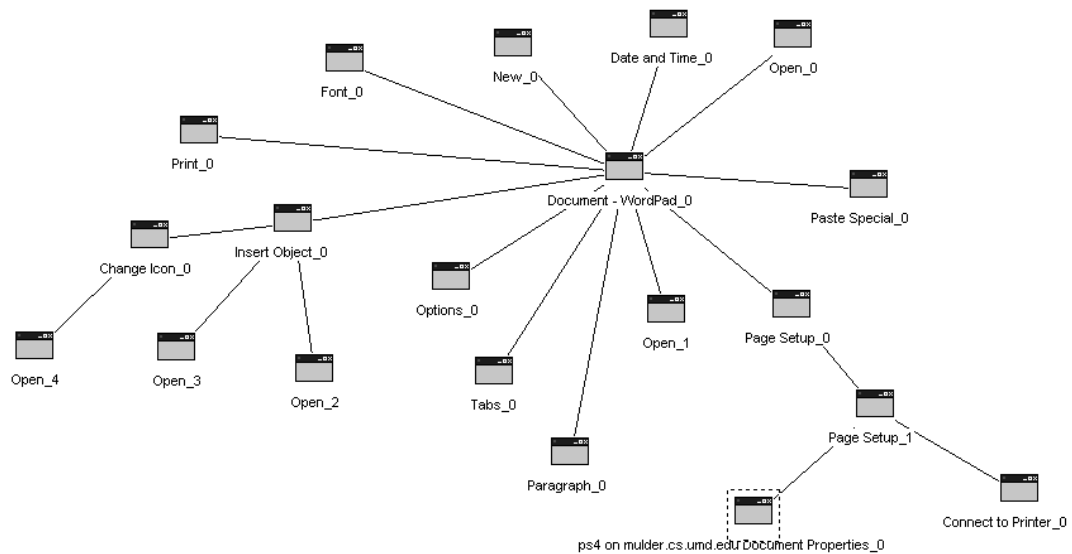


Figure 17: Integration Tree for WordPad (taken from www.cs.umd.edu/~atif/GUITARWeb)

Memon [129] defines intra- and inter-component coverage criteria based on these two models and planning techniques from Artificial Intelligence to automatically generate GUI test cases. He also proposes a solution for regression testing. The original GUI test suite is partitioned into valid and invalid test cases. Invalid test cases are repaired for reuse.

Memon claims that constructing a GUI model that can be used for test case generation is difficult, so he develops an approach to reverse engineer a model directly from an executable GUI [124]. This model represents the GUI's structure as a GUI forest, and its execution behaviour as event-flow graph, and an integration tree. The GUI ripping process opens automatically all the windows of the GUI under test and extracts their widgets, properties, and values.

A problem may occur with the GUI ripping process when a subset of the application functionality is protected by a key. In this case, the tool cannot guess the key and consequently cannot construct the model for the protected part of the GUI application. One solution to this problem could be to mix exploration with interaction. In this case, the tester could manually drive the application until some particularly state and automatically explore the application thereupon. However, this feature is not supported by the tool.

Moreover, to reverse engineer a model from an untested GUI and then use that model to test the same GUI seems useless. The model will describe the behaviour of the GUI as it is, so it will also incorporate GUI errors! In his academic experiences, Memon uses this ripping process on a correct GUI and then tests an incorrect GUI based on the model extracted from the first one.

Nevertheless, GUI ripping can be useful. It can be used to extract a preliminary model of the structure of the GUI and part of its behaviour and then complete it manually with more behaviour and details. Also, some errors can be detected if the algorithm used to construct the test cases from the model uses a traversal algorithm different from the one used by the exploration. This process drives the application through non-explored paths, which can find errors.

A problem with Memon's approach is lack of explanation about the structure and the meaning of the models extracted by the ripping tool. So, it is complicated to refine them. In [127], a model constructed by operators with pre-conditions and effects is mentioned, but it seems there is no relation between these models and the one extracted by the ripping tool. In particular, it is not clear how the models automatically constructed deal with message boxes, neither how to know which menu option opens a dialog, nor how to describe that an interactive control can enable another.

In summary, the current versions of the tool developed by Memon seem not yet sufficiently mature for being used outside academic environments.

2.6. Conclusions

Tools used in industry to build GUIs lack support for modelling, verification, and maintenance phases of the GUI development process.

Model-based tools cut across the GUI development process phases but they are still uncommon in industrial environments. In addition, the first generation of model-based tools focused on GUI automatic generation but was limited as far as the type of GUI they were able to generate is concerned. Some tools of the second generation were able to evaluate the quality of the models and supported user centred design. Even so, they impose a complete divorce with the current practices for GUI development. Developers have to learn new modelling languages and new practices, which explains why these tools haven't gained adepts in industrial environments.

GUI testing can be performed manually or with the help of tools. Manual tests are good for exploratory or initial tests, and for those tests performed by the end user. They can find more bugs per test cases executed when performed by experts. Bugs found can provide hints to find other bugs. Manual tests are particularly well suited for usability tests performed by real users. One of the problems with some approaches for manual testing is their lack in systematization. This problem can be reduced by using checklists of standard tests and application of specific tests. Even so, manual tests require too much effort while providing weak coverage criteria. Test cases are difficult to reproduce and the success of test case execution (number of errors found) is very dependent on the capabilities of the tester. In addition, experienced test specialists are hard to find.

Automated testing is faster than the manual one. The increase of execution speed makes it possible to run more tests in less time, more often, and covering more functionality. One example is the testing of a strange sequence of events where bugs can be found and that are usually not covered by manual tests. In addition, automated tests may be reused and repeated every time a bug is found. Although automated tests are more efficient in terms of time needed and better use of resources, they may be a source of false sense of security. It is known that "program testing can be used to show the presence of bugs, but never to show their absence" [56].

Testing approaches can vary with respect to their support for the testing phases. Some of them do not provide automatic support for any of the testing phases and others provide automatic support for every test phase. This is the case of some random testing tools. In between, one finds several degrees of automation: tools that only provide support for test execution (unit testing frameworks); tools that also assist the construction of test cases (capture/replay tools); and tools that provide automatic support for test case generation and execution (model-based testing tools).

Unit testing frameworks only provide support for executing the test cases which must be programmed manually by the testers. In the case of GUI testing, a manual test case construction can leave several parts of the application untested.

Capture/replay tools also do not provide support for designing test cases but they provide a capture functionality that allows the construction of the test cases by interacting with the GUI under test (assisted test case construction). The user actions are saved in a test script that can be made more generic by programming and replayed later. Whenever the tester makes a mistake or the software application gives an error, the test case must be redone from beginning. Maintenance of test cases remains a huge problem. This leads to a main criticism to these tools, which points out that they can only be used when the software application is working correctly. So, what is it being tested for? For GUI testing, Capture/replay tools are not sufficient.

Model-based testing tools lead to a higher degree of automation. In addition to the automatically generation of test cases, these tools also provide support for automatically executing those tests. This requires a model of the application under test. More time is spent with this activity when compared with the other automated approaches but no time is spent on the generation of test cases since they are calculated automatically. Some of these tools reduce the time spent in constructing the model by reverse engineering existing applications. One of the problems of these tools is test case explosion. Test case generation has to be controlled appropriately to generate test cases of manageable size.

Random-input tools can vary from those that do not require a model of the GUI to those that require a state table to generate test cases. The first kind of tools is the one that requires the less effort for testing GUIs. However, these tools cannot identify a bug so they are only adapted to find bugs that make the system to crash.

Although current testing approaches are still not satisfactory, they have points in favour which deserve to be noted:

- Separation of logical names from physical properties of GUI objects can be found in some capture/replay tools but could also be used in other approaches. This is a positive aspect since both levels, logical and physical, remain independent which makes it possible to change one of the levels without changing the other.
- GUI test libraries can reduce the time spent in programming the test cases manually like what is done with unit testing frameworks. They can also be reused by model-based testing tools.
- Recording techniques are available in capture/replay tools. This capability could also be useful to tell how high-level user actions described in a model are mapped to concrete actions in the application.
- Manual tests can be combined with automatic tests to drive the application to a specific state from which other kinds of tests could be run. This could be useful in regression testing when some functionality remains unchanged while others are modified.

As will be seen in the remainder of this dissertation, some of these points in favour will be taken into account by the testing approach proposed in this dissertation, while others will be left for future work.

Chapter III

Specification-based GUI Testing

This chapter starts by presenting the main challenges of Graphical User Interface (GUI) testing either when compared to Application Programming Interface (API) testing or when one wishes to automate the test process. Then it presents a survey on the work related with GUI specification-based testing. It starts by describing different ways of modelling GUIs using different kinds of formal specification languages and then presents different techniques used to generate test cases from different formal specifications. At the end, different strategies of performing automatically verification of the test results (conformity check) influenced by the kind or style of the specification used are presented.

The goal of specification-based testing is to check dynamically if an implementation of a software system conforms to the specification (or model) of that system. The specification captures the requirements and the conformity tests check if those requirements are fulfilled by the implementation. Given an executable implementation and a specification of a software system, the generic activities involved in specification-based testing are test case generation (from the specification), test case execution, and comparison of the actual results obtained from the implementation with the expected results derived from the specification (which plays the role of a test oracle). Test inputs and expected results are generated from the specification.

Formal specifications (or models), in particular the executable ones, can be used to automate the testing of software applications. In fact, an executable formal

specification can be used both as a test oracle and as a basis for the automatic generation of test cases.

Although it is possible to achieve high levels of automation with specification-based testing, it may be difficult to automate the entire process. In particular, the specification of the system under test is most of the times constructed manually. However, there are techniques to reverse engineer legacy systems constructing a preliminary model in which details can be added to perform specification-based testing. These techniques reduce the effort required for constructing the specification of system under test.

The same applies to Graphical User Interface specification-based testing, but in this case, the techniques should be specialized to deal with its particular characteristics.

3.1. GUI test automation challenges

With GUI test automation it is possible to run more tests, more often, and explore uncommon sequences of events where sometimes errors can be found and that would be difficult to cover with manual tests. However, testing of graphical user interfaces poses well-known challenges either when compared to API testing or when one wishes to automate the test process.

Time

- GUIs respond slower than APIs. They have a time overhead due to the rendering of the output to the user.

Test case explosion

- "Many ways in": GUIs may provide multiple ways to achieve the same goal – e.g., mouse, keyboard, and different navigation paths to reach the same state. Sometimes errors can only be detected in uncommon sequences of events that are usually not covered by manual tests.
- GUIs are very different from command-based interfaces. GUIs neither impose a particular order for performing the available tasks, nor a fixed order for providing the inputs. The number of different permutations of inputs and events increase the input space size and makes even worse the state explosion problem and consequently the test case explosion problem.

Controllability

- Controlling GUI actions can be difficult and involve several small steps, for instance, drag and drop is split into three steps: press the

mouse button in the origin point; drag the mouse to the destination point; release the mouse button.

- In automated testing, find the proper way to simulate the inputs from the user (mouse, keyboard and other higher-level events that are generated by the user) may be difficult.

Observability

- How to check the outputs to the user without excessive sensitivity to formatting and rendering details? Sometimes, to observe GUI visible state, image processing techniques like character recognition may be needed.
- Observe GUI state may be tricky or almost impossible. For instance, to observe a huge text through a small window a scroll bar is needed. If there is no scroll bar it may be impossible to observe the entire text.

Testing techniques

- "Many ways out": Graphical characteristics make it more difficult to determine the expected results of an operation (colours, fonts size, ...) [115].
- GUIs have unique properties and errors that may require different testing techniques to find all of them – e.g., display properties, navigation properties, and usability properties.

Documentation

- The lack of appropriate documentation makes more difficult the construction of GUI models as a basis for test automation. GUIs are constructed by reusing interactive components. The documentation supplied with those interactive components is usually scarce and not rigorous enough for more advanced uses, such as advanced customization and thorough testing. This usually leads to a "trial-and-error" style of application programming and poor application quality, and also complicates the design of test cases. For example, from the documentation, it is difficult to know precisely:
 - when are events signalled and by what order;
 - what is the internal state of a component when it signals an event;
 - what is safe for an event handler to do;
 - what interactions exist among events.

Some of the issues and challenges described in this section will be addressed by our testing approach and discussed in the next sections.

This chapter will describe different approaches to specify formally GUIs, then how to generate test cases from those models, and at the end different ways of

checking conformity automatically between a specification and an implementation.

3.2. Formal GUI Specification

Formal methods are becoming more accepted in the development of software systems but their applicability to the specification of user interfaces is not so common. The user interface model is most of the times given as a prototype or through other non-formal techniques. This can give rise to ambiguities and misunderstandings that can lead to different interpretations among the stakeholders and to the construction of a final useless UI. A formal specification can help finding inconsistencies and problems before the implementation begins which can result in time and resources savings.

Over the years, a number of formal models have been used for specifying user interfaces. The kind of specification used depends on the characteristics of the target user interface and the characteristics considered relevant from the modeller perspective. Also, the set of tools available to support the formal method can be a relevant point for the decision. Like other systems, user interfaces can be sequential or concurrent, synchronous or asynchronous, and timed, timeless or real time (see section 2.1).

The command-based interfaces were the subject of the first attempts to apply formal methods to user interfaces development. The synchronous and sequential characteristics of these interfaces allow the application of formal languages like context-free grammars and state transition diagrams. The specification of these interfaces can be constructed as an enumeration of the available commands and the definition of its syntax.

GUIs are very different from command-based interfaces. They present a much more complex structure and more complex event-driven behaviour.

We will go through each formal method describing, based on the literature, how it can be used to specify user interfaces and more concretely GUIs, and which techniques are available to generate test cases automatically from the specification. At the end, different ways of checking the conformity between the specification and the implementation are presented.

3.2.1. Grammars

A formal grammar can define precisely a formal language by a set of rules which can be used to generate all possible strings in the language by rewriting steps from a starting symbol (generative grammar), or to analyse if an input string is a member of the language (analytic grammar).

A generative grammar can be defined formally by a quad-tuple (N, Σ, P, S) , where,

- N is a finite set of non-terminals;
- Σ is a finite set of terminal symbols, disjoint from N ;
- P is a finite set of production rules;
- S is the start symbol (a non-terminal from N).

Generically, a production rule is of the form $v \rightarrow w$, where v and w are strings of terminals and non-terminals, formally $v, w \in (\Sigma \cup N)^*$. Non-terminals are symbols representing language constructs. When the left-hand side of all production rules of a grammar is a string formed only by a single non-terminal symbol, that grammar is called Context-Free Grammar (CFG).

Backus-Naur Form (BNF) is an example of a notation used to describe Context-Free Grammars. Each rule is composed of a more abstract non-terminal at the left-hand side that is defined ($:=$) as a more specific term at the right-hand side. Alternatives, succession and options are indicated by an "or" ($|$), an "and" ($+$), and enclosed brackets ($[...]$) respectively. Several of the grammars that will be described next are based on the BNF notation.

UI modelling with grammars

Context-Free Grammars were fairly common for command-based interfaces. They specify textual commands or expressions that a program would understand. The terminals in the grammar are input tokens generated by the presentation component. These tokens represent the user's actions. The terminals are combined by the productions in the grammar to form higher level structures called non-terminals. The collection of productions in a grammar defines the language employed by the user in his interaction with the computer.

GUIs present a more complex structure than command-based interfaces. Even so, grammars can also be used to specify form-based interfaces where typically there are several possible tasks available for the user at each time. To take that fact into account, grammars can define different productions rules with alternative sequences of the same symbols at the right-hand side. Another possibility is to use $A|B$ notation at the right side of the production rule to indicate that the input order of A and B is irrelevant.

Hanau et al., in [86], use BNF to describe the dialog control of an interactive picture drawing system. They also developed a set of prototyping and simulating tools that are capable of generating snapshots of the system display for different selected stages of the user/system dialog.

The Reisner's Action Language, presented in [165], extends Bachus-Naur Form (BNF) to include cognitive actions, written in angle brackets ($\langle \rangle$), and physical observable actions, written in capital characters. Every action is associated with a grammar rule. Whenever the rule applies to the input language stream (received so far) the associated action occurs.

Shneiderman's multiparty grammars are another example of grammar-based specifications (referred in [43]). They are an extension of the Reisner's Action Language (the "psychological" BNF). The evolution is to add expressiveness for representing the interaction decomposition regarding both elements involved in human-computer interaction. They divide non-terminals into user-input, computer, and mixed. Multiparty grammars allow direct association of interface feedback to user inputs but they are not well adapted to model the variety of user actions found in a direct manipulation interface.

Task-Action Grammar's (TAG) [78] goal is to describe the system tasks in the closest way possible to the meaning they may have for the user so he can learn easily how to use the system. TAG is a feature grammar. It does consider neither the screen, nor the meaning of the features. The tasks are described by their structure, which was not possible in the original versions of BNF. For example, to represent character movements of a cursor, BNF representations would need four rules (up, down, left, and right) whereas TAG would need only one by setting the value of the parameter accordingly:

```
Move_Cursor[Direction] ::= Cursor_Key[Direction]
```

TAG specifications can be used as input to measure the consistency of the user interfaces. The description of the task structures allow the measurement of the degree to which the methods used to achieve goals share the same structure, and is one of the factors that influence the learnability of the system.

Scott and Yap, in [169], extend Context-Free Grammars with two concepts to deal with multi-threaded dialogs: fork productions and context attributes. The former is used to cope with concurrency and interleaved conversations. It is implemented by two new operators between productions: "parallel and" (&& – used when the order of input is not important), and "parallel or" (|| – used when the production is complete when one of the sub-productions succeeds). The latter is used to cope with multi-window application. Two attributes are added to all tokens: value is used for the type of the token; and context is used to distinguish inputs of the same type but from a different source i.e., originated by a different window.

Iizuka et al., in [98], use Constraint Multiset Grammars (CMG) with actions to describe a simple drawing editor. Chok and Marriott, in [44], use CMG description for automatic construction of user interfaces. Another example of automatic generation of a user interface from a grammar notation can be found in [146]. In this case, Olsen and Dempsey describe a system called SYNGRAPH (SYNTAX directed GRAPHics) that uses an extended version of BNF to generate automatically GUIs.

More recently, Campi developed the VEG (Visual Event Grammar) notation and a tool for supporting the formal specification, verification, design and implementation of graphical user interfaces [40]. The VEG specification abstracts away presentation aspects of the GUI. It is only concerned with the description of the dialog control of the GUIs by means of modular, communicating grammars

with a visual notation supported by a visual editor called Dialog Control Editor (DCE).

UI analysis with grammars

Grammars provide a way to describe formally the aspects of a system in a level of abstraction in which it is possible to reason about general properties without concerns about implementation particular details. The formal description of the system can be verified for completeness and consistency. Also, grammar-based specifications of user interactions were commonly used for usability evaluation [83] based on cognitive and psychological theories: task environment analysis; analysis of user knowledge; user performance prediction; representation for design.

Task environment analysis models the tasks in the real world environment and the related task provided by a computer system. The complexity of the rules mapping the two environments determines the difficulty of transferring knowledge between them or the knowledge necessary for task reformulation, e.g., External Task – Internal Task mapping (ETIT) [138].

Analysis of user knowledge aims to give an indication of how much the user has to learn in order to perform his tasks through actions required to operate a new system. The complexity of the formal rules describing the interaction language between man and computer (or tasks and actions) is used as such indicator. The complexity is measured by counting the number of rules, the depth of the derivation of rules and the number of exceptional rules [85]. Reisner's Action Language, Shneiderman's multiparty grammars, and Task-Action grammar (TAG) are examples of specifications that can be used for that purpose.

In [95], Howes et al. show how consistency evaluators, written in Prolog, can be used to predict the learnability of a system described by a TAG specification and Brown, in [32], presents a method to identify learnability problems based on a TAG specification of an interface.

User performance prediction models aim to predict user performance aspects at an earlier stage in the development process. Examples are GOMS (Goals, Operators, Methods and Selection Rules) firstly developed by Card, Moran and Newell, and CCT (Cognitive Complexity Theory).

A GOMS model contains goals and sub-goals, methods and operators, and selection rules. To achieve one goal, the corresponding sub-goals must be carried out. Operators or actions are structured into sequences, named methods, which accomplish a goal. There can be more than one method for each goal. Selection rules are used to select one of those methods. For example, to delete more than eight characters two methods are possible [102]: firstly select those characters and then delete them (*mark-and-delete* method); or delete one character at each time (*delete-characters* method).

GOMS techniques are used to predict the execution time needed to achieve one goal, the sequence of operators or actions to achieve that goal, and the time

needed to learn the methods. There are different kinds of GOMS models for user performance prediction [102]: Keystroke-Level model (KLM), Card, Moran, & Newell GOMS (CMN-GOMS), Natural GOMS language (NGOMSL), Cognitive-Perceptual-Motor GOMS (CPM-GOMS).

Cognitive Complexity Theory (CCT) models the complexity of the system from the user perspective to predict the usability of that system. It uses two different models: one to describe how the user understands one task, and the other to describe the system task from a technical point of view. The relation between both can be used for many purposes such as modelling errors [78].

The models used in "**representation for design**" describe the knowledge a user must have about it in order to be able to perform tasks. Examples of these models are ETAG (Extended Task Action Grammar) [84], and CLG (Command Language Grammar) [138].

Disadvantages

Grammar based techniques are difficult to use for describing more modern windowed and mouse driven interfaces, like direct manipulation interfaces, where rigid sequences of required actions are almost always undesirable.

Grammars do not scale well, are not good at representing concurrency, and do not support an explicit representation of state. In addition, grammars are difficult to write and read.

Another problem with grammars is that the order in which production rules are used depends on the kind of algorithm used by the parser. In the case of a bottom-up parse, a production is used when all symbols on its right-hand side have been recognized. In the case of a top-down parse, a production is used when the first terminal that could be generated by the right-hand side is encountered.

That's why the use of grammars to model user interfaces tends to be rather scarce recently.

3.2.2. Finite state machines

Finite State Machines (FSMs) (or Finite State Automata) are very widely used in modelling system behaviour. The model is composed of states, actions and transitions and can be represented using a state diagram. There are different kinds of state machines: Deterministic Finite State Automaton (DFA), where for each pair of state and input symbol there is a deterministic next state, and Nondeterministic Finite State Automaton (NFA), where there may be several possible next states for each pair of state and input symbol. In addition, FSMs can have outputs determined only by the current state, in which case they are called Moore machine, or they can have outputs determined by the current state and the inputs, in which case they are called Mealy machines.

UI modelling with state machines

State machines are well suited to model reactive systems. GUIs are reactive systems in the sense that they respond/react to user actions. Finite State Machines can be used to model interactive systems. Typically, when an interactive system is modelled by a deterministic Mealy Finite State Machine, it is expressed by a sextuple $\langle S, X, Y, \delta, \lambda, s_0 \rangle$, where

- S is a finite set of possible states;
- X is a finite set of inputs;
- Y is a finite set of outputs;
- δ is the state transition function $S \times X \rightarrow S$;
- λ is the output function $S \times X \rightarrow Y$; and
- $s_0 \in S$ is the initial state.

Each transition is triggered by a user input. In response to the user input, the system performs an action that can change the state and produces outputs to the user.

Parnas was the first using State Transition Diagrams to specify user interfaces [153].

One of the problems about modelling interactive systems with state machines is the state explosion problem. This is due to the huge number of possible user actions and input values. There are several extensions to finite state machines in order to deal with that problem. In general, these approaches allow simplifying the transition state diagram and focus the attention on more relevant aspects of the state. One of those examples is the Variable Finite State Machine (VFMS) [170]. VFMSs are FSMs with an added condition associated to each transition. The transition can be expressed by:

```
name <state> <input> <next state> <output>
```

VFMS allows modelling systems with fewer states than an equivalent FSM. VFMS augments FSM with global variables which can assume a finite number of values. These global state variables are used to build Boolean expressions that are associated with transitions:

```
@req <variable> <value_required>
```

This expression or pre-condition determines when the related transition can occur and is written as a prefix of the transition name. Transitions can have also associated post-conditions to update the value of the global variables:

```
@set <variable> <new_value>
```

In [170], it is possible to find an example of a user interface modelled with a VFSM and modelled with a correspondent FSM. The former model has 20 states while the latter requires 580 states to model the same interface.

Andrews, in [11], uses HFSMs (Hierarchical Finite State Machines) to model Web applications and uses constraints to reduce the set of input values and to help solving the state explosion problem.

Harel, in [87], describes the semantics of the Statechart formalism and how it can be used to describe reactive systems like a Multi-Alarm watch. Statecharts extend state-transition diagrams with hierarchy, concurrency and communication. These extensions allow the description of complex behaviour in a compact manner at different levels of abstraction which makes specifications manageable and comprehensive.

Besides the approaches described above to reduce the state explosion problem, there are also generic techniques for the same purpose. These techniques were mentioned in section 2.5.2.

FSM and their variations are often used in specification-based testing as will be described in section 3.3.5.

3.2.3. Model-based specifications

In model-based specifications, the state of a system is modelled explicitly by mathematical constructions like sets, maps, functions, and relations. System operations are specified by defining how they affect the state of the system. Axiomatic set theory, lambda calculus, and first order predicate logic are the standard mathematical notations used in this kind of specification languages.

Typically, model-based specification languages have states and operations that change state. Invariants are Boolean expressions that restrict the set of valid states. Operations can have pre- and post-conditions associated. Pre-conditions determine the set of states where the operation can occur and post-conditions determine the state reached after executing the operation (as well as the value returned by the operation) or just restrict the set of states in which the system ends after executing the operation.

There are different kinds and styles of model-based specifications. They can be executable vs. non-executable, and explicit vs. implicit. An executable abstract specification eases validation against informal customer requirements since tests suggested by him can be quickly checked [68]. An implicit specification describes functionality by means of operations/methods with pre- and (implicit) post-conditions. An implicit post-condition allows checking the validity of the result obtained from the specification method but does not allow calculating it. An explicit specification describes functionality by means of explicit post-conditions or algorithmic method bodies from which it is possible to calculate the result expected.

The most widely used notations for developing model-based specifications are VDM-SL (Vienna Development Method Specification Language) [66], Z [179], and their object-orientation extensions VDM++ [67], and Object-Z [176] respectively.

The VDM-SL language has its origins in the IBM Laboratory in Vienna. An ISO Standard for the language was released in 1996 (ISO/IEC JTC1/SC22/WG19) [158]. Z was developed by the Programming Research Group at Oxford University in the late 1970s. The ISO completed a Z standardization effort in 2002 (ISO/IEC JTC1/SC22/WG19).

UI modelling with model-based specification languages

It is possible to find in the literature several examples of applying model-based techniques to specify user interfaces. These specifications are not as abstract as property-based specifications because the state is modelled explicitly.

Bowen, in [29], is one of the first to specify user interfaces in a model-based formal notation. He provides an abstract model of a small part of the X windows system with operations to create, destroy, and manipulate windows. Clement, in [48], specify a window interface using VDM.

Abowd et al., in [1], present the PIE model (described in section 2.5.2) rendered in the Z specification notation and then describes the model developed from it which offers a bridge between the very abstract models, like the PIE model, and methods such as formal grammars and state transition diagrams.

VDM and Z [59] have also been used to express the behaviour of interactors (described in section 2.5.2).

Gieskens and Foley claim that attaching pre- and post-conditions to interface objects can be useful because it provides a mechanism to selectively enable controls, can be used for rapid prototyping, and can be used as a base to generate automatically explanations and help text [75]. They describe an architecture supporting pre- and post-conditions which can be integrated in different environments.

Hussey et al. use Object-Z specifications for usability analysis of user interfaces [97]. They model two different user interfaces, A and B, in Object-Z and then analyse those specifications formally to access usability properties as task efficiency, consistency, and flexibility, in order to select the best suited user interface.

Model-based notations are good at representing the state but not so good at representing behaviour. There are several examples in the literature that extend model-based notations to overcome their limitations. Generally, they combine behavioural notations, like CSP, with model-based notations. The former one is used to describe the behaviour, and the latter to represent state. The inconvenient of hybrid languages is the necessity to develop tools for supporting the

verification of the result obtained by the combination of the different languages. These hybrid approaches will be subject of the section 3.2.6.

3.2.4. Property-based

With property-based specifications, systems are specified in terms of properties that must be satisfied. It does not contain the model of the system like model-based specifications do. Property-based systems can be classified into axiomatic (where the operations on the system are defined by logical assertions) or algebraic (where operations on the system are defined by collections of equivalence relations).

An algebraic specification consists of a syntactic part, and a semantic part. The syntactic part defines the syntax of the operations that is possible to perform on the system. It is described by a signature

$$\Sigma = (S, C, F)$$

with a set of sort symbols, S , a set of constructor symbols, C , and a set of function symbols, F . The semantic part characterises the behaviour of the system by defining the semantic of its operations. This semantic is described by a set of axioms, Ax , of the form $t = r$, where t and r are terms. Larch [82], and OBJ [76] are examples of sequential algebraic specifications, while Lotos [27] is an example of a concurrent one. Anna is an example of an axiomatic property based methods (pavg.stanford.edu/previous_research/index.html#anna).

UI modelling with property-based notations

Cabrera et al., in [38], use GRAPLA which is an algebraic specification language, to specify graphical user interfaces with windows, buttons, and menus. The language is later enriched with such concepts as interactive objects, and user actions [114].

Bernhard Bauer also uses an algebraic specification to model user interfaces [17]. He extends the notion of algebraic specifications distinguishing a subset of the sorts S as observable sorts (*obs-sorts*) and a subset of the functions as *interface functions*. The former set corresponds to conceptual objects which are observable to the user. The latter set corresponds to the function symbols applicable to the conceptual objects. The user interface algebraic specification is used to generate the dynamic behaviour of the UI which in turn is the input for an existing UI generator called BOSS (BedienOberflächenSpezifikationsSystem, the German translation of "user interface specification system"). BOSS is a component of the formal UI development environment, called FUSE (Formal User Interface Specification Environment) [113]. This environment also gets a formal specification of the user and tasks as input to generate the user interface.

Besides the mathematical properties of algebraic specifications and the implicit definition of behaviour in the form of axioms without a commitment with a

particularly representation, the algebraic specification use is rather cumbersome. In particular, it is difficult to find a minimal set of axioms for a given component and to evaluate when an algebraic specification is complete.

Algebraic specifications force a specific style of thinking that does not match well with the imperative paradigm in which most programmers think and implement.

3.2.5. Behaviour-based

The main advantage of behaviour-based notations is that they allow applying model checking techniques for verifying properties of user interfaces automatically. This kind of specifications is well suited to model concurrent and asynchronous systems. They specify software systems as possible sequences of states. Examples of these notations are Petri nets, process algebras, and temporal logic.

Petri nets

A Petri net consists of places (circles), transitions, and directed arcs (arrows) (Figure 18). At each moment during its execution, places can hold zero or more tokens (dots inside circles). A transition consumes tokens from the input states and outputs tokens to output places. A transition occurs when its input places contain the required number of tokens.

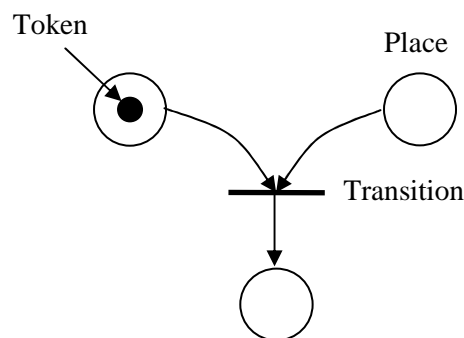


Figure 18: Petri net

Petri nets have a strong mathematical foundation on top of which several analysis techniques have been developed to carry out general validations. The main problem with Petri nets is that the "assembly line way of thinking" that characterises Petri nets is not the normal way of thinking when dealing with user interfaces. In addition, modelling complex systems using Petri nets can give rise to models of unmanageable size.

UI modelling with Petri nets

There are variations of Petri net notations aiming to reduce the size of the models. High level Petri nets like coloured Petri nets and annotated Petri nets are some of those examples.

Keh and Lewis, in [109], use annotated Petri nets to model direct-manipulation user interfaces. The annotations permit the specification of conditional flow and execution order of concurrently activated objects and do not violate the underlying Petri net theory. The model serves as the basis for the UIMS (User Interface Management System) of OSU (Oregon Speedcode Universe) and can be translated into the implementation language. The method described integrates the phases of specification, simulation, verification, and rapid prototyping of the direct-manipulation user interfaces.

Palanque, in [152], presents an object-oriented formalism specially designed for the modelling of event-driven interfaces (Figure 19). This formalism, called Interactive Cooperative Objects (ICO), is based on Petri nets. Each object is composed of four components: data structure, operations, presentation, and behaviour. ICO is used to describe the structural and static aspects of systems while their dynamic or behavioural aspects are modelled by a high-level Petri net with objects called Object Control Structure (ObCS). Transitions are labelled with variable names that are bound to objects when the transition occurs. A transition may occur when the input places are populated with required tokens (objects). At that time, the related transition action is executed. Actions can generate new objects, delete objects, and update objects. The modified and the new objects are output to the output places. The places are typed, which means that the tokens inside them should be of the same type.

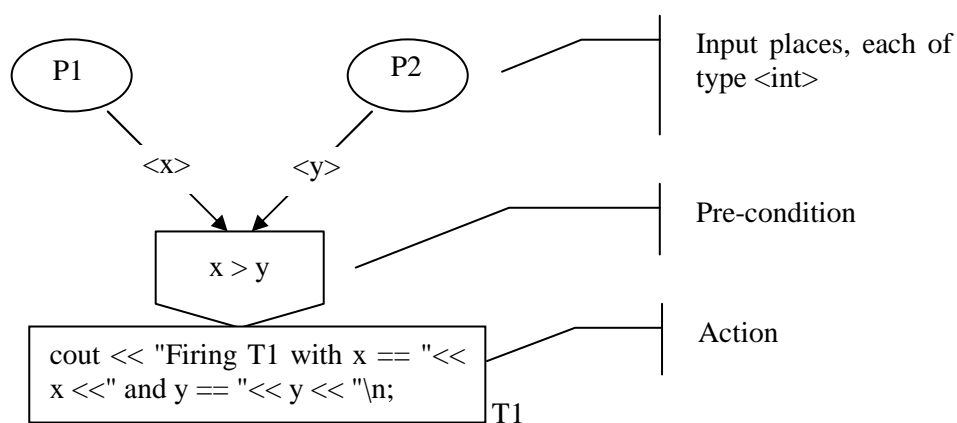


Figure 19: ObCS notation (taken from [16])

Process algebras

Process algebras are an algebraic approach to describe the behaviour of concurrent systems. The behaviour is described through processes defined in terms of synchronous events and atomic communications between them and their environment. The interaction is described through discrete points of connection called channels. Parallel composition of two processes involves connecting their interaction point by links, whenever they share the same name. Interaction happens along linked channels by handshaking or synchronisation between a sender and a receiver. The formal language also includes primitives for describing sequential composition, nondeterministic choice, concealment, and recursion. Examples of process algebras are Communicating Sequential Process (CSP) [94], Algebra for Communicating Processes (ACP), and Calculus for Communicating Systems (CCS).

UI modelling with process algebras

Process algebras are good for modelling behaviour but are not well adapted for state modelling. This is the reason why process algebras are commonly used in combination with model-based specification languages to model interactive systems. Even so, it is possible to find at least two examples of a modelling technique based on the process algebra CSP.

One of the examples uses CSP to model virtual environments that have concurrent characteristics, for which process algebras are well adapted [168].

Abowd and Dix use extension of CSP, integrating status and event phenomena, in formal specifications of interactive systems [2]. The goal is to construct a specification language that supports input and output events and status overcoming the limitations of the other specification languages that were object of analysis in their work.

Temporal Logic

Modal logic is an extension of propositional logic with operators to express different modes of truth. Temporal logic is a special kind of modal logic. It adds operators to express time which allow expressions to get different Boolean values over time:

- $\Diamond P$ or $F P$ – Finally P, means that P will happen in the future;
- $\Box P$ or $G P$ – Globally P, means that P is always true;
- $\bigcirc P$ or $X P$ – Next P, means that P will happen in the next time instance;
- $P U Q$ – P until Q, means that P happens until Q happens.

There are different kinds of time models: Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and Timed CTL (TCTL). In Linear Temporal Logic, each time instance can have only one successor, while in Computation Tree Logic each instance time can have more than one successor. In addition, CTL adds two more operators to express properties about all possible successors (A - Always), and to express properties about one path within all future possible paths (E - Exists).

A specification in temporal logic can describe safety, liveness, and fairness properties. Safety properties express the things that should not happen in the system. Liveness properties describe things that should happen in the system. Fairness properties are used to solve indeterminism.

UI modelling with temporal logic

Johnson and Harrison use temporal logic to specify interactive control systems and as a means of analysing usability requirements [103]. They overcome the previous weaknesses of the abstract specifications by capturing temporal properties identified as crucial to the success or failure of interactive control systems. They developed a tool called Prelog (Presentation and Rendering of LOGic specifications) which combines a temporal logic interpreter with a structured graphic system and high level device abstractions to support prototyping of an executable subset of the formalism as a means of accessing the qualitative "look and feel" of potential implementations.

Mezzanotte and Paternó, in [131], use Action Computation Tree Logic (ACTL), which is a branching-time temporal logic, to express high level properties of user interfaces like the possibility of performing a task at any state

```
AGEF <task_performance> true
```

and visibility

```
AG([user_actionx] EF<User interface appearance>true)
```

meaning that each user action will give feedback to the user by modifying the presentation.

Butterworth and Cooke, in [37], use a notation based on Temporal Logic of Actions (TLA) to model a window with a scroll bar. At each instant, there is a relation, *rep*, between the position of the scroll bar button and the windowed data shown to the user (section 2.5.2).

Although Temporal Logic allows reasoning about generic properties of interactive systems and verifying properties automatically through model checking, it also rises problems when someone wants to express more specific properties related to particular aspects of some systems. Also, expressing properties in Temporal Logic is not easy and programmers may resist doing so.

3.2.6. Hybrid approaches

The goal of hybrid languages is to combine characteristics of two or more specification languages to construct a richer final language which combines the better of the original ones. Like was already mentioned in previous sections, one popular approach is to combine model-based specification languages with behaviour-based.

MacColl and Carrington use a hybrid specification language constructed from Object-Z and CSP to specify interactive systems [116].

Galloway and Stoddart present a new language called ZCCS constructed on top of the Z and CCS specification languages [71].

Martins, in [120], presents a new formalism, called Interaction Scripts, to model dialogue controllers. The formalism is compositional and powerful enough to express both sequential and concurrent dialogs. Interaction Scripts and UI presentational descriptions are the input language of a prototype system, called GAMA-X [42], for the automatic generation of Assisted User Interfaces able to communicate with the application prototype. Later, the developers of GAMA studied the possibility of extending the system with UI adaptability characteristics. The GAIA system was developed for that purpose [119].

There are three different types of interaction scripts: *Decision* (when a selection among several options needs to be taken); *Synth* (to synthesize a command; these scripts are used to update the state of the application); and *ValSynth* (scripts used to call operations that query the state of the application). Other kinds of scripts do not have an explicit type defined.

A script has a static block (`GIDecls`) to introduce all the identifiers used by the script and a dynamic block (`GIBehav`) to describe the interactive behaviour controlled by it according to the following syntax:

```
GIDecls:: [ValT: ValType]
          Symbol: SYM-set
          Type: GIType
          Args: IdVar -> IdType
          Var-UI: IdVar -> IdType
          Var-Apl: Ldecl: IdVar -> IdType
                 Atribs: IdVar -> IdVar
          Extern: GIName-set
          SubGi: GIName-set

GIBehav:: Init: IdVar -> ExpValue
          Context: [BoolExp]
          EvSeq: ExprComp
          Trans: TrDescr
          Exec: [ExecDescr]
```

Interaction Scripts use CSP (or CCS) operators ("." – for sequence; "||" – parallelism synchronous; "|" – parallelism asynchronous; "+" – alternative; "*" – repetition) to model the order in which the arguments of an operation are read (`EvSeq` clause within behaviour block).

The behaviour composition of different Scripts is described by Labelled Petri nets with added expressive power:

- it is possible to associate a condition to a transition that determines when the transition can occur;
- it is possible to interrupt the execution of a Petri net *A* so as to execute completely another Petri net *B* at which time *A* execution can go on.

Although the most commonly examples are the ones that combine model-based with behaviour-based specification languages, there are also examples that combine other kinds of specification languages.

Bramwell combines behaviour-based with action systems. He uses CSP and an action system [30].

The RAISE (Rigorous Approach to Industrial Software Engineering) uses the RSL (RAISE Specification Language) which is another example of a hybrid specification language (spd-web.terma.com/Projects/RAISE). It has characteristics of the model-based languages, like VDM, algebraic methods, like ACT ONE and OBJ, and process algebras, like CSP and CCS.

These new languages constructed from the combination of others have a rich description power but require an additional effort to combine the semantic of the sublanguages that were used to construct them.

Another drawback of these hybrid languages is that the new semantic may require the development of new tools to support them.

3.3. Specification-based test case generation

Specification-based testing allows higher degrees of test automation. After constructing the model, it can be used as input to a test case generator. The technique used by the generator depends on the characteristics of the model. There are several approaches to automate the generation of test cases from models. However, there are some problems and challenges that cross all models: how to determine when to stop the generation; and how to evaluate the quality of the test suite generated. **Coverage criteria** can be used for both purposes. They can determine when to stop the generation and can also be used to assess the quality of the generated test suite. A good test suite should combine a good code coverage with a good requirements (or specification) coverage.

When the source code of the software application is available, **white-box** testing can be applied by analysing the source code and applying coverage criteria on the implementation to measure the quality of tests. However, often source code is not available, and **black-box** testing must be performed. In these cases, using model-based testing allows to apply coverage metrics on the model as a quality measurement. Although model-based testing can have many advantages like the automatic generation of test cases, it also often suffers from the gap between the

modelling paradigm and the programming paradigm. In addition to absent source code, often the access to the actual functionality of the software application is barred by a GUI that represents the only interface to the software. Anyway, even when GUI code is available, it may be interesting to test the system through the same interface that is used by final users (as addition to the other test methods used).

Besides the characteristics of the models, the test strategy used also influences the test generation method used. The so called **tests-to-pass** are usually used as a first iteration and check if the fundamental parts of the software work using valid input values. **Tests-to-fail** are used in subsequent test iterations and try to break the system using invalid inputs or valid inputs at the operational limits. **Random input** generation algorithms and **fault-based methods** are examples of test-to-fail methods. The random input generation goal is to drive the system to crash (see section 2.5.3). Fault-based methods attempt to ensure that the software does not contain certain types of faults (e.g., mutation testing).

3.3.1. Test data generation

An important issue related to the generation of test cases is the generation of test data, that is to say, the input values of the test cases. The available methods for this purpose can be implemented either statically or dynamically and classified as random, goal-oriented (generate test data for an unspecific path), and path-oriented (generate test data for a specific path) [62].

Random methods

Randomly test data generation is a relatively easy technique to implement but results in weak coverage. It generates random values from the input domain of the program.

Goal-oriented methods

Goal-oriented methods try to drive the system into a given goal by two different methods [62]: the chaining approach and assertion-oriented approach. The first one tries to find a path to the execution of a given goal node based on data dependence analysis [65]. The second tries to find any path to an assertion that does not hold.

Several goal-oriented methods use AI planning techniques. Mayhauser et al., in [121], use an AI planner assisted approach to generate test cases based on high level test objectives for testing a robot controlled tape silo. Memon et al., in [126], also use AI planning techniques for generating automatically test cases for GUIs.

Path-oriented methods

Symbolic testing is an example of a path-oriented test data generation method. It replaces program variables by symbols and calculates constraints that represent possible symbolic execution paths. When a program variable is changed during execution, the new value is expressed as a constraint over the symbolic variables. A constraint solver system can be used to find, when possible, concrete values that cause the execution of the path described by each constraint (Figure 20).

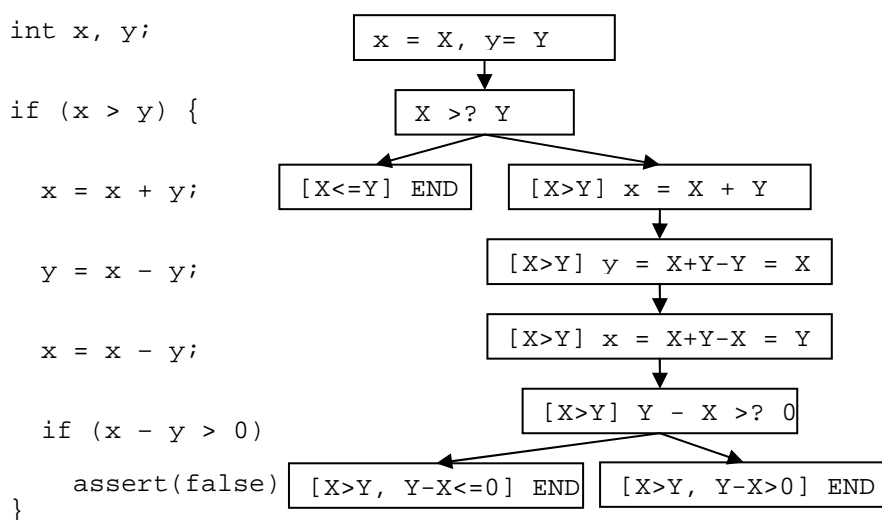


Figure 20: Symbolic execution tree example

Nikolai, in [185], describes a prototype tool for unit testing based on symbolic execution and constraint solving. The tool can automatically find test cases that cover all statements. Pretschner [159] translates an AUTOFOCUS specification into Constraint Logic Programming and symbolically executes the resulting system. Meudec, in [130], presents an automatic test data generator based on constraint logic programming (CLP) and symbolic execution.

Other approaches combine **symbolic execution with model-checking** [12]. The main idea is to use the model-checker to traverse the symbolic execution paths. The test coverage criterion is encoded in the property the model checker should check for.

3.3.2. Generation of expected test results

Test cases are sequences of methods with input parameters and results expected. Formal specifications can be used to generate test data (input parameter values) and also as a test oracle (to calculate the expected results). This is the main advantage of specification-based testing when compared with techniques that

generate test cases (sequence of methods and input parameters) from software code but cannot calculate expected results.

However, there are different kinds of formal specifications (and different specification styles) and it is not possible to calculate the expected results from all of them.

Formal specification can be explicit or implicit. A specification is explicit if it has the behaviour fully described allowing the exact determination of the result expected for each method call with input parameters as well as the next state. A specification is implicit if it describes the behaviour of the system in a higher level of abstraction, for example, as contracts with pre- and post-conditions (without specifying the body of the methods), that allow checking if the results (and next state) obtained from the implementation under test are valid but does not allow calculating the expected results (and next state). For example:

```
Seq<int> Sort (Seq<int> arg)
ensures Forall {i,j in result.Indices, i<j;
              result[i]<=result[j]};
{}
```

The *Sort* method defined implicitly above does not describe how to sort a sequence of integers. However, the post-condition (*ensures* clause) allows checking if the sequence of integers provided as result is sorted, i.e., if it is a valid result.

3.3.3. Coverage analysis

Coverage analysis aims to measure the extent to which a given verification activity has achieved its objectives and can be used to evaluate the quality of the test suite used and also determine when to stop the verification process. It is usually expressed as a percentage referring to the accomplished part of an activity.

