

2007

A model-based testing analysis methodology for software installation testing

Karen K. Kwok
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Kwok, Karen K., "A model-based testing analysis methodology for software installation testing" (2007). *Master's Theses*. 3559.
DOI: <https://doi.org/10.31979/etd.9yvc-e8be>
https://scholarworks.sjsu.edu/etd_theses/3559

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

A MODEL-BASED TESTING ANALYSIS
METHODOLOGY FOR
SOFTWARE INSTALLATION TESTING

A Thesis

Presented to

The Faculty of the Department of Aviation and Technology
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Karen K. Kwok

December 2007

UMI Number: 1452038

Copyright 2007 by
Kwok, Karen K.

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1452038

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

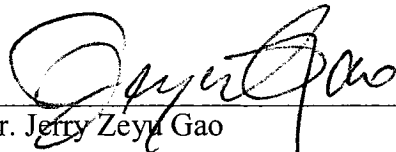
ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

© 2007

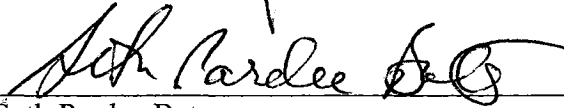
Karen K. Kwok

ALL RIGHTS RESERVED

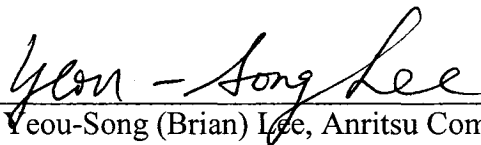
**APPROVED FOR THE DEPARTMENT OF AVIATION AND
TECHNOLOGY**



Dr. Jeffrey Zeyu Gao



Dr. Seth Pardee Bates



Dr. Yeou-Song (Brian) Lee, Anritsu Company

APPROVED FOR SAN JOSE STATE UNIVERSITY



ABSTRACT
A MODEL-BASED TESTING ANALYSIS METHODOLOGY FOR
SOFTWARE INSTALLATION TESTING

by Karen K. Kwok

Software installation testing holds the key to improving software quality because the installation stage is the initial step that unlocks the software. It is critical that customers can install the software smoothly because this can set the tone for the customers' impression of the software's quality. The purpose of software installation testing is to validate that the software can be installed correctly on different computer configurations and conditions by testing its installation functionalities. To accomplish this task, this research proposes both a semantic tree model and a weighted semantic tree model. These two models will help engineers simulate the environment configurations, system running conditions, and installation functions. In addition, this paper will provide some test analysis to conduct software installation testing. As a final step, the two models were validated by applying them in a case study on Turbo Tax software.

ACKNOWLEDGMENTS

A big appreciative thank you to my thesis committee: Dr. Jerry Gao, Dr. Seth Bates, and Dr. Brian Lee for their guidance to put this research and thesis together. Thank you for each of your endless time to read my thesis and to provide invaluable feedback. Thank you Dr. Gao for being an inspiring advisor.

A grateful thank you to my coworkers: Jon Howarth, Eddie Park, Chris Mangone, Joe Golike, Kiet Tram, and the rest of my coworkers, for providing feedback on my thesis and presentation, challenging me to think outside of the box to generate solutions whenever I hit a stumbling block, and being so flexible with my work schedule. An enormous thank you to Eddie Park for spending long hours to proofread my long thesis.

A deep sincere thank you to my family: my dad, my mom, my brother Dennis, my grandparents, and my Aunt Jenny for their continual support and love in my life. I would not be where I am today without them.

A heartfelt thank you to all my friends (Catherine Hornig, Joyce So, Jonathan Chew, Sherrolynn Lee, Dorina Jow, and the rest of my friends too many to mention here) for their never-ending encouragement, optimism, prayers, and being there for me when the going got tough as I journeyed through my master's program. I really appreciate all your support!

And last but not least, I would like to thank God for His strength, wisdom, and love to carry me through my master's program. Without Him, it would not be possible.

Table of Contents

Chapter 1: The Problem and its Setting.....	1
<i>Introduction</i>	1
<i>Statement of the Problem</i>	3
<i>Research Objectives</i>	4
<i>Abbreviations Used in this Study</i>	5
<i>Definitions of Terms</i>	5
<i>Importance of the Study</i>	6
Chapter 2: Review of Related Literature.....	7
<i>Software Installation Testing</i>	7
<i>Model-Based Testing</i>	9
<i>Test Analysis & Measurements</i>	15
<i>Test coverage criteria</i>	15
<i>Cost and test metrics</i>	18
Chapter 3: Model-Based Test Analysis Methodology for Software Installation Testing. 21	
<i>High Level Procedures</i>	21
<i>Software Installation Testing Model</i>	23
<i>Semantic tree model</i>	23
<i>Weighted semantic tree model</i>	24
<i>Software installation testing models</i>	25
<i>Environment configuration semantic tree</i>	26
<i>System running condition semantic tree</i>	27
<i>Installation function semantic tree</i>	29
<i>Spanning Tree Generation Algorithm</i>	30
<i>Spanning tree generation algorithm for the semantic tree model</i>	31
<i>Spanning tree generation algorithm for the weighted semantic tree model</i>	37
<i>Software Installation Testing Measurement & Analysis</i>	45
<i>Test complexity</i>	45
<i>Semantic test complexity equations</i>	45
<i>Spanning tree test complexity equations</i>	50
<i>Ranking semantic tree graphs</i>	53
<i>Inductive proofs for spanning tree rankings</i>	53
<i>Ranking for the SEC and SIC models</i>	55
<i>Ranking for SIF model</i>	61
<i>Test coverage criteria</i>	62
<i>Test metrics</i>	63

<i>Test cost estimation method #1</i>	63
<i>Test cost estimation method #2</i>	67
<i>Alternative method for estimating test cost</i>	69
Chapter 4: Software Installation Testing Results.....	70
<i>Treatment of the Data</i>	70
<i>Model for turbo tax software installation testing</i>	70
<i>Measurement and test analysis for turbo tax</i>	73
<i>Data Analysis</i>	78
Chapter 5: Summary, Conclusions, and Recommendations.....	83
<i>Summary</i>	83
<i>Conclusions</i>	84
<i>Recommendations for Further Research</i>	86
References.....	87
Appendix A: Turbo Tax's SEC Model.....	90
Appendix B: Turbo Tax's SIC Model	97
Appendix C: Turbo Tax's SIF Model.....	103

List of Tables

Table	Page
1. <i>SEC Model Semantic Relations</i>	27
2. <i>SIC Model Semantic Relations</i>	28
3. <i>SIF Model Semantic Relations</i>	30
4. <i>Test Cost Estimation Chart</i>	64
5. <i>Test Complexity for Turbo Tax's SEC Model</i>	73
6. <i>Test Complexity for Turbo Tax's SIC Model</i>	74
7. <i>Ranking for Turbo Tax's SEC Model</i>	75
8. <i>Ranking for Turbo Tax's SIC Model</i>	75
9. <i>Ranking for Turbo Tax's SIF Model</i>	75
10. <i>Test Cost Metrics for Turbo Tax's SEC Model</i>	76
11. <i>Test Cost Metrics for Turbo Tax's SIC Model</i>	77
12. <i>Test Cost Metrics for Turbo Tax's SIF Model</i>	77
13. <i>Summary of Results for the Turbo Tax Software</i>	79

List of Figures

<i>Figure</i>	<i>Page</i>
1. Example of semantic tree.....	23
2. Semantic relation notation.....	24
3. Example of weighted semantic tree.....	25
4. Test space for software installation testing.....	26
5. Example of SEC semantic tree.....	27
6. Example of SIC semantic tree.....	28
7. Example of SIF semantic tree.....	29
8. Semantic spanning tree algorithm: TestComplexity function.....	32
9. Semantic spanning tree algorithm: SetupNodeStack function.....	33
10. Semantic spanning tree algorithm: GenerateSemanticSpanningTree function.....	34
11. Semantic spanning tree algorithm: GenerateAllSemanticSpanningTrees function....	35
12. SEC semantic spanning trees.....	35
13. SIC semantic spanning trees.....	36
14. Weighted semantic spanning tree algorithm: TestComplexity function.....	38
15. Weighted semantic spanning tree algorithm: SetupNodeStack function.....	39
16. Weighted semantic spanning tree algorithm: GenerateWeightedSemanticSpanningTree function.....	40
17. Weighted semantic spanning tree algorithm: GenerateAllWeightedSemanticSpanningTrees function.....	41
18. Example of SEC weighted semantic tree.....	42
19. SEC weighted semantic spanning trees.....	42
20. Example of SIC weighted semantic tree.....	43
21. SIC weighted semantic spanning trees.....	43
22. SEC semantic tree with semantic test complexity.....	48
23. SIC semantic tree with semantic test complexity.....	49
24. SIF semantic tree with semantic test complexity.....	50

25. SEC semantic tree with spanning tree test complexity.....	51
26. SIC semantic tree with spanning tree test complexity.	52
27. Inductive proof #1: Semantic spanning trees within a sub-tree.	54
28. Inductive proof #2: Semantic spanning trees within a semantic tree.....	55
29. SEC semantic tree weighted graph with rankings.	57
30. SEC ranking semantic spanning trees.....	58
31. SIC semantic tree weighted graph with rankings.	59
32. SIC ranking semantic spanning trees.....	59
33. SIF semantic tree weighted graph with rankings.....	62
34. SEC semantic spanning trees with test cost estimation method #1.	65
35. SIC semantic spanning trees with test cost estimation method #1.	66
36. SIF semantic tree with test cost estimations method #2.	68
37. Turbo Tax's SEC semantic tree.	71
38. Turbo Tax's SIC semantic tree.	71
39. Turbo Tax's SIF semantic tree.....	72
40. Graph of Turbo Tax's models' data.....	80
41. Graph of Turbo Tax's spanning trees' data.	80
42. Diagram of the relations between the measurements in Table 13.	81
<i>A1.</i> Turbo Tax's SEC semantic spanning trees.....	90
<i>A2.</i> Turbo Tax's SEC weighted semantic tree.	91
<i>A3.</i> Turbo Tax's SEC weighted semantic spanning trees.....	92
<i>A4.</i> Turbo Tax's SEC semantic tree with semantic test complexity.....	93
<i>A5.</i> Turbo Tax's SEC semantic tree with spanning tree test complexity.....	94
<i>A6.</i> Turbo Tax's SEC ranking semantic tree.	94
<i>A7.</i> Turbo Tax's SEC ranking semantic spanning trees.	95
<i>A8.</i> Turbo Tax's SEC test cost semantic spanning trees using method #1.....	96
<i>B1.</i> Turbo Tax's SIC semantic spanning trees.....	97
<i>B2.</i> Turbo Tax's SIC weighted semantic tree.	98
<i>B3.</i> Turbo Tax's SIC weighted semantic spanning trees.	98
<i>B4.</i> Turbo Tax's SIC semantic tree with semantic test complexity.....	99
<i>B5.</i> Turbo Tax's SIC semantic tree with spanning tree test complexity.....	99

<i>B6.</i> Turbo Tax's SIC ranking semantic tree.....	100
<i>B7.</i> Turbo Tax's SIC ranking semantic spanning trees.....	100
<i>B8.</i> Turbo Tax's SIC test cost semantic spanning trees using method #1.....	101
<i>C1.</i> Turbo Tax's SIF weighted semantic tree.....	103
<i>C2.</i> Turbo Tax's SIF semantic tree with semantic test complexity.....	103
<i>C3.</i> Turbo Tax's SIF semantic tree with spanning tree test complexity.....	104
<i>C4.</i> Turbo Tax's SIF ranking semantic tree.....	104
<i>C5.</i> Turbo Tax's SIF ranking graph.....	105
<i>C6.</i> Turbo Tax's SIF test cost semantic tree using method #2.....	105

Chapter 1

The Problem and its Setting

Introduction

Software testing is the last line of defense in the software development life cycle because this phase is the last chance to detect any lingering defects before the software gets released onto the market. One category of software testing consists of software installation testing, which tests the software installation functions to validate that the software is installed correctly. The current methodology to conduct software installation testing is to use an ad hoc approach, which is not very efficient. In ad hoc testing, specifications are used to tailor individual test cases (Wikipedia, 2007) and to create and manage the test beds (Gao, Tsao, & Wu, 2003, p. 39). This approach is unorganized and not cost-effective. Since ad hoc testing does not offer an organized way to ensure adequate software installation test coverage, a fresh new approach is needed to fill this gap. This research addresses this need by proposing a new model to perform software installation testing.

Since planning and executing software installation testing takes much effort, time, and resources, a well-defined model is needed. This model must be efficient in organizing testing activities and minimizing costs. Thus, this research proposes a new model and analytical approach to test software installation using a method called the

semantic tree model. Within this semantic tree model, there are three core models: the computer environment configuration setup, the system running condition setup, and the software installation functions to be tested. The test analysis that will be utilized includes defining the test criteria, test complexity, and cost metrics. In addition to the semantic tree model, a second model called the *weighted* semantic tree model will extend the first model to present a cost-effective and more efficient test strategy to prioritize the testing by using a ranking system, which adds weights onto the semantic trees.

Currently, there are many software testing models, such as finite state machine models and test generation models, but there are few models that have been designed to perform software installation testing. Some models, such as test automation and test case generation models, offer some approaches to conduct software testing. Because of the complexity of software installation testing, there is always room for improvement on these pre-existing models. Thus, this research proposes a new and different perspective to develop a model that can specifically help improve software installation testing. This research plans to build a semantic tree model along with a weighted semantic tree model that outlines the different combinations of computer settings for testing scenarios when installing software. The environment configuration and system condition models are used to set up the computer settings for software installation. The test cases for the installation function model are then executed to validate that the software was installed correctly. Following the testing, a test analysis methodology will be performed on the new model to demonstrate its cost effectiveness.

Statement of the Problem

Off-the-shelf software must be able to be installed across various computers because users may have different computer settings, such as different operating systems, CPU and hard disk speed and capacity, and amount of RAM. Testing these setting combinations becomes critical because the first contact that customers have with the software involves installing it (Gao, Kwok, & Fitch, 2007, p. 1). If a problem arises during the installation phase, the customer might quit trying to use the software, which means that the customer did not even have a chance to explore the software and see what it has to offer. This would leave a bad first impression of the software quality, and in turn, the customer might consider not buying the same company's software in the future.

This study focuses on developing and validating a new testing model along with defining test coverage and test cost metrics to provide a more efficient and cost effective methodology to conduct software installation testing. In the software industry, not much emphasis has been placed on developing a model for software installation testing because this type of testing is extremely complex and requires many resources (Gao, Kwok, & Fitch, 2007, p. 1). Hence, not much advancement has been made to improve the testing process. Currently, software installation testing lacks a well-defined testing process that can create practical and cost-effective test criteria, systematic test generation methods, and metrics measuring the test complexity, coverage, and costs (Gao, Tirumalasetti, & Hsu, 2006, p. 1; Gao, Kwok, & Fitch, 2007, p. 1). For example, in ad hoc testing, it is difficult to determine the test coverage and cost because it is difficult to know how many test cases are needed. As a result, it has become desirable to develop a new model to

provide a new perspective on conducting software installation testing that may shed some light on improving the current process. This research offers 1) the semantic tree model to help organize the software installation testing process and 2) the weighted semantic tree model to make the software installation testing process more efficient and cost-effective. This research performs testing analysis by defining the test criteria, test complexity, and test cost estimations. The goal of this research is to develop a useful model, assisted by test analysis, to improve the software installation testing process by offering a way to generate the minimal amount of test cases to sufficiently test the specifications.

Research Objectives

The objectives for this study are to define, develop, and apply a new model for software installation testing. The proposed model is called the “semantic tree” model, also known as the “non-weighted” model, and is in fact composed of three semantic tree models: environment configuration, system running condition, and installation function. Within these semantic trees, an algorithm to generate the spanning trees is designed, and the analysis for these models measures and examines the test complexity and the cost metrics. This research also enhances this basic model by designing a “weighted semantic tree” model, also known as the “weighted” model, which assigns weights to each node in addition to having the same exact semantic trees as the non-weighted model. Using the weighted semantic tree model, a strategy called the ranking system is created to prioritize the testing. This can reduce the testing costs by comparing the test execution and cost

savings between the weighted model and testing with the ad hoc approach. Lastly, an application case study using the Turbo Tax software product helps validate the semantic tree and weighted semantic tree models.

Abbreviations Used in this Study

SEC. System Environment Configuration

SIC. System Installation Condition

SIF. System Installation Function

Definitions of Terms

Ad hoc testing. A custom design solution made especially for a specific problem or purpose, where the makeshift solution cannot be generalized nor can be adapted for other purposes (Wikipedia, 2007). Ad hoc testing can stem from inadequate planning or improvising.

Black box testing. This type of testing tests the software from the user's perspective. It validates the functional inputs and outputs of the specifications (Wikipedia, 2006). The source code is not needed in black box testing.

Manual testing. Human interaction is needed in order to execute and validate the test cases. The opposite of manual testing is automation testing, where test cases are automated into scripts to be executed through an automation tool.

Semantics. A set of rules to determine which child nodes to select in a tree.

Software development life cycle. The stages of developing software that includes planning, gathering requirements, designing, analyzing, implementing, generating code, testing, and maintaining.

Software installation testing. A testing process to test the software installation functions to validate that the software is installed correctly.

Spanning tree. An acyclic and undirected graph that connects the tree nodes with links.

Test complexity. The number of test cases needed to achieve adequate test coverage.

Test coverage criteria. A set of rules to determine what needs to be tested.

Importance of the Study

Software installation testing is important because it validates that the software can be installed correctly across different environment configurations and system running conditions by testing the software's installation functions (Gao, Tirumalasetti, & Hsu, 2006, p. 2). If users struggle with installing the software, this can set the tone of how they might feel about the quality of the software. Since the ultimate goal of producing the software stems from satisfying the customers by exceeding their expectations, the quality of the software must be at its highest. Thus, software installation testing becomes essential for software installation validation.

Chapter 2

Review of Related Literature

Software Installation Testing

Software testing is a process that executes the software to verify whether it meets the specifications (Salem, Rekab, & Whittaker, 2004). The purpose of testing software is “not only to ensure the quality of [the] products, but to polish up products by removing faults from the software” (Matsuodani & Tsuda, 2004, p. 1). During software testing, the most important concern is finding all the defects before releasing the software to the public. If defects are not detected and fixed prior to release, it could give the company a bad reputation by causing potentially devastating consequences (Salem, Rekab, & Whittaker, 2004), such as affecting other software to crash or breaking down the whole system and making the software completely unusable. Similarly, software installation testing shares the same objectives of validating the software and catching potential defects.

Basic software installation testing can be performed by running manual or automated test cases on different computer configurations. In a case study by Christopher Agruss (2000), he used installers as an example to illustrate executing automated test cases for installing the software and uninstalling it. To test software installation, first, establish a baseline. Next, decide the types of checking needed, such as disk space, and the testing tool needed to run the test cases. Then designing a flow

diagram of the installer's behavior provides a clear roadmap of what needs to be tested. Finally, the last step verifies the test results by checking whether the installer has written the expected files from the installation to the disk. After testing the software installation, the uninstalling of the software must also be tested in a similar manner to ensure the software installation cycle is complete. This case study shows one approach to test software installation by creating flow diagrams.

Another way to conduct software installation testing is to use the black box methodology. Black box testing validates that the software does what it is supposed to do based on the software's specifications and requirements. The objective of black box testing focuses on the interaction between the software and its environment (Wikipedia, 2006) so that testers do not have to waste their time learning the source code (Kaner, Falk, & Nguyen, 1999). The advantage of using the black box methodology is that the internal source code is not required in order to do the testing. For example, the black box methodology is the best approach to test third party components (Gao, Tsao, & Wu, 2003) because the source code is usually made private and confidential, and thus, not available to people outside of the company. Gao, Tsao, and Wu also suggested that it is important for a tester using the black box testing methods to have the following requirements: 1) to know the software's requirements and specifications, 2) to be familiar with the software testing criteria and method being implemented, and 3) to have a very good understanding of and be knowledgeable about the software. As a result, black box testing can be used to test the software installation functionalities to validate the successfulness of the installation.

Model-Based Testing

The core part of this research revolves around building a model to efficiently carry out software installation testing. By using a model to structure the testing process, it provides a more organized and well-defined approach to efficiently test the specifications. The purpose of a model intends to help “understand, specify, and develop the systems” (Apfelbaum & Doyle, 1997, p. 1). Some of the benefits of using a model consist of promoting more understanding of the system or product, providing a reusable framework for future development, and offering an easy way to update tests that are constantly changing (Apfelbaum & Doyle, 1997; Robinson, 1999b). In addition, according to Robinson, models help eliminate the ambiguity in designing test cases, which becomes a major advantage because the root cause of 60% to 80% of all defects comes from having incorrect requirement specifications (Robinson, 1999b).

A model serves as a framework to generate and verify test cases (El-Far & Whittaker, 2001). El-Far and Whittaker stated that “by using a model that encapsulates the requirements of the system under test, test results can be evaluated based on matches and deviations from what the model specifies and what the software actually does” (p. 2). Moreover, different types of models exist to perform model based testing. Some of the categories for the different models include the following: finite state machine diagram, dynamical system, data model, test generation system, and hierarchical flow graph.

Models can be extremely helpful because they represent simpler versions of a system that helps predict the system’s behavior (Robinson, 1999a). Hence, “model-based testing is a technique that generates software tests from explicit descriptions of an

application's behavior" (Robinson, 1999a, p. 1). One type of model is a finite state machine model, which "consists of a set of states, a set of input events, and the relations between them," where when "given a current state and an input event, the next current state of the model can be determined" (Robinson, 1999a, p. 1). In Robinson's research, he applied the finite state machine model to the Window NT Clock Application. Using this clock on the computer, the testing consisted of starting and stopping the clock and toggling between the analog and digital modes. Then he wrote test code to generate all the possible combinations of the model's states and transitions. To verify that the application behaved correctly, a test oracle was implemented. As a result, defects can be detected by executing and validating the application with the finite state machine model to monitor the transition states and compare them to the expected states. The advantage of using a model comes from the need to update the tests model because "model-based tests are far easier to maintain, review, and update than traditional automated tests" (Robinson, 1999a, p. 10).

To extend Robinson's (1999b) finite state machine model, he combined it with some graph theory techniques. The difference by adding the graph theory is that the "graph theory techniques allow the behavioral information to be stored in the models to generate new and useful tests" (Robinson, 1999b, p. 15). Graph theory techniques can deal directly with a model because "1) new traversals can be automatically generated when the model changes, 2) tests can be constantly changing on the same model, 3) different types of traversals can meet different needs of testers, and 4) the traversal techniques are general and can be reused on different methods" (Robinson, 1999b, p. 15).

Another approach to improve a testing process stems from implementing a model using a dynamical system. In a case study by Stikkel (2005), he developed a dynamic system model to gain a better understanding of the behavior of the system testing process. His model is based on work time effort measurement. Stikkel used a predator-prey analogy to model the testing process where the predators represent the testers who are trying to catch their prey which are the software defects. Using this analogy, Stikkel proposed a dynamic model that measures “the testing effort and the number of faults found during testing” (Stikkel, 2005, pp. 578-579). To implement this model, he applied it to three telecommunication software development projects. The model comprises of collecting data on the number of hours that the workers took to do their testing and computing the number of defects found during testing, then he graphed these results and analyzed them by comparing them with the other models. As a result, Stikkel concluded that the model is reliable and can “achieve the same results in comparison” (Stikkel, 2005, p. 584).

To look at model-based testing in another way, some researchers (Dalal et al., 1999) researched, developed, and applied an automatic test generation model-based method to conduct software testing. The technique involves using an automatic test generation data model to generate test cases. But the main emphasis is on using the test data or specifications as inputs to the software to run test cases and to catch defects. Their approach uses a “specification notation called AETGSpec which is [part] of the AETG software system” (Dalal et al., 1999, pp. 286-287, 293). This is used to generate the different combinations of the input values so that test cases can be crafted for

functional testing to test the usability of the software. The researchers used four case studies to illustrate that the AETG software system generated test cases that effectively detected many of the defects that only could have been caught when using certain pairs of input values. Thus, using a test automation model to generate test cases “reduces the cost of test generation, increases the effectiveness of the tests, and shortens the testing cycle” (Dalal et al., 1999, p. 285).

Similarly, in another case study by some of the same researchers (Dalal, Jain, Karunanithi, Leaton, & Lott, 1998), they expanded their research on model-based testing by implementing a highly programmable system on the United States telephone network. Since talking on the telephone is one of the major communication lines between people, telephones are in high demand. So to improve the reliability of the telephone system while keeping the cost down, the purpose of this case study is to generate comprehensive test sets for each telephone network component. The model consisted of using a real life system called the Integrated Services Control Point (ISCP) to perform testing. Then their AETG software system helped generate tuples, which are combinations of the test data, to be used as inputs to provide pairwise coverage for the valid values. By using different combinations of data inputs, they detected many defects that traditional testing could not have caught. The AETG software system created these input combinations in a manipulator testing table. Then the test cases got executed by inputting the data combinations into the ISCP to test the incoming messages and the outgoing messages by simulating a telephone call. By performing this type of model-based testing, more defects were detected, and in turn, fixed to enhance the quality of the service.

Likewise, another case study (Pretschner et al., 2005) also demonstrates that implementing automated and manual test cases using model-based test suites can increase the number of detected errors. The researchers created test cases for an automotive network controller for infotainment systems to experiment with the automated model-based testing by evaluating the number of error detections and analyzing the model coverage and implementation coverage. After creating their model using different tools, such as AUTOFOCUS and Media Oriented Systems Transport (MOST), they derived four sets of test suites: automated test cases with and without a model and manual test cases with and without a model. Through the experiment, the researchers found that the test suites using the models significantly increased the number of detected errors by 11%. This case study illustrates that using a model to generate test cases can improve the error detection rate.

Another type of model that can be used to generate test cases and to describe a system's behavior is a graphical model. In Apfelbaum and Doyle's case study (1997), they designed a model to conduct testing at the integration and system phase. To generate the test cases, a path with the sequence of user actions traverses through the model that defines the actual scenario of the system. Each path represents one test script to be executed to validate that the system behaves properly. In their long distance calling service case study, they used a hierarchical call flow model to test the call flow sequences and the billing options. The hierarchical call flow model consisted of having a sequence of actions that moved from one state to another based on its behavior. After developing the framework for this model, they reused it to expand their testing for a call-waiting

feature. Their research found that by using a model, the testing time for this new feature was accomplished in 12% of the time usually required and the productivity of the test engineers increased by a factor of five to ten. Therefore, “model based testing is a methodology that has proven its ability to provide dramatic improvements in lowering the cost, increasing the quality, and reducing the time to place the product on the market” (Apfelbaum & Doyle, 1997, p. 12).

Some models that other researches have used include some models that are not so traditional or conventional. Such as in a case study on the Mars Polar Lander performed at Lockheed Martin, “the Test Automation Framework approach was used to model the Touchdown Monitor software requirements” (Blackburn, Busser, Nauman, Knickerbocker, & Kasuda, 2002, p. 1). Using a model to simulate the situation, it helped in identifying the mission failure by recreating the behavior. They found that a premature shutdown of the Mars Polar Lander resulted in a mission failure because “the electrical transient was not processed properly when the three landing legs were extended into their deployed position” (Blackburn et al., 2002, p. 1). The objective of their research was to show that the Test Automation Framework model was capable of detecting any embedded problems in the implementation of the Mars Polar Lander component called the Lander Touchdown Monitor. To implement their model, their toolbox consisted of a SCRtool to simulate the textual requirements, the Test Automation Framework toolset to translate the model, and the T-VEC test generation system to produce test vectors and test drivers. The procedure started off with identifying the requirements for the Touchdown Monitor and then used the SCRtool to translate the textual requirements into the T-VEC

test specifications. Then the T-VEC test generator produced the test cases, which included the inputs, expected outputs, and traceability information. In addition, the T-VEC also generated the test drivers so that the test cases can execute against the Touchdown Monitor code. This Test Automation Framework model was applied to the Mars Polar Lander and helped identify the cause of the failure, which demonstrates that this model can potentially provide a systematic and cost effective approach for testing to verify the software functionalities.

