


## 一、緒論

隨著高科技產業的發展，電腦的運算速度和能力越來越快，也因此許多新的軟體需求和開發系統的方法也慢慢的越來越多，越來越要求人性化。而所謂的人性化方面從使用和開發的角度來看可以分成兩大類。對軟體系統使用者而言是好學易用，而最常看到的就是圖形化的使用者介面。而對程式開發人員來說就是一直推行了很久的物件導向、軟體元件...，這其中都隱含了所謂的重複使用的觀念[8][15][16]。而現今許許多多看起來很新的系統，如果去看它的原始碼也會發現，許多部份都是繼承前一代的系統而來，然後一代接一代的發展下去。

由上面的敘述可以知道，圖形化的介面，元件重複使用以及改良式開發是現今軟體開發非常普遍的現象。因此軟體測試也需要針對這些狀況來加以改變，探討這方面相關的測試理論非常的多，有些已經不只是理論，相關產品在市面上找得到的也已經不勝枚舉了[11][13]。

在這麼多的理論和相關產品裡面，各有各的優缺點。如果真正套用到實際應用面上是不是會發生什麼問題，還有什麼改善的空間這都是值得我們探討的議題。

### 1.1 背景與動機



相信對每一位測試人員而言，最頭痛的事就是如何在最短的時間內完成最完整的測試。在軟體測試中對於完整測試的定義通常就是：『測試案例集中包含了所有不同狀況的測試案例』。而如何想出全面且完整的測試案例，也是許多測試人員在測試過程中更頭痛的一部分了。在現今如果是比較大的軟體系統開發專案，通常都是使用漸進式開發(Incremental development)的方式。所謂的漸進式開發，就是先將每一個階段的目標訂出來，然後隨著每一個版本的發行，進行所謂的回歸測試(regression testing)檢查程式是否有錯誤，以及檢視方向是否需要調整[1]，也因此測試的工作就更為加重了。

對於軟體公司而言在開發一套系統的過程中，理論上測試的時間應該會佔很大的比例，如果是使用漸進式開發方法時，不斷的回歸測試又會佔了測試大部分的時間。但是事實上在緊湊的時程規劃中測試的時間都被異常的壓縮了，每一個版本的發行都是測試人員的一次煎熬，要檢視舊的也要想新的測試案例，並且要把每一個案例執行過，但是在異常壓縮的時間下並無法將所有的測試確實的完成，也因此常常造成產品的測試不完全。大部分的測試人員通常會採取只測試有新增或修改的部分，而放棄其他部分。可是這並不能保證沒有測試到的部分不會有問題。而客戶在使用這套系統時就像在地雷區行走一樣不知道何時會發生問題。

另外隨著現代企業電子資訊化的演進，越來越多的製造業特別是高科技相關產業都是要靠電子資訊化來達到全面自動化以提升生產的速度和品質。也因此製造及生管自動

化系統在製造業是很常見的，而公司在無法自行開發每一套系統的狀況下，便開始了買現成的軟體來做客制化或外包給廠商開發的一種模式。但是不管是外包還是自行開發，許多企業仍然是苦於行走在地雷區，引爆了一顆就少一顆的現象。但是每一次的引爆少則影響員工工作，大則導致公司虧損、形象受創。但是許多公司為了搶市場、做績效。相同的故事便在每一段時間就又大同小異的重演一次。也因此在此過程中如何將一套系統在最短的時間內做最完整的接受度測試(Acceptance testing)就成為了一個很重要的議題。

## 1.2 研究目的與預期成果

由上述可知一個完整的軟體測試不管是對軟體公司或是購買軟體的企業而言都是很重要的，但是在完整和時效的要求下如何兩者兼顧的確是個難題。特別是對於產品製造流程十分複雜的高科技製造業而言，系統的正確性對生產的品質而言又特別的重要。因此不管是新系統的引進，或是舊系統的改版，時效以及正確性是十分重要的。本研究希望能藉著參考過去和現在各種測試的方法，並且從其中找出比較適用的方法加以運用或改良，使其能適用於執行製造及生管自動化系統的接受度測試，且能達到較完整測試的方法。

預期成果如下：提出一個改善的接受度測試流程，並完成一個輔助工具程式協助測試人員能以比較方便的方式在較短的時間內產生少量但完整的測試案例。



## 1.3 章節概要

本論文的第二章藉由探討過去和現在軟體測試的方法、理論和工具來了解和說明軟體測試既有方法的優缺點以及套用到現代的軟體來做接受度測試時會發生的問題；第三章在尋找並驗證一個比較適用於接受度測試的測試案例產生方法；第四章則是討論如何將產生後的諸多測試案例進行化減的動作以求得較佳的解決方案；第五章描述將上述的方法實做出一個輔助工具程式的方法和邏輯以及最後套用到真正的系統上的結果；第六章則對本研究做一個結論，並說明未來規劃的工作。

## 二、軟體測試的研究

### 2.1 背景與動機

相信很多人都買過電子產品，以數位相機為例，很多人在買之前一定會問相機的功能以及規格，在確定要買以及交貨同時大部分的人會要求試試看有沒有問題。相對的公司在買了一套系統之後也會進行所謂的驗收，但是一般的公司是如何驗收的呢？我想不外乎是找一些員工來測試一番，怎麼測試呢？大部分是安裝成功後，照著說明書操作一次確認無誤，然後就上線使用了。這樣的測試是不是足夠呢？我想是不夠的。

由此可知，很多公司對於軟體系統的測試是不足夠的。也許會有人問軟體系統不是應該在交付給客戶的時候就應該測試無誤了嗎？這個問題的答案在理論上絕對是對的，但是在事實上則是許多的軟體公司迫於上市時間的壓力下，測試被忽略了，或是有做但不完整。為什麼會如此？撇開老闆對測試的不了解、不重視或是時間不夠等主客觀因素，軟體測試是一門很複雜的學問。裡面包含了如何做測試計畫，如何執行測試，有哪些測試的方法以及該有哪些的測試文件...等[1][11][17][18][19][20]。因此真正確實的去執行的話的確會花掉不少的時間以及精神，但是卻不一定有明顯的成效。這也是我個人認為為什麼很多測試會被忽略的原因。

是不是有什麼方法可以解決上述的問題呢？為了要找出這樣一個適用於現今圖形化使用者介面做接受度測試的方法，我們必須對軟體測試過去以及現在的理論和工具做一番研究。藉由研究這些理論的過程中尋找出適合的解決方案。

### 2.2 軟體測試

相信大家都認同一個軟體系統的好壞除了良好的設計之外，完整而確實的測試也是不可或缺的。在 V diagram(如圖 1 所示)[1]中指出了軟體發展的各個階段所要做的事，很清楚的每一個階段都有相對應的測試規劃以及測試執行，例如，當我們跟使用者談定了需求之後，便應該要有一份需求規格書，藉由這份需求規格書我們可以擬定驗收時的測試計畫，規劃驗收時該如何測試，測試項目以及測試範圍。而我們今天所要探討是軟體測試中的最後一個，接受度測試(Acceptance testing)。

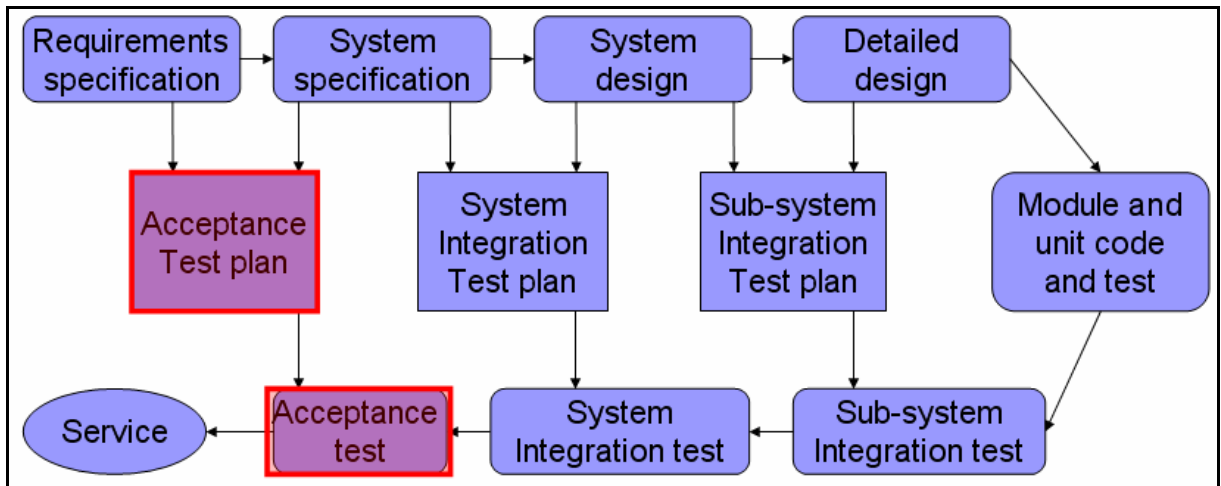


圖 1 The V diagram of testing

### 2.2.1 接受度測試

在系統可以接受操作使用之前，這是測試程序中的最後一個階段。系統是使用系統採購者所提供的資料來測試。接受度測試可以顯露出系統需求定義中的錯誤與遺漏，也可以顯露出系統功能不能實際符合使用者的需要或是系統效能無法接受的問題。[1]

接受度測試有時候稱為  $\alpha$  測試(alpha testing)，測試程序會一直進行直到系統發展者與客戶同意此系統是可接受的。而當一個系統被稱為產品之後，通常會進行  $\beta$  測試(beta testing)，測試包含了將系統送交給同意使用此系統的一些潛在客戶手中。他們會回報問題給系統發展者。例如微軟(Microsoft)在發行新版的作業系統之前就會舉辦類似的活動，廣邀一般民眾進行測試並回報錯誤。這會將產品顯露在真實使用的狀況並且可以偵測到尚未被系統建立者所預期到的錯誤。有了這些回饋之後，系統會進行修改並且發行新版並進行下一階段的測試。若沒有嚴重的問題就正式上線作為真正發行的產品了。

大致上來說，接受度測試會是最接近使用者的測試，也因此使用者介面會變成了在測試過程中最常被測試到的部分。也可以說使用者是透過使用者介面對系統進行測試。因此測試的過程及結果，通常就是不同的使用者會有不同的操作行為，也就是所謂的操作流程，因此許多的使用者就會有許多的操作流程。大量而不同的操作流程是在所有測試中都需要的，每一個操作流程在測試中就是一個測試案例(test case)。而越完整的測試案例就表示產品出錯的機率越小。但是愈大量的測試案例就表示了完整的測試嗎？答案肯定不是。但是完整的測試有時候的代價會是大量的測試案例，不過這並不是絕對的。

### 2.2.2 軟體測試的基本方法

接下來我們先來看一下軟體測試的基本方法。對於軟體測試技術，可以從不同的角度加以分類：從是否需要執行被測軟體的角度，可分為靜態測試和動態測試。從測試是

否針對系統的內部結構和具體實現演算法的角度來看，可分為白箱測試和黑箱測試。

- 黑箱測試(Black Box Testing)[1][11][17][20]

黑箱測試也稱功能測試或資料驅動測試。它是已知產品所應具有的功能，通過測試來檢測每個功能是否都能正常使用。在測試時，把程式看作一個不能打開的黑箱子，在完全不考慮程式內部結構和內部特性的情況下，測試者對程式介面進行測試，它只檢查程式功能是否能按照需求規格說明書的規定正常使用。黑箱測試方法主要有等價類劃分、邊界值分析、錯誤推測法、因果圖等，主要用於軟體確認測試。“黑箱”法是窮舉輸入測試，只有把所有可能的輸入都作為測試情況使用，才能以這種方法查出程式中所有的錯誤。

- 白箱測試(White Box Testing)[1][11][19][20]

白箱測試也稱結構測試或邏輯驅動測試。它是知道產品內部工作過程，檢測產品內部動作是否按照規格正常進行，按照程式內部的結構測試程式，檢驗程式每條通路是否都能按要求正確工作。白箱測試的主要方法有邏輯覆蓋、基本路徑測試等，主要用於軟體驗證。

“白箱”法需要全面瞭解程式內部邏輯結構，對所有邏輯路徑進行測試。“白箱”法也可以用窮舉路徑測試。在使用這一方案時，測試者必須檢查程式的內部結構，從檢查程式的邏輯著手，得出測試資料。貫穿程式的獨立路徑數是個天文數字。但即使每條路徑都測試了仍然可能有錯誤。因為，第一，窮舉路徑測試無法查出程式違反了設計規格，即程式本身是個錯誤的程式。第二，窮舉路徑測試無法查出程式中因遺漏路徑而出錯的部分。第三，窮舉路徑測試可能發現不了一些與資料相關的錯誤。

仔細思考上面的敘述可以得知，黑箱測試比較貼近使用者的操作行為，而白箱測試則是偏重於程式的邏輯正確性。不管是用哪一種方法都跟測試案例的多寡有關聯。在黑箱測試中所使用的窮舉法，在白箱測試中卻因為數量上的差別而造成最後執行測試時可行性的不同。

## 2.3 最佳化的測試案例

到目前為止我們可以很清楚的了解到，測試的重要性以及複雜性。而因為它的複雜性也造成了測試的疏漏以及人力及時間上巨大的耗費。也因此長久以來最佳化的測試案例一直是很多人在研究的議題，相關的研究大致可分為以下兩大類。產生測試案例以及化減測試案例。