Coverage measures can be generally classified into requirements coverage and structural software coverage. Requirements coverage analysis measures the extent to which requirements have been verified while structural coverage analysis measures the extent to which code structure has been executed [91].

Although in the literature coverage analysis is usually applied to code, it can also be applied to the specification. For instance, requirements coverage of the specification can be used to verify if higher level requirements are met in the specification, and structural coverage on the specification can be used as a quality evaluation of the test suit and as a stop criterion.

Requirements coverage

Requirements coverage analysis precedes structural analysis and is less systematic because it usually does not contain a complete specification of the behaviour of the system. One example could be a coverage criterion measuring the degree in

which use cases or scenarios were verified. Scenarios describe how the system and the user should interact to achieve a specific goal. They usually refer to common usages of the system and may not be a full description of the behaviour of the system. Scenarios are not designed to cover the entire program so, scenarios coverage is not a sufficient test coverage criterion [106].

Structural coverage

Structural coverage analysis is used to measure the degree to which code (or specification) has been exercised. There are different types of structural coverage criteria [91]:

- **Statement coverage** – every executable statement in the program is invoked at least once during testing;
- **Decision coverage** – requires testing the expressions' outcome for true and false evaluation. For instance, the Boolean expression (A or B) must be tested for true, e.g., TF, and for false, e.g., FF. But, this criterion does not guarantee testing the effect of all clauses within an expression, e.g., the effect of B is not tested, it is always False.
- **Condition coverage** – requires that each condition within an expression takes all possible outcomes, overcoming the problem of the previous criterion. But, it drops the requirement that each expression takes all possible outcomes. So, to test (A or B) two tests, TF and FT, are enough.
- **Condition/decision coverage** – combines requirements of the two previous criteria. The tests should be constructed in a way that all possible outcomes of both decisions and conditions must be tested. So, to test (A or B) two test cases are needed: TT and FF.
- **Modified condition / decision coverage (MC/DC)** – increases the condition/decision coverage with an additional requirement that is to show that each condition affects independently the outcome of the decision. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. Usually MC/DC requires $n+1$ test cases for a decision with n inputs. To test (A or B) three test cases are needed: TF, FT, and FF. This type of coverage criterion is considered necessary for adequate testing of critical software.
- **Multiple condition coverage** – it requires that each possible combination of inputs to a decision is executed at least once (exhaustive testing). That is to say, 2^n tests for a decision with n inputs. This criterion is most of the times unpractical.

Although there are some general testing strategies (test case and test data generation, and coverage analysis), there are also some testing techniques that are closer to the characteristics of the specification from which test cases are generated. We will go through each type of formal specification illustrating based

on the scientific literature the techniques available to generate test cases form them.

3.3.4. Test generation from grammars

Grammars are often used for usability evaluation but they can also be used to generate test cases by applying rewriting techniques. The idea is to apply rewriting rules to generate valid sentences within the described language which can then be used as a test case.

Sirer and Bershad, in [174], describe an experiment using production grammars for generating test cases for testing the Java virtual machine. Production rules are described in a domain specific language called *lava*.

3.3.5. Test generation from FSMs

Most of the test case generation techniques from FSMs are based on traversal algorithms that calculate paths within the FSM to achieve a defined test coverage criteria like transition coverage, transition-pair coverage, complete sequence, and full predicate coverage (described by Offutt in [144]). The transition coverage criterion is satisfied by a test case capable of testing every transition in the state-based specification. The transition-pair coverage criterion is satisfied by a test case that traverses all possible pairs of adjacent transitions. The goal of the complete transition coverage criterion is to traverse paths that have some special meaning to the tester based on his knowledge and experience. In the case of FSM variants like VFSM (with guard conditions), the full predicate coverage criteria ensures that every clause in a predicate are tested independently.

The size of the test suite is influenced by the coverage criterion used. In particular, a test case that satisfies full predicate coverage criteria also satisfies transition coverage criteria.

Model-checking

Model checking is a static analysis verification method performed on the specification (see section 2.5.2). It is a technique for verifying properties expressed in temporal logic, which is a kind of behaviour-based specification, over a system described as a finite state machine and can also be used as a technique to generate test cases. Whenever a property, expressed in temporal logic, does not hold in a system described as a finite state machine, model-checking tries to generate a counter-example. When a counter-example is produced, it can be used as a test case. It is a sequence of transitions, or trace, in the state machine with inputs and expected outputs. To be effective as a test-case generation technique, the properties about the system should be described in such a way that counter-examples produced by them can be used as test cases.

Model-checking and mutation testing

Model-checking in combination with mutation can be used as a fault-based testing technique [10,24]. Mutation techniques introduce small changes (faults) by applying mutation operators into the original specification. The changed specifications are called mutants. The goal is to construct test cases that distinguish each mutant from the original by producing different results. If that happens, it is said that the test case has killed the mutant. A good test case should be capable of killing the mutants because if it is able to detect the small differences generated by the mutation operators it is expected that it will be good at finding real faults. One of the problems of mutation testing is the incapacity of the technique to generate test data.

Black et al., in [24], use mutation analysis and model checking technique to generate automatically tests from formal specifications. Okun et al., in [145], describe two specification-based mutation testing methods that use a model checker to guarantee propagation of faults to the visible outputs. Ammann, in [10], is another example of applying model-checking and mutation techniques to generate test cases (Figure 21).

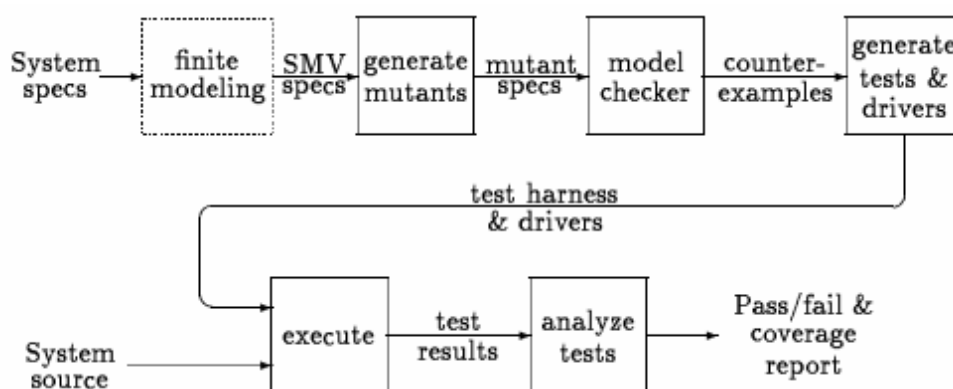


Figure 21: Testing flow (taken from [10])

Mutation operators can be applied to the finite state machine or to the temporal logic constraints. The former case is a failing test in the sense that a good implementation should produce different result values from the corresponding tests since the FSM is not a good description of the system anymore, it is mutated. The latter case is a passing test because test cases are generated from a FSM that models correctly the system so a good implementation should produce the same results for the same inputs.

The main problem with FSMs is the state explosion problem. Most of the FSM variations try to diminish that problem.

3.3.6. Test generation from model-based specifications

An approach to generate test cases from model-based specifications is called **equivalence class partitioning**. Assuming that the program behaves analogously for inputs in the same class, one test with a representative value from a class is sufficient. A partition of some set, S , is a set of non-empty subsets SS_1, \dots, SS_n , such that each SS_i and SS_j are disjoint, and the union of all SS_i 's equals S . If a defect is detected by one member of a class, it is expected that the same defect would be detected by any other element of the same class. So, the test cases can be significantly reduced depending on the granularity of the classes considered. There are different techniques to split the input domain into classes. Dick and Faivre, in [55], developed one technique to partition the domain by rewriting the pre- and post-conditions of the specification into disjunctive normal form (DNF). Each disjunction is used as an equivalence class. Hierons, in [93], presents an algorithm that starts by rewriting the specification into the form $\bigvee_i (P_i \wedge Q_i)$, in which P_i represents a pre-condition and Q_i represents a post-condition, to divide the input domain into classes.

The problem that comes after the domain has been partitioned into different classes is to generate input values for each different class. Instead of selecting arbitrary/random values within a class, **boundary value analysis** tests boundary conditions of equivalence classes choosing input boundary values. This technique is based on the knowledge that input values at the boundaries or just beyond the boundaries of the input domain tend to cause errors in the system.

A variation of equivalence class partitioning is **type-based selection** [189]. In this case, the type of each input variable is used as suggestion of equivalence classes. For example, for an input variable of the type set, the specification should be tested with the empty set, one set with a single element, and a set with more than one element. After having a partitioning of the domain, one test case for each class should be constructed.

Aichernig [5] uses fault injection on the modelling level to generate test cases and to validate executable models. The test case generation algorithm gets a specification with pre and post-conditions $D(Pre \vdash Post)$ and its faulty design $D'(Pre' \vdash Post')$ as inputs.

In other approaches, the user defines manually the input values and then test cases are generated based on those defined domains.

Additional care must be taken so as to check if the input parameters calculated do not forbid calling all the methods specified. This can happen when the input parameters do not generate states where a pre-condition of one of the methods gets the true value.

After defining (or generating) the input domains, test case sequences may be constructed by essentially two different methods. One of those methods is called test "on-the-fly" that evaluates after each method call the set of available methods (i.e., the pre-condition is true) and calls one of those methods (arbitrarily selected) with appropriate input values [192]. The other method explores completely the

specification, i.e., after each method call, it calls all the available methods with all possible input parameter values. This process constructs a FSM that can be saved and used later to produce test sequences that fulfil defined coverage criteria. These two methods are supported by the Spec Explorer tool [39] which is a model-based testing tool built by Microsoft Research. Besides supporting test "on-the-fly", it also provides a way to translate AsmL or Spec# specification into a FSM [79] that is subsequently used as a base to generate test cases that fulfil defined coverage criteria.

3.3.7. Test generation from property-based specifications

Property-based specifications describe systems by a set of properties or axioms that they must satisfy. Rewriting and constraint solving are techniques used to generate test cases from these specifications. Given a set of expressions (logical assertions or equivalence relations) and the set of variables within those expressions, constraint solving techniques try to find an instantiation of the variables which reduce the expressions to true.

Gannon et al., in [73], describe a system, DAISTS, where test cases are written as axioms that are used to exercise the implementation. The system uses the axioms to write the test drivers. After providing the values for the required inputs, the test process is automated.

Dan et al. [53] propose an approach to derive test cases from a RSL (RAISE Specification Language) specification using a combination of partition analysis (used by model-based languages) and rewriting (used by algebraic languages) test case generation techniques. This is particularly well suited for a RSL specification due to its hybrid characteristics that combine features of both model-based and algebraic specification languages.

DeMillo, in [54], combines the mutation technique and algebraic constraints that describe how to find particularly types of faults to generate test data automatically.

3.3.8. Test generation from behaviour-based specifications

There are different examples of behaviour-based specification languages. Temporal logic is one of those examples and can be used by model checking techniques for test case generation, as described in the above state-based section.

There are also approaches which analyse the execution traces to generate test cases. A trace in CSP is a finite sequence of events. Another example of test case generation from CSP specifications is illustrated in [31]. The goal is to test Universal Mobile Telecommunications Systems (UMTS). They start by constructing a transition graph with all possible interleaving and parallel tasks. Then, a test driver computes all paths through this graph that are used as test sequences.

3.3.9. Test case generation from GUI models

There are several examples in the literature of generating test cases from formal specifications of GUIs. In particular, FSMs and their variations are frequently used to model GUIs and to generate test cases.

FSMs

Shehady, in [170], uses Variable Finite State Machines (VFSM) to model GUIs and to cope with FSM scaling problems (see section 3.2.2). The VFSM is converted into a FSM to generate test cases using the partial W algorithm [70]. The test cases are applied to the GUI and the results obtained are compared with the results expected. The comparison is performed at the end of the test case execution so that, even if the inconsistencies are found at the beginning of the test cases, the execution of an entire case is required.

Belli, in [19], presents an approach to model the legal and the illegal behaviour of GUIs using FSA, Finite State Automata, and regular expressions. Belli starts by identifying all legal sequences of user system interaction and then expands the model with illegal behaviour. The final model is used to generate test cases that can bring the system into legal states, producing the desired system response, or into a faulty situation, producing an error message.

Andrews, in [11], uses hierarchies of FSMs to model Web applications and uses constraints to reduce the set of input values and to help solving the state explosion problem. The Web application is divided into clusters and each of those clusters is described as a FSM. These clusters are structured into a hierarchy with different levels of abstraction. The bottom level of clusters corresponds to software modules and Web pages. The top level of abstraction is the application finite state machine where detailed clusters are represented as a single node. In this level, arcs represent possible transitions between lower level FSMs. They can be annotated with input constraints and propagated information. Test cases are generated from detailed FSMs by applying transition coverage criteria which are then substituted into the aggregate sequences for the aggregate FSM (the upper level).

White and Husain [194] identify complete interaction sequences (CIS) of GUI objects and selections needed for invoking responsibilities which are activities that produce an observable effect. Each CIS is described as a FSM that is subject to several transformations to deal with the state explosion problem. One of the transformations is an abstraction technique based on strongly connected components and the other is a merging technique of the CIS states that are structurally symmetric. Each identified component is substituted by a super-state. Test cases are generated from the reduced FSM by traversing all paths in the FSM. Every time a super-state is found, the test path of the corresponding component is inserted into the test at that point. Each test path of a component should be included at least once in the overall test suite. One of the problems of this approach is the difficulty to identify strongly connected components and structural symmetry.

Planning

Memon uses a model with a hierarchical structure in his work [128] to model GUIs and to guide the generation of test cases, but not to reduce the size of the test suite. He defines a set of operators organized in hierarchies that correspond to user actions. The operators at upper levels are constructed from simpler ones at lower levels. These simpler operators correspond to user actions. Each operator has a pre-condition that must be true before executing the operator, and an effect. Memon uses planning from Artificial Intelligence to generate test cases. Given a set of operators, an initial state, and a goal state, a planner produces a sequence of operators that will change the initial state to the goal state. He generates test cases from the upper hierarchical levels of abstraction and then nested invocations to the planner during abstract operator decomposition. Alternative test cases can be obtained by substituting the different test cases obtained for the lower levels into the high-level plan.

3.4. Conformity Check

The purpose of specification-based testing is to verify if the implementation is conforming to the specification. This activity of the specification-based testing process can be performed manually, which involves too much work, or automatically. Conceptually, to compare the expected value with the one obtained, an abstraction function from the implementation to the specification level comprising one or two maps need to be defined:

- A mapping (R) from the state variables of the implementation to the state variables of the specification, which describes how the abstract states of the specification are represented in the implementation [4]. One implementation is adequate if it can represent all the states that could be represented by the abstract specification. Since the implementation is more detailed, multiple concrete states (at the implementation level) may correspond to the same abstract state (at the specification level).
- A mapping (T) from operations at the implementation level to operations at the specification level (including input and output parameters), so related operations can be run on both levels and results obtained compared.

With these two maps it is possible to run related operations, at the specification and implementation levels, comparing the results and also the initial and final states. However, the map between state variables (R) can be dispensable if additional methods are defined to observe the state (or some part of the state). This is the approach followed by the Spec Explorer tool [39] that provides observable methods to read state without performing any updates.

Let's assume the following execution model (both at the specification and implementation levels) [148] to describe conformity tests:

- The system behaviour is described by transitions between states caused by operations executed in response to user actions or events.
- The operations' effect may be described as a function F from initial state S_1 , and possible input arguments $FArgs$, to final state S_2 , and possible outputs $FOut$ (assuming deterministic behaviour).
- The outputs produced can be a message or sequence of messages sent to the user.
- The system state may be or not observable by the user. A specification can describe behaviour of the user interface, from the user perspective, by making internal state observable (with observable state variables or by providing methods to read the state) or by sending appropriate output messages to the user.

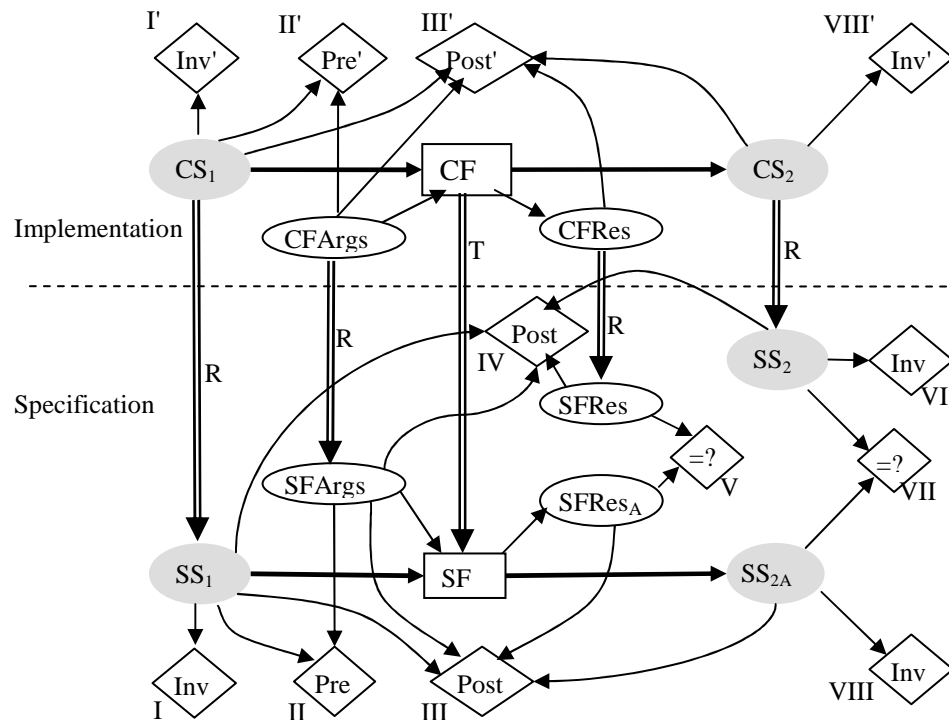


Figure 22: Conformity tests model

Assume that we start at a concrete state CS_i when we apply concrete function CF . In consequence, at the specification level, we start at SS_i , that corresponds to CS_i ($SS_i=R(CS_i)$), over which it is applied the specification function SF (equivalent of CF , i.e., $SF=T(CF)$).

Figure 22 summarizes the several elementary tests, I to VIII and I' to VIII', that may be performed to check the conformity between the specification and the implementation.

The characteristics of the specification that serves as test oracle determine the subset of elementary tests shown in Figure 22 to perform. With a contract-based specification (followed, for instance, in [75]), with implicit operation definition in the form of post-conditions (see section 3.3.2), there are two possible situations:

- The post-condition is verified on the specification level after mapping the state obtained from the implementation level onto the specification level (test IV). Additionally, initial state invariant (I), pre-condition (II), and final state invariant (VI) may be tested.
- The specified post-condition goes through a code generation process for being tested at the implementation level (test III'). This approach can be found in [4]. It is supported by VDMTools. Additionally, initial state invariant (I'), pre-condition (II'), and final state invariant (VIII'), all at the implementation level, may be tested.

With an executable specification with explicit operation definitions (called model programs in Spec# and algorithmic bodies in VDM++) it is possible to compare outputs obtained from both levels (V). When the internal state of the specification is visible, it is possible to perform additionally tests: initial invariant (I), pre-condition (II), and final states (VII).

Another issue related to conformity check is the execution model. The related operations of both levels can be run in a "lock-step" mode in which results are compared after each step, or in a batch-oriented way, in which case the test suite is run as a whole in the specification level, and expected results are kept in memory for later comparison with the results obtained from the execution of the implementation (which is performed in a different execution time instant). One advantage of the batch-oriented way is the need to execute the model only once and not every time test cases are executed. The main drawback is the additional need of memory to keep the results expected.

In particular, the so called "on-the fly testing" combines in a single algorithm the test case generation and execution and executes each operation as a lock-step in each level comparing results after each of those execution steps.

3.5. Conclusions

In this chapter it was described the specification-based testing process. In particular, different ways of modelling GUIs, different techniques available to generate test cases from different formal specifications, and different ways of conformity evaluation regarding the characteristics of the formal specification used.

Grammars were very common to specify command-based user interfaces but they are so not well adapted to model direct-manipulation and concurrency of the modern windowed and mouse driven interfaces.

A grammar-based specification does not represent state explicitly. The state is represented by an expression built as a sequence of operations/actions. Without an explicit representation of state, it is difficult to represent the state observed by the user.

One of the problems about modelling interactive systems with state machines is the state explosion problem. This problem can be even worse when modelling GUIs and the techniques available to diminish this problem may not be sufficient. Even so, state-based specifications are well adapted to model GUIs and there are several techniques to generate test cases from these specifications. Since state is explicit in these specifications, a map between states of both levels, specification and implementation, can be easily established to perform conformity tests.

Model-based notations are good at representing state but not so good at representing behaviour. In particular, some model-based notations do not have support for events which can be a major drawback when modelling GUIs. Even so, the fact of modelling state explicitly by mathematical constructions like sets, maps, sequences, tuples, and so on, facilitates establishing a map between specification and implementation states which may be helpful for performing specification-based testing or conformity testing automatically. In addition, model-based specification languages are closer to the imperative paradigm of the programming languages commonly used by programmers. This characteristic makes them one of the best positioned candidates for being accepted in industrial environments.

Furthermore, there are several test case generation techniques on top of model-based specification languages (see section 3.2.3) that makes possible to implement algorithms to generate test cases automatically from them.

A model-based notation can be conceptually seen as a sequence of states, and transitions between those states that correspond to the methods described in the model and that are responsible to evaluate the system from state to state. There are algorithms that convert model-based specification languages into state-based specification language representation, like FSM, which makes possible to apply techniques and traversal algorithms developed on top of these languages for test case generation (see section 3.3.5) and also apply static verification techniques like model-checking that prove properties expressed in temporal logic automatically.

Property-based specification languages do not represent state explicitly. These specifications are good for modelling behaviour but not so well adapted for state modelling. That's why they are commonly used in combination with model-based notations to model interactive systems.

Algebraic specifications force a specific style of thinking that does not match well the imperative paradigm in which most programmers think and implement.

Properties about interactive systems expressed in temporal logic can be verified automatically by using model-checking techniques. Even so, it is difficult to

express properties in temporal logic and to express more specific properties related to particular aspects of the systems.

Besides verifying properties automatically, model-checking techniques can also be used to generate test cases. The properties in temporal logic must be constructed in such a way that counter-examples produced can be used as test cases.

Considering that one of the goals of this research work is to promote the use of formal methods in industrial environments, the specification language to use should not force a complete divorce with the normal way of thinking of the programmers. This requirement excludes grammars, and property-based specification languages. The specification languages that can be, in our point of view, more easily accepted by programmers are the ones more closely to the imperative programming implementation languages commonly used by programmers which are the model-based and state-based notations.

Another criterion followed to guide the decision of which specification language to use in this research work was the expressive power of the language. It would be desirable to use a language with

- **explicit state** – to model the state of the GUI, for instance, the content of a textbox, and to facilitate establishing a map between states of the specification and implementation to perform conformity tests automatically;
- **support for scenarios** – to model some user visible function or high-level requirement that achieves a user goal and model typical ways of using the GUI;

The primary goal of this research work is to improve the current GUI testing methods and tools. So, the specification language tool support is also important to enable the experimentation and validation of the ideas developed. The tools available were studied according to:

- **facility to extend functionalities** – the tool should have an API or some other mechanism to facilitate extending its functionalities in order to automate the activities involved in specification-based testing, such as, test case generation, test case execution, and conformity evaluation;
- **test automation** – the set of testing activities already supported and automated by the tool.

A more detailed studied was carried out comparing two different tools, VDMTools (www.csk.com/support_e/vdm/index.html) and Spec Explorer (research.microsoft.com/SpecExplorer/), supporting VDM++ and Spec# specification languages respectively.

The VDM Toolbox provides a Corba compliant API, which allows other programs to access a running Toolbox. Thus, any code such as a graphical front-end or existing legacy code may control any Toolbox component. So, it is possible to program extensions to the tool to run a GUI, simulating user actions, and the model of that GUI written in VDM++ and compare the results obtained from both

to evaluate conformity between the model and its implementation. This tool enables the manual definition of a set of tests and check after running those tests, the degree of specification coverage achieved by those tests.

Spec Explorer has support for test case generation, facilities to establish maps between specification actions and implementation methods, support for test cases execution, and conformity evaluation. It is well adapted for performing specification-based testing of software applications through their code or API but requires extensions for testing software applications through their GUI. It provides an API that allows extending easily the tool functionalities.

After analysing all these aspects, the choice was the Spec# specification language, developed by Microsoft Research in Redmond, and the model-based testing tool, Spec Explorer. They will be presented in more detail in the next chapter.

Nevertheless, the aim is that the main ideas developed in this research work can be applied in other environments following similar paradigms.

Chapter IV

Specification-based GUI Test Automation

This chapter presents a new approach to model and test GUIs. Models are written in Spec# and possibly structured in different levels of abstraction, whether modelling atomic user actions, scenarios, or high level properties. A FSM is extracted from the model and validated according to standard test adequacy criteria. Test cases are generated from the extracted FSM based on a new test coverage criterion that ensures coverage of a particular level of abstraction obtained from a navigation map view and other views for each dialog within the GUI application under test. A tool prototype supporting this kind of specification-based GUI testing is described. This tool is an extension of the specification-based testing tool, Spec Explorer, developed at Microsoft Research, which already supports the automatic generation and execution of test cases for API testing, but requires too much work when testing software applications through their GUI.

GUI testing is laborious, boring, and time and resource consuming. The approaches and tools available to aid the testing process are not satisfactory (see section 2.5). The goal of this research is to improve current GUI testing methods and tools, taking advantage of formal behavioural models to enable the automatic generation of test cases and the automatic conformity checking of the implementation with respect to the specification. On the whole, we want to contribute to the construction of higher quality graphical user interfaces.

The contributions of this research spread over modelling (section 4.2), test case generation (section 4.3), and test case execution (section 4.4). A prototype tool was developed to support the overall testing process of software applications through their GUIs based on a formal specification written in Spec#. It is an extension to Spec Explorer, a model-based testing tool developed at Microsoft Research, that already supports automatic generation and execution of test cases for API testing, but requires that the actions described in the model are bound to methods in a .NET assembly.

The Notepad application that is shipped with the Microsoft Window operating system is used along this chapter as a running example to illustrate the approach. It is a basic text editor that can be used to edit, view, and create or update simple text files. This software application is also used as a case study to validate and evaluate the specification-based testing approach proposed in this dissertation in Chapter V.

4.1. GUI Testing Process

Specification-based testing checks if an implementation of a software system conforms to its specification. The main activities of the GUI model-based testing process proposed in this dissertation are presented in Figure 23.

The starting activity is the construction of the GUI specification/model. The model may be constructed from the requirements, in a forward engineering process, or from an existing application, by a reverse engineering process. The set of modelling techniques proposed in this approach is suited for testing purposes, and promote modularity and reusability (see section 4.2). The specification captures the requirements and enables checking if those requirements are fulfilled by an implementation. The model may be constructed at different levels of abstraction whether modelling atomic user actions, high level scenarios, or high level properties of the system. There is one module or class to describe each window within the GUI under test.

Generically, there are two different kinds of actions in the model: actions to observe the state of the system (e.g., actions that model the eyes of the user reading the text shown by a textbox); and actions to control the system (e.g., actions that describe the user events sending text to a textbox). Inside Spec Explorer [39], the former actions are annotated as *probe* while the latter actions are annotated as *controllable*.

The model is written in Spec# and converted into a FSM that results from the bounded exploration of the model. The exploration process, supported by the Spec Explorer tool, infers the set of methods available in each state (pre-condition true) and calls them with appropriate parameter values. Domains of such parameters are defined manually by the tester and have a deep influence on the generated FSM. If the FSM does not have the desired properties it may be regenerated with new

defined bounds (input domains). The quality of the generated FSM is assessed according to adequate and coverage criteria based on the defined scenarios, high level properties, and testing goals. The way to access the quality of the FSM will be explained in section 4.3.3.

After generating the adequate FSM it is possible to calculate the test cases from it based on FSM coverage criteria. However, executing all possible test cases generated from this FSM may be not realistic due to the huge size of the FSM generated and consequently the huge number of test cases.

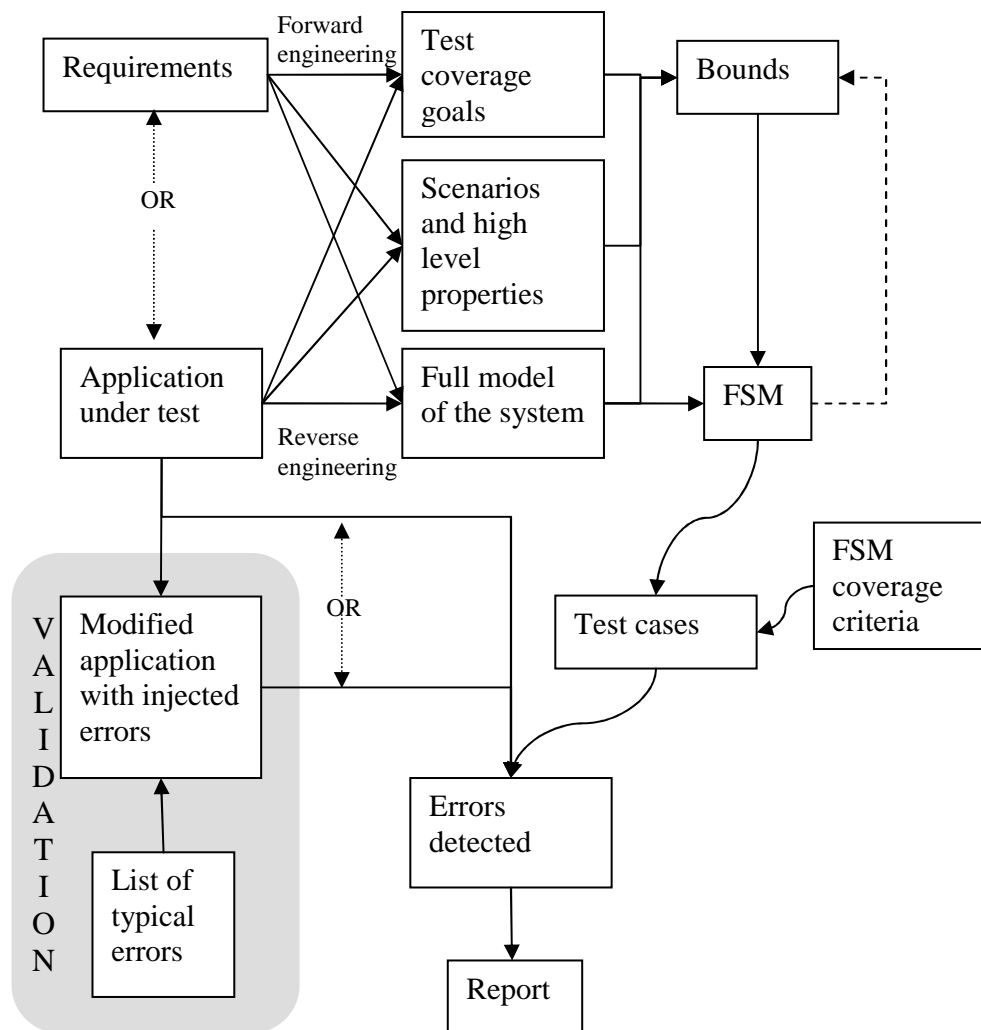


Figure 23: Overview of the GUI modelling and testing process

A new algorithm will be presented in section 4.3.4 to reduce the FSM while guaranteeing coverage of the intermediate level of abstraction defined by the high level GUI properties described by the navigation map and dialog views. Once this pruning technique reduces the size of the initial FSM, test cases may be generated based on the full transition coverage criterion and then executed.

However, the generation process of the initial FSM itself may be unfeasible due to memory space or time restrictions. In this case, different approaches (not necessarily disjoint) can be followed:

- stop the generation of the FSM when all the identified scenarios and test boundary conditions are covered by it, or
- build scenarios to drive the software application into test boundary states if it is not possible to obtain a FSM that covers them within time and resource limits, or
- split the software application into different sets of functionalities and test them independently, or
- build scenarios to shortcut some functionalities where a exhaustive testing is not needed, e.g., build a scenario to open a file kept in disk avoiding exhaustive testing of the complete Open dialog.

Test cases are generated from the FSM model after selecting FSM coverage criteria. Once generated, test cases are executed on the specification and on the implementation (constructed software application or modified software application with injected errors) and the results obtained are compared. The specification plays the role of a test oracle describing the expected results. Every time there is an inconsistency between results obtained from both levels they are reported. Reasons for such inconsistencies are three-fold:

- test cases are trying to trigger events in a window that is not reachable or is not opened (e.g., when a modal dialog is open and the window we want to reach is behind the modal dialog);
- test cases are trying to interact with a control that cannot be found;
- the expected result was not displayed (e.g., a text box does not display the expected content).

To execute test cases automatically over the GUI under test some intermediate code to simulate the user actions is needed. This code is constructed automatically by a tool (GUI Mapping Tool) developed on purpose and presented in section 4.4.

The GUI Mapping Tool extends Spec Explorer to automate the GUI testing:

- it adds the capability of gathering information about the physical GUI objects that are the target of the user actions described in the model;
- it automatically generates a .NET assembly with methods that simulate the user actions upon the GUI application under test;
- it automatically maps the methods in the generated .NET assembly to the model of such methods described in the specification.

The capacity of detecting errors of the overall testing approach is evaluated by using a modified application with a list of known injected errors as a GUI under test.

The Spec# system and the automated model-based testing process with the Spec Explorer tool are described in next sub-sections.

4.1.1. Spec# System

The Spec# programming system (Figure 24) developed at Microsoft Research lab in Redmond, USA, consists of the object-oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier (Figure 25) [15]. It is an attempt to support more cost effective production of high-quality software and is fully integrated into the Microsoft Visual Studio.

Spec# supports literate programming in allowing a Spec# program to appear spread over several separate sections in a document along documentation like text, tables, and diagrams. It uses a special style for the program (Spec# style) different from the style/styles used for the documentation. The compiler and other tools can extract the code from the document.

The programming language, Spec#, extends the existing object-oriented .NET programming language C# with specification constructs like pre-conditions, post-conditions, invariants, and non-null types; the compiler emits run-time checks to enforce these specifications; and the verifier can check the consistency between a program and its specifications [15].

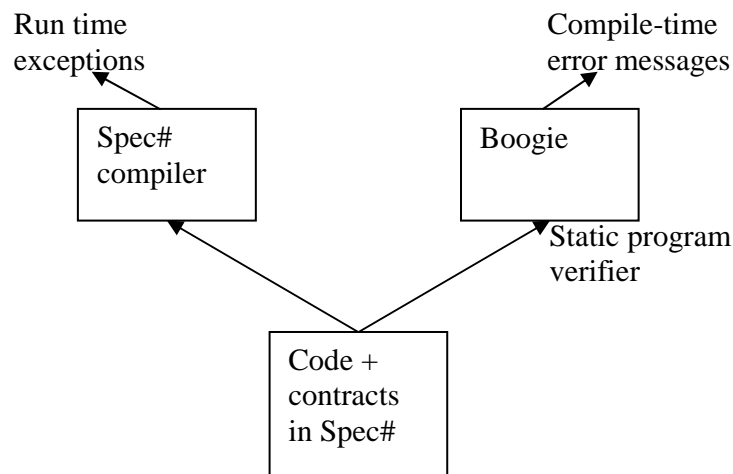


Figure 24: Spec# system

Besides producing executable code from a program written in the Spec# language, the Spec# compiler also serializes all specifications into a language-independent format and attaches these serialized specifications to the program components in which they were defined. Instead of working on source code, the Boogie static program verifier works on top of the compiled code and can, for that, be used to verify code written in other languages than Spec# as long as they provide a process to attach contracts/specifications to the code.

The Boogie static verifier (Figure 25) translates the intermediate language, MSIL, and metadata into its own intermediate language, BoogiePL. Then an inference mechanism obtains properties such as loop invariants from this BoogiePL language [14]. The BoogiePL program and properties derived go through the

weakest-precondition generator which performs a sequence of transformations till ending as a verification condition that is then used by the automatic theorem prover (Simplify).

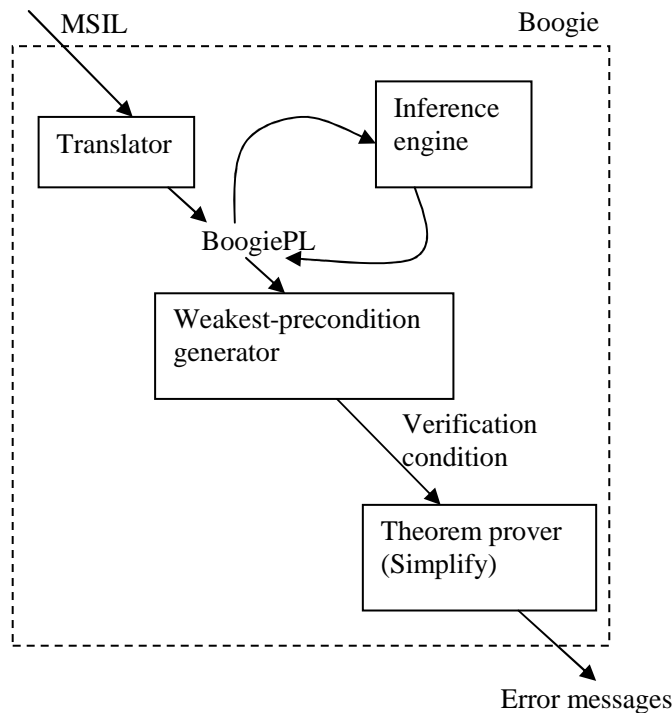


Figure 25: Boogie static verifier

4.1.2. Automated model-based testing with Spec Explorer

Spec Explorer [39] is a software modelling and testing tool from Microsoft Research. A formal executable model can be written in the abstract state machine language (AsmL) (research.microsoft.com/fse/AsmL) or Spec# [15]. AsmL is an executable specification language based on the theory of Abstract State Machines (ASMs) [28].

A model written in Spec# describes a possibly infinite state transition system. States are modelled by state variables. Some of the methods in the specification are annotated as actions that represent the possible transitions of a transition system. These actions can have pre-conditions, written as “requires” clauses that define the states in which they are enabled. Thus, actions can be seen as the guarded update rules of an ASM. It is important to note that the states can have a very rich structure. In the case of GUIs, this allows to faithfully model the GUI’s state from a user perspective. For example, a state variable can hold the textual content of a field. Methods annotated as actions can be used to model complex user actions (e.g., enter a string into a field, issue a command, load contents from files, etc.) and describe its effect on the state of the system.

There are four different kinds of actions: *observable*, *controllable*, *probe*, and *scenario*. *Observable* actions are asynchronous and describe the spontaneous execution of an action in the AUT (application under test) possibly caused by some internal thread. *Controllable* actions describe actions that are controlled by the user of the modelled system. *Probe* actions describe actions that do not update internal state of the modelled system but only read the state of the system. Probe actions are invoked by the test harness in every state where they are enabled to check whether the model and the implementation have the same characteristics in a given state. *Scenario* actions describe sequences of sub-actions. A scenario can be used to drive the system into a desired initial state.

From a Spec# model, it is possible to extract a Finite State Machine (FSM) by an exploration process. This process will execute the actions of the model and, at each action call, it will use values for the parameters taken from domain sets defined manually by the tester. Besides some default values defined for specific types like Booleans (true, and false), Spec Explorer does not provide any support for the definition of parameters' domains. Even so, the choice of these domains has profound impact in the characteristics of the generated FSM and is a crucial point of the whole process. There are different ways to combine the parameters' values so as to use them in each action call: Cartesian product, pair-wise combination, and an enumeration of tuples.

Besides the domains' definition, one of the main difficulties in FSM extraction is due to the fact that an ASM specification can have a huge, possibly infinite, number of states, so a good pruning technique is needed to deal with the state explosion problem. Griskamp [79] uses hyper-states as a form of abstraction to extract a FSM from an ASM. In addition, Spec Explorer allows the user to limit the exploration of the model in various ways [191]: definition of additional pre-conditions; restrict the parameters' domains; definition of state filters; definition of state groups; and definition of stop conditions. This will be explained in more detail in section 4.3.1.

A state group is a set of expressions, $G = g_1, \dots, g_k$, over one or more states. Two different states, s and t , are in the same group, or are G-equivalent, if they evaluate the expressions in the same manner ($\forall 1 \leq i \leq k \cdot g_i^s = g_i^t$). A group is a maximal set of G-equivalent states.

The FSM and the expected results of each execution step that result from the exploration of a given Spec#/Asml model are kept in memory. Once the FSM is built, a test suite with a set of test segments (sequences of actions with input parameters and results expected) is generated. Spec Explorer provides different algorithms to generate test suites: full transition coverage; shortest path; and random walk. This will be explained in more detail in section 4.3.1.

After constructing the test suite, test conformance between the specification and the implementation can be performed. Conformance between model and an implementation can be established by binding model actions to implementation methods, executing the test suites on the implementation, and comparing the results obtained with the expected ones kept in memory. Spec Explorer provides a

mechanism that binds the action methods in the model with matching signatures in the AUT. Whenever the map needs to be established between methods with different signatures, the user must relate those methods one by one. The implementation can be written in any language supported by the .NET framework.

To track observable actions, Spec Explorer instruments the AUT at the binary (MSIL) level [191]. During execution, the instrumented AUT calls back into the conformance engine, notifying it about occurrences of observable method calls. These occurrences are buffered which allows them to occur even during the execution of a controllable method in the implementation.

All inconsistencies detected are reported to the tester that can select any of the reported errors and check the FSM path which gave rise to the error. This path can then be analysed in order to correct the implementation or the specification.

Spec Explorer also supports "on-the-fly" testing. In this case, the test generation and test execution are connected into a single algorithm [192].

Another functionality of Spec Explorer is the graphical visualization of the FSM obtained by the exploration of the Spec# or AsmL models. Sometimes, the FSM obtained is so huge that viewing it graphically can not be very useful. Besides being used for pruning the exploration of the model, state groups can also be used to define different views of the model. This feature can be helpful to define different levels of abstraction of the same model and also to see some specific features of a huge model that otherwise could not be analysed. For models with scenario actions, graph visualization includes a property that controls whether the graph will show scenarios in collapsed (sub-actions are hidden from the graph) or expanded form (sub-actions are visible in the graph).

Spec Explorer is well adapted to test software systems through their API. However, when the source code of an application is not available and the only way of interacting with it is through the GUI it requires too much work [149]. This happens for two main reasons:

- As explained above, it is necessary to define a map between specification and implementation actions so as to compare results obtained at each execution step. When the only way to interact with a software application is through its GUI, this map cannot be established (the same happens when the source code of the software is not available and it does not provide an API). To overcome this limitation, it is necessary to build intermediate code to simulate the user actions, for instance, clicking on a button, sending text to a textbox, observing the text shown in a textbox that is the result of some operation. The methods inside this code will be mapped to specification methods and the related methods will be run step-by-step and results obtained compared.
- The manual construction of the intermediate code is laborious and takes so much time that may compromise the whole process.

The prototype tool developed in this research work is intended to overcome such limitations of GUI black-box testing by automatically generating the mapping code that allows interacting with a software application.

4.2. GUI modelling with Spec# and Spec Explorer

State machines are well suited to model reactive systems. A state machine defines a set of states and transitions between states caused by actions. GUIs are reactive systems in the sense that they can respond to user actions. State machines can be very useful to guide the testing of software applications [112].

A specification written in Spec# is executable. Besides invariants, pre-conditions (written as *requires* clauses), and post-conditions (written as *ensures* clauses), one can write executable method bodies (also called model programs) in a high-level action language, with primitives to change the value of state variables, and even call external methods defined in .NET assemblies. (The execution model of Spec# is based on the formalism of abstract state machines [28]). This allows the specification to be used as a test oracle: the expected effect of a user action can be computed by executing the specification, and compared with the actual effect of the same user action on the application under test. This process is currently automated by the tool developed during this research work with the help of the Spec Explorer tool and using an intermediate library to simulate user actions over the implemented application.

In order to be effectively used as a test oracle, the specification should be written for testability. That is, it should describe user requirements with enough detail and rigor to allow a person or a machine to decide whether an implementation, as perceived through its GUI, obeys the specification. In particular, names for actions and state variables in the specification should be chosen in a way such that their counterparts in the user interface of the implemented application can be found straightforwardly (and automatically).

Another advantage of an executable specification is that it can be tested *per se*, to validate it and check its internal consistency (check that method bodies do not violate pre-conditions, pos-conditions and invariants, check explicit assertions, etc.). However, this possibility will not be exploited here.

Besides being used as test oracles, formal specifications can further be exploited to automatically generate test sequences (sequences of user actions and action parameters). A common two-step approach, currently supported by the Spec Explorer tool, is as follows: first, a finite state machine (FSM) is generated from the specification, by exploring all the states that can be reached from a given initial state or set of initial states (each state is a possible combination of values of the state variables, and each transition corresponds to a user action with actual

parameters); secondly, a test suite, comprising one or more test sequences, is generated from the FSM, so that all states and transitions are covered.

Unfortunately, there are also common problems with this approach: the state explosion and, ultimately, the test case explosion problem. The test case explosion problem is particularly important in presence of interactive applications, because of the slow response of GUI's to user actions, when compared to in-memory operations.

The challenge we address in the sequel is that of modelling GUIs in a way such as to deal with the state explosion problem and automatically generate test suites of manageable size that still guarantee adequate testing.

4.2.1. Modelling GUI structure and behaviour

The models used by Spec Explorer find their inspiration in the Abstract State Machines (ASMs) formalism [28]. ASMs provide a way to model any system at any level of abstraction. This is adequate for GUI modelling, because, depending on the context, one may want to model user actions at different levels of abstraction: at operating system level (where a click event is the sequence of pressing and releasing the mouse button), at API level (where a click event is seen as an atomic action), at user task level, etc.

Independently of the level of abstraction considered (lower level messages, or higher level messages that correspond to sequences of lower level messages), a GUI implementation places the messages in a queue and processes those messages in order. This behaviour can also be adequately modelled as an ASM with guarded actions which fire only when appropriate messages are fetched from the queue.

Using Spec#, one can build a formal specification of an interactive application, describing the actions a user can perform at each moment (press a button, fill a text box, etc.), and the expected effect of each user action, in terms of changes to the state of the application (according to a model of the application state as perceived by the user) and possible effects to the environment (e.g., write a file to disk). The effect of user actions may depend not only on the current state of the application, but also on environment conditions (e.g., existing files in disk).

The state of the application is described by means of state variables (static or instance variables). Without restrictions, the state space S of an application manipulating a set of variables $V = \{v_1, \dots, v_{|V|}\}$ will be the Cartesian product of the domain values of the variables in the set V , i.e., $S = dom(v_1) \times \dots \times dom(v_{|V|})$.