Test Analysis & Measurements

Test coverage criteria. Test coverage can be used as a tool to help define a model to perform software installation testing. Test coverage is a quality indicator that determines the percentage of the test bed that is currently being tested (Connell & Menzies, 1996). To calculate the test coverage, divide the number of test cases currently being executed by the total number of test cases in the test bed. The goal of any test bed strives for achieving 100% test coverage because it ensures that all the specifications both valid and invalid conditions are being covered so that if a defect does occur, it has a higher probability of catching it. But be aware that “achieving 100% test coverage [although in reality is not achievable] does not indicate that the application is error free” (Connell & Menzies, 1996, p. 2) because errors are unpredictable and can be caused by a variety of events that sometimes cannot be simulated.

In a case study by Connell and Menzies (1996), they explore the idea of test coverage and how to implement it using different techniques, and then they apply

building a test coverage tool in the computer language Smalltalk. They describe test coverage as “determining what proportion of a defined piece of computer code has actually been executed during a testing cycle” (Connell & Menzies, 1996, p. 2). But it does not provide any clues about the quality of the testing process. The test coverage analysis only indicates which code has not been tested yet. For the test coverage that exists in the test bed, they can be categorized into the following test coverage categories: statement, branch, loop, Boolean expression, and logical operations. Using the different types of test coverage for testing, the researchers implemented three techniques for a test coverage tool in Smalltalk, in where they found that “100% coverage should not be aimed for” (Connell & Menzies, 1996, p. 11), but suggested that achieving the majority of the coverage is a reasonable goal.

To ensure adequate test coverage, coverage criteria can be used. Coverage criteria are “sets of rules to help determine whether a test suite has adequately tested a program and to guide the testing process” (Memon, Soffa, & Pollack, 2001, p. 1). In their research, Memon, Soffa, and Pollack propose a new way to define the coverage criteria for graphical user interface (GUI) testing “based on the GUI events and their interactions” (p. 11). A software GUI contains graphical objects where the user performs events that manipulate objects that activate change to the state of the software. This can cause the appearance of the GUI objects to change. There is a “need to develop coverage criteria based on events in a GUI” (Memon et al., 2001, p. 1) because 1) GUI’s are developed by using precompiled elements from a library, where the source code may not be accessible for coverage evaluation, and 2) the GUI’s input contains event sequences,

which is conceptually more abstract so it cannot be obtained from the source code.

Their strategy requires the GUI to be broken down into smaller units so that the coverage criteria can be developed for the events within each unit. They define the event coverage, event-interaction coverage, and the length-n event-sequence coverage to ensure that all the interactions in each unit get tested. Using an algorithm, the researchers construct event flow graphs to evaluate these GUI coverage criteria on the test suites. As an example, the authors use MS WordPad to demonstrate their algorithm by creating event flow graphs to test its functionalities such as the main menu, the help menu, and the replace selection. This case study provides a new way of identifying and using coverage criteria to ensure adequate testing is being performed.

Two graduate students at San Jose State University, Tirumalasetti and Hsu (2006), also worked on developing a more efficient way to perform software installation testing by utilizing test coverage criteria. Their methodology to generate test cases involves defining an algorithm for their custom tool called the Install and Patch Test Tool (IPTest) to reduce the cost of testing and to increase customer satisfaction. In addition, they used data from a local software company called Intuit to develop their methodology, and in turn, used it to improve Intuit's testing process. They defined the coverage criteria for a single node coverage, all-node coverage, single link coverage, all-link coverage, single condition-link coverage, all-condition-link coverage, path coverage, and minimum-set path coverage. Using coverage criteria, the test complexity can be defined, and thus, used in the calculation of test coverage. By outlining a well-defined process

and ranking the test cases, this methodology reduces the time and costs to perform testing.

Cost and test metrics. Software testing is extremely expensive, time consuming, and a difficult process (Diaz, Tuya, & Blanco, 2004). But software testing is worth the steep price because in the long run, the benefits outweigh the costs. During testing, defects get detected and fixed which increases the quality of the software and in turn increases customer satisfaction. The most beneficial aspect of software testing comes when the defects get caught early because it costs less to fix in the beginning stages of testing. During the software testing phase when the defects are caught, fixing them is usually done by putting out a new build. This development work does not cost that much compared to finding the defect in the software after customers buy it. If the defect is detected after the software gets released, it gets fixed in a new patch that has to be created and distributed to everyone who bought the software. This cost is much greater than finding the defect during the development and testing phases.

In Woodward and Hennell's research (2005), they also agree with detecting defects at an earlier stage of the software life cycle, such as during the development process phase, in order to cut down on the costs of testing. They found that the National Institute of Standards and Technology (NIST) carried out a study to measure the impact of inadequate software testing. NIST reported that insufficient testing cost the United States economy an estimated amount of \$22.2 to \$59.5 billion annually. NIST suggested that the root of the problem stems from the fact that "most bugs are introduced at the unit

stage” (Woodward & Hennell, 2005, p. 115). So they offered a solution to this problem by advising to implement more effective testing methods at the unit level testing to find the bugs before the units get combined. Based on Woodward and Hennell’s findings, a good strategy to reduce the costs in the long run is to start testing small, such as at the unit or module level, and during the early stages of development.

Like measuring the cost metrics, test metrics measurements play a vital role in the testing process. Test metrics provide a measurable indicator that enables engineers “to detect trends and to anticipate problems, thus providing better control of costs, reducing risks, and improving quality” (Chen, Probert, & Robeson, 2004, p. 1). In a case study by Chen, Probert, and Robeson, they gathered data, calculated the metrics, and analyzed the results for a test process measurement project performed by the IBM Electronic Commerce Development test teams. Their goal of collecting test metrics was to demonstrate that an “effective test process measurement is useful for designing and evaluating cost-effective test strategy” (Chen et al., 2004, pp. 1, 6) and that test metrics also offer suggestions for future improvements. Their findings concluded that in general, the right test metrics selected for a particular project can beneficially measure the testing process and can provide valuable information for important decision-making and future improvements.

The purpose of having test metrics is to “give the software developer assurance that a given test set is sufficiently” effective to test the structure of an application’s specification (Ammann & Black, 1999, p. 1). Ammann and Black’s case study uses “formal methods and testing defined by specification-based coverage metrics to evaluate

test sets” (p. 1). The researchers applied this method by implementing a model checker and mutation analysis. The goal is to detect any syntactic changes made to the program’s structure, which is also known as mutants. To develop the mutation metrics, they used “reflection, expounding, mutation operators, and winnowing procedures” (Ammann & Black, 1999, p. 9) to define test set coverage metrics. The researchers demonstrated that metrics could help provide better test set coverage for specifications, which leads to a higher chance of detecting more mutants or defects.

Chapter 3

Model-Based Test Analysis Methodology for Software Installation Testing

The objective of this research is to develop and validate an efficient model to perform software installation testing. This research presents two models to improve the software installation testing process. The first model, known as the semantic tree model, consists of three non-weighted semantic trees, while the second model, known as the weighted semantic tree model, offers a cost-effective test strategy that contains three weighted semantic trees. Using these models, test analysis can be performed by defining the test criteria, test complexity, and cost metrics. By defining, developing, and applying these models to test software installation, this model-based approach provides an organized way to plan and execute test cases in the testing process.

High Level Procedures

This research will define the software installation testing model and apply it to a case study. This research will carry out the following procedures to develop and validate the software installation testing model. Note that to implement the following procedures, steps one through three must be carried out in this specific order to determine the environment configurations, the system running conditions, and lastly, to execute the test cases to test the software installation functions. This process is required because the

computer needs to be setup first by knowing which configurations and conditions the software installation is being tested on. The installation function test cases can then be used to test whether the software successfully installed on the specified setup.

1. **Identify all the environment configurations.** Create a semantic tree model for these configurations. The goal is to show all the different types of configurations that the software can be installed on.
2. **Identify all the system running conditions.** Create a semantic tree model for these conditions. The goal is to display all the different types of conditions that the software can be installed on.
3. **Identify all the software installation functions to be tested.** Create a semantic tree model for these functionalities. The goal is to illustrate all the installation functions that can be tested to validate the installation.
4. **Design an algorithm to generate all the possible spanning trees.** Applies only to the environment configuration and system running condition models.
5. **Design an algorithm to generate all possible weighted semantic spanning trees.** Use a ranking system to prioritize the environment configurations and system running conditions.
6. **Measure and analyze the test metrics by defining, examining, and reporting the test criteria, test complexity, and cost metrics.** Compare the 3 models by analyzing their test results.
7. **Validate the software installation testing model.** Apply this model to a case study using Turbo Tax software.

Software Installation Testing Model

Semantic tree model. The semantic tree model consists of three acyclic and undirected graphs that can be defined as a 3-tuple $S = (N, E, R)$. N contains the set of nodes in the tree, where each node can have one of the following types: a single root node, an intermediate node, or a leaf node (Gao, Kwok, & Fitch, 2007). E represents the set of edges in the tree, where each edge links a parent node to a child node. R symbolizes the set of relations, where each parent node possesses one of the following semantic relations to its child nodes: AND, EOR, SELECT-1, NAND, or NOT. Figure 1 illustrates an example of a semantic tree while Figure 2 portrays these semantic relationships.

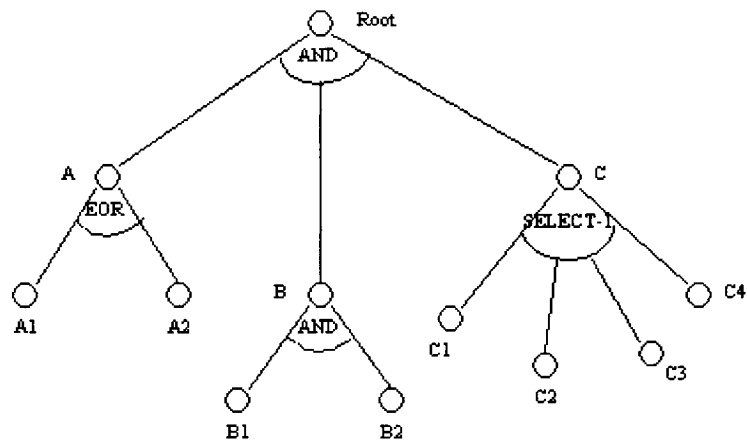


Figure 1. Example of semantic tree.

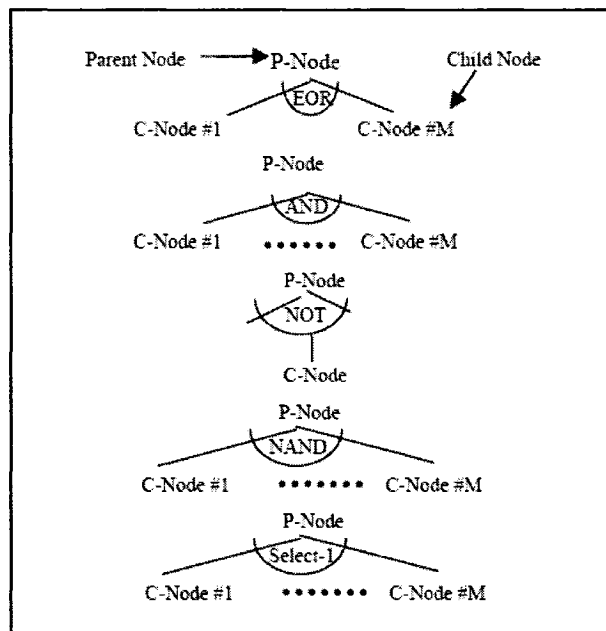


Figure 2. Semantic relation notation.

In this research, a tree is used instead of a graph because a tree does not have cycles and it lays out the choices for the specified computer settings. The environment configuration and system running condition models must be illustrated in a tree, in order to display the different possible setting combinations. To be consistent with the first two tree models, the installation function tree gets presented in a tree as well.

Weighted semantic tree model. The weighted semantic tree model also comprises of acyclic and undirected graphs, but it also possesses weights on each node, where the sum of the weights of the sibling nodes equals to 1 meaning 100%. Figure 3 displays an example of the weighted semantic tree. Since the weighted semantic tree model extends the semantic tree model, the model can be defined in the same way with the weights

attached to each link. So the weighted semantic tree is defined as a 3-tuple $W = (N, E_w, R)$, where N equals to the set of nodes, E symbolizes the set of edges in the tree with a weight factor, and R contains the set of relations for the parent nodes in the tree. The semantic relations are the same as the semantic relations in the semantic tree model. The weighted semantic tree model is created for the purpose of offering a cost-effective testing model by prioritizing the choices in the testing model.

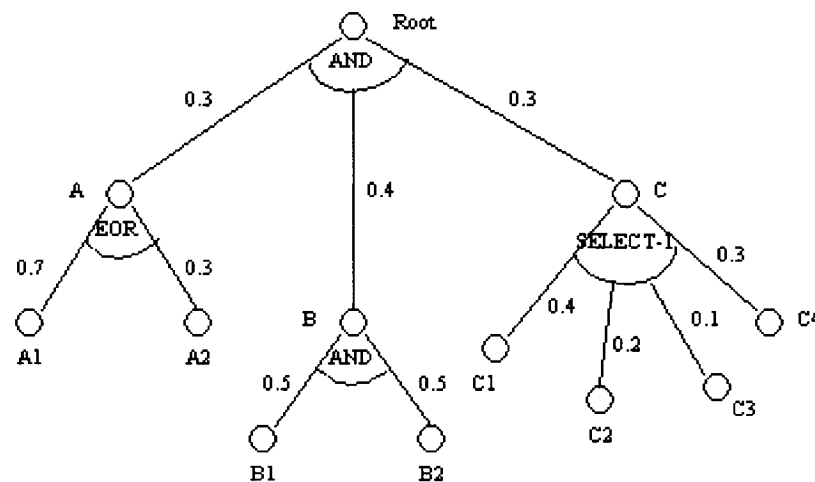


Figure 3. Example of weighted semantic tree.

Software installation testing models. Both the semantic tree model and the weighted semantic tree model are comprised of three tree models: the environment configurations semantic tree, the system running conditions semantic tree, and the installation function semantic tree. By separating the model into three models, this helps present the testing requirements more clearly. These conditions can be seen as a software installation test space, as shown in Figure 4 (Gao, Kwok, & Fitch, 2007). The software

installation test space consists of the x-axis that represents the specified environment configurations, the y-axis that represents the specified system running conditions, and the z-axis that represents the installation functionalities to be tested.

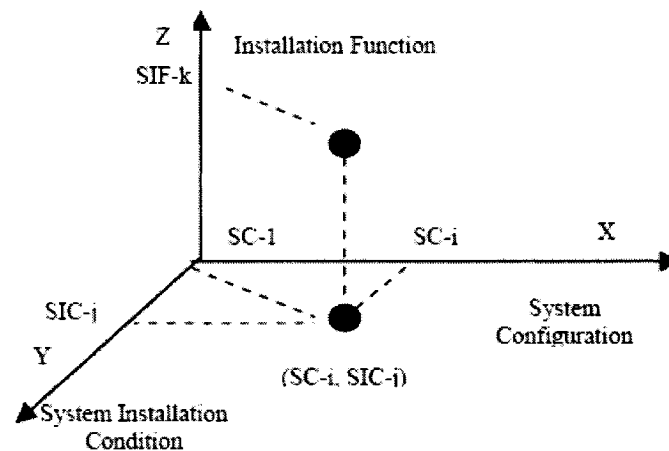


Figure 4. Test space for software installation testing.

Environment configuration semantic tree. Figure 5 shows an example of the environment configuration semantic tree. The same software program will be installed onto each environment configuration combination, where each combination is a semantic spanning tree. Examples of environment configurations include the type of operating system and the service pack version. Table 1 displays the description of each semantic relation's meaning in the environment configuration tree (Gao, Kwok, & Fitch, 2007). This environment configuration part of the software installation testing can be labeled as the System Environment Configuration (SEC) model.

Table 1

SEC Model Semantic Relations

Relations	Semantics in a System Environment Configuration Model
EOR	Node must be provided and set up with only one of its exclusive parts, which are denoted as two child nodes. In other words, the two parts can't be set up at the same time.
AND	Node must be provided only when all of its child nodes are set up.
NOT	Node must be provided without setting up its specific part, denoted as the only child node.
NAND	Node must be provided without the support of some parts, denoted as its child nodes.
Select-1	Node can be set up with any of one of its child nodes.

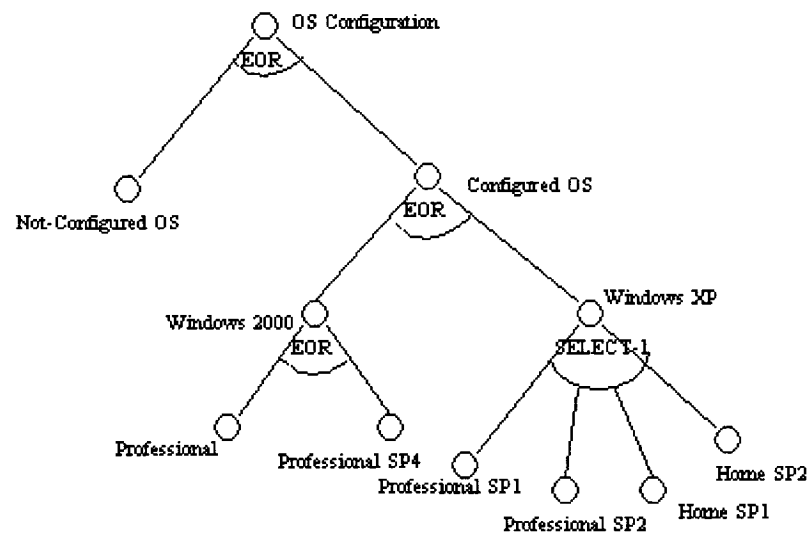


Figure 5. Example of SEC semantic tree.

System running condition semantic tree. After choosing the environment configurations, the system running conditions can vary, so testing must be done on the different combinations of the running conditions. Figure 6 below shows the different components of the system running condition semantic tree, such as the type of user

access, the existence of the software, and the status of the software installation. Table 2 explains the semantic relations in the system running condition tree (Gao, Kwok, & Fitch, 2007). This system running condition part of the software installation testing is named the System Installation Condition (SIC) model.

Table 2

SIC Model Semantic Relations

Relations	Semantics in an Installation Condition Model
EOR	Condition holds true only when one of its two exclusive sub-conditions (denoted as child conditions) holds true.
AND	Condition holds true only when all of its child conditions hold true.
NOT	Condition holds true only when its child condition does not hold true.
NAND	Condition holds true only when all of its child conditions do not hold true.
Select-1	Condition holds true when any one of its child conditions holds true.

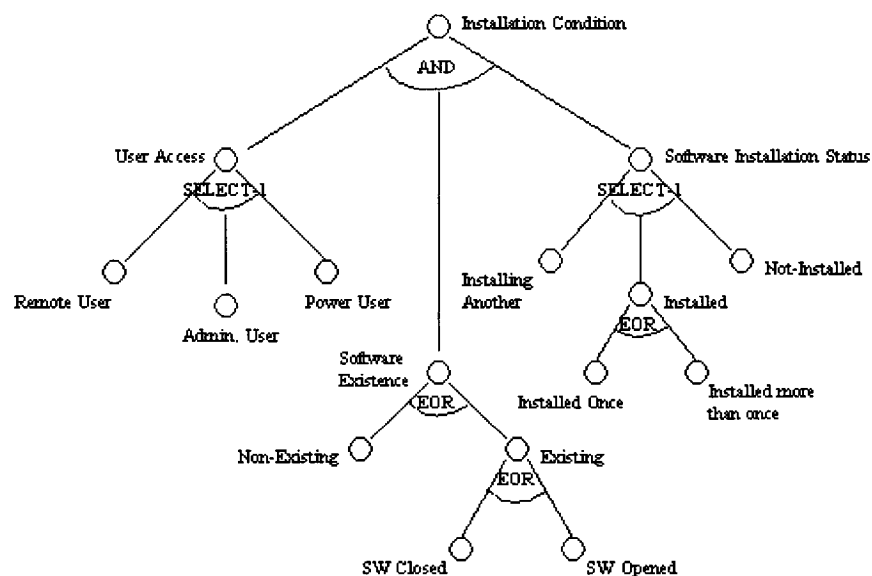


Figure 6. Example of SIC semantic tree.

Installation function semantic tree. After the software program gets installed on a selected configuration and running condition combination, the installation function testing begins. The installation function semantic tree in Figure 7 illustrates the different functions that must be tested, such as the network connections, launching the application after the installation, registering the software, and uninstalling the software. The tree in this figure is only an excerpt of the whole tree because there are too many functions to show. Table 3 describes the semantic relations in the function installation tree (Gao, Kwok, & Fitch, 2007). This function testing part is defined as the System Installation Function (SIF) model.

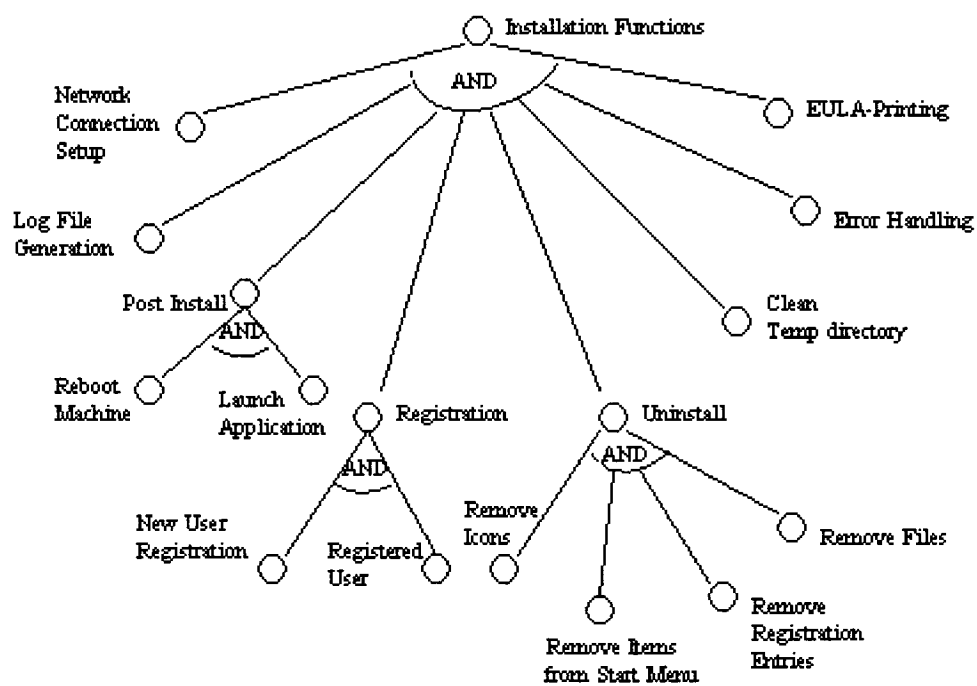


Figure 7. Example of SIF semantic tree.

Table 3

SIF Model Semantic Relations

Relations	Semantics in a System Installation Function Model
EOR	The function is supported only when any of its two exclusive sub-functions (denoted as child nodes) is provided.
AND	The function is supported only when all of its sub-functions (as denoted child nodes) are provided.
NOT	The function is supported without its specific sub-function, denoted as the only child node.
NAND	The function is supported without the support of some parts, denoted as its child nodes.
Select-1	The function is provided when anyone of its sub-functions (denoted as child nodes) is provided.

Spanning Tree Generation Algorithm

In order to produce the semantic spanning trees used in this testing model, an algorithm is developed for creating each possible combination of testing choices. As a result, only the environment configurations model and the system running conditions model need semantic spanning trees. The installation function model does not require any spanning trees because all the functionalities of the software must be tested in order to verify the validity of the software installation.

A semantic tree is a tree where the root node and each intermediate node have one of the following semantic relationships: AND, EOR, SELECT-1, NOT, or NAND. The semantic AND means that all the child nodes must be tested. The semantic EOR symbolizes that only one of the child nodes needs to be chosen. The semantic SELECT-1 represents that one child node out of many nodes must be selected. The semantic NOT assumes that the node only has one child node, but that child node is included in the

spanning tree to show that it is not being supported. Similarly, the semantic NAND has more than one child node and all of the child nodes are included in the spanning tree to indicate that they are not being supported in the software. A semantic spanning tree is where certain nodes are chosen based on the semantic relation to form a tree. To generate the semantic spanning trees, the following algorithms are proposed, one for the semantic tree model and one for the weighted semantic tree model.

Spanning tree generation algorithm for the semantic tree model. In a semantic spanning tree, all the nodes have equal weights. To decide which node to pick based on the semantic rules, an algorithm written in psuedo code has been designed to generate the different spanning trees. Assume the general rule is to go in the direction from left to right when selecting a child node, unless indicated otherwise. The algorithm to generate a semantic spanning tree consists of taking in an input file of a semantic tree. All the nodes and its links are stored in a link list data structure. The semantic relation for each intermediate node gets stored with the parent node in a link list. This algorithm can be seen as generating a forest of spanning trees. A forest can have multiple trees and each tree consists of one or more nodes. In the end, each tree represents one semantic spanning tree and the forest represents a collection of all the possible semantic spanning trees.

This algorithm starts off by determining the test complexity for each node by executing the TestComplexity function. Then each node with the semantic relation of EOR or SELECT-1 needs to setup a stack to keep track of its child nodes. This step is

accomplished by running the SetupNodeStack function. After using these functions to setup the components needed to execute the GenerateSemanticSpanningTree function, the GenerateAllSemanticSpanningTrees function can be executed to generate all the possible semantic spanning trees for a given semantic tree. The GenerateSemanticSpanningTree function generates one semantic spanning tree each time it is executed by selecting different combinations of child nodes to build a spanning tree. The final output of the algorithm produces all the possible semantic spanning trees. The psuedo code for these algorithm functions can be seen in Figures 8 to 11.

```

int testComplexity = 0
TestComplexity(node N)
{
    if node == leaf node
        node.testComplexity = 1;
        return node.testComplexity;
    // else node = parent node
    switch (node N's semantic relation) {
        case 'AND':
            node.testComplexity = 1; // initializing test complexity
            for all child nodes, do
                node.testComplexity =
                    node.testComplexity * TestComplexity(child node);
            break;
        case 'EOR' or 'SELECT-1':
            node.testComplexity = 0; // initializing test complexity
            for each child node, do
                node.testComplexity =
                    node.testComplexity + TestComplexity(child node);
            break;
        case 'NAND':
            node.testComplexity = 0;
            break;
        default 'NOT':
            node.testComplexity = 0;
            break;
    }
    return testComplexity = root node's testComplexity;
}

```

Figure 8. Semantic spanning tree algorithm: TestComplexity function.

```

boolean EORSELAND = false;

SetupNodeStack(node N)
{
    if node is a leaf node
        return;
    else // node is a parent node
        switch (node N's semantic relation) {
            case 'AND' or 'NAND':
                for all child nodes, do // go from RIGHT to LEFT
                    if (child node is the leftmost child with semantic relation
                        EOR or SELECT-1)
                        childNode.color = green;
                    if (EORselAND == true && child node has semantic
                        relation EOR or SELECT-1)
                        node.color = purple;
                        EORselAND = false;
                    if (EORselAND == true && child node DOES NOT have semantic
                        relation EOR or SELECT-1 && childNode.color == green)
                        EORselAND = false;
                        SetupNodeStack(child node);
                return;
            case 'EOR' or 'SELECT-1':
                create stack // node.stack
                for each child node, do // go from RIGHT to LEFT
                    push child node onto node.stack
                    childNode.iteration = 0; // initialize
                    if (parent node has semantic relation AND && child node
                        has semantic relation EOR or SELECT-1)
                        childNode.color = red;
                    else if (child node has semantic relation AND)
                        EORselAND = true;
                    if (node's parent node has semantic relation = AND)
                        childNode.iteration = parent.testComplexity / node.testComplexity;
                    if (node.color != green)
                        childNode.iteration = childNode.iteration /
                            ( $\prod$  (test complexity of siblings left of child node));
                    if (childNode.color == red)
                        childNode.iteration =
                            childNode.iteration * node.testComplexity;
                    else if (node.color == red)
                        childNode.iteration = node.iteration / node.testComplexity;
                    SetupNodeStack(child node);
                return;
            default 'NOT':
                for all child nodes, do
                    SetupNodeStack(child node);
                return;
        }
    return;
}

```

Figure 9. Semantic spanning tree algorithm: SetupNodeStack function.

```

GenerateSemanticSpanningTree(node N)
{
    if node == leaf node
        add node to spanning tree
        return;

    // else it's a parent node
    switch (node N's semantic relation) {
        case 'AND': // semantic relationship = AND
            add node to spanning tree
            // select ALL child nodes from LEFT to RIGHT
            for all child nodes, do
                GenerateSemanticSpanningTree(child node);
            if (node.color == purple)
                push node back onto parent.stack
            return;
        case 'EOR' || 'SELECT-1': // semantic relationship = EOR or SELECT-1
            add node to spanning tree
            // select child node from LEFT to RIGHT
            // choose child node from top of node.stack
            childNode = pop(node.stack);
            childNode.counter++;
            GenerateSemanticSpanningTree(childNode);
            if (childNode.counter < childNode.iteration)
                push child node back onto node.stack
            if (node.color == red && node.stack != empty)
                push node back onto parent.stack
            if (node.color == green && node.stack == empty)
                parent.color = white;
            if (node.stack == empty)
                push all child nodes back onto node.stack
            return;
        case 'NAND': // semantic relationship = NAND
            // include in spanning tree to show that the child nodes are not supported
            add node to spanning tree
            for all child nodes, do // select child node from LEFT to RIGHT
                GenerateSemanticSpanningTree(child node);
            return;
        default 'NOT': // semantic relationship = NOT
            // include in spanning tree to show that the child node is not supported
            add node to spanning tree
            GenerateSemanticSpanningTree(child node);
            return;
    }
return;
}

```

Figure 10. Semantic spanning tree algorithm: GenerateSemanticSpanningTree function.

```

int SpanningTreeCounter = 0;

GenerateAllSemanticSpanningTrees(node N)
{
    GenerateSemanticSpanningTree(N);
    SpanningTreeCounter++;
    if SpanningTreeCounter < testComplexity
        GenerateAllSemanticSpanningTrees(root node);

    // if testComplexity == SpanningTreeCounter
    return all semantic spanning trees;
}

```

Figure 11. Semantic spanning tree algorithm: GenerateAllSemanticSpanningTrees function.

