國立交通大學

資訊科學與工程研究所

碩 士 論 文

應用動態樁技術於合作式網頁應用程式測試

Applying Dynamic Stubbing Technique to Support Collaborative Testing of Web Application

研 究 生：李佳玫

指導教授：曾憲雄 博士

黃世昆 博士

中 華 民 國 一百 年 八 月

應用動態樁技術於合作式網頁應用程式測試
Applying Dynamic Stubbing Technique to Support Collaborative Testing of Web Application

研 究 生：李佳玫　　Student：Jia-Mei Lee

指導教授：曾憲雄 博士　　Advisors：Dr. Shian-Shyong Tseng

黃世昆 博士　　Dr. Shin-Kun Huang

國立交通大學
資訊科學與工程研究所
碩士論文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

June 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年八月

應用動態樁技術於合作式網頁應用程式測試

研究生：李佳玫　　　　　　　　　　　　　指導教授：曾憲雄博士

黃世昆博士

國立交通大學資訊科學與工程學系研究所碩士班

摘要

現今，在網際網路上大量的免費人力資源通常被運用來減少測試成本與驗證軟體，如：線上遊戲與開放原始碼軟體。然而，傳統的合作測試方法在不考慮大眾測試者有共同的測試偏好與測試者素質參差不齊的情況下，往往會有測試時間難以收斂與測試報告的可信度不高等議題。為了加速收斂網站測試，有必要應用細顆粒的網頁應用程式模型來平行化測試工作。在這篇研究中，首先我們提出狀態轉換圖來為使用者的執行行為建模以達到分解網頁應用程式測試問題的目的。更進一步為了達到加速測試速度與改善測試報告的品質，並且減少合作測試的總成本，我們提出了一個動態樁技術來引導使用者進行測試。動態樁技術結合所提出的細顆粒的網頁應用程式模型可以在大眾測試者沒有察覺的情況下改變測試環境來導引他們解決子問題。實驗結果顯示我們所提出的方法可以減少 50% 的測試成本與增加 30% 的偵測效能。

關鍵字：大眾化分類、合作式測試、網頁應用程式測試、狀態轉換圖、程式相依圖、動態樁技術

Applying Dynamic Stubbing Technique to Support Collaborative Testing of Web Application

Student: Jia-Mei Lee **Advisors:Dr. Shian-Shyong Tseng**

Dr. Shin-Kun Huang

Institute of Computer Science and Engineering

Nation Chiao Tung University

Abstract

Nowadays, large volunteers creeping on internet are usually treated as free human resources for reducing test cost and validating software, like online games and open source software. However, traditional collaborative testing design and management approach encounters the long due time and doubting test report resulting from the common preferences of users and unqualified testers, respectively. A fine-grained Web application model is essential to refine job assignments for speeding up test coverage. In this thesis, we first propose State Transition Diagram to model the users' runtime behaviors for decomposing Web application testing problem. Then, based on this fine-grained Web application model, a dynamic stubbing technique which allows folk testers contribute themselves in solving sub-problems with barely noticing the change of test environment is proposed for achieving faster test coverage speed and improving the quality of test report, and hence reduces the total cost of collaborative testing. The experimental results show that our proposed approach can reduce 50% test cost and increase 30% detection performance.

Keywords: folksonomy-based approach, collaborative testing, Web application testing, state transition diagram, program dependence graph, dynamic stubbing technique.

誌謝

本篇論文的完成，首先我要感謝我的指導教授，曾憲雄老師，老師總是犧牲自己休息的時間來指導我們並且不厭其煩的領導我一步一步的思考與表達方法的訓練，我所得到的不只是一篇論文，還有整個領域的研究方法、思考邏輯。這是在碩士兩年的求學過程中，最寶貴的知識。此外，也要特別感謝我的共同指導教授，黃世昆老師，在口試時給我很豐富的意見，同時也要感謝我的口試委員，黃國禎教授、袁賢銘教授，感謝你們許多寶貴的意見，讓整篇論文更完整。

再來也要感謝在研究過程中，給我許多鼓勵及建議的學長，宗儒學長，感謝你的付出與教導讓我得以順利完成此篇論文。還有元昕學長，曾經給我很多鼓勵與豐富的知識。還要感謝605的學長姐：喚宇學長、怡利學姐、佳榕學姐，在我最艱難的過程中給了我很大的鼓勵與指導，讓我可以更努力的堅持下去，還有其他實驗室的學長姐，瑞鋒學長、哲青學長、莉玲學姐，謝謝你們曾經給過的意見指導和幫助。再來是我最親愛的同學們，楷元、嘉凱，這兩年來我們一起成長，所有苦的甜的回憶我會一輩子珍藏，要一直保持聯絡！

最後我要感謝愛我的家人及朋友們，感謝爸媽從小栽培與關懷才能有現在的我，也要感謝阿君、阿淳、旻旻、小瀞、呆瓜、小白，總是不斷的給我勇氣，一路上的支持與鼓勵，很開心我的碩士生涯能夠有你們陪伴在旁。

感謝這一路上所經歷過的一切，所有的酸甜苦辣，都是我成長的力量，在此我獻上最誠摯的感謝，謝謝你們。

Table of Content

摘要 i

Abstract ii

致謝 iii

Table of Content iv

List of Figures vi

List of Tables viii

Chapter 1. Introduction 1

Chapter 2. Related Work 5

2.1 Structure of software systems 5

2.2 Behavior of software systems 5

2.3 Three-level dependence graph 6

Chapter 3. Web application testing model 8

3.1 Motivation 8

3.2 Value-oriented dependence graph 9

3.2.1 Tainted variables in branch predicate 10

3.2.2 Definition of value-oriented dependence graph 11

3.2.3 Value-oriented Dependence Graph (VDG) construction algorithm 13

3.2.4 Example of value-oriented dependence graph 15

3.3. State transition diagram 22

3.3.1 Problem decomposition 22

3.3.2 Definition of state transition diagram 23

3.3.3 State Transition Diagram (STD) construction algorithm 25

3.3.4 Example of state transition diagram 28

Chapter 4. Dynamic stubbing technique for Collaborative testing 30

4.1 Motivation example 30

4.2 Minimum Test Cost Problem in Collaborative Testing 32

4.2.1 Problem formulation 34

4.2.2 NP-Complete problem...36

4.3 Dynamic stubbing algorithm for Minimum Test Cost Problem....................................38

Chapter 5. Implementation and Experiment...43

5.1 System architecture and implementation 43

5.1.1 System architecture 43

5.1.2 System implementation 44

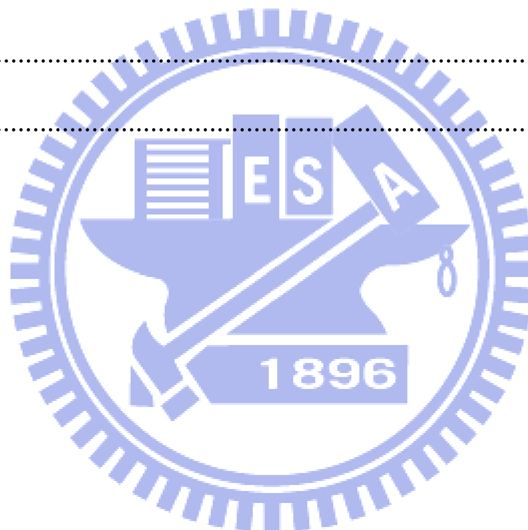
5.2 Experimental design and results...47

5.2.1 Experimental design 47

5.2.2 Experimental results...48

Chapter 6. Conclusion 54

References 55



List of Figures

Figure 1 Flow chart of program 3

Figure 2 Three-level dependence graph 7

Figure 3 Source code of Pay.aspx page (a) with its flow chart (b) in Example 9

Figure 4 Differents of three variables in Example 1 11

Figure 5 Value-oriented dependence graph in Example 1 13

Figure 6-1 Value-oriented dependence graph after executing Step 1 of VDG algorithm..........18

Figure 6-2 Value-oriented dependence graph after executing Step 2 of VDG algorithm..........18

Figure 6-3 Value-oriented dependence graph after executing Step 3 of VDG algorithm..........19

Figure 6-4 Value-oriented dependence graph after executing Step 4 of VDG algorithm..........19

Figure 6-5 Value-oriented dependence graph after executing Step 5 of VDG algorithm..........20

Figure 6-6 Value-oriented dependence graph after executing Step 6 of VDG algorithm..........20

Figure 6-7 Value-oriented dependence graph after executing Step 7 of VDG algorithm..........21

Figure 6-8 Value-oriented dependence graph after executing Step 8 of VDG algorithm..........21

Figure 6-9 Value-oriented dependence graph after executing Step 9 of VDG algorithm..........21

Figure 6-10 Value-oriented dependence graph after executing Step 10 of VDG algorithm....22

Figure 6-11 Value-oriented dependence graph after executing Step 11 of VDG algorithm....22

Figure 7 Idea of problem decomposition 23

Figure 8 State transition diagram in Example 1 25

Figure 9-1 State transition diagram after executing Step 1 of STD algorithm..........28

Figure 9-2 State transition diagram after executing Step 2 of STD algorithm..........29

Figure 10 Problem decomposition scenario 30

Figure 11 Due time of assignment 1 32

Figure 12 Due time of assignment 2 32

Figure 13 Intelligent collaborative testing system architecture..........43

Figure 14 Screen shot of register page of ICTS 45

Figure 15 Screenshot of ICTS tutorial 46

Figure 16 Screenshot of ICTS guiding..46

Figure 17 Screenshot of complete information of ICTS 47

Figure 18 Comparison of state complete degree of testing 49

Figure 19 Due time comparison..50

List of Tables

Table 1 Property of Patterns of Different level 16

Table 2 Property of Patterns of Different variable 17

Table 3 Sub-problem completion time of each folk tester matrix 31

Table 4 Assignment 1 for shopping Web-site testing 31

Table 5 Assignment 2 for shopping Web-site testing 31

Table 6 Notations of MTCP in collaborative testing 34

Table 7 Notations of Dynamic stubbing algorithm 39

Table 8 Value-oriented dependence graph statistics 48

Table 9 State complete degree of testing comparing 49

Table 10 Comparison of test time between two groups 50

Table 11 Comparison of the folk testers per due time 50

Table 12 Ten defects in Web application "BookStore" 51

Table 13 Comparison of test time of each state between two groups 53

Table 14 Comparison of fault-detection ability between two groups 53

Table 15 Fault-detection ability of two collaborative testing approaches...............................53

Chapter 1 Introduction

Collaborative testing is widely used in industries to reduce test cost and assure the software quality, especially in online game, open source software and Web applications. Low reliability of Web application will lead to serious detrimental effects for businesses, consumers, and the government because users increasingly depend on the Internet for routine daily operations. However, lacking of good test design and management approach, unqualified folk testers with some common preferences will slow down the test convergence and produce suspect test report. Due to the untrammeled nature of folks, restricted test scenario according to different test design and management approach may decrease their willingness and hence reduce available human resources. It procures more cost of collaborative testing. A good test design and management approach should take the willingness and common preferences of folk testers into consideration.

Problem decomposition which reduces original collaborative testing problem into several sub-problems is an efficient and effective approach for speeding up collaborative test and increasing the quality of test report. The reduced sub-problems are easier and can be assigned in parallel for decreasing the due time. Besides, the solution of sub-problems can be easily merged to the one of original problem, and hence can complete test. In the meanwhile, testers concentrating on small problem can detect faults more easily, and hence improving the quality of test report.

The success of problem decomposition relies on a proper Web application model. However, traditional Web application models, like page navigation diagram [1]-[9] and finite state machine (FSM) [10]-[20] and Petri-net [21]-[25], do not take software fault into consideration and hence suffer the risk of causing Type I and II error. A software fault classification based on program dependence graph has been proposed [26]. Tung et al. propose a novel test case generation algorithm based on this fault classification to generate a

test suite with full fault-detection [27]. Huang further extended Tung's model to three-level program dependence graph (page level, function level and code level) for considering the perspectives from folk testers (Web pages) and test objective from developer (basic blocks coverage) [28]. However, the three-level program dependence graph is a coarse-grained model because it does not model input values. Hence it cannot be applied to further problem decomposition according to runtime behavior of users.

In this thesis, we refine coarse-grained three-level dependence graph to fine-grained value-oriented dependence graph (VDG) which models users' runtime behavior by considering runtime input value. Figure 1 shows flow chart of program containing basic blocks 1, 2, and 3 where each basic block means they are a maximal code fragments without branching of a function [29]. If value of variable a is more than 0 then basic block 1 and basic block 2 can be covered. Otherwise, basic block 1 and basic block 3 can be covered. So, different values of variable may have different behaviors of program. By considering input value, the testing problem can be further decomposed into two sub-problems which are still able to meet the test criteria, basic blocks coverage. For the purpose of fine turning collaborative test plan, we further propose State Transition Diagram (STD) based on VDG in Chapter 3. Each state of STD represents a program behavior of a page. Based on this state transition diagram, we can assign job more precisely in collaborative testing and guide testers to meet the test objective, basic blocks coverage.

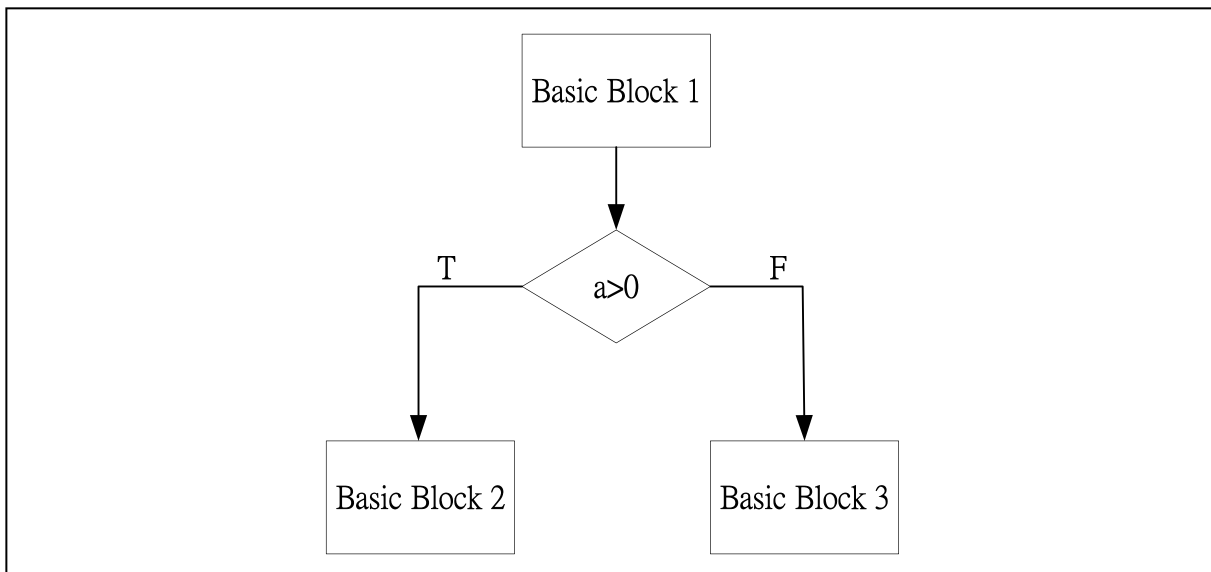


Figure 1. Flow chart of program

The Minimum Testing Cost Problem in collaborative testing (MTCP) can be considered as a variant of Job Assignment Problem (or Optimum Representative Set Problem) [30] with further constraints on testing resources and tester trustworthy. We reduce Job Assignment Problem to MTCP and prove that MTCP is NP-complete in Chapter 4. Therefore, a heuristic-based dynamic stubbing algorithm for job assignment is proposed to solve this problem by overcoming the issue of centralized preferences and willingness of folk testers which can speed up the test coverage and hence reduce the test cost.

We implement prototype system with our proposed approach by the dynamic stubbing technique which allows folk testers contributing their human resources with barely noticing varying test environment. Our collaborative testing system can collect tester user sessions during testing processes. To support collaborative testing, we record the testing logs and guide the test activities on our collaborative testing system. These testing logs are analyzed by proposed report analysis algorithm to form high quality test report. The experimental results show that our approach which reduces 50% due time and increases 30% detection rate is both efficient and effective.