4.2.1.1. Modelling windows' controls

Typically, windows are composed of several interactive controls with which users interact. There are different kinds of interactive controls, e.g., buttons, text boxes, check boxes, list/combo boxes, etc. The state of controls is modelled by state variables. One or more variables are used for that purpose and depend on the

characteristics that are considered relevant from the modeller perspective. For instance, the state of a textbox can be modelled by several state variables (Figure 26):

```
// a string keeping the text
string text = "";

//an integer keeping the position of the cursor
int posCursor = 0;

// a string keeping the text selected
string selText = "";

// a Boolean variable that tells whether the text
// has been changed
bool dirty = false;

// etc...
```

Figure 26: State variables of a textbox

In addition, the user actions interacting with each control are modelled by methods. Methods have pre-conditions that determine the states where the modelled actions are possible. Typically, pre-conditions include a clause that checks when the window where the control is placed is enabled and possible others that select among the first set of states the ones where the control is enabled (Figure 27). For instance, the "Find Next" button inside the find dialog of the Notepad application (Figure 28) is enabled whenever the dialog is enabled and the text inside the "Find what" textbox of the same dialog is filled. Each dialog/window is uniquely identified by a name, e.g., the find dialog is identified by "Find".

```
namespace FindDialog;
//...
[Action] FindNext()
requires IsEnabled("Find") and FindWhat!="";
{ //... }
```

Figure 27: Find Next pre-condition



Figure 28: Find dialog inside Notepad software application

To enable conformance testing of the outputs displayed to the user, methods annotated as actions should also be provided to observe the state of the GUI that is exposed to the users' eyes. A query method can be provided for each observable state variable, with the name of the variable and a suitable prefix. Spec Explorer refers to such actions as *probes* (Figure 29).

```
namespace Notepad;
//...
// keeps the state of the text inside the main window
string text;

[Action(Kind=ActionAttributeKind.Probe)]
string GetText()
requires isEnabled("Notepad"); {
    return text;
}
```

Figure 29: Probe action example extracted from the Notepad's GUI model

A probe only observes the current state and does not change it. Probes are treated differently from ordinary actions during test case generation, as we will see later.

4.2.1.2. Modelling windows

For modularity reasons, except for trivial applications, the top-level windows of the application are better modelled in separate namespaces or classes.

Inside each module (namespace or class) corresponding to a top-level window, state variables are used to model the abstract state of that window and the controls inside the window, and methods annotated as actions are used to model the possible user actions on that window and on the controls of the window. All the actions inside each module, except the one that launches the application, have at least one pre-condition: that the corresponding window is enabled.

Windows can be modal or modeless. When a modal window is open (e.g., the Save and Open windows in the Notepad application), the other windows of the application are disabled. Since this is a common feature of GUIs, a separate reusable module – a window manager – was created to handle it (see Appendix A.3.). The window manager is part of the model, and its state is part of the model state.

The window manager provides the following self-explanatory helper methods:

```
namespace WindowManager;
void AddWindow(wndName, parentWndName, isModal)
void RemoveWindow(wndName)
bool IsEnabled(wndName)
void SetFocus(wndName)
bool HasFocus(wndName)
```



```
bool IsOpen(wndName)
string GetWindowWithFocus()
Set<string> GetEnabledWindows()
```

Figure 30: Window manager

When a method opens/closes a window it should add/remove that window to/from the window manager. When a window is removed, all its child windows are also removed. Message boxes are also registered in the window manager but are not modelled as separate modules because of their simple structure. Message boxes have a set of buttons (typically two or three) that correspond to different possible answers to a question. Acknowledge messages are a special kind of message box with only one button. Such button is pressed by the user as a way to acknowledge the information displayed in it.

There is only one window with input focus, at each time, within the same application. This is the window to which user actions are redirected to. The window with the input focus must be in the set of the enabled windows. A window is enabled when it is open and does not have a child modal window on top. Typically, two modeless windows belonging to the same application can be opened at the same time and it is possible to switch input focus between them.

The model of the GUIs can abstract the focus property of the windows. In each moment, only the set of windows (`GetEnabledWindows()`) with which is possible to interact with is relevant. This modelling technique will abstract all "switch focus" transitions between modeless windows. When the focus property is modelled, the pre-condition of each method inside a window (module) should have a clause checking if that window has focus (`HasFocus(windowName)`); otherwise, the pre-condition should include a clause checking if the window is enabled (`IsEnabled(windowName)`).

4.2.1.3. Modelling message boxes

As already mentioned above, message boxes are not modelled as separate modules (namespace or class). Message boxes have a simple structure that only requires the user to press one of the shown buttons. This can easily be modelled as a method with a parameter carrying the user's answer.

There are two different kinds of message boxes: the ones that give some information to the user and that ask the user to press an "ok" button. These are called acknowledge messages boxes (see Figure 31 noting the "MsgAck" prefix);

```
[Action] void MsgAckCannotFindWord()
requires IsEnabled("MsgAckCannotFindWord"); {
    RemoveWindow("MsgAckCannotFindWord");
}
```

Figure 31: Message box of acknowledge

and the ones that ask for input from the user and wait until the answer is chosen from a set of available options (buttons) (see Figure 32 noting the "Msg" prefix).

```
[Action] void MsgSaveChanges (string option)
requires IsEnabled("MsgSaveChanges");
{
    RemoveWindow("MsgSaveChanges");
    // ...
    switch (option){
        case "y": //...;
        case "n": //...;
        case "c": //...;
        default: //...;
    }
    //...
}
```

Figure 32: Message box with different possible answers

4.2.1.5. Modelling communication between windows

Windows are modelled as separate modules (namespaces or classes) for modularity reasons and to promote reuse. The designer of a reusable module (window) defines its state and methods but does not know in advance which kind of application will make use of them. Method calls between the reusable module and an application that reuses it occur in both directions:

- The application (or test driver) may call methods of the reused module. From the testing perspective, inputs are methods invoked with parameters while outputs are the values returned by those methods. This is the traditional situation in unit testing.
- The reusable module may generate events (originated from the user or internally generated) that cause the invocation of methods in the application (or test stub), by some kind of callback mechanism (event handlers, or sub-classing and method overriding, or interface implementation). Again, from the testing perspective, the outputs are the events and parameters passed to the application, while inputs are returned parameters.

Testing the second kind of interaction (callbacks) poses specific issues and challenges, as already noted in [184]:

- An application method invoked in a callback may, in turn, invoke methods of the reusable module (reentrancy situation) and have access or change its intermediate state. Hence, the internal state of the reusable module when it issues a callback is not irrelevant. Moreover, some restrictions may have to be posed on the state changes that an application may request when processing a callback.
- During testing, one has to check that: (1) the appropriate callbacks are being issued; (2) when a callback is issued, the reusable module is put

in the appropriate internal state; (3) during the processing of a callback, the application doesn't try to change the state of the reusable module in ways that are not allowed.

Buttons are common in GUIs and are a good example to illustrate communication between a reusable module and an application. Buttons usually have associated methods that are called when users press them. These methods can communicate with the other elements of the application where the module is being reused.

Another example are dialog windows that can be reused across several applications such as Open and Save dialog windows that can be found, for instance, in Microsoft Notepad, Microsoft Word and Microsoft Excel. In Appendix A.1 it is possible to see one solution to model these windows promoting reusability. Callbacks from the dialog window to the application that uses it (e.g., Notepad) are modelled by applying the Observer design pattern [72].

4.2.2. Modelling scenarios

It is also useful to model high-level scenarios that capture some user visible function (or high level requirement) that achieves a user goal and model typical ways of using the GUI. Scenarios are constructed on top of atomic actions. Usually, independent scenarios are used to model normal and exceptional user sequences of interactions. Parameterized scenarios model the external behaviour of a specific user visible GUI functionality for all possible parameters' values (Figure 33).

Scenarios can be used as a mechanism for structuring the GUI model in different levels of abstraction; for testing purposes as a way to identify test conditions that would be covered by manual tests and that can be seen as the minimum set of conditions to automatically test; as a technique to drive the application into a desirable specific state; as a technique to prune the exploration process; and to guide the process to determine the parameters' domains of the model actions that will be used by the exploration process to generate the FSM.

Spec Explorer has a mechanism that supports modelling scenarios. Scenarios are a special kind of actions that are capable of invoking other model actions. Scenario actions are enabled by their pre-conditions. However, unlike other kinds of actions, when a scenario calls other actions, Spec Explorer records the intermediate states. When test cases are generated, the scenario sub-steps (or sub-actions) are used.

Let us first see which scenarios will have to be modelled in our Notepad illustration:

OpenScenario: It is possible to load (open) data from a file in disk. If the text in the main window was updated give an opportunity to save the content to a text file before opening the new file. Inform the user if the name of the file to open does not exist. This can be modelled in Spec# as shown in Figure 33.

```

[Action(Kind=ActionAttributeKind.Scenario)]
void OpenScenario(string fileToOpen,
                 string saveChanges,
                 string fileToSave,
                 bool overwrite)
requires IsEnabled("Notepad") &&
        saveChanges in Set{"y","n","c"}; {
    Open();
    if (IsEnabled("MsgSaveChanges")) // if dirty
    {
        MsgSvBfrOpen(saveChanges);
        if (IsEnabled("Save")) // saveChanges == true
        {
            SaveDialog.SetFileName(fileToSave);
            SaveDialog.Save();
            // file exists
            if (IsEnabled("MsgOverwriteFile"))
            {
                SaveDialog.MsgOverwriteFile(overwrite);
                if (IsEnabled("Save")) // overwrite=false,
                    // so get out of
                    // the cycle
                    SaveDialog.Cancel(); // scenario end
            }
        }
    }
}
//(saveChanges != c || !dirty)
if (IsEnabled("Open")) {
    OpenFileDialog.SetFileName(fileToOpen);
    OpenFileDialog.Open();
    if (IsEnabled("MsgAckFileNotFound"))
    {
        OpenFileDialog.MsgAckFileNotFound();
        OpenFileDialog.Cancel(); // end of the scenario
    }
}
}

```

Figure 33: Open file scenario within the Notepad application

SaveScenario: It is possible to save text (new or updated) to a text file (new or existing). If the text file already exists, ask the user if its content should be updated.

FindScenario: It is possible to search a string within a text:

- In a case sensitive or case insensitive way;
- By looking for the string backwards or forwards the current mouse position.

EditScenario: It is possible to type, select, cut, copy, paste, and delete text. The occurrences of a given string in a text can be replaced by another one all at once or step-by-step. Inform the user whenever the string does not occur in the text.

4.2.3.State machine views

The Spec# specification can be viewed graphically as a FSM by a bounded exploration process. The graphical view gives a more perceivable way to validate the model and serves also as a basis for test case generation. However, for non-trivial systems, the FSM obtained can be so huge that analysing it as a whole may be unfeasible. To overcome this problem, it is possible to construct different views of the same FSM by abstracting some properties. These views are smaller FSMs which allow for validating the model and also defining different testing objectives and test coverage criteria.

Without restrictions, the state space of a software application, S , manipulating a set of variables, V

$$V = \{v_1, \dots, v_{|V|}\}. \quad (1)$$

is the Cartesian product of the sets of values which can be assigned to the variables in the set V .

$$S = dom(v_1) \times \dots \times dom(v_{|V|}). \quad (2)$$

Consider a FSM described as a set of states, S as above, an initial state, s_{init} in S , and transitions $T = \{(s, a, s')\}$, where s and s' are states in S , and a is an action. Each action triggers transitions which drive the system from origin state s to destination state s' .

State machine views are slices of the original FSM which affect different state variable subsets (techniques for automatically identifying pieces of a program which affect a selected subset of its variables are known as *program slicing* [22,88]). State machine views are obtained by projecting the state onto the variable within v_i , $PFSM_{v_i}$, where v_i is set of variables that are relevant to the view/property to analyse. By using operators of relational algebra¹ [188], these projections (views/slices) are expressed by

$$PFSM_{v_i} = \{(\pi_{v_i}s, a, \pi_{v_i}s') \mid (s, a, s') \in T\} \quad (3)$$

¹ π - projection, e.g., $\pi_B T = (2,5)$; $\pi_2 T = (2,5)$; $\pi_{A,B} T = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$;

σ - selection e.g., $\sigma_{A=1} T = (1,2,3)$

T =

	A	B	C
1	1	2	3
4	4	5	6

State machine views can also be obtained by projecting the state onto a set of expressions over state variables.

There are some typical views that result directly from the structure of the GUIs. One of them is the navigation map that describes how to open/close windows of the application and also how to switch between the windows of the same application. Other views are the ones that describe the behaviour of each dialog of the application abstracting away from the behaviour of the other dialogs.

Although the size of the FSMs that describe each dialog independently is much smaller than the original FSM, it may remain unmanageable. The challenge is to determine views with a manageable size (to analyse and test) that still describe the relevant behaviour of the system. So, other views for other higher level properties can be defined as will be explained later on.

4.2.3.1. Navigation map view

The navigation map of an interactive software system modelled as explained above, using the window manager and the focus property, can be easily obtained from the `GetWindowWithFocus()` method defined within the window manager (Figure 30). This view can be expressed mathematically as the projection of the FSM states onto the variable that keeps the name of the window with the focus (Figure 34).

$$\begin{aligned} \text{Navigation map}(FSM) = & \quad (4) \\ & \{(\pi_{\text{GetWindowWithFocus}})^s, a, \pi_{\text{GetWindowWithFocus}}^{s'} \mid \\ & (s, a, s') \in T\}. \end{aligned}$$

The navigation map view can be obtained in Spec# by

```
string NavigationMap { get {  
    return GetWindowWithFocus();  
}}
```

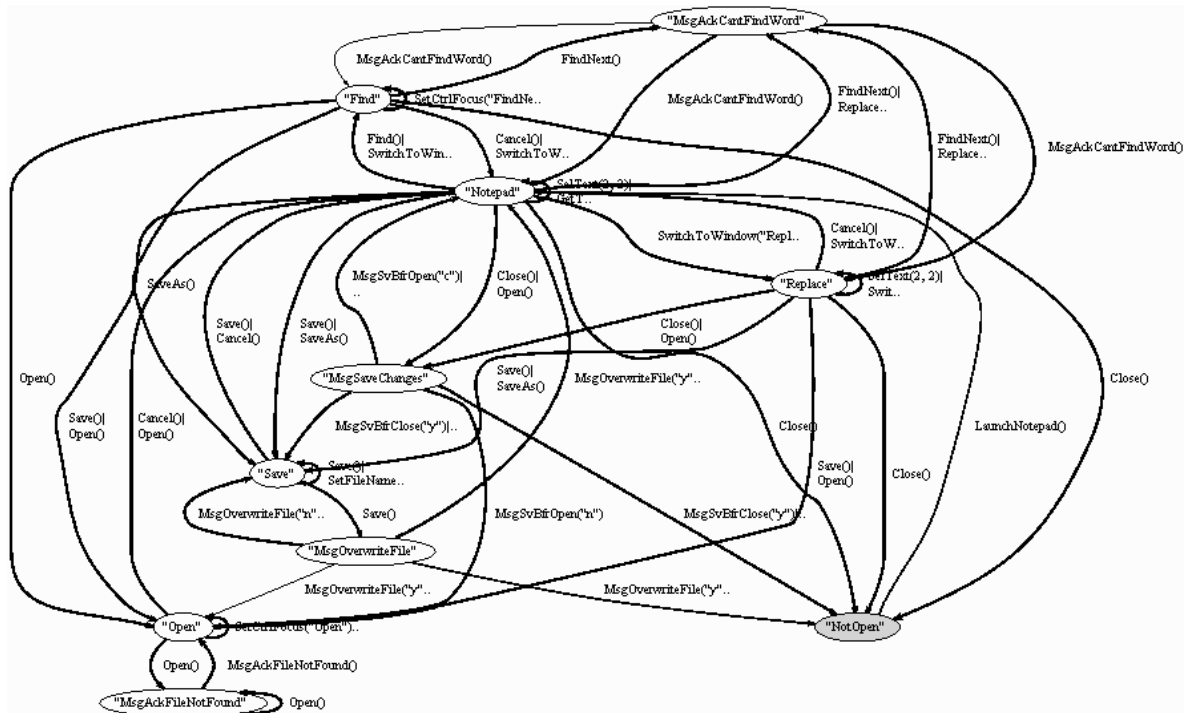


Figure 34: Navigation map obtained from focus property of the windows

The transitions visible at this level of abstraction are the switch focus transitions between modeless windows opened at the same time (e.g., Find dialog and Notepad main window) and transitions that open/close windows of the application. All transitions that occur inside the windows/dialogs are abstracted as one transition from the state group, representing the dialog, to itself.

When the model of the application abstracts the focus property, the navigation map can be obtained from the method `GetEnabledWindows()` defined inside the window manager module (Figure 30).

$$\text{Navigation map}(FSM) = \quad (5)$$

$$\{(\pi_{\text{GetEnabledWindows}})s, a, \pi_{\text{GetEnabledWindows}}s'\} \\ |(s, a, s') \in T\}.$$

In the presence of modeless windows, there may be more than one window enabled at the same time, in which case, the method returns a set of more than one window name. This is the case of the Find and Replace dialogs that appear in states paired with the Notepad main window in Figure 35.

```
Set<string> NavigationMap { get{
    return GetEnabledWindows();
}}
```

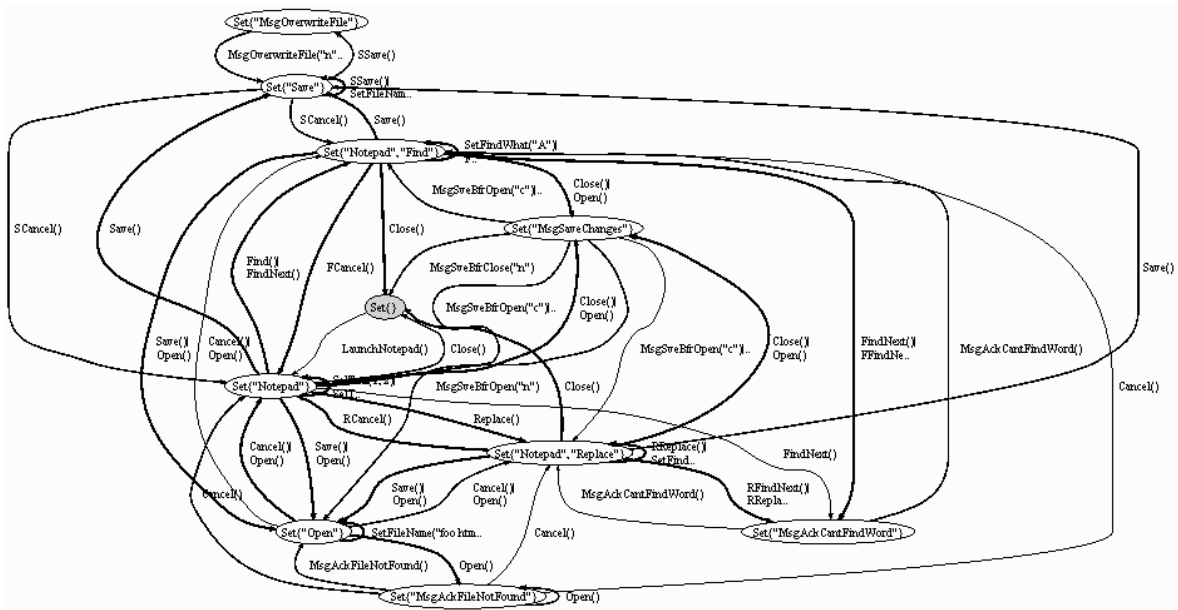


Figure 35: Navigation map obtained from the enabled windows' property

Message boxes are a special kind of windows. Showing them at this level of abstraction may introduce too many details. It is possible to construct another navigation map abstracting from those message boxes (Figure 36).

```
string NavigationMap { get {
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (IsOpen("Open")) return "Open";
    else if (IsOpen("Save")) return "Save";
    else if (IsOpen("Find")) return "Notepad/Find";
    else if (IsOpen("Replace")) return "Notepad/Replace";
    else return "Notepad"; }}
```

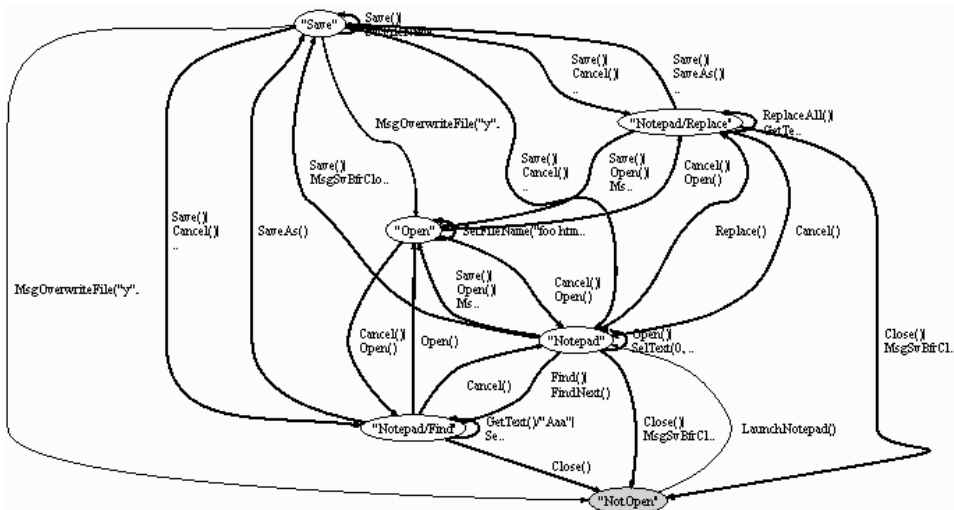


Figure 36: Navigation map obtained from opened windows abstracting away the message boxes

Test cases generated from these views can be used to test all possible navigation paths allowed by the modelled system.

4.2.3.2. How to obtain one view for each dialog/window

The state machine view describing the behaviour of each dialog i , $PFSM_{Di}$, is constructed by abstracting the states where the dialog i does not have input focus. If the focus property of the windows is not modelled, the $PFSM_{Di}$ is constructed by abstracting the states where the dialog i is not open.

Additionally, it is also possible to model the focus property of the interactive objects inside each window. In that case, the view of the dialog behaviour can be obtained by projecting the state of the dialog onto the state variable that points out the interactive object with the input focus at each moment. The $PFSM_{OpenDialog}$ is given by the 3 groups of states inside the rounded rectangle with dashed line (excluding the groups that enclose the states where the Open dialog is closed and the states where the Notapd is closed). (Figure 37).

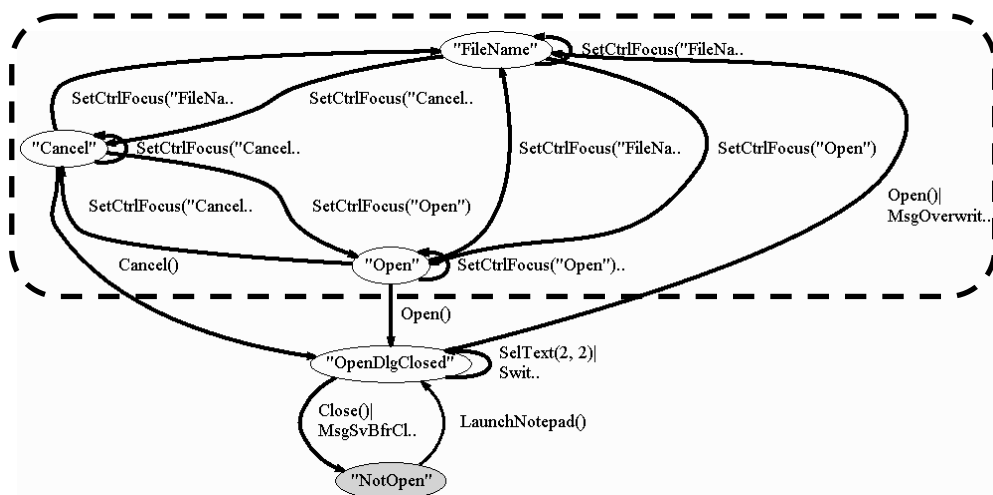


Figure 37: Open dialog view obtained from the projection onto the interactive object with the focus in each moment

When the focus property of the interactive objects is abstracted from the model, it is possible to obtain the view of the dialog behaviour by projecting the global state of the modelled system onto the variables that are manipulated (read or written) by the dialog (Figure 38). The concept of manipulated variable will be explained and formalized in the sequel.

As an example, the second level of the hierarchical structure of the Notepad model for the Open dialog, $PFSM_{OpenDialog}$, that is to say, the projection of $FSM_{OpenDialog}$ onto the variables manipulated by the *Open* dialog, which are `fileNameO` (keeps the name of the file to open), and `dirO` (keeps the directory of the file to open), can be given by the expression:

$$\begin{aligned}
 PFSM_{OpenDialog} \log = & \{(\pi_{Manipulate\ dVariables}("Open")(s), a, \quad (6) \\
 & \pi_{Manipulate\ dVariables}("Open")(s')) \\
 & | (s, a, s') \in T \wedge s, s' \in \sigma_{IsOpen}("Open")S\}.
 \end{aligned}$$

```

<string, string> OpenFileDialog {
  get {
    if (IsOpen("Open")) return
      <"fileNameO="+fileNameO, "dirO="+dirO>;
    else return
      <"NotOpen", "NotOpen">;
  }
}

```

This view will have one state group grouping all states in which the *OpenDialog* is closed and other state groups (3) grouping all state instances of the dialog that evaluate the expression in the same manner, that is to say, have the same values for the manipulated variables and different values for the non-manipulated variables (Figure 38). The $PFSM_{OpenDialog}$ is given by the 3 groups of states inside the rounded rectangle with dashed line (excluding the group that encloses the states where the Open dialog is closed).

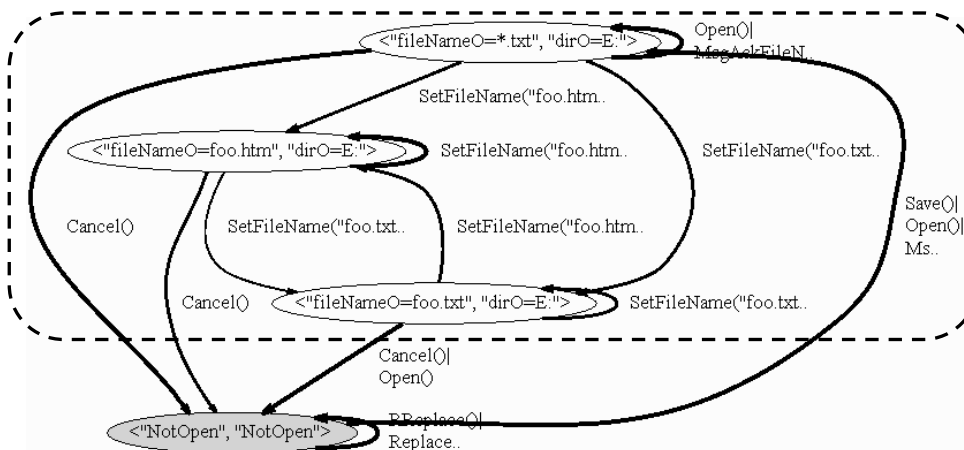


Figure 38: Open dialog view obtained from the projection onto the manipulated variables

4.2.3.3. How to obtain views showing currently enabled actions

Abstracting the behaviour outside each dialog produces a huge reduction in the number of states of the overall FSM. Even so, this may be not enough. In this case, it is possible to describe the system at a higher level of abstraction by distinguishing, for instance, the states where the set of available actions is different. This is helpful to check dependencies between interactive objects. In the case of the Find dialog inside the Notepad application, it is possible to see clearly

with this view that after filling the text inside the "Find What" text box, the "Find Next" button becomes enabled (Figure 39).

```
string FindCtrlsEnabledGroup { get {
    if (GetWindowWithFocus() == "Find") {
        if (FindDialog.findWhatF != "")
            return "Find Next enabled";
        else return "Find Next disabled";
    }
    else return "OutFind";
}}
```

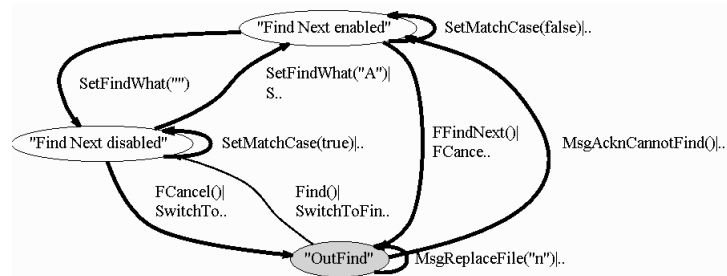


Figure 39: Changes in the set of enabled actions inside Find dialog

These views can also be used as test criteria for checking whether dependencies between interactive objects are correct.

4.2.4. Obtain complete models from navigation maps and dialog views

From the navigation map (containing the transitions between the windows/dialogs of the software application) and the dialog views (obtained by the projection of the state onto the manipulated variables of each dialog) it should be possible to obtain the complete FSM describing the software system. This means that this set of views describes completely the behaviour of the system.

These two views (navigation map and dialogs) can be seen as two different levels of abstraction of a hierarchical structure.

Recall the FSM state explosion problem. Hierarchical Finite State Machines (HFSM) cope with this problem. A HFSM is a FSM whose vertices represent single states or groups of states sharing a common characteristic (and transitions between the members of the group). These groups of states (and transitions) are themselves FSMs. Given a HFSM, it is possible to obtain a "flat" FSM by recursively substituting each group of states by its associated FSM.

A HFSM is well suited to model the behaviour of a GUI: the hierarchical structure of the HFSM can mimic the hierarchical structure of objects and dialogs of the GUI. For example, a GUI might have a main window with a top menu (possibly with sub-menus) allowing the user to open modal dialog windows. While a modal dialog window is opened, user interaction with all other currently open windows

of the same application is disabled. This very common structure can be modelled by a HFSM exhibiting one group of states per modal dialog. Whether not considering nested modal dialogs, each modal dialog can be seen as an independent FSM.

Such two views of the HFSM and the method to obtain the complete behaviour of the system from them will be formalized next.

The case of the navigation map and dialog views for test coverage criteria purposes will be explained in section 4.3.4.

4.2.4.1. Variables manipulated by each dialog

Recall that without restrictions, the state space of the application, S , is the Cartesian product of the sets of values which can be assigned to the variables manipulated by that application:

$$S = \text{dom}(v_1) \times \cdots \times \text{dom}(v_{|V|}). \quad (7)$$

Consider an application with two dialogs, D_1 and D_2 . From the complete FSM of the application, FSM_A , it is possible to obtain the subsets of FSM_A that describe each dialog FSM_{D_i} , by state grouping according to a criteria provided by the developer. For example, FSM_{D_i} could correspond to the group of states where dialog D_i is enabled (and the transitions among those states).

Having delimited the state machine FSM_D that describes the behaviour of a dialog D , it is possible to automatically deduce which variables are manipulated (read or written) by that dialog.

A variable v_i is written by (or is affected by) a dialog D if there is a transition in T_D (transitions of FSM_D) that changes the value of v_i [64]. Formally,

$$[\exists(s, a, s') \in T_D \cdot \pi_{v_i} s \neq \pi_{v_i} s'] \Rightarrow v_i \text{ is written by } D. \quad (8)$$

A variable v_i is read by (or influences the behaviour of) a dialog D if at least one of the following conditions holds [64]:

- there are two transitions t and t' in T_D and variable v , and v_k in V (not necessarily $i \neq k$) such that:
 - (i) the source states of t and t' are different only in the value of v_i ;
 - (ii) t and t' have the same triggering action (action name and arguments);
 - (iii) the destination states of t and t' have different values of v_k ; and
 - (iv) at least one of the transitions (say t) changes the value of v_k ;

- there are two states s and s' in S and a transition t in T_D such that:
 - (i) s and s' are different only in the value of v_i ;
 - (ii) the source of t is s ;
 - (iii) there is no transition t' with source s' and the same action as t .

Formally,

$$\left. \begin{array}{l}
 \left[\begin{array}{l}
 \exists t, t' \in T_D \cdot \pi_1(t) = s \wedge \pi_1(t') = s' \wedge \\
 \pi_{v_i} s \neq \pi_{v_i} s' \wedge (\forall j \neq i \cdot \pi_{v_j} s = \pi_{v_j} s') \wedge \pi_2(t) = \pi_2(t') \wedge \\
 \wedge \exists v_k \in V \cdot \pi_{v_k}(\pi_3(t)) \neq \pi_{v_k}(\pi_3(t')) \wedge \pi_{v_k} s \neq \pi_{v_k}(\pi_3(t)).
 \end{array} \right] \\
 \vee \\
 \left[\begin{array}{l}
 \exists s, s' \in S_D \cdot \exists t \in T_D \cdot \pi_1(t) = s \wedge \\
 \neg \exists t' \in T_D \cdot \pi_1(t') = s' \wedge \\
 \wedge \pi_2(t) = \pi_2(t') \wedge \pi_{v_i} s \neq \pi_{v_i} s' \wedge (\forall j \neq i \cdot \pi_{v_j} s = \pi_{v_j} s')
 \end{array} \right]
 \end{array} \right\} \Rightarrow \quad (9)$$

$\Rightarrow v_i$ is read by D .

Informally, this means that the response to user actions (and the actions available) in the context of dialog D depends on the value of v_i . In practice, this means that any implementation of dialog D must read (or query) the value of v_i when responding to user actions (or when determining which actions are available).

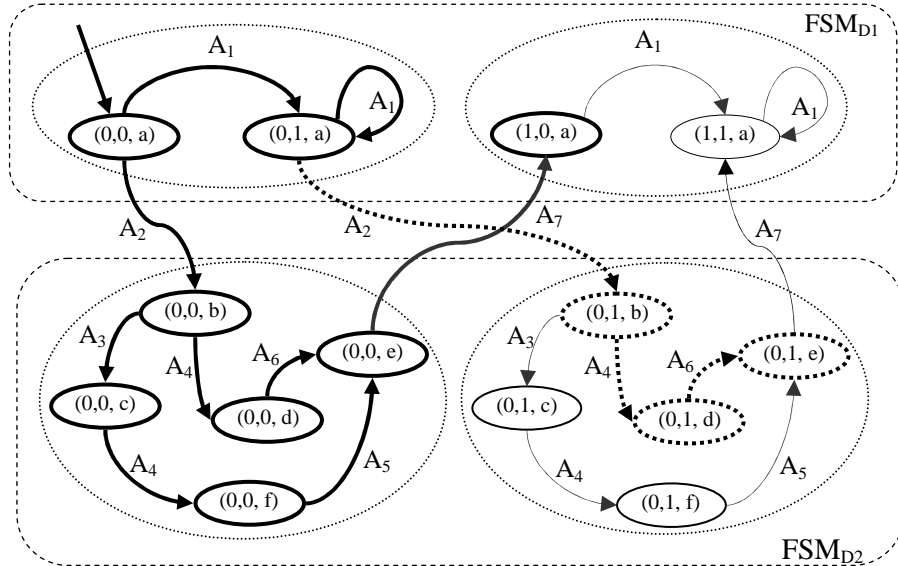


Figure 40: State machine of an application with dialogs D1 (action A1) and D2 (actions A3 to A6)

For instance, consider an application with state variables $V=\{v_1, v_2, v_3\}$, and two dialogs D_1 and D_2 with the behaviour described by the state machine of Figure 40. The state machine also includes transitions (labelled A_2 and A_7) that do not belong to any of the dialogs, but allow switching between them.

Dialog D_1 is enabled when $v_3=a$. Dialog D_2 is enabled when $v_3 \neq a$ and $v_1=0$. Given the transition $(0,0,a) \xrightarrow{A_1} (0,1,a)$ in D_1 , we conclude, by formula 8, that v_2 is written by D_1 . This is the only variable manipulated by D_1 . From transition $(0,0,b) \xrightarrow{A_3} (0,0,c)$ in D_2 and formula 8, we conclude that v_3 is written by D_2 . From transitions $(0,0,b) \xrightarrow{A_4} (0,0,d)$ and $(0,0,c) \xrightarrow{A_4} (0,0,f)$ in D_2 , we conclude, by formula 9, that v_3 is also read by D_2 . This is the only variable manipulated by D_2 .

4.2.4.2. HFSM structuring based on Variables Manipulated by each Dialog

Under certain conditions, there is a relationship between the state variables manipulated by each dialog D_i and the structure of the FSM of the application (FSM_A), that allow us to structure FSM_A into a HFSM (a sufficient condition is that the enabling condition of each dialog restricts the domain of each variable independently of the other variables).

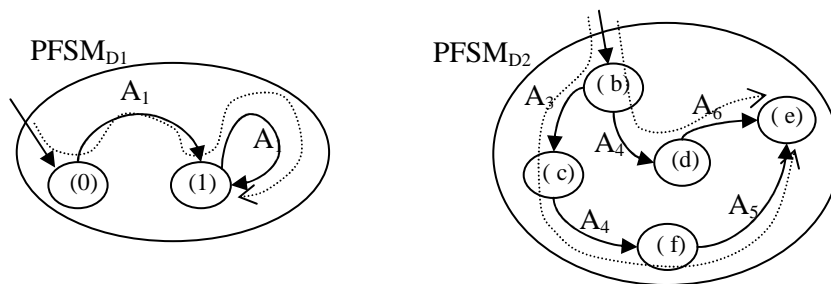


Figure 41: State machines of dialogs D1 and D2 projected from the FSM depicted in Figure 40. Dotted lines represent test cases

Let $PFSM_{D_i}$ be the projection of FSM_{D_i} onto the variables manipulated by D_i , as illustrated in Figure 41. $PFSM_{D_i}$ describes the behaviour of D_i .

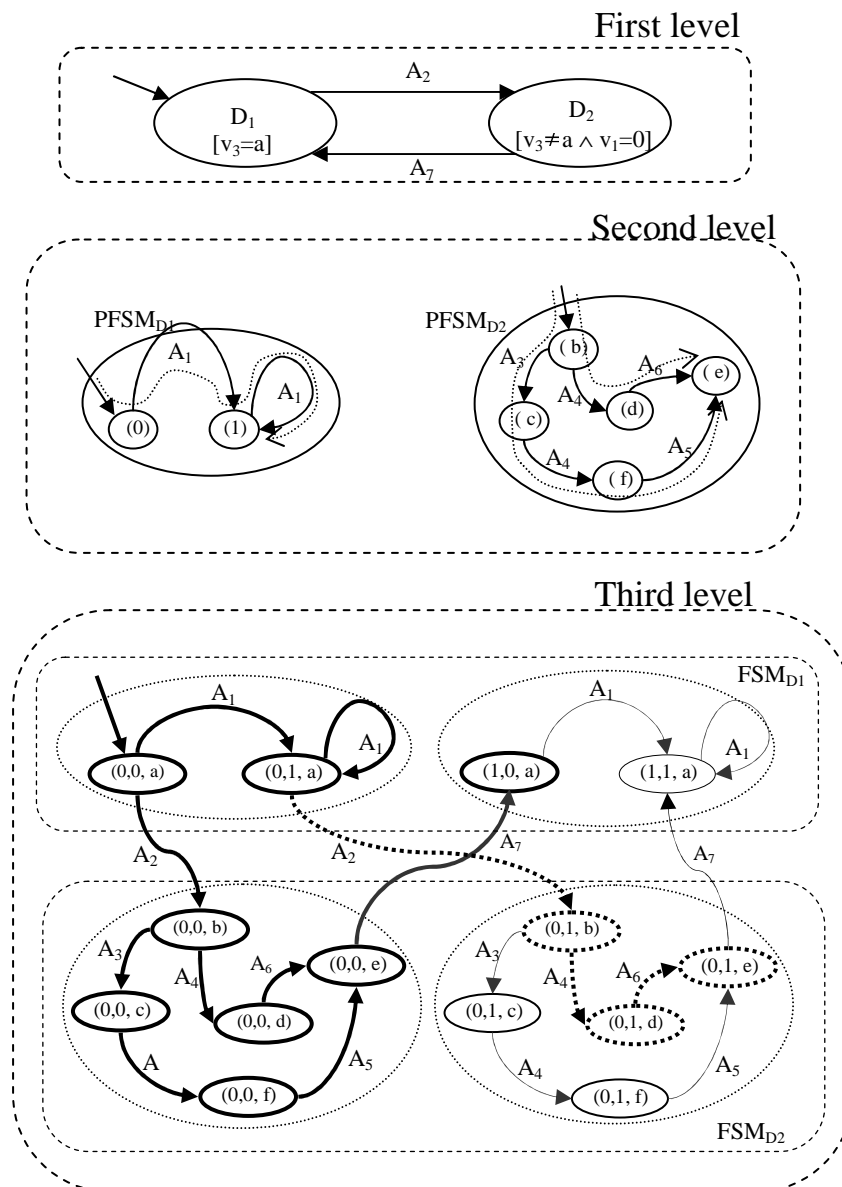


Figure 42: HFSM with three levels

In the opposite direction, FSM_{D_i} is the union of the instances of $PFSM_{D_i}$ for all possible combinations of values of the variables that are not manipulated by D_i (restricted to the enabling condition of D_i). For example, FSM_{D_1} (Figure 40) has 2 instances of $PFSM_{D_1}$ (Figure 41) with $v_1=0 \wedge v_3=a$, and $v_1=1 \wedge v_3=a$.

Given this, FSM_A can be organized into a 3-level HFSM, as illustrated in Figure 42.

4.2.4.3. Obtaining the complete FSM from the projections

Consider an application with m dialogs, $D = \{D_1, \dots, D_m\}$, manipulating a set of variables.

$$V = \{v_1, \dots, v_{|V|}\}. \quad (10)$$

Since there is a domain defined for each variable in V , the state space of the application, S , considering the restrictions imposed by the application, is a subset of the Cartesian product of the variables' domains:

$$S \subseteq \text{dom}(v_1) \times \dots \times \text{dom}(v_{|V|}). \quad (11)$$

Each dialog has an associated enabling condition, C_i , that restricts the set of states where the dialog i is enabled:

$$C_i : S \rightarrow \text{Bool}. \quad (12)$$

For all s in S , at least one enabling condition, $C_i \in \{C_1, \dots, C_m\}$, evaluates to true:

$$\forall s \in S \cdot \exists i \in \{1, \dots, m\} \cdot C_i(s). \quad (13)$$

The states in each dialog can be obtained by selecting the states of the application where C_i is true.

$$S_i = \sigma_{C_i} S, \quad (i = 1..m) \quad (14)$$

Also, the states of the system are obtained from the union of states of all dialogs.

$$\bigcup_{i=1}^m S_i = S. \quad (15)$$

Let $A = \{a_1, \dots, a_k\}$ be the set of actions which can be performed in this software application and $T \subseteq S \times A \times S$ the set of transitions whereby. The system can evolve from one state to another. For each dialog i , $1 \leq i \leq m$, we can compute the set of its internal transitions T_i .

$$T_i = \{(s, a, s') \in T \cdot s, s' \in S_i\}, \quad (i = 1..m) \quad (16)$$

For a GUI defined as explained above, it is possible to partition the system so as to obtain the navigation map that shows the transitions that switch between different dialogs of the system, and one view describing the behaviour of each dialog independently. The transitions, $T^{(1)}$, of the navigation map are given by:

$$T^{(1)} = \{(s, a, s') \in T \cdot C_i(s) \wedge C_j(s') \wedge i \neq j\}, \quad (i = 1..m, j = 1, \dots, m). \quad (17)$$

The states, $S^{(1)}$, of the navigation map can be obtained by:

$$S^{(1)} = \pi_1(T^{(1)}) \cup \pi_3(T^{(1)}). \quad (18)$$

Another partition of the system isolates the behaviour of each dialog independently. The transitions, $T_i^{(2)}$, of each dialog, i , shown in this view, are obtained by projecting the transitions (source and destination states) of dialog i onto the variables, v_i , manipulated by that dialog:

$$T_i^{(2)} = \{(\pi_{vi}s, a, \pi_{vi}s') \cdot (s, a, s') \in T_i\}, \quad (i = 1..m) \quad (19)$$

Accordingly, the states of each dialog i are obtained by projecting the states of the dialog onto the variables manipulated by that dialog i :

$$S_i^{(2)} = \{\pi_{vi}S_i\}, \quad (i = 1..m) \quad (20)$$

$T_i^{(2)}$ and $S_i^{(2)}$ correspond to the second level of the hierarchical structure shown in Figure 42, $PFSM_{Di}$.

From these two views, the navigation map (formula 17) and the dialogs (formula 19), it is possible to construct the entire system that corresponds to the third level shown in Figure 42:

$$T_i^{(3)} = \left\{ \begin{array}{l} (s, a, s') \in T_i \cdot (\pi_{vi}(s), a, \pi_{vi}(s')) \in T_i^{(2)} \wedge \\ \pi_{V \setminus vi}(s) = \pi_{V \setminus vi}(s') \end{array} \right\}, \quad (i = 1..m) \quad (21)$$

$$T = \bigcup_{i=1..m} T_i^{(3)} \cup T^{(1)} \wedge \text{there is a path from } S_{init} \text{ to } \pi_1(T_i^{(3)}). \quad (22)$$

In view of the fact that it is possible to construct the description of the full behaviour of the system from the description of the behaviour of the dialogs and the navigation map, they will be used as test coverage criteria for the generation of test cases to test the behaviour of the GUI. This is an interest test goal because these views still capture the requirements of the system and have a much lesser size than the complete FSM.

The following section analyses in more detail the size reduction that can be achieved when considering the second level of abstraction as a test coverage criterion instead of the third level.

4.2.5. Independent dialogs

Let's quantify the size reduction that is possible to achieve by considering the second level of abstraction instead of the third level as a testing goal. There are two different possible situations that result in two different size reductions: the case of independent dialogs and the case of dependent dialogs.

Independent Dialogs

Given two dialogs, if the set of variables written by one of the dialogs is disjoint from the set of variables manipulated by the other dialog, then they are independent. Informally, two dialogs are independent if the behaviour of any of the dialogs is not affected by the state and interactions that occur in the other dialog.

For example, dialogs D_1 and D_2 , in Figure 42, are independent.

The existence of independent modal dialogs allows us to reduce the number of states to consider. Assume we have an application with one main window, described by a FSM with m states, and k independent modal dialogs D_1, D_2, \dots, D_k that can be accessed from the main window, each D_i being described by a FSM with n_i states. If the dialogs were not independent (which could happen if they were modeless), the total number of states of the complete application would be the product $m \cdot n_1 \cdot \dots \cdot n_k$ (because a state of the application is a combination of states of the main window and the dialog windows). Since we assume that the D_i s are modal, only one dialog can be open at each time. Assume that, in the state machine that describes each dialog D_i , there is one distinctive state that represents the situation where the dialog is closed, and all the other $n_i - 1$ states represent situations where the dialog is open. The possible states of the application can be grouped as follows:

- a group representing the situation where all the dialogs are closed and only the main window is active; this group will have $m \cdot 1 \cdot \dots \cdot 1 = m$ states;
- for each dialog D_i , a group representing the situation where dialog D_i is open and all the other dialogs are closed; this group will have $m \cdot 1 \cdot \dots \cdot (n_i - 1) \cdot \dots \cdot 1 = m \cdot (n_i - 1)$ states.

Summing up, the total number of states of the application is $m \cdot (n_1 + \dots + n_k - k + 1)$.

In the case of an application with modeless dialog windows, a similar reduction of the number of states cannot be achieved, because any number of modeless windows can be open at the same time. But, if the behaviour of a modeless dialog window is not affected by the state of another modeless dialog window, they can be considered independent. Formally, given two dialogs, if the set of variables written by one of the dialogs is disjoint from the set of variables manipulated by the other dialog, then they are independent.

For testing purposes, it is not necessary to consider all the combinations of states of the different dialogs, as will be explained in the next section. Basically, it will suffice to fully test the behaviour of one dialog, for only one particular state of all the other dialogs (an instance). Roughly speaking, this corresponds to consider a reduced state machine similar to the one obtained in the case of modal dialogs, for testing purposes.