To illustrate that this algorithm generates semantic spanning trees, the first couple of semantic spanning trees will be generated for the environment configuration and system running condition semantic trees. The environment configuration's spanning trees can be combined with the system running condition spanning trees so that software can be installed on a computer with these specified setup conditions. Figure 12 illustrates the first three semantic spanning trees for the SEC model.

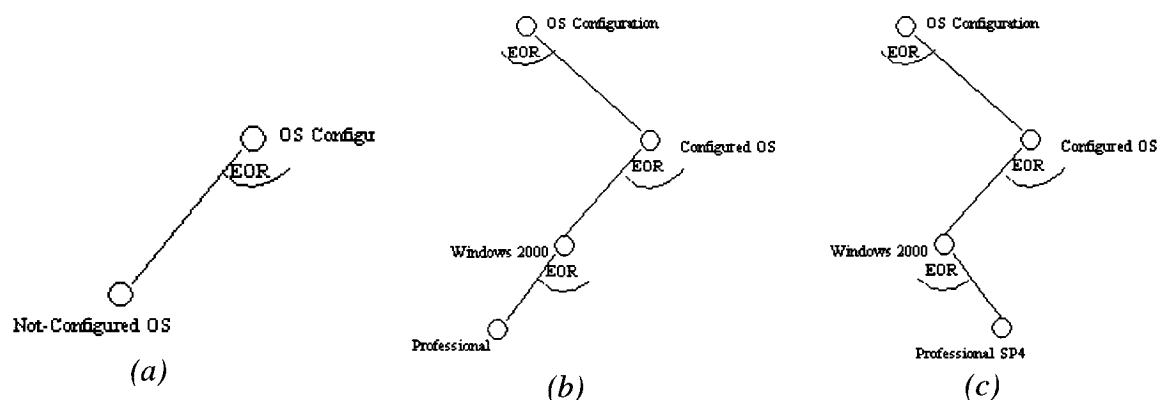


Figure 12. SEC semantic spanning trees.

Figure 13 demonstrates the first three system running condition semantic spanning trees that can be generated using the semantic spanning tree generation algorithm. The number of spanning trees represents the number of possible combinations within the system running condition semantic tree.

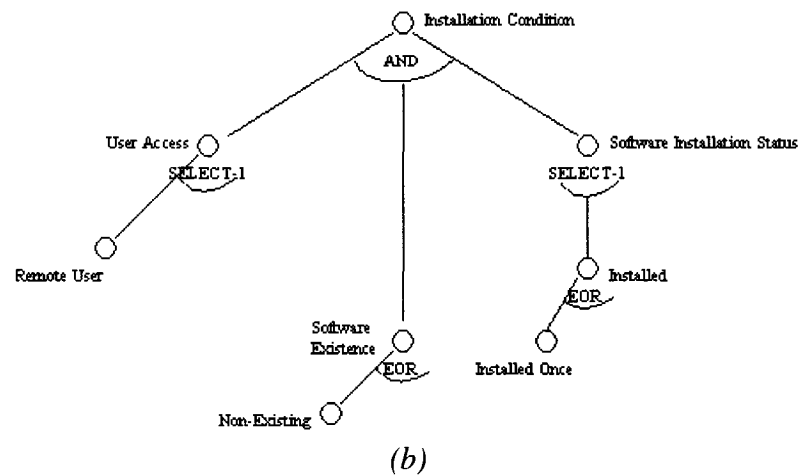
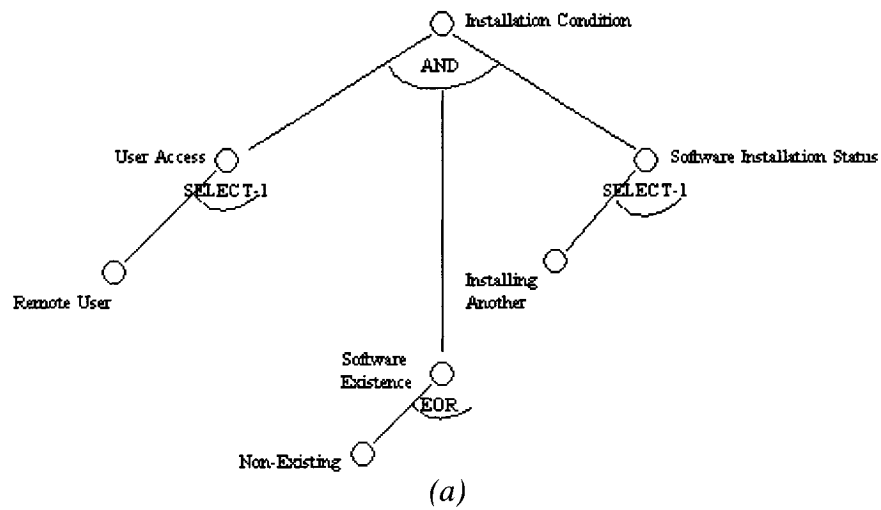


Figure 13. SIC semantic spanning trees.

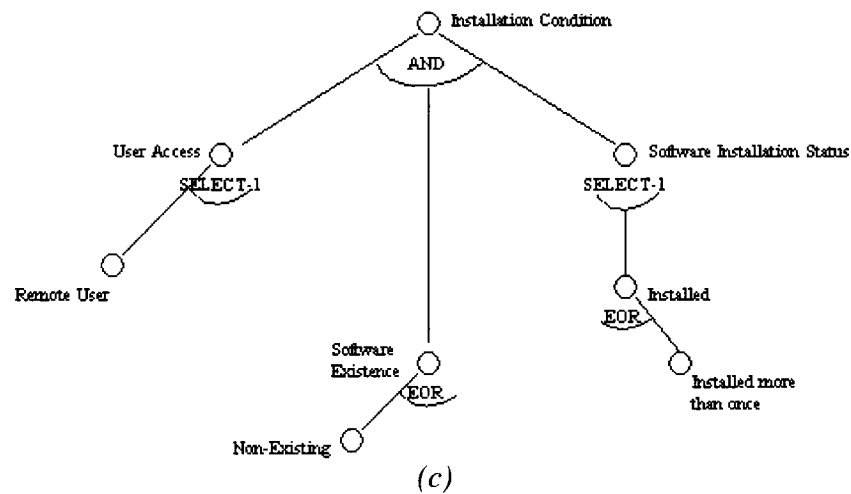


Figure 13. SIC semantic spanning trees (continued).

Spanning tree generation algorithm for the weighted semantic tree model. In a weighted semantic spanning tree, each node has a weight assigned to it in the form of a percentage. A high percentage means that it is critical to test the component. To generate all possible weighted semantic spanning trees, the following algorithm has been developed. This algorithm is very similar to the semantic spanning tree algorithm above, with the exceptions of having weights on each link in the semantic tree and choosing the child node with the largest weight first. The TestComplexity function gets executed first just like the semantic spanning tree algorithm. The SetupNodeStack function is run next to set up the stacks for all parent nodes. The stack sequence consists of the child node with the highest weight at the top of the stack all the way to the child node with the lowest weight at the bottom of the stack. Finally, the GenerateAllWeightedSemanticSpanningTrees function is executed to generate all the possible weighted semantic spanning trees from the most importance to the least

importance, using the `GenerateWeightedSemanticSpanningTree` function, which generates one unique weighted semantic spanning tree each time it is executed. The final output for this algorithm produces all the possible weighted semantic spanning trees. The psuedo code for these algorithm functions can be seen in Figures 14 to 17.

```

int testComplexity = 0
TestComplexity(node N)
{
    if node == leaf node
        node.testComplexity = 1;
        return node.testComplexity;
    // else node = parent node
    switch (node N's semantic relation) {
        case 'AND':
            node.testComplexity = 1; // initializing test complexity
            for all child nodes, do
                node.testComplexity =
                    node.testComplexity * TestComplexity(child node);
            break;
        case 'EOR' or 'SELECT-1':
            node.testComplexity = 0; // initializing test complexity
            for each child node, do
                node.testComplexity =
                    node.testComplexity + TestComplexity(child node);
            break;
        case 'NAND':
            node.testComplexity = 0;
            break;
        default 'NOT':
            node.testComplexity = 0;
            break;    }
    return testComplexity = root node's testComplexity;
}

```

Figure 14. Weighted semantic spanning tree algorithm: TestComplexity function.

```

boolean EORSELAND = false;

SetupNodeStack(node N)
{
    if node is a leaf node
        return;
    else // node is a parent node
        switch (node N's semantic relation) {
            case 'AND' or 'NAND':
                create stack // node.stack
                // stack sequence: node with smallest weight on bottom, largest weight on top
                // arrange child nodes from smallest to largest weight
                for all child nodes, do
                    push child node onto node.stack
                    if (child node is leftmost child with semantic relation EOR or SELECT-1)
                        childNode.color = green;
                    if (EORselAND == true && child node has semantic relation EOR or
                        SELECT-1)
                        node.color = purple;
                        EORselAND = false;
                    if (EORselAND == true && child node DOES NOT have semantic
                        relation EOR or SELECT-1 && childNode.color == green)
                            EORselAND = false;
                            SetupNodeStack(child node);
                return;
            case 'EOR' or 'SELECT-1':
                create stack // node.stack
                // stack sequence: node with smallest weight on bottom, largest weight on top
                // arrange child nodes from smallest to largest weight
                for each child node, do // go from RIGHT to LEFT
                    push child node onto node.stack
                    childNode.iteration = 0; // initialize
                    if (parent node has semantic relation AND && child node has semantic
                        relation EOR or SELECT-1)
                        childNode.color = red;
                    else if (child node has semantic relation AND)
                        EORselAND = true;
                    if (node's parent node has semantic relation = AND)
                        childNode.iteration = parent.testComplexity / node.testComplexity;
                        if (node.color != green)
                            childNode.iteration = childNode.iteration /
                                ([ test complexity of siblings left of child node);
                        if (childNode.color == red)
                            childNode.iteration =
                                childNode.iteration * node.testComplexity;
                    else if (node.color == red)
                        childNode.iteration = node.iteration / node.testComplexity;
                    SetupNodeStack(child node);
                return;
            default 'NOT':
                for all child nodes, do
                    SetupNodeStack(child node);
                return;
        }
    return;
}

```

Figure 15. Weighted semantic spanning tree algorithm: SetupNodeStack function.

```

GenerateWeightedSemanticSpanningTree(node N)
{
    if node == leaf node
        add node to spanning tree
        return;

    // else it's a parent node
    switch (node N's semantic relation) {
        case 'AND': // semantic relationship = AND
            add node to spanning tree
            // select ALL child nodes from largest to smallest weight
            while node.stack != empty, do
                childNode = pop(node.stack);
                GenerateWeightedSemanticSpanningTree(childNode);
            if (node.color == purple)
                push node back onto parent.stack
            return;
        case 'EOR' || 'SELECT-1': // semantic relationship = EOR or SELECT-1
            add node to spanning tree
            // select child node from largest to smallest weight
            // choose child node from top of node.stack
            childNode = pop(node.stack);
            childNode.counter++;
            GenerateWeightedSemanticSpanningTree(childNode);
            if (childNode.counter < childNode.iteration)
                push child node back onto node.stack
            if (node.color == red && node.stack != empty)
                push node back onto parent.stack
            if (node.color == green && node.stack == empty)
                parent.color = white;
            if (node.stack == empty)
                push all child nodes back onto node.stack
            return;
        case 'NAND': // semantic relationship = NAND
            // include in spanning tree to show that the child nodes are not supported
            add node to spanning tree
            // select child node from largest to smallest weight
            while node.stack != empty, do
                GenerateWeightedSemanticSpanningTree(child node);
            return;
        default 'NOT': // semantic relationship = NOT
            // include in spanning tree to show that the child node is not supported
            add node to spanning tree
            GenerateWeightedSemanticSpanningTree(child node);
            return;
    }
}
return;
}

```

Figure 16. Weighted semantic spanning tree algorithm:

GenerateWeightedSemanticSpanningTree function.

```
int WeightedSpanningTreeCounter = 0;

GenerateAllWeightedSemanticSpanningTrees(node N)
{
    WeightedSemanticSpanningTree (N);
    WeightedSpanningTreeCounter++;
    if WeightedSpanningTreeCounter < testComplexity
        GenerateAllWeightedSemanticSpanningTrees(root node);

    // if testComplexity == WeightedSpanningTreeCounter
    return all weighted semantic spanning trees;
}
```

Figure 17. Weighted semantic spanning tree algorithm:

GenerateAllWeightedSemanticSpanningTrees function.

To demonstrate that this algorithm generates a weighted semantic spanning tree, a few of the weighted semantic spanning trees will be generated for the environment configuration and system running condition semantic trees. Each node has been assigned a subjective weight to reflect its priority. The sum of all the weights of the sibling nodes equals to 1, meaning 100%. Figure 18 illustrates the environment configuration weighted semantic tree that serves as the input for the algorithm. Figure 19 shows the first three weighted semantic spanning trees for the SEC model.

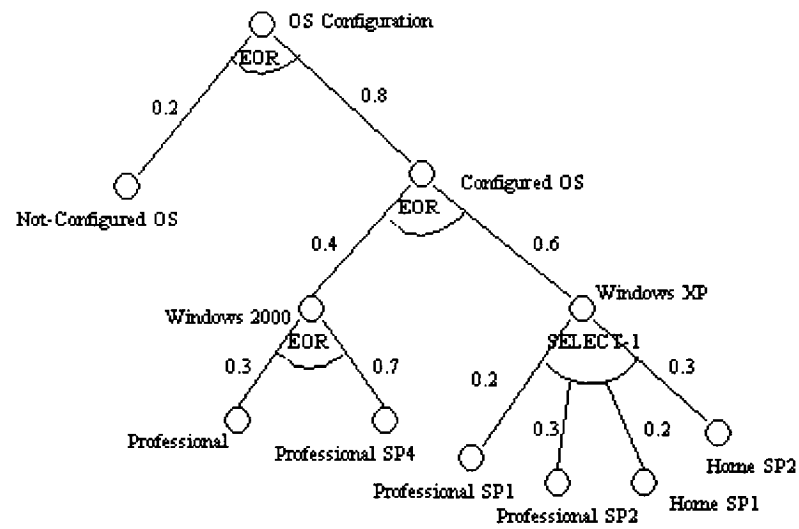


Figure 18. Example of SEC weighted semantic tree.

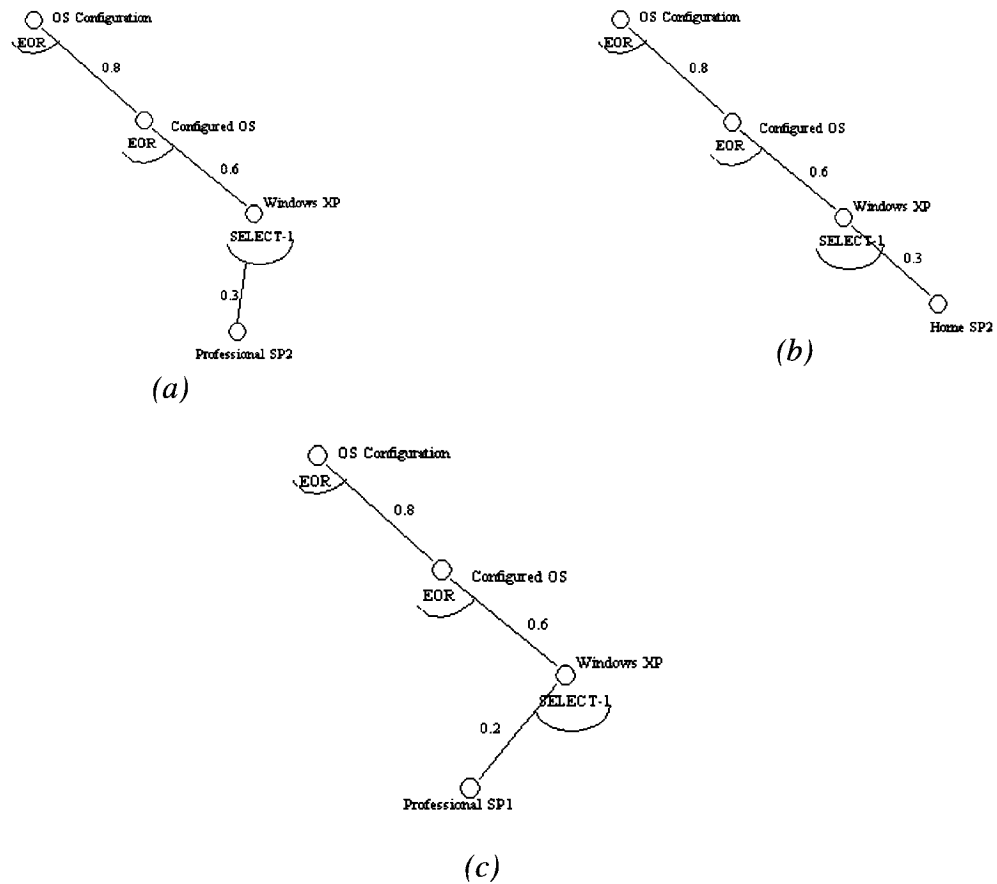


Figure 19. SEC weighted semantic spanning trees.

In the SIC model, the weighted semantic spanning trees are produced in the same way as the SEC model. Figure 20 illustrates the system running condition weighted semantic tree. Figure 21 displays the first three weighted semantic spanning trees generated for the SIC model.

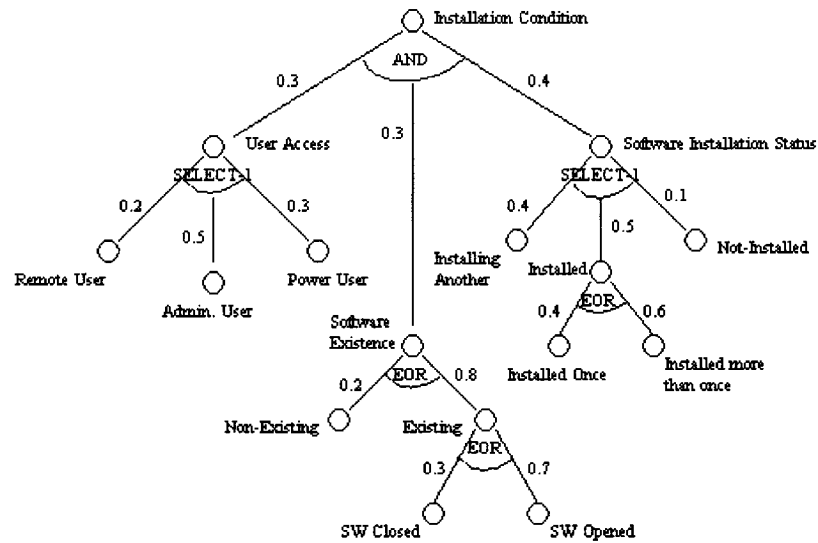


Figure 20. Example of SIC weighted semantic tree.

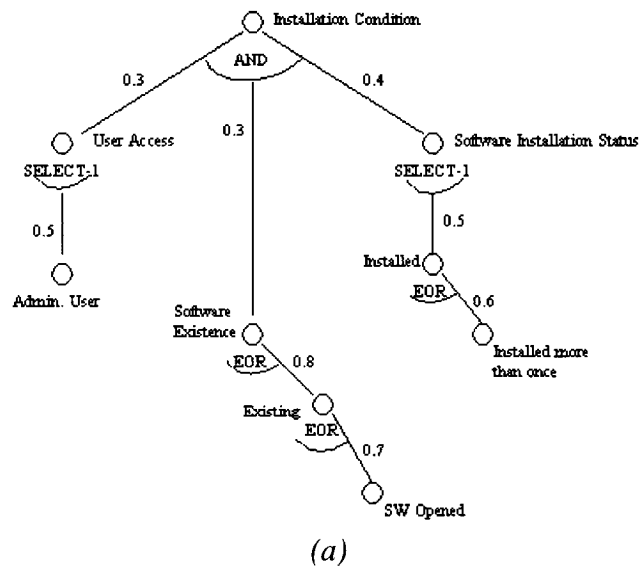
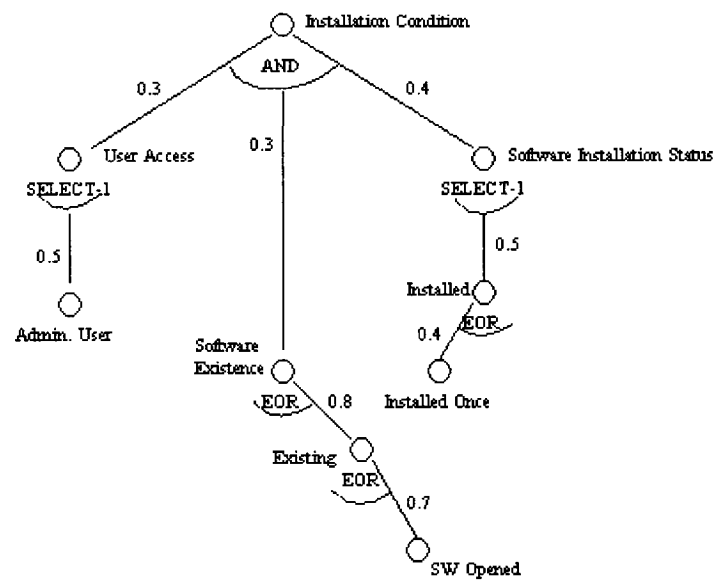
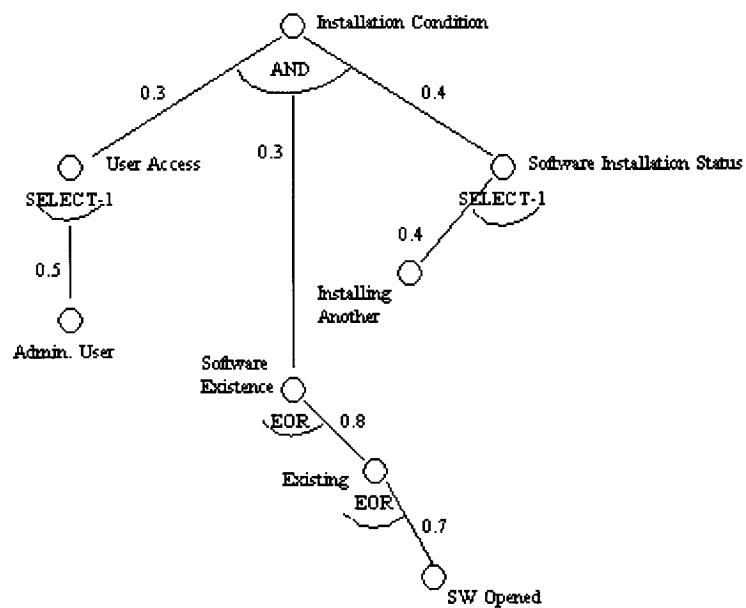


Figure 21. SIC weighted semantic spanning trees.



(b)



(c)

Figure 21. SIC weighted semantic spanning trees (continued).

Software Installation Testing Measurement & Analysis

Now that the algorithms for generating semantic spanning trees have been presented, let's utilize these algorithms by measuring the test complexities, prioritizing the test conditions and functionalities, and estimating the cost. These testing measurements and analyses will help demonstrate the benefits and cost-effectiveness of these new models.

Test complexity. Software testing involves testing everything from checking the expected result to catching the invalid results. Testing the expected results ensures that the software functions correctly. Testing the invalid results ensures that the software catches errors and throws the appropriate error messages because if these errors go uncaught and not fixed, the software might crash, go into infinite loops, or produce misleading results. Since many different test cases must be tested, test coverage helps keep track of the large amount of test cases. To break down the test coverage to a more workable size, the test complexity calculates 1) the number of test cases needed to sufficiently cover the specifications and 2) the number of possible spanning trees that can be generated. The test complexity uses the bottom-up approach and the parent node's semantic relationship to compute the complexity for each parent node in the tree. The test complexity will be calculated for the three models: SEC, SIC, and SIF.

Semantic test complexity equations. The semantic test complexity calculates the number of test cases that is needed to adequately test the specifications. A typical

situation has leaf nodes with given test complexities of integer values greater than 1.

Thus, the semantic test complexity can be calculated with the following formulas.

If the node has the semantic relation **AND**, this means that *all* the child nodes for that node must be tested. Thus, the test complexity, $T_{Complexity}$, for the node equals the summation of the test complexity of each child node, C_i , which is shown in Equation 1.

$$T_{Complexity} = \sum_{i=1}^n C_i \quad (\text{Equation 1})$$

If the node has the semantic relation **EOR**, this means that only *one* of the two child nodes for that node requires to be tested. Thus, the test complexity, $T_{Complexity}$, for the node equals the sum of the test complexity of each child node, C_i , which is also the same as the semantic AND in Equation 1.

If the node has the semantic relation **SELECT-1**, this means that *one* of the n child nodes for that node needs to be tested. Thus, the node's test complexity, $T_{Complexity}$, equals the sum of the test complexity of each child node, C_i , which is also the same as the semantic AND and EOR in Equation 1.

If the node has the semantic relation **NOT**, this means *do not include the child node* for the node to be tested. Note that the semantic NOT can only be used in the case where the node only has one child node. Thus, the test complexity, $T_{Complexity}$, for the node equals to 0 because no child nodes are supported or tested. The test complexity for the semantic NOT is shown in Equation 2.

$$T_{Complexity} = 0 \quad (\text{Equation 2})$$

If the node has the semantic relation **NAND**, this means *do not include any of the child nodes* for the node to be tested. The semantic NAND is usually used for a node

with more than one child node. Thus, the test complexity, $T_{Complexity}$, for the node equals to 0 because none of the child nodes are supported or tested. The test complexity for the semantic NAND is the same as the semantic NOT, shown in Equation 2.

To help illustrate the calculations of these semantic test complexity formulas, the test complexity for the environment configuration and system running condition semantic trees will be computed for leaf nodes with assigned test complexities greater than 1. The calculations start from the bottom of the tree and work its way up to the root node. The test complexity of the root node is the test complexity for the whole tree, which means that this is the number of test cases required to have adequate coverage for the components tested in the particular tree.