We briefly outline the contents of this thesis. Chapter 2 provides related works. Chapter 3 gives the Web application modeling and the corresponding value-oriented dependence graph construction algorithm and state transition diagram construction algorithm. We also present the dynamic stubbing technique to support the Minimum Collaborative Testing Problem with the proposed heuristic-based dynamic stubbing algorithm in Chapter 4. Chapter 5 explains how the experimental design and experimental result. We conclude this thesis in Chapter 6.

Chapter 2 Related Works

In the chapter, we will discuss several different models of the web applications testing, and each model has different test goals. We briefly describe two types of the models, including structure of software systems only considering structure of systems and behavior of software systems considering both structure and behavior of systems.

2.1 Structure of software systems

There are some models which describe structure of program as follows. A program dependence graph [31]-[37] is a directed graph in which the nodes are statements and predicate expressions and the edges are dependences between the nodes. Two types of dependences are data dependence and control dependence between nodes. The data dependence between two statements means that the input variable of a statement is transferred from the other statement. The control dependence means that the flow of statements is decided by certain predicate expressions. The UML class diagrams [38] describing structures of software systems are also a directed graph in which nodes are classes of an object-oriented system and edges are dependences and inheritances between classes.

These structure-based models are unable to describe behaviors of folk testers and hence not suitable for collaborative testing.

2.2 Behavior of software systems

Another kind of software models, including page navigation diagram and finite state machines, aim to describe behavior of program. The page navigation diagram [1]-[9] is a directed graph where nodes are Web pages and edges are links between Web pages. The model can describe all test paths for Web applications testing. Huaikou et al. [1] proposed an approach which uses a regular expression characterizing the directed graph on page navigation diagram to generate test paths. Benedikt et al. [2] presented VeriWeb, which is a

tool for automatically navigating links of Web applications and exploring execution test paths through dynamic components of Web applications with a search algorithm. Shengbo et al. [4] and Zhongsheng et al. [7] proposed an algorithm to generate Test-Trees from page navigation diagram for satisfying link and page coverage.

The Final State Machine (FSM) [10]-[20] is used to model Web applications and it is a directed graph including nodes and edges. There are two types of the nodes: Web pages and associated components. There are three types of the edges: link edge, call edge and build edge. The link edge means that Web page can link the other Web page or component. The call edge means that Web page can call component through delivering requests to it. The build edge means that the component can build the new web page as responses to the requests. Andrews et al. used FSM to model and test Web applications, and then proposed an approach which decomposes Web application into several subsystems based on FSM with constraints to generate test sequences [10]. Liping et al. proposed the Kripke structure that is a model of FSM to model Web applications from the user's viewpoint [15]. The model can generate test sequences satisfying state and transition coverage for Web applications. However, page navigation diagram and finite state machines did not consider fault-detection ability and were risk of suffering Type I and II error. Furthermore, these models are coarse-grained because of the lack of input domain information. Three-level dependence graph which is also a model of behavior of software system is presented in next section.

2.3 Three-level dependence graph

Huang [28] considered the perspectives from folk testers (Web pages), test objective from developer (basic blocks coverage) and fault-detection ability to propose three-level dependence graph (page-level, function-level and code-level), illustrated in Figure 2. In page-level, the nodes are Web pages and the edges are links between Web pages. In function-level, the nodes are functional statements and the edges are constructed by functional

statements. And in code-level, the nodes are basic blocks that are a maximal code fragments without branching of a function and the edges are control flow between the basic blocks. Dependence edge and independence edge are intra-level edges where dependence edge can transfer data between Web pages, functions, and basic blocks. And the other edge is containing edge which can connect different nodes between page-level, function-level, and code-level. However, this three-level dependence graph cannot model the users' runtime behavior, and is hence coarse-grained and improper for fine job assignment in collaborative testing.

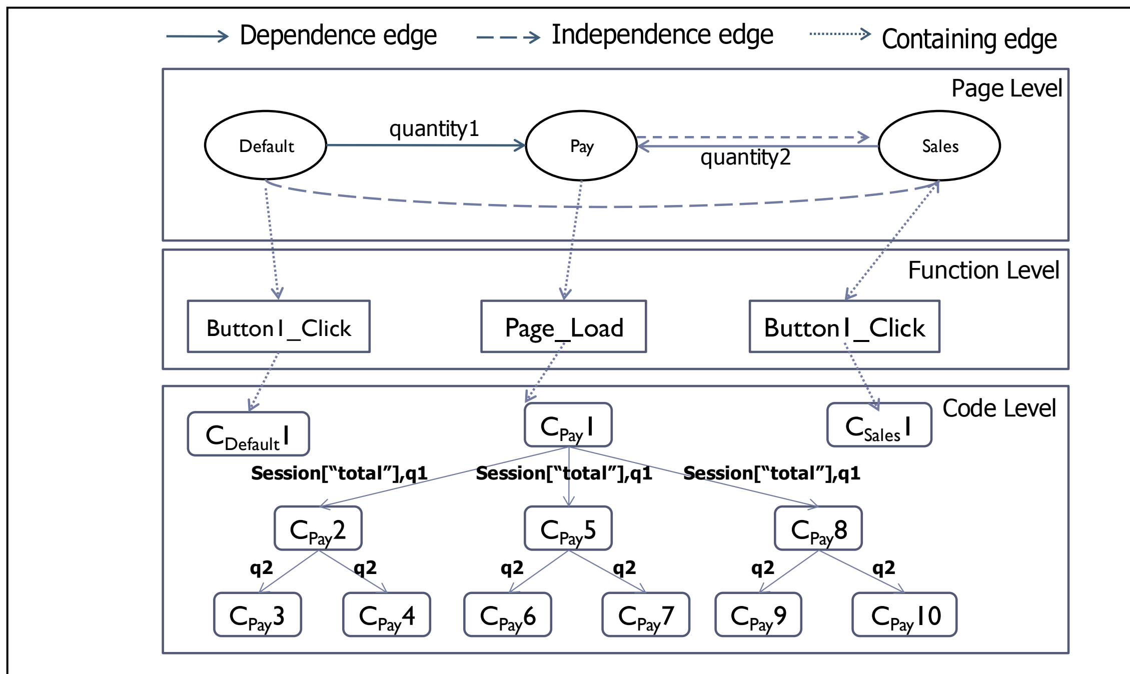


Figure 2. Three-level dependence graph

Chapter 3 Web application testing model

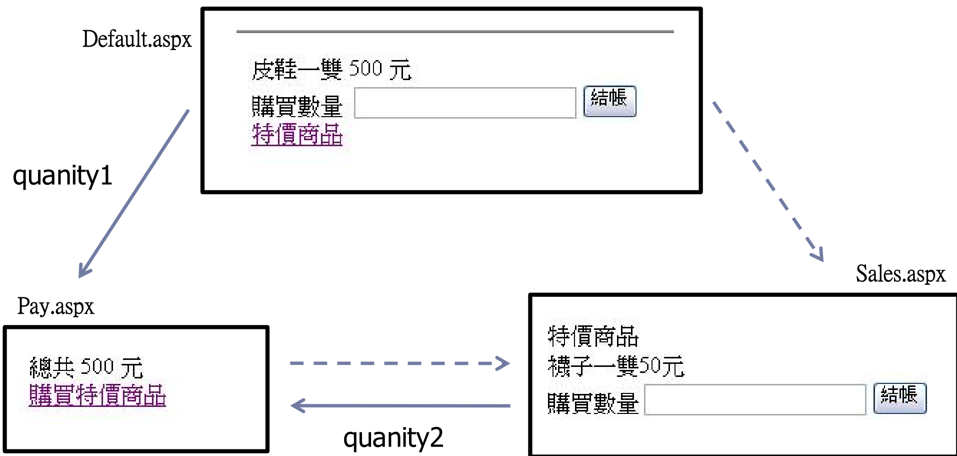
In the chapter, we will propose a novel model to support the collaborative testing for Web application. First, we start with a motivation example and then propose a fine-grained value-oriented dependence graph which models user's input domain information based on three-level dependence graph [28]. Finally, we propose state transition diagram which is a novel runtime behavior model for decomposing original collaborative testing problem into several sub-problems.

3.1 Motivation

One of the most importance objectives in software testing is to assure the software quality by covering all the code statements. However, covered code statements in each execution depend on input values. The Example 1 is given to motivate us that a fine job assignment needs to take variable value into consideration.

Example 1:

The shopping Web-site contains Default.aspx, Pay.aspx, and Sales.aspx pages. Consumers decide the amount of shoes (quantity1) and socks (quantity2) they want to buy in Default.aspx and Sales.aspx, and then this purchase information will be transferred to Pay.aspx for following payment. Figure 3a shows source code of Pay.aspx page. Figure 3b shows the flow chart for the source code of Pay.aspx page, and these basic blocks can be covered depend on consumer's input different values from Default.aspx page. If consumers key in quantity1 value which is less than or equal to 0, then $C_{Pay}1$ and $C_{Pay}2$ will be covered in this execution. Otherwise, $C_{Pay}1$ and $C_{Pay}3$ will be covered.



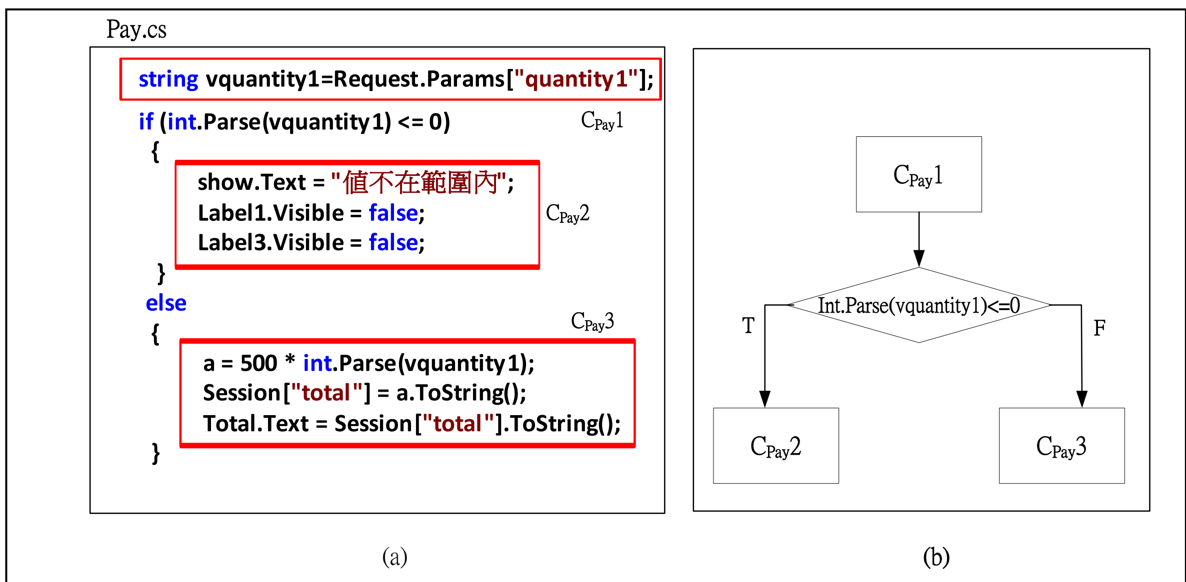


Figure 3. Source code of Pay.aspx page (a) with its flow chart (b) in Example 1

3.2 Value- oriented dependence graph

Based on the above motivated example, the three-level dependence graph shown in section 2.3 cannot describe different executions in different values because it cannot model users' input values. We first define the tainted variables in branch predicate which can be controlled by user and decide the flow of executions, and then we refined three-level dependence graph to value-oriented dependence graph. The definition and construction algorithm of value-oriented dependence graph are also presented in this section. Finally, an example is provided to illustrate value-oriented dependence graph construction algorithm.

3.2.1 Tainted variables in branch predicate

Figure 3 motivates us that a fine job assignment requires the value information of variables. However, only some variables will depend on user's input to influence the program execution flow. Therefore, we first consider the variable which can decide the flow of a program. This kind of variables dominating the execution flow is defined as branch predicate variables. In collaborative testing, testers can only use input variable and hyperlink to validate Web site. Hence, we define input variable and tainted variable which are controllable variable of testers. Finally, the tainted variables in branch predicate in a fine collaborative testing are defined if it is branch predicate variables and tainted variable. The formal definitions and notations of branch predicate variables, input variable, tainted variable and tainted variables in branch predicate are provided below:

Definition 1 (Branch predicate variables V_{BP}^i)

A variable V_{BP}^i is the i_{th} branch predicate variable if it is related to program behavior of a page. It affects code flow chart of the page.

Definition 2 (Input variable V_{IN}^i)

A variable V_{IN}^i is the i_{th} input variable if it is a frame of each page which user can input value.

Definition 3 (Tainted variable V_T^i)

A variable V_T^i is the i_{th} tainted variable if there is related to the input variable V_I^i of the page.

Definition 4 (Tainted variable in branch predicate V_{TBP}^i)

A variable V_{TBP}^i is the i_{th} tainted variable in branch predicate if it is branch predicate variable and tainted variable which means it affects program behavior of the page.

Example 1 of the above variables is shown in Figure 4, where quantity1 is the only input variable and the tainted variable of quantity1 consists quantity1, vquantity1, a, Session["total"]

and Total.Text. vquantity1 is the only variable which will influence the flow of Pay.aspx and hence is branch predicate variable. Therefore, in this simple example, vquantity1 is both branch predicate variable and an element of tainted variable, and hence is the tainted variable in branch predicate in collaborative testing on this shopping website.

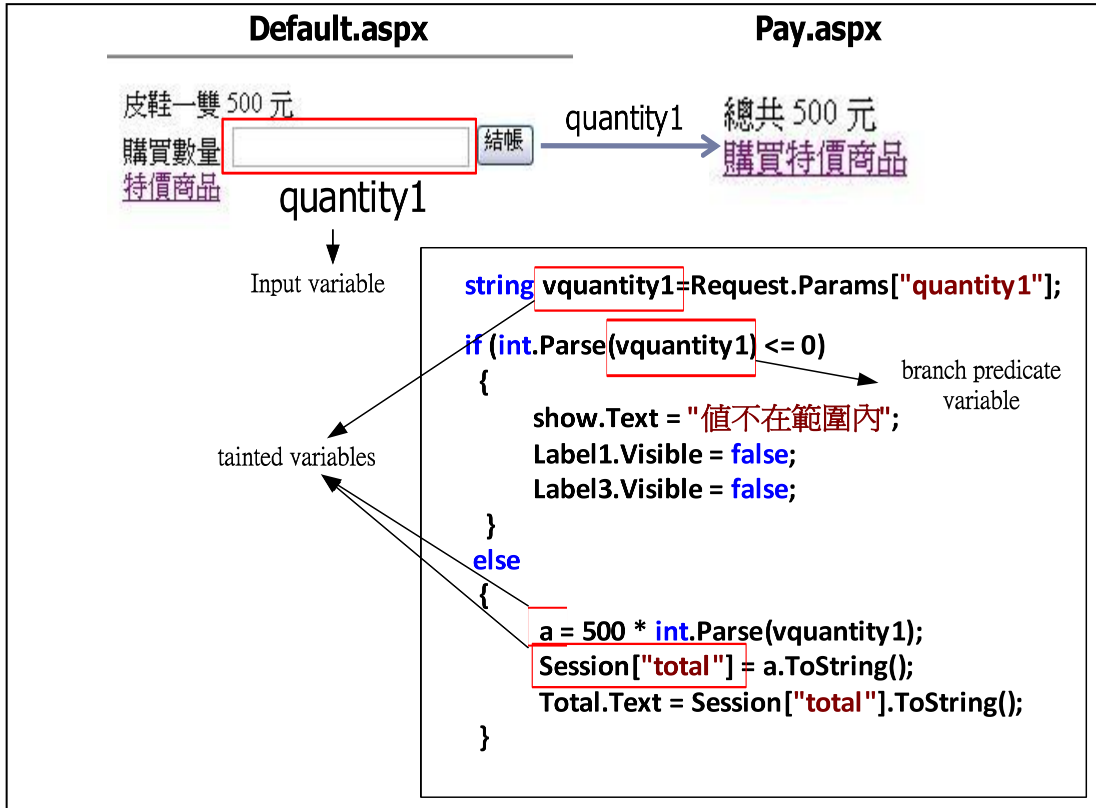


Figure 4. Differents of three variables in Example 1

3.2.2 Definition of Value- oriented dependence graph

Three-level dependence graph [28] is proposed to connect the perspectives from folk testers (Web pages), test objective from developer (basic blocks coverage). However, the lack of variable value information of three-level dependence graph cannot lead to fine collaborative testing. We further extend three-level dependence graph to value-oriented dependence graph by considering variable value information. There are three types of the vertexes (a set of pages, a set of functions, and a set of basic blocks) in the value-oriented dependence graph. And there are five types of the edges in the value-oriented dependence

graph. Three of them are intra-level edges representing the connections within page-level, function-level, and code-level. The other two are inter-level edges representing the connections between different levels. The formal definition of value-oriented dependence graph is provided below:

Definition 5: Value-oriented dependence graph

Value-oriented Dependence graph G= (V, E)

V_{IN}^{i} is an input variable

I= $\{V_{IN}^{i} | V_{IN}^{i}$ is an input variable$\}$ is the set of input variables

R= $\prod_{i=1}^{n} R(V_{IN}^{i}) = R(V_{IN}^{1}) \times R(V_{IN}^{2}) \times \ldots \times R(V_{IN}^{n})$, where $R(V_{IN}^{i})$ is an image of variable V_{IN}^{i}

$V = V_P \cup V_F \cup V_C$, where

$V_P = \{PA_i | PA_i$ is a page$\}$ is the set of the pages

$V_F = \{F_i | F_i$ is a function$\}$ is the set of the functions

$V_C = \{C_i | C_i$ is a basic block$\}$ is the set of the basic blocks

$E = E_P \cup E_F \cup E_C \cup E_{PF} \cup E_{FC}$, where

$E_P = \{(PA_i, PA_j, I_{ij}, R_{ij}) | PA_i, PA_j \in V_p, I_{ij} \in I, R_{ij} \in R\}$

$E_F = \{(F_i, F_j, I_{ij}, R_{ij}) | F_i, F_j \in V_F, I_{ij} \in I, R_{ij} \in R\}$

$E_c = \{(C_i, C_j, I_{ij}, R_{ij}) | C_i, C_j \in V_c, I_{ij} \in I, R_{ij} \in R\}$

$E_{PF} = \{(P_i, F_j) | P_i \in V_P, F_j \in V_F\}$

$E_{FC} = \{(F_i, C_j) | F_i \in V_F, C_j \in V_C\}$

Figure 5 shows value-oriented dependence graph in Example 1. Compared to three-level dependence graph, we further record tainted variable in branch predicate value information in edges. Different tainted variable in branch predicate values decide the permission of distinct edge and hence decide which basic blocks will be covered in different execution. The details of value-oriented dependence graph construction algorithm will be presented in the next section.

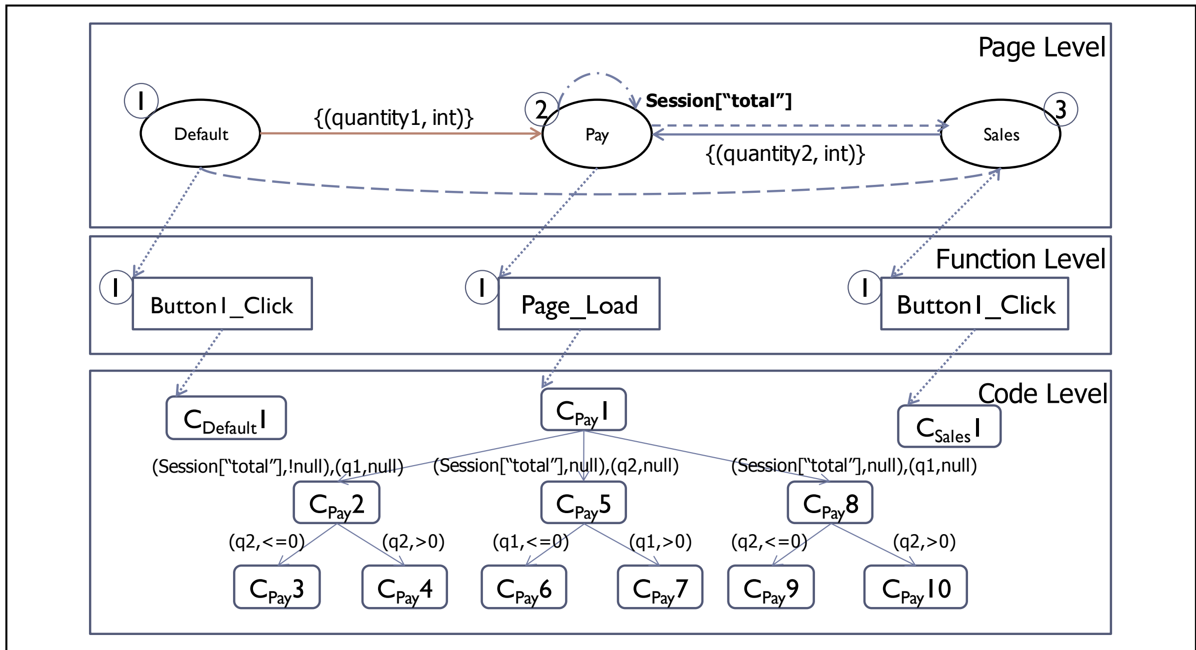


Figure 5. Value-oriented dependence graph in Example 1

3.2.3 Value-oriented Dependence Graph (VDG) construction algorithm

First, we will detect nodes of page-level, function-level, and code-level based on different pattern sets of each level, and then construct edges of each node in page-level, function-level, and code-level with the subroutine "EdgeConstruction" based on the pattern sets of each level. Finally, we construct the variables of each type with the subroutine "VariablesSetConstruction" based on the pattern sets of each type.

Value-oriented dependence graph construction algorithm

Input:

Folder which stores file of each page

Page code segments $B_a=\{B_a^1,\ldots,B_a^n\}$, $B_c=\{B_c^1,\ldots,B_c^n\}$

Pattern set of declare page P_P

Pattern set of declare function P_F

Pattern set of declare code P_C

Pattern set of declare input variables P_I

Pattern set of declare global variables P_G

Pattern set of declare branch predicate variables P_{BP}

T_P, where $T_P=P_P\cup P_F\cup P_C$

Output: Value-oriented Dependence graph，G=(V,E) , where $V=V_P\cup V_F\cup V_C$，$E=E_P\cup E_F\cup E_C\cup E_{PF}\cup E_{FC}$

Method:

Initial: $V=\varnothing,E=\varnothing$

Step1: For each line L in folder

If L contains pattern P_P then add the node into VP

Step2: For each B_c^i in B_c

2.1: For each line L in B_c^i

If L contains pattern P_F then add the node into V_F

2.2: Add edge from the page node to the function node

Step3: For each B_c^i in B_c

3.1: For each line in B_c^i

If L contains pattern P_C then add the node into V_C

3.2: Add edge from the function node to the basic block code

Step4:

4.1: For each B_a^i in B_a

EdgeConstruction(B_a^i,P_P)

4.2: For each B_c^i in B_c

EdgeConstruction(B_c^i,P_P)

Step5: For each B_F^i in V_F

EdgeConstruction(B_c^i,P_F)

Step6: For each B in VC

EdgeConstruction(B_c^i,P_C)

Step7: For each B_a^i in B_a

$I(V_i)$ = VariablesSetConstruction (B_a^i,P_I)

Step8: For each B_c^i in B_c

$G(V_i)$ = VariablesSetConstruction (B_c^i,P_G)

Step9: For each B_c^i in B_c

$T(V_i)$ = VariablesSetConstruction (B_c^i,$I(V_i)$)

Step10: For each B_c^i in B_c

$BP(P_i)$ = VariablesSetConstruction (B_c^i,P_{BP})

Step11: $TBP(P_i) = (\bigcup_{i=1}^{n} BP(P_i)) \cap (\bigcup_{j=1}^{m} T(V_j))$

Subroutine: EdgeConstruction

Input: Code segment B 、T_P

Output: Dependence graph of each level

Method:

Step1: For each line L in B

If L contains pattern T_P then construct edge of nodes

Step2: Return Dependence graph of each level

Subroutine: VariablesSetConstruction

Input: Code segment B 、T_P

Output: Set of variable S

Method:

Step1: For each line L in B

If L contains pattern T_P then add the variable into S

Step2: Return S

3.2.4 Example of Value-oriented dependence graph

At server side, there are various techniques to develop dynamic web page such as ASP.NET with C#, JSP, and PHP. At client side, HTML and JavaScript are used widely to support the development of the web applications. Hence, we only consider the ASP.NET with C# language, and the other language such as HTML, JavaScript can be considered by simple extension. In Table 1, according to different levels with page, function, and code we classify patterns of the nodes and the edges in the value-oriented dependence graph. In Table 2,

according to different variables with input, global, and branch predicate we classify patterns of these variables in the value-oriented dependence graph.

Table 1. Property of Patterns of Different level

Level	Pattern
Page Node	.aspx
Page Edge	Response.Redirect
Page Edge	NavigateUrl
Function Node	(protected| private | public)?[ ](void|bool|int|float)
Function Edge	Function caller
Code Node	(if|else|for|while|switch)
Code Edge	Flow chart

Table 2. Property of Patterns of Different variable

<table>
<tr><th>Level</th><th>Pattern</th></tr>
<tr><td>Input Variable</td><td><asp:TextBox ID=/"(a-zA-Z0-9)+/"</td></tr>
<tr><td>Input Variable</td><td><asp:DropDownList ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Input Variable</td><td><asp:ListBox ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Input Variable</td><td><asp:CheckBox ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Input Variable</td><td><asp:CheckBoxList ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Input Variable</td><td><asp:RadioButton ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Input Variable</td><td><asp:RadioButtonList ID=/"(a-zA-Z0-9)+ /"</td></tr>
<tr><td>Global Variable</td><td>Session[/ "(a-zA-Z0-9)+ /"]=</td></tr>
<tr><td>Global Variable</td><td>=Session[/ "(a-zA-Z0-9)+ /"]</td></tr>
<tr><td>Branch predicate variable</td><td>if((a-zA-Z0-9_>=<&|())+)</td></tr>
<tr><td>Branch predicate variable</td><td>while((a-zA-Z0-9_>=<&|())+)</td></tr>
<tr><td>Branch predicate variable</td><td>for((a-zA-Z0-9_>=<&|())+)</td></tr>
<tr><td>Branch predicate variable</td><td>do
while((a-zA-Z0-9_>=<&|())+)</td></tr>
<tr><td>Branch predicate variable</td><td>Switch
case 1: (a-zA-Z0-9_>=<&|())+
case 2: (a-zA-Z0-9_>=<&|())+
…
case i: (a-zA-Z0-9_>=<&|())+
…</td></tr>
</table>

The Example 2 is given to illustrate the value-oriented dependence graph construction algorithm, where Figures 6-1 to 6-11 show the results after executing steps 1 to 11, respectively.

Example 2:

The Example 2 continues Example 1. According to the source code of each page, “Property of Patterns of Different level” in Table 1, and “Property of Patterns of Different variable” in Table 2, we can construct the value-oriented dependence graph.

Since the pattern of page node is “.aspx”, after executing step 1, we can construct each page to the node in page-level as shown in Figure 6-1.

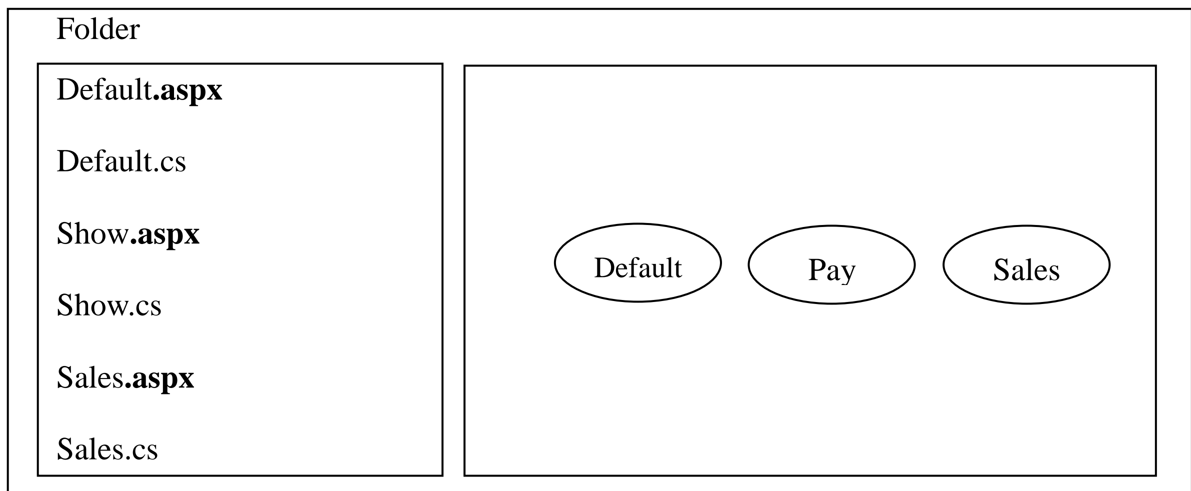


Figure 6-1. Value-oriented dependence graph after executing Step 1 of VDG algorithm

Since the pattern of function node is “(protected| private | public)?[ ](void|bool|int|float)” in Table 1, after executing step 2, we can construct each function for the given page to the node in function-level and connect the given page and the function with corresponding edge as shown in Figure 6-2.

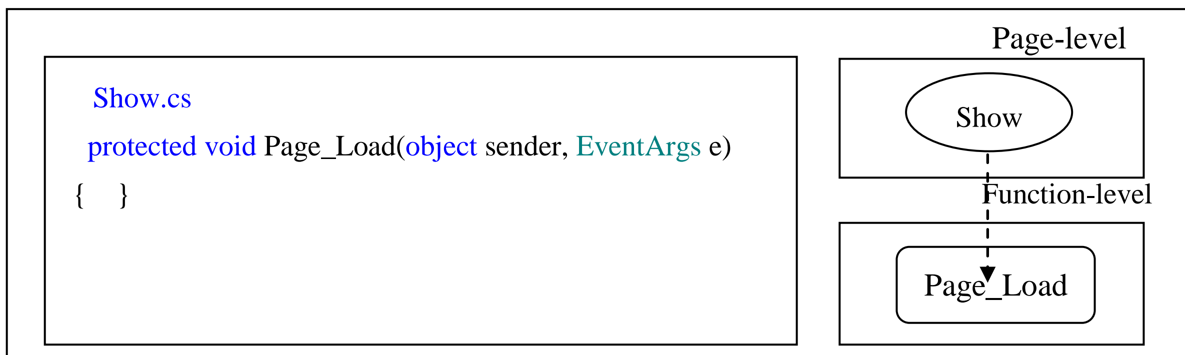


Figure 6-2. Value-oriented dependence graph after executing Step 2 of VDG algorithm

Since the pattern of code node is “(if|else|for|while|switch)” in Table 1, after executing step 3, we can construct each basic block for the given function to the node in code-level, and connect the function and the basic block with corresponding edge as shown in Figure 6-3.