### 2.3.1 產生測試案例

一般而言產生測試案例的方法大致上可分為 code-based 和 specification-based 兩

類，這兩類的方法各有優缺點，不過也相輔相成[21]。

不過我們發現在文章的數量上由程式碼來產生測試案例的理論比由規格產生的理論來的多，可能是因為現代自動化的技術越來越進步[25][26]，也因此具有固定寫作規則特性的程式碼也就越來越多人討論，而相對的沒有固定寫作規則的規格書，產生測試案例的理論也就比較少。

- Code-based[11][20]

這種方式被視為是白箱測試，因為他是由程式碼來產生測試案例的。由程式碼來產生測試案例的這種方法通常被應用在單元測試(unit testing)，因為它可以保證每一行的程式碼都會被執行過。部分 code-based testing 的理論還會包含 statement coverage, branch coverage, extended branch coverage, special values testing, domain partitioning, and symbolic evaluation. 我們通常會用 mutation analysis 的方法來評估用 code-based testing 產生的測試案例集。

以程式碼為基礎產生測試案例最大的缺點是它只有根據程式碼而完全沒有去考慮需求規格。然而，以程式碼為基礎產生的測試案例的確是有用的，因為它是程式怎麼寫就會有相對應固定的測試案例，而不是你經過思考認為程式怎麼寫再想出來的測試案例。

- Specification-based[11][20]

這種方式則被視為是黑箱測試，因為軟體被視為是一個會將輸入基於規格書轉成特定輸出的黑盒子。以規格書為基礎產生測試案例的方法通常是用於整合測試(integration testing) 和系統測試(system testing)以確保軟體是依照我們想要的去做出來的。

因為以規格書為基礎的測試只考慮軟體的外在介面，因此可以說是測試產品而非測試他的設計也因此可能無法做到對程式全面的測試。依照規格書來測試可能會造成你經常遺漏了某些規格書中沒有提及的問題。基於規格書來產生測試案例的技術中很多是跟以程式碼為基礎的理論中的一樣[4][21]。

### 2.3.2 化減測試案例

在產生了許多的測試案例之後，接下來該如何去測試。我想大概很多人會直覺的認為將測試案例丟給自動化測試系統做就好了。這的確是個好主意，但是如果測試案例數量很多，以目前的自動測試工具而言，第一次測試該案例時還是要先手動建立測試案例，往後才能自動測試。也因此手動建立測試案例還是會耗費掉許多的時間，是不是有辦法在不降低測試涵蓋率的狀態下，把測試案例盡可能的化減[2][10]？我們找到了2個經過數學驗證過可以求出最佳測試案例集的方法如下：

- Zero-One method. [2]

將大量的測試案例轉換為方程式的矩陣，並利用線性規劃的方法求出最佳解以達到

化減測試案例的目的。

- Minimum flow method. [5]

這個方法的主要觀念是藉由先將一部分問題的輸入預先處理過以增進處理效率。這個方法在許多的資訊科學各種領域中的研究中被引用。

### 2.3.3 軟體開發的現況

在看完上面兩節的敘述後，我們可以知道尋求最佳化的測試案例集合一直是許多人努力的目標，也有很多方法可行了。如此一來還有什麼好研究的呢？讓我們先來看一看現代應用軟體的特色以及軟體開發的現象。

首先，現今大部分應用軟體的人機介面都是使用圖形化使用者介面(GUI: Graphic User Interface)。對於視窗類的應用軟體系統而言圖形化使用者介面的設計和實作又更為的重要。現在的應用程式大部分都是圖形化使用者介面主要原因是，對使用者而言十分的直覺、方便易學且容易操作。這些好處都是文字模式所無法比擬的。

另外，現在開發一套大的應用軟體時經由重複使用現有的軟體元件並在不足之處加上新的程式碼的這種開發方式已經很廣泛被使用了。已經很少有人會重新開發一套新的系統了。產業界由於對時效性的要求，在購買軟體系統時通常會先尋求一個堪用的產品，然後再加以客制化。而客制化的過程中元件的重複使用或修改都是很常見的事情，現在市面上看的到的知名套裝軟體通常也都是經歷了好幾個版本的产品了。

### 2.3.4 將測試案例最佳化理論應用在現今軟體系統時會產生的問題

在看完前面那麼多軟體測試的理論之後，我們接下來看看這些理論在經過了軟體的演化之後是不是還適用。會不會產生什麼問題？

首先，如果採用白箱測試以原始碼來產生測試案例的方法來測試，一般而言如果程式碼不多還可行，但是隨者程式碼的變多這個方法會越來越不可行[1]。而真正值得我們大費周章去測試的系統通常程式碼都不會太少。這也就是為什麼驗收系統時一般來說是不會用白箱法來做測試的主要原因。而且並不是每一套軟體都會附有原始碼。如果是商業用的套裝軟體通常都不會有原始碼。另外在產業界中客制化套裝軟體的購買並不一定包含原始碼，大部分要原始碼又是另外議價了，通常老闆不一定會願意買，就算你願意買軟體公司還不一定願意賣。這個時候你怎麼用這個方法去驗證系統的正確性呢？

如果你有長時間維護修改過系統的程式的話，你會發現基於快速開發的目的很多系統是使用現成的軟體元件或是將舊系統翻新的手法來開發的。因此裡面就會參雜了許許多多沒有被用到但是又不知道能不能移除而被留在裡面的元件，如果你是利用原始碼來產生測試案例時，這些通常也會被加入到測試案例中。很明顯的這些多出來的部分你就

會浪費你的時間。

## 範例

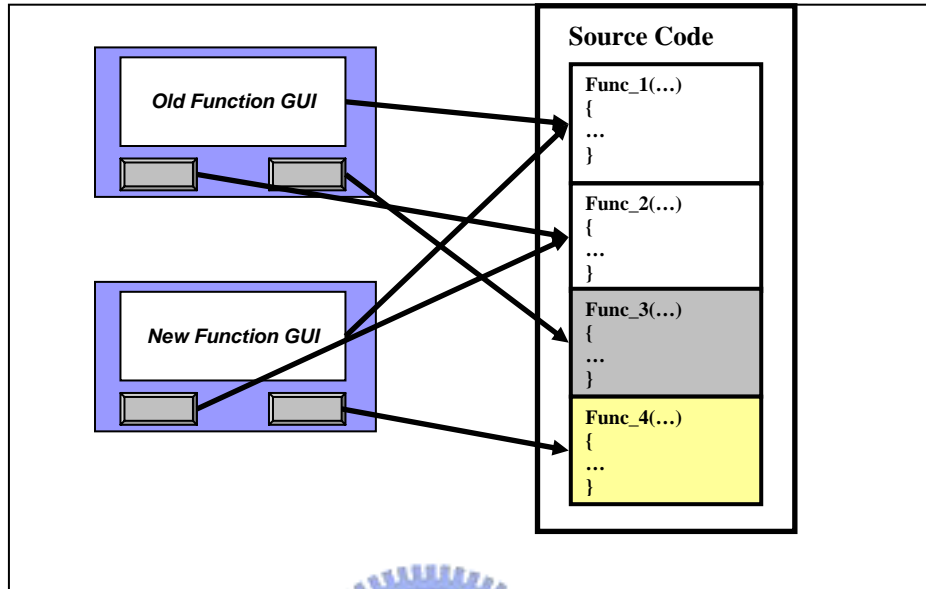


圖 2 GUI 與 Function 對應圖

以圖 2 為例，在新的應用系統中只會使用到 Func\_1, Func\_2 和 Func\_4，也因此只要測到這三項即可，但是如果是由原始碼來產生測試案例卻連 Func\_3 都會有測試案例。以此類推，在現今如此龐大的系統中又會有多少不需要存在的程式碼在裡面，有些可能對某些人是用的到，可是對其他人而言卻是用不到的。這又該如何解決？其它不是經由原始碼產生測試案例的方式通常都很複雜。要由人工手動來做的話會十分的繁瑣而且沒有效率。

綜合上面的問題，以及站在使用者的角度來看。我們可以歸納出以下兩點(1)白箱測試最大的缺點是只有根據程式碼而完全沒有去考慮需求規格，且產生出來的測試案例較多。但是優點是每段程式碼有相對應固定的測試案例而不需要由人去思考如何產生測試案例。(2)黑箱測試的優點是方便，只考慮軟體的外在介面，是測試產品而非測試設計，不過缺點就是會遺漏了某些規格書中沒有提及的問題。但是對使用者而言只要所有的功能運作正常就好了，他並不理會程式碼是否有問題。這就是為什麼我們要選擇以黑箱測試來討論接受度測試這個議題的原因了。而至於黑箱測試的缺點部分我們也會在下面的章節中討論如何改善。

## 2.4 現有的軟體黑箱測試工具

基於上述的問題，我們參考了一些市面上比較知名的而且有提供 GUI 測試自動化的



工具軟體列於表 1[11][13]。因為這些工具都是由使用者介面來進行測試，所以是屬於黑箱測試的工具。從最簡單的紀錄使用者行為，再重新執行，或是使用 Script Language 的方式，到使用直接分析原始碼的方式。這些工具可以將許多人工手動的部分自動化，很多也對於測試計畫的時程進度控管上，有許多的輔助功能。使用者可以從這些工具上知道目前的測試狀況以及進度。甚至有些還可以及時產生網頁格式的報告，讓團隊成員，特別是專案負責人隨時掌握狀況。這些工具的確在測試程式的時候有不少的幫助。

Tool	Description	Company
Win Runner[13]	Writing TSL scripts to enable automation testing.	Mercury
Silk Test [25]	Provides object-oriented programming language called <i>4Test</i> .	Segue
Rational Robot [26]	Generates test scripts in SQABasic, an integrated MDI scripting environment that allows you to view and edit your test script while you are recording.	IBM
QA Run [27]	A capture-replay tool with an integrated database, The captured test sequence is encoded in a scripting language, so you can tweak the tests with a text editor.	Compuware

表 1 常見黑箱測試工具列表

上述的軟體測試工具中目前以 Mercury 公司的 Winner 市場佔有率最高約 70%，所以我們採用 Mercury 的方案為例，整個測試的流程大致上如下：

#### Step1. Test Requirement Phase:

將專案的需求依照功能性與非功能性，鍵入 TD (TestDirector)的 requirement 模組，並且訂出 priority，指派負責的測試人員。負責的測試人員，就要針對他負責的需求，參考其他開發文件或是使用手冊，去深入了解需求的目的是、需求是否可測試，開始要去想該怎麼測試、要準備哪些測試資料。

#### Step2. Test Plan Phase:

根據 test requirement，負責的測試人員，開始建立 test plan→test case→test step，運用 WinRunner 或 LoadRunner 手動建立自動化的測試腳本(script)。

#### Step3. Relation Creation Phase:

將 test requirement 與 test case 作關聯，如此一來，當 requirement 有變更，哪些相關的 test case 也要隨著作修改，另外也可以知道 priority 高 requirement 是否有設計夠多的 test case 去測試。

#### Step4. Test Execution phase:

依照測試的性質，將 test case 作 grouping，如每一個 build 都要做的 smoke testing，將相關的 test case 都拉在一起。安排測試的時程。除了少數是以人工測試以外，其餘盡量都以工具測試。

#### Step5. Defect Tracking phase:

負責的測試人員將測到的問題(還不確定是否為 BUG)，填入 PVCS Tracker 系統，接下來就進入 SCM 流程了。Developer 修改完了，測試人員再重測一遍。

#### Step6. Management:

在這整個測試了流程中，PM 隨時透過 TestDirector 了解進度與品質，基本上所有的測試資料都進入 TestDirector，所以已經沒有什麼 word 或是紙本的測試計劃與測試報告了，因為最新的資料都在 TestDirector 裡面了，上網就可以看最新的狀況。

### 2.4.1 市面上黑箱測試工具的缺點

在導入工具後，雖然剛開始時是十分痛苦的，但是軟體的品質是有提昇，因為測試的執行更為頻繁。另外專案經理也比較能掌控進度與品質，譬如測了哪些需求，測試的結果如何，時程趕的時候，也能以優先權高的需求優先測試。市面上既然已經有這麼多好用的工具，為什麼使用率還是不普遍呢？排除價格因素這些工具還是有許多的問題。常見的問題如下：

- 每一次手動 capture-replay 的動作只會產生一個測試案例，而且不是很容易建立。
- 如果要一次產生很多不同的測試案例則一定要用內建的 script 寫程式，更是麻煩。
- 這些工具是標準的黑箱測試工具，也就繼承了黑箱測試的缺點只提供對已經產生的測試案例套入不同的資料作資料驅動的測試，無法以現有的測試案例產生不同邏輯的測試案例。
- 在這些軟體工具中都沒有測試案例化減的功能。
- 這些工具大多著重在測試案例以及測試資料的管理以及自動化執行測試。

讓我們來討論這些問題，首先，一般市面上大部分的系統都有提供紀錄使用者動作這個功能，單純紀錄使用者行為，這是最直覺的，也是最友善的操作介面，只要你會操

作系統就能產生測試案例，不過其涵蓋率可能偏低。如果你想產生越多的測試案例，你就必須越勤勞的將每個案例至少跑過一次。如果遇上複雜的大系統這個方法通常是會很不好執行。

有了第一個問題後，也因此有業者提供了 Script Language 來做加強，不過就算加上使用 Script Language 涵蓋率要達到令人滿意的程度還要花費很多的心思和時間。而且一般使用者並不一定會寫 Script Language，通常只有軟體公司才会有培養這方面的人才。排除人才問題之外，Script Language 也算是一種程式語言。只要是程式就有可能會有邏輯上設計上的錯誤，錯誤的邏輯出來的結果自然是不對的。