In the SEC model, to calculate the semantic test complexity in Figure 22, Equation 1 is used because this tree only contains the semantics EOR and SELECT-1. As an example, to calculate the semantic test complexity of Windows XP, the test complexity of all its child nodes adds up to 12. The overall semantic test complexity of this SEC semantic tree equals 25, which means that it takes 25 test cases to sufficiently test the computer's environment configuration specifications.

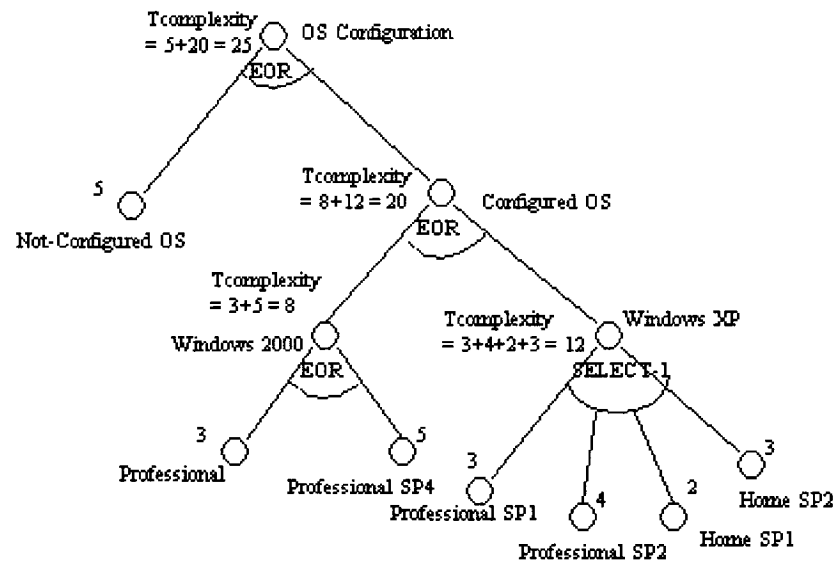


Figure 22. SEC semantic tree with semantic test complexity.

Figure 23 illustrates the semantic test complexity calculations for the SIC semantic tree. Assume that each leaf node has been given a test complexity that is greater than 1. For example, to compute the semantic test complexity for the User Access node, add the test complexity of all its child nodes, which comes out to 11. The overall semantic test complexity for the SIC semantic tree equals 28, meaning that it requires 28 test cases to adequately test the computer's running condition specifications.

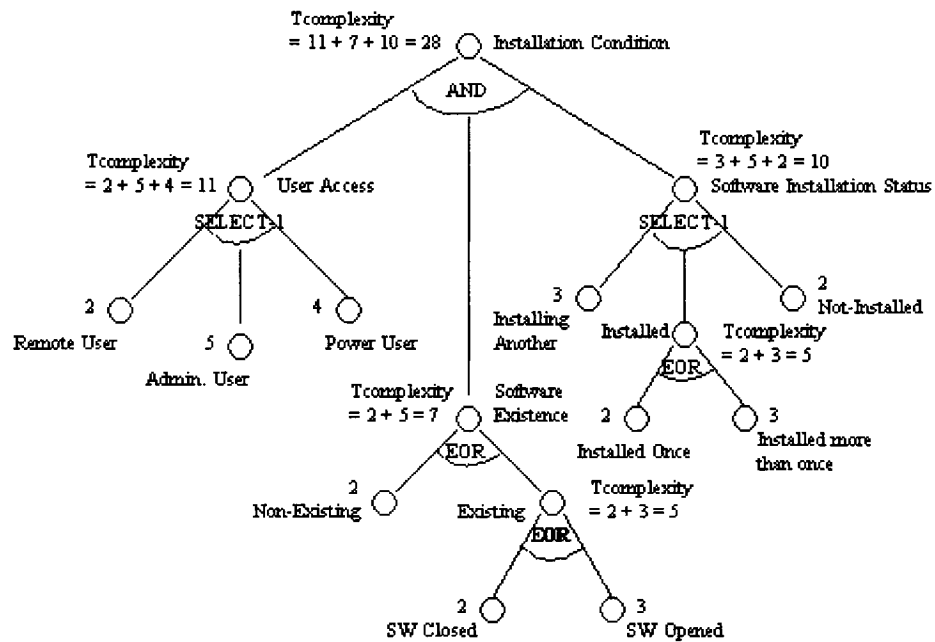


Figure 23. SIC semantic tree with semantic test complexity.

Figure 24 displays the semantic test complexity calculations for the installation function semantic tree. Assume that the test complexity has been assigned to each leaf node. After adding all the test complexities for the leaf nodes, the semantic test complexity for the SIF semantic tree equals 21 test cases. This means that 21 test cases need to be tested in order to sufficiently test all the functionalities.

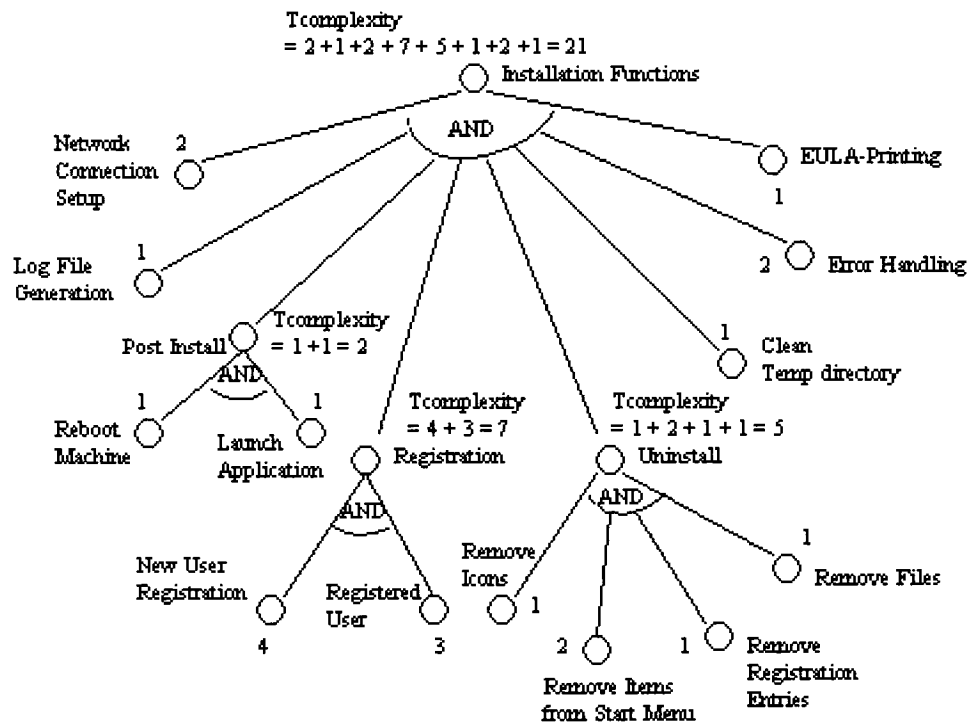


Figure 24. SIF semantic tree with semantic test complexity.

Spanning tree test complexity equations. The spanning tree test complexity calculates the number of possible spanning trees that can be generated using the algorithm in Figures 8 to 11 and Figures 14 to 17. To calculate the spanning tree test complexity, assume that all the leaf nodes have a given test complexity of 1. All of the equations to calculate the semantic test complexity apply to calculating the spanning tree test complexity except for the semantic AND test complexity. Since all the child nodes for that node must be tested for the semantic AND node and all the leaf nodes have a test complexity of 1, only one spanning tree needs to be tested. Thus, the test complexity, $T_{\text{Complexity}}$, for the semantic AND node equals the multiplication of the test complexity of each child node, C_i , which is shown in Equation 3.

$$T_{\text{Complexity}} = \prod_{i=1}^n C_i \quad (\text{Equation 3})$$

In the SEC tree model, to figure out the number of different combinations that exist within a semantic tree, the spanning tree test complexity needs to be calculated. Figure 25 demonstrates the spanning tree test complexity calculations for the environment configuration semantic tree. For example, to compute the spanning tree test complexity for a configuration with Windows XP, first look at the semantic relation. Since it has the semantic SELECT-1, the spanning tree test complexity equals the summation of the child nodes' test complexities, which equals 4. Likewise, the final spanning tree test complexity for this example computes to 7. This implies that this environment configuration semantic tree has 7 possible spanning trees.

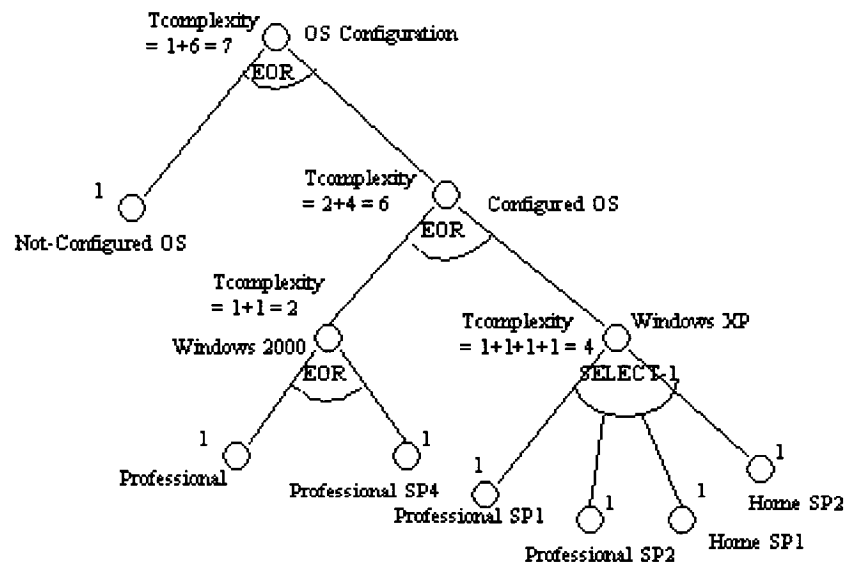


Figure 25. SEC semantic tree with spanning tree test complexity.

Figure 26 illustrates the spanning tree test complexity calculations for the SIC model. Assume that each leaf nodes has an assigned test complexity of 1. For example, the User Access node has the semantic relationship SELECT-1 and has three child nodes. Using equation 1, the spanning tree test complexity for the User Access node equals 3 test cases. Thus, the root node has a spanning tree test complexity of 36, meaning that this system running condition semantic tree has 36 possible spanning trees.

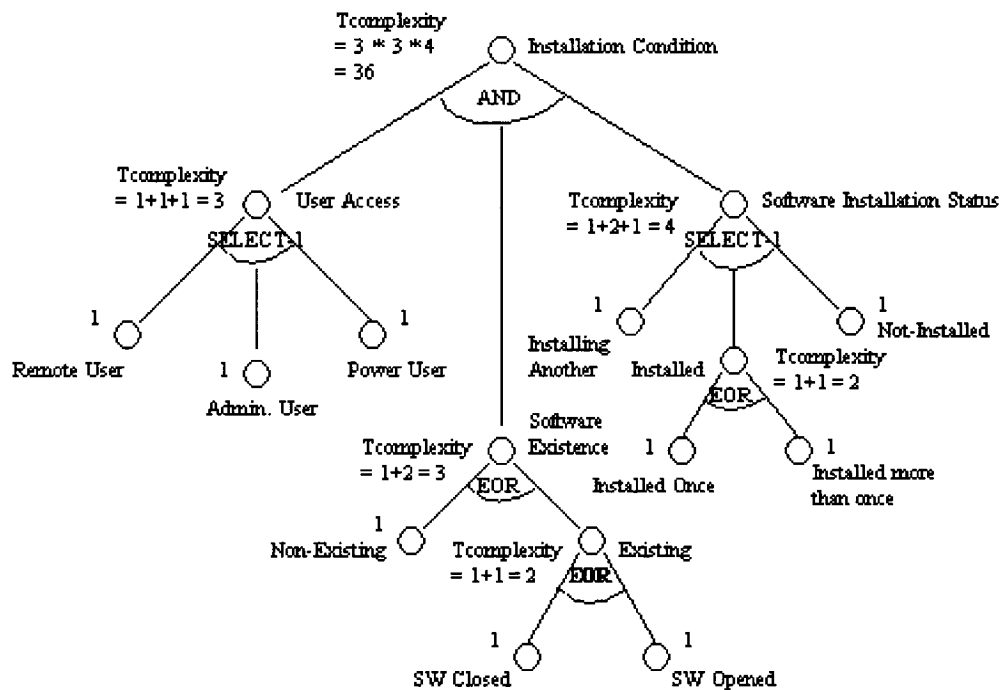


Figure 26. SIC semantic tree with spanning tree test complexity.

The SIF model does not require any spanning trees because all of the functions must be tested. Thus, in most general cases, the installation function semantic tree has a spanning tree test complexity equal to 1.

Ranking semantic tree graphs. A weighted graph can be used to prioritize the settings and the test execution of the three models. This prioritization is necessary in order to battle one of software's all-time enemies: time. Testing can go on forever because there are always new conditions to test, new inputs to try, or new strategies to implement, but time puts a constraint on the number of test cases that can be executed. For this reason, test cases must be organized by priority, where the critical tests get identified and executed first. A higher weight represents a higher importance and must be tested, while a lower weight symbolizes a lower importance and not as crucial to test. Each node in the semantic tree gets assigned a weight, which is a decimal value representing a percentage. In the weighted semantic tree, the weights near the top of the tree have a heavier emphasis than the weights at the bottom of the tree because the top of the tree starts out with more general conditions or functionalities to test, then as the tree gets deeper, the bottom of the tree has more specific conditions or functionalities to test. This ranking strategy provides a cost-effective model to test software installation.

Inductive proofs for spanning tree rankings. When the semantic trees have weights, the generated semantic spanning trees receive a total ranking. This ranking can be used to prioritize the setup sequence or the order of test execution. Therefore, semantic spanning tree #1 must have a larger ranking than semantic spanning tree #2. The following two inductive proofs in Figures 27 and 28 support this theory.

Conjecture: the ranking of semantic spanning tree T_A is greater than or equaled to the ranking of semantic spanning tree T_B for a sub-tree.

1) Assume that there are 2 semantic spanning trees with rankings. Let the weight in spanning tree $T_A = x$, and the weight in spanning tree $T_B = y$, where $x \geq y$. Then

$$\text{Ranking}_A = \sum_{i=1}^n (x * 10^{(1-i)}) \geq \text{Ranking}_B = \sum_{i=1}^n (y * 10^{(1-i)})$$

2) Base Case: If $n = 1$, then

$$\text{Ranking}_A = \sum_{i=1}^1 (x * 10^{(1-i)}) \geq \text{Ranking}_B = \sum_{i=1}^1 (y * 10^{(1-i)})$$

Since both sides have $10^{(1-i)}$, this gets cancelled out and the only part that is left are the weights on both sides: $x \geq y$, which holds true from step #1.

3) Inductive Hypothesis: If $n = k$, then

$$\text{Ranking}_A = \sum_{i=1}^k (x * 10^{(1-i)}) \geq \text{Ranking}_B = \sum_{i=1}^k (y * 10^{(1-i)})$$

Likewise, $10^{(1-i)}$ is on both sides, so this gets cancelled out. Since the top of the tree has a heavier impact on the weights, the only part of the equation that matters is the weight, so the equation equals to: $x \geq y$, which is true.

4) Inductive Step: If $n = k+1$, then

$$\text{Ranking}_A = \sum_{i=1}^{k+1} (x * 10^{(1-i)}) \geq \text{Ranking}_B = \sum_{i=1}^{k+1} (y * 10^{(1-i)})$$

The term $10^{(1-i)}$ gets cancelled out on both sides, and since the top of the tree has a larger emphasis, only the weights are of importance, so the equation equals: $x \geq y$, which is true.

5) Therefore, since the base case holds true and the inductive step holds true, the inductive hypothesis is true for all n .

Figure 27. Inductive proof #1: Semantic spanning trees within a sub-tree.

Conjecture: the ranking of semantic spanning tree T_A is greater than or equaled to the ranking of semantic spanning tree T_B for a semantic tree.

1) Assume that there are two semantic spanning trees with rankings. Let the weight in spanning tree $T_A = x$, and the weight in spanning tree $T_B = y$, where $x \geq y$. Then

$$\text{Ranking}_A = \sum_{i=1}^n \left(\sum_{j=1}^m (x * 10^{(1-j)}) \right)_i \geq \text{Ranking}_B = \sum_{i=1}^n \left(\sum_{j=1}^m (y * 10^{(1-j)}) \right)_i$$

2) Base Case: If $n = 1$, then

$$\text{Ranking}_A = \sum_{i=1}^1 \left(\sum_{j=1}^m (x * 10^{(1-j)}) \right)_i \geq \text{Ranking}_B = \sum_{i=1}^1 \left(\sum_{j=1}^m (y * 10^{(1-j)}) \right)_i$$

Since the weights are the only important factors that separate the two rankings, the equation is left with: $x \geq y$, which holds true from step #1.

3) Inductive Hypothesis: If $n = k$, then

$$\text{Ranking}_A = \sum_{i=1}^k \left(\sum_{j=1}^m (x * 10^{(1-j)}) \right)_i \geq \text{Ranking}_B = \sum_{i=1}^k \left(\sum_{j=1}^m (y * 10^{(1-j)}) \right)_i$$

Since the top of the tree has a heavier impact on the weights, the only part of the equation that matters is the weight, so the equation equals to: $x \geq y$, which is true.

4) Inductive Step: If $n = k+1$, then

$$\text{Ranking}_A = \sum_{i=1}^{k+1} \left(\sum_{j=1}^m (x * 10^{(1-j)}) \right)_i \geq \text{Ranking}_B = \sum_{i=1}^{k+1} \left(\sum_{j=1}^m (y * 10^{(1-j)}) \right)_i$$

Again, since the top of the tree has a larger emphasis, only the weights are of importance, so the equation equals: $x \geq y$, which is true.

5) Therefore, since the base case holds true and the inductive step holds true, the inductive hypothesis is true for all n .

Figure 28. Inductive proof #2: Semantic spanning trees within a semantic tree.

Ranking for the SEC and SIC models. The ranking system in the SEC and SIC models can be used to create a cost-effective model. To calculate the ranking for each SEC and SIC semantic spanning trees, the ranking at each leaf node must be computed first by using Equation 4 below. This equation is derived by using the powers of 10 because it represents the importance of the weights by tree level. The weight at the top levels of the tree has more significance than the weight at the bottom. After finding the

ranking for each leaf node, the ranking for each spanning tree can be calculated by adding all its leaf node rankings, as shown in Equation 5. The sequence to implement the computer setting combinations is prioritized from the highest spanning tree's ranking to the lowest. In the event of a tie where more than one spanning tree has the same ranking, the spanning tree with the higher first level ranking has precedence.

$$\text{Ranking}_i = \sum_{i=1}^n (w_i * 10^{(1-i)}), \text{ where } w_i = \text{weight of each link} \quad (\text{Equation 4})$$

$$\text{TotalRanking}_i = \sum_{i=1}^n (\text{Ranking}_i) \quad (\text{Equation 5})$$

Assume that the weights are assigned subjectively according to its priority. The weights are in the form of a decimal percentage, where the sum of the weights of the sibling nodes equals 1. To compute the ranking on this tree, each leaf node needs to be calculated first using Equation 4, as demonstrated in Figure 29. The total ranking for each spanning tree equals the summation of the leaf node rankings using Equation 5.

In the SEC model, by prioritizing the configuration settings, the software gets installed on the most important configurations first for testing. The sequence for configuration testing depends on the total ranking for each spanning tree. Figure 29 shows the ranking calculations for each leaf node by using Equation 4. As an example to find the ranking for the spanning tree with the configuration of Windows XP Home SP2, first take the sum of the leaf node rankings, which equals 0.863. Since this is the only leaf node for this spanning tree, the total ranking for this spanning tree equals 0.863. Since some configuration rankings are tied in the Windows XP sub-tree, the general rule

is to pick the order of configurations in the direction from left to right. In this example, the highest spanning tree ranking is 0.863, which belongs to Windows XP Professional SP2 and Home SP2, and the lowest ranking is 0.2, which goes to the not-configured OS. The spanning trees are prioritized in the following sequence: Windows XP Professional SP2, Windows XP Home SP2, Windows XP Professional SP1, Windows XP Home SP1, Windows 2000 Professional SP4, Windows 2000 Professional, and the not-configured OS. Figure 30 illustrates the first three spanning trees with their ranking calculations.

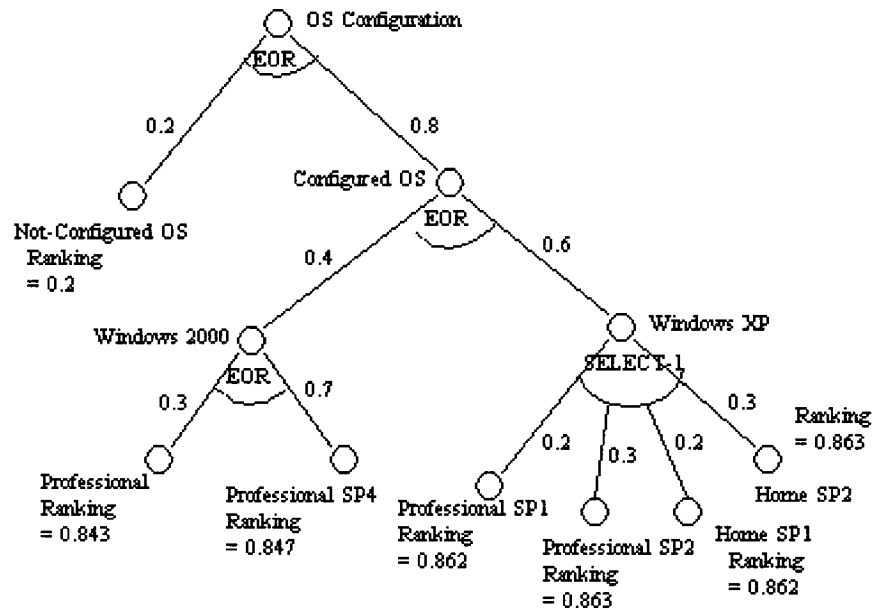


Figure 29. SEC semantic tree weighted graph with rankings.

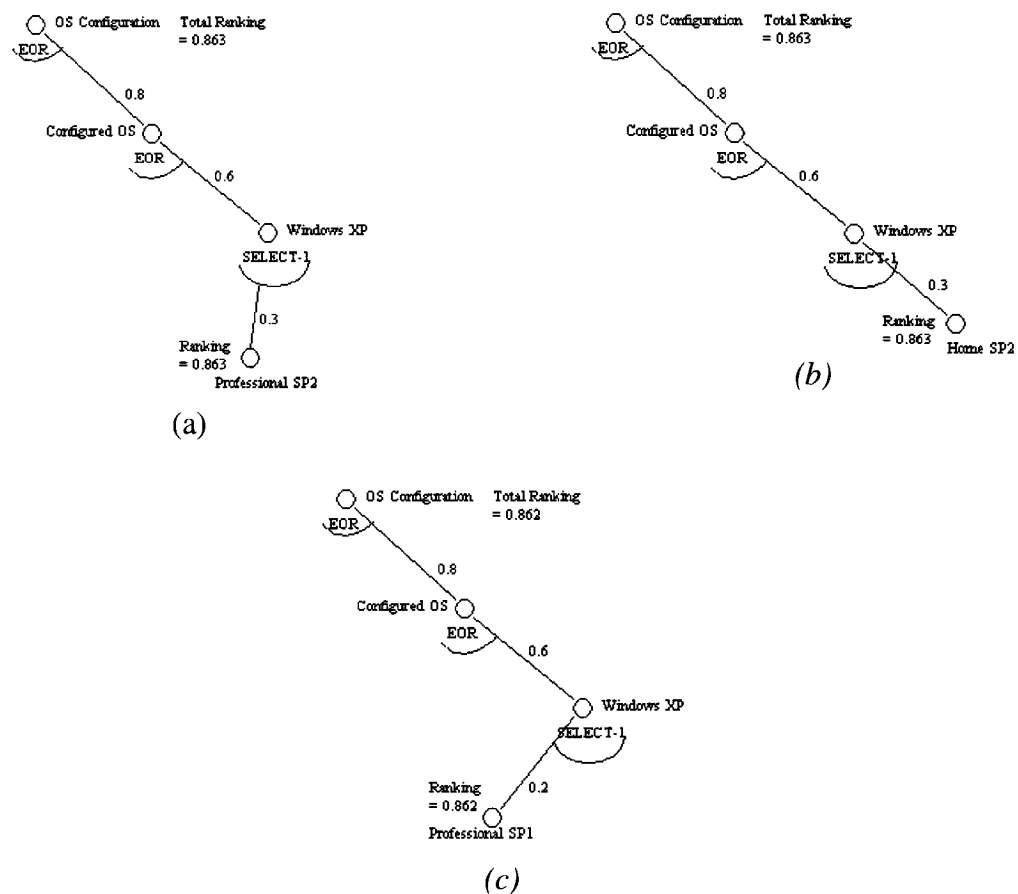


Figure 30. SEC ranking semantic spanning trees.

Like the SEC model, the ranking for the SIC model is calculated for each spanning tree using Equations 4 and 5. Figure 31 displays the ranking computations for the leaf nodes. As an example, for the system running condition with Admin User Access (ranking = 0.35), Software Opened (ranking = 0.387), and Installed the Software Once (ranking = 0.454), the total ranking for this spanning tree equals $0.35 + 0.387 + 0.454 = 1.191$. Figure 32 illustrates an example of the first three spanning trees with ranking calculations for the system running condition semantic tree.

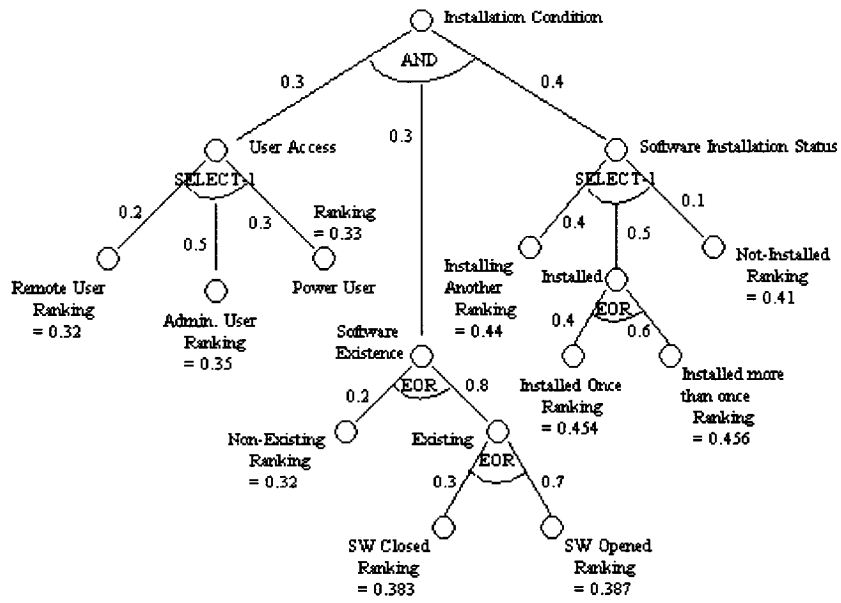


Figure 31. SIC semantic tree weighted graph with rankings.