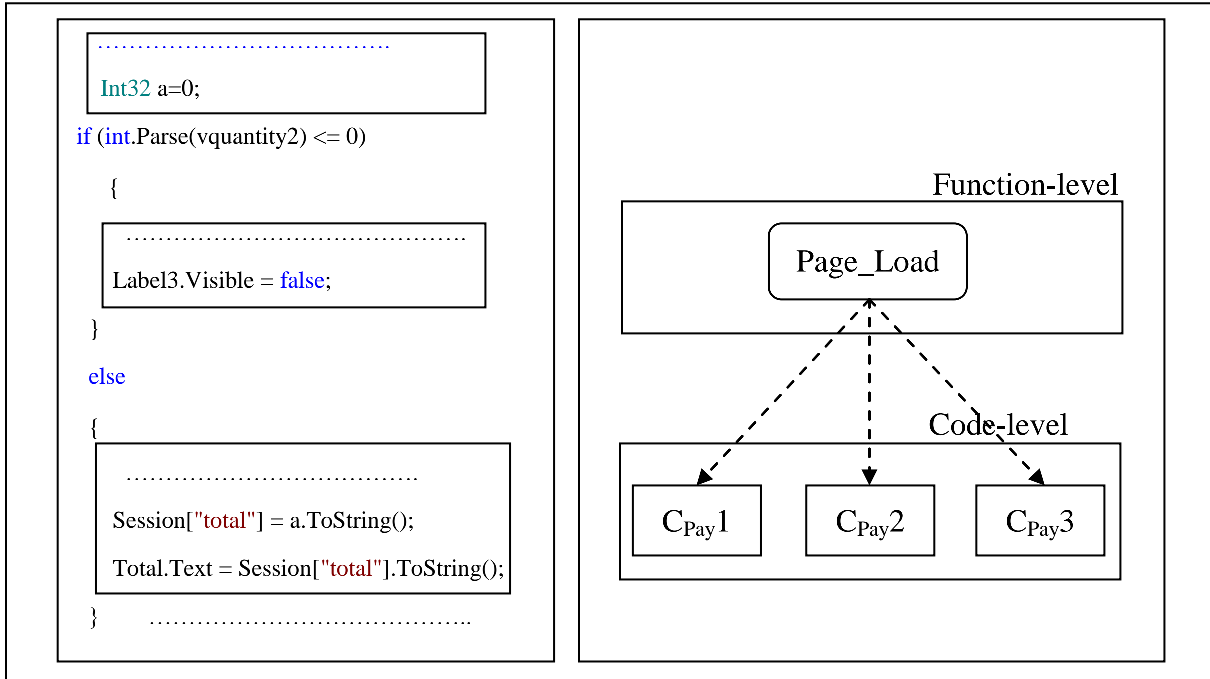


Figure 6-3. Value-oriented dependence graph after executing Step 3 of VDG algorithm

Since the pattern of page edge is “Response.Redirect” and “NavigateUrl” in Table 1, after executing step 4, we can connect nodes in page-level with corresponding edges as shown in Figure 6-4.

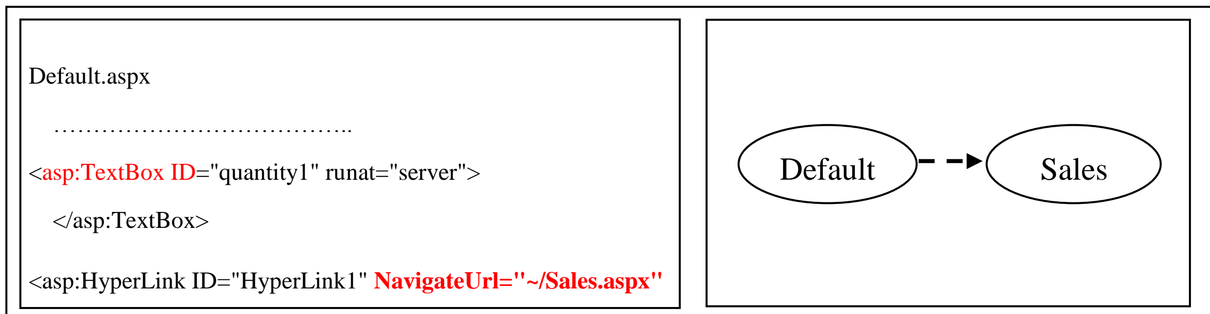


Figure 6-4. Value-oriented dependence graph after executing Step 4 of VDG algorithm

Since the pattern of function edge in Table 1, after executing step 5, we can connect nodes in function-level with corresponding edges as shown in Figure 6-5.

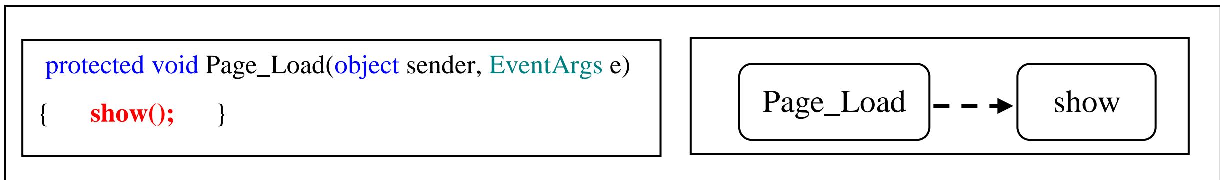


Figure 6-5. Value-oriented dependence graph after executing Step 5 of VDG algorithm

Since the pattern of code edge in Table 1, after executing step 6, we can connect nodes in code-level with corresponding edges as shown in Figure 6-6.

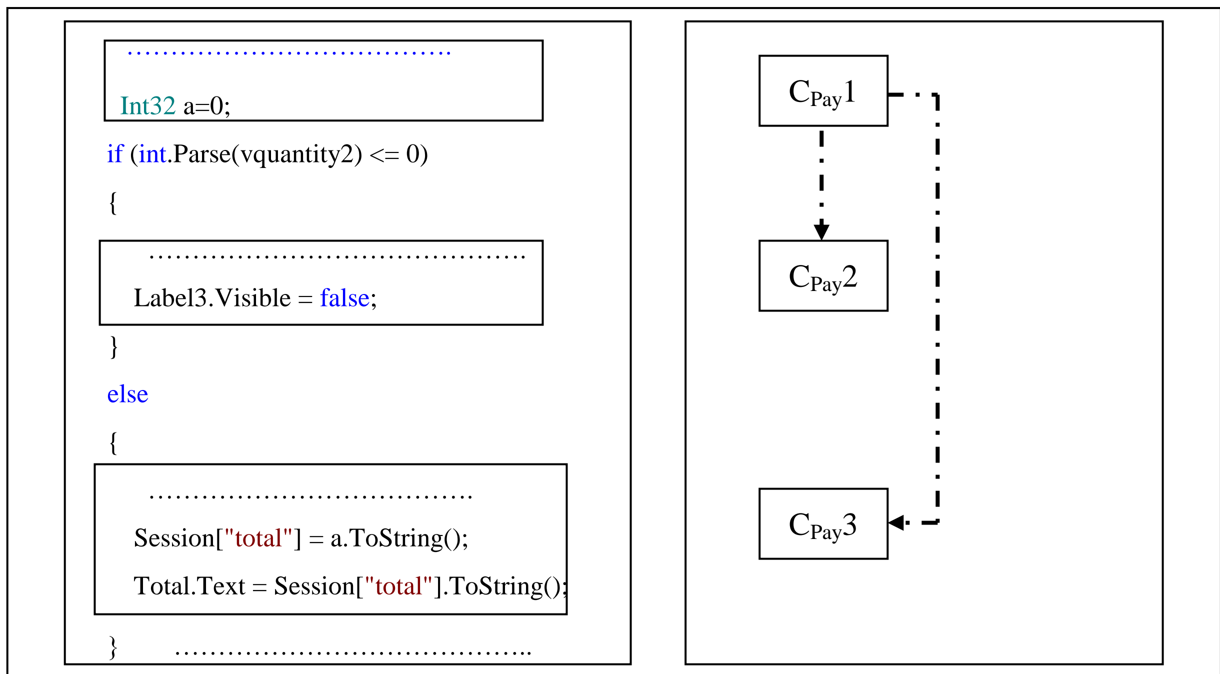


Figure 6-6. Value-oriented dependence graph after executing Step 6 of VDG algorithm

Since the pattern of input variable in Table 2, after executing step 7, we can construct input variables of each page as shown in Figure 6-7.

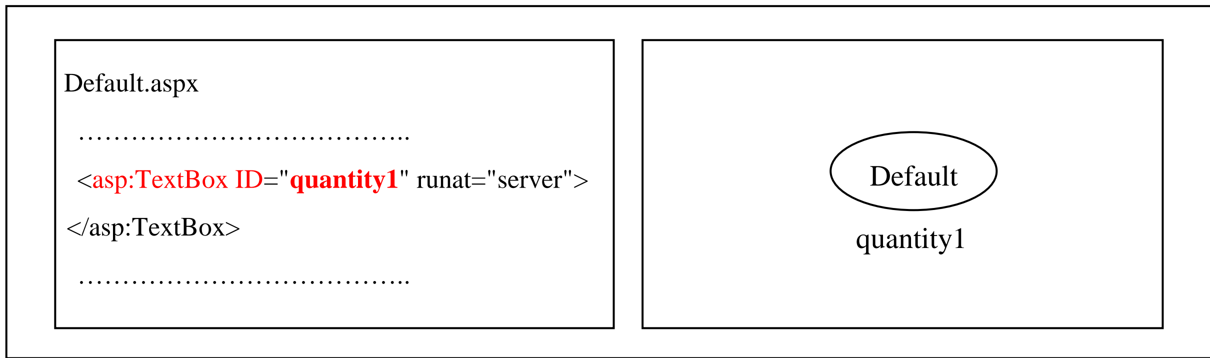


Figure 6-7. Value-oriented dependence graph after executing Step 7 of VDG algorithm

Since the pattern of global variable in Table 2, after executing step 8, we can construct global variables of each page as shown in Figure 6-8.

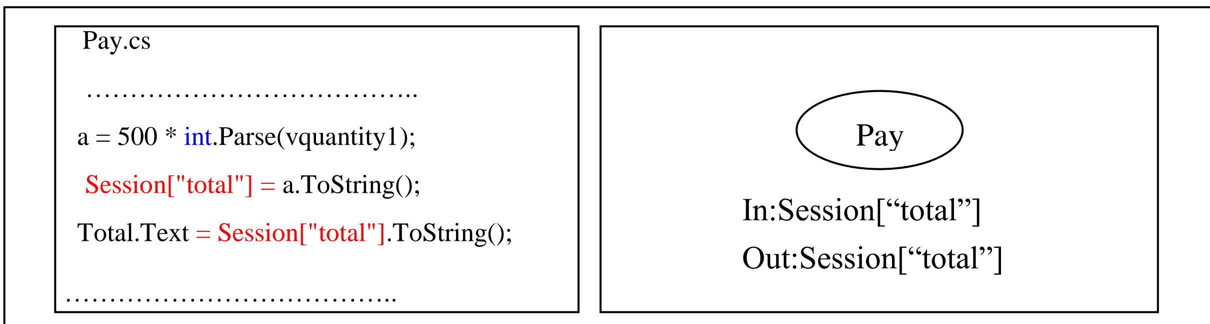


Figure 6-8. Value-oriented dependence graph after executing Step 8 of VDG algorithm

Since the pattern of input variable, after executing step 9, we can construct tainted variables of each page as shown in Figure 6-9.

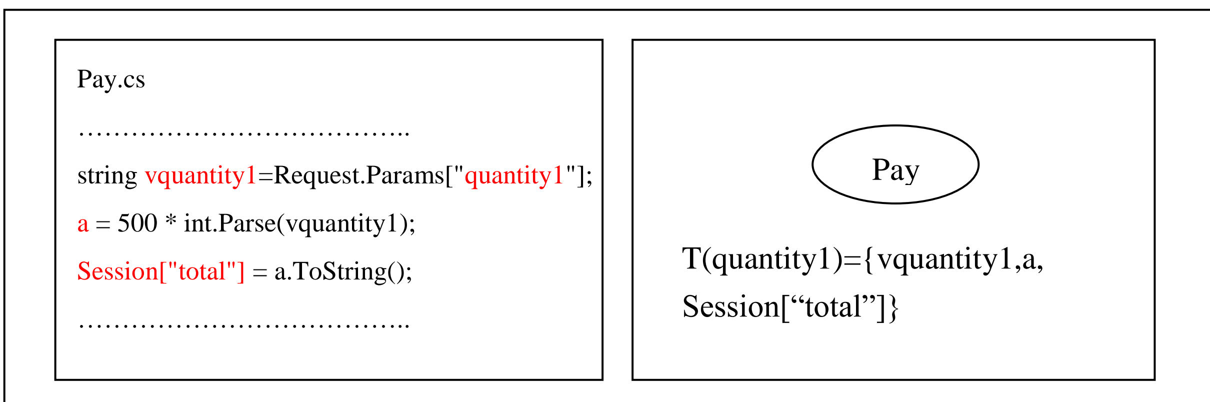


Figure 6-9. Value-oriented dependence graph after executing Step 9 of VDG algorithm

Since the pattern of branch predicate variables in Table 2, after executing step 10, we can construct branch predicate variables of each page as shown in Figure 6-10.

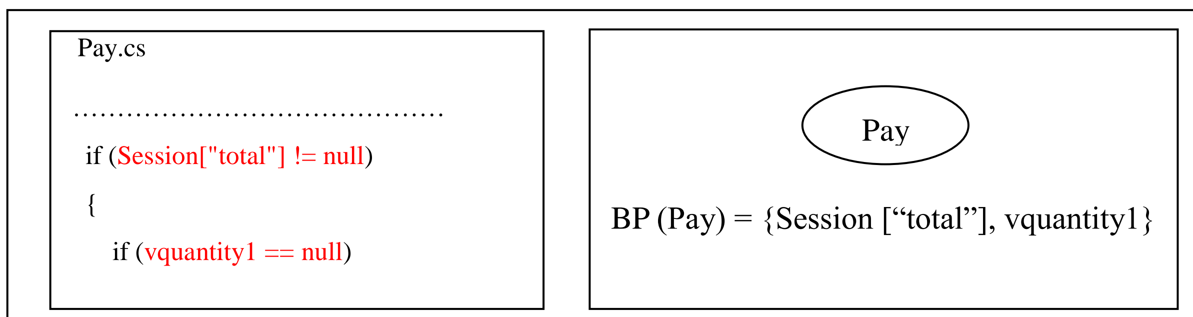


Figure 6-10. Value-oriented dependence graph after executing Step 10 of VDG algorithm

Since the branch predicate variables and tainted variable, after executing step 11, we can construct tainted variables in branch predicate of each page as shown in Figure 6-11.

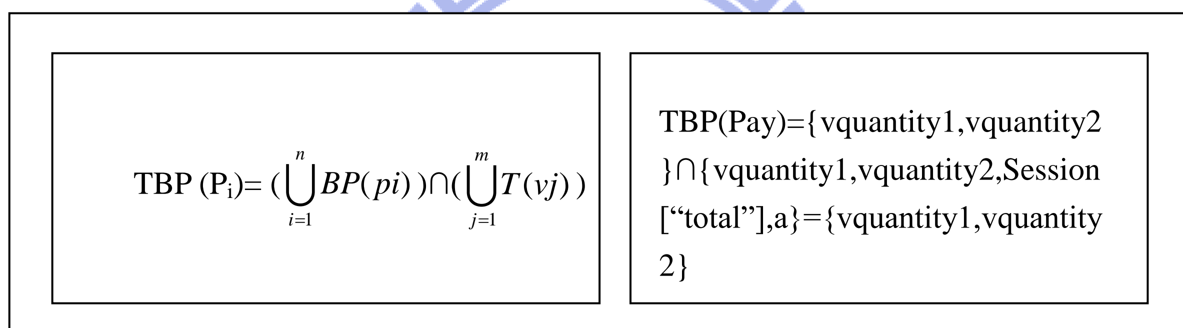


Figure 6-11. Value-oriented dependence graph after executing Step 11 of VDG algorithm

3.3 State Transition Diagram

In the section, we first introduce problem decomposition idea to improve collaborative testing, and then a novel Web application model, state transition diagram, and the corresponding construction algorithm will be presented in this section. An example of state transition diagram construction algorithm will be also provided in this section.

3.3.1 Problem decomposition

Value-oriented dependence graph provides an opportunity to refine collaborative testing. For further improving collaborative testing, we decompose the Web application testing problem into several sub-problems as shown in Figure 7. Each sub-problem which is easier

than original problem because of the smaller size of sub-problem can be independently solved. Afterwards, we can parallelly assign these sub-problems to different folk testers. This parallel property can speed up the whole collaborative testing. However, this parallel assignment requires a proper label design to integrate different test results for following job assignments and final test report. And then, we can easily merge these sub-problems by tracing back according to the labels of all sub-problems.