上述的工具大都強調其輔助性，協助測試計畫的執行和管理。協助將手動的操作轉為自動，利用現有的測試案例結合測試資料產生大量的測試案例，但是就是僅止於相同的測試案例套用不同的測試資料。但是在測試案例中雖然不同的測試資料很重要，但是不同操作流程或邏輯的測試案例也很重要。但是受限於這些工具是黑箱測試的工具因此大部分的工具都只著重在管理測試案例和測試資料部分，頂多提供流程控制的功能。但是沒有看到有提供利用現有的測試案例來產生不同操作流程不同邏輯的測試案例這種功能。如果有的話測試人員就可以節省不少時間，不用花時間來想這種可以自動化產出的測試案例了。另外也沒有看到有工具提供使用者將測試案例化減的功能。甚至連檢查測試案例內容是否重複的功能也沒有，或許是因為認為再多的測試案例都有工具會自動執行而不重視，也很可能是因為產品的定位是黑箱測試工具的關係。對於小系統而言這是可行的，但是對於複雜的大系統，太多的測試案例是會浪費掉大家許多的寶貴時間。

這個時候我們不禁會想，一般市面上的測試工具對使用者而言是否有更好更快速的方式來產生測試案例？對現今 GUI 的應用軟體而言我們是否能夠在可接受的涵蓋率之下產生最佳數量的測試案例？

## 2.4.2 建議的解決方案

針對上一小節的問題，我們認為如果能夠有一套方法能夠同時滿足下面兩個條件的話，就可以解決上述的問題了，條件如下：

- 針對圖形化使用者介面系統操作流程產生測試案例的方法。
- 一個可以配合上述產生測試案例方法且有效率化減測試案例的方法。

以下分別說明兩個條件的意義。首先目前市面上的軟體大部分都有圖形化的使用者介面，製造業用的系統也不例外。因此如何提供一個方便的方法讓測試人員能夠很簡單而直覺的產生測試案例就變的很重要了。如果介面能夠讓受測系統的直接使用者也很容易上手而且不會花太多時間及精神就可以產生測試案例那就更好了，畢竟受測系統的直接使用者通常會是該領域的專家，但是他們通常未必有時間全力配合

測試。如何留住他們的測試資料也是很重要的。

另外如何利用獲得的資料來產生最佳化的測試案例集，也是攸關最後要執行測試時所要花的時間以及達到的品質。在這裡所謂最佳化的測試案例集，是指能以最少量的測試案例達到百分之百的涵蓋率。這樣才能夠達到我們想要在最短時間內對系統做最完整測試的這個目的。



## 三、產生測試案例的方法

### 3.1 背景與動機

不管在做哪一類的測試，不可否認的測試案例都是相當重要的一部分。而如何產生測試案例也是許多人研究的議題[18][20][21][22][24]。一個好的方法可以讓測試案例在產生的時候就能夠兼顧涵蓋率以及數量[23]，一般的狀況下大家都會希望涵蓋率能夠越高越好，同時又希望必要執行的測試案例能夠越少越好。較高的涵蓋率可以保證我們系統出錯的機率較低，但是通常隨之而來的就是比較多的測試案例，而太多的測試案例可能會導致測試時間的拉長，甚至是變成無法完成的測試。如何取捨端看每個系統的需求，而在以圖形化介面為主對製造執行系統做可接受度測試的這個狀況下又是如何？我們將一一討論。在這一章我們將先討論如何產生涵蓋率比較高的測試案例，而至於測試案例的化減我們將在下一章進行討論。

### 3.2 產生測試案例

*A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly. A test case should contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results. Note that the process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's useful to prepare test cases early in the development cycle if possible. [10]*

由上面的敘述可知測試案例的定義。一般而言測試人員除了測試系統之外，最主要的工作便是想出所有可能的測試案例，除了一般正確的例子之外，也要想一些可能會造成系統出錯的例子。在 2.2.2 節時有提到三種測試的基本方式，裡面有提到所謂的窮舉法。

#### 3.2.1 窮舉法

窮舉法顧名思義就是將所有可能的測試案例通通列出來後一一進行測試。這是大家第一個想到的方法最直接也可能最完整，也是黑箱測試中標準的方法。可是如何將所有的測試案例列出來又是一個問題了。很幸運的這個問題很久之前就有很普遍的解法了，深究其精神就是透過控制流程圖來產生測試案例。因為程式碼是流程的程式化，而使用

者操作的過程也一定會被限制在程式許可的範圍內。

也因此透過控制流程圖這種觀念產生測試案例的這種方法是很常被大家所採用的。也因此有許多產生控制流程圖的方法和理論。而在白箱測試的方法中由原始碼來產生測試案例是目前最普遍的方式。測試人員可以透過轉譯器將程式碼轉成控制流程圖再產生測試案例(如圖 3)，不過相信大家也知道白箱測試的方法並不適用於接受度測試，一般而言接受度測試並不會牽涉到程式碼，而且如果採用由原始碼產生的測試案例來做為接受度測試的案例的話，會遇到測試案例太多以及多餘測試案例的問題

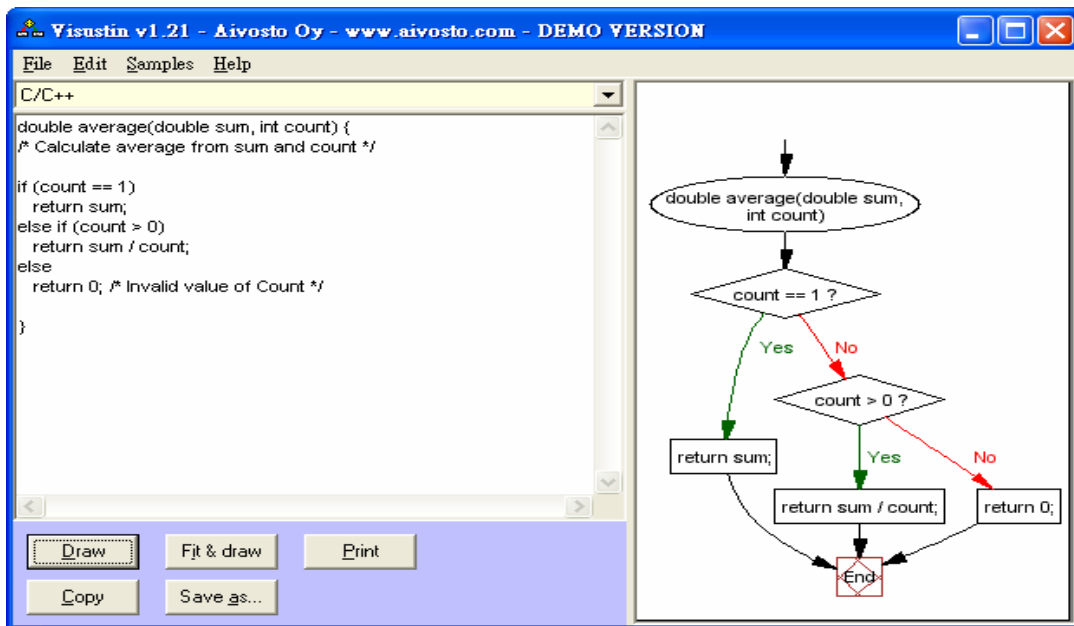


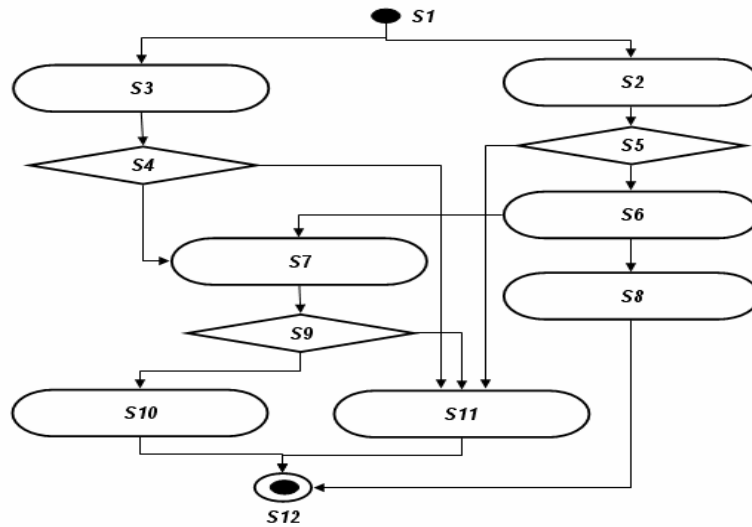
圖 3 程式碼轉成控制流程圖

一般而言接受度測試採用黑箱測試方式會比較適合。但是就無法也不適合套用由程式碼產生的測試案例。因此在這種狀況下測試案例的產生就變的比較麻煩。

### 3.2.2 控制流程圖與測試案例

是不是採用黑箱的方式測試時就沒有比較方便的方式來產生測試案例嗎？如果我們試著從 GUI 的角度來看，對應用程式而言，測試案例其實也可以說就是一連串使用者操作的動作。而這些動作，操作的流程必然是一定要遵循某些規則，而這些規則就是規格書裡面所訂定。因此很多人會認為我們只要將規格書上面所提到的規則一一測試過便可以了，不過這是會有問題的，你可能會漏掉了許多的排列組合。但是也不是所有的排列組合都是需要測試的。一般而言規格書內會有控制流程圖或是可以藉由裡面的描述產生控制流程圖。通常只要將控制流程圖內所有可能的路徑(All Path)測試過大概就可以達到一個比較令人滿意的涵蓋率了。如圖 4 所示只要你測試過下面所有的測試案例大概就能保證所有的功能你都測試過了，不過這並不表示這套系統萬無一失了，他可能會有其他的非功能面的問題，例如效能不好、Race condition 以及 Dead lock...等問題，不過這

不在本文考量的範圍內。



Node No.	1	2	3	4	5	6	7	8
Case No.								
Case1	S1	S2	S5	S6	S7	S9	S10	S12
Case2	S1	S2	S5	S6	S7	S9	S11	S12
Case3	S1	S2	S5	S6	S8	S12		
Case4	S1	S2	S5	S11	S12			
Case5	S1	S3	S4	S7	S9	S10	S12	
Case6	S1	S3	S4	S7	S9	S11	S12	
Case7	S1	S3	S4	S11	S12			

圖 4 控制流程圖與測試案例

### 3.2.3 控制流程圖與使用者介面

在上一小節提到路徑(Path)，在這裡路徑的定義就是在控制流程圖中由起點往終點的方向走，直到終點為止稱為一個路徑。而控制流程圖其實從另一個角度來看也可以說是許多有共同特性路徑組合成的一個集合。而一般的系統在不同的層次中會有不同的控制流程圖，會有由最上層來看的控制流程圖，這一類的控制流程圖會偏重在系統運作的大原則而非細節，在這個控制流程圖中的每個步驟可能又包含了另一個下一層次的控制流程圖。當然相對的到了最下層的控制流程圖就會偏向使用者的操作流程。也因此測試的策略中會有分成所謂的由下而上或是由上而下兩種策略。

在前面的章節我們一直有提到測試案例，在這裡我們可以很清楚的看到其實每一個測試案例和每一個路徑是互相對應的。然而每一個測試案例的底下從系統使用者介面的角度來看通常代表的意義是一連串使用者對系統的動作。由此我們可以更清楚控制流程圖和使用者介面之間的關係。

由於前面我們看到市面上有許多知名的軟體測試工具是利用紀錄使用者操作流程

來當成測試案例，一次的操作便是一個測試案例。因此我們不禁會想，使用者是不是可以經由紀錄操作 GUI 時的每一個動作來產生操作控制流程圖。

### 3.3 由 GUI 產生控制流程圖

在上一節我們提到由紀錄操作 GUI 時的動作來產生控制流程圖，若是我們能將每一次的操作流程紀錄並轉成控制流程圖的一部分，當我們把畫面中所有元件的關係及流程都建立完成之後，我們就可以藉此產生所有的測試案例，也就不需要把時間花在想各種測試案例的排列組合上面了，更不用一次一個測試案例慢慢建立了，也比起寫 script 簡單多了。在這裡我們便要討論是否可行，以及若是可行該怎麼做。

#### 3.3.1 路徑測試

路徑測試為白箱測試策略，它的目的為透過元件執行每一條可獨立執行的路徑。假如要執行每一條獨立路徑，程式中所有的敘述至少都必須被執行過一次。再者，所有狀況的敘述都要測試真(true)和假(false)的狀況。路徑測試的開始點為程式控制流程圖，這是通過程式所有路徑的骨架模型，控制流程圖包含表示決定的節點(node)和說明控制流程的邊緣(edge)，控制流程圖的架構是將程式的流程控制敘述用等價圖(equivalent diagram)取代，在流程控制中主要可以分成兩類：判斷式與迴圈。判斷式用於控制某程式區段是否被執行，共有兩類，一類是 if...elseif...else，另一類則是 switch...case。而迴圈則是用於控制某一程式區段的重複執行，共有 for, while 與 do...while 三種。圖 5 說明了控制流程圖如何表示 if...elseif...else、switch...case、while 以及 do while 五種基本的狀況。