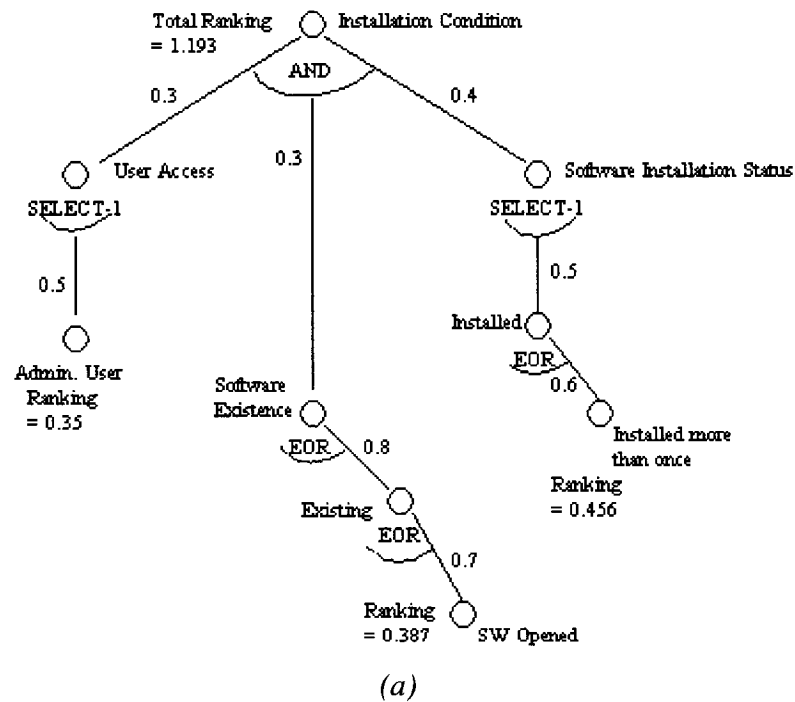


Figure 32. SIC ranking semantic spanning trees.

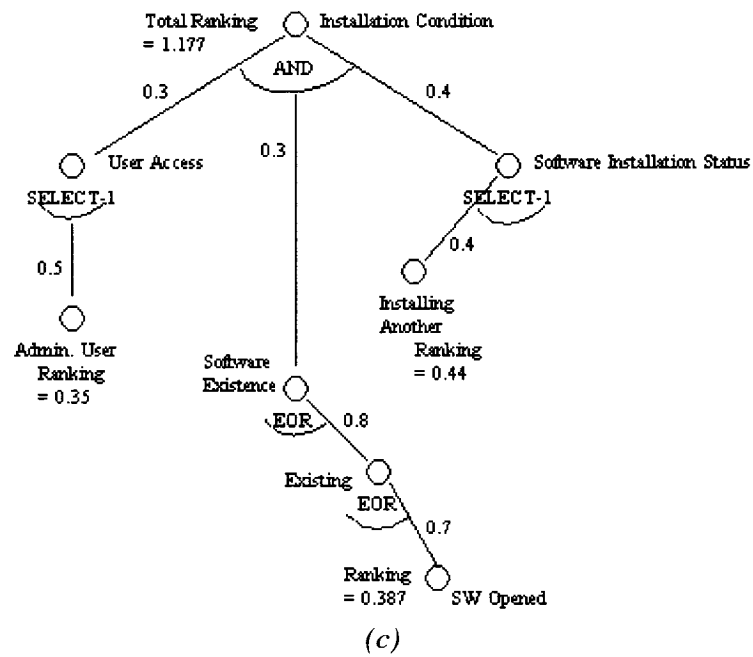
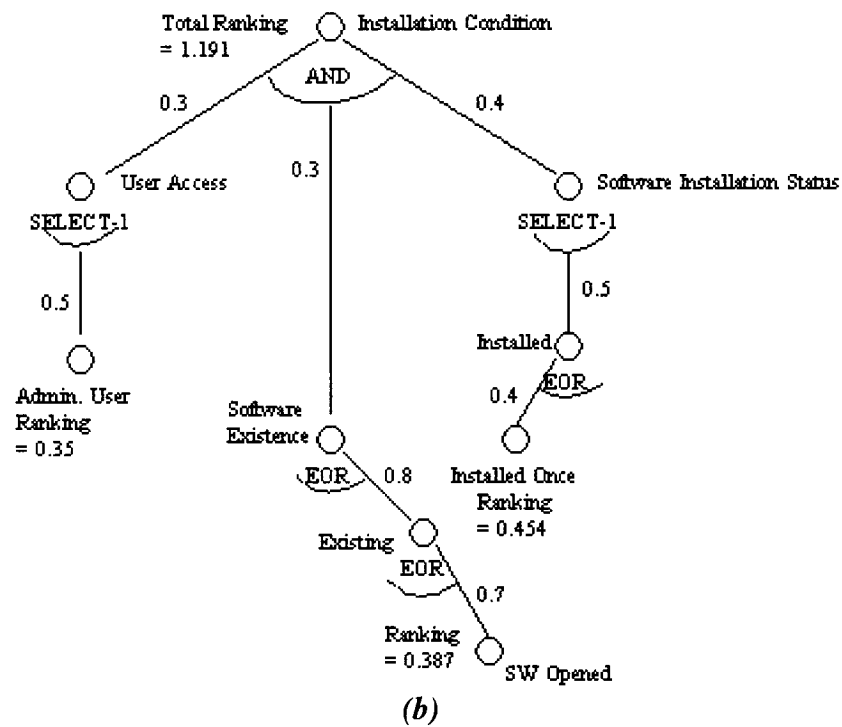


Figure 32. SIC ranking semantic spanning trees (continued).

Ranking for SIF model. For the SIF model, the ranking is computed at the leaf node level only. Because there is only one spanning tree for this model, prioritizing by spanning trees does not make sense. Since only the functions need to be tested and these functions are located at the leaf nodes, the ranking at the leaf node level can be used to prioritize the sequence of the functions to be tested. The ranking for the SIF semantic tree can be computed by using Equation 4. In the case of a tie where more than one function has the same ranking, the order to run the functions goes from left to right according to where the functions are positioned in the tree with that ranking.

Assume the weights on each link are given and the ranking is computed with Equation 4. As an example, the ranking for the leaf node “Remove Files After Uninstalling” equals 0.23. After the ranking has been calculated for each leaf node, the test cases for each function gets rearranged to prioritize them from highest to lowest rankings. In this example, the sequence to execute the test cases is Launch Application, Reboot Machine, Remove Registration Entries, Remove Files, Remove Icons, Remove Items from Start Menu, Network Connection Setup, Registered User, New User Registration, Log File Generation, Clean Temp Directory, Error Handling, and EULA-Printing. Figure 33 demonstrates the ranking calculations for each function in the SIF model.

the number of test cases already executed is 24. So the test coverage equals $24 \div 36 = 0.667$, which means that the test bed has a test coverage of about 67%.

Test metrics. Test metrics provide important feedback that measures the present performance, tracks the on-going progress, and indicates the effectiveness of the testing (Pusala, 2006, p. 2). These test metrics can help prioritize the testing and offer suggestions to improve the testing process. This research will demonstrate two ways of calculating the test metrics in terms of cost in hours.

Test cost estimation method #1. The first method for computing the cost consists of calculating the test complexity and then calculating the cost at the root node. This method is used mainly for the SEC and SIC models to calculate its cost for each spanning tree. There are three types of test cost that factor into the expenses, which is measured in terms of time, such as hours. These test cost types will serve as the cost constants S , R , and D_{RPT} for the setup cost, running cost, and defect reporting cost, respectively. Using these constants and incorporating them into the semantic test complexity Equations 1 to 3, the cost can be estimated for a semantic tree in Equations 7 and 8 below. For the semantics AND, EOR, and SELECT-1, which uses Equation 7, the test metric equation consists of an overall setup cost for the whole entity, a running cost for each test case, and a defect reporting cost for each test case. For the semantics NOT and NAND, which uses Equation 8, the grand total cost equals zero because the functionalities with these semantic relations do not need to be implemented, but instead are shown to display the current components featured and the components that got removed from the latest version

of the software. Table 4 displays the breakdown of each test cost type and how each constant factor came about.

$$\begin{array}{l} \text{AND} \\ \text{EOR} \\ \text{SELECT-1} \end{array} \quad M_{\text{Cost}} = S + R * \left(\sum_{i=1}^n C_i \right) + D_{\text{RPT}} * \left(\sum_{i=1}^n C_i \right) \quad (\text{Equation 7})$$

$$\begin{array}{l} \text{NOT} \\ \text{NAND} \end{array} \quad M_{\text{Cost}} = 0 \quad (\text{Equation 8})$$

Table 4

Test Cost Estimation Chart

Types of Test Metrics	Categories	Cost (in hours)	Total Cost (in hours)
setup costs (S)	gathering requirements/specs	---	---
	test case design/development	---	
	data creation	---	
running costs (R)	test case execution	---	---
	test case analysis	---	
	test case reporting	---	
defect reporting (D_{RPT})	failure rate	---	---
	severity of defects	---	
	defect documentation	---	

For example in the SEC model, the only factors to incorporate into the test metrics are the setup costs and the defect reporting costs because the configuration settings only needs to be set up and not executed. If it did not get installed successfully, then the defect reporting cost needs to be calculated as well. So the constant S equals 0.2 and constant D_{RPT} equals 0.2. To calculate the test cost, it requires the test complexity to be computed for each spanning tree first because the total number of test cases affects the cost. Assume that a test complexity is assigned to each leaf node. After the test complexity has been determined for the spanning tree, then Equation 7 helps calculate the

test cost for the spanning tree. For example, in Figure 34a, the test complexity for the spanning tree equals 5, so the test cost equals 2. These calculations are performed for each spanning tree in the SEC model. The first three spanning trees with its test cost calculations are illustrated in Figure 34. The overall test cost estimation for this SEC model equals 10 hours to setup and analyze the configuration settings.

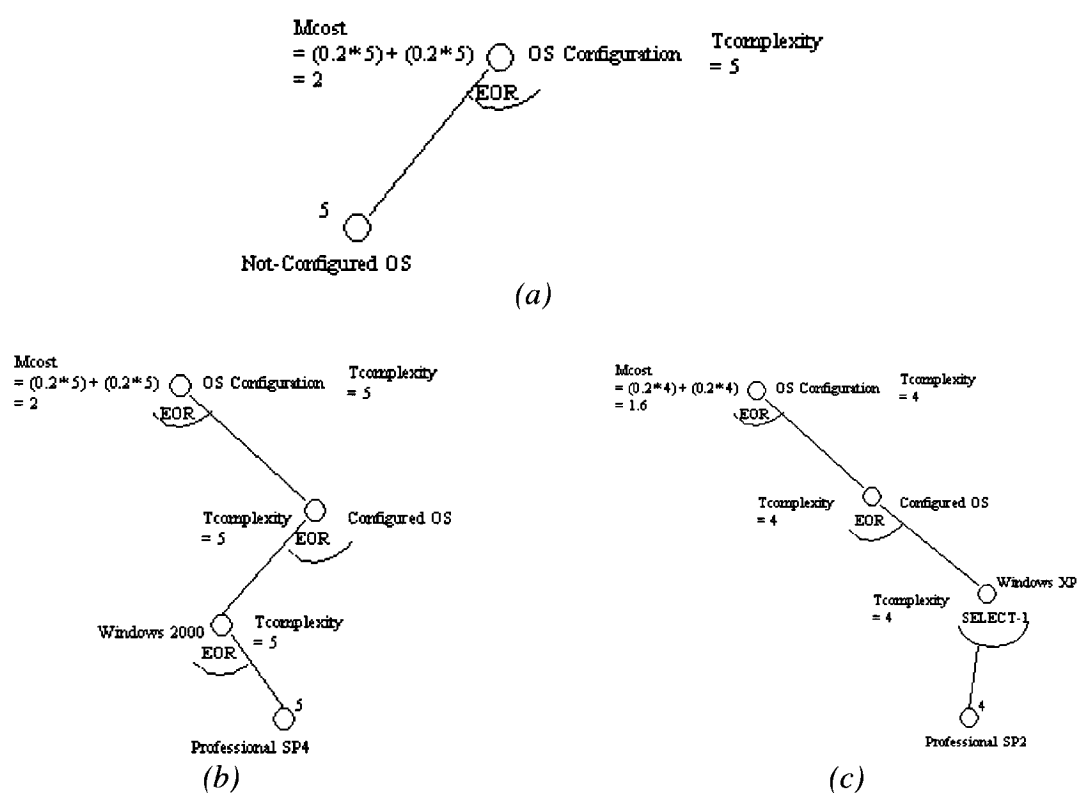


Figure 34. SEC semantic spanning trees with test cost estimation method #1.

The same test cost strategy for the SIC model can be used to evaluate the current testing performance. The constants for the test cost equations for the SIC model have the following values: $S = 0.3$ and $D_{RPT} = 0.2$. Given the test complexities for each leaf node,

the test cost is calculated by first computing all the test complexities for each parent node in each spanning tree. Equation 7 is applied to the root node to calculate the test cost for each spanning tree. Figure 35 illustrates the first three spanning trees with its ranking calculations. The total cost for the entire SIC model equals the sum of the costs of the spanning trees. It will take 153 hours to set up the running conditions and to detect and report the defects for this tree.

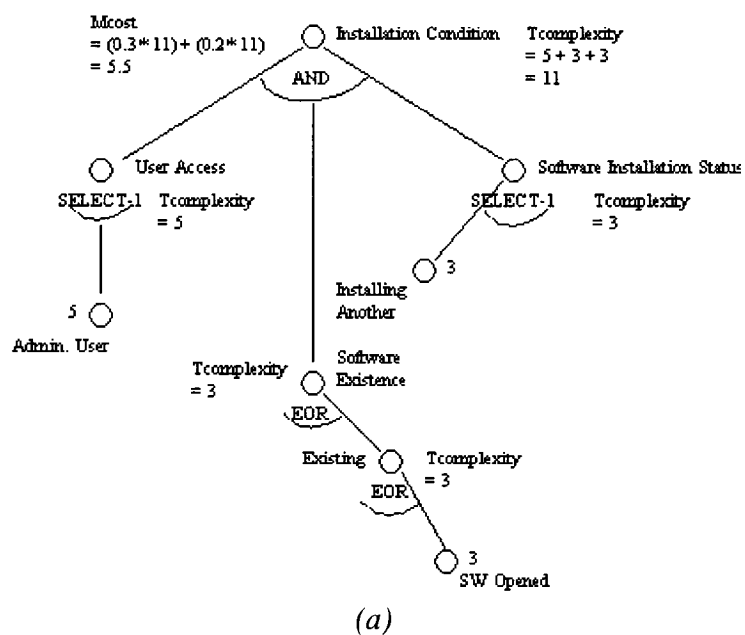


Figure 35. SIC semantic spanning trees with test cost estimation method #1.

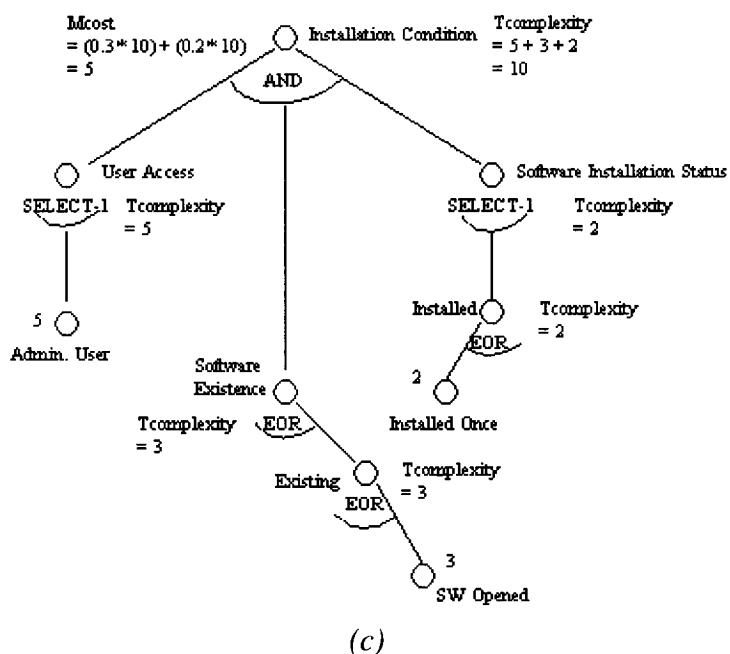
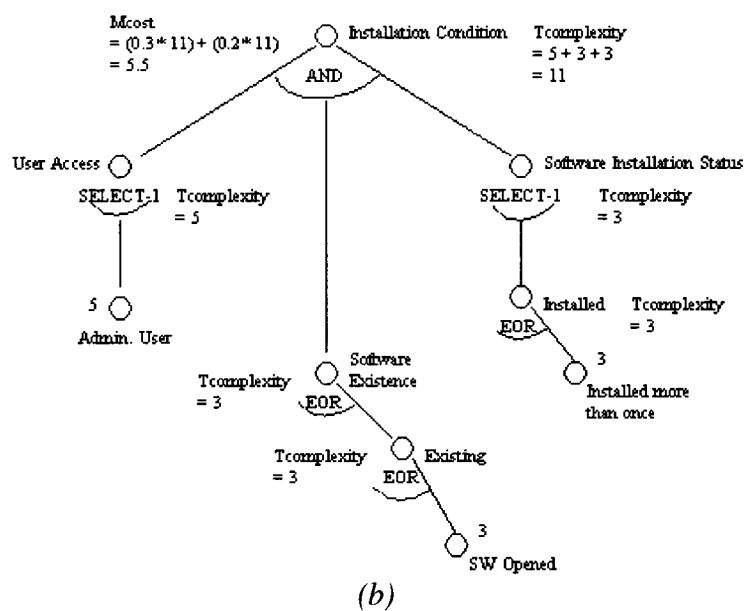


Figure 35. SIC semantic spanning trees with test cost estimation method #1 (continued).

Test cost estimation method #2. Similar to the first method, the second way of calculating the test cost estimation involves computing the cost for each leaf node by using Equation 7. To get the total cost for the whole tree, add all of these costs together.

Again, the cost is measured in terms of hours. The difference between method #1 and method #2 is that the cost equation is applied at different nodes; method #1 calculates the cost at the root node, while method #2 computes the cost at the leaf nodes. The second method is only used for the SIF model to calculate its cost because in general, it has no spanning trees. Method #2 is a simple way to calculate the cost for each software feature on the semantic tree.

For the SIF model, the second test cost method computes the cost at the leaf node level, then all these leaf node costs are added together to equal the total cost. Figure 36 illustrates the test cost estimations for the SIF model. The total cost for the installation function semantic tree equals 48 hours to setup test cases, to execute test cases, and to detect and report defects.

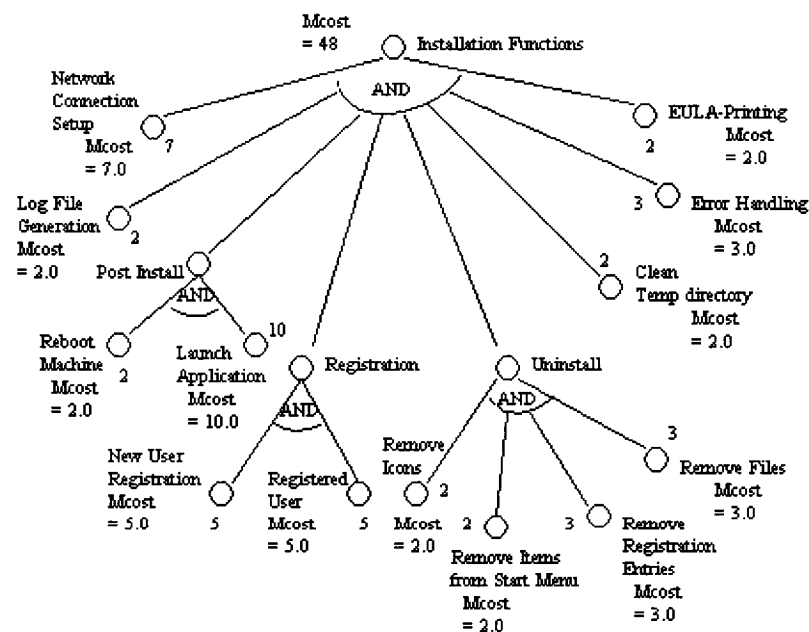


Figure 36. SIF semantic tree with test cost estimations method #2.

Alternative method for estimating test cost. Another way to look at estimating the cost of the models is to utilize Equation 9. This equation calculates the overall cost of the entire model, which consists of the SEC, SIC, and SIF models. T stands for the test complexity of each node and S represents the semantic tree model's cost, which includes the setup cost, the running cost, and the defect reporting cost. For each of the three models, the cost equals the sum of the node's test complexity T multiplied by the constant cost S. By computing the cost of the entire model, this figure provides a tangible estimate to plan the testing effort and can be used for comparison purposes for testing future cost saving testing methods.

$$\text{cost}_{\text{model}} = \sum (T_i * S_{\text{SEC}}) + \sum (T_j * S_{\text{SIC}}) + \sum (T_k * S_{\text{SIF}}) \quad (\text{Equation 9})$$

Chapter 4

Software Installation Testing Results

Treatment of the Data

Model for turbo tax software installation testing. This research will implement this new model a little differently; it will be applied to an application case study. The software that will be used in this case study is one of Intuit's products, Turbo Tax, the end user version. Turbo Tax is a software product that assists users in calculating their income taxes. Using this software, the SEC, SIC, and SIF models will be designed accordingly. In addition, the algorithm and formulas derived in the methodology section will demonstrate the generation of the semantic spanning trees, both test complexity calculations, and the test cost estimations for the software installation testing of Turbo Tax. Finally, the analysis will be examined and conclusions will be drawn. Figures 37 to 39 show the semantic trees for the environment configuration, system running condition, and the installation function for the Turbo Tax software.

Within each of these semantic trees, spanning semantic trees exist, where each spanning tree represents a set of computer settings to test the software installation. The idea of having spanning trees is important because this indicates the number of possible combinations in setting up a computer. Software must be installed onto a computer with each of these combination settings tested thoroughly because it is difficult to predict the

type of computers that the consumers may own. In order to make the software more marketable, the software should be compatible on at least the most popular computer configurations and running conditions. Thus, the spanning tree generation algorithm only applies to the environment configuration and system running condition semantic trees.

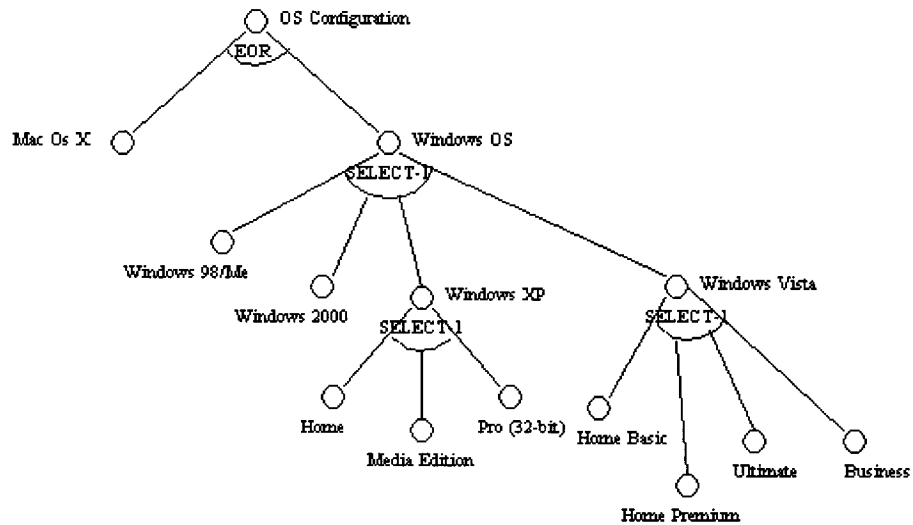


Figure 37. Turbo Tax's SEC semantic tree.

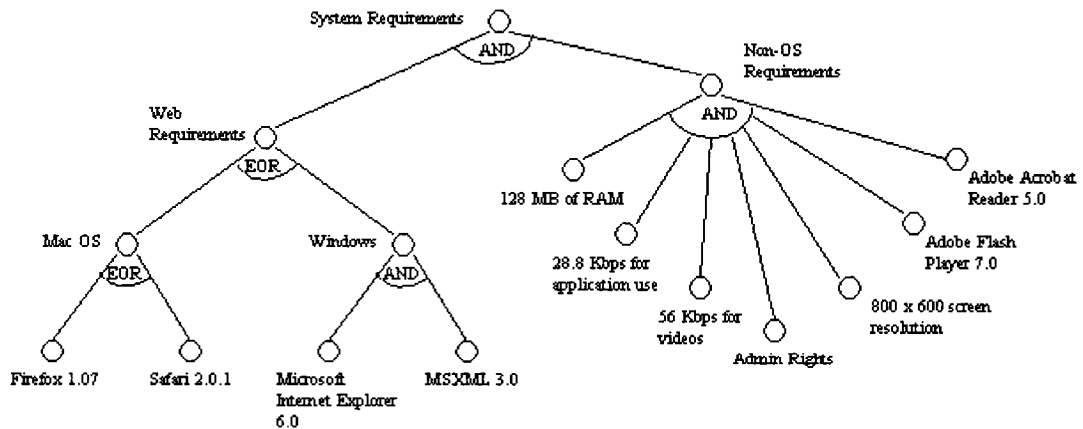


Figure 38. Turbo Tax's SIC semantic tree.

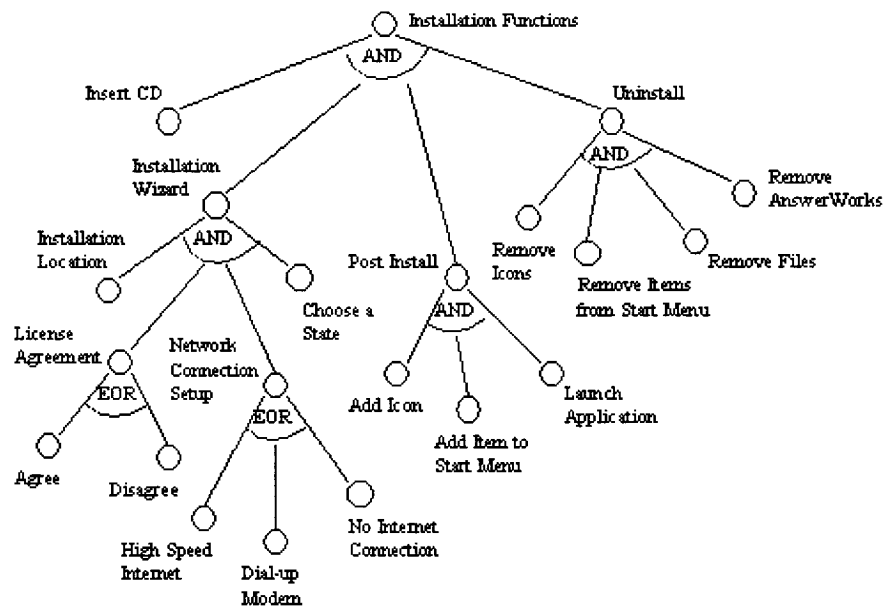


Figure 39. Turbo Tax's SIF semantic tree.

Two different algorithms exist for generating the spanning trees: semantic spanning tree and weighted semantic spanning tree. The differences between the two algorithms include the algorithm's input and the selection process for a child node. The semantic spanning tree inputs a semantic tree with no weights and chooses the child node based on the position of the child node in the tree, from left to right. The weighted semantic spanning tree requires a semantic tree with weights and selects the child node with the largest weight. By following the two spanning tree algorithms in chapter 3, the semantic spanning trees and weighted semantic spanning trees are generated for Turbo Tax's SEC and SIC models. Examples of these semantic spanning trees can be seen in Appendices A and B, which show that the SEC model generates ten semantic spanning trees while the SIC model produces three semantic spanning trees.

Measurement and test analysis for turbo tax. The Turbo Tax models above are created according to the end user version product. To measure and analyze the testing for Turbo Tax, this research uses simulated data to examine the test complexities, the ranking system, and the test cost estimations. Test complexity presents the minimum number of test cases required for adequate test coverage, also known as the semantic test complexity. It also determines the number of spanning trees that can be generated from a given semantic tree, which is known as the spanning tree test complexity. Both types of test complexities for Turbo Tax's three models are presented in Appendices A, B, and C.

Turbo Tax's SEC model has semantic test complexities ranging from 4 to 10 as shown in Table 5 below for each of its spanning trees. The design of this SEC semantic tree turns out to have only one leaf node in each spanning tree, so by summing up these semantic test complexities, the total semantic test complexity for the SEC model equals 62. Additionally, the overall spanning tree test complexity for Turbo Tax's SEC model equals 10, where each spanning tree has a spanning tree test complexity of 1.