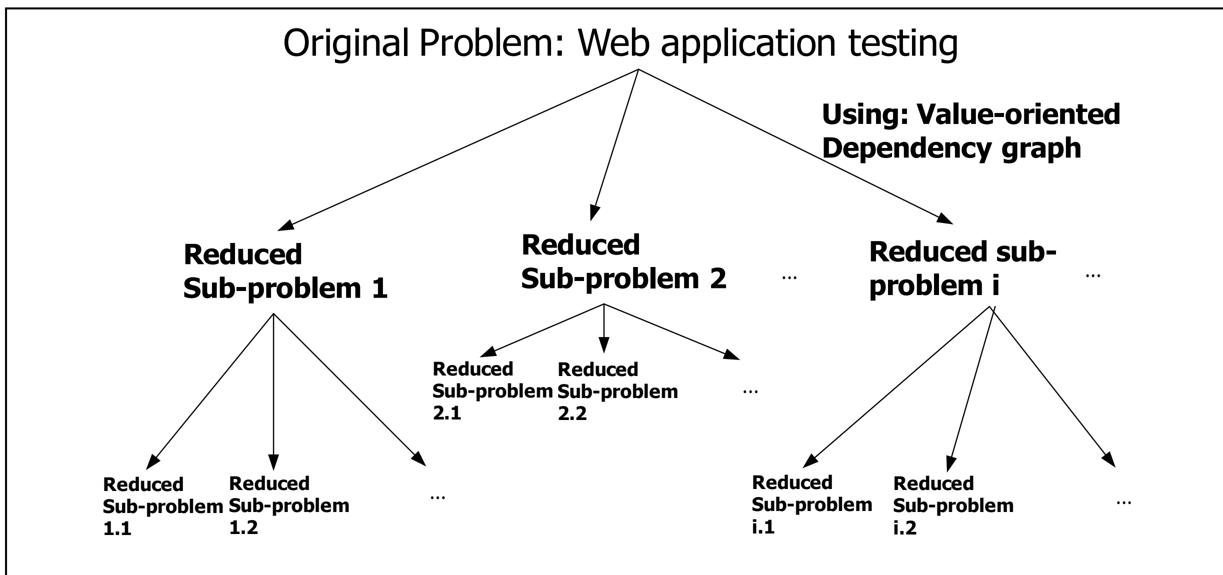


Figure 7. Idea of problem decomposition

3.3.2 Definition of State Transition Diagram

According to the idea of problem decomposition, we propose a novel Web application model, state transition diagram, which is a fine-grained tree structure model. And it also overcomes the cyclic problem in Web application testing under basic blocks coverage criteria. The state transition diagram models users' runtime behaviors according to different basic blocks coverage. Each state including two parts represents a behavior of the page. The first part is the page number, and the second part is the set of the tainted variables in branch predicate and its values which lead to different behaviors of program. The formal definition of state transition diagram is provided below:

Definition 6: State transition diagram

State transition diagram

V_{TBP}^{i} is a tainted variable in branch predicate

D= (S, δ, S_0), where

S = {S_i | S_i is a state}, where

S_i= (N, C), where

N ∈ { i | PA_i is a page}

C ⊆ {(V_{TBP}^{i}, R_i) | V_{TBP}^{i} ∈ V, R_i ∈ R, i=1~n}, where

V= {V_{TBP}^{i} | V_{TBP}^{i} is a tainted variable in branch predicate} is a set of tainted variables in branch predicate

R= $\prod_{i=1}^{n} R(V_{TBP}^{i}) = R(V_{TBP}^{1}) \times R(V_{TBP}^{2}) \times \ldots \times R(V_{TBP}^{n})$, where $R(V_{TBP}^{i})$ is an image of variable V_{TBP}^{i}

δ is a transition function

δ (S_i, A)=S_j, where

A ⊆ {(V_{TBP}, R_i)| V_i ∈ V, R_i ∈ R, i=1~n}

S_0 is an initial state

Figure 8 shows the state transition diagram in Example 1. Compared to value-oriented dependence graph, the tree structure provides a kind of problem decomposition. Each path from root represents a sub-problem of Web application testing. The details of state transition diagram construction algorithm will be presented in the next section.

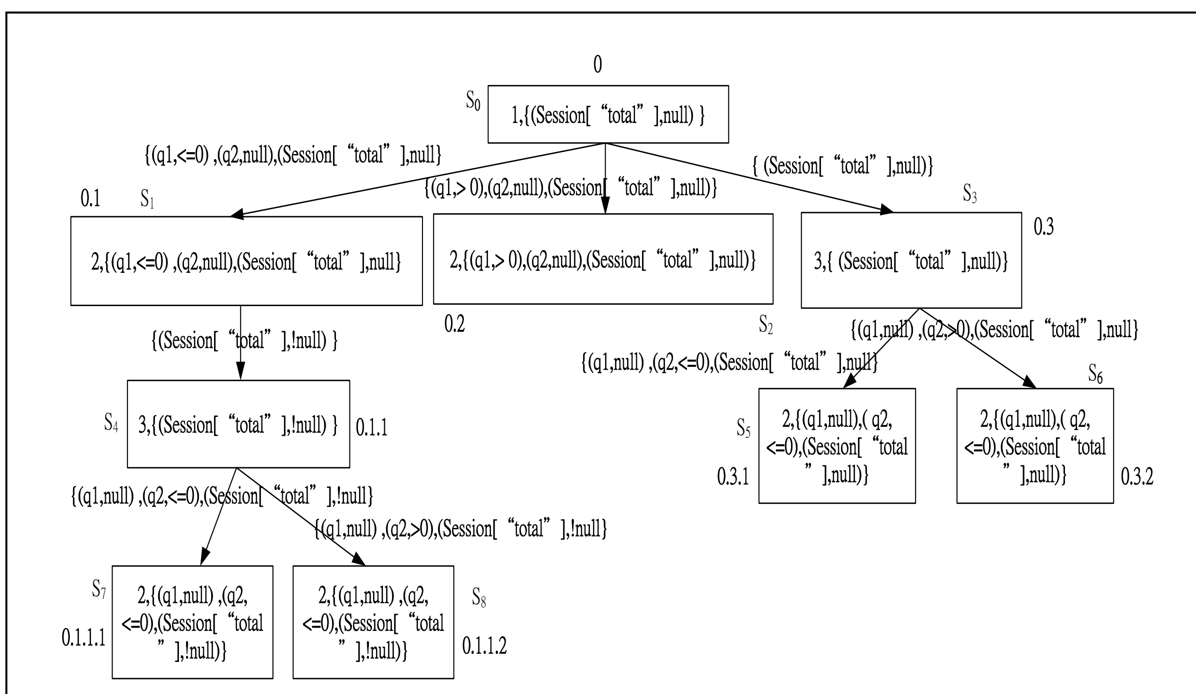


Figure 8. State transition diagram in Example 1

3.3.3 State Transition Diagram (STD) construction algorithm

From the purpose of speeding up collaborative testing, we first apply Breadth-First Search (BFS) on the page-level of value-oriented dependence graph to parallelly distribute folk testers. At each new visiting page, we further drill down to code-level of value-oriented dependence graph to identify different states based on distinct program flow. In the meanwhile, the proper label is created for these new states. The details of state transition diagram are presented below:

State transition diagram algorithm

Input:

Value-oriented Dependence graph G,

C[i]: tainted variable in branch predicate C of each page PA_i,

S_0:initial state which Label 0,

N[i]:start basic block of each page PA_i

color(i,c): node c is in code level of page PA_i

Output: State transition diagram STD D=(S, δ, S_0)

Method:

Initial: S=∅, δ=∅ , j=1, k=1

Step1: for each page number P which from small to large ∈ adj [S_0.N] in G

ConstructionNextState (S, P)

Step2: For each new state s' in S

2.1: k=1

2.2: for each page number P which from small to large ∈ adj[s'.N] in G

ConstructionNextState (s', P)

Step3: Repeat Step2 until there is no new state.

Subroutine: ConstructionNextState

Input:

S: State

P: Page number

Output: null

Step1: IF S_0 modify global value update global variable value of C[i]

ELSE update global variable value with S_0 of C[p]

Step2: add the variable of edge into C[p]

Step3: for each node c

Color (p,c)=white

Step4: StateCreation (S_0,p, C[p],N[p],0)

Step5: Change back global variable value of C[p]

Subroutine: IsANewState

```
Input:
        S_P: Previous state
        s: state
        i: Previous label
Output: null
Method:
Step1: IF the first part of s exists
            IF the second part of s exists
                    Remove the state s
            Else
                    Add s into S in S_j, j++
                    Add second part of S_j into edge from S_P to S_j
                    Label i.k, k++
```

Subroutine: StateCreation

```
Input:
        S_P:Previous state
        p: page number
        C[p]: tainted variable in branch predicate of p
        c_0: start basic block of each page
        i: Previous label
Output: null
Method:
Step1: IF number of out-degree of c_0 is zero or Color is black
      1.1: color (p, c_0)=black
      1.2: Add p into first part of S'
      1.3: Add coming edge condition into second part of S'
      1.4: IsANewState(S_P,S',i)
Step2: for each basic block c' in neighborhood of c_0
          IF condition of coming edge of c' accords with C[p]
                2.1: add condition of coming edge of c' into second part of S'
                2.2: color (p, c_0)=black
                2.3: StateCreation (S_P ,p, C[p],c',i)
```

3.3.4 Example of Constructing State Transition Diagram

The Example 3 is given to illustrate the state transition diagram construction algorithm, where Figures 9-1 to 9-2 show the result after executing steps 1 to 2, respectively.

Example 3:

The Example 3 continues Example 2. According to the value-oriented dependence graph, tainted variables in branch predicate of each page, and initial state, we can construct the state transition diagram.

Since the first part of initial state, we find outgoing edge in page-level of VDG, and then according to tainted variable in branch predicate of the page we search code-level of VDG, after executing step 1, we can construct the new state of initial state. Second, if the state doesn't exist then label and add to state transition diagram as shown in Figure 9-1.

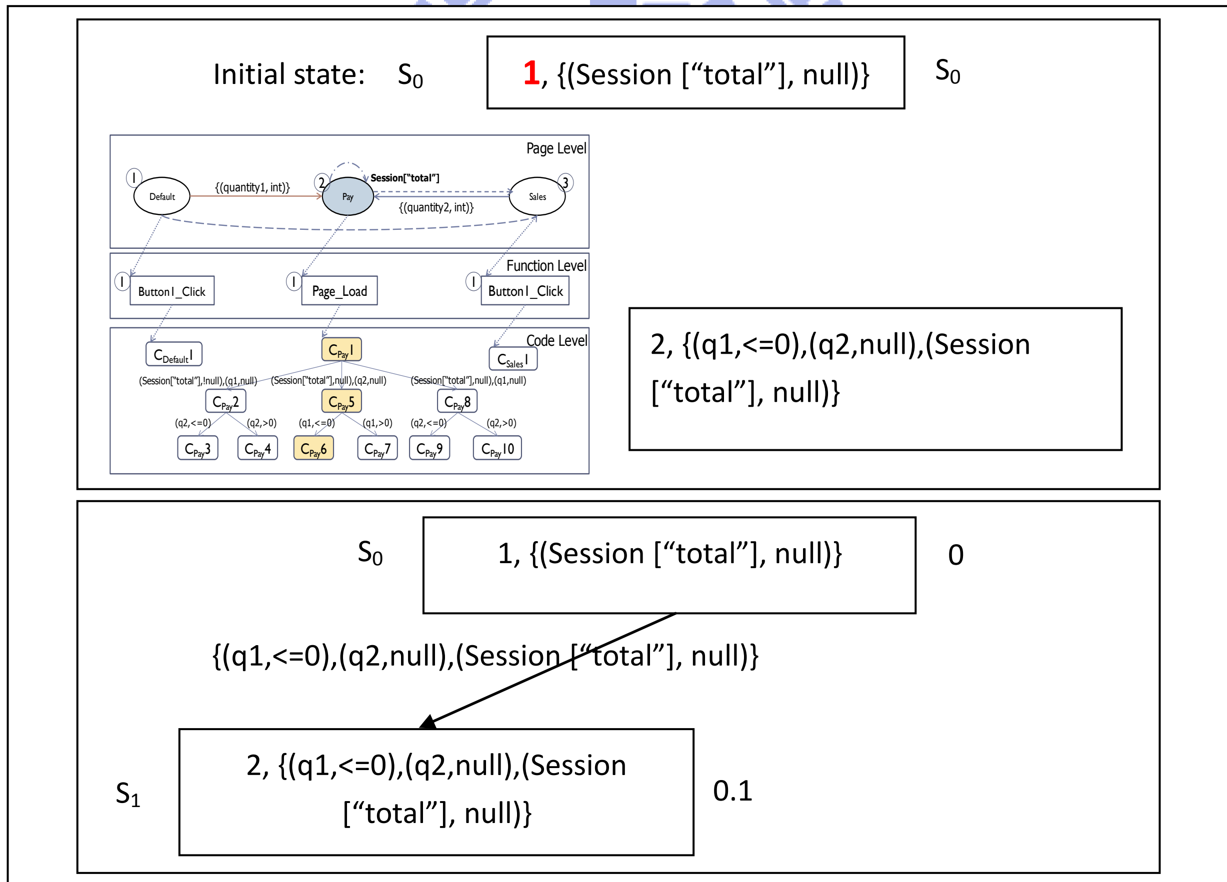


Figure 9-1. State transition diagram after executing Step 1 of STD algorithm

Since the first part of the given new state we find outgoing edge in page-level of VDG, and then according to tainted variable in branch predicate of the page we search code-level of VDG, after executing step 2, we can construct new state of the given new state. Second, if the state doesn't exist then label and add to state transition diagram as shown in Figure 9-2.

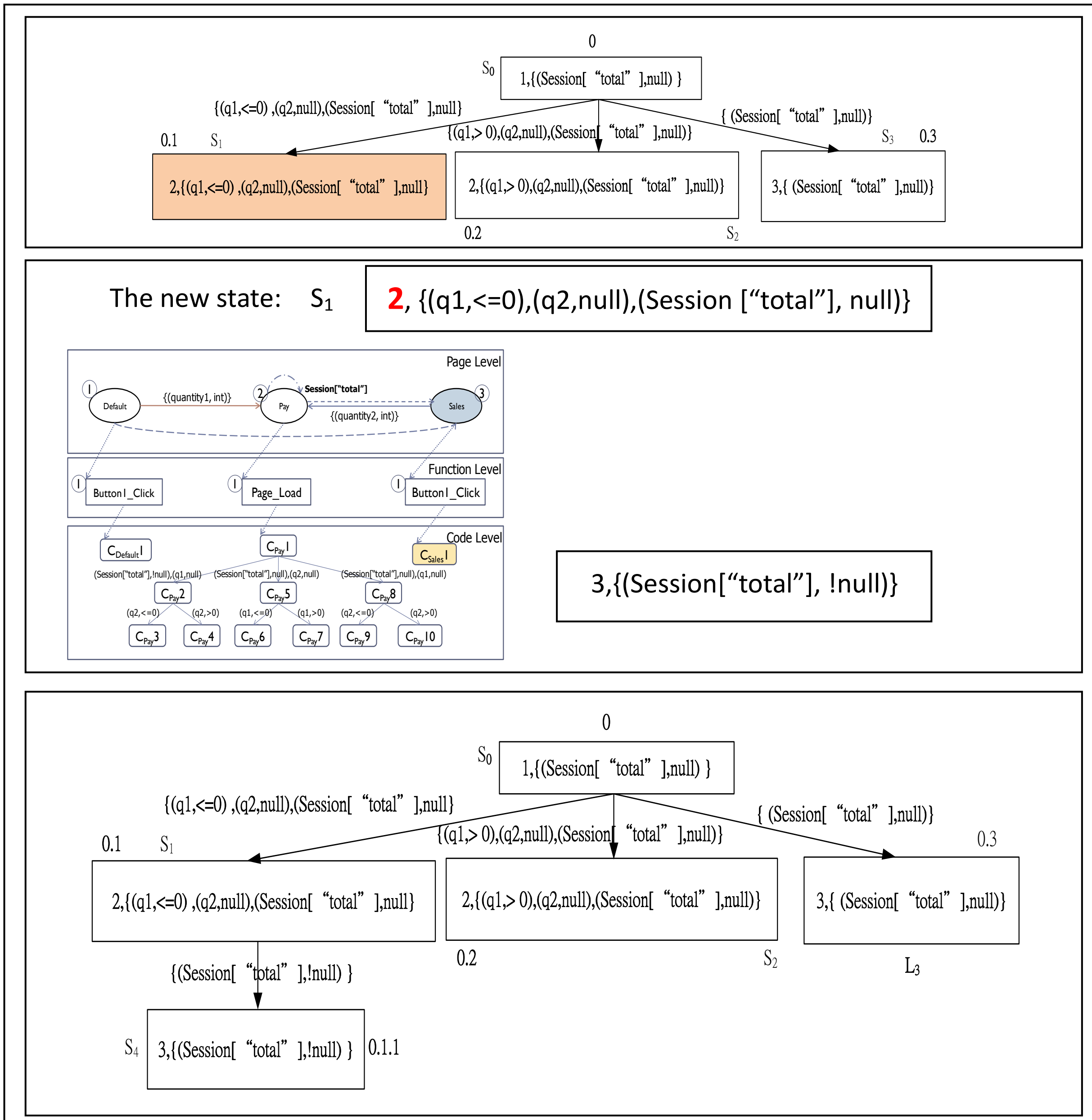


Figure 9-2. State transition diagram after executing Step 2 of STD algorithm

Repeat step 2 until has no new states and state transition diagram as shown in Figure 9-1.