Dependent Dialogs

Two dialogs, D_1 and D_2 , are dependent of each other if they can be opened at the same time and manipulate non-disjoint sets of variables. This behaviour is illustrated graphically by Figure 43 where the states in each dialog result from the projection of the states onto the variables manipulated by each dialog. The illustrated situation means that there is at least one state in dialog D_1 that is possible to achieve by interacting with another dialog D_2 . In other words, in Figure 43, the user leaves dialog D_1 , from state s_1 , interacts with dialog D_2 , and when coming back to dialog D_1 it will be in a state, s_4 , different from the one in which he left the dialog, s_1 .

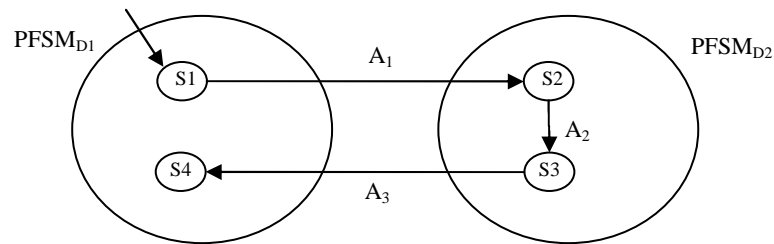


Figure 43: Dependent dialogs

Reducing the state machine with dependent dialogs as if it were a state machine with independent dialogs may remove the behaviour between dependent dialogs (actions A_1 and A_3 , in Figure 43). This is not desirable because this behaviour may be interesting for testing purposes. One way to overcome this situation is to consider both dialogs in one group. So, the possible states of the application can be grouped as follows:

- a group representing the situation where all the dialogs are closed and only the main window is active; this group will have $m \cdot 1 \dots 1 = m$ states;
- for each independent dialog D_i , a group representing the situation where dialog D_i is open and all the other dialogs are closed; this group will have $m \cdot 1 \dots (n_i - 1) \dots 1 = m \cdot (n_i - 1)$ states;
- for each set of dependent dialogs $\{D_i, \dots, D_j\}$ a group representing the situation where at least one of the dialogs of the set is opened and all the other dialogs that do not belong to the set are closed; this group will have $m \cdot 1 \dots [(n_i \dots n_j) - 1] \cdot 1 = m \cdot [(n_i \dots n_j) - 1]$ states.

Summing up, the total number of states of the software system is

$$m \cdot (n_1 + \dots + n_{i-1} - i - 1 + [(n_i \dots n_j) - 1] + 1).$$

4.3. Test Case Generation

Upon structuring the model as explained above, it is possible to reduce the FSM so as to maintain one instance of each dialog that corresponds to a particular state of all the other dialogs. This reduced FSM will in turn be used as input to a test case generation algorithm.

4.3.1. Overview of test case generation with Spec Explorer

Spec Explorer automatically generates test cases from a Spec# or AsmL specification in two steps (Figure 44). In the first step, a FSM is generated from the given Spec# or AsmL specification. In the second step, test cases that fulfil some coverage criteria are generated from the FSM.

The FSM is generated by bounded exploration of the state space of the model. Some techniques available in Spec Explorer to prune this exploration are:

- **state filters** – Boolean expressions that determine which states to explore. If the state does not satisfy the given filters then the transition to a new state is ignored;
- **additional pre-conditions** – definition of additional pre-conditions to limit the applicability of actions [191];
- **restriction of the domains** – the domains of actions' parameters are bounded to a finite set of possible values;
- **equivalence classes** – this technique partitions states into equivalence classes and prevents further exploration from any state of such a class once a specified number of representatives has been reached. The exploration algorithm can be configured so as to explore only n states in each state group with m states;
- **stop conditions** – conditions over states that stop the exploration once true;
- **scenarios** – allows substituting programmatically generated sequences of actions into the test cases produced by the Spec Explorer in places where a full exploration is not needed;
- **on-the-fly exploration** – combines test derivation from a model and test execution [192] into a single algorithm. This solves non-determinism by getting immediate feedback from the implementation and avoiding the pre-computation of the possible huge test case with all possible responses of the system under test.

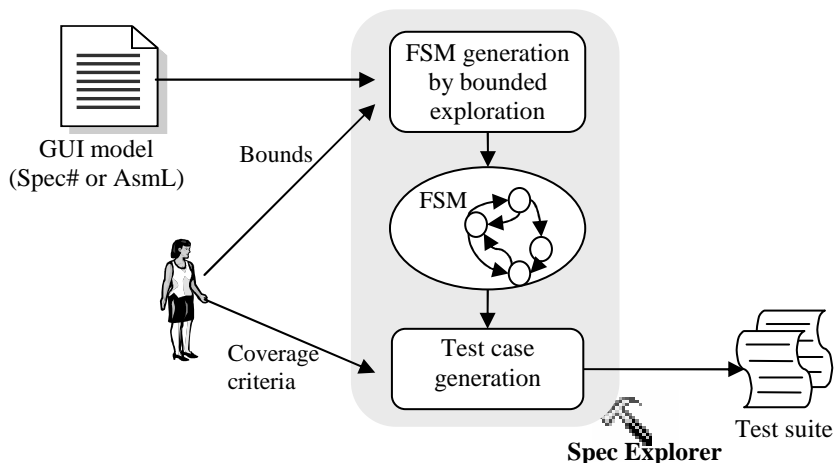


Figure 44: Test case generation

The pruning of exploration becomes crucial when talking about modelling and GUI testing. This is because testing an application through its GUI by simulating user events entails a significant overhead and results in much slower test

execution than testing an application through its API. The main challenge is to generate a test suit of manageable size while still guaranteeing adequate testing.

As soon as the FSM is constructed, and the coverage criteria chosen, a traversal engine is used to unwind the resulting FSM to produce behavioural tests that fulfil the coverage criteria. The coverage criteria can be set to:

- **Full Transition Coverage:** the test suite generated covers all transitions of the FSM. In addition, this algorithm can be configured so as to generate test segments/paths that whenever possible return to the initial state, and can also be pruned so as to generate test segments bound by a given number of transitions;
- **Shortest Path:** the test suite generated is the shortest path (sequence of transitions) that reaches a specified goal state;
- **Random Walk:** generates a test suite with a single sequence of invocations. At each state, one of the outgoing transitions is randomly selected.

Actions known as *probes* are checked in every state of the resulting tests, and do not take part in coverage considerations.

4.3.2. Domain definition

Spec Explorer does not provide support for the definition of the domains of the parameter actions. This has to be done manually by the user and it is a crucial point in the testing process. Domains have deep impact in the FSM generated by exploration of the model. A random definition of the domains may result in a FSM that does not have relevant properties from the testing point of view. Generating test cases from a FSM like this will not be very useful because it will not test some relevant properties or, in the worst case, it will not test anything useful.

The high level scenarios, described in section 4.2.2, identify the main functionalities of the Notepad application. They describe the requirements of the application and can be used for requirement coverage analysis. One way to do it is to apply structural coverage analysis on the scenarios' descriptions in order to determine the domains' variables needed to achieve their full coverage. There are different types of structural coverage criteria: statement coverage, decision coverage, condition coverage, condition/decision coverage, modified condition / decision coverage (MC/DC), and multiple condition coverage [91]. The scenarios identified are analysed so as to determine which domains to associate to the variables to use. The criterion used in this analysis is a generalization for non-Boolean variables of the MC/DC criterion by showing that each input variable affects independently the functionality under test. The result of analysing the scenario of Figure 33 with such coverage criterion, which we call "full coverage of functional dependencies", is summarized by Table 1.

The variable domains needed to test the open and save effects described by scenario in Figure 33 are the rows with grey shading in Table 1. This set of rows show that each condition affects independently the outcome of the decision (save

or open). The MC/DC criterion needs a minimum of $n+1$ test cases for a decision with n input variables. In this case (Table 1), $n+2$ test cases are needed because *saveChanges* can be set to three different values (Y , N , or C).

Domains are necessary to produce all the effects identified by the scenario but are not sufficient. The FSM generated from the atomic actions may not explore states or intermediate states that would be needed to produce the complete desired effect. So, after defining the parameters' domains that are used in the generation of the FSM (without scenarios), it is necessary to check if the FSM generated has the properties considered relevant from the testing perspective. This is a process of model validation that should precede the test case generation activity.

Inputs					Effect	
dirty	Exists(fileToOpen)	saveChanges	Exists(fileToSave)	overwrite	Saved?	Opened?
T	T	Y	-	T	T	T
T	T	Y	F	-	T	T
T	T	Y	T	F	F	T
T	T	N	-	-	F	T
T	-	C	-	-	F	F
F	F	-	-	-	F	F
T	F	N	-	-	F	F
T	F	Y	-	T	T	F
T	F	Y	F	-	T	F
F	T	-	-	-	F	T

Table 1: Conditions for testing the save and open effects inside the Open scenario

4.3.3. Test coverage and adequacy criteria on the FSM

The definition of a good test criterion is important for scalability purposes. Executing all possible test cases during software testing is not realistic due to the number of test cases, meaning that we need to select test cases. Test coverage and adequacy criteria are a set of rules that guide the generation of a test suite determining when to stop the generation, whether enough testing has been performed or further tests are needed, and provide an objective measure of the test suite quality (adequacy for testing the software system). An ideal test criterion would be capable of generating the smallest test suite that could find (if not all) the maximum number of errors of a software system.

Spec Explorer provides a set of test coverage criteria to construct a test suite from the generated FSM (transition coverage; shortest path; and random walk). However, if the FSM from which test cases are generated does not have the

desired properties, these criteria can compromise the quality of the tests. A proper choice of the parameters' domains is crucial but is not sufficient to assure a generation of a good FSM. For example, it could be possible to have actions that are allowed only in specific states (where pre-condition holds) that are not possible to reach because the model does not allow it (model error), or because it is not possible to reach them within certain time limits or within certain memory limits. So, adequacy criteria must be defined on the FSM to evaluate its quality (model validation) in terms of relevant properties from the testing perspective.

There are several adequacy criteria that could be used to evaluate the quality of the generated FSM (from the previously defined domain's variables): specification coverage; scenarios; functional dependencies; special case situations; and projections of the state machine.

Some of these criteria can be easily checked with the current functionalities of the Spec Explorer, others could be easily implemented as extensions to the tool like, for instance, adding model checking techniques.

Specification coverage

Specification coverage criteria aim to evaluate if the generated FSM covers the specification. This corresponds to applying white-box techniques on the specification that are traditionally applied on code.

It is possible to define coverage criteria to cover more or less detailed aspects of the specification. The minimum required specification coverage criterion would be to assure that all the actions in the specification are within the generated FSM.

Other coverage criteria exist which aim, for instance, at covering all statements or conditions within the specification. One way to evaluate these coverage criteria with the current functionalities available in the Spec Explorer tool would be to change the specification so as to construct one action for each of the statements or conditions, with appropriate pre-conditions, and check if those actions are within the generated FSM.

Scenario coverage

Scenario coverage criteria aim at evaluating if the generated FSM covers all possible statements and branches in the specified high level scenarios that describe the main functionalities of the system. One way to perform this check is to specify FSM views so as to describe each scenario independently. Such views show the windows and dialogs with which the user interacts with along the described scenario. Then, each of those views are analysed (inspected visually) in order to evaluate if all possible paths described by the parameterized scenario are present.

For instance, one way to check if the scenario described in Figure 33 is present in the FSM generated by the atomic actions is to build the view in Figure 45.

```

string OpenScenarioGroup { get{
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (IsEnabled("MsgSaveChanges") && svBfrOpen)
        return "MsgSaveChanges";
    else if (IsEnabled("Save") && svBfrOpen) return "Save";
    else if (IsEnabled("MsgAckFileNotFound"))
        return "MsgAckFileNotFound";
    else if (IsEnabled("MsgOverwriteFile") && svBfrOpen)
        return "MsgOverwriteFile";
    else if (IsEnabled("Open")) return "Open";
    else if (dirty) return "Dirty";
    else return "NotDirty";
}}

```

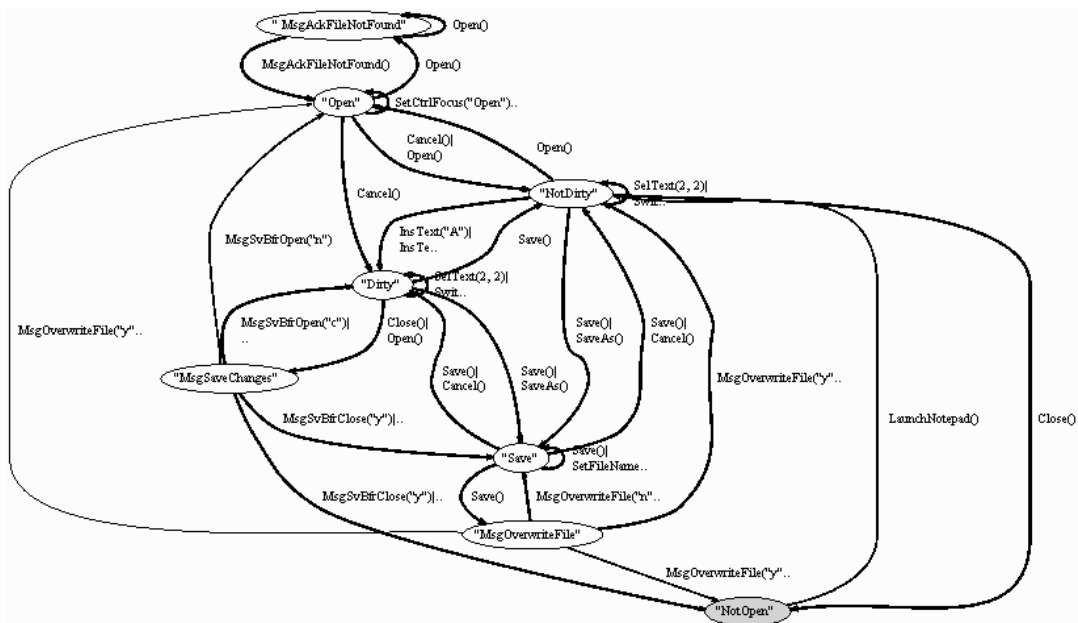


Figure 45: Open scenario view

Another way to check if the scenario is present in the generated FSM, but currently not supported by Spec Explorer, would be to express all different paths of the scenario as high level temporal logic formula and use model checking techniques to produce counter-examples showing that those paths are within the model.

```

~E[◇(IsEnabled("SaveChanges") ->◇(IsEnabled("Open")))]
~E[◇(IsEnabled("SaveChanges") ->◇(IsEnabled("Save")))]
~E[IsEnabled("SaveChanges") ->
    ◇(IsEnabled("Save") ->◇(IsEnabled("Open")))]
...

```

Since the high level temporal logic properties are negated, should the model checker find a counter-example for each of them then the scenario is fully within the generated FSM.

Functional dependencies coverage

Functional dependency coverage criteria aim to evaluate if the generated FSM covers all functional dependencies needed to show that all variables affect independently the behaviour of the system. This coverage criterion is a generalization for non-Boolean variables of the MC/DC criterion. The tables constructed throughout the domain's definition may be used as a base to perform this check.

Special cases coverage

Special case coverage criteria aim at evaluating if the generated FSM covers all the identified boundary test conditions. Boundary test conditions correspond to situations near limits of valid ranges where errors are most likely to occur. Some of these situations may be covered only by huge FSMs and sometimes it may be useful to define scenario actions to drive the application into such states, goal states, where those boundary situations happen as a way to reduce the required FSM size needed to cover them.

One way to check if the special cases are present in the FSM generated by the atomic actions is to define different views of the model expressing those situations as FSMs.

An example of a boundary test condition related to the *find* functionality inside the Notepad application can be: "the cursor's position is in the middle of the word to look for". This can be expressed in Spec# as:

```
string AtTheMiddleGroup { get {
    if (Exists{ i in Set{0..text.Length};
        posCursor>i && posCursor<i+findWhat.Length &&
        i==text.IndexOf(findWhat)})
        // IndexOf reports the index of the
        // first occurrence in this instance of
        // the findWhat word
        return "InTheMiddle";
    else return "NotInTheMiddle";
}
```

and visualized in the Figure 46.

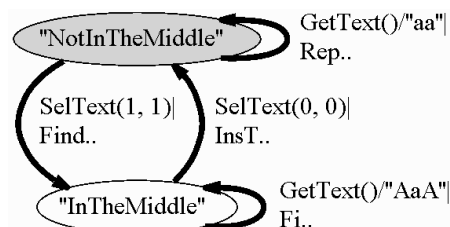


Figure 46: Coverage analysis of a special case condition

State machine projection coverage

State machine projection coverage criteria evaluate if the generated FSM covers relevant projections of the system. One of these projections is the one that describes the independent behaviour of the dialogs within the application and the navigation map. The algorithm developed to reduce the complete FSM while guaranteeing this coverage degree will be the subject of the following section.

These views are of interest as testing goals because they still capture the requirements of the system and have a much lesser size than the complete FSM.

4.3.4. FSM reduction

A pruning technique, based on the state machine projections criteria, was added to Spec Explorer to reduce the size of FSMs obtained from GUI models [151]. The FSM is organized in a hierarchical structure (as illustrated by Figure 42) that is used as input to the FSM reduction algorithm. Firstly, independent dialogs are identified and highlighted in a HFSM built from the FSM. Then, the portion of the FSM that describes each dialog is reduced. Spec Explorer generates test cases from the reduced FSM, and tests the conformity between the specification and the implementation. To evaluate the conformity between a specification and an implementation/GUI, additional functionalities must be developed to observe the GUI updates resulting from the interaction. The `GetText()` method to observe/read the text in a textbox is one example of those functionalities (see Figure 29 at page 96).

Using the transitions' state coverage criteria to generate test cases from the state machine of Figure 40 (with 18 transitions) we would get 4 test cases (paths) with 21 steps:

$$\begin{aligned}
 &(0,0,a) \xrightarrow{A_2} (0,0,b) \xrightarrow{A_4} (0,0,d) \xrightarrow{A_6} (0,0,e) \xrightarrow{A_7} (1,0,a) \xrightarrow{A_1} (1,1,a) \xrightarrow{A_1} (1,1,a) \\
 &(0,0,a) \xrightarrow{A_2} (0,0,b) \xrightarrow{A_3} (0,0,c) \xrightarrow{A_4} (0,0,f) \xrightarrow{A_5} (0,0,e) \\
 &(0,0,a) \xrightarrow{A_1} (0,1,a) \xrightarrow{A_1} (0,1,a) \xrightarrow{A_2} (0,1,b) \xrightarrow{A_4} (0,1,d) \xrightarrow{A_6} (0,1,e) \xrightarrow{A_7} (1,1,a) \\
 &(0,0,a) \xrightarrow{A_1} (0,1,a) \xrightarrow{A_2} (0,1,b) \xrightarrow{A_3} (0,1,c) \xrightarrow{A_4} (0,1,f) \xrightarrow{A_5} (0,1,e)
 \end{aligned}$$

Since D_1 and D_2 are independent dialogs, they don't need to be tested every time variables on which they don't depend change. Only one instance of each dialog needs to be tested. To test dialog D_i , the values of the variables that are not manipulated by D_i are fixed to a particular value, and the transitions' state coverage criteria is applied to the PFSM of D_i to generate test cases. For example, to test D_1 we could fix $v_1=0$ ($v_3=a$ is already fixed) and generate the test case illustrated by the dotted line in Figure 41. To test D_2 we could fix $v_2=0$ ($v_1=0$ is already fixed). With this approach, only 7 transitions are exercised, instead of 21. The instances of D_1 and D_2 that are tested are the ones shown on the left-hand side of Figure 40.

To fully test the application, actions that do not belong to these dialogs, also have to be exercised. This is the case of actions A_2 and A_7 in Figure 40. Applying the same approach to each of these actions (each one can be regarded as a dialog with

a single action), we conclude that only one instance of each action need be tested in this case. For example, we can exercise (test) the instances of A_2 and A_7 shown as thick lines in Figure 40. Overall, the transitions that need be exercised are all the transitions shown as thick lines in Figure 40. Three test cases (paths), with a total of 10 steps, are enough to cover them. So the size of the test suite is reduced from 21 steps to 10 steps:

$$\begin{aligned}
 &(0,0,a) \xrightarrow{A_1} (0,1,a) \xrightarrow{A_1} (0,1,a) \\
 &(0,0,a) \xrightarrow{A_2} (0,1,b) \xrightarrow{A_4} (0,0,d) \xrightarrow{A_6} (0,0,e) \xrightarrow{A_7} (1,0,a) \\
 &(0,0,a) \xrightarrow{A_2} (0,0,b) \xrightarrow{A_3} (0,0,c) \xrightarrow{A_4} (0,0,f) \xrightarrow{A_5} (0,0,e)
 \end{aligned}$$

In some cases, it is not sufficient to test only one instance of each dialog. After assuring that one instance is fully tested, a second instance may have to be traversed (usually only in part, by the shortest path) in order to reach some state or transition that has to be exercised. For example, assume that, with respect to Figure 40, it is important to reach state $(0,1,e)$, because it is the source of a transition that has not been tested yet (not represented in Figure 40). In such case, the path shown by the dotted lines of Figure 40 also has to be included in the test suite.

In order to explain this FSM reduction algorithm, consider,

- S – set of all states of the software application;
- D_i , where $1 \leq i \leq m$ – dialog i (first level of Figure 42);

The algorithm starts by selecting one instance to test, I_{TTi} , for each dialog/window, i . Each dialog can have different instances that correspond to different values for the non-manipulated variables of that dialog. In Figure 40 it is possible to see that dialog D_i has two instances that correspond to two different values for the non-manipulated variable (the first one) namely 0 and 1. The set of all instances of one dialog i , I_i , can be obtained by projecting its states, S_i , onto the variables non-manipulated by that dialog ($V \setminus v_i$).

$$I_i = \pi_{V \setminus v_i}(S_i), \quad (i = 1..m) \quad (23)$$

The instance to test is selected from the set I_i and corresponds to fixing the value of the non-manipulated variables

$$I_{TTi} \in I_i, \quad (i = 1..m) \quad (24)$$

and then calculating the states to test in each dialog i ($STTi$), which are given by

$$STTi = \{s \in S_i \cdot \pi_{V \setminus v_i}(s) = I_{TTi}\}, \quad (i = 1..m) \quad (25)$$

The states not to test (the excluded states) in dialog i are given by,

$$SNTTi = S_i \setminus STTi, \quad (i = 1..m) \quad (26)$$

The states that do not belong to any dialog are the states of the main window. To ensure that those states are not excluded from the FSM another step is performed in the algorithm: it selects all states of the main window given by

$$S \setminus \left\{ \bigcup_{i=1..m} STT_i \cup \bigcup_{i=1..m} SNTT_i \right\} \quad (27)$$

and ensures that there is a path to each of those states by calculating the minimum path to reach them from the starting state. All states that are traversed by these paths are added to the set of states to test (*STT*). It may be possible to add states that were previously in an instance not to test.

It is important to have in mind that the exploration process may be stopped by the user before ending. This means that the instances of the dialogs may not be completed. So, instead of selecting randomly an instance to test, it is important to test an instance, *k*, with maximum number of states, that is, which obeys

$$\forall j = 1..|I_i|, j \neq k \wedge \#\{s \in S_i \cdot \pi_{V \setminus v_i}(s) = I_j\} \leq \#\{s \in S_i \cdot \pi_{V \setminus v_i}(s) = I_k\} \quad (28)$$

Once FSM is reduced, an algorithm to calculate the test suite may be applied. In general, the selection of sequences ensuring that all of the application's behaviour is exercised, is a problem as hard as deciding the reachability of a state. Partial order reduction (POR) techniques used in model checking [156] address a very similar problem: Given a property of the system, e.g., a temporal property describing the reachability of a state, POR reduces the number of states that must be explored in order to decide whether the property holds for the entire state space. POR exploits redundancies of the state space like the commutativity of enabled transitions.

4.4. GUI Mapping Tool

As already mentioned, to perform conformance tests with Spec Explorer, a binding or mapping between the model actions and implementation methods in a .NET assembly must be provided. When the implementation is a .NET application, the mapping can be easily established since the model is written in a .NET language as well. For APIs exposed by other means, some glue code might be needed to map forth and back the data and method calls. However, when the application's functionality is only exposed through its GUI, then the application must be driven through the GUI's abstraction layer, by simulating the actions of a user interacting with it.

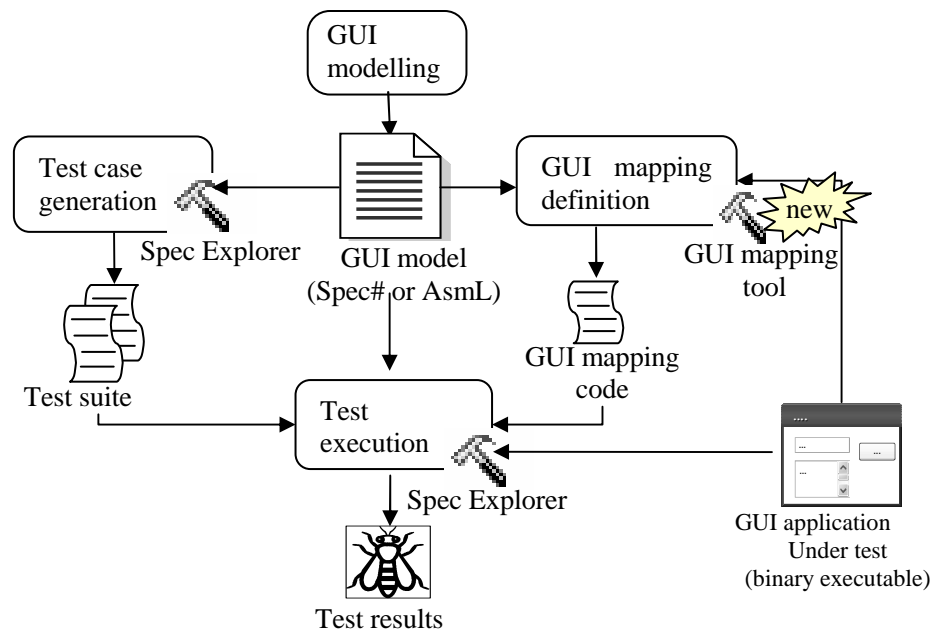


Figure 47: GUI modelling and testing process

In previous experiences of using Spec Explorer to model and test GUI applications [151], it was realised that, even in the case of simple applications such as Notepad, the manual building of the GUI mapping code, the code that maps forth and back the data and method calls, was unpractical and required too much effort. To solve that problem, a GUI Mapping Tool was developed and integrated with Spec Explorer (see Figure 47).

The GUI Mapping Tool assists the user in relating the model actions ("logical" actions) to "physical" actions on "physical" GUI objects. A major difficulty solved by the tool is the identification of the GUI physical objects that the model actions refer to. The mapping code is automatically generated from high-level mapping information and methods of the intermediate code are automatically bound to related modelled actions of the specification. After all these steps, test cases can be finally generated and executed and inconsistencies between the specification and the implementation are reported. Further information about this tool will be provided in the sections which follow.

Model-to-implementation mapping with the GUI Mapping Tool

The aim of the GUI Mapping Tool is to reduce the manual work involved in model-based testing of software applications through their GUI.

As already mentioned above, the GUI Mapping Tool assists the user in relating the logical actions described in the model to physical actions on physical GUI objects of the application under test (AUT). This tool (Figure 48) has a front-end (Figure 49) that shows the mapping information gathered so far and gives access to the GUI Spy tool and the GUI Mapping Code Generator. The Spy tool is used to get information about physical GUI objects in the AUT, in a way similar to the

Spy++ tool that ships with Microsoft Visual Studio. The code generator exports mapping information to XML files and C# the mapping information gathered. The C# code generated is based on calls to a reusable GUI Test Library. Further details will be provided in the sequel.

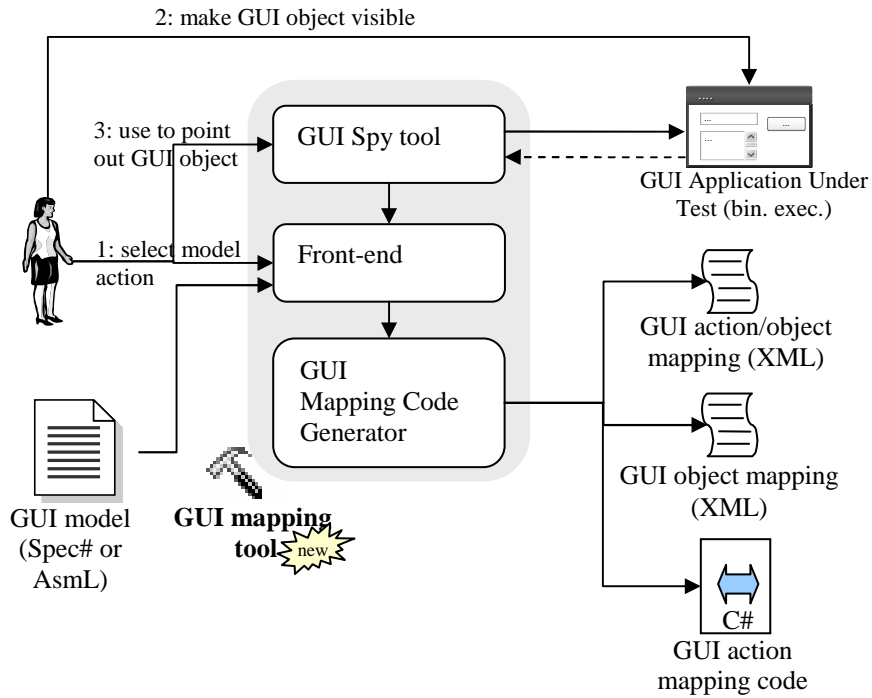


Figure 48: Architecture of the GUI Mapping Tool

The GUI Spy tool

The GUI Spy Tool is accessible from the front-end of the GUI Mapping Tool (see Figure 49). It allows the user to point out the physical GUI object that is the target of each logical action specified in the model.

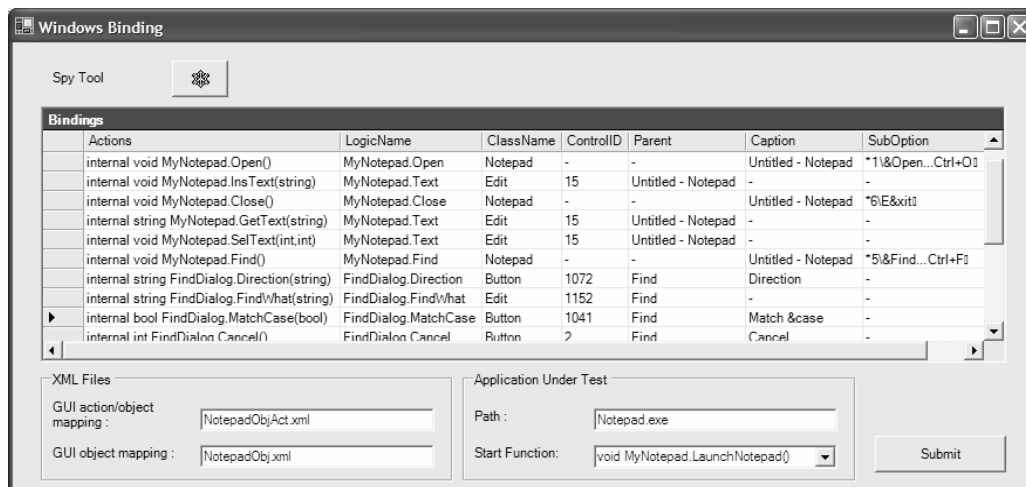


Figure 49: Front-end of the GUI Mapping Tool

After selecting the logical action in the main grid (first column), the user drags and drops the spy icon on top of the corresponding physical GUI object in the AUT. If the desired GUI object is not visible, the user will have to interact also with the AUT in order to make it visible. The physical properties of the GUI object selected, as well as a logical name inferred by the tool (to be explained later on), are then displayed in the grid (see Figure 49).

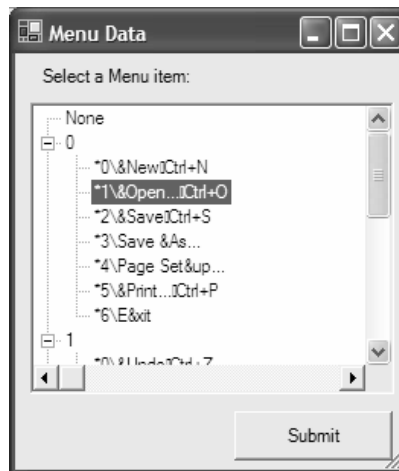


Figure 50: Selection of menu options

The Microsoft Visual Studio Spy++ tool can only gather information about proper windows (or GUI objects with a window handle). Our tool goes a bit further: it can also gather information about window menus. So, testers wanting to establish a relation between a specification method and an item inside a menu, can drag and drop the mouse on top of the window that contains the menu at which time another window (at Figure 50) is opened with all the submenu options, allowing then to choose submenu options ("SubOption" column of Figure 49). A similar option exists for controls such as tab pages and toolboxes.

Logical names of GUI objects

Every physical GUI object is associated to a logical name. This keeps specification and implementation levels independent and allows the generation of code more readable and easier to construct manually, if desired.

Default logical names are automatically generated by the tool. The logical name is equal to the namespace name followed by the name of the specification method without prefix (Set, Get, etc.). In order to obtain the same logical name for all the logical actions with the same target physical object, it is desirable that the names of those actions are constructed with a different prefix and the same suffix.

XML files generated

The mapping information captured is saved into two XML text files:

- a file with the mapping between model actions and the logical names of the target GUI objects (GUI action/object mapping file in Figure 48);

```
<Action id="internal void MyNotepad.Open()">
  <LogicalName>MyNotepad.Open</LogicalName>
</Action>
```

- a file with the mapping between logical names and physical properties of GUI objects (GUI object mapping file in Figure 48).

```
<GUIObject logicalName="MyNotepad.Open">
  <ClassName>Notepad</ClassName>
  <Caption>Untitled - Notepad</Caption>
  <SubClassName>menu</SubClassName>
  <SubOption>&Open...Ctrl+O</SubOption>
</GUIObject>
```

The mapping information needs to be gathered just once for each application. But if the specification is changed and the mapping information has to be updated, the XML files can be loaded by the GUI Mapping Tool for updating. The XML files can also be changed directly by the user.

These XML files are also used for code generation and test execution, as is explained in the sequel.

GUI Test Library

The C# code generated is based on calls to a reusable GUI test library that provides methods to simulate the actions of a user interacting with a GUI application and observe the content of GUI objects. This library was constructed in C# extending a previous existing library to best fit the needs.

The GUI test library provides three kinds of methods (Figure 51):

- methods that act upon GUI objects simulating the user, like sending text to a control that accepts text input (`SendText`). The target GUI object is identified by its logical name. Each method may have additional parameters with information needed to perform the action.
- methods that observe properties of GUI objects, like the text (`GetText`), insertion point (`GetInsertionPoint`), and selected text (`GetSelectedText`) of a text box. The target GUI object is also identified by its logical name. The return value conveys the information requested.
- methods that provide physical information about GUI objects identified by their logical names in order to identify those objects in the real AUT. This information may be loaded from a XML file.


```
// To act upon GUI objects
void Click(string GUIObjName);
void SendText(string GUIObjName, string txt);
void SelectText(string GUIObjName, int start, int end);
void SelectSubOption(string GUIObjName, string option);
void SelectCheckBox(string GUIObjName, bool check);
void SelectListIndex(string GUIObjName, int index);
void SelectMsgBoxOp(string GUIObjName, string option);

// To observe properties of GUI objects
string GetText(string GUIObjName);
string GetSelectedText(string GUIObjName);
int GetInsertionPoint(string GUIObjName);
bool GetCheckBox(string GUIObjName);
int GetListIndex(string GUIObjName);

// To map logical object names to physical objects
void LoadXMLObjMapping(string XMLFileName);
```

Figure 51: Examples of methods implemented in the GUI test library

Rules for mapping logical actions into physical actions

Besides identifying the physical GUI object that is the target of each model action, it is also necessary to select the appropriate method from the GUI test library, which will simulate a physical action of the user on that GUI object.

The GUI Mapping Tool automatically infers the appropriate library method based on the type of the GUI object, and the signature of the model action.

Some required heuristic rules are:

- When the sub option is filled in the mapping information, it is assumed that the logical action is modelling the action of a user selecting a sub menu option, a tab option or a tool button inside a toolbox (`SelectSubOption` method in the test library). This is the case of actions `Open`, `Close` and `Find` in Figure 49.
- When the logical action is an inspection method, has a string as return value and is mapped to a textbox, it is assumed that it is modelling the eyes of the user looking at the content of the textbox, thereby retrieving the text (`GetText` method in the test library). This is the case of action `GetText` in Figure 49.
- When the logical action's name has `set` as prefix, is mapped to a textbox, and has one parameter of string type, it is assumed that it is modelling an action that replaces the content of the related textbox with the contents passed in the parameter.
- When a logical action has a string parameter and is mapped to a textbox, we assume that the action is modelling an event that sends text (`SendText` method in the test library). This is the case of actions `InsText` and `FindWhat` in Figure 49.

- When the prefix of the modelled action's name is `msg` and the logical action has one parameter of string type, it is assumed that the specification action is modelling the interaction with a message box window by pressing the specific button that has the caption passed in the parameter.
- When the prefix of the modelled action's name is `ack`, it is assumed that the specification action is modelling the physical action of pressing the button of an acknowledge message box.
- When the logical action has neither parameters nor return value, and is mapped to a button, we assume that physical action is to click the button (`Click` method in the test library). This is the case of action `Cancel` in Figure 49.
- When the logical action is mapped to a `ComboBox` and has one parameter of type `int`, it is assumed that it is modelling an action that selects the item from the list of items in the position given by the parameter.

Code generation

Spec Explorer requires actions in the model to be bound to implementation methods (in a .NET assembly) with identical signatures (identical return type, number of parameters, and parameters' types). To fulfil this requirement, the tool generates C# code with methods with the same signature as the model actions, as illustrated in Figure 52. For each logical action, a method is generated with the same signature, calling the method of the GUI Test Library inferred according to the rules described before, with the logical name of the target GUI object as additional parameter.

```
#region automatically generated code
class GeneratedCode{
public static void LaunchNotepad(){
    LoadXMLObjMapping("C:\\temp\\Notepad.xml");
    new App(@"Notepad.exe");
}
public static void Open(){
    UserEvents.SelectOption("Notepad.Open");
}
public static void InsText(string p0){
    UserEvents.SendText("Notepad.Text",p0);
}
public static string GetText(){
    return UserEvents.GetText("Notepad.Text");
}
//...
}
#endregion
```

Figure 52: Excerpt of the code generated automatically for the Notepad example

The start function launches the application and reads the mapping information between logical and physical GUI objects from the GUI object mapping XML file (in Figure 48). Every function has one parameter with the logical name of the interactive object where the action will take effect and possibly other parameters with data needed for the action, e.g., text to send to a textbox.

Only one instance of the AUT should be opened when executing the test cases. Otherwise, the tool can pick the wrong window thus compromising the test cases. This problem can be partially solved by generating test cases that return to the initial state. However, when a specific path does not run till the end, for instance, because an error was detected, it may leave windows of the application opened. To overcome this problem, some code is added manually to the start method (`LaunchNotepad`) to close all windows that were opened by the previous testing trace/path.

Test execution

As soon as the mapping code is available and compiled into a library, a reference to this library is added to the Spec Explorer project, and the test cases are generated, it is possible to execute the test cases autonomously without user intervention.

Let's assume we have a deterministic model. Then, each test case consists of a sequence of steps. For each step, a specification action and its related implementation method are executed in lock-step mode (e.g., the `Close()` method in Figure 53). At the implementation level, each method does a call to a method defined in the generic GUI test library (e.g., `Click()` in Figure 53) that interacts with the GUI AUT simulating the user actions. The query actions (with the `Get` prefix) get information about interaction object properties that are compared with the expected values obtained from the specification. Whenever inconsistencies are detected, they are reported.

In GUI testing, inconsistencies between specifications and implementations can rise for several reasons:

- the model is trying to act on a control that is not enabled or cannot be found;
- the model is trying to act on a window that is not reachable or is not opened (e.g., a modal dialog is open and the window we want to reach is behind that dialog);
- the expected result was not displayed (e.g., a textbox does not display the expected content).

The path that gave rise to the error must be analysed to infer the actual reason for the error to happen.

While testing Notepad, we discovered two sequences of actions which lead to an inconsistency between our intuitive model and the actual Notepad application:

1. Type text.
2. Search for text using the find dialog (Ctrl-F). Close the dialog.
3. Open the replace dialog (Ctrl-H). Close the dialog.
4. Press the F3 key (shortcut for "Find Next").

Notepad will search upwards instead of downwards.

1. Type text, for instance, "aaa".
2. Search for text (e.g., "a") using find dialog (Ctrl-F) in upward direction. Close the dialog.
3. Open the find dialog (Ctrl-F) and close it immediately (press Cancel button).
4. Press the F3 (shortcut for "Find Next").

Notepad will search downward instead of upward as expected.

These are sequences of events that manual test would probably miss since they aren't common sequences of events.

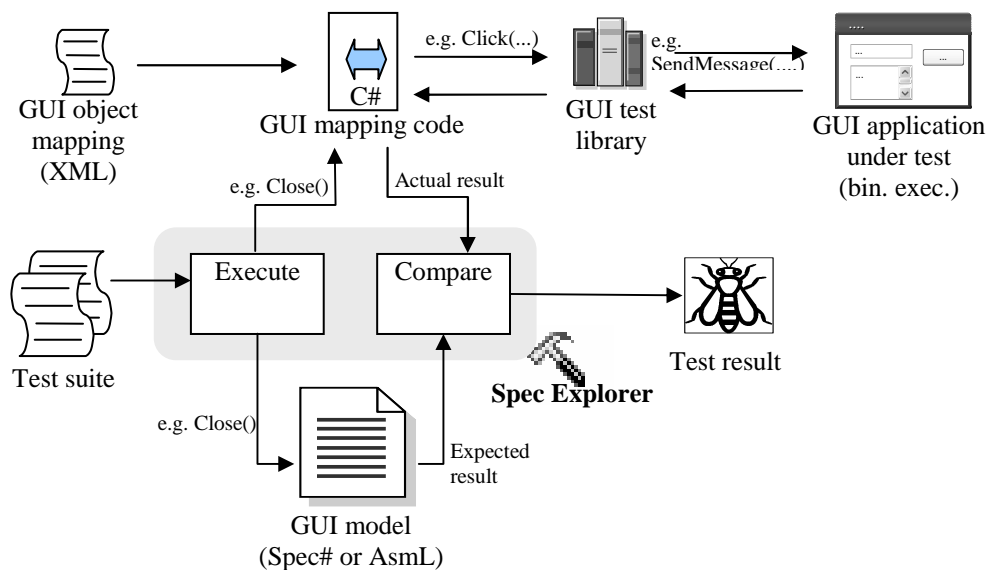


Figure 53: Test execution

4.5. Conclusions

This chapter presented the main contributions of our work, namely, an approach to model GUIs with HFSMs and to generate test cases from such models in an optimized way, taking advantage of the hierarchical structure.

The Spec# specification language, developed by Microsoft Research based on Abstract State Machines, is used to construct the model of the application. This model was converted automatically into a FSM using the Spec Explorer tool

which is a model-based testing tool also developed by Microsoft Research. With the definition of expressions to construct state groups, it was also possible to structure the model into a HFSM. This tool is also used to generate test cases and execute them to perform conformity checks between a specification and an implementation.

The Notepad application was used as a running example to illustrate our approach. It was possible to reduce the states of an initial incomplete model of the Notepad application from 69 to 41 states using the structure of the corresponding HFSM.

To test conformity between the specification and the implementation, intermediate code in C# code is needed to simulate the user actions interacting with the application. Our approach automates this task thanks to a tool, called GUI Mapping Tool, which was developed on purpose.

The Mapping tool reduces the effort to test applications through their GUI based on a formal specification in Spec#. This tool is an extension to Spec Explorer tool that already supports modelling, test case generation, and test case execution.

An overview of the GUI model and test process was provided and the components of Spec Explorer as well as the components of the tool extensions were described.

The GUI Mapping Tool has three components:

- a Spy tool that captures information about the real interactive objects where modelled actions occur;
- a front-end that maps the modelled actions to real objects by dragging and dropping the mouse on the real interactive objects;
- a code generator to construct code simulating the user actions interacting with the GUI AUT;

The tool has some limitations: it requires manual definition of input domains; it only addresses Windows applications; and it does not deal with internationalization, i.e., variable name mappings.

Spec Explorer together with the GUI Mapping Tool can be used to test existing software applications, or it can be used to assist the development of new software applications and to test them through their GUI. In the former case, a reverse engineering process could be useful to construct a model, or part of the model, of an arbitrary application exhibited by its GUI. In the latter case, the specification of the application (or part of the application) is constructed later on to be implemented and tested using automatically generated mapping code.

Chapter V

Case studies

This chapter presents some case studies which illustrate and evaluate the specification-based testing approach proposed in this dissertation.

The specification-based testing approach put forward by this dissertation was validated with the help of two experiments performed on two different kinds of software application: Microsoft's text editor Notepad and a Java software application which manages database files of contacts (Address Book).

Each of these experiments involved the construction of the corresponding software application models, test case generation, and execution.

The Address Book application is based on the Standard Widget Toolkit (SWT). SWT is a set of GUI widgets and related classes which are integrated with the native window system and can be used to build rich client user interfaces in Java. SWT has been developed by the Eclipse Foundation (IBM, Intel, Borland, Computer Associates, etc.) as a part of the open-source Eclipse platform made available in an operating system independent manner.

The Address Book software application was modified with injected errors so as to evaluate how sharp the approach is in fault detecting. The same was not performed on the Notepad application because its source code was not available.

The experiments were performed by a HewlettPackard Pavilion Notebook dv1140EA with the following characteristics:

- CPU: 1.60GHz Intel Pentium M 725 processor;
- RAM: 1.21GB

- Operating System: Microsoft Windows XP.

Whenever possible, quantitative measures concerning these two case studies are presented.

5.1. Notepad application

Notepad (Figure 54) is a basic text editor that ships with the Microsoft Windows operating system. It can be used to edit, view, create and update simple text files:

Edit – The GUI makes it possible to type text; select text; cut, copy, paste, and delete text; and replace, all at once or one by one, the occurrences of one string in a text by another one.

View – The user can open an existing text file in disk, browse through the text, and search for the occurrences of a string in the text (Find) in the following rules:

- case sensitive or case insensitive way;
- backwards or forwards with respect to the current mouse position.

Create or update – the user can create a new text file (save), or update an existing text file (overwrite) in disk.



Figure 54: Notepad main window

5.1.1. Model

The model of the Notepad software application captures the atomic actions available at each time to the user and can be consulted in Appendix A.1. (*Format* and *View* functionalities are not taken into account). Only the *Open* and *Find* functionalities will be used to illustrate the approach. The *Open* dialog is a modal dialog and the *Find* dialog is a modeless dialog. Two models were constructed at different levels of abstraction: taking the focus property of the windows and interactive controls into account, and abstracting from such properties.