Table 5

Test Complexity for Turbo Tax's SEC Model

weighted spanning tree #	1	2	3	4	5	6	7	8	9	10	Overall
semantic test complexity	9	5	7	6	5	4	7	5	4	10	62
spanning tree test complexity	1	1	1	1	1	1	1	1	1	1	10

Turbo Tax's SIC model has semantic test complexities from the range of 18 to 23 as displayed in Table 6 below for each of its spanning trees. By applying the semantic

test complexity equations, the total semantic test complexity for Turbo Tax's SIC model equals 28. In addition, the overall spanning tree test complexity for Turbo Tax's SIC model equals 3, where each spanning tree has a spanning tree test complexity of 1.

Table 6

Test Complexity for Turbo Tax's SIC Model

weighted spanning tree #	1	2	3	Overall
semantic test complexity	23	19	18	28
spanning tree test complexity	1	1	1	3

To enhance the semantic tree model, the weighted semantic tree was designed, which produced the idea of having rankings. The main reason for having a ranking system lies in prioritization. Ordering the spanning trees and functions from highest importance to lowest importance comes in handy when testers find themselves in a tight timeframe to complete and sign off on testing. For the SEC and SIC models, ranking can be used on the spanning trees so that the most critical configurations and conditions can be tested first. The SIF model uses the ranking system to prioritize the functions at the leaf node level.

In the Turbo Tax application example, the ranking for the SEC model ranges from 0.3 to 0.746 and the SIC model has rankings from 4.734 to 5.25, as illustrated in Tables 7 and 8. For the SIF model, since spanning trees do not exist, the ranking is for each installation function. According to Table 9, the highest ranking is 0.445 and the lowest ranking is 0.10. The average between these 15 leaf node rankings is 0.336 and the

median equals 0.36. Appendices A, B, and C demonstrate some examples of the ranking calculations for the SEC, SIC, and SIF models for Turbo Tax.

Table 7

Ranking for Turbo Tax's SEC Model

weighted spanning tree #	1	2	3	4	5	6	7	8	9	10
ranking	0.746	0.742	0.742	0.733	0.733	0.732	0.732	0.72	0.71	0.3

Table 8

Ranking for Turbo Tax's SIC Model

weighted spanning tree #	1	2	3
ranking	5.25	4.736	4.734

Table 9

Ranking for Turbo Tax's SIF Model

	Max	Min	Average	Median
ranking	0.445	0.10	0.336	0.36

To analyze the Turbo Tax model further, this research takes the test cost metrics into account. These metrics provide feedback on the testing performance and indicates the effectiveness of the testing process. In this study, two approaches for estimating the test cost are presented. The two methods differ in the node to which the test cost equation is applied, such as method #1 calculates the cost at the root node level for the

SEC and SIC models, while method #2 computes the cost at the leaf node level for the SIF model. The test cost equation contains the setup cost, running cost, and defect reporting cost.

The SEC model only utilizes the setup cost and the defect reporting cost constants because this model only sets up the computer's settings and no test execution is necessary. In addition, the defect reporting gets factored in for cases where defects are detected in the settings. The setup cost (S) equals 0.25 hours and the defect reporting cost (D_{RPT}) equals 0.2 hours. Appendix A illustrates the computation of the test complexity for each intermediate node and the calculation of the test cost at the root node for the first three spanning trees. The overall test cost using method #1 comes out to 27.9 hours to setup and analyze all the settings in Turbo Tax's SEC model, as displayed in Table 10.

Table 10

Test Cost Metrics for Turbo Tax's SEC Model

weighted spanning tree #	1	2	3	4	5	6	7	8	9	10	Overall
test cost method #1 (h)	4.05	2.25	3.15	2.7	2.25	1.8	3.15	2.25	1.8	4.5	27.9

Similarly, the SIC model acts in the same manner as the SEC model to estimate the test cost with the exception of using different values for the constants. The constants S equals 0.9 hours to setup all the conditions and D_{RPT} equals 0.2 hours to analyze and detect any defects. The total test cost for Turbo Tax's SIC model calculates 66 hours

using method #1, as shown in Table 11 below. Appendix B presents the three spanning trees and its test cost estimations for Turbo Tax's SIC model.

Table 11

Test Cost Metrics for Turbo Tax's SIC Model

weighted spanning tree	1	2	3	Overall
test cost method #1 (h)	25.3	20.9	19.8	66

The test cost for Turbo Tax's SIF model is computed using method #2. This model includes all three constants, S, R, and D_{RPT} because all the software installation function test cases are stored in this tree. This means that these test cases must be setup, executed, and analyzed, and the defects must be caught and reported. For the SIF model, the constants S equals 0.4 hours to setup all the test cases, R equals 0.3 hours to execute each test case, and D_{RPT} equals 0.2 hours to detect and report any defects that may occur. In Appendix C, the calculations show that test costs for Turbo Tax's SIF semantic tree equals 35.1 hours using method #2, as shown in Table 12 below.

Table 12

Test Cost Metrics for Turbo Tax's SIF Model

weighted spanning tree	1	2	3	4	5	6	Overall
test cost method #2 (h)	28.8	28.8	27.9	27.9	27	27	35.1

Data Analysis

Now that the algorithm and formulas have been applied to an application case study on Turbo Tax software to prove that they work, let's take a closer look at the results. Table 13 below sums up the data results for the Turbo Tax case study by categorizing the measurement analysis, such as the number of nodes used and the number of leaf nodes in each tree. It also includes the semantic and spanning tree test complexities, the ranking, and the test cost estimations for each spanning tree in all three models and the overall results for each of the three models.

Table 13 presents the measurement results for each spanning tree in each model along with the total results for each model labeled as SEC tree, SIC tree, and SIF tree. Figure 40 displays a graph of the total data results for each of the three models, while Figure 41 shows a graph of all the spanning trees between the three models. All the line curves on both graphs show similarities in shape. From the data table and the graphs, four trends can be seen as diagrammed in Figure 42 and as outlined in the following:

1. weighted spanning tree number corresponds to the ranking
2. \uparrow semantic test complexity = \uparrow test cost estimation (using methods 1 and 2)
3. \uparrow semantic test complexity = \uparrow ranking
4. \uparrow ranking = \uparrow test cost estimation (using methods 1 and 2)

Table 13

Summary of Results for the Turbo Tax Software

Model	Weighted Spanning Tree #	# Nodes	# Leaf Nodes	Semantic Test Complexity	Spanning Tree Test Complexity	Ranking	Test Cost
SEC	1	4	1	9	1	0.746	4.05
	2	4	1	5	1	0.742	2.25
	3	4	1	7	1	0.742	3.15
	4	4	1	6	1	0.733	2.7
	5	4	1	5	1	0.733	2.25
	6	4	1	4	1	0.732	1.8
	7	4	1	7	1	0.732	3.15
	8	3	1	5	1	0.72	2.25
	9	3	1	4	1	0.71	1.8
	10	2	1	10	1	0.3	4.5
	SEC tree	14	10	62	10	0.3 - 0.746	27.9
SIC	1	13	9	23	1	5.25	25.3
	2	12	8	19	1	4.736	20.9
	3	12	8	18	1	4.734	19.8
	SIC tree	16	11	28	3	4.734 - 5.25	66
SIF	1	18	12	32	--	--	28.8
	2	18	12	32	--	--	28.8
	3	18	12	31	--	--	27.9
	4	18	12	31	--	--	27.9
	5	18	12	30	--	--	27
	6	18	12	30	--	--	27
	SIF tree	21	15	39	--	--	35.1

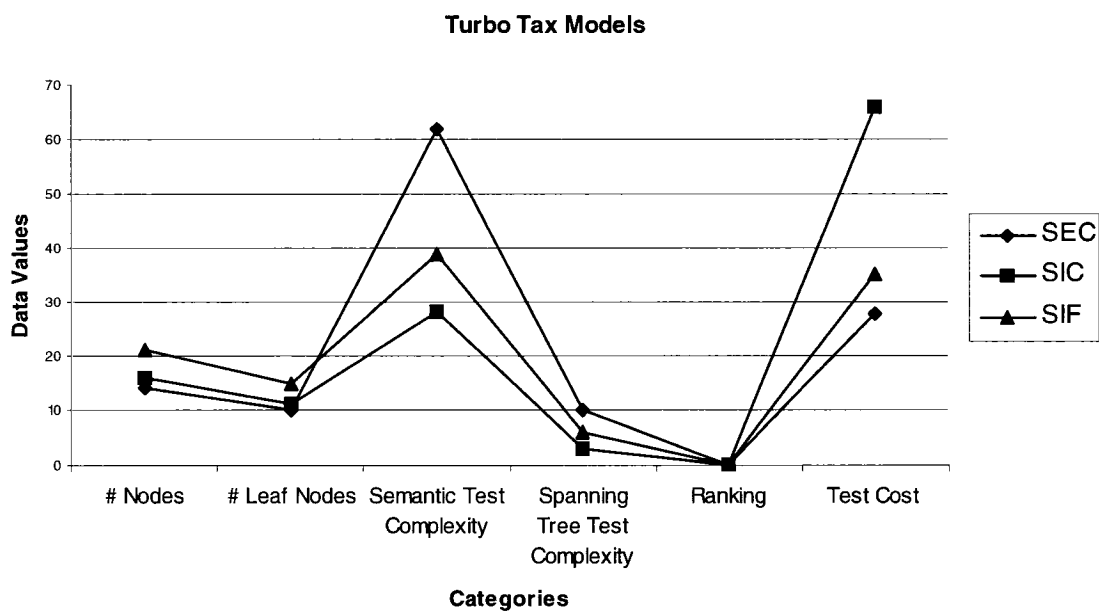


Figure 40. Graph of Turbo Tax's models' data.

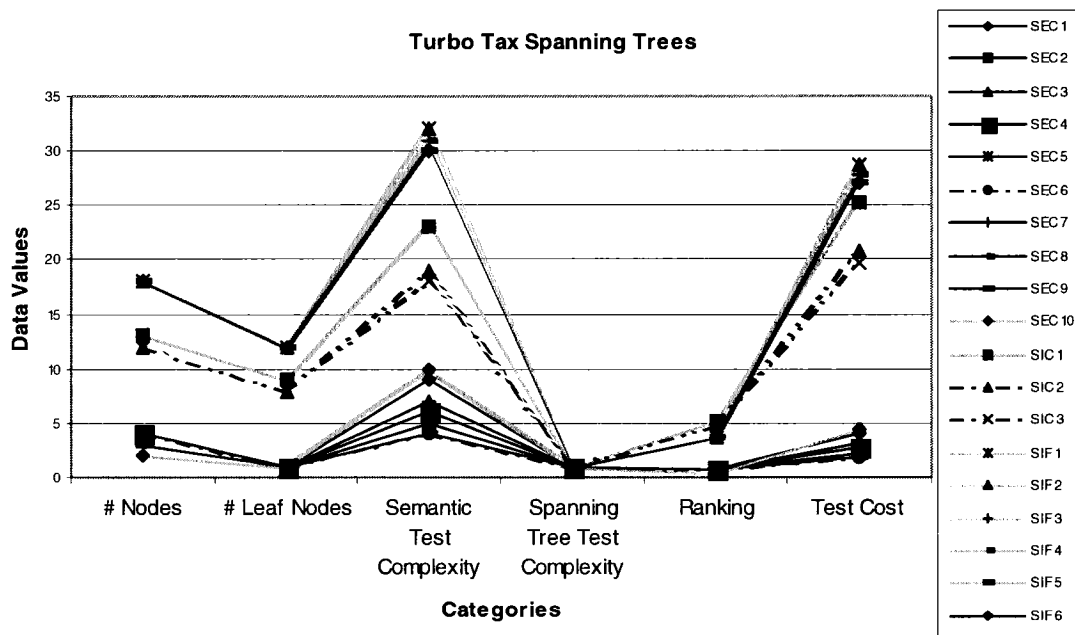


Figure 41. Graph of Turbo Tax's spanning trees' data.

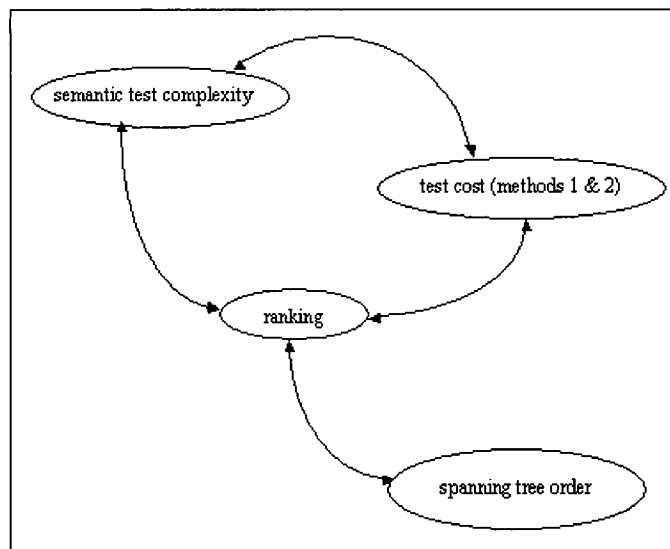


Figure 42. Diagram of the relations between the measurements in Table 13.

Trend #1 examines that the order of the weighted spanning tree relies on the ranking, therefore, a correlation between them exists. In order to obtain the sequence of the spanning trees, a weighted system is used to rank the importance of each spanning tree. Since the ranking plays a major role in determining the priority, the sequence of testing the spanning trees corresponds to the ranking.

Trend #2 shows that a direct proportional relationship between the semantic test complexity and the test cost exists, where a high semantic test complexity equals a high testing cost. This makes sense because the test cost estimation depends on the semantic test complexity since it costs more to perform testing if the test bed holds more test cases.

The next two trends only can be seen in Turbo Tax's SIC and SIF models. The following two trends are not evident in Turbo Tax's SEC model because it only contains one leaf node for each spanning tree, while the other two models have multiple leaf nodes for each of their spanning trees. This behavior makes a difference because it changes the

dynamics of the spanning tree and the amount of testing that needs to be done. Assuming that if the SEC model had more leaf nodes in its spanning trees, it is projected that it will also display the following trends.

Trend #3 looks at the relationship between the semantic test complexity and the ranking. A direct proportional relationship also exists here. If the spanning tree has a high ranking, this means that it is critical to test this scenario because it requires more test cases to test the specifications.

Likewise, trend #4 also displays that the ranking and the test cost estimations have a direct proportional relationship. Since the semantic test complexity has a direct proportional relationship with both the ranking and the test cost, by transitivity, the ranking must be directly proportional to the test cost. If the spanning tree has a high ranking, then more test cases need to be executed, which means that it costs more to conduct the testing.

The overall cost for all three models combined equals $27.9 + 66 + 69.3 = 163.2$ hours. This cost analysis shows that the weighted semantic model has an inexpensive cost compared to ad hoc testing. In ad hoc testing, the cost is difficult to estimate because the testing is endless since test cases need to be created each time a specific problem arises. On the other hand, the testing cost can be estimated for the weighted semantic model using the test cost estimation formula in this research. Knowing the cost is important because the appropriate funding can then be allotted for testing, which in turn, can produce products with better quality.

Chapter 5

Summary, Conclusions, and Recommendations

Summary

As the software industry grows and as computer technologies keep on advancing, software will eventually be used in almost all aspects of everyone's lives, either directly or indirectly. But software installation testing is quite complicated and is an area of research that has not been developing much, and thus, is in need of advancement. As a solution to this need, this research successfully develops a new model to validate software installation testing in a more efficient and cost-effective way.

In an effort to create a new model to perform software installation testing, this research proposed a semantic tree model, which is composed of the environment configuration model, the system condition model, and the installation function model. The configuration and condition models lay the foundation of the computer settings for which the software must be tested on. After installing the software, its features are tested by using the installation function model. Based on the test criteria and pass/fail rate, the success of the software installation can be determined. In addition, this research introduced a weighted semantic tree model "to assist engineers to identify a cost-effective test strategy using weighted validation test sequences" (Gao, Kwok, & Fitch, 2007, p. 1). This weighted semantic tree model "provides a cost-effective method

for installation test planning to focus on important or most frequently used system configurations and running conditions first” (Gao, Kwok, & Fitch, 2007, p. 10).

This research also addressed test analysis to assist the models during the software installation testing process. The test analysis included: the test complexity computations, an algorithm to generate the spanning trees, a ranking system, and test cost estimations. The semantic test complexity calculated the number of test cases required to provide adequate test coverage based on the semantic relationship of each parent node. While the spanning tree test complexity computed the number of possible spanning trees. To generate these spanning trees, an algorithm was designed. The ranking system enhanced the model by prioritizing the test cases so that the critical tests would execute first. The test cost estimations kept track of the progress and monitored the cost of the testing process. The two newly proposed models and the test analysis were implemented and validated in a case study using Turbo Tax software.

Conclusions

Although software installation testing is complicated, expensive, and time consuming, it is all worth it in the end. If the defects can be detected and fixed to enhance the quality of the software, then it is meeting the ultimate goal of increasing the satisfaction of the customers by meeting their expectations. In this case, software installation testing makes a big difference in improving software quality because one of the first steps to using software is to install it properly. Thus, it is important to establish

an efficient method to test software installation. This research offers a semantic tree model and a weighted semantic tree model to perform software installation testing.

In addition, some test analyses were examined to complement the models. Calculations were made for the number of spanning trees, test complexity, ranking, and test cost estimation. The results demonstrate that the weighted spanning tree sequence corresponds to the ranking sequence. Also, a direct proportional relationship exists between the ranking, the semantic test complexity, and the test cost. For example, a high-ranking specification indicates that more test cases are required to adequately test it, meaning that its semantic test complexity will be high, which means that it will cost more to test. The test analysis assisted the models in making it more efficient and cost-effective by providing an estimation of the number of test cases needed to adequately test each requirement and offering a ranking system to prioritize the testing. Thus, this leads to being able to estimate the cost of the model. Since the models in this research can put a price tag on its testing, the newly proposed models cost less to conduct testing than using ad hoc testing. It is difficult to estimate cost for ad hoc testing since there is no real end to its testing.

The semantic tree model laid a good foundation for the weighted semantic tree model to build on because the weighted semantic tree model turned an efficient model into a cost-effective model. In the long run, the weighted semantic tree model reduces the cost to execute testing because the most critical test cases testing the most commonly used functions of the software will always run first. By prioritizing test cases in this way,

there is a higher chance that the critical defects can be caught earlier. Because of this, it will cost less to fix, and thus, improve the software.

Recommendations for Further Research

For future development, more solutions and test analyses to conduct software installation are encouraged to be contributed. This research developed one possible solution for the challenges that software installation testing presents. The next step would be to implement this solution by installing software onto computers with different configurations. In addition, all the test cases could get automated into scripts to be executed through an automation tool to speed up the execution time.

To further enhance this model, a GUI model would be designed to complement the solution in this research because it would simplify the testing process by laying out the steps. The GUI model would provide the visual aspect of reproducing the steps clearly to test each software function.

References

- Agruss, C. (2000). Software installation testing. *Software Testing & Quality Engineering*, 2, 32-37. Retrieved February 20, 2007, from <http://www.stickyminds.com/sitewide.asp?ObjectId=5001&Function=edetail&ObjectType=MAGAZINE>
- Ammann, P. & Black, P. (1999, November 17-19). *A specification-based coverage metric to evaluate test sets*. Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering. Retrieved April 15, 2007. (Digital Object Identifier 10.1109/HASE.1999.809499)
- Apfelbaum, L. & Doyle, J. (1997, May). *Model based testing*. Paper distributed at the 1997 Software Quality Week Conference. Retrieved May 5, 2007, from http://www.geocities.com/model_based_testing/sqw97.pdf
- Blackburn, M., Busser, R., Nauman, A., Knickerbocker, R., & Kasuda, R. (2002). *Mars polar lander fault identification using model-based testing*. Presented at the Eighth IEEE International Conference on Engineering of Complex Computer Systems. Retrieved May 5, 2007, from <http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/iceccs/2002/1757/00/1757toc.xml&DOI=10.1109/ICEC CS.2002.1181509>
- Chen, Y., Probert, R., & Robeson, K. (2004). *Effective test metrics for test strategy evolution*. Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research. Retrieved February 7, 2007, from http://www.site.uottawa.ca/~ychen/Paper_CASCON04.pdf
- Connell, M. & Menzies, T. (1996). *Quality metrics: Test coverage analysis for Smalltalk*. Presented at Tools Pacific 1996 in Melbourne, Australia. Retrieved August 12, 2006, from <http://menzies.us/pdf/96conel.pdf>
- Dalal, S., Jain, A., Karunanithi, N., Leaton, J., & Lott, C. (1998). *Model-based testing of a highly programmable system*. Proceedings of the Ninth International Symposium on Software Reliability Engineering. Retrieved April 15, 2007. (Digital Object Identifier 10.1109/ISSRE.1998.730876)

- Dalal, S., Jain, A., Karunanithi, N., Leaton, J., Lott, C., Patton, G., & Horowitz, B. (1999). *Model-based testing in practice*. Proceeding of the 1999 International Conference on Software Engineering. Retrieved August 25, 2006. (Digital Object Identifier 10.1109/ICSE.1999.841019)
- Diaz, E., Tuya, J., & Blanco, R. (2004). *A modular tool for automated coverage in software testing*. Paper appears in the Eleventh Annual International Workshop on Software Technology and Engineering Practice. Retrieved August 25, 2006. (Digital Object Identifier 10.1109/STEP.2003.2)
- El-Far, I. & Whittaker, J. (2001). Model-based software testing. *Encyclopedia on Software Engineering*. Retrieved June 9, 2007, from http://www.geocities.com/model_based_testing/ModelBasedSoftwareTesting.pdf
- Gao, J., Kwok, K., & Fitch, T. (2007). *Modeling and analysis for software installation testing*. Paper presented at the 2007 ROSATEA Conference. Retrieved May 25, 2007.
- Gao, J., Tirumalasetti, S., & Hsu, C. (2006). *Automatic software installation testing using a model-based approach*. San Jose, California: San Jose State University, Department of Computer Engineering.
- Gao, J., Tsao, H.S., & Wu, Y. (2003). *Testing and Quality Assurance for Component-Based Software*. Norwood, Massachusetts: Artech House, Inc.
- Kaner, C., Falk, J., & Nguyen, H. (1999). *Testing Computer Software* (2nd ed.). New York, New York: John Wiley & Sons, Inc.
- Matsuodani, T. & Tsuda, K. (2004). Evaluation of debug-testing efficiency by duplication of the detected fault and delay time of repair. *Information Sciences – Informatics and Computer Science: An International Journal*, 166, 83-103. Retrieved August 20, 2006. (Digital Object Identifier 10.1016/j.ins.2003.11.001)
- Memon, A., Soffa, M., & Pollack, M. (2001). *Coverage criteria for GUI testing*. Proceedings of the Eighth European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Retrieved September 12, 2006. (Digital Object Identifier 10.1145/503209.503244)
- Pretschner, A., Prenninger, W., Wagner, S., Kuhnel, C., Baumgartner, M., Sostawa, B., Zolch, R., & Stauner, T. (2005). *One evaluation of model-based testing and its automation*. Proceedings of the 27th International Conference on Software Engineering. Retrieved April 15, 2007, from

<http://www4.in.tum.de/~wagnerst/publ/icse05.pdf>

- Pusala, R. (2006). *Operational excellence through efficient software testing metrics*. Retrieved February 19, 2007, from Infosys Website:
<http://www.infosys.com/services/enterprise-quality-services/white-papers/operational-excellence.pdf>
- Robinson, H. (1999a). *Finite state model-based testing on a shoestring*. Paper presented at the STARWEST '99 Conference. Retrieved May 5, 2007, from
http://www.geocities.com/model_based_testing/shoestring.htm
- Robinson, H. (1999b). *Graph theory techniques in model-based testing*. Paper presented at the 1999 International Conference on Testing Computer Software. Retrieved May 5, 2007, from
http://www.geocities.com/harry_robinson_testing/graph_theory.htm
- Salem, A., Rekab, K., & Whittaker, J. (2004). Prediction of software failures through logistic regression. *Information & Software Technology*, 46, 781-789. Retrieved August 20, 2006. (Digital Object Identifier 10.1016/j.infsof.2003.10.008)
- Stikkel, G. (2005). Dynamic model for the system testing process. *Information & Software Technology*, 48, 578-585. Retrieved August 19, 2006. (Digital Object Identifier 10.1016/j.infsof.2005.06.003)
- Tirumalasetti, S. & Hsu, C. (2006). *A systematic solution for software installation testing*. Unpublished master's thesis project, San Jose State University, San Jose, California, USA.
- Wikipedia (2006). *Black box testing*. Retrieved September 5, 2006, from
http://en.wikipedia.org/wiki/Black_box_testing
- Wikipedia (2007). *Ad hoc*. Retrieved May 5, 2007, from
http://en.wikipedia.org/wiki/Ad_hoc
- Woodward, M. & Hennell, M. (2005). Strategic benefits of software test management: A case study. *Journal of Engineering & Technology Management*, 22, 113-140. Retrieved August 20, 2006. (Digital Object Identifier 10.1016/j.jengtecman.2004.11.006)

Appendix A
Turbo Tax's SEC Model

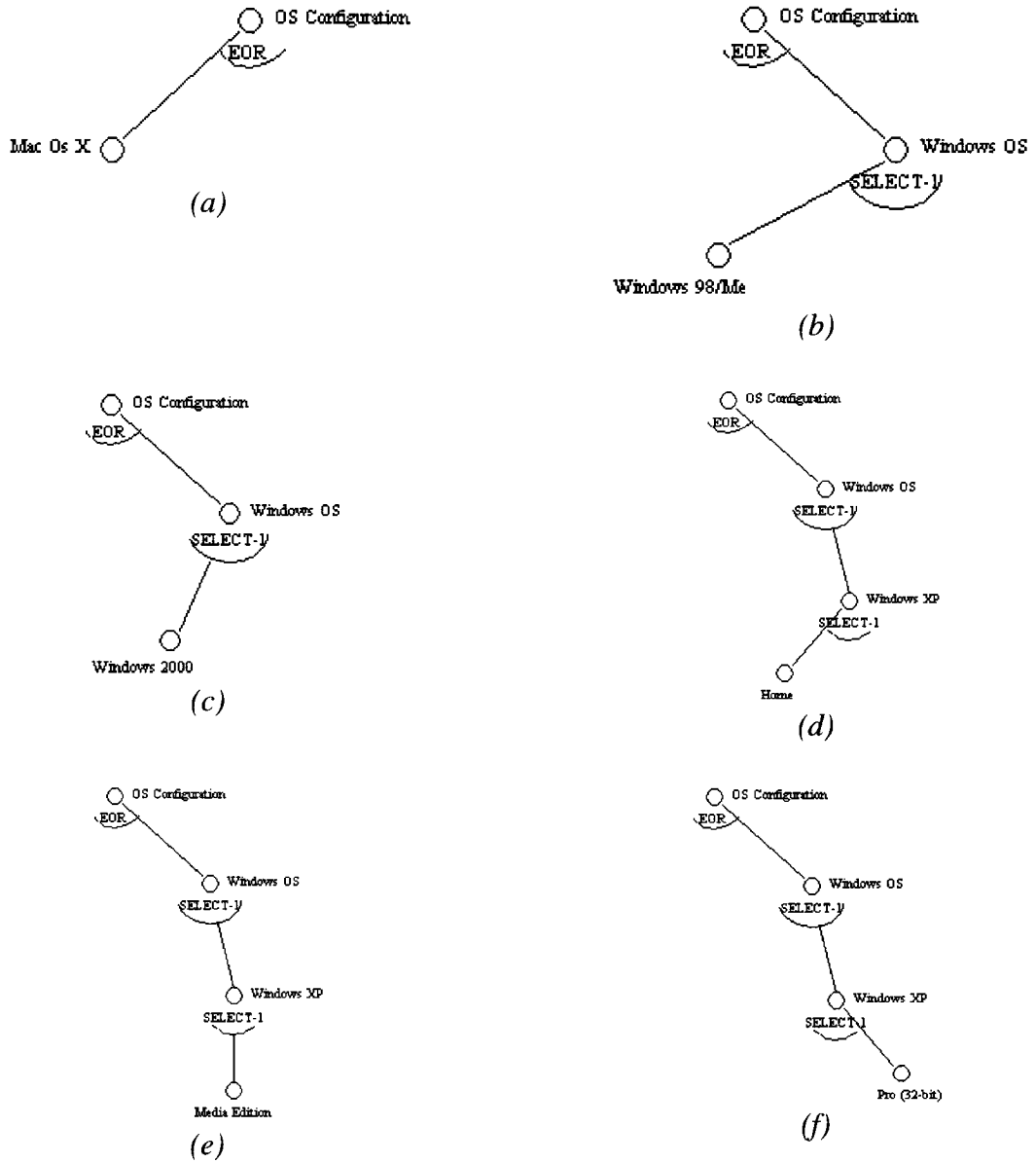


Figure A1. Turbo Tax's SEC semantic spanning trees.