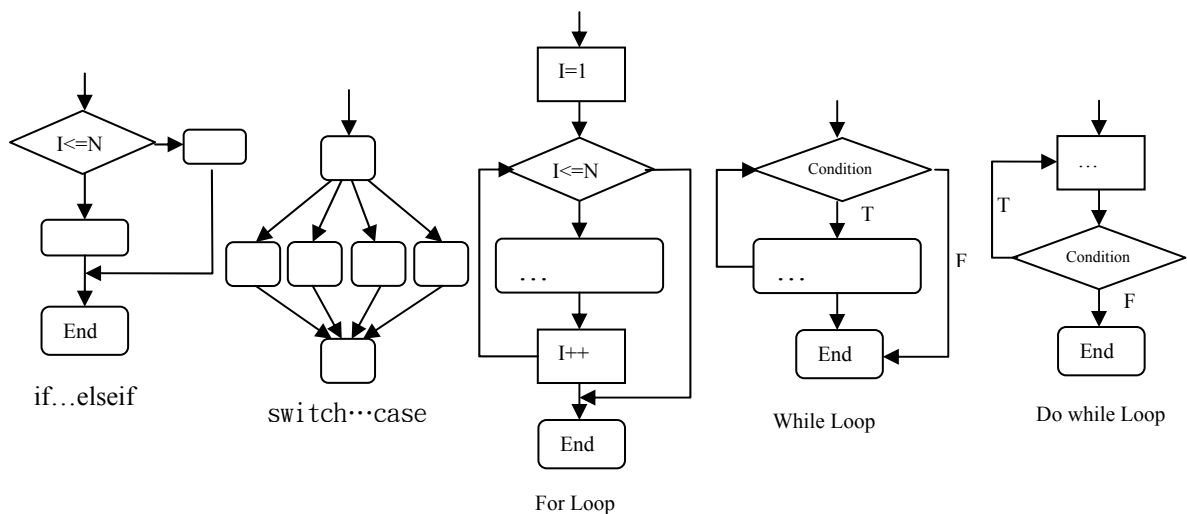


圖 5 控制流程圖表示法



看到這裡大家可能會有疑問，那就是前面一直強調是做接受度驗收也提到要用黑箱測試的策略，怎麼又提出適合白箱測試用的路徑測試呢？相信大家也知道路徑測試在白箱測試中使用時比較適合用在核心系統的程式，原因是因為核心系統的程式碼比較少。若是應用到測試整個系統相信就比較不適合了，前面有提過，原因是會有太多的路徑。

但是，如果我們能利用畫面產生控制流程圖，然後產生測試案例。相對於程式碼產生的測試案例，在數量上會少很多。就算整個系統所有畫面所構成的細部控制流程圖產生的測試案例，相對於核心系統程式碼所產生的測試案例也不算多。如果將這個方法應用在簡單的應用程式或比較單純的網頁上會比較沒有意義，因為這一類程式的畫面和流程都太單純太簡單了，然而在以輔助流程控管的製造執行系統中這會是相當適合的一種方式。

### 3.3.2 圖形化使用者介面中控制流程圖的基本定義

如果上一節所說的方法要成立的話，我們要先對使用在 GUI 上面的控制流程圖做一些基本定義以及說明。在這裡的控制流程圖原則上還是基於控制流程圖的基本定義，一樣是完成一件事的過程每一個步驟該做什麼，在什麼狀況該如何反應。比較特別的地方會是，在這裡的控制流程圖會偏重在使用者的每一個動作，而條件的判斷會是依據前面的動作來決定。這裡的節點會是畫面中的每一個物件，而路徑便是每個節點在操作過程中所產生的先後關係。

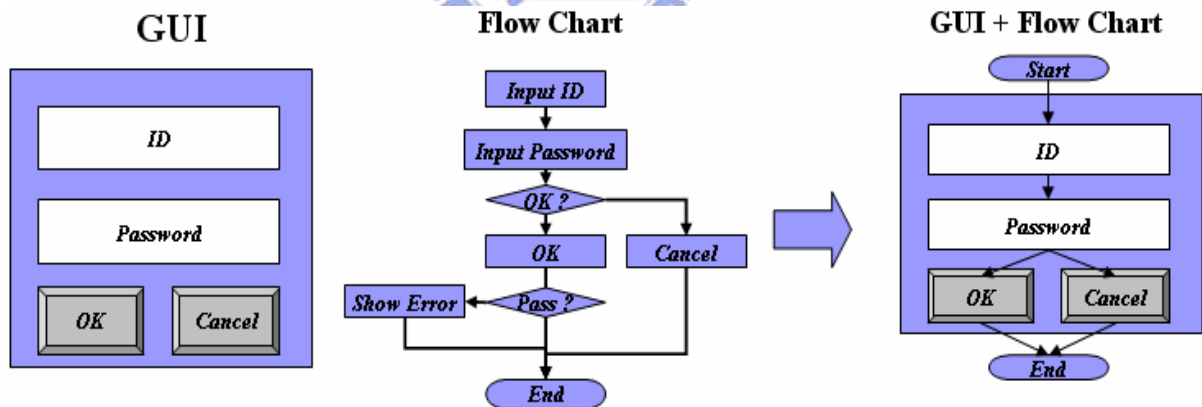


圖 6 GUI 加控制流程圖

以圖 6 為例，假設現在有一個常見的登入畫面，首先會要求輸入使用者帳號以及密碼，輸入完畢後按下[OK]鈕，如果密碼正確就可以順利登入了。當然如果你輸入錯誤或按下[Cancel]就不會登入了。但是不管登入成功與否對這個功能而言都是任務結束了。他的控制流程圖會像圖 6 中間的控制流程圖一樣。我們該如何將控制流程圖直接對應到操做介面來呢？就像圖 6 最右側的圖。首先我們先將這個功能所有可能的測試案例列出來如表 2

編號	測試案例	預期結果
1	輸入正確的 ID 以及正確的 Password 然後按[OK]	Pass
2	輸入正確的 ID 以及正確的 Password 然後按[Cancel]	No Pass
3	輸入正確的 ID 以及錯誤的 Password 然後按[OK]	No Pass
4	輸入正確的 ID 以及錯誤的 Password 然後按[Cancel]	No Pass
5	輸入錯誤的 ID 以及錯誤的 Password 然後按[OK]	No Pass
6	輸入錯誤的 ID 以及錯誤的 Password 然後按[Cancel]	No Pass
7	輸入錯誤的 ID 以及正確的 Password 然後按[OK]	No Pass
8	輸入錯誤的 ID 以及正確的 Password 然後按[Cancel]	No Pass

表 2 測試案例列表

由表 2 我們可以很清楚的看到這些測試案例都是在描述使用者與畫面互動之後該有的結果。而與使用者互動的元件則有[ID]、[Password]、[OK]以及[Cancel]這四個元件如果我們單純以這些元件的操作流程製表可得表 3。

編號	測試案例	預期結果
1	正確[ID]→正確[Password]→[OK]	Pass
2	正確[ID]→正確[Password]→[Cancel]	No Pass
3	正確[ID]→錯誤[Password]→[OK]	No Pass
4	正確[ID]→錯誤[Password]→[Cancel]	No Pass
5	錯誤[ID]→錯誤[Password]→[OK]	No Pass
6	錯誤[ID]→錯誤[Password]→[Cancel]	No Pass
7	錯誤[ID]→正確[Password]→[OK]	No Pass
8	錯誤[ID]→正確[Password]→[Cancel]	No Pass

表 3 描述操作流程的測試案例列表

如果我們不管輸入的值，那測試案例會如表 4。

編號	測試案例	預期結果
1	[ID]→[Password]→[OK]	Pass
2	[ID]→[Password]→[Cancel]	No Pass
3	[ID]→[Password]→[OK]	No Pass
4	[ID]→[Password]→[Cancel]	No Pass
5	[ID]→[Password]→[OK]	No Pass
6	[ID]→[Password]→[Cancel]	No Pass
7	[ID]→[Password]→[OK]	No Pass
8	[ID]→[Password]→[Cancel]	No Pass

表 4 無輸入值的測試案例列表

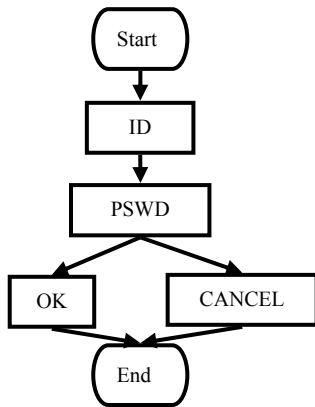
如果不看預期結果，很明顯的這個列表可以化減成為表 5。

編號	測試案例	預期結果
1	[ID]→[Password]→[OK]	
2	[ID]→[Password]→[Cancel]	

表 5 化減後的測試案例列表

而這個結果正好跟使用圖 6 最右側圖來產生的測試案例結果是一樣的。也因此可以得知直接由使用者操作圖的動作自動產生測試案例不是不可能的。不過這個圖還有一些部分需要加強，我們將在下面討論。

在上一節我們有提到控制流程圖的五個基本狀況，如果套用到這裡該怎麼解釋呢？首先是if判斷敘述，if判斷結構可分為兩類，1.單一條件判斷敘述：利用一條件式控制程式是否執行某程式敘述或由兩程式敘述中擇一執行。此判斷敘述將利用if...else...建立。在上面的範例中剛好就是這個狀況〔如圖 7〕。2.多條件判斷敘述：利用多種條件控制程式所執行的敘述，此判斷敘述將以if...else...if...else...建立。又可以分為兩種狀況來討論。一者為條列式，一者為巢狀式。不過兩者在使用相同於先前的方式，先一一列出所有的測試案例之後再化簡，兩者所得的圖卻都相同如圖8。



Case1: START→ID→PSWD→OK→END  
 Case2: START→ID→PSWD→CANCEL→END

圖 7 if-else example

<p>多條件判斷敘述          條列式：</p> <pre> if (condition A) {     Process 1...; } else if (condition B) {     Process 2...; else if ... . . . else {     Process N...; } </pre>	<p>巢狀式：</p> <pre> if (condition A) {     if (condition B)         Process 1...;     else         Process 2...; } else {     if (condition C)         Process 3...;     else         Process 4...; } </pre>	<p>多條件判斷敘述</p>
---	--	----------------

圖 8 多條件判斷敘述

接下來討論的是switch-case。在GUI中switch-case出現的頻率會是最高的，所幸這個是最簡單的只要很單純的將每個元件的關係建立，最後直接以條列式的方式產生案例即可，有幾個case就有幾個path。產生出來的圖也會如同圖8中最右側的圖一樣。其原因我們將下一小節中討論。

然後是 for-loop，在程式中很常見，但是在 GUI 出現頻率是比較少，繼續以登入畫面為例，假設登入畫面允許使用者輸入錯誤 3 次之後程式才會結束，可以用下面的圖(如圖 9)來表示。產生的測試案例結果會如圖中表所列。另外 while-loop 和 do while-loop

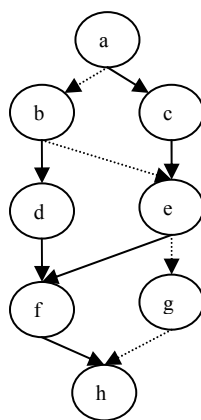
的圖也會類似圖 9。大家應該會發現產生出來的測試案例好像不太對勁，主要是因為沒有一起考慮條件式(condition)的結果。如果排除掉條件式，上面測試案例的流程是沒錯的，那該如何修正這個問題呢？我們在下一節有詳細的解說。



圖 9 for-loop example

### 3.3.3 圖形化使用者介面中控制流程圖的進階說明

看完上一節，你可能會覺得有疑問。產生出來的圖最後只剩判斷式和迴圈兩大類，這是因為我們希望能夠達到控制流程圖能夠與輸入值無關(data independent)這個目的所造成的。因此這些路徑都不會考量輸入值對選擇路徑的影響，也因此一般在控制流程圖中常見的條件式(condition)被省略掉了。所以產生出來的測試案例雖然可以包含全部的路徑或是節點，但是裡面卻有很多測試案例是無法完整執行的，如果是全測的話測試案例太多，如果做最佳化的話萬一選到的測試案例是無法完整執行的，那這個最佳化也會是失敗的。以圖 10 為例如果 node e 的邏輯是經過 node b 的只能往 node f，而經過 node c 的只能往 node g。因此 case3 很明顯是無法完整執行的測試案例。這種狀況我們通常會將 node e 在分成 e1 和 e2，但是在 GUI 上它是指同一個元件，如此便很不直覺。



Test Cases:

Case1 : a-->b-->d-->f-->h

Case2 : a-->b-->e-->f-->h

**Case3 : a-->b-->e-->g-->h**

Case4 : a-->c-->e-->f-->h

Case5 : a-->c-->e-->g-->h

圖 10 無法完整執行的測試案例

因此在這一節中我們將為您介紹一種，純粹以使用者操作方便而言補強的方式，雖然違背了 data independent 的原則不過對於初步的資料化簡和最佳化後測試案例的可完整執行性卻是有幫助的，可以幫助於更接近我們的目的。在本篇論文中我們並不會對於如何產生可能的數值作討論，我們只對迴圈和條件式做一些配合的動作，讓產生出來的路徑更接近真實的路徑。方法很簡單就是在控制流程圖中加入 token、參數以及條件式的觀念。加入了這幾項元素之後，使用者可以在每一個節點及路徑中設定參數以及條件式，當 token 經過進入路徑前先檢查是否符合條件式，進入節點後加入參數以供其他節點或路徑使用。如此便可以輕易解決上面的問題同時控制流程圖又很簡單，也可以產生正確的測試案例。只是要多設定一些東西，以圖 11 為例，在建完控制流程圖之後在[ID]的部分設定其值為(T,F)，表示其可能值為 T 或 F，用來代表正確的 ID 和錯誤的 ID，在[PSWD]的部分是一樣的，多了 Counter 的部分則表示計數器，在[OK]到[End]以及[OK]到[ID]的線上設定允許通過的條件式。當執行程式時 Token 經過[ID]會將 ID 的值設為選項中的任一值，並成為自己的參數，假設為 T。經過[PSWD]時則將 PSWD 的值設為選項中的任一值，假設為 T，以及 Counter=0 加入自己的參數列中。到了[OK]時 Counter 會加一，並依條件式選擇可以走的路徑。使用遞迴的方式拜訪過所有的點之後便可以產生所有可能值的測試案例如表 6。