The main difference between the two models remains in the fact that to model the focus property of windows and interactive controls inside windows, additional

state variables and methods are needed. The windows focus property is modelled inside the window manager by one state variable that keeps the name of the window/dialog which has the focus at each moment, and methods to manipulate (write and read) that property: `SetFocus`, `GetWindowWithFocus`, and `HasFocus`. Furthermore, there is one state variable inside each window/dialog to indicate the control that has the input focus at each moment and additional methods to switch focus between controls belonging to the same window/dialog. Each method has at least one pre-condition requiring the focus to be set to the interactive control where the action will occur. It is possible to set focus on a window whenever that window is enabled (meaning that it is open and does not have a modal window belonging to the same software application on top of it).

Should the focus property be abstracted away, the state variables `ctrlWithFocus` and the methods `SetControlFocus` and `SwitchToWindow` are not needed. Each method has at least one pre-condition ensuring that the window where the modelled action occurs is enabled, instead of checking if the window is focused.

Abstracting from the focus property will decrease the total number of actions within the model as there will be no actions to switch focus between windows and interactive controls. This has an impact on navigation map views which will be dealt in the sequel.

Notepad specification with the focus property modelled

The state of the Notepad application main window and the actions on the main window are defined inside a namespace called `Notepad` (see display below).

When a new window is created (`AddWindow`), the window manager set the input focus immediately on it.

After launching the application, it is possible to interact with the client area by typing text (`InsText(string txt)`), selecting text (`SelText(int p0, int p1)`) – where `p0` and `p1` are text positions, and with the main menu to open the Open or Replace dialogs (`Open()`, `Replace()`) or close the application (`Close()`). Should the contents of the main window have changed, closing the application or opening another file will be preceded by a message offering the user the opportunity to save changes (`MsgSvBfrClose(string op)` and `MsgSvBfrOpen(string op)`). Actions modelling message boxes will have at least one pre-condition requiring focus set to the message box window.

Module

```
namespace Notepad;
```

Types

```
type dir = string where value in Set{"Up", "Down"};
type windows = string where value in
    {"Notepad", "Find", "Replace"};
```

Variables

```
// editing status
string text      = "", // the text of the main window
    selText      = ""; // text selected
int posCursor   = 0;  // cursor position within the text
bool dirty      = false; // has text been updated?
// file being edited
string fileOpened = "",
    directory = "E:"; // for testing purposes
// find and replace settings
string findWhat  = "", // string to search
    replaceWord = ""; // string to replace for
dir    direction = ""; // "Up" or "Down"
bool matchCase = false, // case sensitive search?
// temporary state of the open feature
bool svBfrOpen = false;
// temporary state of the close feature
bool svBfrClose = false;
```

Controllable actions

```
void LaunchNotepad() // start the Notepad application
void Close() // close the Notepad application
void MsgSvBfrClose(string op) // save changes?
void Open() // open the open dialog
void MsgSvBfrOpen(string op) // save changes?
void Save() // save text in memory to disk
void SaveAs() // open the save dialog
void InsText(string txt) // insert text in the main window
void SelText(int p0,int p1) //select text between p0 and p1
void Find() // open the find dialog
void FindNext // find another occurrence of the "findWhat"
void MsgAckCantFindWord() // can't find the word
void Replace() // open the replace dialog
void SwitchToWindow(Windows win) // switch window focus
```

Observable action

```
string GetText() // observe the text within the main window
```

When the Open dialog is open (Figure 55), it is possible to type a file name (`SetFileName(string fn)`), and press buttons Cancel (`Cancel()`), to close the dialog, or Open (`Open()`), to open an existing text file (the other interactive controls were not modelled). Since Open dialog is modal, it is not possible to interact with the main window of the application until this dialog is closed. When trying to open a nonexistent file, a message box informs the user of that fact (Figure 56).

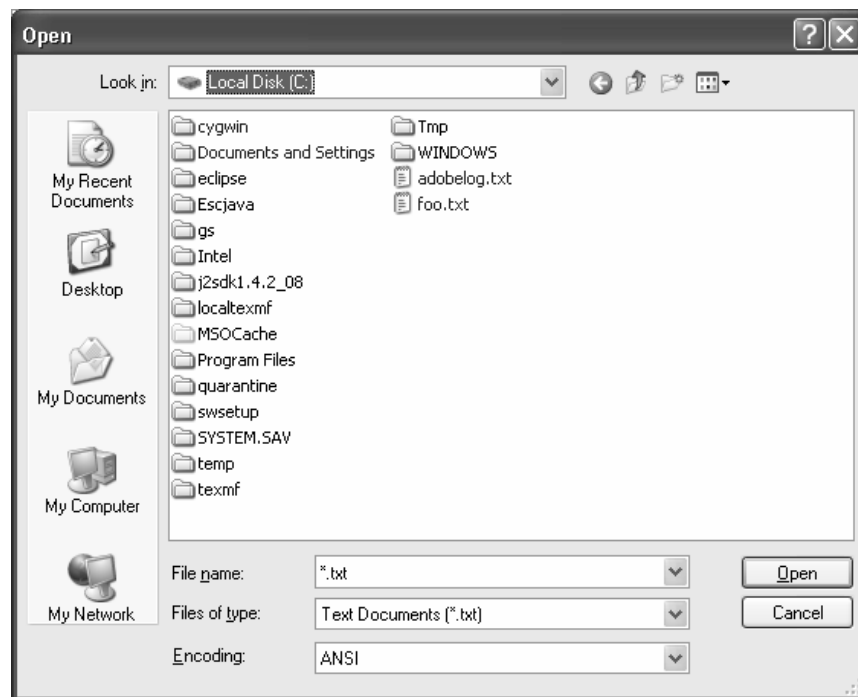


Figure 55: Open dialog

The only action the user can perform on that message box is to acknowledge the message by pressing the "Ok" button (`MsgAckFileNotFound()`).



Figure 56: File not found message box

Module

```
namespace OpenFileDialog;
```

Types

```
type OpenCtrls = string where value in
    Set{"Cancel", "Open", "FileName"}
```

Variables

```
string fileNameO = "*.txt", //name of the file
    dirO = "E:"; // current directory
    // ("E:" for testing purposes)
OpenCtrls openCtrlWthFocus = "FileName"; // control
    // with the input focus
```

Controllable actions

```

void Cancel() // press the cancel button
void Open() // press the open button
void MsgAckFileNotFound() // acknowledge error message
void SetFileName(string fn)//fill in the file name text box
void SetCtrlFocus (OpenCtrls c) // switch control focus

```

The relevant state of the file system is modelled inside a namespace called `FileManager` (Figure 57) by a table/map that associates keys (file names) with values (file contents). Each individual key-value pair (called a maplet) models a file.

This module has methods to create, read, and delete files and also methods to query the state of the file system such as asses if a filename exists (`FileExists`), and asses if a file name is valid (`IsValid`). A file name is valid if it does not have weird characters e.g., `\\, '*', '/', ':', '?', '\", '<', '>', '|'`.

```

namespace FileManager;

Map<string,string> files = Map{};

public void CreateFile(string fileName, string text)
  requires !FileExists(fileName); {
  files = files + Map{fileName :> text};
}
public bool FileExists(string fileName) {
  choose (i in files, i == fileName) return true;
  else return false;
}
public string ReadFile(string fileName)
  requires FileExists(fileName); {
  return files[fileName];
}
public void DeleteFile(string fileName)
  requires FileExists(fileName); {
  files[fileName] = none;
}
bool IsValid(string fileName) {
  if (file == "") return false;
  // IndexOfAny reports the index of the first occurrence
  // in this instance of any character in a specified
  // array of Unicode characters
  if (file.IndexOfAny(new char[8]{
    '\\', '*', '/', ':', '?', '\", '<', '>', '|'}) >= 0)
    return false;
  else return true;
}

```

Figure 57: File manager module

Upon opening the Find dialog (Figure 58), it is possible to fill in the word to search for (`SetFindWhat(string txt)`), to choose the direction to look for

(SetDirection(string d)), to choose if the search is case sensitive or case insensitive (SetMatchCase(bool op)), and also to press the buttons Find Next (FindNext()), and Cancel (Cancel()). The Find Next button is enabled only when the Find What text box is not empty. After the Find Next button is pressed, if the string to look for does not exist, a message box is shown to the user who is expected to acknowledge it by pressing the Ok button (MsgAckCantFindWord()). Since Find is a modeless dialog, it will be also possible to switch to the Notepad main window (Notepad.SwitchToWindow("Notepad")) and the other way around (Notepad.SwitchToWindow("Find")).



Figure 58: Find dialog

Module

```
namespace FindDialog;
```

Types

```
type FindCtrls = string where value in
    Set{"FindWhat", "Direction", "MatchCase",
        "FindNext", "Cancel"}
type dir = string where value in {"Up", "Down"};
```

Variables

```
string    findWhatF    = "";        //word to look for
dir       directionF  = "Down";    //direction to look for
bool      matchCaseF   = false;    //case sensitive search?
FindCtrls findCtrlWthFocus = "FindWhat"; //control with the
                                           //focus
```

Controllable actions

```
void Cancel() // press the cancel button
void SetFindWhat (string fw) // fill "Find what" text box
void SetMatchCase (bool op) // match case option
void SetDirection(dir d) // select direction
void FindNext() // press the "Find Next" button
void SetCtrlFocus() // switch control focus
void MsgAckCantFindWord() // acknowledge user message
```

The complete model of the Notepad application can be found in Appendix A.1.

5.1.2. Scenarios

Models of the GUI under test can be built at different levels of abstraction. The specification of the Notepad application presented above describes the possible atomic actions the user can perform when interacting with the GUI. However, it is possible to describe the main functionalities of the Notepad application at a higher level of abstraction as the main usage scenarios of the GUI under test. High level scenarios capture user visible functions (or high level requirements) to achieve user goals and model typical ways of using the GUI. Scenarios can be described by "scenario actions" inside Spec Explorer. The high level scenarios are constructed on top of atomic user actions that are defined in the complete model of the system (in Appendix A.1.). Scenarios describe possible sequences of atomic user actions. For example, the *FindScenario* presented below describes the sequence of actions a user should perform to search for an occurrence of a string (as indicated by the `word` parameter) in backward or forward direction (as indicated by the `direction` parameter), and in case sensitive or case insensitive way (as indicated by the `matchCase` parameter).

FindScenario: It is possible to search a string within a text:

- In a case sensitive or case insensitive way;
- Look for the string backwards or forwards relative to the mouse position within the text.

```
[Action(Kind=ActionAttributeKind.Scenario)]
void FindScenario(string word, dir direction,
                 bool matchCase)
requires IsEnabled("Notepad") && text != "";
{
    Notepad.Find();
    assert IsEnabled("Find");
    FindDialog.SetFindWhat(word);
    FindDialog.SetDirection(direction);
    FindDialog.SetMatchCase(matchCase);
    FindDialog.FindNext();
    if (IsEnabled("MsgAckCantFindWord"))
        FindDialog.MsgAckCantFindWord();
    FindDialog.Cancel();
}
```

Figure 59: Find scenario within Notepad application

The *assert* clause is used to express a condition that must hold when it is reached. Although it will not be checked by the implementation (only by the model), it was introduced to improve the documentation of the scenario.

ReplaceScenario: It is possible to find a word (indicated by the `word` parameter) in a text file, in a case sensitive or case insensitive way (indicated by the

matchCase parameter), and replace that word by another one (indicated by the replaceWord parameter) (Figure 60).

- A message box will inform the user whenever the word to look for does not exist in the text. In this case, the user should acknowledge the message box by pressing the Ok button (MsgAckCantFindWord());
- It is possible to replace one by one the occurrences of the string in the text or replace all occurrences of the string in one step (as indicated by the repAll parameter).

```
[Action(Kind=ActionAttributeKind.Scenario)]
void ReplaceScenario(string word,
                    string replaceWord,
                    bool matchCase, bool repAll)
requires IsEnabled("Notepad");
{
    Notepad.Replace();
    assert IsEnabled("Replace");
    ReplaceDialog.SetFindWhat();
    ReplaceDialog.SetReplaceWith(replaceWord);
    ReplaceDialog.SetMatchCase(matchCase);
    if (repAll) Replace.ReplaceAll();
    else {
        ReplaceDialog.FindNext();
        ReplaceDialog.Replace();
    }
    if (IsEnabled("MsgAckCantFindWord"))
        ReplaceDialog.MsgAckCantFindWord();
    ReplaceDialog.Cancel();
}
```

Figure 60: Replace scenario within Notepad application

OpenScenario: It is possible to load (open) data from a file in disk (the name of the file to open is indicated by the fileToOpen parameter). If the file name to open does not exist, a message box appears which the user is expected to acknowledge by pressing the Ok button (OpenDialog.MsgAckFileNotFound()). If the text in the main window was updated, a message box will ask the user whether he/she wants to save contents in memory to a text file before opening a new one (as indicated by the saveChanges parameter). If this filename (as indicated by the fileToSave parameter) already exists, a message box appears allowing the user to choose between overwriting and non-overwriting it (as indicated by the overwrite parameter) (Figure 61).

```
[Action(Kind=ActionAttributeKind.Scenario)]
void OpenScenario(string fileToOpen,
                 string saveChanges,
                 string fileToSave,
                 bool overwrite)
```

```

requires IsEnabled("Notepad") &&
        saveChanges in Set{"y","n","c"};
{
    Notepad.Open();
    if (IsEnabled("MsgSaveChanges")) // if dirty
    {
        MsgSvBfrOpen(saveChanges);
        if (saveChanges)
        {
            assert IsEnabled("Save");
            SaveDialog.SetFileName(fileToSave);
            SaveDialog.Save();
            // file exists
            if (IsEnabled("MsgOverwriteFile"))
            {
                SaveDialog.MsgOverwriteFile(overwrite);
                if (!overwrite) {
                    assert IsEnabled("Save");
                    SaveDialog.Cancel(); // close save dialog
                }
            }
        }
    }
    //(saveChanges != c || !dirty)
    if (IsEnabled("Open")) {
        OpenFileDialog.SetFileName(fileToOpen);
        OpenFileDialog.Open();
        if (IsEnabled("MsgAckFileNotFound"))
        {
            OpenFileDialog.MsgAckFileNotFound();
            OpenFileDialog.Cancel(); // end of the scenario
        }
    }
}

```

Figure 61: Open file scenario within the Notepad application

SaveScenario: It is possible to save text (new or updated) to a (new or existing) text file (as indicated by `fileName` parameter). If the file already exists a message box appears allowing the user to choose between overwriting and non-overwriting it (as indicated by the `overwrite` parameter) (Figure 62).

```

[Action(Kind=ActionAttributeKind.Scenario)]
void SaveScenario(string fileName, bool overwrite)
requires IsEnabled("Notepad");
{
    Notepad.SaveAs();
    SaveDialog.SetFileName(fileName);
    SaveDialog.Save();
    if (IsEnabled("MsgOverwriteFile"))
    {
        SaveDialog.MsgOverwriteFile(overwrite);
        if (!overwrite) {
            assert IsEnabled("Save");
            SaveDialog.Cancel();
        }
    }
}

```

Figure 62: Save scenario within Notepad application

5.1.3. Testing goals

It is important to define testing goals as a way to deal with scalability problems and decide when to stop testing.

Testing goals for the Notepad software application were defined based on the following coverage criteria on the generated FSM from which test cases are generated. They aim at defining and checking the set of the FSM testing properties as a way to assess the quality of the FSM from the testing perspective. If the FSM fails such desired properties then the process must go through a new iteration, in which a new FSM is constructed from an exploration of the model after providing new bounds.

The testing goals are:

- Full coverage of the actions in the model – all the modelled actions should be present in the FSM;
- Full coverage of scenarios – all the modelled scenarios should be present in the FSM. The scenarios may be described as model views to check if they are present in the FSM;
- Full coverage of functional dependencies – check if the chosen domains allow showing that all variables affect independently the behaviour of the system (generalization for non-Boolean variables of the MC/DC criterion);
- Full coverage of the test boundary and special conditions – check if the FSM contains the states or sequences of states that describe boundary and special conditions (to be defined ahead);
- Full coverage of the navigation map and dialog views – check if the navigation map and dialog views are fully within the FSM (to be defined ahead).

5.1.4. Choosing domain values for adequate testing

Once the model program of the GUI is written up, Spec Explorer allows us to generate a FSM by bounded exploration. This FSM consists of the states of the model program and method invocations that move from state to state as transitions. In order to explore the model by calling each of the actions available at each state, it is necessary to define the domains of the actions' parameters. Should the set of possible values that a parameter can get be small, the general rule is to define the domain based on that set. Such is the case in the methods which follow:

```

Notepad.SwitchToWindow(window win)
  where window = Set{"Notepad", "Find", "Replace"}
Notepad.MsgSvBfrClose(string op)
Notepad.MsgSvBfrOpen(string op)
  where op in Set{"y", "n", "c"}

OpenDialog.SetCtrlFocus(OpenCtrls c)
  where OpenCtrls = Set{"Cancel", "Open", "FileName"}

```

```

SaveDialog.SetCtrlFocus(SaveCtrls c)
  where SaveCtrls = Set{"FileName", "Save", "Cancel"}
SaveDialog.MsgOverwriteFile(string op)
  where op in Set{"y", "n"}

FindDialog.SetCtrlFocus(FindCtrls c)
  where FindCtrls = Set{"FindWhat", "Direction",
                       "MatchCase", "FindNext",
                       "Cancel"}
FindDialog.SetMatchCase(bool op)
  where op in Set{true, false}
FindDialog.SetDirection(dir d)
  where dir = Set{"Up", "Down"}

ReplaceDialog.SetCtrlFocus(ReplaceCtrls c)
  where ReplaceCtrls = Set{"Cancel", "Replace",
                           "ReplaceWith", "FindWhat",
                           "ReplaceAll",
                           "MatchCase", "FindNext"}
ReplaceDialog.SetMatchCase(bool op)
  where op on Set{true, false}

```

For the other cases, a reduction of the number of possible values is on demand. This domain reduction is applied according to the testing goals defined for the current GUI under test. The domains chosen (Table 2) must allow for full coverage of the functional dependencies and full coverage of test boundary and special conditions.

Actions with parameters	Test conditions	Domains
namespace Notepad: InsText(char txt)	Upper and lower case to test the "match case" option inside the find dialog	{'a', 'A'}
SelText(int p0, int p1)	All pairs of integers that satisfy the pre-condition	a)
namespace OpenFileDialog: SetFileName(string fn)	Test for existing and non-existing files	{"foo.txt", "foo.htm"}
namespace SaveDialog: SetFileName(string fn)	Test for existing and non-existing files	{"foo.txt", "foo.html"}
namespace FindDialog: SetFindWhat(string str)	Test for existing and non-existing words	{"A", "aA"}
namespace ReplaceDialog: SetFindWhat(string str)	Test for existing and non-existing words	{"A", "aA"}
SetReplaceWith(string str)	A char possibly different from the ones within the text	{"a"}

Table 2: Domains for actions' parameters

- a) This is a dynamic set of values because it depends on text contents at each state. In Spec Explorer, dynamic domains can be defined by properties.

SelectText property defined below reads the text values at each state and calculates the set of values valid for the SelText parameters.

```
Set<<int,int>> SelectText { get {
  if (text.Length>0)
    return Set{p0 in Set{0..text.Length-1},
              p1 in Set{p0+1..text.Length};<p0,p1>};
  else return Set{<0,0>};
}}
```

Domain definition is an iterative process involving the need to verify full coverage of functional dependencies. This is checked by Table 3 for the example under test.

Inputs						Find effects		
text	selText	posCursor	findWhat	direction	matchCase	Change posCursor?	Change selText?	Appears Message?
aaA	""	0	A	Down ↑	F	T ↑	T ↑	F ↑
aaA	""	0	A	Up ↓	F	F ↓	F ↓	T ↓
aaa	""	1	aA	Down	T ↑	F ↑	F ↑	T ↑
aaa	""	1	aA	Down	F ↓	T ↓	T ↓	F ↓
aAa	Aa	3	aA	Up	T	T ↑	T ↑	F ↑
aAa	aAa	3	aA	Up	T	F ↓	F ↓	T ↓
aaA	""	1	aA	Down	T	T ↓	T ↓	F ↓
aaA	""	3	A	Down	F	F ↓	F ↓	T ↓
Aa	""	1	aA	Down	F	F ↓	F ↓	T ↓
Aa	""	1	A	Down	F	T ↓	T ↓	F ↓

Table 3: Test data and coverage analysis for the Find effect

By analysing Table 3 one concludes that the domains defined above allow for testing the find effect according to the full coverage of functional dependencies criterion. Column "Message" refers to the effect of a message box showing up to inform the user that the word to look for could not be found in the text.

It should be stressed that this kind of analysis can be automated. In such case, the manual task remaining would be to provide additional domain values when the test goals are not met yet.

Besides the test conditions identified in Table 3, it may be interesting to identify additional boundary test conditions and other special conditions. Boundary test conditions correspond to situations located near limits of valid ranges where errors

are most likely to occur. Examples of boundary test conditions for the find effect are:

- The word to look for is at the beginning of the text.

```
text.IndexOf(findWhat) == 0
```

- The word to look for is at the end of the text.

```
text.LastIndexOf(findWhat) == text.Length -  
findWhat.Length
```

- The word to look for is equal to the text content.

```
text == findWhat
```

- The cursor's position is in middle of the word to look for.

```
Exists{ i in Set{0..text.Length};  
posCursor>i && posCursor<i+findWhat.Length &&  
i==text.IndexOf(findWhat)}
```

- The word occurs several times within the text and the different occurrences overlap each other

```
Example: text      = "aAaAa";  
findWhat  = "aAa";  
matchCase = false;
```

This can be written in Spec# as

```
if ((Exists{i in Set{1..findWhat.Length-1};  
findWhat.Substring(0,i)==  
findWhat.Substring(findWhat.Length-i,i)  
&&  
text.IndexOf(findWhat+  
findWhat.Substring(i,text.Length))>=0}  
||  
(Exists{i in Set{1..findWhat.Length-1};  
findWhat.Substring(0,i).ToLower() ==  
findWhat.Substring(findWhat.Length-i,  
i).ToLower()  
&&  
text.ToLower().IndexOf(findWhat.ToLower()+  
findWhat.Substring(i,text.Length).ToLower())  
>=0}  
&& !matchCase))
```

- The word occurs several times within the text and the different occurrences are side by side

```
Example: Text="aAaAa"; word="Aa"; MatchCase=true
```

This can be written in Spec# as

```
text!=" " && findWhat!=" " &&  
(text.IndexOf(findWhat+findWhat)>=0
```

```

||
text.ToLower().IndexOf(findWhat.ToLower()+
    findWhat.ToLower())>=0 && !matchCase)

```

By analysing Table 4 one conclude that the domains defined allow for testing of the replace effect meeting the full coverage of functional dependencies criterion.

Inputs					Effect
Text	selText	findWhat	matchCase	replaceWith	Replace
aaA	A	A	F	a	T
aaA	A	A	T	a	F
aaA	A	aA	F	a	F
aaA	aA	aA	F	a	T

Table 4: Test conditions for the Replace effect

Table 5 checks the full coverage of functional dependencies criterion for the Open and Save effects inside the Open scenario.

Inputs					Effect	
dirty	Exists(fileToOpen)	saveChanges	Exists(fileToSave)	overwrite	Saved?	Opened?
T	T	Y	-	T	T	T
T	T	Y	F	-	T	T
T	T	Y	T	F	F	T
T	T	N	-	-	F	T
T	-	C	-	-	F	F
F	F	-	-	-	F	F
T	F	N	-	-	F	F
T	F	Y	-	T	T	F
T	F	Y	F	-	T	F
F	T	-	-	-	F	T

Table 5: Conditions to test the save and open effects inside the Open scenario

5.1.5. State filtering

Once domains are defined and checked for achieving full coverage of functional dependencies criterion, additional techniques can be used to prune the exploration

process in order to generate a FSM with manageable size. For this purpose, one may define state filters excluding from the exploration process all states where the specified state condition does not hold.

An additional state filter was added to the Notepad software application limiting the size of the text variable that models the text inside the Notepad main window.

```
text.Length <= 3
```

The size of the text should be chosen in a way so as not to forbid achieving states where boundary and special conditions hold. For example, a text size limited to 2 (instead of 3) would not allow states where a word occurs several times within the text and the different occurrences overlap each other.

5.1.6. FSM generation and reduction

The generation of the full FSM for the domains and state filtering defined previously in a single step was not practical so FSMs for subsets of the model were generated. One of those subsets containing the behaviour of the dialog Find is reported in section 5.1.10. Although the complete FSM was never generated, a FSM with enough size, i.e., covering all the testing goals defined, was used to illustrate the process of FSM validation in the next section.

5.1.7. FSM validation

Once parameter domains and state filters are settled, the Spec Explorer tool generates automatically a FSM by exploring the model program within defined bounds. By default, all states of the model that are reachable within such bounds will be explored and represented in the FSM.

For visualization purposes, Spec Explorer allows us to provide criteria to group together in the same vertex states sharing a common characteristic (two states are grouped together if an expression provided by the user evaluates identically). These expressions can be used to construct views (or projections) at different levels of abstraction that may be used for FSM validation. This can be done by generating different views for different coverage goals (dialog views, scenarios, functional dependencies, and special cases) and visually inspecting those views to check for testing goals coverage.

Projections obtained from the Notepad model with focus property modelled

In our example, the variable `hasFocus` inside the window manager refers to the window or dialog with the input focus at each moment. Variables `ctrlWthFocus` defined inside each dialog are used to point out the interaction object that has the input focus in the dialog. By querying such variables it is

possible to obtain two different views of the model: navigation map view and dialog view.

```
string NavigationMap { get {
    if (GetWindowWithFocus() == "")
        return "NotOpen";
    else return GetWindowWithFocus();
}}
```

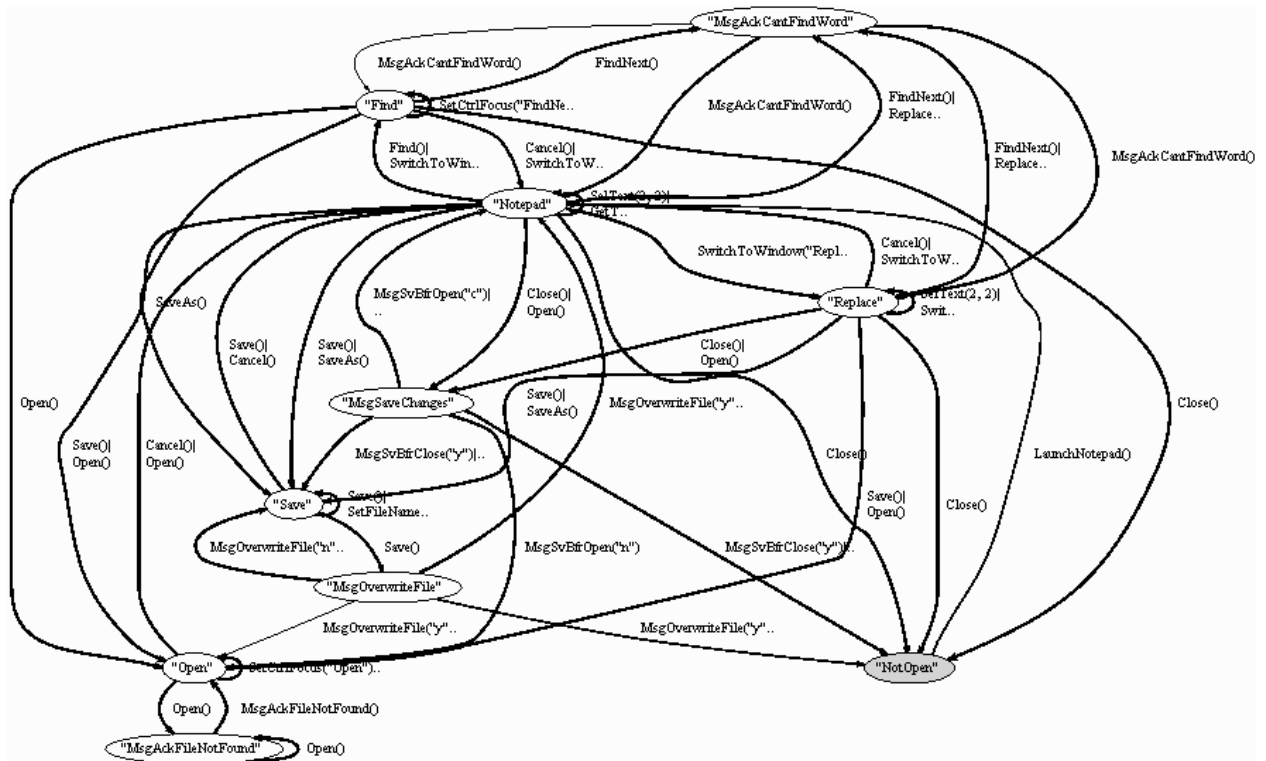


Figure 63: Navigation map obtained from focus property of the windows

The navigation map view is obtained from the model by projecting the state onto the name of the window with the input focus (Figure 63).

Within the navigation map view each vertex corresponds to a group of states where a specific window has the input focus. In this view, it is possible to see that the user can interact with the main window of the Notepad application by interacting with the menu to open one of the dialogs, open (e.g., `Open()`) and find (e.g., `Find()`) or by interacting with the client area selecting text (`SelectText(...)`). It is also possible to switch focus between Find and Replace dialogs (`SwitchToWindow("Find")`, `SwitchToWindow("Replace")`) whenever one of them is opened. The interaction inside such dialogs is detailed at the lower level of abstraction.

The dialog views are obtained by projecting the states where the dialog has the focus onto the `ctrlWthFocus` variable. This can be obtained by

```
string OpenDialogGroup { get {
    if (!IsOpen("Notepad")) return "NotOpen";
```

```

else if (IsOpen("Open")) return openCtrlWthFocus;
else return "OpenDlgClosed"; }}

```

and is illustrated in Figure 64.

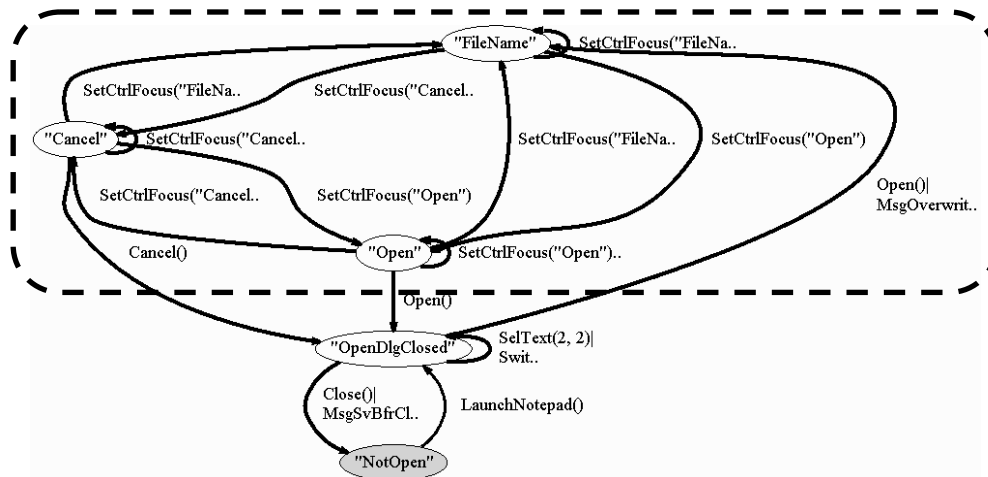


Figure 64: Open dialog view

Inside the open dialog, the user can interact with the file name textbox (`SetFileName(...)`), open a file (`Open()`), and close the dialog (`Cancel()`). By default, when the Open dialog is opened (`Open()` transition), the interaction object with the focus is the `FileName` textbox.

The find dialog view can be obtained by

```

string FindDialogGroup { get {
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (HasFocus("Find")) return findCtrlWthFocus;
    else if (IsOpen("Find")) return "FindDlgNotActive";
    else return "FindDlgClosed";
}}

```

and is illustrated by Figure 65.

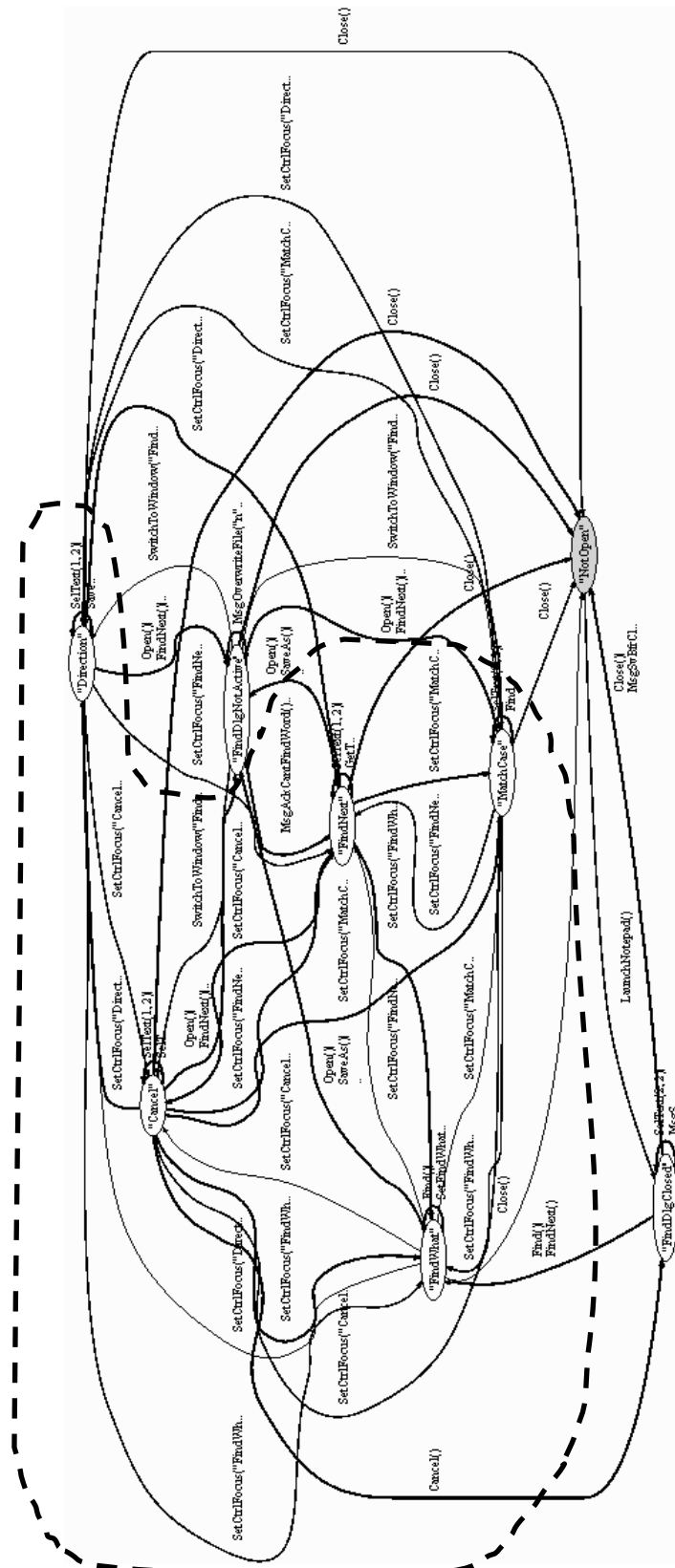


Figure 65: Find dialog view

The Find dialog can have the focus (i.e., be active), in which case there is an interactive control with focus, or may be opened without focus, in which case it is NotActive. When the Find dialog is active, the user can fill in the "find what" textbox (FindWhat(...)), choose the search direction (SetDirection()), choose if the search is case sensitive or not (SetMatchCase()), and press the buttons find next (FindNext()), and cancel (Cancel()).

Projections obtained from the Notepad model abstracting from the focus property

Modelling the focus property requires too much additional effort that is not rewarding if the test goal does not include checking which interactive object has the input focus at each moment. Although the navigation map view and dialog views can be easily obtained from models where the focus property is modelled explicitly, it is also possible to obtain other views from models where the focus property is abstracted. The navigation map view is obtained by querying which dialogs are enabled instead of querying which dialog has the focus at each moment.

```
Set<string> NavigationMap { get {
    return GetEnabledWindows();
}}
```

The diagram obtained from the expression above is illustrated by Figure 66. In this view it is possible to see the set of enabled windows at each moment and the actions available in each of those sets. There are groups of states where two different modeless windows are enabled, e.g., the Notepad main windows and the Find dialog, or the Notepad main window and the Replace dialog.

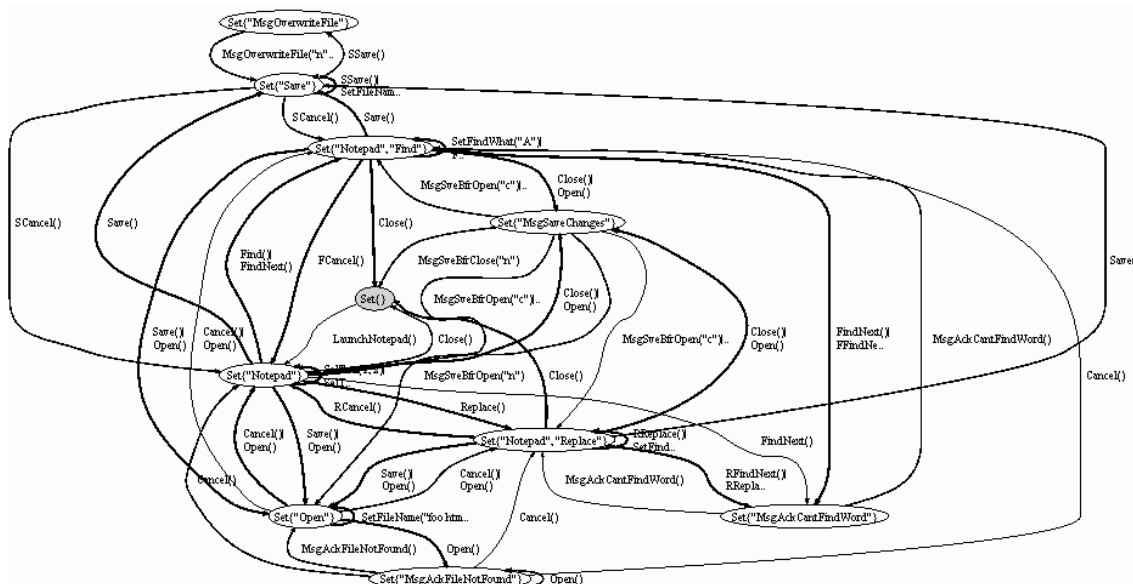


Figure 66: Navigation map obtained from the enabled windows' property

The dialog views are obtained by projecting the state onto the variables manipulated by each dialog.

```

<string,string> OpenFileDialog { get {
  if (IsOpen("Open"))
    return <"fileNameO="+fileNameO, "dirO="+dirO>;
  else return <"NotOpen", "NotOpen">;
}}

```

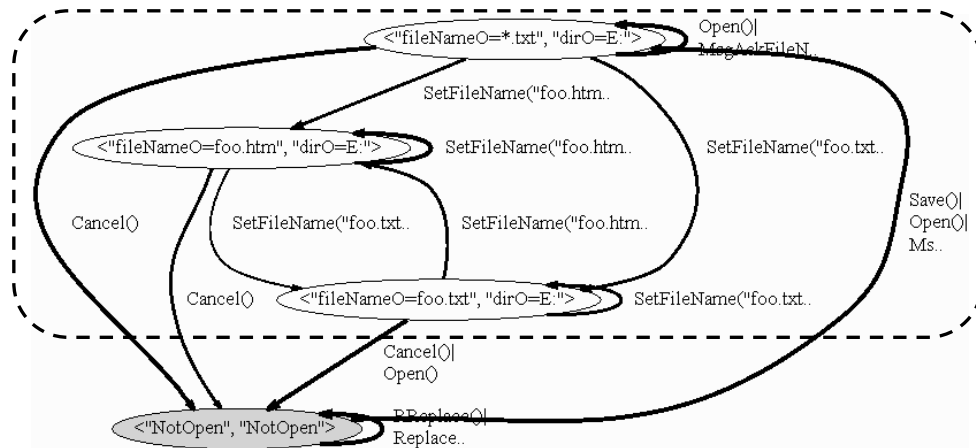


Figure 67: Open dialog view obtained from the projection onto the manipulated variables

In this view (Figure 67) it is possible to see the set of states of the Open dialog that correspond to the different possible combinations of the manipulated variables of the dialog.

Scenarios

In order to check if the identified scenarios are covered by the generated FSM, one should construct views that can be inspected visually to infer if there is full branch coverage of the scenarios.

Open scenario: The view corresponding to the open scenario is illustrated in Figure 45.

Save scenario: The view corresponding to the save scenario can be defined by the following state group:

```

// save scenario
string SaveScenario { get {
  if (!IsOpen("Notepad")) return "NotOpen";
  else if (!IsOpen("Save")) return "SaveDlgClosed";
  else if (IsEnabled("Save")) return "Save";
  else return "MsgOverwriteFile";
}}

```

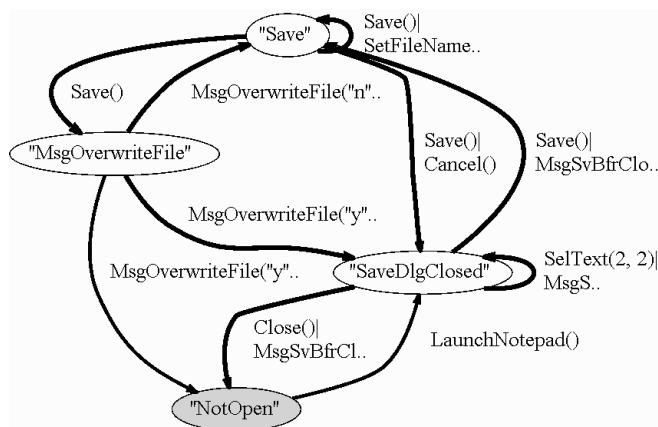


Figure 68: Save scenario view

Find Scenario: The view corresponding to the find scenario described by Figure 59 can be defined by the following state group:

```

string FindScenario { get {
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (!IsOpen("Find")) return "FindDlgClosed";
    else if (HasFocus("Find")) return "Find";
    else if (IsOpen("MsgAckCantFindWord"))
        return "MsgAckCantFindWord";
    else "FindDlgNotActive";
}}
  
```

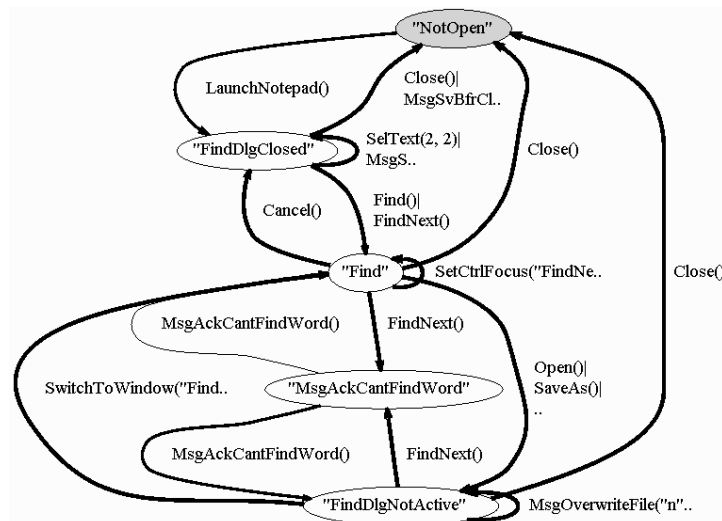


Figure 69: Find scenario view

Checking if the boundary (and special) conditions are covered by the generated FSM can be done by visual inspection of the views generated from each of the formal expressions that describe them. In case some of the test conditions are not covered, it is still possible to construct scenarios to drive the system into the desired states or to redefine the domains and generate the FSM again.

Functional dependencies

Table 3 (on page 147), Table 4 (in page 149), and Table 5 (on page 149) show that it is possible to define states with the identified domains that guaranties full coverage of functional dependencies. Even so, it is possible to check if all the states identified in the tables are present in the generated FSM by writing a state expression for each pair of lines in the table that show result dependency on one of the input parameters. For instance, to check if the direction parameter affects the result independently, it is possible to construct a view based on the first and second lines of Table 3 as:

```
string FindDirectionDependency { get {
    if (text=="aaA" && selText==" " && posCursor==0
        && findWhat == "A" && !matchCase)
        if (direction == "Down")
            return "first line";
        else return "second line";
    else return "any other state";
}}
```

and inspected visually in Figure 70.

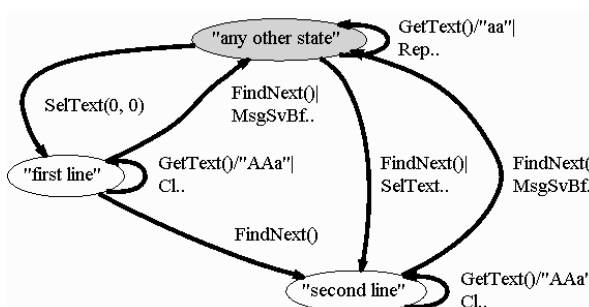


Figure 70: Coverage analysis of a functional dependency

Special cases

The same process can be used to check if boundary and special conditions are within the generated FSM. Formal expressions for this purpose are given in section 5.1.4, whereby it is possible to construct views and inspect them visually for coverage analysis. Figure 71 shows a view intended to analyse the coverage of the special situation where a word occurs several times within the text and those occurrences overlap with each other. This can be expressed in Spec# by writing

```
string OverlapGroup { get {
    if ((Exists{i in Set{1..findWhat.Length-1};
        findWhat.Substring(0,i)==
        findWhat.Substring(findWhat.Length-i,i) &&
        text.IndexOf(findWhat+
            findWhat.Substring(i,text.Length))>=0))
```

```

||
(Exists{i in Set{1..findWhat.Length-1};
 findWhat.Substring(0,i).ToLower() ==
 findWhat.Substring(findWhat.Length-i,i).ToLower() &&
 text.ToLower().IndexOf(findWhat.ToLower()+
 findWhat.Substring(i,text.Length).ToLower())>=0}
 && !matchCase))
return "Overlap";
else return "NotOverlap";

```

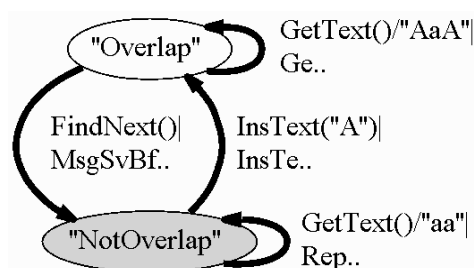


Figure 71: Coverage analysis of a special case situation "several occurrences overlapping each other"

5.1.8. Test case generation and execution

Upon defining domains for the methods' parameters and generating and validating the FSM thus assessing its quality based on scenarios and boundary test conditions, it is possible to generate test cases from the FSM thus obtained. However, executing all possible test cases may be not realistic due to the huge size of the FSM generated and consequently the huge number of test cases.

A new algorithm is presented in section 4.3.4 to reduce the FSM while guaranteeing coverage of the intermediate level of abstraction defined by the navigation map and dialog views. After applying this pruning technique to the initial FSM, the size of the FSM is reduced and test cases may be generated from it based on full transition coverage criterion, later to be executed.