Chapter 4 Dynamic stubbing technique for collaborative testing

In the chapter, we first provide an example to illustrate that different assignment leads to distinct test cost in collaborative testing. The formal problem formulation of Minimum Test Cost Problem (MTCP) in collaborative testing is presented in this chapter. We also prove MTCP is NP-complete and hence propose a heuristic-based dynamic stubbing algorithm to solve this optimization problem.

4.1 Motivating example

Figure 10 shows a simple Web application which contains nine states and eight sub-problems. Assume that each sub-problem needs to be executed once for completing test and three folk testers A, B, and C are involved in this test. The completion time of each sub-problem by different folk tester is listed in Table 3. The due time of collaborative testing is set to be the last folk tester finishing the assigned jobs.

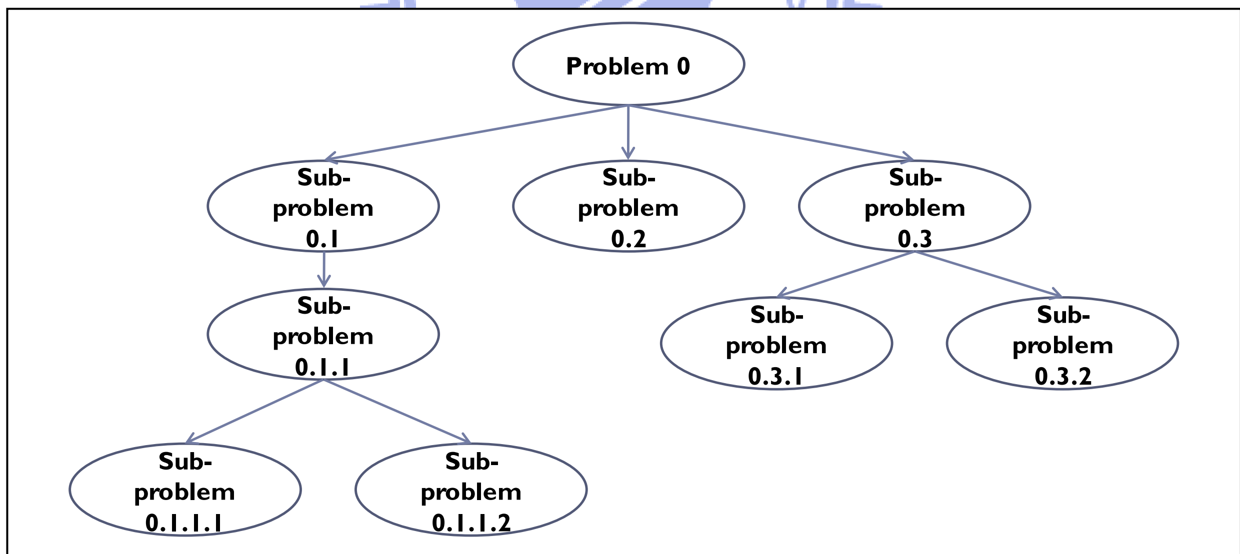


Figure 10. Problem decomposition scenario

Table 3. Sub-problem completion time of each folk tester matrix

	0.1.1.1	0.1.1.2	0.2	0.3.1	0.3.2
Tester A	160 sec	160 sec	80 sec	120 sec	120 sec
Tester B	165 sec	165 sec	85 sec	125 sec	125 sec
Tester C	170 sec	170 sec	90 sec	130 sec	130 sec

Two different assignments of the example are shown in Table 4 and Table 5. In each table, $entry_{ij}$=1 represents that the j_{th} job is assigned to i_{th} tester. Assignment 1 represents that three folk testers have centralized preference on sub-problem 0.1.1.1 and assignment 2 is unbiased on testers' preferences. The due time of these two assignments are 390 sec and 300 sec as shown in Figure 11 and 12. This example illustrates that different assignment leads to distinct test cost (due time) and the centralized preferences of folk testers will delay the whole collaborative testing.

Table 4. Assignment 1 for shopping Web-site testing

	0.1.1.1	0.1.1.2	0.2	0.3.1	0.3.2
Tester A	1	0	0	1	0
Tester B	1	1	0	0	0
Tester C	1	0	1	0	1

Table 5. Assignment 2 for shopping Web-site testing

	0.1.1.1	0.1.1.2	0.2	0.3.1	0.3.2
Tester A	1	0	0	1	0
Tester B	0	0	0	0	1
Tester C	0	1	1	0	0

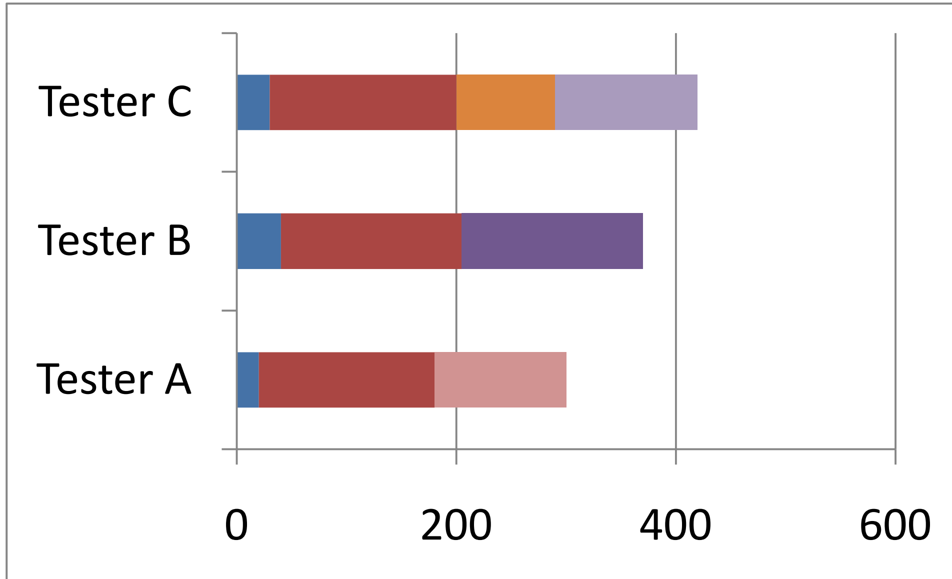


Figure 11. Due time of assignment 1

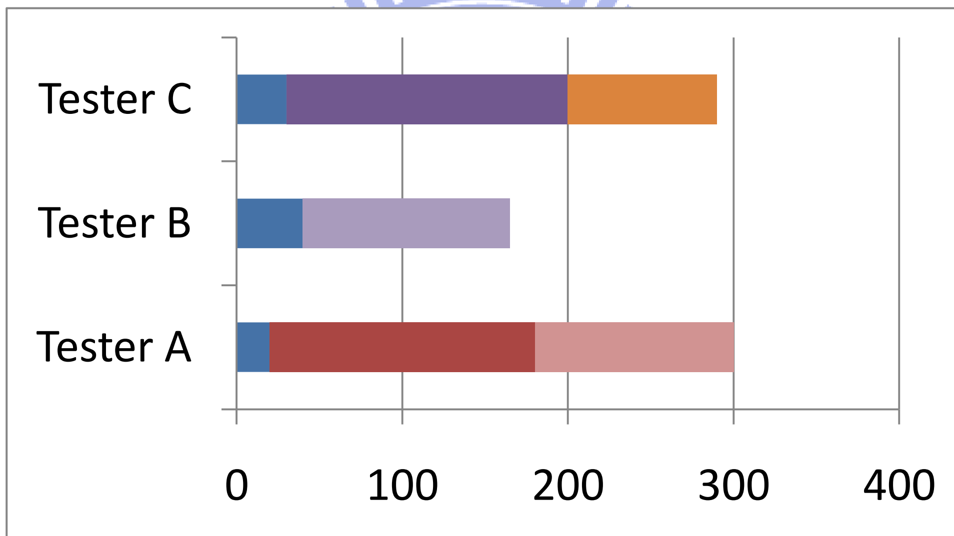


Figure 12. Due time of assignment 2

4.2 Minimum Test Cost Problem in Collaborative Testing

Under Internet environment, there are a large number of free and experienced folk human resources. Collaborative testing (or called Beta test) is usually used in online game and open source software to reduce test cost in software development stage. Based on the idea of beta test, we use folk testers in Internet to help us finding and reporting bugs. There are some constraints while applying collaborative testing. First, folk testers may not start to test at the

same time and they may delay to start the next test job after finishing current jobs. Second, we only consider those folk testers contributing themselves to collaborative testing. Therefore, each folk tester solves one sub-problem at least. Third, due to different complexity of each sub-problem, each sub-problem has distinct support threshold for completion. And then, the quality of folk testers (trustworthy) is different in Internet; therefore, it needs to consider the accumulation of the testers' trustworthy, rather than the number of testing. Final, each folk tester has different workload.

Before presenting our problem formulation, we first introduce the notations in Table 6. The variables i represents the i_{th} folk tester, the variable j represents j_{th} sub-problem, and the variable k represents k_{th} assignment of the sub-problems. We use the binary variable. σ_{ijk} equals to 1 represents that the j_{th} sub-problem is assigned to the i_{th} folk tester in the k_{th} assignment. Otherwise, σ_{ijk} equals to 0. $T_{De}(i, j)$ stands for the delay time of the i_{th} folk tester start to solve the j_{th} sub-problem, $T_{Ex}(I, j)$ stands for the j_{th} sub-problem execution time of the i_{th} folk tester, and T (i, j) stands for the j_{th} sub-problem completion time of i_{th} folk tester. W_i represents the trustworthy of i_{th} folk tester and H_i represents the workload of the i_{th} folk tester. ST_j represents the support threshold of j_{th} sub-problem.

Table 6. Notations of MTCP in collaborative testing

$i = i_{th}$ folk tester
$j = j_{th}$ sub-problem
$k = k_{th}$ assignment of the sub-problems
$\sigma_{ijk} = i_{th}$ folk tester does the j_{th} sub-problem in the k_{th} assignment
$T_{De}(i, j)$= the j_{th} sub-problem delay time of the i_{th} folk tester
$T_{Ex}(i, j)$= the j_{th} sub-problem execution time of the i_{th} folk tester
$T(i, j)$= the j_{th} sub-problem completion time of i_{th} folk tester
W_i= the trustworthy weight of i_{th} folk tester
H_i= the available time limit of the i_{th} folk tester
ST_j=support threshold of the j_{th} sub-problem

4.2.1 Problem formulation

The Minimum Test Cost Problem (MTCP) in collaborative testing can be formulated as IP-formulation. The objective function is the minimum of due time on different assignment, when due time is the maximum of the sum of sub-problem's time of each folk tester. The constraint 6 is the complete condition that each sub-problem need to be tested at least support threshold. The formulation of MTCP is presented below:

Definition 7: Minimum Test Cost Problem (MTCP) definition

Objective function:

$$\min_{k} \max_{i} \sum_{j} T(i, j)\sigma_{ijk}$$

Subject to:

1. $\sigma_{ijk} \in \{0, 1\}$, $\forall i, j, k$

2. $T_{De}(i, j) \in R^{+}$

3. $T_{Ex}(i, j) \in R^{+}$

4. $T(i, j) \in R^{+}$

5. $\sum_{j} \sigma_{ijk} \geq 1, \forall k$

6. $\sum_{i} W_i * \sigma_{ijk} \geq ST_j, \forall k$

7. $ST_j > 0$

8. $\sum_{j} T(i, j)\sigma_{ijk} \leq H_i, \forall k$

9. $H_i \in R^{+}$

10. $\sum_{i,j} (\sigma_{ijk} - \sigma_{ijk'})^2 \neq 0, \forall k, k', k \neq k'$

11. $0 < W_i \leq 1$

4.2.2 NP-Complete problem

In the section, we introduce an NP-complete problem, Job Assignment Problem (JAP) [30]. Then, JAP can be reduced in polynomial time to MTCP to complete the proof as followed.

The corresponding decision problem of Minimum Test Cost Problem (MTCP):

MTCP = $\{<D, J, U, ST, W, H, T, t>\}$
D= (S, δ, S_0) is a directed tree.
J is a set of sub-problems in D.
U is a set of folk testers.
ST is a function form $J \rightarrow R^+$
W is a function form $U \rightarrow (0, 1]$
H is a function form $U \rightarrow R^+$
T is a function form $U \times J \rightarrow R^+$
And there is an assignment with due time at most t

1896

Theorem 1

Minimum Test Cost Problem is NP-Complete.

Proof:

First, we show that MTCP belongs to NP. Given an instance of the problem, the verification algorithm checks that sum of trustworthy W_i of assigned sub-problems of every folk tester i of each sub-problem j exceeds the support threshold ST_j, the sum of the completion time of assigned sub-problems of each folk tester i does not exceed H_i, the assignment of the sub-problems differs from the other assignment of the sub-problems, and checks whether the maximum of the sum of the completion time of assigned sub-problems of each folk tester is at most t. This process can certainly be done in polynomial time.

Second, to prove that MTCP is NP-Hard, we show that JAP $\leq_p$ MTCP. Let G=(V',E'), J'=$\{P_i|i=1,\ldots,n\}$, U'=$\{U_i|i=1,\ldots,m\}$, d(i, j)=0, the confirm function S(k) $\forall k \in P_i$, the trustworthy function w by w(i, j) where i $\in$U'and j $\in$J', the human resource function H by H(i) $\forall i \in$U', the execution time function T by T(P_i, j) $\forall$Pi$\in$J', j$\in$U', and the maximum total cost at most t of JAP. We construct an instance of MTCP as follows. We form the tree D= (S, δ, S_0) where S=P_i, δ=0, S_0=P_i and we define the test sub-problem set J=J', the folk tester U=U', the support threshold function ST by ST_j=1 $\forall j \in J$, the trustworthy function W by W_i=w(i,j) $\forall i \in U$, the available time function H by H_i=H(i) $\forall i \in U$, the completion time function T by T(i, j)=T(Pi, j) $\forall i \in U$, $\forall j \in J$.

The instance of MTCP is then $<D, J, U, ST, W, H, T, t>$, where is easily formed in polynomial time.

We now show that graph G' has an assignment δ of the maximum total cost at t if and only if the tree D' has an assignmentσ' of the maximum due time at most t. Suppose there is an assignmentδwith maximum cost at most t. Therefore, there exists an assignment σ' such that $\sigma_{ijk} = 1$ if $\delta_{ij} = 1$, the support threshold ST_j=1 $\forall j \in J$, the trustworthy weight of folk tester W_i=w(i,j) $\forall i \in U$, the available time H_i=H(i), the completion time T(i, j)=T(P_i,j) $\forall i \in U$, $\forall j \in J$. Thus, the assignment σ' is feasible solution and the maximum due time is t. Conversely, suppose that there is an assignment σ' with the maximum due time is t. Then, there existed an assignment a such that $\delta_{ij} = \sigma_{ijk}$, the trustworthy w(i, j)=W_i $\forall i \in$U' and j$\in$J', the human resource H(i)=H_i $\forall i \in$U', the execution time T(P_i, j)=T(i,j) $\forall P_i \in$J', $\forall j \in$U'. Thus, the assignment is a feasible solution and the maximum total cost is t. Hence, MTCP is NP-Complete #

4.3 Dynamic stubbing algorithm for Minimum Test Cost Problem

Because Minimum Test Cost problem (MTCP) is an NP-Complete problem, we propose heuristic approach to solve MTCP. First heuristic is to assign new coming tester the job which requires the most effort to compete. This heuristic can speed up the whole testing. However, the sub-problem completion time of different folk testers is distinct. Therefore, we further predict the completion time to balance the following job assignments. Second heuristic is to assign tester with high trustworthy the most doubting job. Since the quality of folk testers is not the same, there may be opposite result on the same sub-problem. The second heuristic is used to improve the quality of test report.