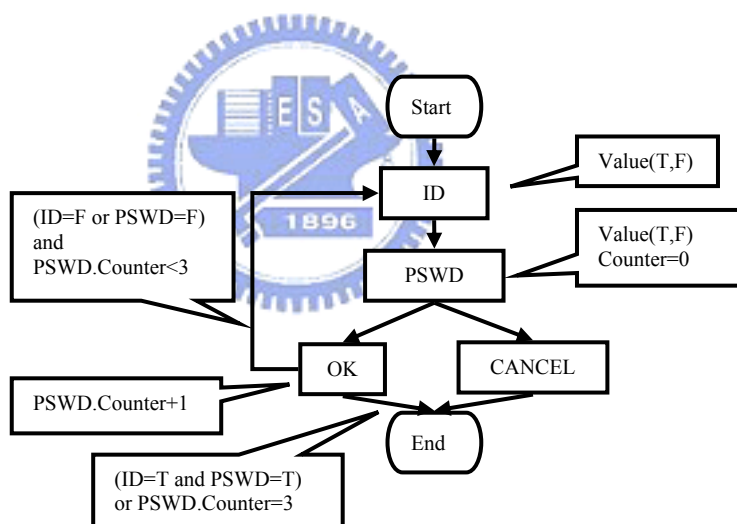


圖 11 控制流程圖中加入參數和條件式

編號	測試案例(操作流程)
1	START→ID (T) →PSWD (T, CNT=1) →OK→END
2	START→ID (T) →PSWD (T, CNT=1) →CANCEL→END
3	START→ID (T) →PSWD (F, CNT=1) →CANCEL→END
4	START→ID (F) →PSWD (T, CNT=1) →CANCEL→END
5	START→ID (F) →PSWD (F, CNT=1) →CANCEL→END



36	START→ID (F) →PSWD (F, CNT=1) →OK→ ID (F) →PSWD (F, CNT=2) →OK→ID (T) →PSWD (F, CNT=3) →OK→END
37	START→ID (F) →PSWD (F, CNT=1) →OK→ ID (F) →PSWD (F, CNT=2) →OK→ID (F) →PSWD (F, CNT=3) →OK→END

表 6 Loop-while 測試案例列表

若上面的範例拿到winrunner中來做測試，若你想要把各種使用者可能操作的流程都產生出來你只能有兩種選擇。一個是自己想所有可能的測試案例後一個一個建立到系統中。另一個是想出一個或數個測試案例然後再寫程式去控制他的操作流程以達到相同的目的，相較起來用這個方法使用者只要建好物件彼此之間的相互關係，然後加上參數和條件式就可以了，相對而言簡單多了。





## 四、化簡測試案例的方法

### 4.1 背景與動機

看完上一章之後我們得到了一個較為快速產生測試案例的方法，但是在產生許多測試案例之後，許多測試工具都是要求使用者手動建立測試案例的。這麼多的測試案例光是建立可能就要花掉不少的時間，當然我們是可以在產生測試案例的同時順便一起產生測試工具所需的 script。這個方法是可行的，不過花費時間來測試重複或是可以省略的測試案例實在浪費時間以及資源。是不是有什麼好方法來化減這些數量龐大的測試案例呢？我們將在這一章裡一一探討。

### 4.2 最佳路徑選擇問題

不管你是使用哪一種方式產生測試案例，誠如前面所提及都會是一連串使用者對物件輸入或是物件本身輸出的一種行為。在上一章我們也將這些描述簡化為以物件以及使用者輸入的操作流程來表示，如第三章的圖 4 和表 6...。這也是一般控制流程圖條列時常用的表示法。如何以最少的路徑滿足涵蓋圖中所有點的問題，稱之為最佳路徑選擇問題。在現今這個問題雖然有許多人提出很多的解決方案，但是卻只有 zero-one optimal path set selection method 和 minimum flow method 這兩種方法可以保證他的方法是最佳的。而這兩者之間又以 zero-one 的方式較為簡單[2]。因此在下一節我們將使用 Zero-One Optimal Path Set Selection 方法來做最佳化。

#### 4.2.1 Zero-One Optimal Path Set Selection Method

Zero-one 方法是屬於線性規劃中的一種，而線性規劃(Linear Programming)是運籌學的一個分支，用來處理在線性等式及不等式組的條件下，求線性目標函數的極值問題的方法。它所研究的問題主要有兩類；一、一項任務確定後，如何統籌安排，盡量做到用最少的人力、物力資源去完成這一任務。二、有一定數量的人力物力資源，如何安排使用它們，使得完成任務最多。總之，就是尋求整個問題的某個整體指標最佳化應用在運輸問題、生產的組織與計劃問題、合理下料問題、配料問題、佈局問題等[11]。

因此我們經由操作控制流程圖來產生測試案例的條列式清單。以第三章的圖 4 為例，設圖 G 為測試案例的集合，如果以每一個節點為行，每一個測試案例為列，若案例的流程中有經過該節點便將節點與案例交會處設為 1 其他則為 0。如此可得矩陣 F 如表 7。此矩陣可稱之為節點的涵蓋率(coverage frequency)矩陣。對圖 G 而言，求出以最少的案例經過每一個節點最少一次的問題稱之為決策問題。

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
Case1	1	1	0	0	1	1	1	0	1	1	0	1
Case2	1	1	0	0	1	1	1	0	1	0	1	1
Case3	1	1	0	0	1	1	0	1	0	0	0	1
Case4	1	1	0	0	1	0	0	0	0	0	1	1
Case5	1	0	1	1	0	0	1	0	1	1	0	1
Case6	1	0	1	1	0	0	1	0	1	0	1	1
Case7	1	0	1	1	0	0	0	0	0	0	1	1

表 7 節點的涵蓋率矩陣

要解決這個問題就要用Zero-one方法。首先要將矩陣轉換為不等式，設： $x_i$ ,  $i \in \{1, 2, \dots, 7\}$ ,  $x_i$ 對映至案例Case*i*。節點S,  $S_j$ ,  $j \in \{1, 2, \dots, 12\}$ 。如果Case*i*有經過 $S_j$ 則  $x_i=1$ , 否則 $x_i=0$ 。如果要求每一個節點都至少要經過一次，也就是每個點都至少要被測試過一次。以節點S1 為例，其不等式如下：

$$x_1+x_2+x_3+x_4+x_5+x_6+x_7 \geq 1, \text{ or } 1 \times x_1+1 \times x_2+1 \times x_3+1 \times x_4+1 \times x_5+1 \times x_6+1 \times x_7 \geq 1$$

依照相同的方法我們可以將其他節點的不等式列出如下：

$$S2: 1 \times x_1+1 \times x_2+1 \times x_3+1 \times x_4 \geq 1$$

$$S3: 1 \times x_5+1 \times x_6+1 \times x_7 \geq 1$$

$$S4: 1 \times x_5+1 \times x_6+1 \times x_7 \geq 1$$

$$S5: 1 \times x_1+1 \times x_2+1 \times x_3 +1 \times x_4 \geq 1$$

$$S6: 1 \times x_1+1 \times x_2+1 \times x_3 \geq 1$$

$$S7: 1 \times x_1+1 \times x_2+1 \times x_5+1 \times x_6 \geq 1$$

$$S8: 1 \times x_3 \geq 1$$

$$S9: 1 \times x_1+1 \times x_2+1 \times x_5+1 \times x_6 \geq 1$$

$$S10: 1 \times x_1+1 \times x_5 \geq 1$$

$$S11: 1 \times x_2+1 \times x_4+1 \times x_6+1 \times x_7 \geq 1$$

$$S12: 1 \times x_1+1 \times x_2+1 \times x_3+1 \times x_4+1 \times x_5+1 \times x_6+1 \times x_7 \geq 1$$


上述的不等式可以轉換成以矩陣不等式的型式表示

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}^T \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

接下來因為是要求最佳解，設  $z=x_1+x_2+x_3+x_4+x_5+x_6+x_7$  為所有被選上的測試案例之總和，則求  $z$  的最小解就是我們所要的答案了。而這種問題便是典型的 zero-one 整數規劃問題了。其表示式如下：

$$\min(\text{minimize}) z = \sum_{i=1}^7 x_i,$$

s.t. (subject to)



$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}^T \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$x_i = 0 \text{ or } 1, i \in \{1, 2, \dots, 7\}$$

#### 4.2.2 Zero-One 方法的應用與缺點

這個方法除了可以單純的計算出至少經過每個點一次的最佳路徑之外，他還可以處理以下兩種狀況：(1)每個測試案例有不同的成本(cost)，以及(2)只選擇重要的節點作規劃[2]。在成本的部分我們可以加上一個成本的矩陣  $C=[(c_i)]$ ， $c_i$  代表該測試案例的成本，而上面的計算公式將變為：

$$\min z = CX = \sum_{i=1}^m c_i x_i$$

而在只選擇重要節點的部分，可以利用涵蓋率需求矩陣(coverage requirement array)  $R=[(r_i)]$ 其中如果該節點一定要被測試則  $r_i=1$ ，否則為 0。新的公式如下：

$$\min z = CX = \sum_{i=1}^m c_i x_i$$

$$\text{s.t. } F^T X \geq R, x_i = 0 \text{ or } 1, i \in \{1, 2, \dots, m\}.$$

上述兩個功能都是在線性規劃中一定會提到的部分。雖然這個方法很好用不過他的複雜度會是  $2^{|\text{Paths}| \times |\text{Components}|}$ ，其中Paths代表可以選擇的路徑數量，而Components則代表了要求至少要多少個元件有被測試過。在這裡便可以很清楚的看到了，計算的複雜度會隨著路徑的增加而呈指數成長，而這也是這個方法最大的缺點。隨著控制流程圖的越複雜計算時間成指數型成長，而最後將會變成無法計算。

### 4.3 Zero-One 方法的改良

在上一節提到zero-one方法有許多好處，但是有一個最大的缺點就是其複雜度是隨著路徑的增加而呈指數成長。也因此便有人提出化減路徑的方法，其主要的觀念是先將矩陣做初步的化減以後再用zero-one方法求最佳解[2]。矩陣化減了計算的時間也就大幅的下降了。而在實做的部分主要是根據下面四個觀察到的現象來做推論(1)如果某元件並沒有被要求要測試，在做案例選擇時就可以將他省略掉。(2)如果某個被要求要測試的元件，只有一個測試案例包含他，則那個測試案例一定要被選擇。(3)如果包含元件 $c_i$ 的測試案例一定包含元件 $c_j$ 的話，則元件 $c_j$ 就可以被省略。(4)如果測試案例 $t_i$ 裡面所包含的元件在測試案例 $t_j$ 中都有的話，那 $t_i$ 就可以被省略。基於上述的推論我們可以得到5個化簡的規則，我們將於下面中詳述這5個規則。

#### 4.3.1 五個化簡的規則

歸納上面的推論可以得到五個規則如下，我們將對每一個規則的使用方法做解說，這些都是有經過數學驗證無誤[2]，不過本論文不再證明：

- Rule 1. Surely Satisfied Constraint
- Rule 2. Essential Path
- Rule 3. Dominant Component/ Column

- Rule 4. Dominant Path/Row
- Rule 5. Zero Path/Zero Row

- **Rule 1. Surely Satisfied Constraint**

如果某元件並沒有被要求要測試，在做案例選擇時當然就可以將他省略掉。如下表中元件 3，沒有任何一個測試案例有涵蓋他，也因此就可以將他省略了。

Branch \ Path	C1	C2	C3	C4
P1	0	1	0	0
P2	1	1	0	1
P3	1	0	0	1
P4	1	1	0	0

表 8 Surely Satisfied Example

- **Rule 2. Essential Path**

如果某個被要求要測試的元件，只有一個測試案例包含他，則那個測試案例稱之為必選案例。當某案例成為必選案例時，其所包含的元件在其他的案例中都可以被省略了。因為至少有一個案例包含這些元件了。以表 9 為例 C1 只有 P1 有包含他，因此 P1 為必選案例，而 P1 包含 C1 和 C3 這兩個元件，在整個測試中可以保證至少有 P1 會包含他們，因此就可以將這兩個元件的所在列刪除以化簡矩陣。

Branch \ Path	C1	C2	C3	C4	C5
P1	1	0	1	0	0
P2	0	1	0	1	0
P3	0	1	1	0	1
P4	0	0	1	0	1
P5	0	1	1	0	0

表 9 Essential Example

- **Rule 3. Dominant Component/Column**

如果每一個包含元件  $c_i$  的測試案例一定包含元件  $c_j$  的話，則元件  $c_j$  就可以被省略。因為我們可以保證只要選取任何一個有包含  $c_i$  的測試案例一定會有包含  $c_j$  也因此就可以省略  $c_j$  了，不過這個反之就不一定成立。以表 10 為例，在表中可以發現只要有包含