5.1.9. Test results

In order to test the Notepad application without resorting to its source code (that is, running its executable binary file), some intermediate code, in C#, must be written to execute and interact with the application simulating the user (this will trigger events like mouse clicks or keyboard keys). Every method at the specification level will have a corresponding method at the intermediate code that will simulate the user actions. Maps between functions at specification and implementation levels are established so that the tool can run related methods at both levels and compare the results obtained.

The intermediate code needed to simulate the user actions and the maps between methods of the specification and implementation levels is built automatically with the support of the tool described in section 4.4.

Test execution is performed by Spec Explorer tool. Every time there is an inconsistency (i.e., the outcome of an observable action at the specification level is different from the outcome of an related method at the implementation level) it is reported.

Observable actions whose pre-conditions hold are executed after each controllable action. In the case of the Notepad model, there is just one observable action that sees the content of the main window whenever possible (when the main window is enabled).

During the testing of Notepad, we found two sequences of actions which lead to an inconsistency between our intuitive model and the actual Notepad application:

- After executing the next sequence of actions the Notepad will search upwards instead of downwards as expected:
 1. Type text.
 2. Search for text using the find dialog (Ctrl-F) in the downward direction. Close the dialog (press Cancel button).
 3. Open the replace dialog (Ctrl-H). Close the dialog (press Cancel button).
 4. Press the F3 key (shortcut for "Find Next").

- After executing the next sequence of actions, the Notepad will search downward instead of upward as expected:
 1. Type text, for instance, "aaa".
 2. Search for text (e.g., "a") using find dialog (Ctrl-F) in upward direction. Close the dialog (press Cancel button).
 3. Open the find dialog (Ctrl-F) and close it immediately (press Cancel button).
 4. Press the F3 (shortcut for "Find Next").

These are sequences of events that manual tests would probably miss since they are not common scenarios.

Finding only two errors is after all not surprising since the Notepad application has been in use and tested for years already.

5.1.10. Metrics

Several test experiments were performed in order to test Notepad software application and as a way to evaluate the testing approach proposed in this dissertation.

The Notepad model was constructed in a week. It consists of 35 actions and 38 helper methods. The window manager consists of 10 methods. The file manager consists of 5 methods.

For each experiment several metrics were gathered: FSM generation time; size of the original generated FSM; size of the FSM after reduction; time taken to validate the FSM according to coverage criteria defined; test suite length; and errors found. In addition, the configuration (set of actions to be considered for FSM generation and domains for the action parameters) used by each experiment is annotated.

Although several experiments were performed, just one of them is reported here for illustration. The goal of this experiment is to test the find word functionality of Notepad. The subset of actions (and parameter values) of the Notepad model used in this experiment is listed in Table 6.

Actions	Parameter domains
Notepad.LaunchNotepad()	
Notepad.Close()	
Notepad.GetText()	
Notepad.InsText(string txt)	{"a", "A"}
Notepad.SelText(int x, int y)	if text.Length>0 {p0 in Set{0..text.Length-1}, p1 in Set{p0+1,text.Length}; <p0,p1>} else {<0,0>}
Notepad.Find()	
Notepad.FindNext()	
Notepad.Replace()	
Notepad.MsgAckCantFindWord()	
Notepad.MsgSvBfrClose(string op)	{"n"}
FindDialog.FindScn (string fw, string dir, string mc)	{"A","Up",false}, {"A","Down",false} {"aA","Up",true}, {"aA","Down",true} {"aA","Down",false} /* These values were taken from Table 3 */
FindDialog.FindNext()	
FindDialogMsgAckCantFindWord()	
FindDialog.Cancel()	
ReplaceDialog.Cancel()	

Table 6: Actions and parameter domains used in the first test experiment

The time needed to generate the FSM is 1 day, 7 hours and 47 minutes. The FSM has 65701 states, 158571 transitions, and 30 invocations (actions with parameters).

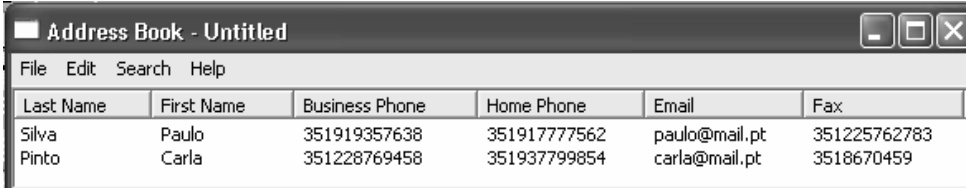
The quality of the FSM was accessed according to coverage criteria defined in section 5.1.7 for the find word functionality. It took half an hour to conclude that the FSM had the desired quality properties. Test suite generated from this FSM has 1 segment with the total length of 257615 steps.

The pruning technique described in section 4.3.4 was applied to the original FSM. After reduction, the FSM has 2478 states, 7573 transitions and 30 invocations. The number of transitions is reduced in 94.6% while the number of states is reduced in 96.2%. The time needed to reduce the FSM is 16 hours. The reduced FSM preserved the desired testing properties. Test suite generated from the reduced FSM has 466 segments with the total length of 15566 steps.

With this experiment it was possible to find the two bugs (reported in section 5.1.9).

5.2. Address book application

The address book application (Figure 72) allows for managing (creating, updating, deleting, and querying) a database file of contacts. The address book data file keeps personal information, like last name, first name, business phone, home phone, email, and fax number for each contact.



Last Name	First Name	Business Phone	Home Phone	Email	Fax
Silva	Paulo	351919357638	35191777562	paulo@mail.pt	351225762783
Pinto	Carla	351228769458	351937799854	carla@mail.pt	3518670459

Figure 72: Address book main window

5.2.1. Model

Modelling the Address Book software application while capturing atomic user actions requires five namespaces that correspond to the different windows/dialogs of the software application: AddressBook (for the main window); OpenFileDialog (to open an existing database file of contacts); SaveDialog (to create a new database file of contacts or update an existing one); ContactDialog (to add a new contact or update an existing one); FindDialog (to query the database).

The AddressBook namespace models the main window of the software application.

Module

```
namespace AddressBook;
```

Types

```
Fields = string where value in Set{"Last Name",
    "First Name", "Business Phone",
    "Home Phone", "Email", "Fax"};
Dir     = string where value in Set{"Up", "Down"};
SortDir = string where value in Set{"Asc", "Desc"};
Contact = <string, string, string, string, string, string>;
```

Variables

```
Contact      contactInMem    = <"", "", "", "", "", "">;
Seq<Contact> dbContacts     = Seq{};
SortDir      sort           = "Asc";
Fields       orderedBy      = "Last Name";
string       fileOpened     = "",
                directory    = "E:", //for test purposes
                nextAction   = "";

int lineSelected = -1;
bool addNew      = true,
    dirty        = false;
bool returnToOpenDlg = false,
    returnToAddressBook = false;
```

Controllable actions

```
void LaunchAddressBook() // start the sw application
void Close() // close the sw application
void MsgSvBfrClose(string op) // save changes?
void NewContact() // open Contact dlg to add a new contact
void SelContact(int line) // select one of the contacts
void EditContact() // edit selected contact
void Copy() // copy selected contact
void Paste() // paste the contact in memory
void Delete() // delete selected contact
void Sort(Fields field) // sort contact by field
void MsgSvBfrNew(string op) // save changes?
void NewAddressBook() // start a new address book
void MsgSvBfrOpen(string op) // save changes?
void OpenAddressBook() // open an existing file of contacts
void SaveAddressBookAs() // save the address book
void SaveAddressBook() // save address book
void Find() // open find dialog
void FindNext() // look for a word
```

Observable actions

```
Contact GetContacts() // observe the contacts shown
                // in the main window
```

The model of the Address Book software application is similar to the Notepad application. The main differences can be found in the edit (Contact dialog) and view (Find dialog) functionalities. The contact dialog allows for adding, one by one, new contacts to a database file, and for updating existing contacts. It is also possible to copy-paste and delete, one by one, existing contacts. Contacts may be sorted by a specific field in ascending or descending order. It is also possible to browse through all contacts in a sequential way using the arrow keys.

The Address Book software application has dialogs to open and save address book files from/to disk that are similar to the ones used by the Notepad application, so the modules of both dialogs are reused by the address book application without any changes. Two different modules were developed to model the Contact (Figure 73) and Find (Figure 74) dialogs.

Figure 73: Contact dialog of the Address Book

Module

```
namespace ContactDialog;
```

Variables

```
Contact contact = <"", "", "", "", "", "">
```

Actions

```
void Cancel() // close the contact dialog
void Ok() // press Ok button
void SetLastName (string ln) // fill the last name
void SetFirstName (string fn) // fill first name
void SetBusinessPhone(string bph) // fill business phone
void SetHomePhone(string hph) // fill home phone
void SetEmail(string email) // fill email
void SetFax(string fax) // fill fax
```

The Find dialog has additional particularities when compared to the corresponding module in the Notepad application: the user can select the field where the word will be searched and there is an additional option: "Match whole word". When "Match whole word" is selected, the search is set for a word in the database field (column) selected which is an exact match of the word in the "Find What" text

box. When this option is not selected, the search may return a word (field value) that contains the word to look for as substring.

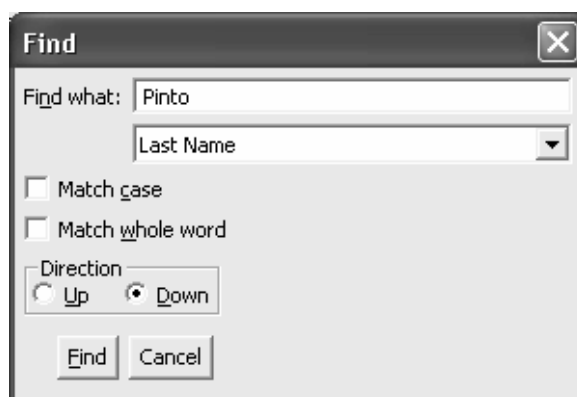


Figure 74: Find dialog of the Address Book

Module

```
namespace FindDialog;
```

Variables

```
String findWhat = "",
      field      = "",
      direction  = "Down";
bool   matchCase      = false,
      matchWholeWord = false;
```

Actions

```
void SetFindWhat(string str) // fill find what
void SetField(string str) // select field
void SetMatchCase(bool op) // choose match case option
void SetMatchWholeWord(bool op) // choose match whole word
void SetDirection(string d) // choose direction
void Find() // press find button
void Cancel() // press cancel button
void MsgAckCannotFindWord() // acknowledge message
```

5.2.2. Scenarios

The main functionalities of the Address Book application may be described by the following high level scenarios: find, open, save, edit, and view.

Find Scenario: It is possible to search contacts that match a search string within one of the contacts' fields:

- in a case sensitive or case insensitive way;
- by looking for a string that is an exact match with the field or that is a substring of the field content;

- by searching backwards or forwards, with respect to the record (Contact) currently selected;
- by issuing a message box informing the user every time the operation tries to find a word that does not exist in the given database field.

```

void FindScenario(string fw, Fields field, bool mc,
                 bool mww, Dir dir)
requires IsEnabled("AddressBook");
{
    AddressBook.Find(); // Opens the Find dialog
    assert IsEnabled("Find");
    FindDialog.SetFindWhat(fw);
    FindDialog.SetField(field);
    FindDialog.SetMatchCase(mc);
    FindDialog.SetMatchWholeWord(mww);
    FindDialog.SetDirection(dir);
    FindDialog.Find();
    if (IsEnabled("MsgAckCantFindWord"))
        FindDialog.MsgAckCantFindWord();
    FindDialog.Cancel();
}

```

Open Scenario: It is possible to load (open) an address book from a file in disk (indicated by the `fileToOpen` parameter). When the file to open does not exist, a message box will appear providing such information to the user, which the user should acknowledge by pressing its Ok button (`OpenDialog.MsgAckFileNotFound()`). Should the address book in the main window be updated, a message box will appear allowing the user to choose between saving and not saving (as indicated by the `svChanges` parameter) the updates to a data file (indicated by the `fileToSave` parameter) before opening the new database. If the file name (indicated by `fileToSave`) already exists, a message box will appear allowing the user to choose between overwriting and cancelling the operation (as indicated by the `overwrite` parameter).

```

void OpenScenario(string fileToOpen, string svChanges,
                 string fileToSave, string overwrite)
requires IsEnabled("AddressBook");
{
    AddressBook.OpenAddressBook();
    if (IsEnabled("MsgSvBfrOpen")) // if dirty
    {
        MsgSvBfrOpen(svChanges);
        if (svChanges)
        {
            assert IsEnabled("Save");
            SaveDialog.SetFileName(fileToSave);
            SaveDialog.Save();
            if (IsEnabled("MsgOverwriteFile")) { // file exists
                SaveDialog.MsgOverwriteFile(overwrite); // yes/no
                if (IsEnabled("Save")) // don't want to overwrite
                    SaveDialog.Cancel(); // so end of the scenario
            }
        }
    }
    //(saveChanges != c || not dirty)
    if (IsEnabled("Open")) {
        OpenDialog.SetFileName(fileToOpen);
        OpenDialog.Open();
    }
}

```

```
        if (IsEnabled("MsgAckFileNotFound"))
        {
            OpenFileDialog.MsgAckFileNotFound();
            OpenFileDialog.Cancel(); // end of the scenario
        }
    }
}
```

Save Scenario: This makes it possible to save an address book (new or updated) to a file. If the file name already exists, a message box appears asking the user for permission to replace/overwrite it or to cancel the operation.

```
void SaveScenario(string fileName, string overwrite)
requires IsEnabled("AddressBook");
{
    AddressBook.SaveAddressBook();
    if (IsEnabled("Save")) //no file currently opened
    {
        SaveDialog.SetFileName(fileName);
        SaveDialog.Save();
        if (IsEnabled("MsgOverwriteFile"))
        {
            SaveDialog.MsgOverwriteFile(overwrite);
            if (IsEnabled("Save"))
                SaveDialog.Cancel();
        }
    }
}
```

Close Scenario: Whenever trying to close the Address Book software application in a state where its content is updated, a message will allow the user to choose among saving the content to a data file (thus preventing potential loss of important information), not to save the content to a data file, and to cancel the operation.

```
void CloseScenario(string svChanges, string fn,
                  string overwrite)
requires IsEnabled("AddressBook");
{
    AddressBook.Close();
    if (IsEnabled("MsgSaveChanges")) {
        AddressBook.MsgSvBfrClose(svChanges);
        if (svChanges == "y")
            if (IsEnabled("Save")) {
                SaveDialog.SetFileName(fn);
                if (IsEnabled("MsgOverwriteFile")) {
                    SaveDialog.MsgOverwriteFile(overwrite);
                    if (overwrite == "c") {
                        AddressBook.Close();
                        if (IsEnabled("MsgSaveChanges"))
                            AddressBook.MsgSvBfrClose("n");
                    }
                }
            }
    }
}
```

5.2.3. Testing goals

As already mentioned, it is important to define test goals as a way to deal with scalability and evaluate when to stop testing.

The testing goals defined for testing the Address Book application are similar to the ones defined for the Notepad application:

- Full coverage of the actions in the model;
- Full coverage of scenarios;
- Full coverage of functional dependencies (a generalization for non-Boolean variables of the MC/DC coverage criterion);
- Full coverage of the test boundary and special conditions;
- Full coverage of the navigation map and dialog views.

5.2.4. Choosing domain values for adequate testing

As already mentioned, domains values must be defined in order to generate a FSM by exploration of the model, the goal being to find domains that allow achieving the testing goals listed in the previous section. As earlier on, whenever the defined domains are not sufficient to achieve the testing goals, they must be redefined.

When the set of possible values that a parameter can get is finite and small, the general rule is to define the domain based on such a set. This is the case of the following methods.

```

AddressBook.MsgSvBfrClose(string op)
AddressBook.MsgSvBfrNew(string op)
AddressBook.MsgSvBfrOpen(string op)
  where op in Set{"y","n","c"}

AddressBook.SelContact(int line)
  where line in Set{0..dbContacts.Size-1}

AddressBook.Sort(Field f)
  where f in Set{"Last Name", "First Name",
                "Business Phone", "Home Phone",
                "Email", "Fax"}

FindDialog.SetDirection(Dir d)
  where d in Set{"Up","Down"}
FindDialog.SetField(Field f)
  where f in Set{"Last Name", "First Name",
                "Business Phone", "Home Phone",
                "Email", "Fax"}
FindDialog.SetMatchCase(bool op)
FindDialog.SetMatchWholeWord(bool op)
  where op in Set{true, false}

SaveDialog.MsgOverwriteFile(string op)
  where op in Set{"y","n"}

```


Inputs							Find effect	
Contacts	Line Selected	find What	direction	match Case	match Whole Word	field	Changed lineSelected ?	Appears message?
<"Pinto", "", "3", "", "", ""> ↑	-1 ↑	"pin" ↑	Down ↑	F ↑	F ↑	Last Name ↑	T	F
<"Pinto", "", "3", "", "", "">	-1	"pin"	Down	F	F	Business Phone ↓	F	T
<"Pinto", "", "3", "", "", "">	-1	"pin"	Down	F ↓	T ↓	Last Name	F	T
<"Pinto", "", "3", "", "", "">	-1	"pin"	Down	T ↓	F	Last Name	F	T
<"Pinto", "", "3", "", "", "">	-1	"pin"	Up ↓	F	F	Last Name	F	T
<"Pinto", "", "3", "", "", "">	-1	"nund" ↓	Down	F	F	Last Name	F	T
<"Pinto", "", "3", "", "", "">	0 ↓	"pin"	Down	F	F	Last Name	F	T
<"Silva", "", "1", "", "", ""> ↓	-1	"pin"	Down	F	F	Last Name	F	T

Table 8: Test data for the Find effect

Table 9 checks for full coverage functional dependencies criterion for the Sort effect. In order to avoid having mutually dependency among input variables the set of contacts was considered instead of its sequence.

Inputs				Effect
Set of Contacts	orderBy	sort	field	Order changed?
Set{<"Pinto", "", "3", "", "", "">, <"Silva", "", "1", "", "", "">}	Business Phone ↑	Desc ↑	Last Name ↑	F ↑↑↑
Set{<"Pinto", "", "3", "", "", "">, <"Silva", "", "1", "", "", "">}	Business Phone	Desc ↓	Business Phone ↓	T ↓↓
Set{<"Silva", "", "1", "", "", "">, <"Pinto", "", "3", "", "", "">}	Business Phone	Asc ↓	Last Name	T ↓↓
Set{<"Silva", "", "1", "", "", "">, <"Pinto", "", "3", "", "", "">}	Last Name ↓	Desc	Last Name	T ↓
Set{<"Pinto", "", "1", "", "", "">, <"Silva", "", "3", "", "", "">}	Business Phone	Asc	Last Name	F ↓

Table 9: Test conditions for the Sort effect

Recall that it is possible to sort the contacts by a specific field in ascending or descending order. The sorting order is toggled every time two sorting operations are sequentially performed on the same database field. When the field changes between two sequentially sorting operations and independently of the last sorting order used, it becomes ascending. The information related to the previous sort operation is kept within two additional state variables called `orderBy` (the

field by which the address book was last sorted) and `sort` (that keeps the order of the last sort operation).

Boundary test conditions

Examples of boundary test conditions for the find effect are:

- The word to look for is at the beginning of the text field.
- The word to look for is at the end of the text field.
- The word to look for is equal to the text field.
- The word to look for is in the field of the currently selected line.

5.2.5. State filtering

An additional state filter was added to the Address Book software application limiting the size of the `dbContacts` variable that models the set of contacts inside the Address Book main window.

```
AddressBook.dbContacts.Size <= 2
```

State filter exclude from the exploration process all states where the specified state condition does not hold.

5.2.6. FSM generation and reduction

The generation of the full FSM for the domains and state filter defined previously in a single step was not practical so FSMs for subsets of the model were generated. One of those subsets containing the behaviour of the dialog Find and sort functionality is reported in section 5.2.10. Although the complete FSM was never generated, a FSM with enough size, i.e., covering all the testing goals defined, was used to illustrate the process of FSM validation in the next section.

5.2.7. FSM validation

The navigation map view of Figure 75 shows which windows/dialogs are enabled at each moment. The Find dialog is a modeless window such that, when it is enabled, the main window of the Address Book application remains enabled. The navigation map view has one state group where both windows/dialogs are enabled at the same time. There is also one state group for each other modal dialog window. It is obtained from the following property in Spec#:

```
Set<string> NavigationGroup { get {  
    return GetEnabledWindows();  
}}
```

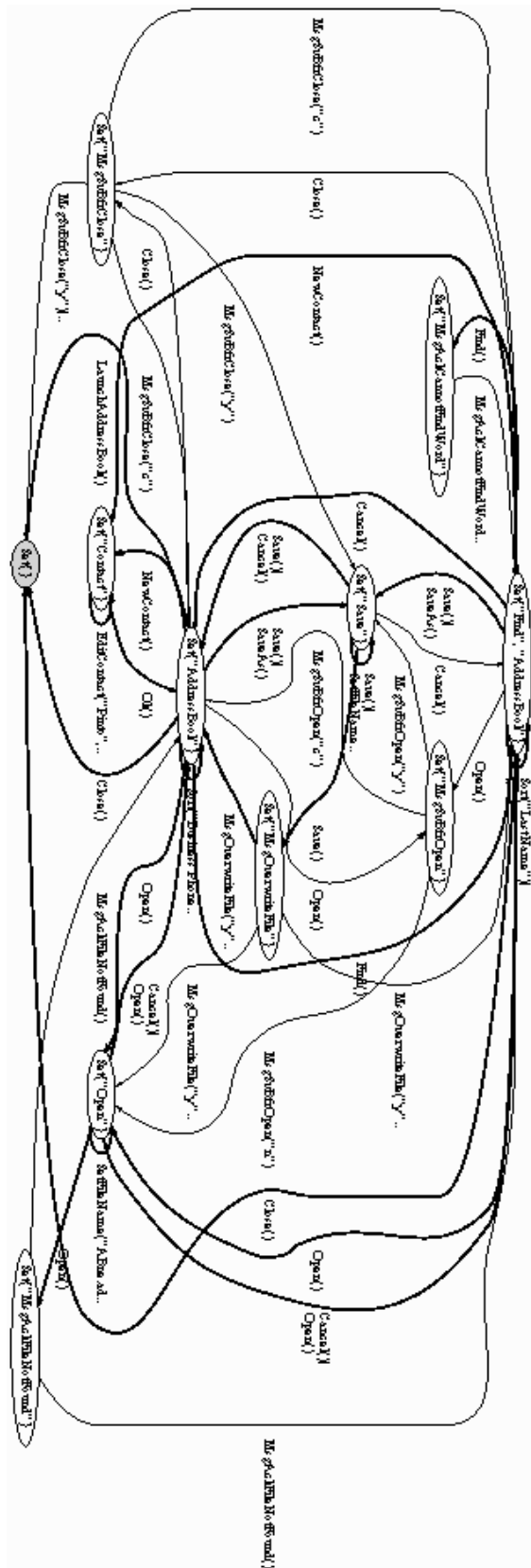


Figure 75: Navigation map view of the Address Book software application

The Open dialog view (Figure 76) shows the states and methods available inside the dialog. According to Table 7 (in page 168), the state variable `fileName` can be set to two different values "AB.adr" (an existing address book database) and "ABne.adr" (a non-existing address book database).

```
<string,string> OpenFileDialogGroup { get {
  if (IsOpen("Open"))
    return <"fileNameO="+fileNameO,"dirO="+dirO>;
  else return <"NotOpen","NotOpen">;
}}
```

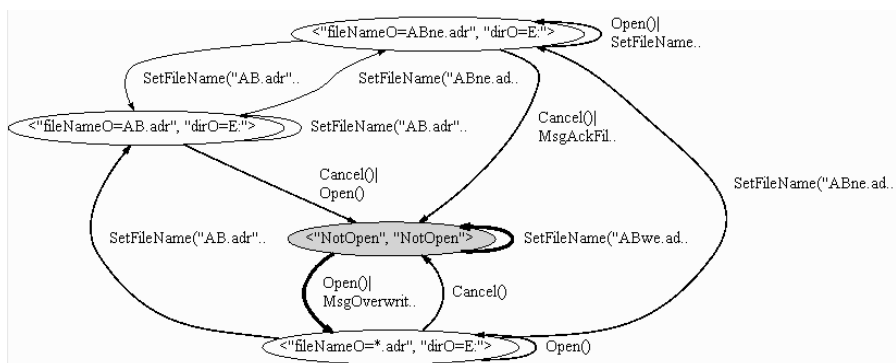


Figure 76: Open dialog view

The save dialog view shows the states and possible actions inside the Save dialog (Figure 77). It can be obtained from the following Spec# code:

```
<string,string> SaveDialogGroup { get {
  if (IsOpen("Save"))
    return <"fileName="+fileName,"dir="+dir>;
  else return <"NotOpen","NotOpen">;
}}
```

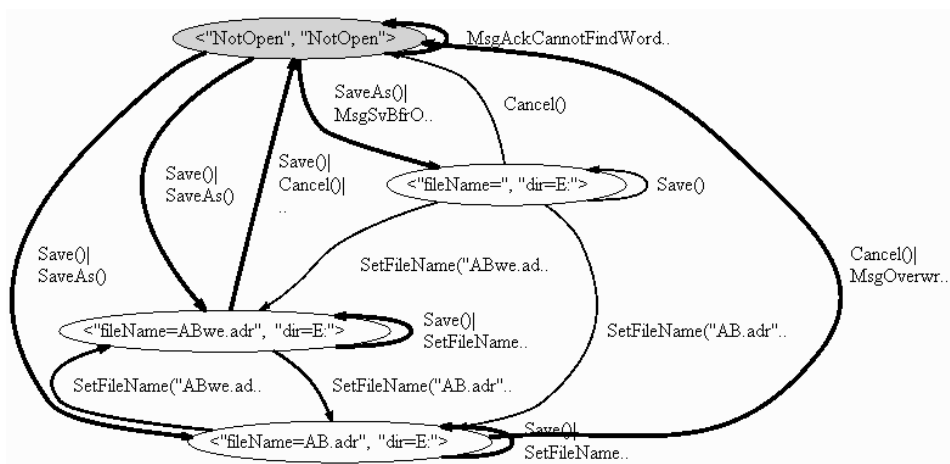


Figure 77: Save dialog view

In our point of view, the order in which the Contact dialog fields are filled in is not relevant for testing purposes. The only thing that is really important is the state of the dialog fields when the Ok button is pressed because that's the moment when the database in memory will be updated (add a new record or update an existing one). Hence, the states and transitions inside the Contact dialog can be reduced by constructing a scenario action (an action constructed as a sequence of controllable actions) that abstracts away the order by which fields are filled in. Substeps/subactions inside this scenario action have been disabled (the attribute action was removed) so as to avoid being explored outside of the scenario.

```
[Action(Kind=ActionAttributeKind.Scenario)]
void ScnEditContact(string LN, string FN, string BPh,
                  string HPh, string E, string F)
requires IsEnabled("Contact");{
    SetLastName(LN);
    SetFirstName(FN);
    SetBusinessPhone(BPh);
    SetHomePhone(HPh);
    SetEmail(E);
    SetFax(F);
}
```

According to Table 8 and Table 9, the domain of the scenario action parameters is defined as a set of four different tuples:

```
<"Pinto", "", "1", "", "", "">,
<"Pinto", "", "3", "", "", "">,
<"Silva", "", "1", "", "", "">,
<"Silva", "", "3", "", "", "">.
```

The Edit Contact dialog view in Figure 78 can be obtained by

```
<string,string,string,string,string,string>
ContactDialogGroup { get {
    if (IsOpen("Contact")) return contc;
    else return <"", "", "", "", "", "">;
}}
```

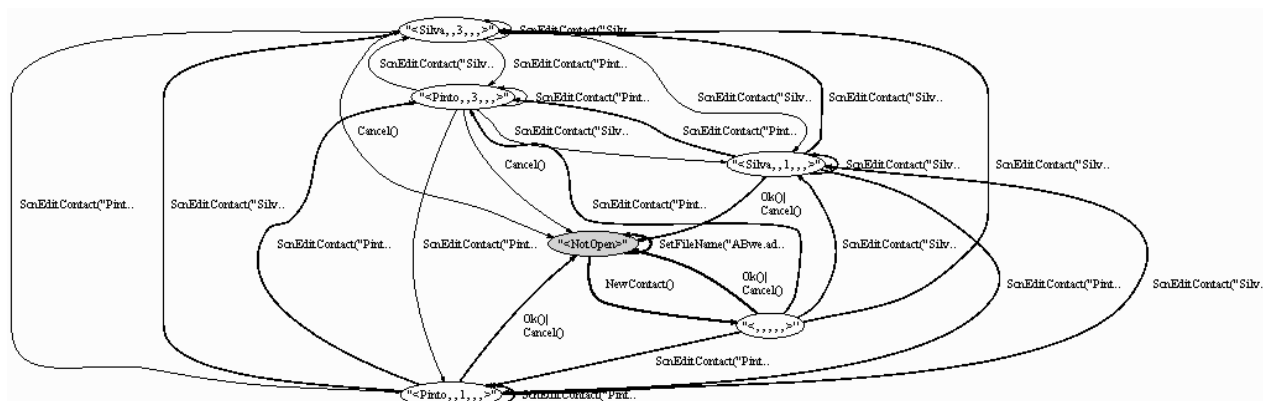


Figure 78: Contact dialog view

Similarly to the Edit Contact dialog, the order in which the Find dialog fields are filled in is irrelevant. So, an action scenario is built to set values to the fields and search the word in the database.

```
[Action(Kind=ActionAttributeKind.Scenario)]
public void ScnFind (string fw, string f, string d,
                   bool mc, bool mww)
  requires IsEnabled("Find") && fw != "";{
  findWhat = fw;
  field = f;
  direction = d;
  matchCase = mc;
  matchWholeWord = mww;
}
```

According to Table 8, the domains for the ScnFind action arguments are defined as a set of six different tuples:

```
Set{<"pin", "Last Name", "Down", false, false>,
<"pin", "Business Phone", "Down", false, false>,
<"pin", "Last Name", "Down", false, true>,
<"pin", "Last Name", "Down", true, false>,
<"pin", "Last Name", "Up", false, false>,
<"nuno", "Last Name", "Down", false, false>}
```

The Find dialog view in Figure 79 can be obtained by

```
string FindDialogGroup { get {
  if (IsOpen("Find")) return
    "<"+findWhat + ";" + field + ";" + direction + ";" +
    matchCase + ";" + matchWholeWord + ">";
  else return "NotOpen";
}}
```

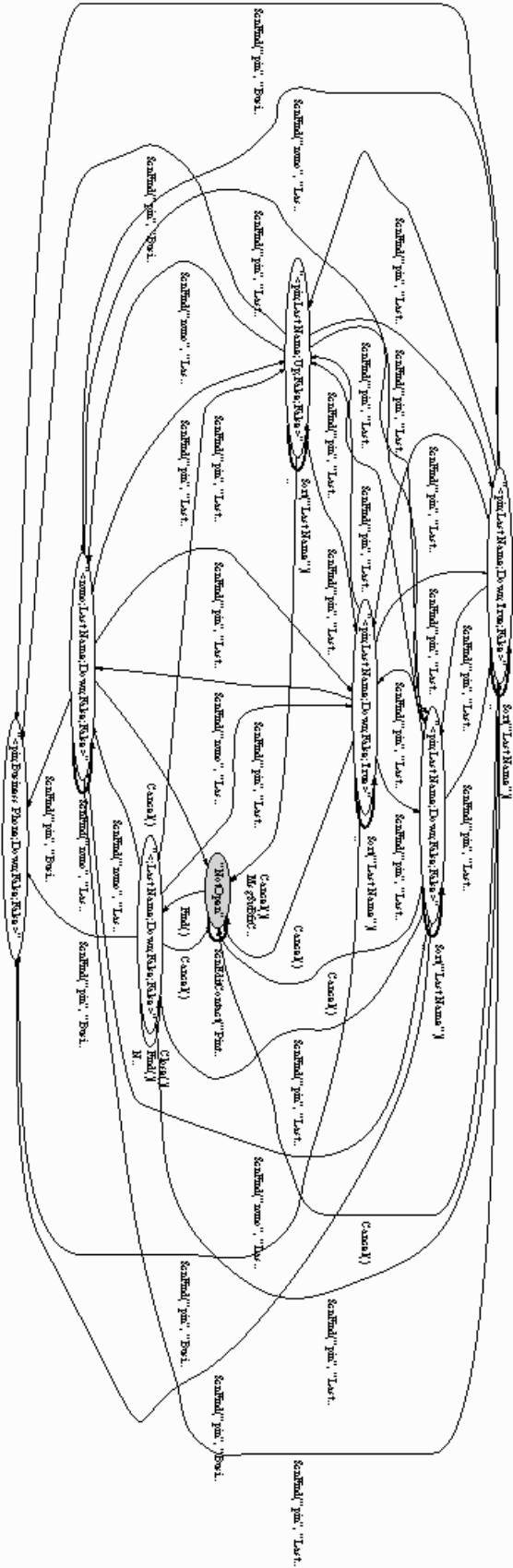


Figure 79: Find dialog view

Scenario validation

In order to check by visual inspection if the scenarios defined are covered by the FSM generated with the domain values defined in section 5.2.4, a view was defined for each scenario, as presented below:

```
string CloseScenarioView { get {
    if (IsOpen("MsgOverwriteFile"))
        return "MsgOverwriteFile?";
    else if (IsEnabled("MsgSvBfrClose"))
        return "MsgSvBfrClose?";
    else if (IsEnabled("Save")) return "Save";
    else if (!IsEnabled("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
```

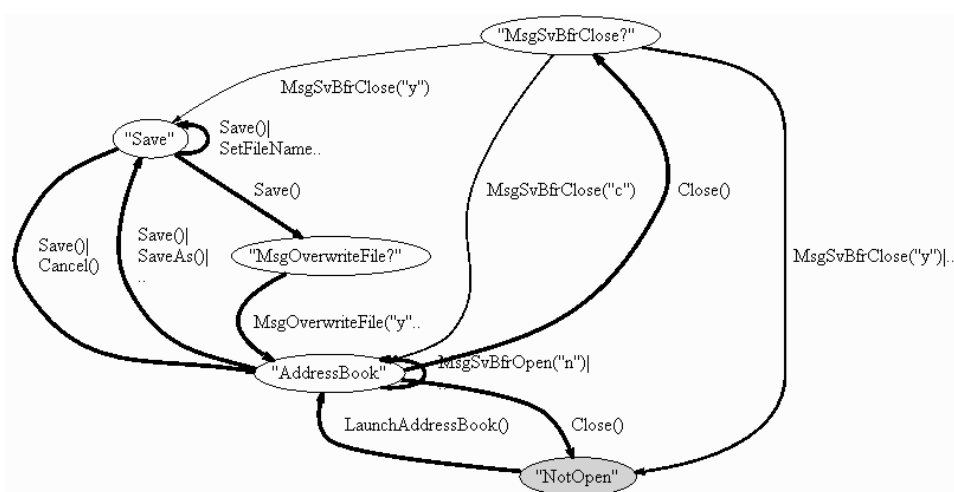


Figure 80: Close scenario view

```
string FindScenarioView { get {
    if (IsEnabled("MsgAckCannotFindWord"))
        return "MsgAckCannotFindWord";
    else if (IsEnabled("Find")) return "Find";
    else if (!IsOpen("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
```

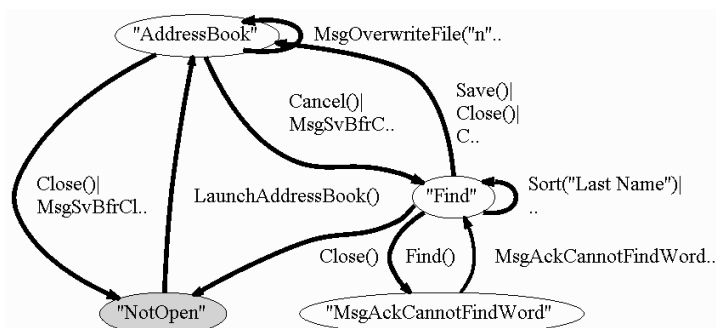


Figure 81: Find scenario view


```

string OpenScenarioView { get {
    if (IsEnabled("MsgAckFileNotFound"))
        return "MsgAckFileNotFound";
    else if (IsEnabled("Open")) return "Open";
    else if (IsEnabled("Save") && returnToOpenDlg)
        return "Save";
    else if (IsEnabled("MsgSvBfrOpen"))
        return "MsgSvBfrOpen";
    else if (IsEnabled("MsgOverwriteFile")
        && returnToOpenDlg) return "MsgOverwriteFile";
    else if (!IsOpen("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}

```

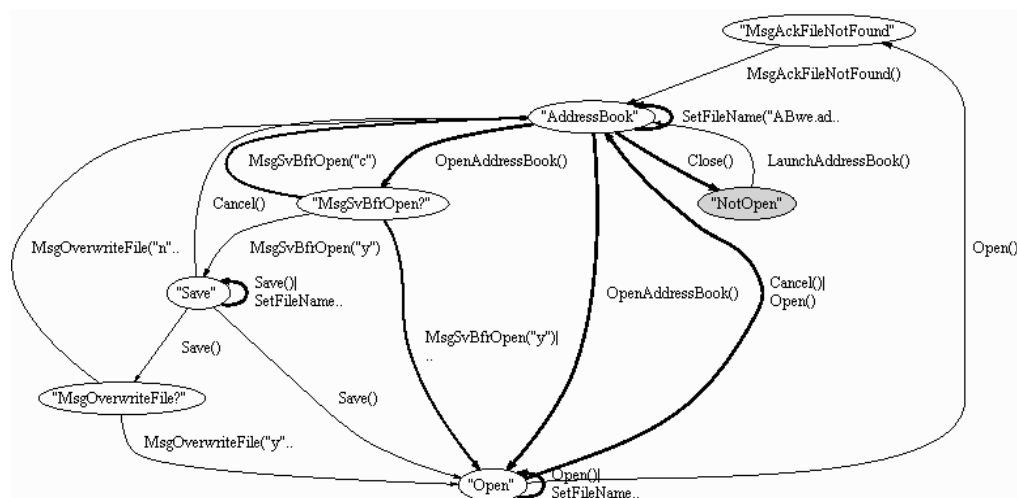


Figure 82: Open scenario view

```

string SaveScenarioView { get {
    if (IsEnabled("MsgOverwriteFile"))
        return "MsgOverwriteFile";
    else if (IsEnabled("Save")) return "Save";
    else if (IsEnabled("MsgOverwriteFile"))
        return "MsgOverwriteFile";
    else if (!IsOpen("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}

```

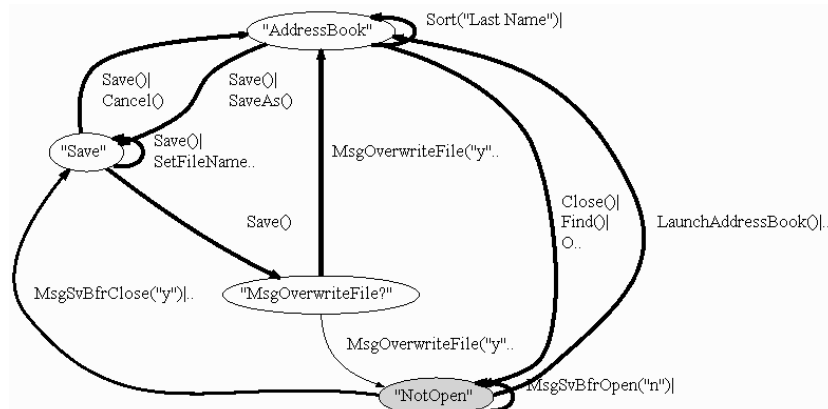


Figure 83: Save scenario view

5.2.8. Test case generation and execution

After assuring that the test goals are met, the algorithm presented in section 4.3.4 was applied on the FSM generated to reduce its size while guaranteeing coverage of the two intermediate levels of abstraction defined by the navigation map and dialog views. Then, test cases that meet full transition coverage criterion were generated from this FSM.

The map between model actions and interactive controls where the modelled actions will occur is established with the GUI Mapping Tool (Figure 84). This makes it possible to point out, for each model action, the interactive control where the modelled action will occur. Two XML files and a C# file are automatically generated for this purpose.

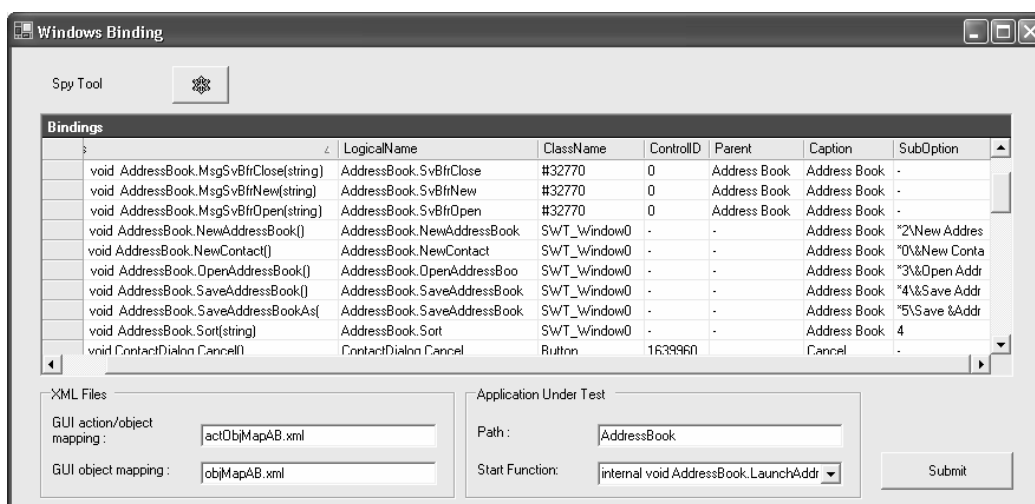


Figure 84: GUI Mapping Tool relating model action of the Address Book application with interactive controls

5.2.9. Capacity of detecting errors

Unlike the Notepad application, the source code of the Address Book is available, thus enabling testing by a particularly kind of fault injection called mutation testing (recall section 3.3). This makes it possible to assess how sharp the developed methodologies and tools are in interactive software error detecting (in a sense, this amounts to "testing the testing toolset" itself).

List of injected errors

The list of errors was constructed having in mind the kind of errors this approach is suited to find and classified as "functionality errors" in section 2.2.

The errors injected spread over several different types:

- Mandatory fields are not mandatory.
- Missing commands.
- Existing commands are disabled when they should be enabled.
- Commands do not do what was expected.
- Incorrect field defaults.
- Windows with incorrect modality.
- Message boxes do not show up when expected or do not show the set of options they should.
- Files are not correctly saved.

5.2.10. Metrics

The Address Book model was constructed in a single day. It reuses the modules Open, Save, and Window and File managers already constructed for the other case study (section 5.1). In addition to the modules reused, it was necessary to model more 38 actions and 20 helper methods to describe the behaviour of the Address Book.

The goal of this experiment is to test the find word and sort functionalities. The subset of actions (and parameter values) of the Address Book model used in this experiment is listed in Table 10.

Actions	Parameter domains
FindDialog.Cancel()	
ContactDialog.Cancel()	
AddressBook.Close()	
AddressBook.EditContact()	
AddressBook.Find()	
AddressBook.LaunchAddressBook()	
FindDialog.MsgAckCantFindWord()	
AddressBook.MsgSvBftClose(string)	{ "n", "c" }
AddressBook.MsgSvBfrNew(string)	{ "n", "c" }
AddressBook.NewContact()	
AddressBook.SelContact(int line)	if (dbContacts.Size>0) return Set{0..dbContacts.Size-1 } else return Set{-1 }
ContactDialog.Ok()	
ContactDialog.ScnEditContact(string,string, string,string)	{ <"Pinto", "", "3", "">, <"Pinto", "", "1", ""> <"Silva", "", "1", "">, <"Silva", "", "3", ""> }
	/* taken from Table 9*/

FindDialog.ScenFind(string,string,string, bool,bool)	<"pin","Last Name","Down",false,false>, <"pin","Business Phone","Down",false,false>, <"pin","Last Name","Down",false,true>, <"pin","Last Name","Down",true,false>, <"pin","Last Name","Up",false,false>, <"nuno","Last Name","Down",false,false>
AddressBook.Sort(Fields)	/* taken from Table 7 */
AddressBook.GetDBLastName()	{"Last Name", "Business Phone" }
AddressBook.GetDBBusinessPhone()	

Table 10: Actions and parameter domains used in the experiment to test find word and sort functionalities of the Address Book

The time needed to generate the FSM is 6 hours and 27 minutes. The FSM has 64797 states, 105317 transitions, and 44 invocations (actions with parameters).

The quality of the FSM was assessed according to coverage criteria defined in section 5.1.7 for the find word and sort functionalities. It took half an hour to conclude that the FSM had the desired quality properties.

The pruning technique described in section 4.3.4 was applied to the original FSM. After reduction, the FSM has 23059 states, 36922 transitions and 44 invocations. The number of transitions is reduced in 64.9% while the number of states is reduced in 64.4%. The time needed to reduce the FSM is irrelevant. The reduced FSM preserved the desired testing properties. Test suite generated from the reduced FSM has 69 segments with the total length of 55801.

All the injected errors were found with this experiment.

5.3. Conclusions

This chapter presented some experiments which illustrate and evaluate the specification-based testing approach proposed in this dissertation. Such experiments were performed on two different kinds of software applications (Microsoft's text editor Notepad, with source code unavailable, and a Java software application which manages database files of contacts, with source code available) and involved the construction of the corresponding software application models, test case generation, and execution.

Quantitative measures were provided for each experiment concerning the time needed to construct the models, the time needed to generate the FSMs, and the time needed to assess the quality of the FSM generated. In addition, the sizes of the models as well as the reduction achieved with the application of the reduction algorithm were provided.

Since the source code of the Address Book software application is available, a mutation testing technique was applied on the source code as a way to evaluate how sharp the approach is in fault detection. All injected defects were found with this experiment. The same approach was not followed for the Notepad application because its source code was not available. Although being used for several years, two so far unreported errors were detected in the Notepad application related to uncommon sequences of events.

The results achieved with the experiments performed gave us enthusiasm to continue our work in the field of model-based GUI testing.

Chapter VI

Conclusions and future work

This chapter presents a summary of the main contributions of the work reported in this dissertation in the fields of interactive software development and testing, and points out topics that deserve future attention.

The starting point of the work which leads to this dissertation was our analysis of current state-of-the-art methods for GUI development which revealed their lack of support for the modelling and verification phases (recall Chapter II). As a rule, the testing activity is performed manually without systematization. Moreover, no guarantee of adequate coverage with respect to some predefined criteria is given.

Although there have been efforts in constructing tools to automate the GUI testing process and diminish the resources (time and money) required, they suffer from many drawbacks that make them unsatisfactory solutions for the problem.

This dissertation reports on the application of specification-based testing techniques as a way to overcome such drawbacks and to make GUI testing more systematic, thus improving overall GUI quality.