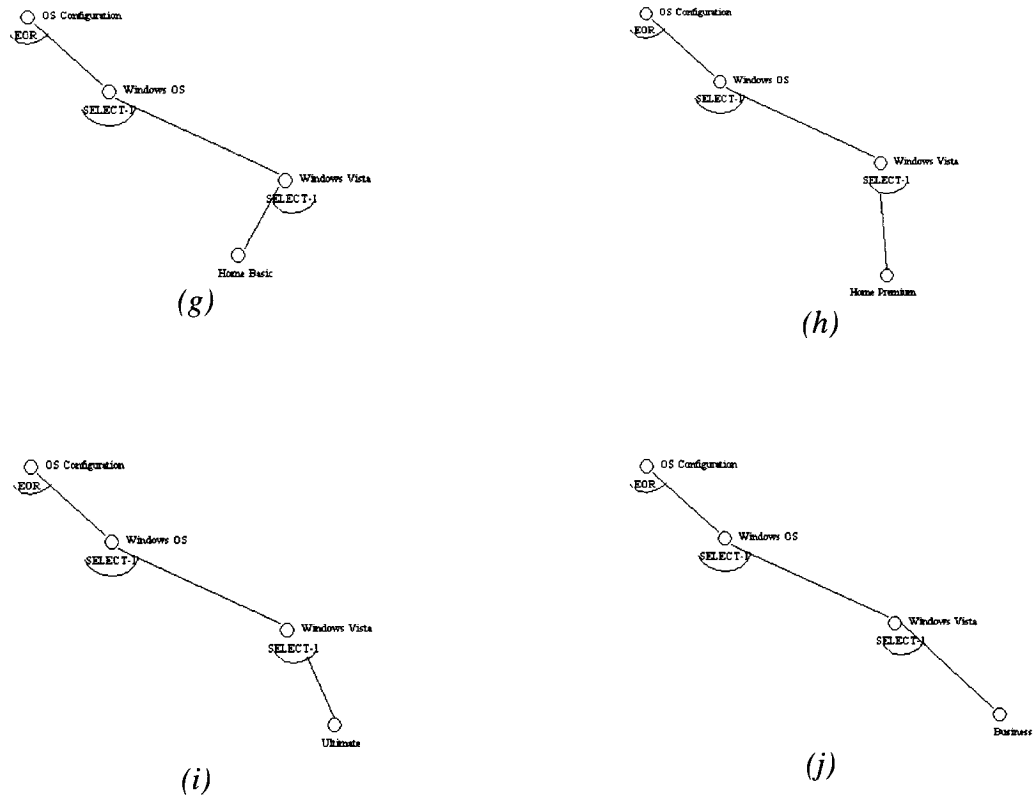


Figure A1. Turbo Tax's SEC semantic spanning trees (continued).

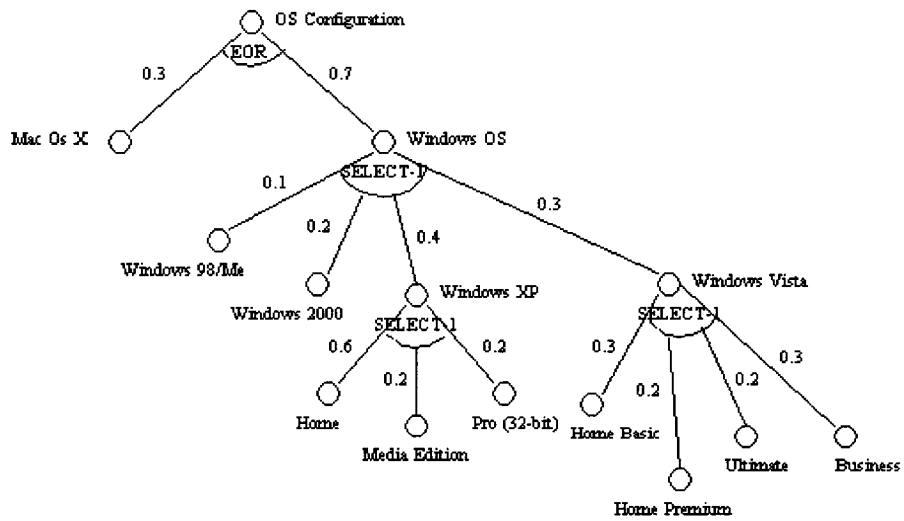


Figure A2. Turbo Tax's SEC weighted semantic tree.

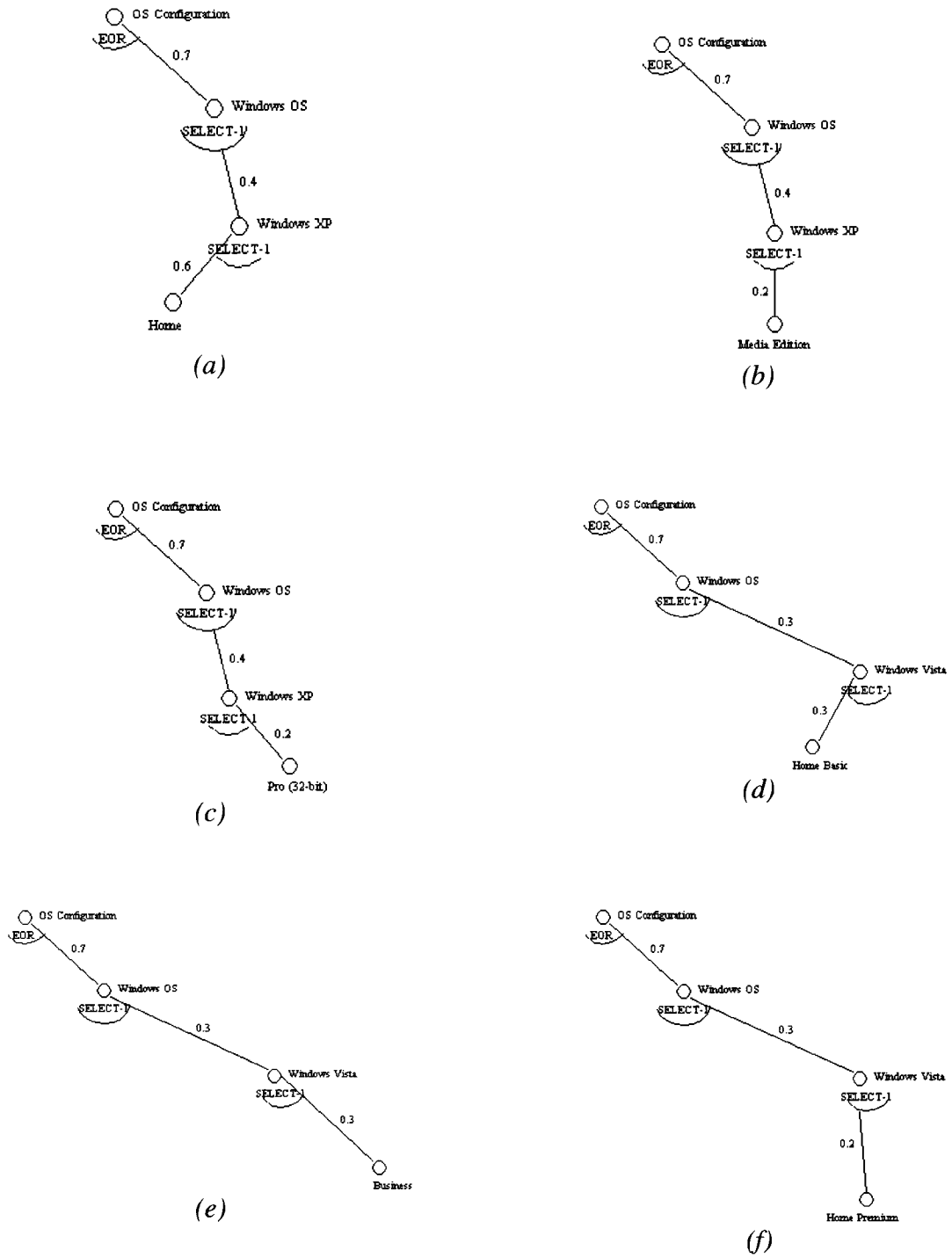


Figure A3. Turbo Tax's SEC weighted semantic spanning trees.

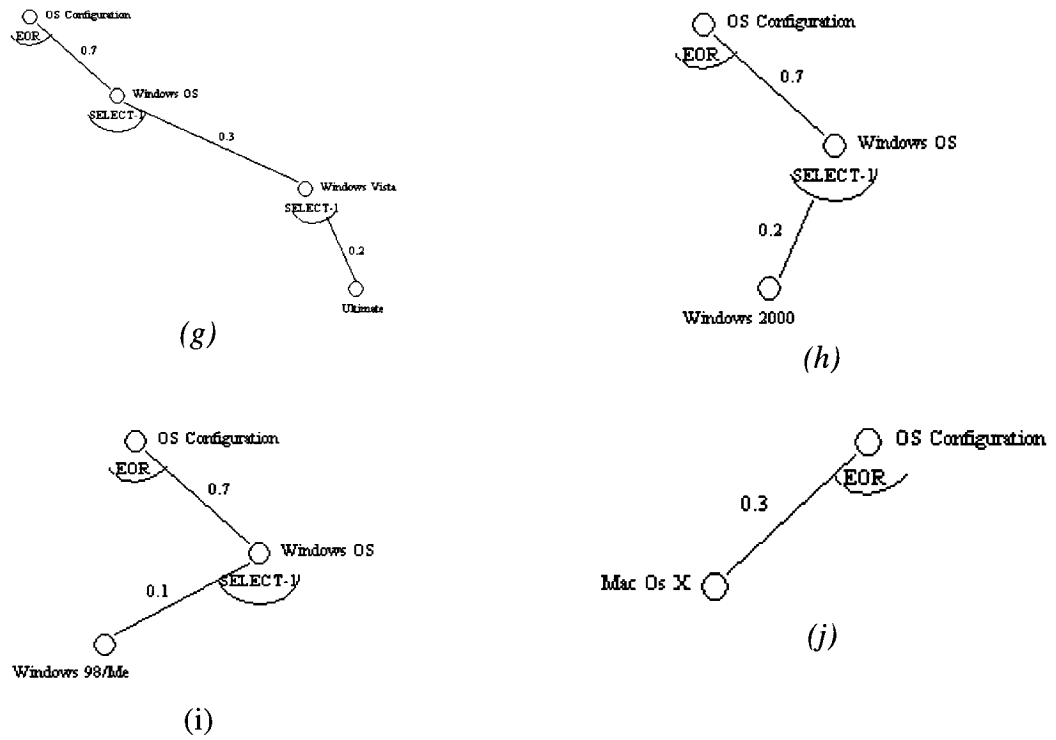


Figure A3. Turbo Tax’s SEC weighted semantic spanning trees (continued).

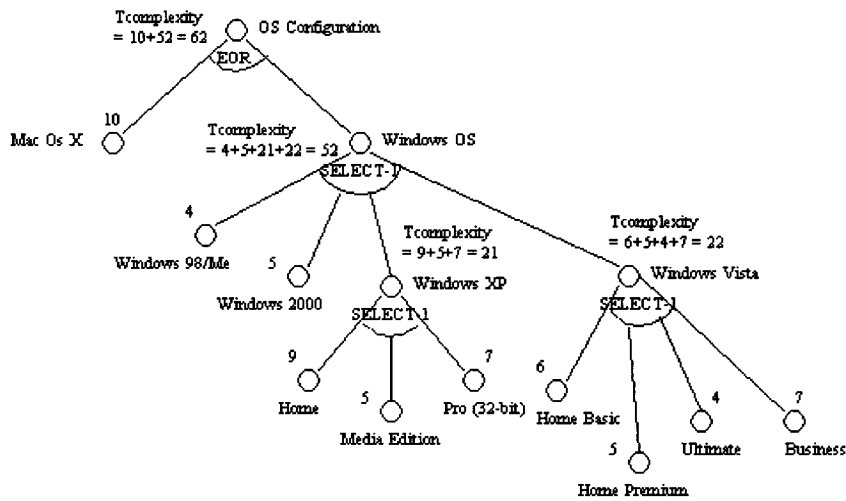


Figure A4. Turbo Tax’s SEC semantic tree with semantic test complexity.

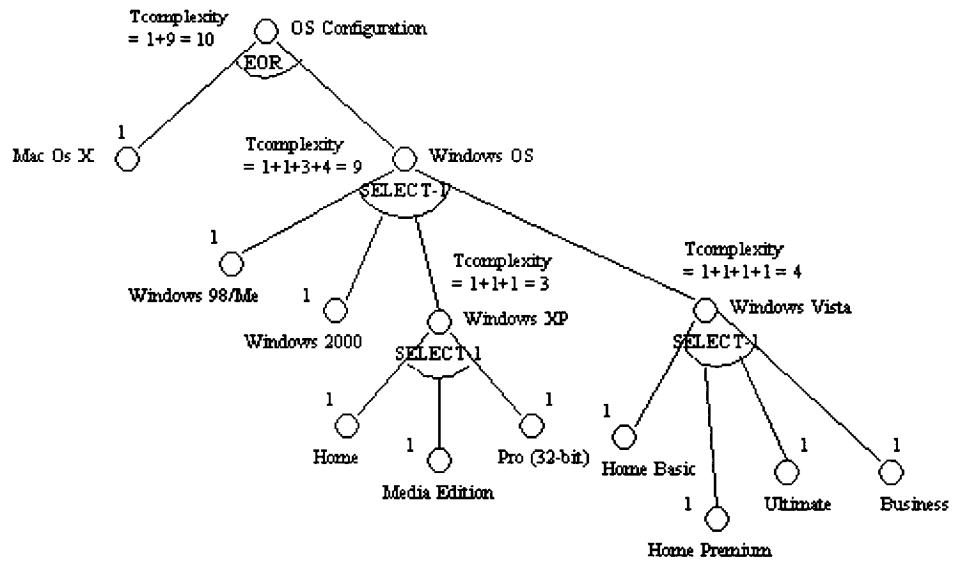


Figure A5. Turbo Tax's SEC semantic tree with spanning tree test complexity.

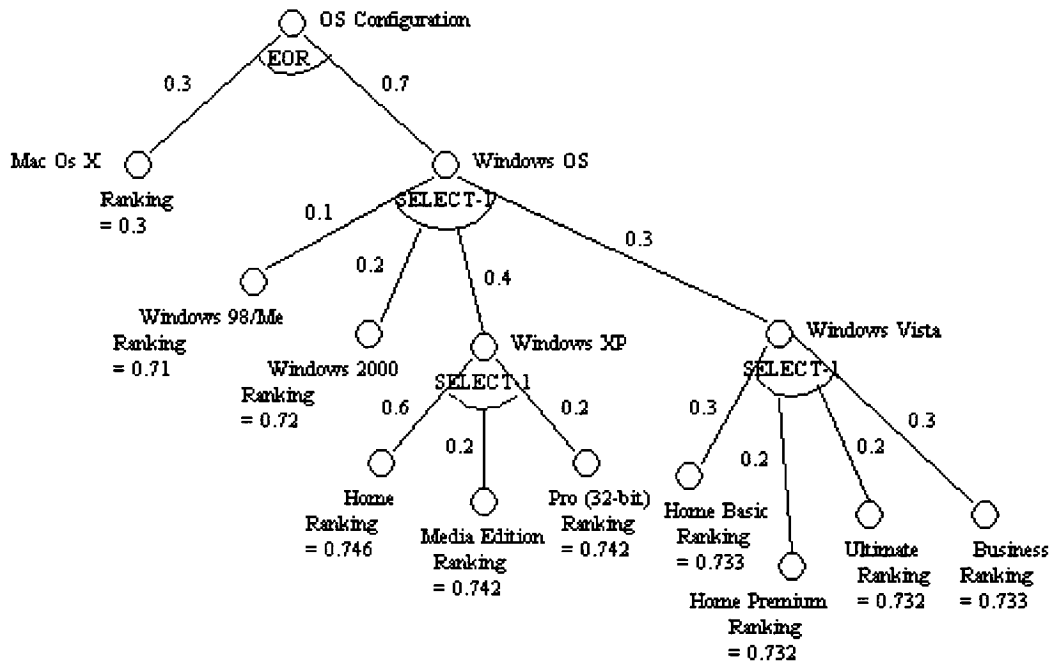


Figure A6. Turbo Tax's SEC ranking semantic tree.

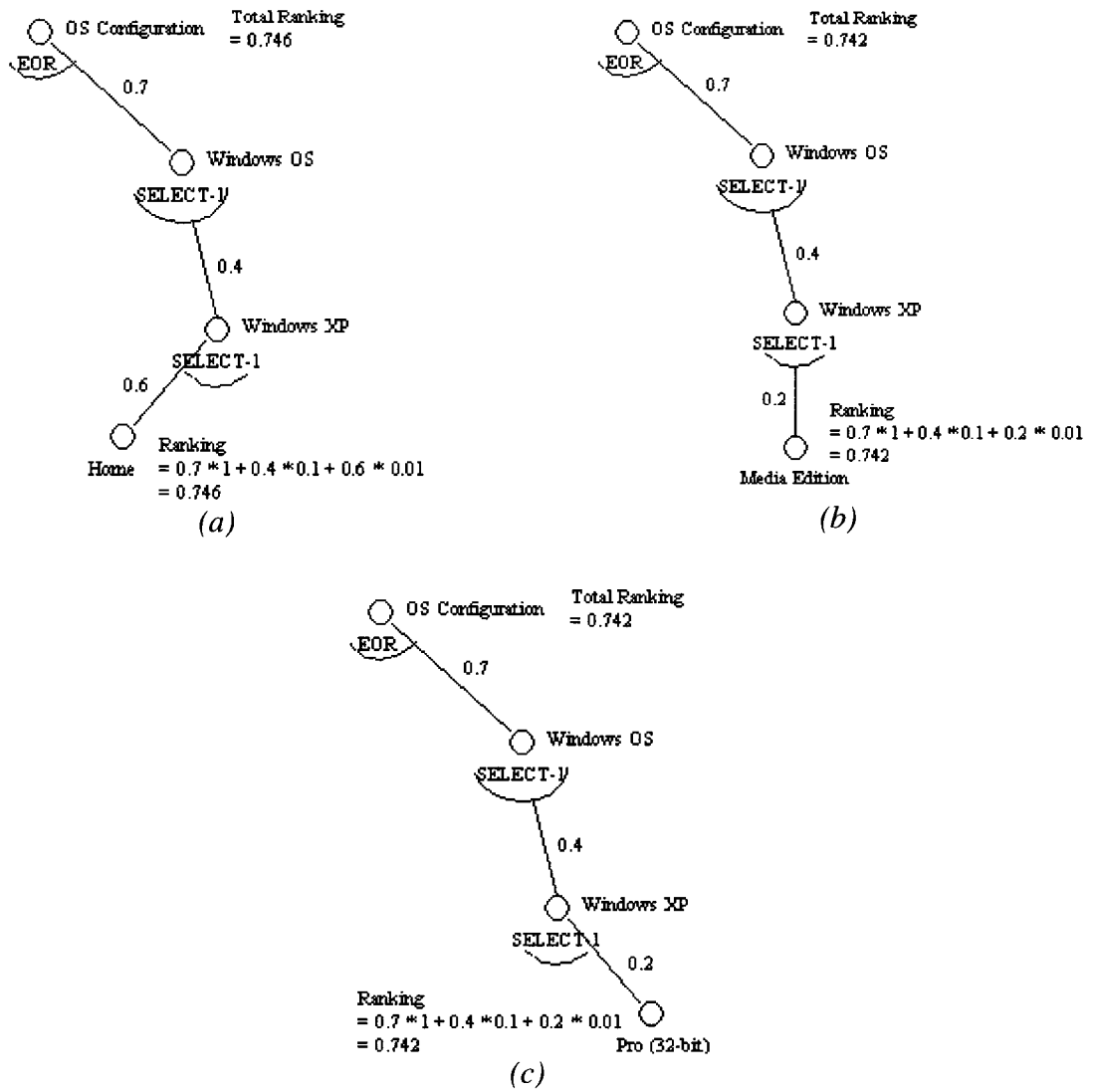
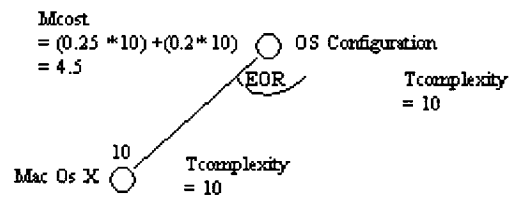
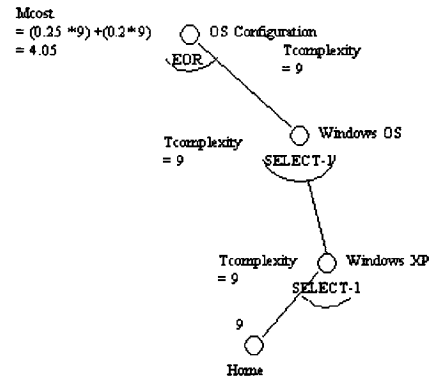


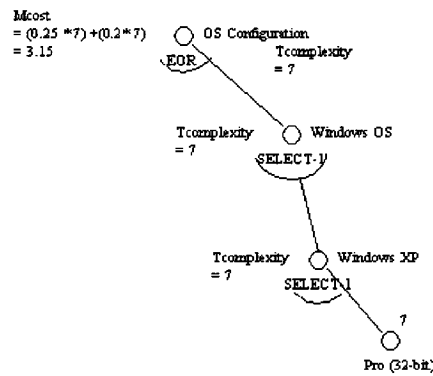
Figure A7. Turbo Tax's SEC ranking semantic spanning trees.



(a)



(b)

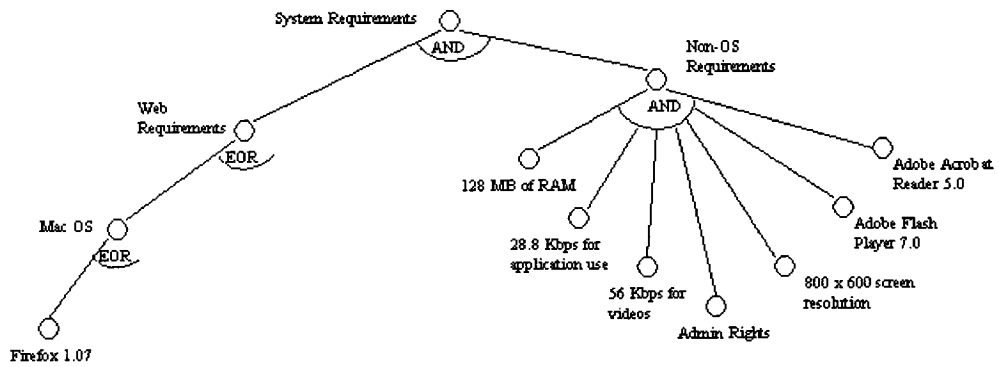


(c)

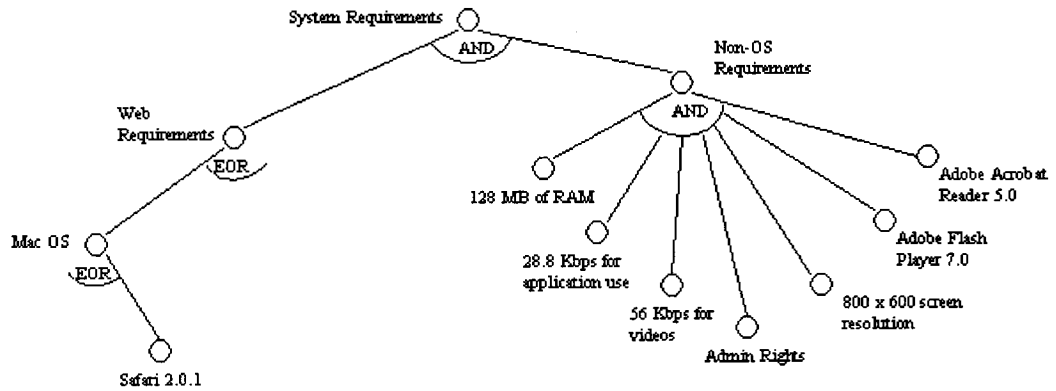
Figure A8. Turbo Tax's SEC test cost semantic spanning trees using method #1.

Appendix B

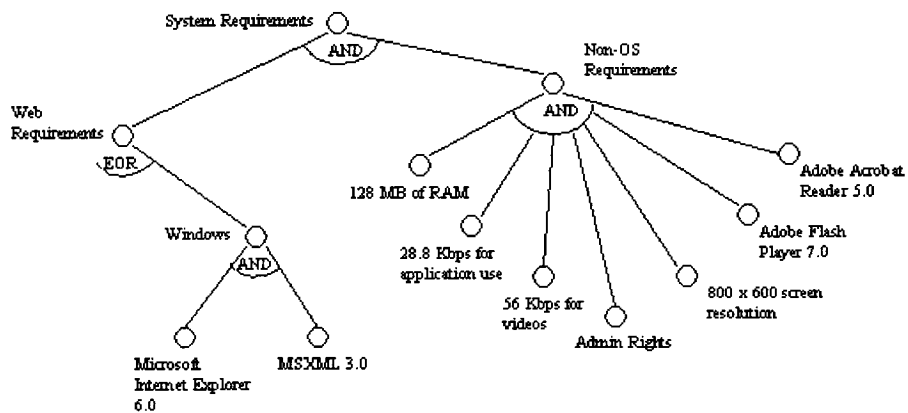
Turbo Tax's SIC Model



(a)



(b)



(c)

Figure B1. Turbo Tax's SIC semantic spanning trees.