C2 的案例就一定會經過 C1，也因此往後我們選到任何一個有包含 C2 的案例也一定會包含 C1，所以我們便可以把 C1 刪除掉而不會影響我們選擇案例的正確性。

Branch Path	C1	C2	C3	C4	C5
P1	1	0	1	0	0
P2	1	1	0	1	0
P3	1	1	1	0	1
P4	0	0	1	0	1
P5	1	1	1	0	0

表 10 Dominant Component/Column Example

● **Rule 4. Dominant Path/Row**

如果測試案例 ti 裡面所包含的元件在測試案例 tj 中都有的話，那 ti 就可以被省略。以表 11 為例，P1 包含 C1 和 C3 而 P5 包含了 C1, C2 和 C3，因此只要選擇 P5 就一定可以把 P1 有的元件都包含了，所以 P1 就可以刪除了。

Branch Path	C1	C2	C3	C4	C5
P1	1	0	1	0	0
P2	1	1	0	1	0
P3	1	1	0	0	1
P4	0	0	1	0	1
P5	1	1	1	0	0

表 11 Dominant Path/Row Example

● **Rule 5. Zero Path/Zero Row**

如果一個測試案例中沒有任何的元件則稱之為空的測試案例。這通常是在執行完第二條規則之後才會發生的狀況。在這種情況只要將空的測試案例刪除即可。以表 12 為例在化簡過程中使 P4 變成了空的測試案例，那我們就可以把 P4 刪除。

Branch Path	C1	C2	C3	C4	C5
P1	1	0	1	0	0
P2	1	1	0	1	0
P3	1	1	0	0	1
P4	0	0	0	0	0
P5	1	1	1	0	0

表 12 Zero Path/Zero Row Example

上述的五個化簡的規則建議依序執行，而且必須重複執行直到矩陣套用每一個規則都沒有作用為止，才能得到最簡的涵蓋率矩陣，也就是在滿足相同條件下大小最小的涵蓋率矩陣。使用這個矩陣用zero-one的方式求最佳解的速度會變的比較快，因為矩陣變小了。該如何整合測試案例、zero-one method以及這五個規則以求得選擇最佳測試案例的步驟如下：

- 第一步：由控制流程圖產生涵蓋率矩陣。
- 第二步：利用上述的五個方式化簡矩陣，如果最後矩陣變成空的，就表示已經得到最佳解了。若沒有則還是需要進行下一個步驟。
- 第三步：利用zero-one方法，以化簡之後的矩陣，求得測試案例的最佳集合。

### 4.3.2 Scenario test cases generation approach

經由上述的五個規則化簡之後，矩陣已經大幅度的縮小了，因此運算的時間也大幅度的加快了。不過在實作的過程中我們遭遇到了一些問題，那就是如果你的系統比較複雜，產生出來的矩陣本身就已經大到無法用電腦來執行 zero-one method，那就更不可能用電腦求最佳解了。就算電腦運算沒有問題，對於測試人員而言維護這些資料也十分困難。很多人會認為那就想辦法把矩陣切開來運算就可以了，可是該怎麼切，應用到測試案例化簡時又會遇到什麼問題？

首先，上述的五個規則，雖然縮短了運算時間，但是無法解決在套用規則之前原始矩陣就已經過大的問題，此時我們就必須要從另一個方向來著手。比較好的方式就是化整為零，將矩陣切割成數個小矩陣，這也是一般常用的方法。對應到執行系統的接受度測試時，可以視為原本是一次測試一整個系統，改為一次只測試一個 function 或是一個畫面。這樣子控制流程圖就會變的很小，相對的矩陣也就變小了，在管理上也相對的簡單容易理解多了。只是運算的次數變多了，運算的時間也就變久了一點，不過還在可以接受的範圍內。

一般來說，在使用者的操作過程中，很多事情會在同一個功能或是同一個畫面中完成的。少部分會有必須要跨越不同的功能或是畫面才能完成的。因此對大部分的系統而言功能和功能之間的關聯性不會太強烈，畫面彼此之間的關聯性則會略高於功能間的關聯性。也因此我們可以將每個功能或是畫面獨立出來建立測試案例，然後再利用彼此之間的關聯性來做整合。而且這個方式也很符合使用者的操作習慣，實際的運作過程如下：

- 假設每一個或多個但同一群組的功能或是畫面我們稱之為一個 module。
- 第一步：在每個 module 中建立操作控制流程圖，然後各自利用上述的五個規則和

zero-one 方法產生最佳化的測試案例。

- 第二步：藉由紀錄使用者操作跨 module 的動作來建立每個控制流程圖之間的關聯性，並以第一步中產生的最佳測試案例為節點產生新的控制流程圖。
- 第三步：利用控制流程圖產生測試案例，再使用上述的五個規則和 zero-one 方法求得最佳解。
- 第四步：比對新的最佳測試案例集是否包含第一步中的案例，若有則將第一步中被包含者刪除。
- 第五步：如果切割成好幾個階層，就必須由下往上重複執行第二步到第四步，直到完成最上層。

### 4.3.3 Illustrative Example

在這一節中我們將利用下面的範例，更清楚的解釋和說明如何完成 scenario test cases generation approach。假設某系統有 3 個 Function，分別為 Function 1、Function 2 以及 Function 3。每個 Function 在各自求得最佳測試案例解之後的圖以及最佳測試案例集合，總共有 9 個案例，如下圖。

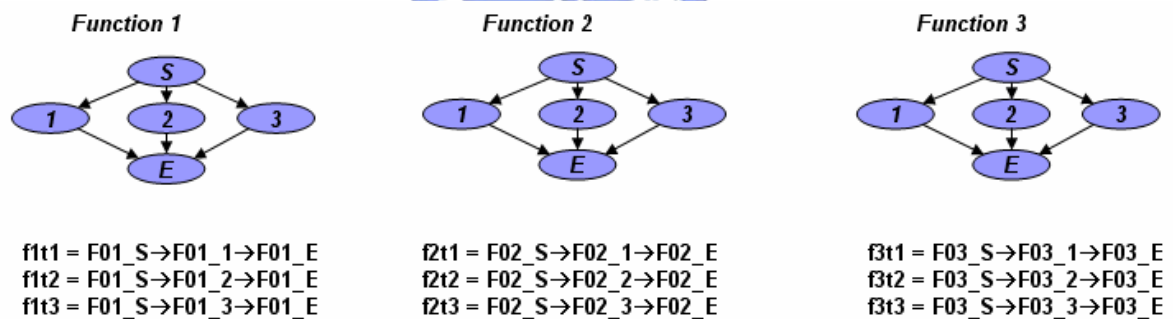


圖 12 Function after reduce test case

假設使用者在跨Function的操作只有一種，而他的操作順序如 $T_G$ ，利用 $T_G$ 所建立的Function之間的關聯圖如下。

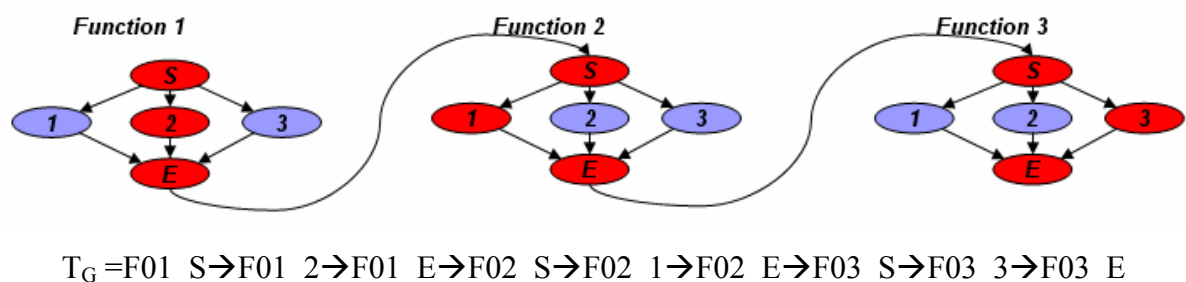


圖 13 Example of scenario test cases generation approach



經過最佳化運算後可得最佳路徑為 $T_G$ ，而 $T_G=f1t2+f2t1+f3t3$ 。因此可以將 $f1t2, f2t1$ 和 $f3t3$ 刪除，因此新的測試案例變成總共有 7 個。

#### 4.3.4 化簡方法的應用與比較

前面提到都是如何改進化簡的方法，在這一節中我們將討論這三者彼此之間的關係和比較。首先我們先列出三者的簡介如下：

- Zero-one方法：是屬於線性規劃中的一種，用來處理在線性等式及不等式組的條件下，求線性目標函數的極值問題的方法。在本論文中用來求取最佳的測試案例集，只要輸入不等式矩陣就可以得到該不等式的最佳解。複雜度是 $mx2^n$ ， $m$ 為待測元件數量， $n$ 為測試案例數量。所需時間主要是隨者測試案例的增加呈指數成長。
- Five reduce rules：利用五個規則化簡矩陣，使輸入 zero-one 方法中的矩陣變小，減少運算時間。複雜度為  $mxn$ ， $m$  為待測元件數量， $n$  為測試案例數量。所需時間隨著待測元件數量和測試案例的增加而呈等比級數成長。
- The proposed solution：提供一個切割系統各自測試，然後再整合的方法，使得輸入的矩陣因分割而變小，但是整合起來可運算的矩陣卻變大。複雜度為  $mxnxo$ ， $m$  為待測元件數量， $n$  為測試案例數量， $o$  為切割後的群組數量。所需時間隨著待測元件數量，測試案例以及群組數量的增加而呈等比級數成長。

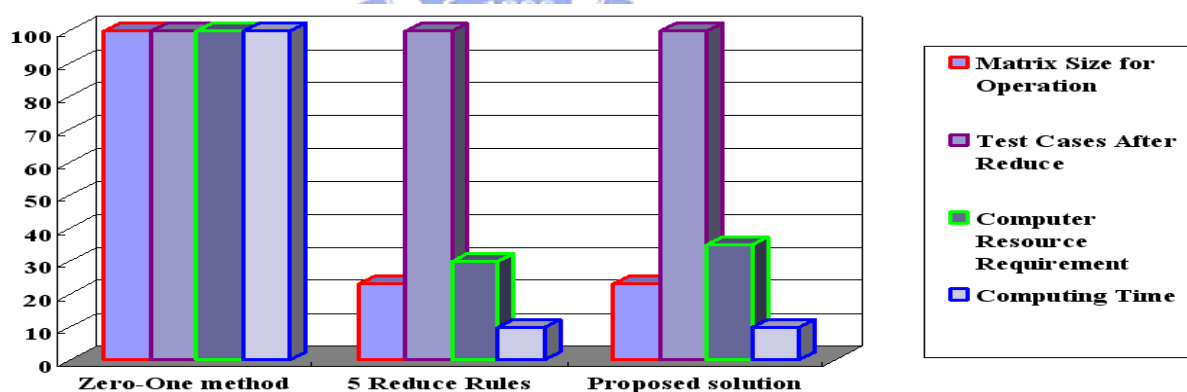


圖 14 三種方法的比較圖

由圖 14 我們可以知道，以 Zero-One 方法為基準，隨者方法的演進，輸入的矩陣越來越小，所需的硬體資源越來越少。但是化簡的數量卻不受影響。而在運算的時間上利用五個規則所需的時間最少，而建議的方式則因為合併的運算部分而多了一些時間，但是仍然比直接將原始矩陣，用 Zero-One 方式運算快的多了。

## 五、實作與應用

### 5.1 背景與動機

在研究探討完了上述的方案後，為了實際驗證理論的正確性，因此實作了一套支援圖形化使用者介面測試案例產生和化減的軟體工具，提供使用者對畫面或是整套系統做測試案例的產生以及化減的功能。其主要目的在驗證上述理論的可行性以及正確性，所以在使用者介面友善度上並沒有做的很好。因為市面上許多工具軟體在這部分已經做的很好了，所以我們便不再做相同的事，我們的目的是使用上述的理論來補足市面上工具軟體上的不足，而非取代。

### 5.2 實做系統簡介

在這一節中我們將列出系統的功能需求，控制流程圖以及所使用的演算法以供參考。

#### 5.2.1 功能需求

##### 2 Toolbar Function Requirements

##### 2.1 New File

- User can open a new file to create screen, nodes, relation and result.

##### 2.2 Save File

- User can save data of screen, nodes, relation and result to files.

##### 2.3 Load File

- User can load data of screen, nodes, relation and result from files.

##### 2.4 Load Screen

- User can load a screen to be used to generate scenario paths.
- Screen can be any size.

##### 2.5 Node Creation

- User can create node on the screen with attribute.
- Attribute can be name, scripts, or rules.
- Name can be numeric or character.



- Script can be combining with other testing tools, for example WinRunner.
- Rules can be condition branch and constraint.

## 2.6 Node-to-Node's Connection Creation

- Operational flow creation based on user's operational behaviors

## 2.7 Independent Scenario Paths Generation

- System can generate all test case bases on the relation of nodes.
- System can select optimal test case set.