6.1. Summary of contributions

The contributions of this research work fall into three areas:

- **GUI testing process** – The GUI testing process proposed in this dissertation is introduced in section 4.1 and detailed in its subsequent

sections. This process involves the following steps: construction of the GUI model, definition of test goals, definition of input domains, assessment of the quality of the FSM generated by exploration of the model, FSM reduction, test case generation, automatic construction of the intermediate code needed to simulate user's actions, test case execution, and analysis of the test results.

- **A set of GUI modelling techniques specially suited for testing purposes, promoting modularity and reusability** – Section 4.2 explains in detail how to model GUIs, in particular how to model windows, windows' controls, and communication among windows. The proposed modelling technique enables GUI description at different levels of abstraction where different properties under analysis (navigation between windows, use case scenarios, atomic user actions) can be expressed and then verified.
- **Specification-based GUI testing tools** – Two extensions to the Spec Explorer tool were developed: the first one (described in section 4.3.4) is an algorithm to reduce the FSM generated by the exploration of the Spec# model, while guaranteeing coverage of the intermediate level of abstraction defined by the navigation map and dialog views; the second extension (the GUI Mapping Tool described in section 4.4) assists the user in relating the model actions ("logical" actions) to "physical" actions of "physical" GUI objects. It then generates intermediate code that simulates the user actions over the GUI under test. This code is automatically bound to related actions in the specification.

These contributions address some of the GUI testing challenges identified in section 3.1, as follows:

- **GUI testing is known to be laborious, costly, extremely time-consuming and difficult to automate** – Our approach automates both test case generation and test case execution. The GUI Mapping Tool automates the execution of the test cases by controlling the GUI and observing the outputs automatically. Test cases generated include uncommon sequences of actions or events that would not be tested by manual tests. Errors detected when testing the Notepad application are reported as examples of errors related to such kinds of sequence.
- **Test case explosion** – The modelling technique allows for defining scenario actions, that is, actions built as sequences of smaller actions that abstract the order in which inputs are provided by eliminating all the other possible permutations. In addition, an algorithm is put forward to reduce the corresponding FSM while guaranteeing coverage of the navigation map and dialog views.
- **Controllability and observability** – The toolset described in this dissertation resorts to a GUI test library designed to control the GUI while simulating users' actions and observing properties of the GUI interactive controls.

- **Need for multiple testing techniques** – The approach proposed can be combined with scenario testing technique.
- **Documentation** – Models built according to our approach document the behaviour of the GUIs under test. Although interactive controls are not modelled in detail, the same approach could be used to model and test interactive controls and document their behaviour. This topic is illustrated in one of the papers published while carrying out the current research work [150]).

6.2. Summary of experimental results

The approach put forward in this research work was validated by two testing experiments on two software applications available under different contexts: the Notepad application that ships with Microsoft Windows (source code inaccessible) and the Address Book application developed for the Eclipse platform (example of a SWT application whose source code is available).

We stress the fact that two so far unreported errors were detected in the Notepad application, despite its widespread use for many years all over the world.

Our model of the Notepad application was built in a week (full time). Such a long time was needed because along the way we were also developing the modelling technique proposed in research work. By contrast, the model of the Address Book application reused some modules of the Notepad specification and was constructed in a single day.

Microsoft testers who use model-based testing tools for GUI testing have reported that modelling accounts for 10% of their work and fixing automation bugs for 90%. Without model-based testing tools, testers spend 60% of their time/effort writing the automation harnessing and 40% in writing tests. Thanks to our approach, the harnessing code can be built automatically. This means that most of the effort and time are spent on the construction of the model. It should be noted that models required by our approach are more detailed than models currently in use at Microsoft. Even so, the time saved during the construction of the harnessing code surpasses beyond doubts the additional time needed for the construction of the model itself.

6.3. Future Work

Although specification-based testing achieves a high level of testing automation, there is still a long way to go before it reaches widespread acceptance in industry-strong environments. Main obstacles to the introduction of specification-based testing techniques are:

- **The specification language itself** – We believe that specification languages should not involve a complete divorce from the current nature of programming languages used by industry programmers, otherwise these will resist to learn and use them. Some modellers resist constructing textual specifications, like those used in this research work in Spec#. They argue that specifying is too close to programming. Because they don't regard themselves as programmers, they would prefer to construct models using **graphical notations** like, for instance, Statecharts [87]. This points to a future direction in our research, that of investigating how to model GUIs in graphical notations and building mechanisms to translate such notations into Spec#, thus hiding the Spec# formalism from the modellers.
- **End-to-end support of specification-based testing in the test process** – Planning how models cover test goals (by test generation and coverage analysis based on test goals) and establishing communication channels among test managers (e.g., automatically providing reports for test management purposes like test cost, test coverage, and defects found) are important aspects of GUI testing. As future work, we intend to support **explicit definition of testing goals** to support the construction of reports with coverage analysis measures.
- **State space explosion of the model and test suite explosion** – Additional pruning techniques must be provided to control models and test suites size. Although two techniques have already been made available within the testing process proposed in this dissertation (scenario actions and a FSM reduction algorithm), we intend to construct an algorithm combining the exploration process itself with test coverage analysis (based on the explicit definition of test goals) so as to stop automatically the exploration process as soon as test goals are reached.
- **Time needed to build the model** – Specification-based testing methods can be criticized for the time and effort needed to construct the model of the system under test. As future work, we intend to derive techniques for reverse engineering existing GUI applications by automatic exploration, leading to automatic generation of Spec# models in a way similar to the one presented by Memon in [124]. Such models will in general be incomplete and only capture the coarse structure of the application; nevertheless, they can serve as starting point for further manual enhancements. This reverse engineering process will trim down the time needed to construct models and will allow us to apply our approach to more complex applications while saving on the effort to construct entire models from scratch.
- **Degree of automation** – The GUI testing method proposed in this research work involves manual definition of input domains. As future work, we will study ways to integrate test data generation approaches (see section 3.3.1) to allow coverage of the testing goals defined. The testing process also involves evaluation of the quality of the generated FSM in terms of meeting test goals previously identified (recall section 4.3.3 in this respect). For instance, one may wish to check if

the FSM covers the scenarios identified, special case situations, and so on. Right now, this phase is performed by expressing those properties as state group views in Spec# and then inspecting those views visually to check if they produce the expected result. Some mechanism to check such properties automatically is on demand.

- **Integration with other testing approaches** – The prototype tool developed in this work can be further extended in the future to transform the test cases generated into scripts written in the input language of a Capture/Replay tool for being executed and taking benefit of the observability capabilities of such tools. Moreover, test suites can be coded automatically and then be used by unit testing frameworks like JUnit and NUnit.

Other topics which deserve further attention are:

- **Usability testing** – The main target of the approach proposed in this research work is that of finding functionality errors, as described in section 2.2. However, further functionalities can be added to support additional analysis of the model in so far as to collect, for instance, information about the steps needed to reach a user goal (complete a task), thus predicting GUI usability.
- **Support for multiple platforms and languages** – The prototype tool developed so far only recognizes interactive controls with window handlers. This works for Windows applications and other software applications constructed with SWT (Standard Widget Toolkit) controls. By using existing libraries it is possible to extend this approach for other platforms, namely Java and Web applications.
- **Configuration testing** – The prototype tool developed so far does not explicitly deal with internationalization, e.g., command keys and data formats may change according to internationalization. To deal with these issues, the mapping tool should be extended to use system configurations (e.g., data formats) and help the user in "translating" user commands.

Pragmatically, we hope that the approach developed in this research work will be used effectively in industrial environments and henceforth contribute to higher quality interactive software. However, we are aware that the specification-based testing technique is not yet widely understood by testers and their managers. May this dissertation be also a contribution to disseminate the knowledge about methodologies and techniques to make testing activities more systematic, automatic, and less resource demanding.

Bibliography

1. G. Abowd, J. Bowen, A. Dix, M. Harrison, and R. Took, "User Interface Languages: A survey of Existing Methods", Programming research group, Oxford University Computing Laboratory, Oxford, Technical Report PRG-TR-5-89, 1989.
2. G. Abowd and A. J. Dix, "Integrating status and event phenomena in formal specifications of interactive systems", in Proceedings of the Symposium on Foundations of Software Engineering - SIGSOFT'94, D. Wile(Eds.), New Orleans, 1994.
3. G. Abowd, H.-M. Wang, and A. F. Monk, "A formal technique for automated dialog development", in Proceedings of the Designing interactive systems: processes, practices, methods & techniques, 1995.
4. B. K. Aichernig, "Automated Black-Box Testing with Abstract VDM Oracles", in Proceedings of the Workshop Materials: VDM in Practice! Part of the FM'99 World Congress on Formal Methods, I. J. F. a. P. G. L. editors(Eds.), Toulouse, September,1999.
5. B. K. Aichernig, "On the value of fault injection on the modeling level", in Proceedings of the Overture Workshop, N. Plat and P. G. Larsen(Eds.), Newcastle upon Tyne, UK, 18 July,2005.
6. Y. Ait-Ameur, M. Baron, and P. Girard, "Formal validation of HCI user tasks", in Proceedings of the International Conference on Software Engineering Research and Practice - SERP 2003, Las Vegas, Nevada, USA, 2003.
7. S. Alagar and K. Periyasamy, *Specification of Software Systems*,ed., Springer-Verlag, New York, Inc., pp.422, isbn:ISBN: 0-387-98430-5, 1998.
8. M. F. Ali, "A Transformation-based Approach to Building Multi-Platform User Interfaces Using a Task Model and the User Interface Markup Language", PhD thesis, Faculty of the Virginia Polytechnic Institute and State University, 2004
9. P. Ammann and P. E. Black, "Model Checkers in Software Testing", National Institute of Standards and Technology, Technical Report NIST-IR 6777, 2002.
10. P. E. Ammann, P. E. Black, and W. Majurski, "Using Model Checking to Generate Tests from Specifications", in Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98), M. G. H. John Staples, and Shaoying Liu(Eds.), Brisbane, Australia, 1998.
11. A. A. Andrews, J. Offutt, and R. T. Alexander, "Testing Web Application by Modeling with FSMs", *Software System Modeling*, vol. 4(3), pp. 326-345, 2005.

12. C. Artho, H. Barringer, A. Goldberg, K. Havelund, and S. Khurshid, "Automated Testing using Symbolic Model Checking and Temporal Monitoring", *submitted to Theoretical Computer Science*, 2004.
13. H. Balzer, F. Hofmann, V. Kruschinski, and C. Niemann, "The JANUS Application Development Environment-Generating More than the User Interface", in Proceedings of the CADUI'96, J. Vanderdonckt(Eds.), 1996.
14. M. Barnett, R. DeLine, B. Jacobs, M. Fähndrich, K. R. M. Leino, W. Schulte, and H. Venter, "The Spec# Programming System: Challenges and Directions", in Proceedings of the VSTTE2005, 2005.
15. M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview", in Proceedings of the CASSIS'04 - International workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, 10-13 Mar,2004.
16. R. Bastide and P. Palanque, "A Petri Net Based Environment for the Design of Event-Driven Interfaces", in Proceedings of the Application and Theory of Petri Nets — ATPN'95, Torino, Italy, 1995.
17. B. Bauer, "Generating User Interfaces from Formal Specifications of the Application", in Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96, J. Vanderdonckt(Eds.), 1996.
18. A. Beer, S. Mohacsi, and C. Stary, "IDATG: An Open Tool for Automated Testing of Interactive Software", in Proceedings of the COMPSAC'98 - The Twenty-Second Annual International Conference Computer Software and Applications, 19-21 Aug,1998.
19. F. Belli, "Finite State Testing and Analysis of Graphical User Interfaces", in Proceedings of the ISSRE 2001 - The 12th International Symposium on Software Reliability Engineering, Hong Kong, 27-30 Nov,2001.
20. E. Bernard, B. Legeard, X. Luck, and F. Peureux, "Generation of test sequences from formal specifications: GSM 11-11 standard case study", *Software Testing, Verification and Reliability*, vol. 34(10), pp. 915-948, 2004.
21. J. Berstel, S. C. Reghizzi, G. Roussel, and P. S. Pietro, "A Scalable Formal Method for Design and Automatic Checking of User Interfaces", in Proceedings of the ICSE'01, 2001.
22. D. W. Binkley and K. B. Gallagher, "Program Slicing", *Advances in Computers*, vol. 43, pp. 1-50, 1996.
23. E. Bishop, "News: conferences - Report on the fourth International Conference on Software Testing (ICSTEST)", in *Professional Tester*, 2003, pp. 6-7.
24. P. E. Black, V. Okun, and Y. Yesha, "Mutation of Model Checker Specifications for Test Generation and Evaluation", in

-
- Proceedings of the Mutation 2000, W. E. Wong(Eds.), Jan Jose, California, 2000.
25. F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, B. Sacré, and J. Vanderdonckt, "Towards a Systematic Building of Software Architecture: the TRIDENT Methodological Guide", in Proceedings of the Workshop on Design, Specification and Verification of Interactive Systems DSVIS'95, P. Baside(Eds.), Toulouse, France, 1995.
 26. K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. CGheorghe, M. Harman, R. M. Hierons, K. Kapoor, P. Krause, G. Luetgen, and A. J. H. Simons, "Working together: Formal Method and Testing", *ACM Computing Surveys*, 2005.
 27. T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", *Computer Networks ISDN Systems. Special Issue: Protocol Specification and Testing*, vol. 14(1), pp. 25-59, 1987.
 28. E. Börger and R. Staerk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, 1st ed., Springer, isbn:3540007024, 2003.
 29. J. P. Bowen, "X: Why Z?" in Proceedings of the Computer Graphics Forum, 1992.
 30. C. J. Bramwell, "Formal Development Methods for Interactive Systems: Combining Interactors and Design Rational", PhD thesis, University of York, The Department of Computer Science, 1996
 31. J. Bredereke and B.-H. Schlingloff, "An Automated, Flexible Testing Environment for UMTS", in Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV, 2002.
 32. J. Brown, "Evaluation of the Task-Action Grammar Method for Assessing Learnability in User Interface Software", in Proceedings of the 6th Australian Conference on Computer-Human Interaction (OZCHI'96), 1996.
 33. P. Bumbulis, P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena, "Combining Formal Techniques and Prototyping in User Interface Construction and Verification", 1995.
 34. P. Bumbulis, P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena, "A Framework for Machine-Assisted User Interface Verification", in Proceedings of the 4th International Conference on Algebraic Methodology and Software Technology (AMAST'95), London, UK, 1995.
 35. R. Butterword, A. Blandford, and D. Duke, "The role of formal proof in modelling interactive behaviour", in Proceedings of the Design, Specification and Verification of Interactive Systems (DSV-IS), P. Markopoulos and P. Johnson(Eds.), February, 1998.
 36. R. Butterword, A. Blandford, D. Duke, and R. Young, "Formal user models and methods for reasoning about interactive behaviour", in Proceedings of the WP17, 1998.

37. R. J. Butterworth and D. J. Cooke, "Using Temporal Logic in the Specification of Reactive and Interactive Systems", in Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, S. H. U. C.R. Roast and J.I. Siddiqi, UK(Eds.), 1996.
38. M. Cabrera, M. Gea, F. Gutierrez, and J. C. Torres, "Algebraic specification of User Interfaces", in Proceedings of the 1st ERCIM Workshop on "User Interfaces for All", C. Stephanidis(Eds.), Crete, Greece, October 30-31,1995.
39. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer", Microsoft Research, MSR-TR-2005-59, May, 2005.
40. A. Campi, E. Martinez, and P. S. Pietro, "Experiences with a Formal Method for Design and Automatic Checking of User Interfaces", in Proceedings of the Position paper in IUI/CADUI'2004 Workshop on Making Model-Based UI Design Practical: usable and open methods and tools, 13th January,2004.
41. J. Campos and M. D. Harrison, "Model Checking Interactor Specifications", in *Automated Software Engineering*, vol. 8, 2001.
42. J. F. C. F. d. Campos, "GAMA-X Geração Semi-Automática de Interfaces Sensíveis ao Contexto", MSc, Universidade do Minho, Departamento de Informática, 1993
43. D. A. Carr, "Specification of Interface Interaction Objects", in Proceedings of the ACM Conference on Human Factors in Computing Systems - CHI, Boston, Massachusetts, USA, 1994.
44. S. S. Chok and K. Marriott, "Automatic Construction of User Interfaces from Constraint Multiset Grammars", in Proceedings of the 11th International IEEE Symposium on Visual Languages (VL'95), Washington, DC, USA, 1995.
45. E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti, "From formal models to formally-based methods: an industrial experience", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 8(1), pp. 79-113, 1999.
46. K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs", in Proceedings of the ICFP'00, Montreal, Canada, 2000.
47. E. M. Clarke, O. Grumberg, M. Minea, and D. Pled, "State space reduction using partial order techniques", *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2(3), pp. 279-287, 1998.
48. T. Clement, "The Formal Development of a Windows Interface", in Proceedings of the 3rd BCS-FACS Northern Formal Methods Workshop, 1998.
49. L. Constantine, "Rapid Abstract Prototyping", Technical Report #100, 1998.

50. L. L. Constantine and L. A. D. Lockwood, "Usage-Centered Engineering for Web Applications", in *IEEE Software*, vol. 19, 2002.
51. J. Coutaz, "Software Architecture Modeling for User Interfaces", in *Encyclopedia of software Engineering*(Eds.), Wiley and sons, 1993.
52. T. Dabóczy, I. Kollár, G. Simon, and T. Megyeri, "How to test Graphical User Interfaces", in *IEEE Instrumentation & Measurement Magazine*, 2003, pp. 27-33.
53. L. Dan and B. K. Aichernig, "Combining Algebraic and Model-based Test Case Generation", in Proceedings of the First International Colloquium in Theoretical Aspects of Computing (ICTAC'04), Guiyang, China, 2004.
54. R. DeMillo and J. Offutt, "Constraint-Based Automatic Test Data Generation", in *IEEE Transactions on Software Engineering*, vol. 17, 1991, pp. 900-910.
55. J. Dick and A. Faivre, "Automating the generation and sequencing of test cases from model-based specifications", in Proceedings of the FME'93: Industrial-Strength Formal Methods, Odense, Denmark, 1993.
56. A. W. Dijkstra, "Notes On Structured Programming", Technological University Eindhoven, The Netherlands, Department of Mathematics, 70-WSK-03, 1970.
57. A. Dix and C. Runciman, "Abstract Models of Interactive Systems, People and Computers: Designing the Interface", in Proceedings of the HCI'85, P. J. S. Cook(Eds.), London, 1985.
58. G. J. Doherty, J. Campos, and M. D. Harrison, "Representational Reasoning and Verification", *Formal Aspects of Computing*, vol. 12(4), pp. 260-277, 2000.
59. D. Duke and M. Harrison, "Towards a Theory of Interactors", Amodeus Esprit Basic Research Project /WP6, February 12, 1993.
60. D. J. Duke and M. D. Harrison, "Abstract Interaction Objects", in Proceedings of the EUROGRAPHICS'93, 1993.
61. M. B. Dwyer, V. Carr, and L. Hines, "Model Checking Graphical User Interfaces Using Abstractions", in Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE), 1997.
62. J. Edvardsson, "A Survey on Automatic Test Data Generation", in Proceedings of the Second Conference on Computer Science and Engineering in Linkoping (ECSEL), October, 1999.
63. M. Evers, "Adaptability Problems of Architectures for Interactive Software", in Proceedings of the Workshop on Object-Oriented Technology (ECOOP'99), Lisbon, Portugal, 1999.
64. J. C. P. d. Faria, "Regras Activas Dirigidas pelos Dados para a Manutenção de Restrições de Integridade e Dados Derivados em Aplicações Interactivas de Bases de Dados", PhD, Faculdade de

Eng^a da Universidade do Porto, Departamento de Eng^a Electrotécnica e de Computadores, 1999

65. R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5(1), pp. 63-86, 1996.
66. J. Fitzgerald and P. G. Larsen, "Modelling Systems: Practical Tools and Techniques in Software Development", *Cambridge University Press*, 1998.
67. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs for Object-oriented Systems*, ed., Springer-Verlag Telos, isbn:1-85233-881-4, New York, 2005.
68. J. S. Fitzgerald and P. G. Larsen, "Formal specification techniques in the commercial development process", in *Proceedings of the International Conference on Software Engineering (ICSE)*, Seattle, April, 1995.
69. S. Flynn, "Expression Refinement Explained", National University of Ireland, Department of Information Technology, Galway, Technical Report, 1999.
70. S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models", *IEEE Transactions on Software Engineering*, vol. 17(6), pp. 591-603, 1991.
71. A. Galloway and B. Stoddart, "Integrated Formal Methods", IRIN - Institute de Recherche en Informatique de Nantes, 1997.
72. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison-Wesley Professional, pp.395, isbn:0201633612, 1995.
73. J. Gannon, P. McMullin, and R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing", *ACM Transactions on Programming Languages and Systems*, vol. 3(3), pp. 211-223, 1981.
74. G. Gediga, K.-C. Hamborg, and I. Dutsch, "Evaluation of Software Systems", in *Encyclopedia of Library and Information Science*, vol. 72, A. Kent and J. G. Williams (Eds.), 2002, pp. 166-102.
75. D. F. Gieskens and J. D. Foley, "Controlling User Interface Objects Through Pre- and Postconditions", GVU Technical Report Number GIT-GVU-91-09, June, 1991.
76. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jounaud, "Introducing OBJ", in *Applications of Algebraic Specification using OBJ*, R. G. a. J. G. D. Coleman (Eds.), Cambridge University Press, 1993.
77. C. Gram and G. Cockton, *Design Principles for Interactive Software*, ed., Chapman & Hall, isbn:0412724707, 1996.

78. A. S. Grant, "Modelling Cognitive Aspects of Complex Control Tasks", University of Strathclyde, Department of Computer Science, 1990
79. W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, "Generating Finite State Machines from Abstract State Machines", in Proceedings of the ISSTA 2002, International Symposium on Software Testing and Analysis, July, 2002.
80. T. Griffiths, "Teallach: A Model-Based User Interface Development Environment for Object Databases", in Proceedings of the User Interfaces to Data Intensive Systems - UIDIS, 1999.
81. T. Griffiths, J. McKirdy, N. Paton, J. Kennedy, R. Cooper, P. Barclay, C. Goble, P. Gray, M. Smyth, A. West, and A. Dinn, "An Open Model-Based Interface Development System: The Teallach Approach", in Proceedings of the DSV-IS'98, 1998.
82. J. V. Guttag, J. J. Horning, and A. Modet, "Report on the Larch Shared Language - Version 2.3", SRC Research Report 58, April 14, 1990.
83. G. d. Haan, "Formal Representation of Human-Computer Interaction", in Proceedings of the Human-Computer Interaction: Preparing for the Ninties, T. N. W. a. A. G. A. G. C. van der Veer(Eds.), Amsterdam, 1991.
84. G. d. Haan, "ETAG, A Formal Model of Competence Knowledge for User Interface Design", Vrije Universiteit, 2000
85. G. d. Haan, G. C. v. d. Veer, and J. C. v. Vliet, "Formal Modelling Techniques in Human Computer Interaction", (<http://home.tiscali.nl/gdehaan/articles/formal-models-review.html>), conferred in October, 2006.
86. P. R. Hanau and D. R. Lenorovitz, "Prototyping and simulation tools for user/computer dialogue design", in Proceedings of the 7th annual International Conference on Computer Graphics and Interactive Techniques, Seattle, Washington, USA, 1980.
87. D. Harel, "Statecharts: a visual formalism for complex systems", in *Science of Computer Programming*, vol. 8(Eds.), 1987, pp. 231-274.
88. M. Harman and S. Danicic, "Using Program Slicing to Simplify Testing", *Software Testing, Verification and Reliability*, vol. 5(143-162), 1995.
89. A. Hartman and K. Nagin, "The AGEDIS Tools for Model Based Testing", in Proceedings of the ISSTA'04, Boston, Massachusetts, USA, July 11-14, 2004.
90. H. R. Hartson, A. C. Siochi, and D. Hix, "The UAN: a user-oriented representation for direct manipulation interface designs", in *ACM Transactions on Information Systems (TOIS)*: ACM Press, 1990.
91. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition / Decision Coverage", NASA/TM-2001-210876, 2001.

92. E. Hendrickson, "Making the Right Choice", in *Software Testing & Quality Engineering*, 1999.
93. R. M. Hierons, "Testing From a Z Specification", *Journal of Software Testing, Verification, and Reliability*, vol. 7(1), pp. 19-33, 1997.
94. C. A. R. Hoare, *Communicating Sequential Processes (CSP)*, ed., Prentice Hall International, 2004.
95. A. Howes, S. J. Payne, and D. Moffat, "Automated Theory-based Procurement Evaluation", in Proceedings of the Interact'97, 1997.
96. A. Hussey and D. Carrington, "Comparing two user-interfaces: MVC and PAC", in Proceedings of the FAHCI'96, 1996.
97. A. Hussey, I. MacColl, and D. Carrington, "Assessing Usability from Formal User-Interface Designs", in Proceedings of the Interact'01, 2001.
98. K. Iizuka, J. Tanaka, and B. Shizuki, "Describing a Drawing Editor by Using Constraint Multiset Grammars", in Proceedings of the Sixth International Symposium on the Future of Software Technology (ISFST), Zhengzhou, China, November, 2001.
99. M. Y. Ivory and M. A. Hearst, "The State of the Art in Automating Usability Evaluation of User Interfaces", *ACM Computing Surveys*, vol. 33(4), pp. 470-516, 2001.
100. C. Janssen, A. Weisbecker, and J. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications", in Proceedings of the Proceedings of the CHI'93, New York, NY, 1993.
101. R. Jeffries, J. R. Miller, C. Wharton, and K. M. Uyeda, "User Interface Evaluation in the Real World: A Comparison of Four Techniques", 1991.
102. B. E. John and D. E. Kieras, "The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast", *ACM ToCHI*, 1996.
103. C. W. Johnson and M. D. Harrison, "Using Temporal Logic To Support The Specification and Prototyping Of Interactive Control Systems", *International Journal Of Man-Machine Studies*, vol. 36, pp. 357-385, 1992.
104. P. Johnson, H. Johnson, R. Waddington, and Shouls, "Task Related Knowledge Structures: Analysis, Modelling and Application", in *People and Computers IV*, Cambridge University Press, 1988, pp. 35-61.
105. C. Kaner, "Improving the Maintainability of Automated Test Suites", in Proceedings of the Tenth International Quality Week, San Francisco, CA, May, 1997.
106. C. Kaner, "Cem Kaner on Scenario Testing: The Power of "What If. " and Nine Ways to Fuel Your Imagination", in *Software Testing & Quality Engineering (STQE)*, 2003.

107. C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*, ed., John Wiley & Sons, 2002.
108. C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software*, ed., Wiley Computer Publishing, isbn:0-471-35846-0, 1999.
109. H. C. Keh and T. G. Lewis, "Direct-Manipulation User Interface Modeling with High-Level Petri Nets", in Proceedings of the 19th annual conference on Computer Science, San Antonio, Texas, United States, 1999.
110. W. C. Kim and J. D. Foley, "Providing high-level control and expert assistance in the user interface presentation design", in Proceedings of the Human Factors in Computing Systems (InterCHI'93 Proceedings), New York, 1993.
111. B. A. Kitchenham, "Evaluating Software Engineering Methods and Tool. Part 1: The Evaluation Context and Evaluation Methods", *ACM SIGSOFT Software Engineering Notes*, vol. 21(1), 1996.
112. D. Lee and M. Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey", *Proceedings of the IEEE*, vol. 84, pp. 1090-1996, 1996.
113. F. Lonczewski, "The FUSE-System: an Integrated User Interface Design Environment", in Proceedings of the CADUI'96, 1996.
114. M. Cabrera, M. Gea, and J. C. Torres, "Towards User Interfaces Prototyping for Algebraic Specification", in Proceedings of the VI Eurographics Workshop on Design, Specification and Verification of Interactive Systems - DSV-IS'99, Braga, Portugal, June, 1999.
115. I. MacColl and D. Carrington, "User Interface Correctness", in Proceedings of the Human Computer Interaction - HCI'97, 1997.
116. I. MacColl and D. Carrington, "Specifying Interactive Systems on Object-Z and CSP", in Proceedings of A. G. a. K. T. K. Araki(Eds.), 1999.
117. P. Markopoulos, J. Pycock, S. Wilson, and P. Jonhson, "Adept — A task based design environment", in Proceedings of the 25th Hawaii International Conference on System Sciences, 1992.
118. C. Märtin, "Software Life Cycle Automation for Interactive Applications: The AME Design Environment", in Proceedings of the Computer-Aided Design of User Interfaces - CADUI'96, J. Vanderdonckt(Eds.), 1996.
119. F. M. Martins, "Semi-Automatic Design and Prototyping of Adaptive User Interfaces", in Proceedings of the 2nd ERCIM Workshop on "User Interfaces for All", C. Stephanidis(Eds.), Prague, Czech Republic, 7-8 November, 1996.
120. F. M. J. Martins, "Métodos Formais na Concepção e Desenvolvimento de Sistemas Interactivos", PhD, Escola de Engenharia da Universidade do Minho, 1995

121. A. v. Mayhauser, M. Scheetz, and E. Dahlman, "Generating Goal-oriented Test Cases", in Proceedings of the The Twenty-Third Annual International Computer Software and Applications Conference (COMPSAC'99), 27-29 Oct,1999.
122. T. McCarthy, "Intro to NEXTSTEP", (www120.pair.com/mccarthy/nextstep/intro.html), conferred in October, 2006.
123. A. Memon, "Using Tasks to Automate Regression Testing of GUIs", in Proceedings of the The IASTED International Conference on Artificial Intelligence and applications (AIA2004), Innsbruck, Austria, Feb. 16-18,2004.
124. A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing", in Proceedings of the WCRE2003 - The 10th Working Conference on Reverse Engineering, Victoria, British Columbia, Canada, 13-16 Nov,2003.
125. A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces", Pittsburgh, 2001
126. A. M. Memon, M. E. Pollack, and M. L. Soffa, "Using a Goal-driven Approach to Generate Test Cases for GUIs", in Proceedings of the International Conference on Software Engineering, Los Angeles, 1999.
127. A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated Test Oracles for GUIs", in Proceedings of the FSE, 2000.
128. A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning", *IEEE Transactions on Software Engineering*, vol. 27(2), 2001.
129. A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage Criteria for GUI Testing", in Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), Sept,2001.
130. C. Meudec, "ATGen: automatic test data generation using constraint logic programming and symbolic execution", *Software Testing, Verification and Reliability*, vol. 11(2), pp. 81-96, 2001.
131. M. Mezzanotte and F. Paternó, "Verification of Properties of Human-Computer Dialogs with an Infinite Number of States", in Proceedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, S. H. U. C.R Roast and J.I. Siddiqi, UK(Eds.), 10-12 September,1996.
132. Microsoft, "Visual Basic Home", (msdn2.microsoft.com/en-us/vbasic), conferred in October, 2006.
133. Microsoft, "Visual Studio", (msdn.microsoft.com/vstudio), conferred in October, 2006.
134. K. Mitchell and J. Kennedy, "DRIVE: An Environment for the Organised Construction of User-Interfaces to Databases", in Proceedings of the 3rd International Workshop on Interfaces to

- Databases, J. K. a. P. Barclay(Eds.), Napier University, Edinburgh, 1996.
135. P.-J. Molina-Moreno, I. Torres-Boigues, and O. Pastor-López, "User Interface Patterns for Object-Oriented Navigation", in *Human-Computer Interaction: Overcoming Barriers*, 2003.
 136. P. J. Molina and J. Hernández, "Just-UI: Using patterns as concepts for IU specification and code generation", in Proceedings of the CHI 2003 workshop on HCI Patterns: Concepts & Tools, Fort Lauderdale, Florida, 2003.
 137. P. J. Molina, S. Martí, and Ó. Pastor, "Prototipado Rápido de Interfaces de Usuario", in Proceedings of the V Workshop Iberoamericano de Ingeniería de Ambientes Software, IDEAS'2002, M. K. e. al.(Eds.), La Habana, Cuba, Abril,2003.
 138. T. P. Moran, "Getting into a System: External-Internal Task Mapping Analysis", in Proceedings of the CHI'83, December,1983.
 139. C. Morgan, *Programming from Specification*, 2nd edition ed., Prentice Hall, isbn:ISBN: 0131232746, 1994.
 140. B. A. Myers, "User Interface Software Tools", *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 2(1), pp. 64-103, 1995.
 141. B. A. Myers and M. B. Rosson, "Survey on user interface programming", in Proceedings of the SIGCHI'92, 1992.
 142. N. Nyman, "In Defense of Monkey Testing", conferred in May, 2006.
 143. N. Nyman, "Using Monkey Test Tools", in *STQE - Software Testing and Quality Engineering Magazine*, 2000.
 144. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications", *Software Testing, Verification and Reliability*, vol. 13(1), pp. 25-53, 2003.
 145. V. Okun, P. E. Black, and Y. Yesha, "Testing with Model Checker: Insuring Fault Visibility", *WSEAS Transactions on Systems*, vol. 2(1), pp. 77-82, 2003.
 146. D. R. Olsen and E. P. Dempsey, "SYNGRAPH: A Graphical User Interface Generator", in *ACM - Computer Graphics*, vol. 17, 1983, pp. 43-50.
 147. T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A Visual Test Development Environment for GUI Systems", in Proceedings of the ISSTA'98, Clearwater Beach Florida, USA, 1998.
 148. A. C. Paiva, J. P. Faria, and R. M. Vidal, "Specification-based Testing of User Interfaces", in Proceedings of the 10th DSV-IS Workshop - Design, Specification and Verification of Interactive Systems, Funchal - Madeira, 4-6 de Junho,2003.
 149. A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A Model-to-implementation Mapping Tool for Automated

- Model-based GUI Testing", in Proceedings of the ICFEM'05, 2005.
150. A. C. R. Paiva, J. C. P. Faria, and R. M. Vidal, "Automated Specification-based Testing of Interactive Components with AsmL", in Proceedings of the 5th edition of the Quatic (Quality: the bridge to the future in ICT) international conference, Porto, 2004.
 151. A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal, "Modeling and Testing Hierarchical GUIs", in Proceedings of the ASM 2005 - 12th International Workshop on Abstract State Machines, Paris - France, March 8-11,2005.
 152. P. Palanque, "Petri Net Based Design Of User-Driven Interfaces Using Interactive Cooperative Objects Formalism", in Proceedings of the Design, Specification and Verification of Interactive Systems - DSV-IS'94, 1994.
 153. D. L. Parnas, "On the use of transition diagrams in the design of a user interface for an interactive computer system", in Proceedings of the 24th National Conference, New York, NY, USA, 1969.
 154. F. Paternò, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models", in Proceedings of the Interact'97, 1997.
 155. F. Paternò and C. Santoro, "Integrating Model Checking and HCI Tools to Help Designers Verifying User Interface Properties", in *7th International Workshop on Design, Specification and Verification of Interactive Systems DSV-IS'2000*. Limerick, Ireland, 2000.
 156. D. Peled, E. Clarke, and O. Grumberg, *Model checking*, ed., MIT Press, isbn:02620327 -08, Cambridge, Massachusetts,2000.
 157. I. Phillips, "A comparative review of HyperCard and Director as tools for time-based expressive work", (www.agocg.ac.uk/reports/graphics/26/node12.htm), conferred in October, 2006.
 158. N. Plat and P. G. Larsen, "An Overview of the ISO/VDM-SL Standard", *ACM SIGPLAN Notices*, vol. 27(8), pp. 76-82, 1992.
 159. A. Pretschner, "Classical search strategies for test case generation with Constraint Logic Programming", in Proceedings of the CONCUR'01 Workshop on Formal Approaches to Testing of Software (FATES'01), Aalborg, Denmark, August,2001.
 160. M. Priestley, "The Logic of Correctness in Software Engineering", in Proceedings of the 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17,2005.
 161. A. R. Puerta, "The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development", in Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces - CADUI'96, 1996.
 162. A. R. Puerta, "Supporting User-Centered Design of Adaptive User Interfaces Via Interface Models", in Proceedings of the First

-
- Annual Workshop On Real-Time Intelligent User Interfaces for Decision Support and Information Visualization, San Francisco, January,1998.
163. A. R. Puerta and D. Maulsby, "Management of Interface Design Knowledge with MOBI-D", in Proceedings of the IUI'97, Orlando, FL, January,1997.
164. S. Rayadurgam and M. P. E. Heimdahl, "Test-Sequence Generation from Formal Requirements Models", in Proceedings of the Sixth IEEE High Assurance in Systems Engineering Workshop, Florida, October,2001.
165. P. Reisner, "Further Developments Toward Using Formal Grammar as a Design Tool", in Proceedings of the Conference on Human Factors in Computing Systems, Gaithersburg, Maryland, United States, 1982.
166. J. Schalken, "Research Methods for the Empirical Assessment of Software Processes", in Proceedings of the Proceedings of the 12th doctoral consortium on Advanced Information Systems Engineering - CAISE'05, H. BOUNIF(Eds.), Porto - Portugal, 13-14 June,2005.
167. E. Schlungbaum, "Model-based User Interface Software Tools Current state of declarative models", Graphics, Visualization & Usability Center, GIT-GVU-96-30, 1996.
168. B. v. Schooten, O. Donk, and J. Zwiers, "Modelling Interaction in Virtual Environments using Process Algebra", in Proceedings of the Interactions in Virtual Worlds - TWLT 15, 1999.
169. M. L. Scott and S.-K. Yap, "A Grammar-based Approach to the Automatic Generation of User-Interface Dialogs", in Proceedings of the CHI'88, 1988.
170. R. K. Shehady and D. P. Siewiorek, "A Method to Automate User Interface Testing Using Variable Finite State Machines", in Proceedings of the 27th International Symposium on Fault-Tolerant Computing, 1997.
171. P. P. d. Silva, "User Interface Declarative Models and Development Environments: A Survey", University of Manchester, 2000.
172. D. Sinnig, P. Forbrig, and A. Seffah, "Patterns in Model-Based Development", in Proceedings of the INTERACT'03 - Workshop entitled: Software and Usability Cross-Pollination: The Role of Usability Patterns, 2003.
173. D. Sinnig, A. Gaffar, A. Seffah, and P. Forbrig, "Patterns, Tools and Models for Interaction Design", in Proceedings of the CADUI'04 - Workshop entitled Making model-based user interface design practical: Usable and open methods and tools, 2004.
174. E. G. Sirer and B. N. Bershad, "Using Production Grammars in Software Testing", in Proceedings of the Second Conference on Domain Specific Languages, Austin, Texas, October 3-5,1999.

175. H.-W. Six and J. Voss, "User Interface Development: Problems and Experiences", in Proceedings of the Symposium on New Results and New Trends in Computer Science, S. L. N. i. C. Science(Eds.), Graz/Austria, 1991.
176. G. Smith, *The Object-Z Specification Language*, vol. 1,ed., Kluwer Academic Publishers, pp.160, isbn:0-7923-8684-1, 2000.
177. S. Software, "Seapine QA Wizard - Automated Functional and Regression Testing", (downloads.seapine.com/pub/product-info/qawizard.pdf), conferred in November, 2006.
178. N. Souchon and J. Vanderdonckt, "A Review of XML-Compliant User Interface Description Languages", in Proceedings of the 10th International Conference on Design, Specification, and Verification of Interactive Systems (DSV-IS'03), Madeira, 4-6 June,2003.
179. J. M. Spivey, *The Z Notation: A Reference Manual*,ed., Prentice Hall International (UK) Ltd, 1998.
180. E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson, "User Interface Reverse Engineering in Support of Interface Migration to the Web", *Automated Software Engineering*, vol. 10, pp. 271-301, 2003.
181. E. Stroulia, M. El-Ramly, L. Kong, P. Sorenson, and B. Matichuk, "Reverse Engineering Legacy Interfaces: An Interaction-Driven Approach", in Proceedings of the WCRE99, 1999.
182. P. Szekely, P. Luo, and R. Neches, "Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design", in Proceedings of the CHI'92, 1992.
183. P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, and E. Salcher, "Declarative interface models for user interface construction tools: the MASTERMIND approach", in Proceedings of the EHCI'95, 1995.
184. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*,ed., Addison-Wesley, pp.411, isbn:ISBN: 0201178885, 1999.
185. N. Tillmann and W. Schulte, "Parameterized Unit Tests", in Proceedings of the ESEC/FSE'05 - Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13), Lisbon - Portugal, Semtember 5-9,2005.
186. H. Traetteberg, "Model-based user interface design", PhD Thesis, Norwegian University of Science and Technology, Department of Computer and Information Sciences, 2002
187. S. Trewin, G. Zimmermann, and G. Vanderheiden, "Abstract User Interface Representations: How Well do they Support Universal Access?" in Proceedings of the CUU'03, Vancouver, British Columbia, Canada, November 10-11,2003.
188. J. D. Ullman and J. D. Widom, *A First Course in Database Systems*, 2nd ed., Prentice Hall, pp.528, isbn:0130353000, 2001.

189. M. Utting, "COMP424 Module 2: Specification-Based Testing", 2004.
190. J. Vanderdonckt, L. Bouillon, and N. Souchon, "Flexible Reverse Engineering of Web Pages with VAQUISTA", in Proceedings of the IEEE 8th Working Conf. on Reverse Engineering, 2001.
191. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, and N. Tillmann, "Online Testing with Model Programs", in Proceedings of the ESEC/FSE'05, 2005.
192. M. Veanes, C. Campbell, W. Schulte, and P. Kohli, "On-The-Fly Testing of Reactive Systems", Technical Report MSR-TR-2005-05, January, 2005.
193. M. v. Welie, G. C. v. d. Veer, and A. Eliens, "An Ontology for Task World Models", in Proceedings of the DSV-IS'98, 1998.
194. L. White and H. Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences", in Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00), San Jose, California, 2000.
195. C. Wiecha, W. Bennett, S. Boies, J. Gould, and S. Green, "ITS: A Tool for Rapidly Developing Interactive Applications", *ACM Transactions on Information Systems*, vol. 8(3), pp. 204-236, 1990.
196. C. Wiecha and S. Boies, "Generating user interfaces: principles and use of ITS style rules", in Proceedings of the Proceedings of the UIST'90, October, 1990.
197. J. M. Wing, "Formal Methods", in *Encyclopedia of Software Engineering*, J. J. Marciniak (Eds.), 1994, pp. 504-517.
198. K. Winter, "Model Checking Abstract State Machines", PhD, Elektrotechnik und Informatik der Technischen Universität Berlin, 2001
199. K. Zambelich, "Totally Data-Driven Automated Testing. Whitepaper", (http://www.sqa-test.com/White_Paper.doc), conferred in October, 2006.
200. M. V. Zelkowitz and D. R. Wallace, "Experimental models for validating technology", *IEEE Computer*, vol. 31(5), pp. 23-31, 1998.