Before presenting our heuristic-based approach, the used notations are introduced in Table 7. U represents the set of the folk testers, and W_i represents the trustworthy of the i_{th} folk tester. M_j represents the complexity of j_{th} sub-problem. S_j represents the testing support of the j_{th} sub-problem and ST_j represents the support threshold of the j_{th} sub-problem. T_A (i, j) stands for the actual execution time of the j_{th} sub-problem done by the i_{th} folk tester, and T_{Ev}(i, j) stands for the evaluation time of the j_{th} sub-problem done by the i_{th} folk tester. F_P (S_k) represents the average trustworthy of folk testers which report bugs at the state k, and F_N (S_k) represents the average trustworthy of folk tester which don't report bugs at the state k. The details of these notations and the following proposed dynamic stubbing algorithm are listed below:

Table 7. Notations of Dynamic stubbing algorithm

Folksonomy user:

$U = \{i \mid i \text{ is the } i_{th} \text{ tester}\}$ is a set of folk testers

Trustworthy weight of the i_{th} folk tester:

$W_i \in (0, 1]$

Complexity:

M_j=#line of code in sub-problem j

Support of the j_{th} sub-problem:

$S_j = \sum_i (T_A(i,j) - (T_{Ev}(i,j)) * W_i)$

Support threshold weight of the j_{th} sub-problem:

$ST_j = M_j * c$, where c is a constant

Actual execution time of the j_{th} sub-problem, which had done by the i_{th} folk tester:

$T_A(i, j) \in R^+$

Evaluation time of the j_{th} sub-problem by the i_{th} folk tester:

$T_{Ev}(i, j) \in R^+$

Average trustworthy weight of folk testers which report bug at the state k

$F_P(S_k) = \frac{1}{|U_t|} \sum_{i \in U_t} W_i$, where U_t is the set of folk tester report bugs at the state k

Average trustworthy weight of folk testers which don't report bug at the state k

$F_N(S_k) = \frac{1}{|U_t|} \sum_{i \in U_t} W_i$, where U_t is the set of folk tester don't report bugs at the state k

Dynamic stubbing algorithm

Input:

User Profile

$T_{Ev}(i, j)$:evaluation time of sub-problem j of tester i

State Transition Diagram $D=(S, \delta, S_0)$

S (PR): a set of all sub-problems

PR: a set of sub-problems

Output:

Testing time

Method:

Initial: PR=null, for each sub-problem j in S (PR) InitialMetadata(j)

Step1: for each sub-problem in S (PR) find the set of sub-problem S (PR) which the most required tested

1.1: PR=FindTestedProblem(S (PR))

Step2: GuidingTester(PR,U_i)

Step3: IF the j_{th} sub-problem of the i_{th} tester has done

3.1: updating $S_j=S_j + (T_A(i, j)-T_{Ev}(i, j))*W_i$

Step4: IF ($S_j>ST_j$, $\forall_j$)

Testing Finish

ELSE

Go to step1

Subroutine: InitialMetadata

Input: Sub-problem j

Output: Sub-problem j'

Method:

Step1: According to the code complexity of j set up the ST_j of sub-problem j

ST_j <- X

S_j <- 0

Step2: return j'

Subroutine: FindTestedProblem

Input: S (PR): a set of all sub-problems

Output: a set of sub-problem PR

Method:

Initial: B=S (PR)

Step1: for each problem j in B

1.1: T (j) = (length (j) +1) [$(S_j/ST_j) + \alpha \sum_{j.k \in sub-problem(j)} T(j)$]

Step2: for n=1 to count (U_i)

2.1: add arc $\min_{j}\{T(j)\}$ into PR

2.2: remove j

Step3: return PR

Subroutine: GuidingFolkTester

Input:

State transition diagram D

F: all user profile

A set of sub-problem PR

A set of folk tester U_i

Output:

Method:

Initial: B=PR

Step1: for each sub-problem j in PR

Return arc $\min_j\{\sum_k (F_P(S_k) - F_N(S_k)) \mid S_k \in j\}$

Step2:

2.1: IF B≠null

Assign j to the most trustworthy of tester i in U_i

ELSE

Assign arc $\max_j\{length(j)\}$ to the tester i in U_i

2.2: $S_j=S_j+T_{Ev}(i,j)*W_i$

Step3: According to j from left to right

3.1: apply D and F to assign input value which is not tested to the Tester

3.2: for each link L in the tested page

Block link button except the link of entering to the next tested page.

3.3: remove i from U_i

3.4: remove j from B

Step4: repeat step1 until U_i is null

Chapter 5 Implementation and Experiment

5.1 System architecture and implementation

5.1.1 System architecture

In this section, we propose our two-phase collaborative testing system architecture including preprocessing phase and testing phase, as shown in Figure 13. In phase I, we convert the ASP.NET with C# language of the web applications into value-oriented dependence graph by value-oriented dependence graph construction algorithm. And then, we transform value-oriented dependence graph into state transition diagram for problem decomposition by considering users' runtime behaviors. In phase II, according to the state transition diagram, the tester profiles and portfolios, dynamic stubbing algorithm assigns jobs to each new coming tester. After the testing finishing, we analyze these bugs which folk testers reported based on report analysis algorithm.

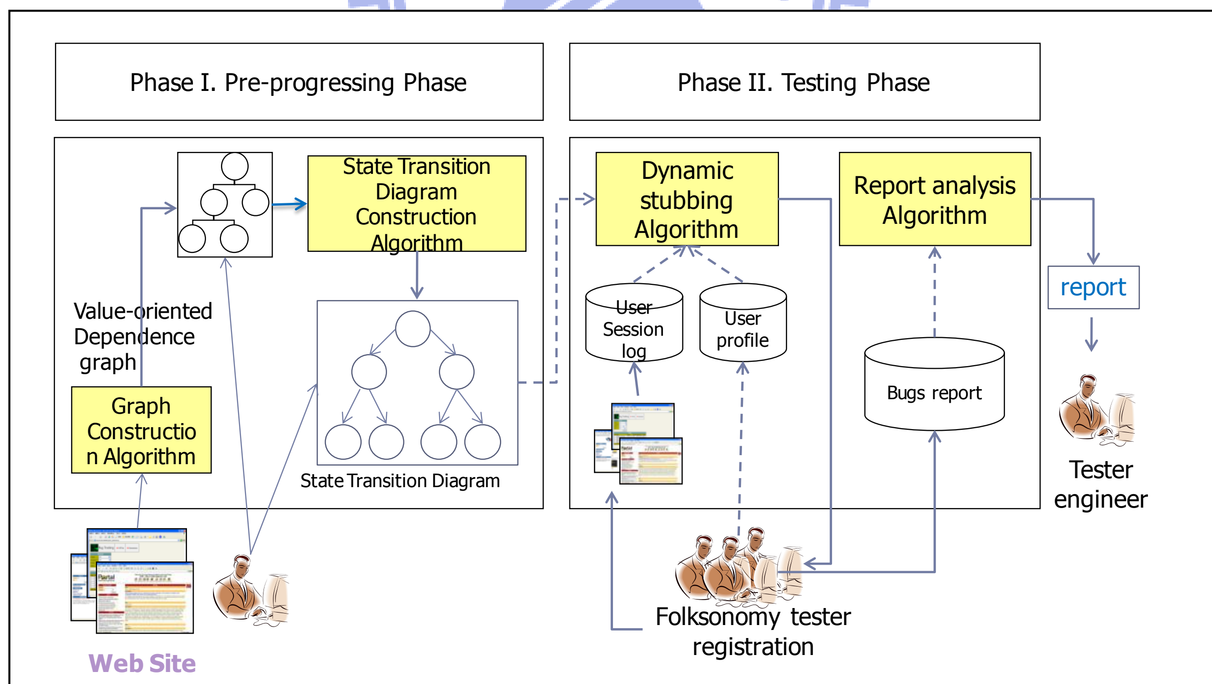
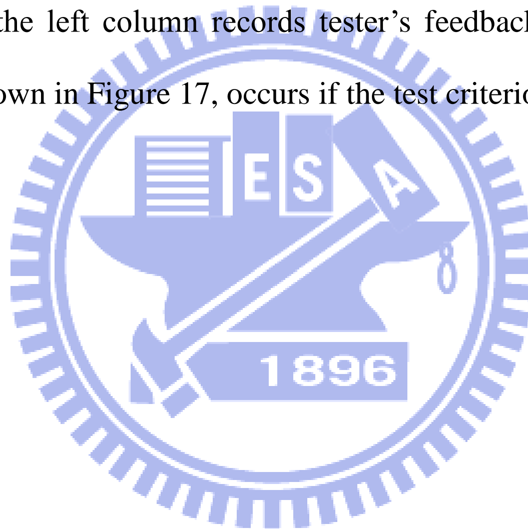


Figure 13. Intelligent collaborative testing system architecture

5.1.2 System implementation

We implement a prototype system, intelligent collaborative testing system (ICTS), to support collaborative testing. When using ICTS, new tester has to register by filling out a simple questionnaire. This questionnaire can be used to estimate the trustworthy of tester and the completion time of each sub-problem. The screenshot of register page of ICTS is shown in Figure 14. The tutorial of ICTS, shown in Figure 15, is provided for testers to understand how to test on ICTS. After understanding how to test on ICTS, testers can start to test Web-site. ICTS guides testers to test state of the most required and sets all values of tainted variables in branch predicate for testers. The guide of ICTS, shown in Figure 16, the main frame is the current tested page and the left column records tester's feedback on current page. Finally, complete information, shown in Figure 17, occurs if the test criterion is met.



1.

測試者請先登入，還沒有註冊帳號請先註冊

帳號: 密碼: 登入 註冊

Register testing account

帳號：May

密碼：xxxx

確認密碼：xxxx

平均每天上網時間：3小時~5小時

職業：研究生 請正確填寫

專長：打電腦 請正確填寫

性別：女

年齡：20~25

註冊

2.

測試者請先登入，還沒有註冊帳號請先註冊

帳號: May 密碼: xxxx 登入 註冊

註冊成功

Register successfully and login

Figure 14. Screen shot of register page of ICTS

Figure 15. Screenshot of ICTS tutorial

Figure 16. Screenshot of ICTS guiding

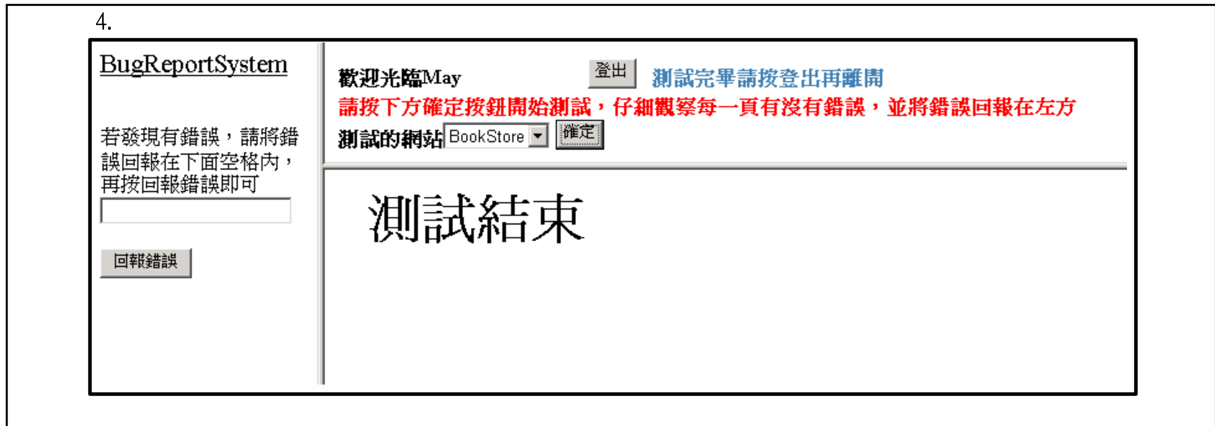


Figure 17. Screenshot of complete information of ICTS

5.2 Experimental design and results

5.2.1 Experimental design

We demonstrate our proposed approach on a open source Web application "BookStore" [http://www.gotocode.com/], which contains nine pages (AdvSearch page, BookDetail page, Books page, Default page, Login page, MyInfo page, Registration page, ShoppingCart page, and ShoppingCartRecord page). We convert "BookStore" into value-oriented dependence graph by the value-oriented dependence graph construction algorithm, and then transform value-oriented dependence graph into state transition diagram. The corresponding statics of value-oriented dependence graph of "Bookstore" are provided in Table 8, respectively. And then we transform value-oriented dependence graph into state transition diagram with 28 states.

Table 8. Value-oriented dependence graph statistics

Level	Type	Quantity
Page Level	Node	9
	Edge	52
Function Level	Node	146
	Edge	136
Code Level	Node	351
	Edge	546
Total	Node	506
	Edge	734

In the experimental design, we design a real testing environment of the web application "BookStore". The folk testers of this experiment are gathered via social network sites such as Facebook and msn. Therefore, these folks have basic internet access skills. The ages of testers are between 15 and 30. These folk testers are further split into control group (85 testers) and experimental group (59 testers). The testing periods of control group and experimental group are 2011/6/1~2011/6/6 and 2011/6/1~2011/6/3, respectively. The stopping criterion of these tests is that the support of each state exceeds its support threshold.

5.2.2 Experimental results

Experimental result I-Efficiency evaluation

Centralized preferences of folk testers are the major cause of the delay of collaborative testing. Figure 18 shows that folk testers in control group prefer to test the first state and then second, 6th, 10th, 14th, 20th, 21th and 22th state. Compared to control group, our proposed intelligent collaborative testing system can reduce this kind of bias. Table 9 shows that experimental group has much less standard deviation in state complete degree than control group (0.2928 v.s. 0.9268). This points out that our proposed algorithm can balance the job assignment.

Table 9. State complete degree of testing comparing

Group	Mean	S.D.
Experimental group	0.6907	0.2928
Control group	0.9404	0.9268

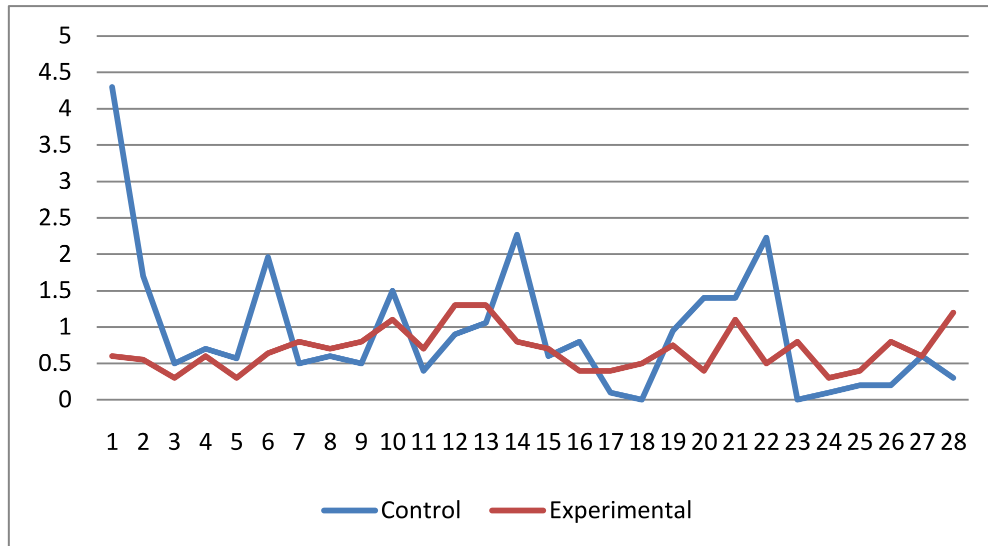


Figure 18. Comparison of state complete degree of testing

Table 10 shows the average contribution (test time) of each folk tester on this experiment. We further apply independent t-test on control group and experimental group. There is no significant difference (p=0.1240) in contribution of each folk tester between control and experimental groups. In fact, testers in experimental group contribute less. Table 11 shows the comparison of unit number of online folk testers of control group and experimental group. This result shows that the contributions of testers in unit time of two groups are the same. According to the above results, these experiments are fair for experimental group and control group. Based on these fair comparisons, the due time of the experimental group can be reduced to 50% of the control group, shown in Figure 19. Hence, our proposed dynamic stubbing algorithm can speed up collaborative testing.