## 3 Single Function Mode Function Requirements

3.1 User can create nodes, relation and generate test case...etc for each single function by using toolbar.

## 4 Global Function Mode Function Requirements

4.1 User can create the relation between each function.

4.2 Operational flow creation based on user's operational behaviors

4.3 System will create the diagram of the relation between each function.

4.4 System will select the optimal test case set for the system which is wanted to be tested.

## 5 Test Case Generation Function Requirements

5.1 System must create the entire available test cases base on the user created relation of node.

## 6 Test Case Reduce Function Requirements

6.1 System must reduce the entire test case base on the 5 rules, zero-one method and proposed approach.

## 7 Debug Mode Function Requirements

7.1 User can dump all of the data which is the input or output of all algorithms when debug mode was enable.

## 5.2.2 系統流程

在這個系統中使用者首先必須建立畫面中所有要測試的元件，彼此之間的關係。在這裡所謂的關係是指操作的先後順序，在此我們也預留了加入條件判斷和 script 的空間以便往後和其他測試軟體工具整合。我們提供存取畫面和元件資料的功能，讓使用者可以存取別人建立的檔案。使用者只要建立好元件的關係之後，系統就可以自動產生測試案例以及化減之後的測試案例，在 zero-one method 求解的部分我們是使用 LINDO 這家公司的 DLL 以求得最佳解。另外本系統也提供了 Single function mode 和 Global function mode 兩種模式供使用者選擇。所謂的 Single function mode 就是當使用者只要測試一個畫面，或是要測試的功能很簡單的時候就可以使用這個模式。而 Global function mode 則是當要被測試的系統有多個畫面或是較為複雜時就可以使用此模式。本系統的控制流程圖如圖 15。

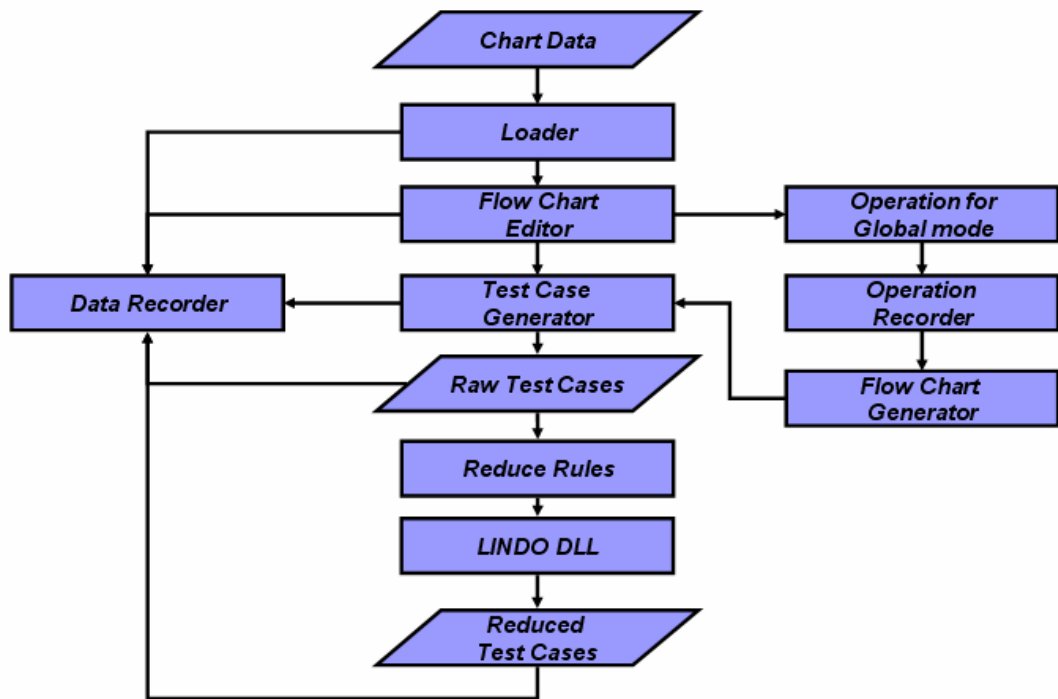


圖 15 System Diagram of the Proposed Tool

## 5.3 範例和資料分析

在這一節中我們將以 e-bay 的註冊網頁為範例，在這個範例中包含了使用者如果去註冊可能發生的情況，包括正確以及錯誤畫面，總共 7 個畫面。我們將簡單的藉此說明系統的運作原理和操作方式，以及分析最後的結果和統計資料。

### 5.3.1 Single Function Mode

如果使用者只想測試單一個畫面時，就可以使用這個模式。

#### Step 1: 讀取要測試的畫面

首先你必須將你要測試的畫面擷取並存成圖檔然後用這個工具開啟(如圖 16)，這個動作主要是讓你在建立元件和彼此之間的關係時可以更容易作業。

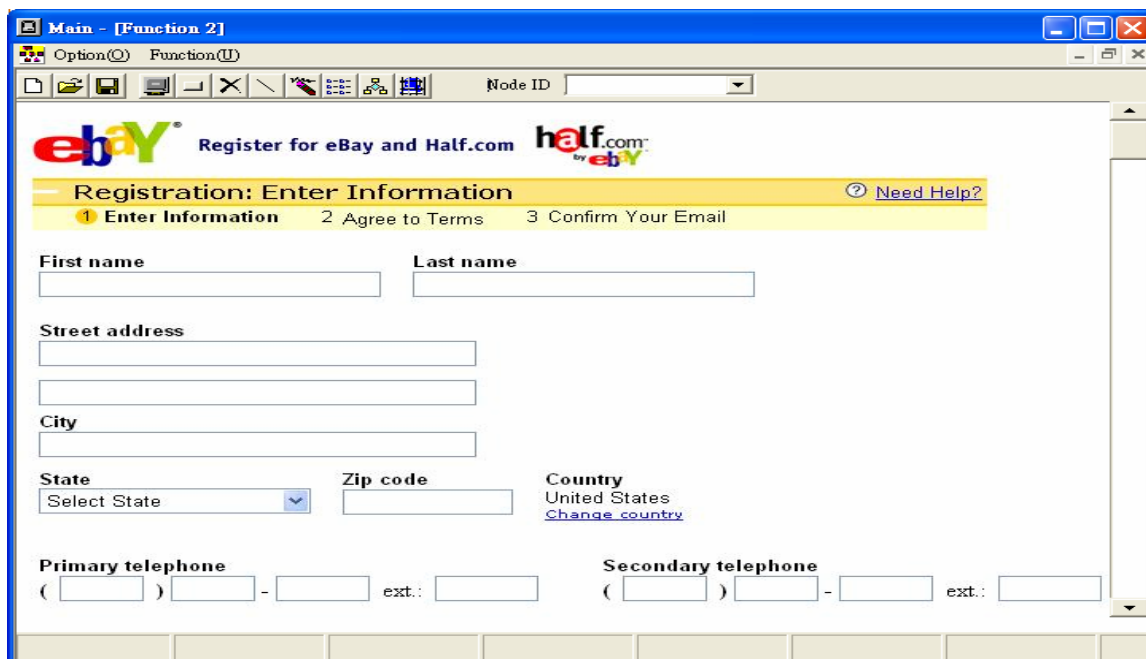


圖 16 讀取要測試的畫面

#### Step 2: 建立要測試的節點

接下來的步驟是建立節點，在這裡所謂的節點就是使用者可能會操作到的動作或元件。例如圖 17 中的 First Name 是一個輸入(Input)的欄位，使用者可能會在這裡輸入各種值，在這個範例中我們假設會有正確和錯誤這兩種值，前面的章節有提到本論文並不對輸入的測試資料做討論，這個部分已經有工具可以從資料庫讀取各種數值輸入到測試案例中的功能了。所以我們只建立節點和在未來將在節點中加入前面提及的限制規則，用來限制不同的資料走不同的路徑。另外在畫面中如果沒有特定的節點是一定會先執行或是最後執行的狀況下，使用者必須建立 START 和 END 這兩個節點使畫面中有固定的進出點。

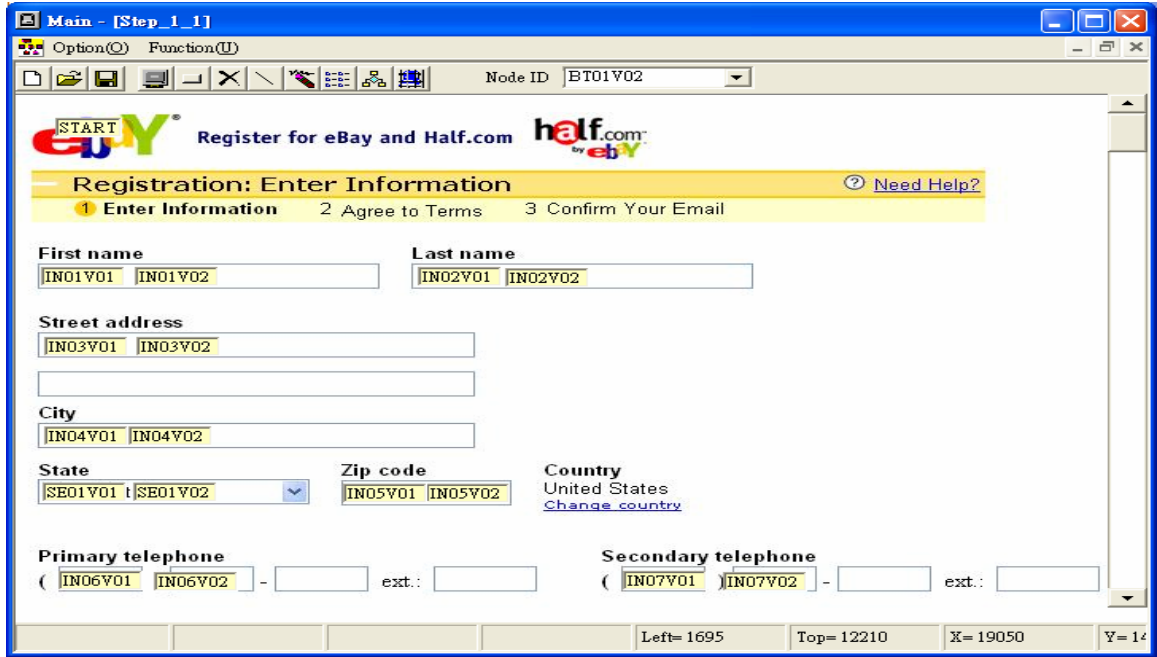


圖 17 建立要測試的節點

### Step 3: 建立點到點之間的關係

在建立完所有要測試的節點之後，下一步就是建立每個節點之間的關係了(如圖 19)。前面有提到就是所謂的操作順序的關係了，如果通常使用者是先操作元件 A 再操作元件 B，那彼此之間的關係就是  $A \rightarrow B$  了。有了節點之間的關係以後系統就可以藉此產生測試案例了。

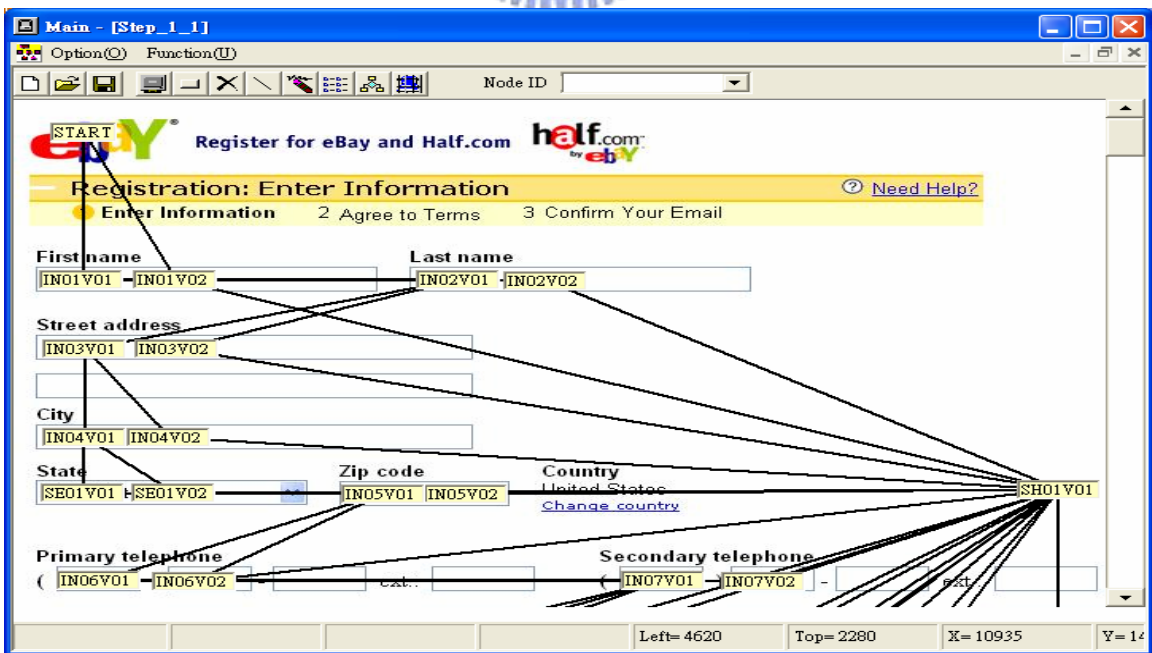


圖 18 建立點到點之間的關係



Coverage frequency matrix :

在這個系統中我們有提供[Debug Mode]，開啟這個模式會將運算過程中所有的，輸出資料全部顯示出來，在這裡我們只秀出最後的矩陣，圖 21 中左邊顯示的是所有被選到的測試案例，而上面則是所有的節點。案例右方方格中著色者表示測試案例有經過該節點，而最下面的方格中有著色者表示該節點至少有一測試案例有包含該節點。由圖中我們可以很清楚的得知每個節點至少都有被測試過一次，確實滿足了我們對涵蓋率的需求。

### 5.3.2 Global Function Mode

如果待測系統是比較複雜的系統或是有許多需要跨畫面操作的流程，這個時候我們就需要使用 Global Function Mode 了。首先使用者還是要先利用 Single Function Mode 將每一個畫面的元件關係圖建立完成並產生測試案例後後才能開始使用 Global Function Mode。