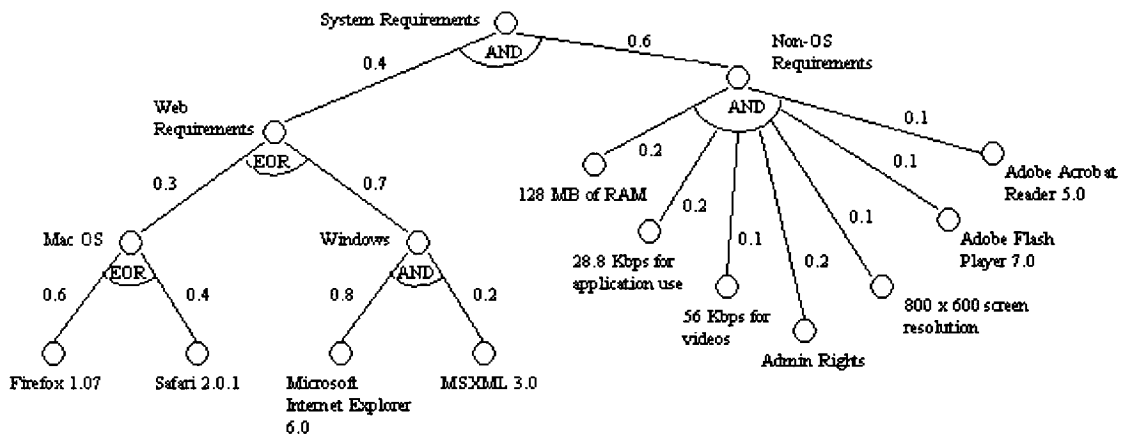
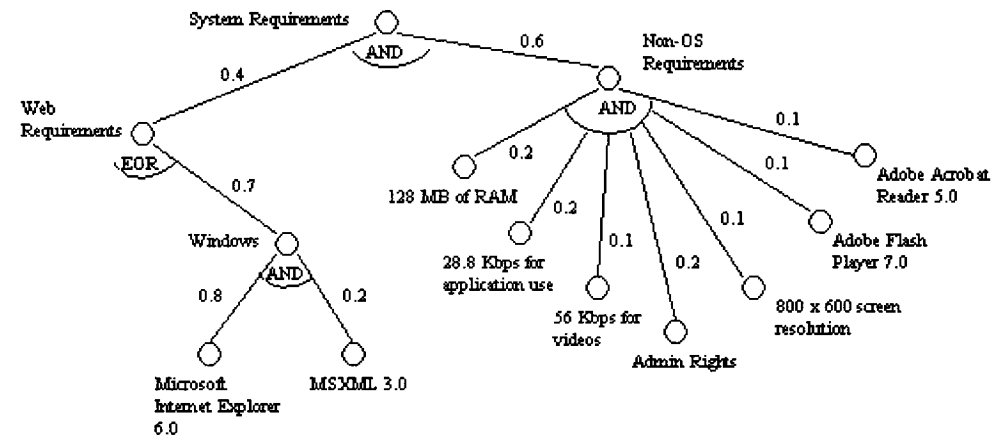
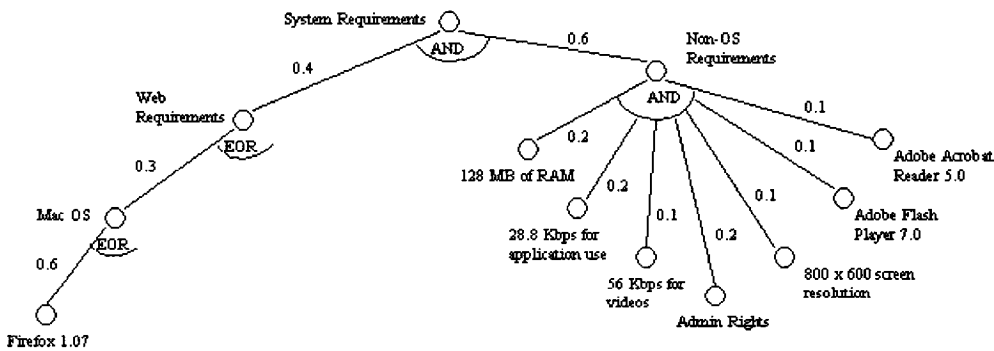


Figure B2. Turbo Tax's SIC weighted semantic tree.

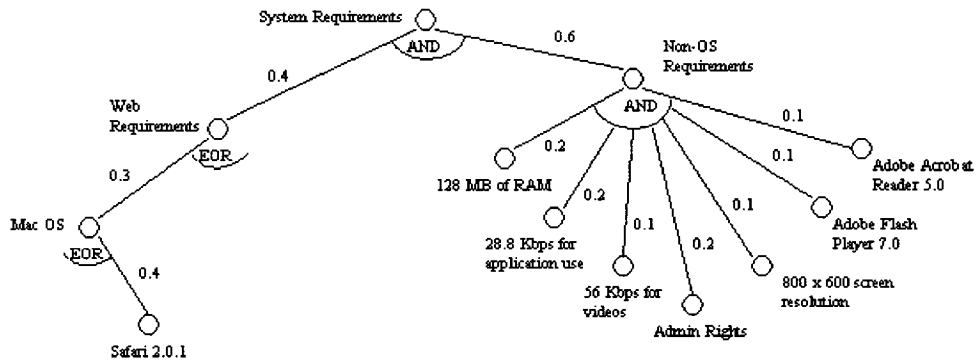


(a)



(b)

Figure B3. Turbo Tax's SIC weighted semantic spanning trees.



(c)

Figure B3. Turbo Tax's SIC weighted semantic spanning trees (continued).

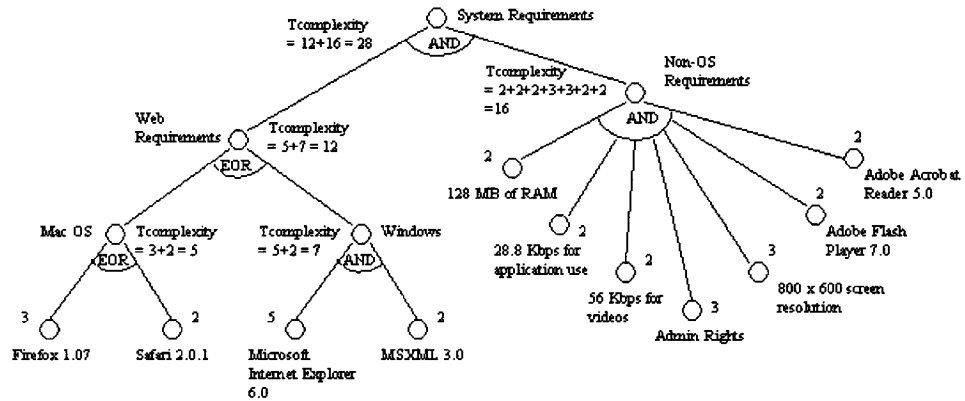


Figure B4. Turbo Tax's SIC semantic tree with semantic test complexity.

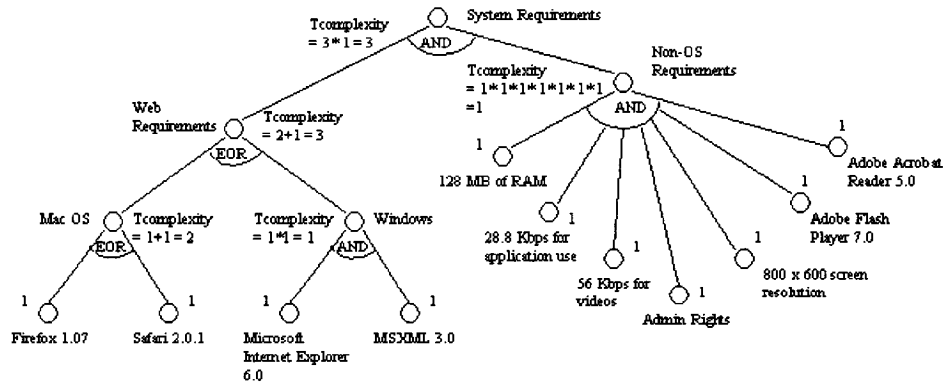


Figure B5. Turbo Tax's SIC semantic tree with spanning tree test complexity.

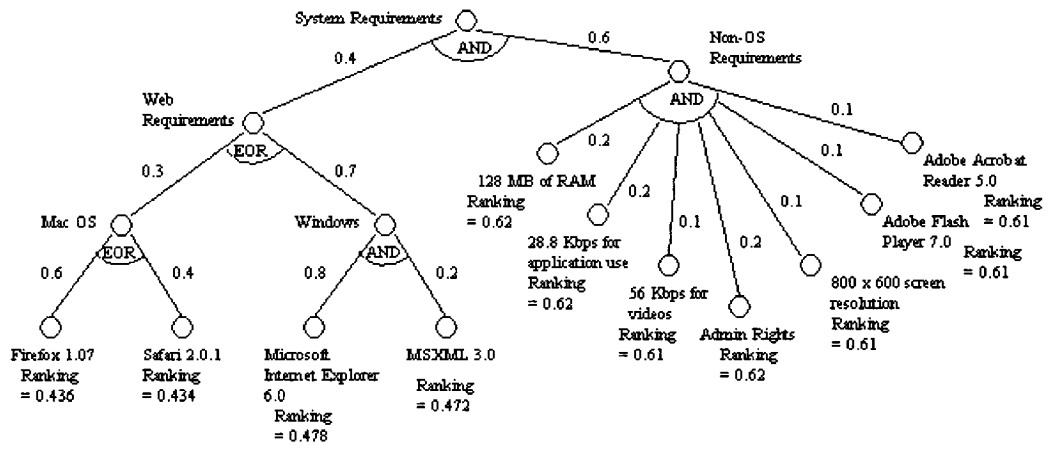


Figure B6. Turbo Tax's SIC ranking semantic tree.

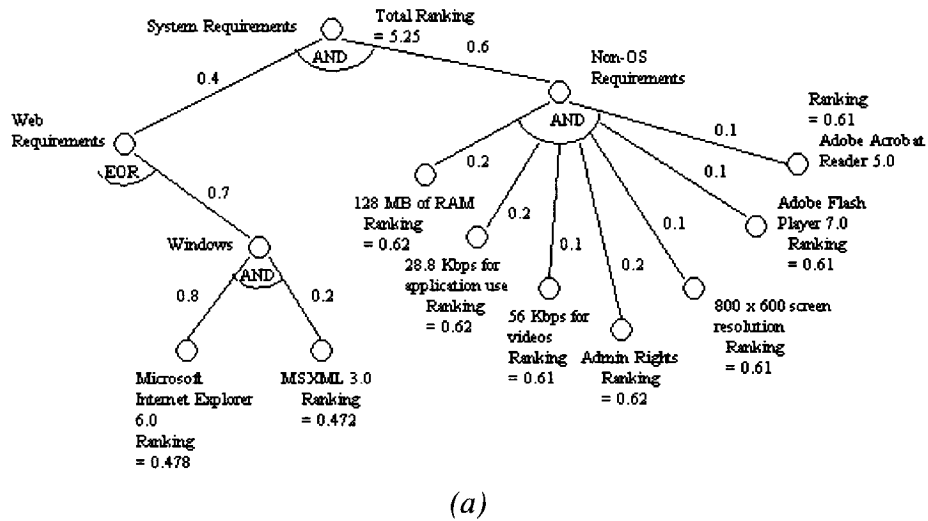


Figure B7. Turbo Tax's SIC ranking semantic spanning trees.

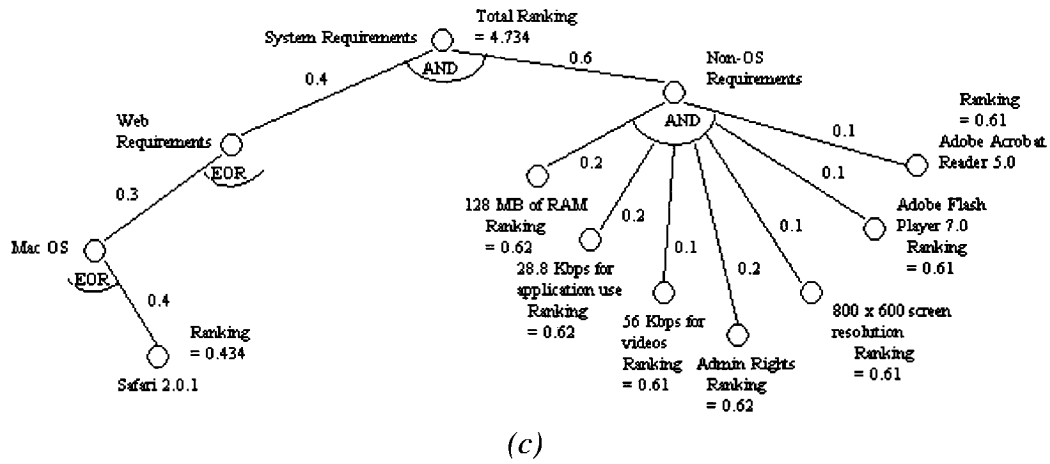
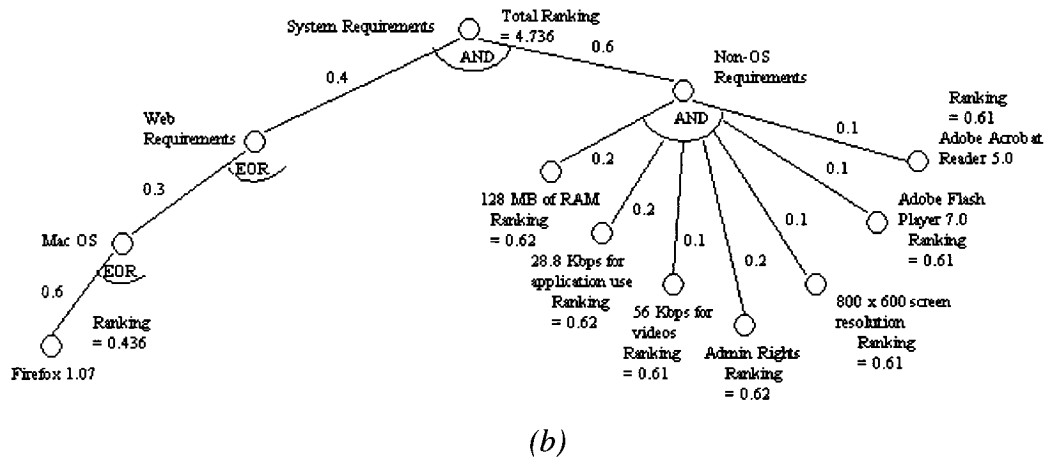


Figure B7. Turbo Tax's SIC ranking semantic spanning trees (continued).

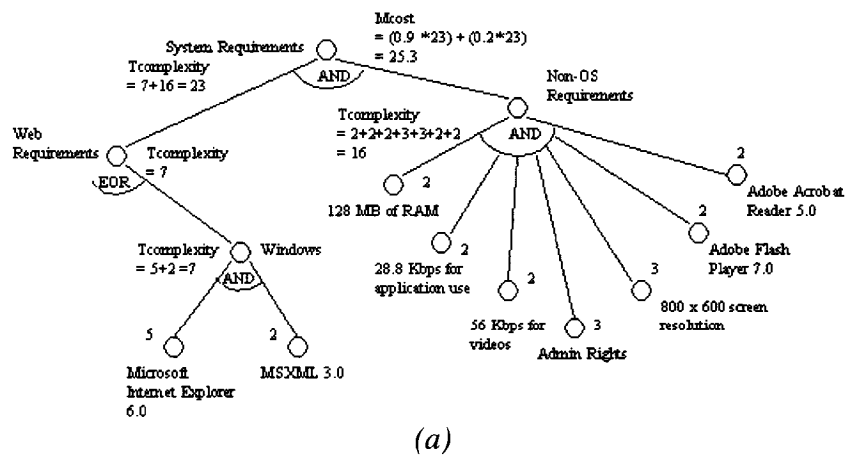
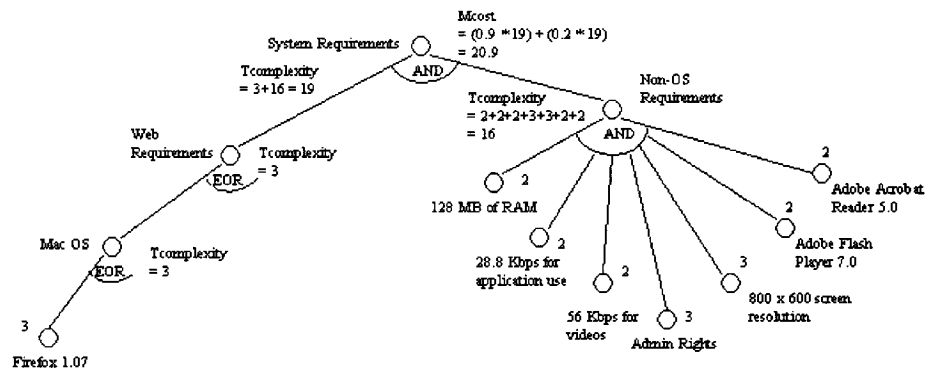
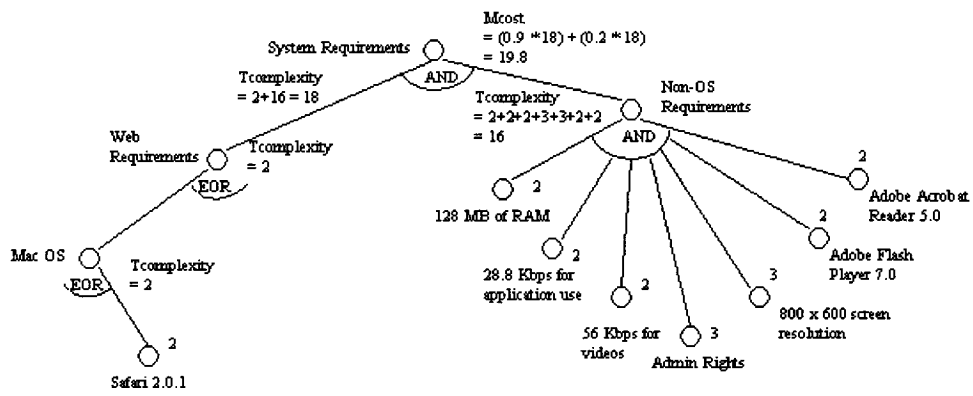


Figure B8. Turbo Tax's SIC test cost semantic spanning trees using method #1.



(b)



(c)

Figure B8. Turbo Tax's SIC test cost semantic spanning trees using method #1

(continued).

Appendix C

Turbo Tax's SIF Model

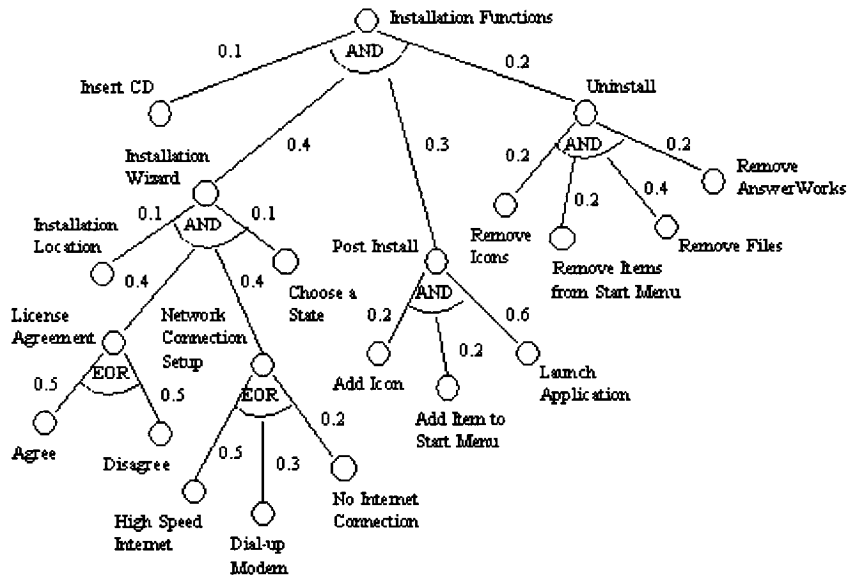


Figure C1. Turbo Tax's SIF weighted semantic tree.

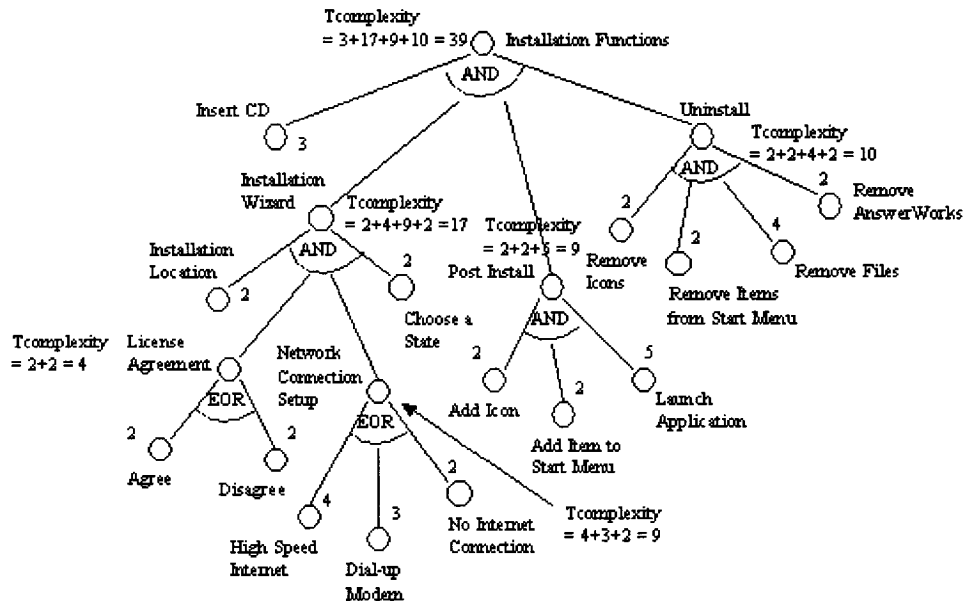


Figure C2. Turbo Tax's SIF semantic tree with semantic test complexity.

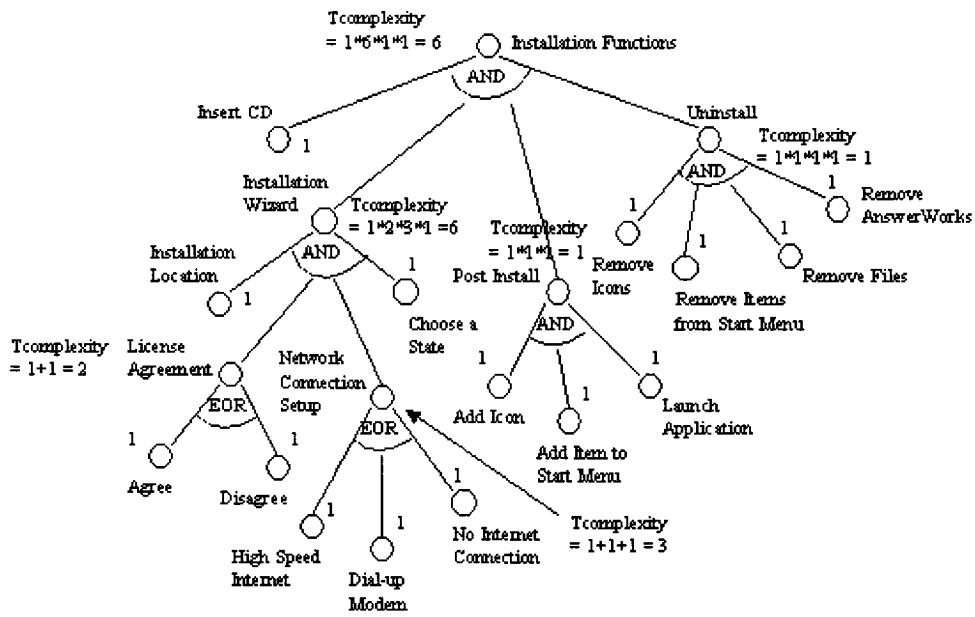


Figure C3. Turbo Tax's SIF semantic tree with spanning tree test complexity.

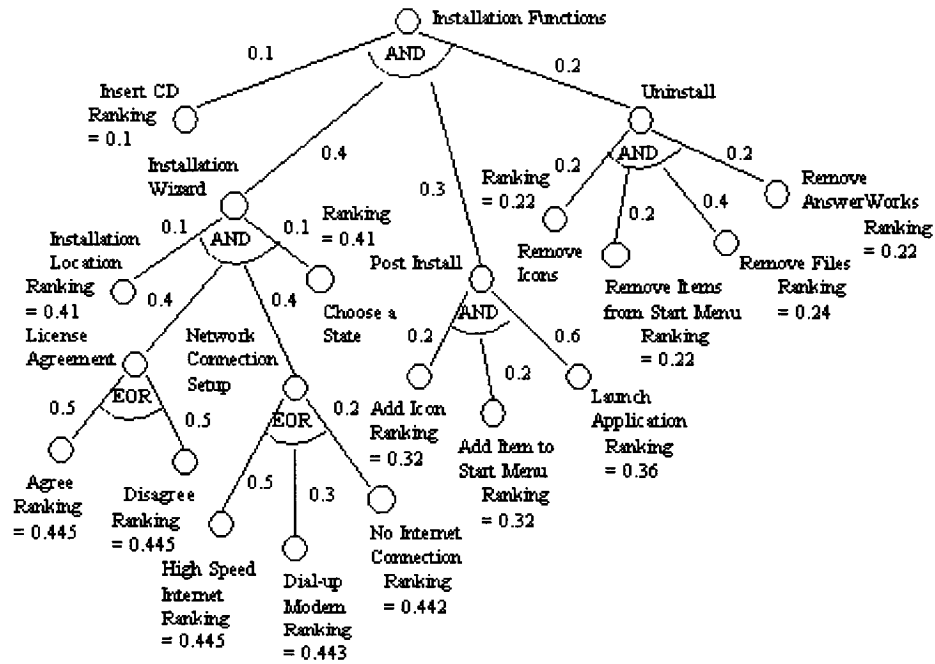


Figure C4. Turbo Tax's SIF ranking semantic tree.

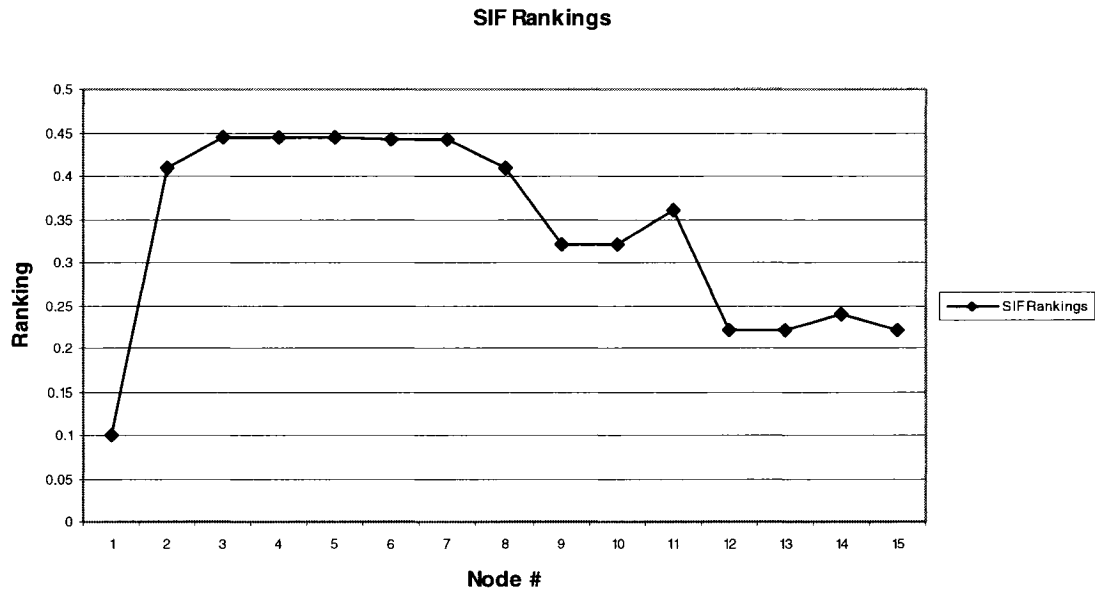


Figure C5. Turbo Tax's SIF ranking graph.

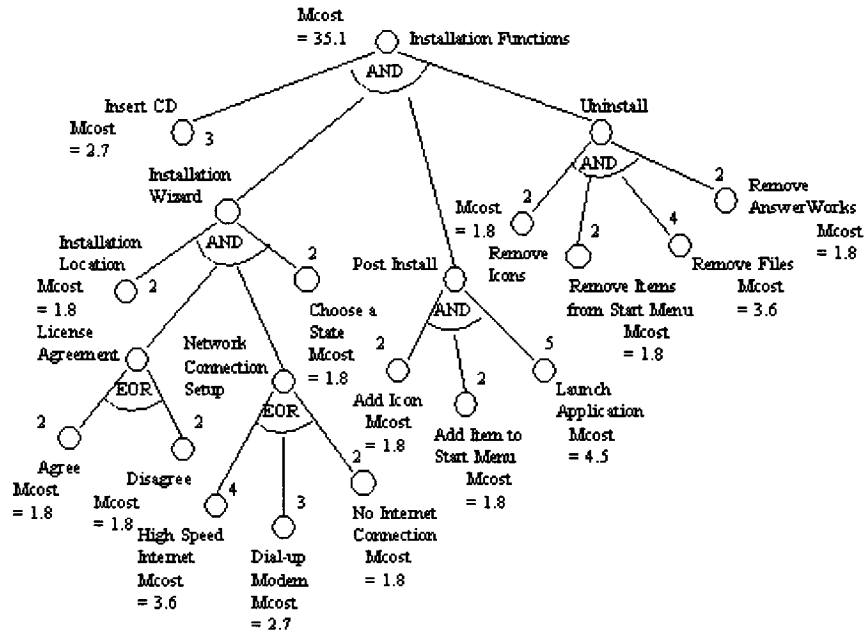


Figure C6. Turbo Tax's SIF test cost semantic tree using method #2.