Appendix A

A.1. Notepad specification

```
//-----  
// Notepad main window  
//-----  
namespace Notepad;  
using WindowManager;  
using FileManager;  
  
// State variables  
// ---- editing status ----  
string text = "",  
selText = "";  
int posCursor = 0;  
bool dirty = false;  
// ---- file being edited ----  
string fileOpened = "",  
directory="E:"; // "E:" for test purposes  
// ---- file and replace settings ----  
string findWhat = "",  
replaceWord = "",  
direction = "Down";  
bool matchCase = false;  
// ---- temporary state of the open feature ----  
bool svBfrOpen = false;  
// ---- temporary state of the close feature ----  
bool svBfrClose = false;  
  
// It is possible to launch the Notepad application  
[Action] void LaunchNotepad()  
{  
requires !IsOpen("Notepad"); {  
AddWindow("Notepad","",false);  
Init();  
}  
}  
void Init()  
{  
FileManager.CreateTextFile("E:\\foo.txt",""); //for test purposes  
text = "";  
posCursor = 0;  
selText = "";  
dirty = false;  
fileOpened = "";  
findWhat = "";  
svBfrOpen = false;  
svBfrClose = false;  
}  
// It is possible to close the application.  
[Action] void Close()  
{  
requires IsEnabled("Notepad") ; {  
if (dirty) {  
AddWindow("MsgSaveChanges","Notepad",true);  
svBfrClose = true;  
svBfrOpen = false;  
}  
else CloseApp();  
}  
}  
void CloseApp(){  
if (IsOpen("Replace")) ReplaceDialog.RemoveReplace();  
if (IsOpen("Find")) FindDialog.RemoveFind();  
Init();  
if (IsOpen("Notepad")) RemoveWindow("Notepad");  
}  
}
```

```

[Action] void MsgSvBfrClose(string option)
requires IsEnabled("MsgSaveChanges") && svBfrClose &&
    option in Set{"y","n","c"}; {
    RemoveWindow("MsgSaveChanges");
    switch (option){
        case "n" : CloseApp();break;
        case "c" : svBfrClose=false; break;
        case "y" : if (fileOpened != "") {
            SaveDlglistener.SaveFile(directory,fileOpened);
            CloseApp();
        }
        else
            AddWindow("Save","Notepad",true);
        return;
    }
    default : return;
}
}
// It is possible to type text
[Action] void InsText(string typedText)
requires IsEnabled("Notepad") &&
    text.Length + typedText.Length < 4; {
    text = text.Substring(0,posCursor-selText.Length) + typedText +
        text.Substring(posCursor,text.Length-posCursor);
    posCursor = posCursor - selText.Length + typedText.Length ;
    selText = "";
    dirty = true;
}
[Action(Kind=ActionAttributeKind.Probe)]
string GetText()
requires IsEnabled("Notepad"); {
    return text;
}
// helper method
Set<<int,int>> SelectText { get {
    if (text.Length==1 || text.Length==2)
        return Set{p0 in Set{0..text.Length-1},
            p1 in Set{p0+1..text.Length}; <p0,p1>};
    else return Set{<0,0>};
}}
[Action] void SelText(int p1,int p2)
requires IsEnabled("Notepad") && text!=" " &&
    p1>=0 && p1< text.Length &&
    p2>=p1 && p2<= text.Length; {
    selText = Substring(text,p1,p2-p1);
    posCursor = p2;
}
// It is possible to open a file
[Action] void MsgSvBfrOpen (string option)
requires IsEnabled("MsgSaveChanges") && svBfrOpen &&
    option in Set{"y","n","c"}; {
    RemoveWindow("MsgSaveChanges");
    switch (option){
        case "y": if (fileOpened != ""){
            SaveDialog.Show("Notepad",directory,fileOpened);
        }
        else
            SaveDlglistener.SaveFile(directory,fileOpened);
        break;
        case "n": OpenFileDialog.Show("Notepad",directory);
        break;
        case "c": break;
        default: return;
    }
}
[Action] void Open()
requires IsEnabled("Notepad"); {
    if (dirty) {
        AddWindow("MsgSaveChanges","Notepad",true);
        svBfrOpen = true;
        svBfrClose = false;
    }
    else {

```

```

        OpenFileDialog.ShowDialog("Notepad",directory);
    }
}
// It is possible to save the content in memory to a file
[Action] void Save()
    requires IsEnabled("Notepad") ;{
        if (fileOpened == "") {
            SaveDialog.ShowDialog("Notepad",directory,fileOpened);
        }
        else
            SaveDlglistener.SaveFile(directory,fileOpened);
    }
[Action] void SaveAs()
    requires IsEnabled("Notepad"); {
        SaveDialog.ShowDialog("Notepad",directory,fileOpened);
    }
}

// It is possible to open the find dialog.
[Action] void Find()
requires text!=" " && !IsOpen("Replace") && IsEnabled("Notepad") ;{
    if (!IsOpen("Find")) {
        FindDialog.ShowDialog("Notepad",findWhat);
    }
}
[Action] void FindNext()
requires text!=" " && IsEnabled("Notepad") ;{
    if (findWhat == "" && !IsOpen("Find") && !IsOpen("Replace")) {
        FindDialog.ShowDialog("Notepad","");
    }
    else if (findWhat!="")
        FindNextWord(findWhat, matchCase, direction);
}
[Action] void MsgAckCantFindWord()
    requires IsEnabled("MsgAckCantFindWord") &&
        windows["MsgAckCantFindWord"].parent == "Notepad"; {
        RemoveWindow("MsgAckCantFindWord");
        SetFocus("Notepad");
    }
}

// It is possible to open the replace dialog.
[Action] void Replace()
requires !IsOpen("Find") && IsEnabled("Notepad") ;{
    if (!IsOpen("Replace"))
        ReplaceDialog.ShowDialog("Notepad",findWhat,replaceWord);
}
}

// Interfaces
// ---- Open dialog interface ----
var CNotepadOpDlg OpDlglistener = new CNotepadOpDlg();

class CNotepadOpDlg : OpenFileDialog.IOpenDlgListener {
    void OpenFile(string dir, string file){
        string path = dir + "\\\" + file;
        text = FileManager.ReadFile(path);
        dirty = false;
        posCursor = 0;
        selText = "";
        directory = dir;
        fileOpened = file;
        svBfrOpen = false;
    }
    CNotepadOpDlg(){
        OpenFileDialog.SetOpenDialogListener(this);
    }
}

// ---- Save dialog interface ----
var CNotepadSaDlg SaveDlglistener = new CNotepadSaDlg();

class CNotepadSaDlg : SaveDialog.ISaveDlgListener {
    string dir=null,file=null;

    void SaveFile(string dir, string file){
        string path = dir + "\\\"+file;

```

```

        if (file != "*.txt" && file != "") {
            CreateTextFile(path,text);
            Notepad.fileOpened = file;
            directory = dir;
            dirty = false;
            if (svBfrOpen) {
                AddWindow("Open", "Notepad", true);
                svBfrOpen = false;
            }
            else if (svBfrClose)
                CloseApp();
        }
    }
    CNotepadSaDlg(){
        SaveDialog.setSaveDialogListener(this);
    }
}

// ---- Find dialog interface ----
var CNotepadFiDlg FiDlglistener = new CNotepadFiDlg();

class CNotepadFiDlg : FindDialog.IFindDlgListener {
    void FindNext(string findWord, bool matchC, string dir)
        requires dir in Set{"Up", "Down"}; {
        int index = -1;
        direction = dir;
        matchCase = matchC;
        Notepad.findWhat = findWord;
        index = FindWord();
        if (index in Set{0..text.Length-findWord.Length} && dir=="Up"){
            selText = text.Substring(index,findWord.Length);
            posCursor = index + findWord.Length;
        }
        else if (index != -1 && dir == "Down") {
            selText = text.Substring(index+posCursor,findWord.Length);
            posCursor = posCursor + index + findWord.Length;
        }
        else
            AddWindow("MsgAckCantFindWord", "Find", true);
        Notepad.findWhat = findWord;
    }
    CNotepadFiDlg(){
        FindDialog.setFindDialogListener(this);
    }
}

// ---- Replace dialog interface ----
var CNotepadReDlg ReDlglistener = new CNotepadReDlg();

class CNotepadReDlg : ReplaceDialog.IReplaceDlgListener {
    public void FindNext(string findWord, string repWord, bool matchC){
        int index = -1;
        Notepad.findWhat=findWord;
        replaceWord=repWord;
        direction="Down";
        matchCase=matchC;
        index = FindWord();
        if (index != -1) {
            posCursor = index + posCursor + findWord.Length;
            selText = findWord;
        }
        else
            AddWindow("MsgAckCantFindWord", "Replace", true);
    }
    public void Replace(string findWord, string repWord, bool matchC)
    {
        Notepad.findWhat = findWord;
        replaceWord = repWord;
        matchCase = matchC;
        direction="Down";
        if ((matchC && selText == findWord) ||
            (!matchC && selText.ToLower() == findWord.ToLower())) {
            text = text.Substring(0, posCursor-findWord.Length) + repWord

```



```

        + text.Substring(posCursor, text.Length - posCursor);
        posCursor = posCursor - findWord.Length + repWord.Length;
        selText = repWord;
    }
    FindNext(findWord,repWord, matchCase);
}
public void ReplaceAll(string findWord, string repWord,
    bool matchCase) {
    int i;
    findWhat = findWord; replaceWord = repWord;
    posCursor = 0; selText = "";
    if (matchCase) text = text.Replace(findWord, repWord);
    else{
        i = text.ToLower().IndexOf(findWord.ToLower());
        if (i>=0 && i<text.Length)
            text = FindRep(text,i,findWord, repWord);
    }
}
string FindRep(string txt,int i, string findWord,string repWord){
    if (i<0 || i>txt.Length) return txt;
    else return txt.Substring(0,i)+ repWord+
        FindRep(txt.Substring(i+findWord.Length,
            txt.Length-i-findWord.Length),
            txt.Substring(i+findWord.Length, txt.Length-i-
                findWord.Length).ToLower().IndexOf(findWord.ToLower()),
            findWord,repWord);
}
CNotepadReDlg(){
    ReplaceDialog.setReplaceDialogListener(this);
}
}

// helper methods
int FindWord(){
    string txt = text;
    string findStr = findWhat;
    int index = -1;

    if (!matchCase) {
        txt = text.ToLower();
        findStr = findWhat.ToLower();
    }
    if (direction == "Down")
        index=txt.Substring(posCursor,
            txt.Length-posCursor).IndexOf(findStr);
    else {
        int i=posCursor-selText.Length+findStr.Length-1;
        if (i<0 || i>text.Length) i=posCursor;
        index = txt.Substring(0,i).LastIndexOf(findStr);
        if (index>-1 && index<posCursor - selText.Length)
            return index;
        else index = -1;
    }
    return index;
}

void FindNextWord(string findWord, bool matchC, string dir)
requires dir in Set{"Up","Down"}; {
    int index = -1;
    direction = dir;
    matchCase = matchC;
    index = FindWord();
    if (index != -1 && dir=="Up"){
        selText = text.Substring(index,findWord.Length);
        posCursor = index + findWord.Length;
    }
    else if (index != -1 && dir == "Down"){
        selText = text.Substring(index+posCursor,findWord.Length);
        posCursor = posCursor + index + findWord.Length;
    }
    else {
        AddWindow("MsgAckCantFindWord", "Notepad", true);
    }
}

```

```

}

//-----
// Open dialog
//-----
namespace OpenFileDialog;
using WindowManager;
using FileManager;

var string fileNameO = "*.txt";
var string! dirO = "E:";

var IOpenDlgListener OpenDlgListener;

public interface IOpenDlgListener{
    void OpenFile(string dirO, string file);
    void Cancel();
}
public void SetOpenDialogListener(IOpenDlgListener listener) {
    OpenDlgListener = listener;
}
public void Show(string parent, string d)
requires !IsOpen("Open");{
    dirO = d;
    AddWindow("Open",parent,true);
}
[Action] void Cancel()
requires IsEnabled("Open");{
    fileNameO = "*.txt";
    RemoveWindow("Open");
    OpenDlgListener.Cancel();
}
[Action] void MsgAckFileNotFound()
requires IsEnabled("MsgAckFileNotFound") ; {
    RemoveWindow("MsgAckFileNotFound");
}
[Action] void Open()
requires IsEnabled("Open");{
    if (FileManager.FileExists(dirO+"\\\\"+fileNameO)) {
        OpenDlgListener.OpenFile(dirO,fileNameO);
        fileNameO = "*.txt";
        RemoveWindow("Open");
    }
    else
        AddWindow("MsgAckFileNotFound","Open",true);
}
[Action] void SetFileName(string fileName)
requires IsEnabled("Open") ;{
    fileNameO = fileName;
}

//-----
// Save dialog
//-----
namespace SaveDialog;
using WindowManager;
using FileManager;

var string fileNameS = "*.txt";
var string! dirS = "E:";

var ISaveDlgListener SaveDlgListener;

public interface ISaveDlgListener{
    void SaveFile(string dir, string file);
    void Cancel();
}
public void setSaveDialogListener(ISaveDlgListener listener) {
    SaveDlgListener = listener;
}

```

```

public void Show(string parent, string dir, string file)
    requires !IsOpen("Save"); {
        dirS = dir; fileNameS = file;
        AddWindow("Save",parent,true);
    }
[Action] void Cancel()
    requires IsEnabled("Save") ; {
        fileNameS = "";
        SaveDlgListener.Cancel();
        RemoveWindow("Save");
    }
[Action] void Save()
    requires IsEnabled("Save") ; {
        if (FileManager.FileExists(dirS + "\\\" + fileNameS)) {
            AddWindow("MsgOverwriteFile", "Save", true);
        }
        else{
            if (IsValid(fileNameS)) {
                fileNameS = "";
                RemoveWindow("Save");
                SaveDlgListener.SaveFile(dirS,fileNameS);
            }
        }
    }
[Action] void MsgOverwriteFile(string option)
    requires IsEnabled("MsgOverwriteFile");{
        RemoveWindow("MsgOverwriteFile");
        switch (option){
            case "n" : return;
            case "y" : RemoveWindow("Save");
                       SaveDlgListener.SaveFile(dirS,fileNameS);
                       fileNameS = "";
                       return;
            default : return;
        }
    }
[Action] void SetFileName(string fName)
    requires IsEnabled("Save"); {
        fileNameS = fName;
    }

//-----
// Find dialog
//-----
namespace FindDialog;
using WindowManager;

// state variables
string findWhatF= "";
string directionF = "Down" ;
bool matchCaseF = false;

var IFindDlgListener FindDlgListener;

public interface IFindDlgListener{
    void FindCancel();
    void FindNext(string findWord, bool matchC, string dir) ;
}
public void setFindDialogListener(IFindDlgListener listener) {
    FindDlgListener = listener;
}
// helper methods
public void Show(string parent, string findWord)
    requires !IsOpen("Find");{
        findWhatF = findWord;
        directionF = "Down" ;
        AddWindow("Find",parent,false);
    }
public void RemoveFind()
    requires IsOpen("Find");
    {
        findWhatF= ""; directionF = "Down" ;
        matchCaseF = false;
    }

```

```

    RemoveWindow("Find");
}
// Actions
[Action(Kind=ActionAttributeKind.Scenario)]
void ScnFind(string fw, string dir, bool mc)
requires IsEnabled("Find") &&
    dir in Set{"Up","Down"} ; {
    SetFindWhat(fw);
    SetDirection(dir);
    SetMatchCase(mc);
}
[Action] void SetFindWhat(string str)
requires IsEnabled("Find");{
    findWhatF = str;
}
[Action] void SetDirection(string dir)
requires dir in Set{"Up","Down"} && IsEnabled("Find");{
    directionF = dir;
}
[Action] void SetMatchCase(bool op)
requires IsEnabled("Find");{
    matchCaseF = op;
}
[Action] void FindNext()
requires IsEnabled("Find") && findWhatF!="";{
    FindDlgListener.FindNext(findWhatF, matchCaseF, directionF);
}
[Action] void Cancel()
requires HasFocus("Find") ;{
    findWhatF= ""; directionF = "Down" ;
    matchCaseF = false;
    RemoveWindow();
}
[Action] void MsgAckCantFindWord()
requires IsEnabled("MsgAckCantFindWord") &&
    windows["MsgAckCantFindWord"].parent == "Find"; {
    RemoveWindow("MsgAckCantFindWord");
}

//-----
// Replace dialog
//-----
namespace ReplaceDialog;
using WindowManager;

// state variables
string findWhatR="",
    replaceWithR="";
bool matchCaseR=false;

var IReplaceDlgListener ReplaceDlgListener;

// interface
public interface IReplaceDlgListener{
    public void FindNext(string findWord, string replaceWord,
        bool matchCase) ;
    public void Replace(string findWord, string replaceWord,
        bool matchCase);
    public void ReplaceAll(string findWord, string replaceWord,
        bool matchCase);
}
// helper methods
public void setReplaceDialogListener(IReplaceDlgListener listener){
    ReplaceDlgListener = listener;
}
public void Show(string parent, string findWord,string replaceWord)
requires !IsOpen("Replace"); {
    findWhatR = findWord; replaceWithR = replaceWord;
    matchCaseR = false;
    AddWindow("Replace",parent,false);
}

```

```

public void RemoveReplace()
requires IsOpen("Replace"); {
    findWhatR = ""; replaceWithR="";
    RemoveWindow("Replace");
}
// Actions
[Action(Kind=ActionAttributeKind.Scenario)]
void ScnReplace(string fw, string rw, bool mc)
requires IsEnabled("Replace");{
    SetFindWhat(fw);
    SetReplaceWith(rw);
    SetMatchCase(mc);
}
[Action] void Cancel()
requires IsEnabled("Replace"); {
    findWhatR = ""; replaceWithR="";
    RemoveWindow("Replace");
}
[Action] void SetFindWhat(string str)
requires IsEnabled("Replace") ; {
    findWhatR = str;
}
[Action] void SetReplaceWith(string str)
requires IsEnabled("Replace") && findWhatR != "" ; {
    replaceWithR = str;
}
[Action] void SetMatchCase(bool value)
requires IsEnabled("Replace"); {
    matchCaseR = value;
}
[Action] void FindNext()
requires IsEnabled("Replace") && findWhatR != "" ; {
    ReplaceDlgListener.FindNext(findWhatR,replaceWithR, matchCaseR);
}
[Action] void Replace()
requires IsEnabled("Replace") && findWhatR != "" ; {
    ReplaceDlgListener.Replace(findWhatR, replaceWithR, matchCaseR);
}
[Action] void ReplaceAll()
requires IsEnabled("Replace") && findWhatR != ""
&& findWhatR.Length>=replaceWithR.Length // for testing purposes
; {
    ReplaceDlgListener.ReplaceAll(findWhatR,replaceWithR,matchCaseR);
}
[Action] void MsgAckCantFindWord()
requires IsEnabled("MsgAckCantFindWord") &&
windows["MsgAckCantFindWord"].parent=="Replace"; {
    RemoveWindow("MsgAckCantFindWord");
}

// -----
//      Notepad views
// -----
// Open scenario.
[Action(Kind=ActionAttributeKind.Scenario)]
void ScnOpen(string fileToOpen, string saveChanges,
             string fileToSave, string overwrite)
requires IsEnabled("Notepad");
{
    Open();
    if (IsEnabled("SaveChanges")) // if dirty
    {
        MsgSvBfrOpen(saveChanges);
        if (IsEnabled("Save")) // saveChanges == true
        {
            SaveDialog.SetFileName(fileToSave);
            SaveDialog.Save();
            if (IsEnabled("MsgReplaceFile")) // file exists
            {
                SaveDialog.MsgOverwriteFile(overwrite); //yes or no
                if (IsEnabled("Save")) // overwrite = no, so get
                    // out of the cycle
                SaveDialog.Cancel(); // end of the scenario
            }
        }
    }
}

```

```

    }
  }
  }
  // (saveChanges != c || !dirty
  if (IsEnabled("Open")) {
    OpenFileDialog.SetFileName(fileToOpen);
    OpenFileDialog.Open();
    if (IsEnabled("AckMsgFileNotFound"))
    {
      OpenFileDialog.MsgAckFileNotFound();
      OpenFileDialog.Cancel(); // end of the scenario
    }
  }
}

// Navigation map view
// with focus property modelled
string NavigationMapWithFocus { get {
  if (GetWindowWithFocus()== "") return "NotOpen";
  else return GetWindowWithFocus();
}}
// without modelling focus property
Set<string> NavigationMapWithoutFocus { get {
  return GetEnabledWindows();
}}
// distinguish the Find dialog states with different enabled
// actions
string ValidationGroup { get {
  if (GetWindowWithFocus()=="Find") {
    if (FindDialog.findWhatF != "") return "Find Next enabled";
    else return "Find Next disabled";
  }
  else return "OutFind";
}}
//Open scenario view
string OpenScenario { get {
  if (!IsOpen("Notepad")) return "NotOpen";
  else if (IsEnabled("MsgSaveChanges") && svNfrOpen)
    return "MsgSaveChanges";
  else if (IsEnabled("Save") && svBfrOpen) return "Save";
  else if (IsEnabled("MsgAckFileNotFound"))
    return "MsgAckFileNotFound";
  else if (IsEnabled("MsgOverwriteFile") && svBfrOpen)
    return "MsgOverwriteFile";
  else if (IsEnabled("Open")) return "Open";
  else if (dirty) return "Dirty";
  else if (!dirty) return "NotDirty";
  else return "";
}}
// save scenario
string SaveScenario { get {
  if (!IsOpen("Notepad")) return "NotOpen";
  else if (IsEnabled("Save")) return "Save";
  else if (IsEnabled("MsgOverwriteFile") && IsOpen("Save"))
    return "MsgOverwriteFile";
  else return "SaveDlgClosed";
}}
// find scenario
string FindScenario { get {
  if (!IsOpen("Notepad")) return "NotOpen";
  else if (!IsOpen("Find")) return "FindDlgClosed";
  else if (HasFocus("Find")) return "Find";
  else if (IsOpen("MsgAckCantFindWord"))
    return "MsgAckCantFindWord";
  else return "FindDlgNotActive";
}}
// the word to look for is at the beginning of the text
string AtBeginningGroup { get {
  if (text!=" " && findWhat.Length<=text.Length &&
    text.Substring(0,findWhat.Length)==findWhat
    && Notepad.findWhat!=" ")
    return "AtBeginning";
  else return "NotAtBeginning";}}

```

```

// the word to look for is at the end of the text
string AtEndGroup { get {
if (text!=" " && findWhat.Length<=text.Length &&
    text.Substring(text.Length-findWhat.Length,findWhat.Length)==
    findWhat && Notepad.findWhat!=" ") return "AtEnd";
else return "NotAtEnd";
}}
// the word to look for is equal to the text content
string WordEQToText { get {
    if (text==findWhat) return "wordEQText";
    else return "NotEQ";
}}
// the cursor position is in the middle of the word to look for
string AtTheMiddleGroup { get {
if (Exists{ i in Set{0..text.Length};
    posCursor>i && posCursor<i+findWhat.Length &&
    i==text.IndexOf(findWhat)})
    return "InTheMiddle";
else return "NotInTheMiddle";
}}
// the several occurrences of the word overlap each other
string OverlapGroup { get {
if (findWhat.Length>1 && (Exists{i in Set{1..findWhat.Length-1};
    findWhat.Substring(0,i)==findWhat.Substring(findWhat.Length-i,i)
    &&
    text.IndexOf(findWhat+findWhat.Substring(0,i))>=0)
    || (Exists{i in Set{1..findWhat.Length-1};
    findWhat.Substring(0,i).ToLower() ==
    findWhat.Substring(findWhat.Length-i,i).ToLower() &&
    text.ToLower().IndexOf(findWhat.ToLower()+
        findWhat.Substring(0,i).ToLower())>=0)
    && !matchCase))
    return "Overlap";
else return "NotOverlap";
}}
// the several occurrences of the word are side by side
string SideBySideGroup { get {
if (text!=" " && findWhat!=" " && (text.IndexOf(findWhat+findWhat)>0
    ||
    text.ToLower().IndexOf(findWhat.ToLower()+findWhat.ToLower())>0
    &&
    !matchCase))
    return "SideBySide";
else
    return "NotSideBySide";
}}
// open view
// without modelling the focus property
<string,string> OpenFileDialog { get {
    if (IsOpen("Open")) return fileNameO="+fileNameO,"dirO="+dirO;
    else return <"NotOpen","NotOpen">;
}}
// open view
// modelling the focus property
string OpenFileDialog { get {
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (IsOpen("Open")) return openCtrlWthFocus;
    else return "OpenDlgClosed";
}}
// save view
// without modelling the focus property
<string,string> SaveDialogGroup { get {
    if (IsOpen("Save")) return <"fileNameS="+fileNameS,"dirS="+dirS>;
    else return <"NotOpen","NotOpen">;
}}
// find dialog view
// without modelling the focus property
<string,string,string> FindDialogGroup { get {
    if (IsOpen("Find") && matchCaseF==true)
        return
            <"findWhatF="+findWhatF,"directionF="+directionF,
            "matchCaseF=true">;
    else

```

```

        if (IsOpen("Find") && matchCaseF==false) return
            <"findWhatF="+findWhatF,"directionF="+directionF,
                "matchCaseF=false">;
        else return <"NotOpen","NotOpen","NotOpen">;
    }}
// find dialog view
// modelling the focus property
string FindDialogGroup { get {
    if (!IsOpen("Notepad")) return "NotOpen";
    else if (HasFocus("Find")) return findCtrlWthFocus;
    else if (IsOpen("Find")) return "FindDlgNotActive";
    else return "FindDlgClosed";
}}
// replace dialog view
// without modelling the focus property
<string,string,string> ReplaceDialogGroup { get {
    if (IsOpen("Replace"))
        return <"findWhatR="+findWhatR,"replaceWithR="+replaceWithR,
            "matchCaseR="+matchCaseR>;
    else return <"NotOpen","NotOpen","NotOpen">;
}}
// replace dialog view
// modelling the focus property
string ReplaceDialogGroup { get {
    if (HasFocus("MsgAckCantFindWord") &&
        windows["MsgAckCantFindWord"].parent != "Replace")
        return "MsgAckcantFindWord";
    else if (HasFocus("Replace")) return "");//replaceObjActive;
    else return "NotOpen";
}}

```


A.2. Address Book specification

```
//-----  
// Address Book main window  
//-----  
using System.String;  
using WindowManager;  
using FileManager;  
namespace AddressBook;  
  
// ---- types and state variables ----  
type Contact = <string,string,string,string,string,string>;  
  
var Contact          contactInMem = <"", "", "", "", "", "">;  
var Seq<Contact>    dbContacts    = Seq{};  
var string sort     = "Asc";  
    orderedBy      = "Last Name";  
    fileOpened     = "";  
    directory      = "E: "; // "E:" for test purposes  
    nextAction     = "";  
var int lineSelected = -1;  
var bool addNew     = true,  
    dirty          = false;  
var bool returnToOpenDlg = false,  
    returnToAddressBook = false;  
  
// Actions  
// To launch the AddressBook application  
[Action] void LaunchAddressBook()  
requires !IsOpen("AddressBook"); {  
    AddWindow("AddressBook", "", false);  
    lineSelected = -1;  
    // for testing purposes  
    FileManager.CreateDataBaseFile("E:\\AB.adr",  
                                   Seq{<"Pinto", "Nuno", "1", "4", "", "">,  
                                       <"Silva", "Ana", "3", "2", "", "">});  
}  
  
// To close the application.  
[Action] void Close()  
requires IsEnabled("AddressBook"); {  
    if (dirty) AddWindow("MsgSvBfrClose", "AddressBook", true);  
    else CloseApp();  
}  
void CloseApp(){  
    dbContacts = Seq{};  
    fileOpened = "";  
    directory="E: "; // "E:" for test purposes  
    nextAction = "";  
    lineSelected = -1;  
    addNew = true;  
    dirty = false;  
    returnToOpenDlg = false;  
    contactInMem = <"", "", "", "", "", "">;  
    if (IsOpen("Find")) FindDialog.Cancel();  
    RemoveWindow("AddressBook");  
}  
  
[Action] void MsgSvBfrClose(string option)  
requires option in Set{"y", "n", "c"} &&  
    IsEnabled("MsgSvBfrClose"); {  
    RemoveWindow("MsgSvBfrClose");  
    switch (option){  
        case "y" : if (fileOpened == "") {  
            AddWindow("Save", "AddressBook", true);  
        }  
        else {  
            SaveDlgListener.SaveFile(directory, fileOpened);  
        }  
    }  
}
```

```

        CloseApp();
    }
    case "n" : CloseApp(); return;
    case "c" : return;
    default : return;
}
}

// To add a new contact
[Action] void NewContact()
requires IsEnabled("AddressBook");{
    addNew = true;
    ContactDialog.Show("AddressBook", <"", "", "", "", "", "", ">");
}

// To edit an existing contact
Set<int> selLine{ get {
    if (dbContacts.Size>0)
        return Set{0..dbContacts.Size-1};
    else return Set{-1};
}}

[Action] void SelContact(int line)
requires IsEnabled("AddressBook") && dbContacts.Size>0 &&
    line>=0 && line<dbContacts.Size; {
    lineSelected = line;
}

[Action] void EditContact()
requires IsEnabled("AddressBook") && dbContacts.Size>0 &&
    lineSelected > -1; {
    addNew = false;
    ContactDialog.Show("AddressBook", dbContacts[lineSelected]);
}

[Action] void Copy()
requires IsEnabled("AddressBook") && dbContacts.Size>0 &&
    lineSelected != -1;
ensures contactInMem == dbContacts[lineSelected]; {
    contactInMem = dbContacts[lineSelected];
}

[Action] void Paste()
requires IsEnabled("AddressBook") &&
    contactInMem != <"", "", "", "", "", ">; {
    dbContacts = dbContacts + Seq{contactInMem};
    dirty = true;
}

[Action] void Delete()
requires IsEnabled("AddressBook") && dbContacts.Size>0 &&
    lineSelected != -1; {
    dbContacts = dbContacts.Subseq(0, lineSelected) +
        dbContacts.Subseq(lineSelected+1, dbContacts.Size);
    dirty = true;
    lineSelected = -1;
}

[Action] void Sort(string field)
requires IsEnabled("AddressBook") && dbContacts.Length>0 &&
    field in Set{"Last Name", "First Name", "Business Phone",
        "Home Phone", "Email", "Fax"}; {
    if (field == orderedBy) {
        if (sort == "Asc") sort = "Des";
        else sort = "Asc";
        SortContacts(field, sort);
    }
    else {
        sort = "Asc";
        SortContacts(field, sort);
    }
    orderedBy = field;
}

void SortContacts(string f, string s)
requires field in Set{"Last Name", "First Name", "Business Phone",
    "Home Phone", "Email", "Fax"} &&
    s in Set{"Up", "Down"}; {
    bool permutation = true;
    Contact x = <"", "", "", "", "", ">;
    while (permutation){

```

```

permutation = false;
for (int i=0,j=1; i<=dbContacts.Length-2 &&
    j<=dbContacts.Length-1;i++,j++)
    if (s=="Asc"){
        if (System.String.Compare(GetField(dbContacts[i],f),
            GetField(dbContacts[j],f))>0){
            x = dbContacts[i];
            dbContacts[i] = dbContacts[j];
            dbContacts[j] = x;
            permutation = true;
        }
    }
    else
        if (System.String.Compare(GetField(dbContacts[i],f),
            GetField(dbContacts[j],f))<0){
            x = dbContacts[i];
            dbContacts[i] = dbContacts[j];
            dbContacts[j] = x;
            permutation = true;
        }
    }
}
// To open a database file
[Action] void MsgSvBfrNew(string option)
requires IsEnabled("MsgSvBfrNew") &&
    option in Set{"y","n","c"}; {
    RemoveWindow("MsgSvBfrNew");
    switch (option){
        case "y": if (fileOpened!="") {
            SaveDlgListener.SaveFile(directory,fileOpened);
            fileOpened = "";
            lineSelected = -1;
            dbContacts = Seq{};
        }
        else {
            returnToAddressBook = true;
            SaveDialog.Show("AddressBook",directory,fileOpened);
        }
        return;
        case "n": fileOpened = "";
            lineSelected = -1;
            dbContacts = Seq{};
            return;
        case "c": return;
        default: return;
    }
}

[Action] void NewAddressBook()
requires IsEnabled("AddressBook"); {
    if (dirty) AddWindow("MsgSvBfrNew","AddressBook",true);
    else {
        fileOpened = "";
        lineSelected = -1;
        dbContacts = Seq{};
    }
}

[Action] void MsgSvBfrOpen (string option)
requires IsEnabled("MsgSvBfrOpen") &&
    option in Set{"y","n","c"}; {
    RemoveWindow("MsgSvBfrOpen");
    switch (option){
        case "y": if (fileOpened!="") {
            SaveDlgListener.SaveFile(directory,fileOpened);
            OpenFileDialog.Show("AddressBook", directory);
        }
        else {
            returnToOpenDlg = true;
            SaveDialog.Show("AddressBook",directory,fileOpened);
        }
        return;
        case "n": OpenFileDialog.Show("AddressBook",directory);
            return;
    }
}

```

```

        case "c": return;
        default: return;
    }
}
[Action] void OpenAddressBook()
requires IsEnabled("AddressBook"); {
    if (dirty) AddWindow("MsgSvBfrOpen", "AddressBook", true);
    else OpenFileDialog.Show("AddressBook", directory);
}
[Action] void SaveAddressBookAs()
requires IsEnabled("AddressBook") && dirty; {
    SaveDialog.Show("AddressBook", directory, fileOpened);
    AddWindow("Save", "AddressBook", true);
}
[Action] void SaveAddressBook()
requires IsEnabled("AddressBook") && fileOpened != "" &&
    dirty == true; {
    SaveDialog.Show("AddressBook", directory, fileOpened);
    AddWindow("Save", "AddressBook", true);
}
// To open the find dialog.
[Action] void Find()
requires IsEnabled("AddressBook"); {
    FindDialog.Show("AddressBook");
}
[Action] void FindNext()
requires IsEnabled("AddressBook"); {
    FindDialog.Show("AddressBook");
}
[Action(Kind=ActionAttributeKind.Probe)]
string GetDBLastName()
requires IsEnabled("AddressBook"); {
    if (dbContacts.Size > 0)
        return GetField(dbContacts[0], "Last Name");
    else return "0";
}
[Action(Kind=ActionAttributeKind.Probe)]
string GetDBBusinessPhone()
requires IsEnabled("AddressBook"); {
    if (dbContacts.Size > 0)
        return GetField(dbContacts[0], "Business Phone");
    else return "0";
}
}

// Interfaces
// ---- Find dialog interface ----
var CAddressBookFiDlg FiDlgListener = new CAddressBookFiDlg();

class CAddressBookFiDlg : FindDialog.IFindDlgListener {
    void FindNext(string fw, string f, string d, bool mc, bool mww) {
        int lineSelOld = lineSelected;
        if (0 < lineSelected && lineSelected < dbContacts.Size - 1)
            if (d == "Up") lineSelected =
                FindWord(dbContacts.Subseq(0, lineSelected), fw, f, d, mc, mww);
            else lineSelected =
                FindWord(dbContacts.Subseq(lineSelected + 1,
                    dbContacts.Size), fw, f, d, mc, mww);
            if (lineSelOld == lineSelected)
                AddWindow("MsgAckCannotFindWord", "Find", true);
        }
    CAddressBookFiDlg() {
        FindDialog.setFindDialogListener(this);
    }
}

// ---- helper methods ----
int FindWord(Seq<Contact> dbC, string w, string f, string d, bool
mc, bool mww)
requires d in Set{"Up", "Down"}; {
    int i = 0;
    if (d == "Up") {
        for (i = dbC.Size - 1; i >= 0; i--) {
            if (CompareStrings(mc, mww, GetField(dbContacts[i], f), w, i) != -1)

```

```

        return i;
    }
    return lineSelected;
}
else {
    for (i=0; i< dbC.Size; i++) {
        if (CompareStrings(mc,mww,GetField(dbContacts[i],f),w,i)!=-1)
            return i+lineSelected;
    }
    return lineSelected;
}
}
}
public int CompareStrings(bool mc,bool mww,
                        string cf,string w,int i) {
    if (mc && mww && System.String.CompareOrdinal(cf,w)==0)
        return i;
    else if (mc && !mww && cf.IndexOf(w)!=-1 &&
            System.String.CompareOrdinal(cf.Substring(
                cf.IndexOf(w),w.Length),w)==0)
        return i;
    else if (!mc && mww &&
            System.String.CompareOrdinal(cf.ToLower(),
                w.ToLower())==0)
        return i;
    else if (!mc && !mww &&
            cf.ToLower().IndexOf(w.ToLower()) !=-1)
        return i;
    else return -1;
}

string GetField(Contact c, string f)
requires f in Set{"Last Name","First Name",
                 "Business Phone","Home Phone","Email","Fax"};
{
    switch (f){
        case "Last Name": return c.First;
        case "First Name": return c.Second;
        case "Business Phone": return c.Third;
        case "Home Phone": return c.Fourth;
        case "Email": return c.Fifth;
        case "Fax": return c.Sixth;
        default : return "";
    }
}

// ---- Open dialog interface ----
var CAddressBookOpDlg OpDlgListener = new CAddressBookOpDlg();

class CAddressBookOpDlg : OpenFileDialog.IOpenDlgListener
{
    void OpenFile(string dir, string file){
        int i = 0;
        string path = dir + "\\\" + file;
        if (FileManager.DataBaseExists(path)) {
            dbContacts = FileManager.ReadDataBase(path);
            dirty = false;
            directory = dir;
            fileOpened = file;
        }
        else
        {
            OpenFileDialog.Cancel();
            AddWindow("MsgAckFileNotFound","AddressBook",true);
        }
    }
    CAddressBookOpDlg(){
        OpenFileDialog.SetOpenDialogListener(this);
    }
}

// ---- Save dialog interface ----
var CAddressBookSaDlg SaveDlgListener = new CAddressBookSaDlg();

class CAddressBookSaDlg : SaveDialog.ISaveDlgListener

```

```

{
    string SaveFile(string dir, string file){
        string path = directory + "\\\" + file;
        string content = "";
        if (file != "*.adr" && file != "") {
            FileManager.CreateDataBaseFile(path,dbContacts);
            AddressBook.fileOpened = file;
            AddressBook.dirty = false;
        }
        if (returnToOpenDlg) {
            OpenFileDialog.Show("AddressBook",directory);
            returnToOpenDlg = false;
        }
        if (returnToAddressBook) {
            fileOpened = "";
            dbContacts = Seq{};
            lineSelected = -1;
            returnToAddressBook = false;
        }
        return "Ok";
    }
}
void Cancel(){
    returnToOpenDlg = false;
    returnToAddressBook = false;
}
CAddressBookSaDlg(){
    SaveDialog.setSaveDialogListener(this);
}
}
// ---- Contact dialog interface ----
var CContactDlg ContactDlgListener = new CContactDlg();

class CContactDlg : ContactDialog.IContactDlgListener {
    public void ContactUpdate(Contact contc) {
        if (addNew)// lineSelected == -1 // add a new contact
            dbContacts = dbContacts + Seq{contc};
        else // update an existing contact
            dbContacts = dbContacts.Subseq(0,lineSelected) +
                Seq{contc} +
                dbContacts.Subseq(lineSelected+1,dbContacts.Size);
        dirty = true;
        addNew = false;
    }
    CContactDlg(){
        ContactDialog.setContactDialogListener(this);
    }
}

//-----
// Contact dialog
//-----
using WindowManager;

namespace ContactDialog;
// types and variables
type Contact = <string,string,string,string,string,string>;

var Contact contc = <"", "", "", "", "", "">;
var IContactDlgListener ContactDlgListener;

public interface IContactDlgListener{
    public void ContactUpdate(Contact contc) ;
}
public void setContactDialogListener(IContactDlgListener listener){
    ContactDlgListener = listener;
}
public void Show(string parent, Contact c)
    requires !IsOpen("Contact"); {
    contc = <c.First,c.Second,c.Third,c.Fourth,c.Fifth,c.Sixth>;
    AddWindow("Contact",parent,true);
}
[Action] void Cancel()

```

```

requires IsEnabled("Contact"); {
    contc = <"", "", "", "", "", "">;
    RemoveWindow("Contact");
}
[Action] void Ok()
requires IsEnabled("Contact"); {
    ContactDlgListener.ContactUpdate(contc) ;
    contc = <"", "", "", "", "", "">;
    RemoveWindow("Contact");
}
[Action] void SetLastName(string str)
requires IsEnabled("Contact"); {
    contc = <str, contc.Second, contc.Third,
        contc.Fourth, contc.Fifth, contc.Sixth>;
}
[Action] void SetFirstName(string str)
requires IsEnabled("Contact"); {
    contc = <contc.First, str, contc.Third,
        contc.Fourth, contc.Fifth, contc.Sixth>;
}
[Action] void SetBusinessPhone(string str)
requires IsEnabled("Contact"); {
    contc = <contc.First, contc.Second, str,
        contc.Fourth, contc.Fifth, contc.Sixth>;
}
[Action] void SetHomePhone(string str)
requires IsEnabled("Contact"); {
    contc = <contc.First, contc.Second, contc.Third,
        str, contc.Fifth, contc.Sixth>;
}
[Action] void SetEmail(string str)
requires IsEnabled("Contact"); {
    contc = <contc.First, contc.Second, contc.Third,
        contc.Fourth, str, contc.Sixth>;
}
[Action] void SetFax(string str)
requires IsEnabled("Contact"); {
    contc = <contc.First, contc.Second, contc.Third,
        contc.Fourth, contc.Fifth, str>;
}
[Action(Kind=ActionAttributeKind.Scenario)]
void ScnEditContact(string LN, string FN, string BPh,
    string HPh, string E, string F)
requires IsEnabled("Contact"); {
    SetLastName(LN);
    SetFirstName(FN);
    SetBusinessPhone(BPh);
    SetHomePhone(HPh);
    // SetEmail(E); SetFax(F); // not tested
    Ok();
}

//-----
// Find dialog
//-----
using WindowManager;
namespace FindDialog;

var IFindDlgListener FindDlgListener;
string findWhat = "",
    field = "",
    direction = "Down";

bool matchCase = false,
    matchWholeWord = false;

public interface IFindDlgListener {
    void FindNext(string fw, string d, bool mc, bool mww)
        requires f in Set {"Last Name", "First Name", "Business Phone",
            "Home Phone", "Email", "Fax"}
            && d in Set {"Up", "Down"};
}

```

```

public void setFindDialogListener(IFindDlgListener listener) {
    FindDlgListener = listener;
}
public void Show(string parent)
requires !IsOpen("Find"); {
    // resets the values of the variables
    findWhat = "";
    field="Last Name";
    direction= "Down";
    matchCase=false;
    matchWholeWord=false;
    AddWindow("Find",parent,false);
}

[Action(Kind=ActionAttributeKind.Scenario)]
public void ScnFind (string fw, int f, string d, bool mc, bool mww)
requires IsEnabled("Find") && fw != "";{
    SetFindWhat(fw);
    if (f == 0) SetField("Last Name");
    else SetField("Business Phone");
    SetDirection(d);
    SetMatchCase(mc);
    SetMatchWholeWord(mww);
    Find();
}
[Action] public void SetFindWhat(string str)
requires IsEnabled("Find");{
    findWhat = str;
}
[Action] public void SetField(string str)
requires IsEnabled("Find") && str in Set{"Last Name","First Name",
    "Business Phone","Home Phone","Email","Fax"}; {
    field = str;
}
[Action] public void SetMatchCase(bool op)
requires IsEnabled("Find"); {
    matchCase = op;
}
[Action] public void SetMatchWholeWord(bool op)
requires IsEnabled("Find") && windows != Map{};{
    matchWholeWord = op;
}
[Action] public void SetDirection(string d)
requires IsEnabled("Find") && d in Set{"Up","Down"};{
    direction = d;
}
[Action] public void Find()
requires IsEnabled("Find") && findWhat!="";{
    FindDlgListener.FindNext(findWhat, field, direction,
        matchCase, matchWholeWord);
}
[Action] public void Cancel()
requires IsEnabled("Find");{
    // reset the value of the variables
    findWhat = "";
    field="Last Name";
    direction= "Down";
    matchCase=false;
    matchWholeWord=false;
    RemoveWindow("Find");
}
[Action] public void MsgAckCannotFindWord()
requires IsEnabled("MsgAckCannotFindWord") &&
    windows["MsgAckCannotFindWord"].parent == "Find"; {
    RemoveWindow("MsgAckCannotFindWord");
}

//-----
// Address Book views
//-----
// navigation map

```



```

Set<string> NavigationGroup { get {
    return GetEnabledWindows();
}}
// view to check the find scenario
string FindViewScn { get {
    if (IsEnabled("MsgAckCannotFindWord"))
        return "MsgAckCannotFindWord";
    else if (IsEnabled("Find")) return "Find";
    else if (!IsOpen("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
// find view
<string,string,string,string,string> FindDialogGroup { get {
    if (IsOpen("Find")) return <"findWhat="+ findWhat,
        "field="+ field, "direction="+ direction,
        "matchCase="+ matchCase,"matchWholeWord="+ matchWholeWord>;
    else return <"NotOpen", "NotOpen", "NotOpen", "NotOpen", "NotOpen">;
}}

// view to check the open scenario
string OpenViewScn { get {
    if (IsEnabled("MsgAckFileNotFound")) return "MsgAckFileNotFound";
    else if (IsEnabled("Open")) return "Open";
    else if (IsEnabled("Save")) return "Save";
    else if (IsEnabled("MsgSvBfrOpen")) return "MsgSvBfrOpen?";
    else if (IsEnabled("MsgOverwriteFile"))
        return "MsgOverwriteFile?";
    else if (!IsOpen("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
// view to check the save scenario
string SaveViewScn { get {
    if (IsEnabled("MsgOverwriteFile")) return "MsgOverwriteFile?";
    else if (IsEnabled("Save")) return "Save";
    else if (IsEnabled("MsgOverwriteFile"))
        return "MsgOverwriteFile?";
    else if (!IsEnabled("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
// dirty and a file opened
string DirtyFileView{ get {
    if (dirty && fileOpened != "") return "fileOpenedDirty";
    else if (!dirty && fileOpened != "") return "fileOpenedNotDirty";
    else if (fileOpened == "" && dirty) return "contentDirty";
    else if (fileOpened == "" && !dirty) return "contentNotDirty";
    else return "other";
}}
// Scenarios
// Find Scenario
[Action(Kind=ActionAttributeKind.Scenario)]
void FindScenario(string findW,string field,bool mc,bool mww,
    string dir)
    requires dir in Set{"Up","Down"} &&
        field in Set{"Last Name","First Name", "Business Phone",
            "Home Phone", "Email","Fax"} &&
            IsEnabled("AddressBook"); {
    Find();
    FindDialog.SetFindWhat(findW);
    FindDialog.SetField(field);
    FindDialog.SetMatchCase(mc);
    FindDialog.SetMatchWholeWord(mww);
    FindDialog.SetDirection(dir);
    FindDialog.Find();
    if (IsEnabled("MsgAckCannotFindWord"))
        FindDialog.MsgAckCannotFindWord();
    FindDialog.Cancel();
}
//Open Scenario
[Action(Kind=ActionAttributeKind.Scenario)]
void OpenScenarioScn(string fileToOpen, string saveChanges,
    string fileToSave, string overwrite)
    requires IsEnabled("AddressBook") &&
        saveChanges in Set{"y","n","c"} &&

```

```

        overwrite in Set{"y","n"}; {
OpenAddressBook();
if (IsEnabled("MsgSvBfrOpen")) // if dirty
{
    MsgSvBfrOpen(saveChanges);
    if (IsEnabled("Save")) // saveChanges == true
    {
        SaveDialog.SetFileName(fileToSave);
        SaveDialog.Save();
        if (IsEnabled("MsgOverwriteFile")) // file exists
        {
            SaveDialog.MsgOverwriteFile(overwrite); //yes or no
            if (IsEnabled("Save")) // overwrite = no, so get
                // out of the cycle
                SaveDialog.Cancel(); // end of the scenario
        }
    }
}
}
//(saveChanges != c || !dirty
if (IsEnabled("Open")) {
    OpenFileDialog.SetFileName(fileToOpen);
    OpenFileDialog.Open();
    if (IsEnabled("MsgAckFileNotFound"))
    {
        OpenFileDialog.MsgAckFileNotFound();
        OpenFileDialog.Cancel(); // end of the scenario
    }
}
}
}
//Save Scenario
[Action(Kind=ActionAttributeKind.Scenario)]
void SaveScenario(string fileName, string overwrite)
requires IsEnabled("Notepad") && overwrite in Set{"y","n"}; {
    SaveAddressBook();
    if (IsEnabled("Save")) //no file currently opened
    {
        SaveDialog.SetFileName(fileName);
        SaveDialog.Save();
        if (IsEnabled("MsgOverwriteFile"))
        {
            SaveDialog.MsgOverwriteFile(overwrite);
            if (IsEnabled("Save"))
                SaveDialog.Cancel();
        }
    }
}
}
// close scenario
string CloseViewScn { get {
    if (IsOpen("MsgOverwriteFile")) return "MsgOverwriteFile?";
    else if (IsEnabled("MsgSvBfrClose")) return "MsgSvBfrClose?";
    else if (IsEnabled("Save")) return "Save";
    else if (!IsEnabled("AddressBook")) return "NotOpen";
    else return "AddressBook";
}}
}

```

A.3. Window manager and file manager

```
//-----  
// Window manager  
//-----  
namespace WindowManager;  
  
string hasFocus = "";  
  
structure winInf{  
    string parent;  
    bool isModal;  
}  
Map<string,winInf> windows = Map{};  
  
bool IsOpen(string name) {  
    return Exists { i in windows; i == name};  
}  
bool IsEnabled(string name) {  
    int id;  
    if (IsOpen(name)) {  
        choose (i in windows, windows[i].isModal &&  
                i!=name && NotParent(i,name))  
        return false;  
    }  
    else return true;  
}  
else return false;  
}  
Set<string> GetEnabledWindows(){  
    return Set{x in windows, IsEnabled(x)};  
}  
bool NotParent(string p, string c) {  
    if (windows[c].parent == "") return true;  
    if (windows[c].parent == p) return false; //p is parent of c  
    else  
        return NotParent(p, windows[c].parent);  
}  
void AddWindow(string name, string parent, bool isModal) {  
    windows = windows + Map{name :> winInf(parent,isModal)};  
    hasFocus = name;  
}  
void RemoveWindow(string name)  
requires Exists { i in windows; i == name}; {  
    RemoveChild(name);  
    hasFocus = windows[name].parent;  
    windows[name] = none;  
}  
void RemoveChild(string name) {  
    foreach (x in windows, windows[x].parent == name)  
        RemoveChild(x);  
    windows = Map{i in windows, windows[i].parent != name;  
                  i:>winInf(windows[i].parent, windows[i].isModal)};  
}  
void SetFocus(string name)  
requires IsEnabled(name) || name == "" {  
    hasFocus = name;  
}  
string GetWindowWithFocus() {  
    return hasFocus;  
}  
bool HasFocus(string name) {  
    return name == hasFocus;  
}
```

```

//-----
// File manager
//-----
namespace FileManager;

Map<string,string> files = Map{};

public void CreateTextFile(string fileName, string fileContent) {
    files = files + Map{fileName :> fileContent};
}
public bool FileExists(string fileName) {
    choose (i in files, i == fileName) return true;
    else return false;
}
public string ReadFile(string fileName)
    requires FileExists(fileName); {
    return files[fileName];
}
public void RemoveFile(string fileName)
    requires FileExists(fileName); {
    files[fileName] = none;
}
bool IsValid(string file) {
    if (file == "") return false;
    if (file.IndexOfAny(new char[8]{'\\','*', '/', ':', '?', '\'', '<',
                                     '>', '|'})>=0)
        return false;
    else return true;
}

```