Table 10. Comparison of test time between two groups

Group	N	Mean	S.D.	p-value
Experimental group	59	150.6271	236.1593	0.1240
Control group	85	244.5647	482.3225	

Table 11. Comparison of the folk testers per due time

	Control group	Experiment group
Folk testers/Due time	0.0050	0.0061

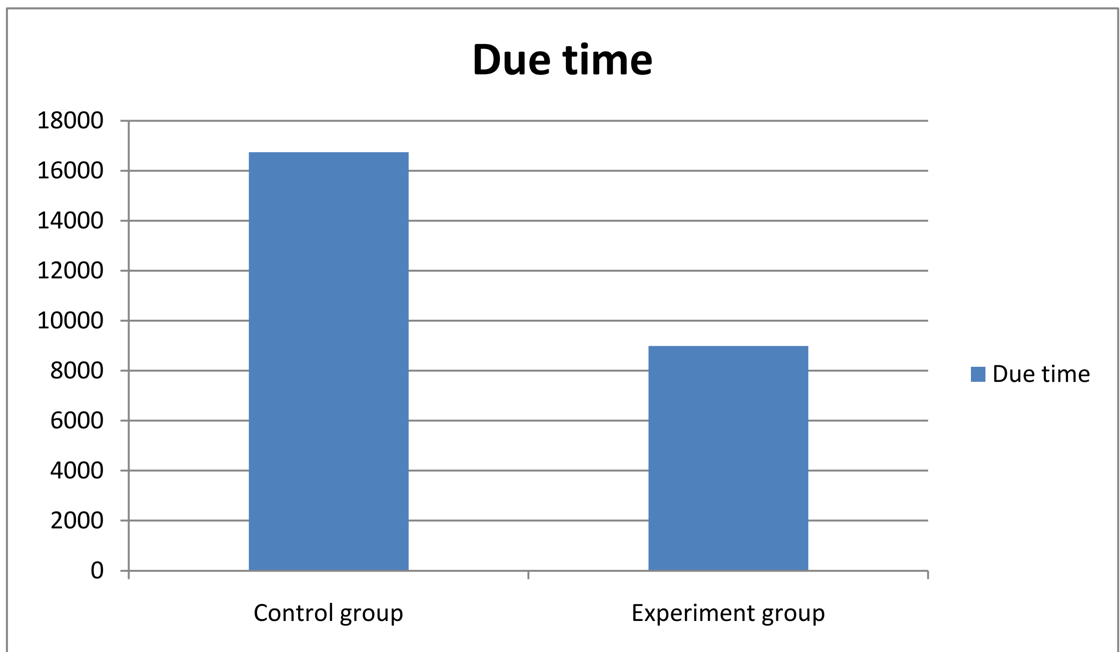


Figure 19. Due time comparison

Experiment result II -Effectiveness evaluation

Fault-detection ability is the most important in software testing, especially in collaborative testing. Unqualified folk testers may produce doubt reports, and hence it requires advanced job assignment and report analysis algorithm to improve the quality of final report. In this experiment, there are ten defects in Web application "BookStore" as shown in Table 12, including the same book image error, notes of book error, category shows error, vote image error, vote rate error, E-mail error, total price error, total price of book error, modify quantity error, and last_name and E-mail error.

Table 12. Ten defects in Web application "BookStore"

	Defect description	Page
1	the same book image error	Default
2	notes description of book error	Default
3	category shows error	Books
4	vote image error	BookDetail
5	vote rate error	BookDetail
6	E-mail error	ShoppingCart
7	total price error	ShoppingCart
8	total price of book error	ShoppingCart
9	modify quantity error	ShoppingCartRecord
10	last_name and E-mail error	MyInfo

Web application testing problem can be treated as binary classification problem where pages with defect are positive instances and normal pages are negative instances. Each folk tester can be considered as a classifier and our proposed system is an advanced classifier integrating every classifiers. Table 13 shows the comparison of processing time of each folk

tester on each state. Folk testers in experimental group spend less time to check the status of each state. However, there is no significant difference (p=0.0845) between control and experimental groups when applying impendent t-test. The proxies of fault-detection ability are selected as true positive rate (TP), true negative rate (TN), false positive rate (FP), false negative rate (FN), precision, recall and F-measure. Table 14 shows the comparison of the fault-detection ability between control group and experimental group. There is no significant difference in TP (p=0.4170), TN (p=0.6019), FP (p=0.8474), FN (p=0.5518), Recall (p=0.6655), Precision (p=0.9321), and f-measure (p=0.7233) between two groups. These results indicate that folk testers in control group and experimental group have similar fault-detection ability. However, the large stand derivation of each proxy indicates that the quality of folk testers varies much. For example, there are almost 70% testers with fault-detection ability (precision) from 0 to 1. This indicates that there exists folk testers with prefect fault-detection ability and none fault-detection. From this observation, it requires report analysis algorithm to form high quality test report.

When forming the final report, our report analysis algorithm adopts winner-takes-all strategy, where the prediction of each state relies on the group with higher average trustworthy. Based on the above comparisons, control group and experimental group have similar fault-detection ability. Table 15 shows that our proposed approach can improve 30% fault-detection ability than traditional collaborative testing.

Table 13. Comparison of test time of each state between two groups

Group	N	Mean	S.D.	p-value
Experimental group	59	22.0596	17.9834	0.0845
Control group	85	29.5177	33.1721	

Table 14. Comparison of fault-detection ability between two groups

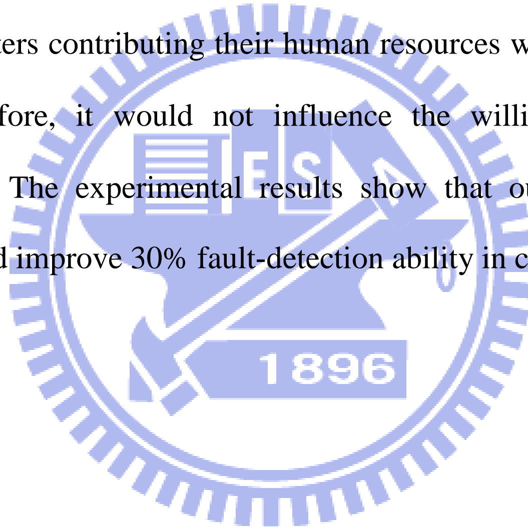
Group	Experimental group		Control group		p-value
	Mean	S.D.	Mean	S.D	
TP	0.2236	0.2927	0.1867	0.2263	0.4170
TN	0.4056	0.2988	0.4317	0.2887	0.6019
FP	0.0394	0.1555	0.0441	0.1295	0.8474
FN	0.2973	0.2756	0.3255	0.2836	0.5518
Recall	0.3325	0.3885	0.3051	0.3515	0.6655
Precision	0.4918	0.4950	0.4848	0.4708	0.9321
f-measure	0.3785	0.4110	0.3546	0.3771	0.7233

Table 15. Fault-detection ability of two collaborative testing approaches

	Control group	Experimental group
TP	0.3214	0.4285
TN	0.3571	0.4642
FP	0.1428	0.0357
FN	0.1785	0.0714
Recall	0.6429	0.8571
Precision	0.6923	0.9230
F-measure	0.6666	0.8888

Chapter 6 Conclusion

In our thesis, we first propose the value-oriented dependence graph which is a fine-grained Web application model. And then, based on value-oriented dependence graph, we further propose a novel Web application model, state transition diagram, for supporting problem decomposition and further advanced job assignment algorithm. We also formulate Minimum Test Cost Problem (MTCP) in collaborative testing by considering the constraints of real environment, and prove that MTCP is an NP-Complete problem. Finally, we propose a heuristic-based dynamic stubbing algorithm to solve MTCP and implement a two-phase intelligent collaborative testing system by applying dynamic stubbing technique. This technique allows folk testers contributing their human resources with barely noticing varying test environment. Therefore, it would not influence the willingness of folk testers to contributing themselves. The experimental results show that out proposed approach can reduce 50％ due time and improve 30% fault-detection ability in collaborative testing.



References

[1] H. Miao, Z. Qain, B. Song, "Towards Automatically Generating Test Paths for Web Application Testing", 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 211-218, Nanjing, China, June 2008.

[2] M. Benedikt, J. Freire, P. Godefroid, "VeriWeb: Automatically Testing Dynamic Web Sites", In Proceedings of 11th International World Wide Web Conference, pp. 654-668, Honolulu, HI, USA, May 2002.

[3] M. Sun, Y. Chen, S. Chen, J. Mei, "A model checking approach to Web application navigation model with session mechanism", 2010 International Conference on Computer Application and System Modeling (ICCASM), pp. 398-403, Taiyuan, China, Oct 2010.

[4] S. Chen, H. Miao, B. Song, Y. Chen, "Towards Practical Modeling of Web Applications and Generating Tests", 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 209-217, Taipei, Taiwan, Aug 2010.

[5] Wenhua Wang, Sampath, S., Yu Lei, Kacker, R., "An Interaction-Based Test Sequence Generation Approach for Testing Web Applications", 11th IEEE High Assurance Systems Engineering Symposium, pp. 209-218, Nanjing, China, Dec 2008.

[6] Y. Wu, J. Offutt, X. Du., "Modeling and testing of dynamic aspects of web applications", Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004.

[7] Z. Qian, H. Miao, H. Zeng, "A Practical Web Testing Model for Web Application Testing". Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, pp. 434-441, Shanghai, China, Dec 2007.

[8] Oliverira, M. C. F. de, and Turine, M. A. S., and Masiero, P. C. "A Statechart-based Model for Hypermedia Applications", ACM Transactions on Information Systems (TOIS), Vol. 19, No. 1, pp. 28-52, Jan 2001.

[9] G. Rossi, D. Schwabe, “Object-oriented design structures in web application models”, Annals of software engineering, Vol. 13, No.1, pp. 97-110, June 2002.

[10] A. Andrews, J. Offutt, R. Alexander, “Testing Web Applications by Modeling with FSMs”, Software and Systems Modeling, Vol. 4, No. 3, pp. 326-345, July 2005.

[11] C. H. Liu, D. Kung, P. Hsia, and C. T. Hsu, “Structure testing of Web applications”, In Proceedings of the 11th Annual International Symposium on Software Reliability Engineering, pp. 84-96, San Jose, CA, USA, October 2000.

[12] Chien-Hung Liu, Kung, D.C., Pei Hsia, “Object-based data flow testing of web applications”, First Asia-Pacific Conference on Quality Software, pp. 7-16, Hong Kong, China, Oct 2000.

[13] D. Kung, C. H. Liu, and P. Hsia, “An object-oriented Web test model for testing Web applications”, In Proc. of IEEE 24th Annual International Computer Software and Application Conference (COMPSAC2000), pp. 111-120, Hong Kong , China, October 2000.

[14] H. Miao, H. Zeng, “Model Checking-based Verification of Web Application”, 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 47-55, Auckland, New Zealand, July 2007.

[15] Liping Li, Huaikou Miao, Shengbo Chen, “Test Generation for Web Applications Using Model-Checking”, 11th ACIS International Conference on Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD), pp.237-242, London, England, June 2010.

[16] Liping Li, Qian Zhongsheng, Tao He, “Test Purpose-Based Test Generation for Web Applications”, First International Conference on Networked Digital Technologies, pp. 238-243, Ostrava, Czech, July 2009.

[17] Liping Li, Zhongsheng Qian, Tao He, “An Approach to Testing Web Applications On-The-Fly”, International Conference on Management of e-Commerce and e-Government, pp. 428-431, Nanchang, China, Sept 2009.

[18] F.M. Donini, M. Mongiello, M. Ruta, and R. Totaro, "A Model Checking-based Method for Verifying Web Application Design", Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 151, No. 2, pp. 19-32, May 2006.

[19] R.M. Hierons, "Adaptive Testing of a Deterministic Implementation against a Nondetermistic Finite State Machine", The Computer J., Vol. 41, No. 5, pp. 349-355, June 1998.

[20] D. Lee, M. Yannakakis, "Principles and Methods of Testing Finite-State Machines—A Survey," Proceedings of the IEEE, Vol. 84, No. 8, pp. 1089-1123, Aug 1996.

[21] PengCheng Xiong, YuShun Fan, MengChu Zhou, "A Petri Net Approach to Analysis and Composition of Web Services", IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol. 40, No. 2, pp. 376-387, March 2010.

[22] Xitong Li, Yushun Fan, Sheng, Q.Z., Maamar, Z., Hongwei Zhu, "A Petri Net Approach to Analyzing Behavioral Compatibility and Similarity of Web Services", IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol. 41, No. 3, pp. 510-521, May 2011.

[23] Robidoux R., Haiping Xu, Liudong Xing, MengChu Zhou, "Automated Modeling of Dynamic Reliability Block Diagrams Using Colored Petri Nets", IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol. 40, No. 2, pp. 337-351, March 2010.

[24] Lefebvre, D., Leclercq, E., "Stochastic Petri Net Identification for the Fault Detection and Isolation of Discrete Event Systems", IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, Vol. 41, No. 2, pp. 213-225, March 2011.

[25] P. Stotts, R. Furuta, "Petri-net-based hypertext: Document structure with browsing semantics", ACM Transactions on Information Systems (TOIS), Vol. 7, No.1, pp. 3–29, Jan 1989.

[26] M. J. Harrold, A. J. Offutt and K. Tewary, "An Approach to Fault Modeling and Fault

Seeding Using the Program Dependence Graph", Journal of Systems and Software, Vol. 36, No. 3, pp. 273-295, Elsevier, Mar 1997.

[27] Y.H. Tung, S.S. Tseng, T.J. Lee and J.F. Weng, "A Novel Approach to Automatic Test Case Generation for Web Applications", 2010 10th International Conference on Quality Software (QSIC), pp. 399-404, Zhangjiajie, Hunan, China, July 2010.

[28] Kuo-Chang Huang, "Applying Folksonomy-Based Approach to Support Collaborative Testing of Web Applications", National Chiao Tung University, Degree of Master, July 2010.

[29] Briand, L.C. and Pfahl, D., "Using simulation for assessing the real impact of test-coverage on defect-coverage", IEEE Transactions on Reliability, Vol. 49, No. 1, pp. 60-70, Mar 2000.

[30] M.R. Garey , D.S. Johnson, In: V. Klee (Ed.), "Computers and intractability, a guide to the theory of NP-completeness", Freeman, New York, 1979.

[31] G.K. Baah, A. Podgurski, M.J. Harrold, "The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis", IEEE Transactions on Software Engineering, Vol. 36, No.4, pp. 528-545, Aug 2010.

[32] J. Ferrante, K. Ottenstcin, and J. Warren, "The program dependence graph and its use in optimizatio", ACM Transactio on Programming Languages and Systems, Vol. 9, No. 3, pp. 319-349, July 1987.

[33] J. Karl, M. Linda, "The program dependence graph in a software development environment", Proc of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Vol. 19, No. 5, pp. 177-184, New York, NY, USA, May 1984.

[34] K. Tewary, M.J. Harrold, "Fault Modeling using the Program Dependence Graph", 5th International Symposium on Software Reliability Engineering, pp. 126-135, Monterey, CA, USA, Nov 1994.

[35] Ray-Yaung Chang, Podgurski, A., Jiong Yang, "Discovering Neglected Conditions in

Software by Mining Dependence Graphs”, IEEE Transactions on Software Engineering, Vol. 34, No. 5, pp. 579-596, Oct 2008.

[36] S. Bates, S. Horwitz, “Incremental Program Testing Using Program Dependence Graphs”, 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 384-396, New York, NY, USA, 1993.

[37] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs”, ACM Trans. Programming Languages and Systems, Vol. 12, no. 1, pp. 26-60, Jan 1990.

[38] B. Baudry, Y. Le Traon, G. Sunye, “Testability Analysis of a UML Class Diagram”, 8th IEEE Symposium on Software Metrics, pp. 54-63, Washington, DC, USA, Aug 2002.