當所有待測的畫面內元件的關係和測試案例都完成建立之後，使用者便可以開啟 Global Function Mode。開啟後使用者可以就經由操作跨畫面的動作來建立每個畫面之間的關係。使用者操作的過程中，系統會紀錄和比對測試案例，當使用者完成操作後每個畫面之間的關係就會被建立起來了。系統會新增一個畫面專門建立案例之間的關係，裡面會紀錄，哪幾個測試案例可以串聯以及先後關係，在 Single Function Mode 中的節點是待測畫面中的元件，而在這裡則是每一個畫面中的測試案例。會以最佳化後的測試案例為優先挑選對象，若無則再挑選非最佳化的案例。將所有的關係建立之後會產生如圖 22 的關係圖，相同的也只要按下產生測試案例的鈕就可以得到最佳解了。

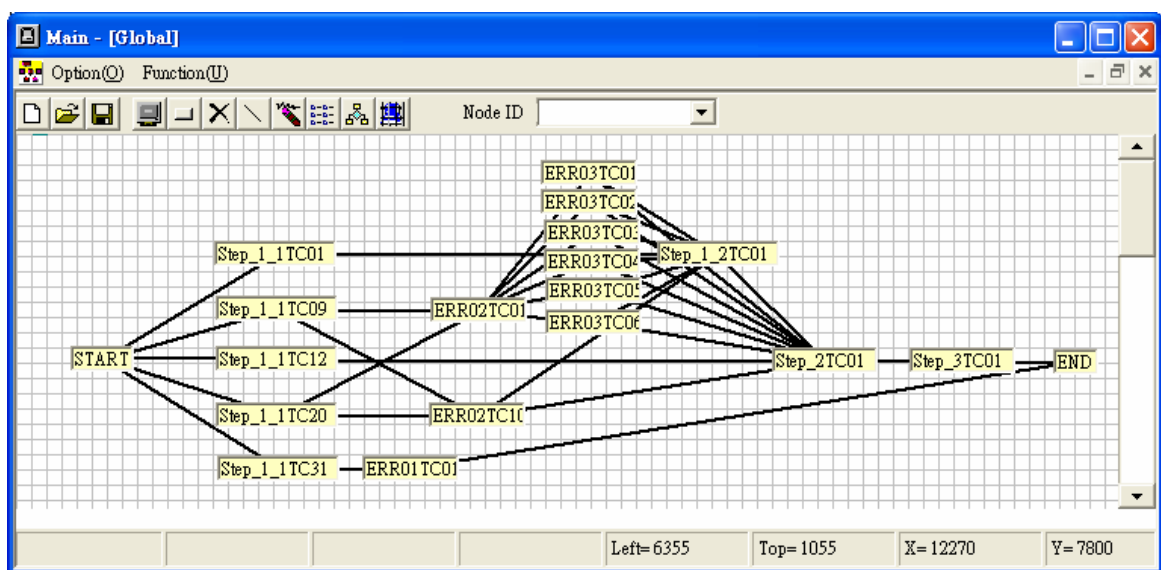


圖 21 Global Mode Diagram



圖 22 產生出來的結果如下：

Case1: START→Step\_1\_1TC01→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case2: START→Step\_1\_1TC12→Step\_2TC01→Step\_3TC01→END

Case3: START→Step\_1\_1TC09→ERR02TC01→ERR03TC01→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case5: START→Step\_1\_1TC09→ERR02TC01→ERR03TC02→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case7: START→Step\_1\_1TC09→ERR02TC01→ERR03TC03→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case9: START→Step\_1\_1TC09→ERR02TC01→ERR03TC04→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case11: START→Step\_1\_1TC09→ERR02TC01→ERR03TC05→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case13: START→Step\_1\_1TC09→ERR02TC01→ERR03TC06→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case29: START→Step\_1\_1TC20→ERR02TC10→Step\_1\_2TC01→Step\_2TC01→Step\_3TC01→END

Case31: START→Step\_1\_1TC31→ERR01TC01→END

上列中所有的測試案例都可以將他從 Single Function Mode 中刪除，因為這裡一定會測過一次，所以在 Single Function Mode 中就不需要再重複一次了。也因此對整套系統的測試除了更完整之外，測試案例的數量也是最少量的。

### 5.3.3 統計與分析

最後對這個範例在使用新的方式前後的數據做一些說明和分析。在表 13 中最左列表示每個功能的編號，最上一列則是說明下面欄位所代表的意義。從表中我們可以看到就整體而言每個功能單獨化減時的化減率是低於整合後的化減率。在此就證實這個方法是有用的，的確不只可以對單獨的 Function 求最佳的測試案例集，也可以求出整各系統的最佳案例集。

在表中你會發現在 Single Function 的部分有些的化減率是零，由圖 23 可以更清楚的看出來。原因是該 Function 操作流程太簡單，只有極少數的動作。這種情形的 Function 通常化減率都很低，而這也是一個負面的案例，說明了太簡單的 Function 就不適用這套理論了。

Function No.	Matrix Size Before Reduce	Matrix Size After Run 5 Reduce Rules	Matrix Size Reduce Rate (%)	Case Number Before Reduce	Case Number After Run 5 Reduce Rules	Case Number After Integrate	Case Reduce Rate (%)	Improve Reduce Rate By Integrate (%)	Total Reduce Rate After Integrate (%)
Step 1-1	322	108	66.46	31	20	17	35.48	9.68	45.16
Step 1-2	30	0	100.00	10	10	9	0.00	10.00	10.00
Step 2	1134	440	61.20	9	3	2	66.67	11.11	77.78
Step 3	42	0	100.00	1	1	0	0.00	100.00	100.00
Err01	168	16	90.48	1	1	0	0.00	100.00	100.00
Err02	448	352	21.43	11	6	4	45.45	18.18	63.64
Err03	171	0	100.00	7	7	1	0.00	85.71	85.71
Global	589	140	76.23	31	10	10	67.74	0.00	67.74
Total	2904	1056	63.64	101	58	43	42.57	14.85	57.43

表 13 Statistic Data of Each Function

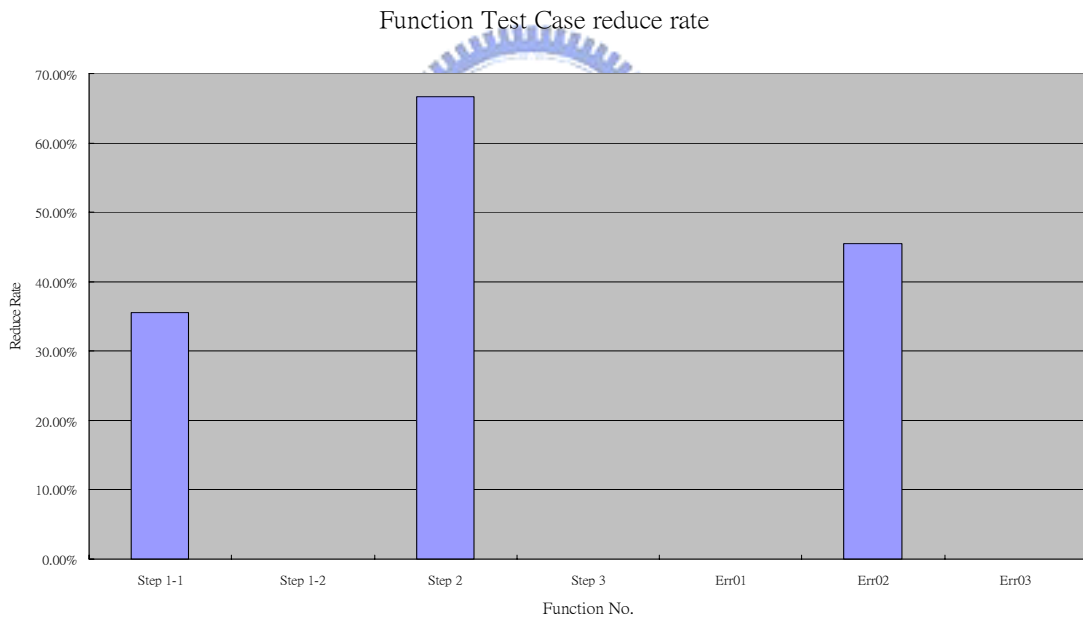


圖 22 Bar Chart of Single Function Test Case Reduce Rate

不過上述的問題在 Global Mode 中，雖然沒有完全化解但是有了改善。因為上述的測試案例中如果在 Global 中有被包含的話，最後也會被化減，也因此提高的不少的化減率。除了這些特例之外，其他的 Function 在使用 Global Mode 之後也都有不同程度百分比的成長。如圖 24。

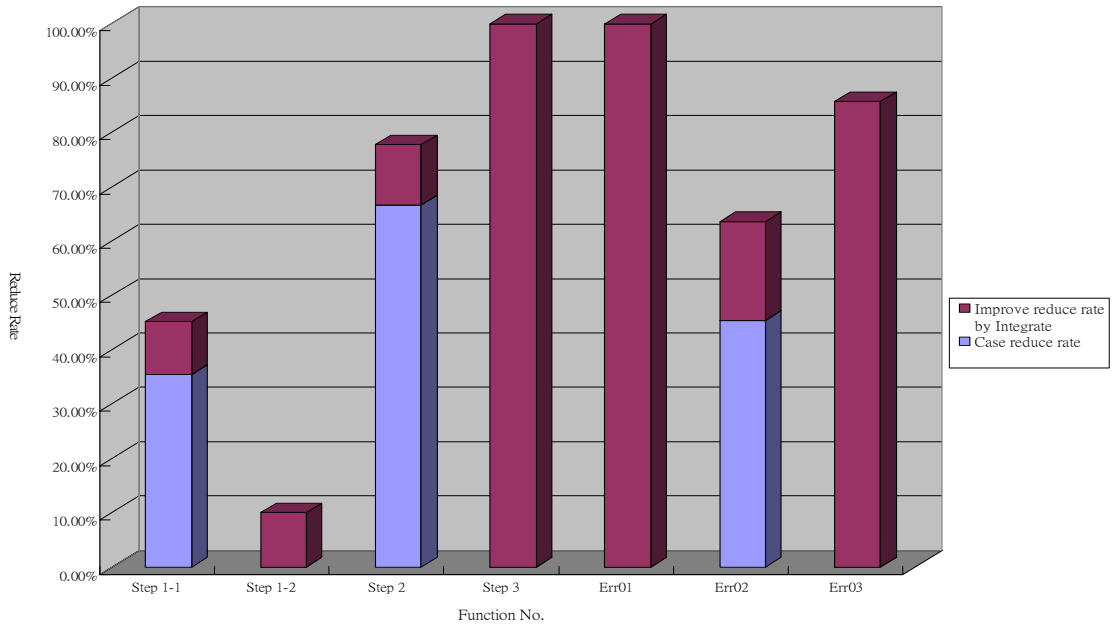


圖 23 Bar Chart of Single Function Test Case in Global Mode Reduce Rate



## 六、結論

### 6.1 結論

大量的測試案例並不能保證測試的完整性，一個完整的測試需要的是完整的測試案例。市面上的軟體測試工具大都強調自動化的執行使用者建立的測試案例、友善的操作介面、完善的測試報告網頁以及測試進度管理的部分。

再好用的工具如果無法完全降低使用者的負擔，還是有缺陷的。很多測試案例的產生是有一定的規則，也可以說是另一種的排列組合。這些都有固定的演算法可以完成，而最佳測試案例集的選取問題也有許多的解決方案。這表示這些都可以藉由電腦來完成這些工作，而不用浪費太多的人力資源在這上。

本論文最主要的目的並不是設計一套取代目前市面上軟體測試工具的理論和工具，而是針對市面上各種軟體黑箱測試工具中共同欠缺的，產生最佳化測試案例集的部分。參考現存的理論，改變後提出一套方法。使軟體測試工具的自動化更完整。使測試人員可更快的產生完整的最佳化測試案例集。

### 6.2 Recommendations for Future Work

在未來的研究部分可以試著將 XP(eXtreme Programming)中，所謂的”先測試，後編碼”[14]，也就是在開發系統之前先建立測試案例的理論套用進來，看看是否可行以及是否有什麼需要改變的地方。

在未來的工作部份最主要的是，改善使用者的操作介面，讓使用者能夠更容易的建立節點的關係。可以參考市面上其他軟體測試工具的功能和介面，商業產品在這方面還是做的很好的。或者是與其他市面上的軟體測試工具做整合。畢竟本論文中時做出來的工具主要是補足市面上測試工具的不足之處，相對的在使用者操作的介面上就做的不甚完善。若兩者能夠整合起來將會是一套十分完善的軟體測試工具。

最後一點便是目前實做出來的工具只能建立物件和其前後順序的關係，日後若能將邏輯判斷以及設定物件屬性和參數傳遞等功能補齊，必能使整套工具更為完整也更為好用。

## 參 考 文 獻

- [1] Ian Sommerville, Software Engineering, 6<sup>th</sup> Edition, Addison-Wesley, 2001
- [2] Jen-Gaw Lee, A Study of Optimal Test Case Selection Problem, Ph.D. dissertation, Department of Information Engineering and Computer Science, NCTU, Taiwan, ROC, July, 2000.
- [3] Jim Dougherty and Keith Haber, Test automation: Reducing time to market, International Conference on Software Testing Analysis & Review, Anaheim, CA USA, November 4-8, 2002
- [4] David Stuppy, Mark Reinig, Kevin C, Verification and Validation Survey/Tutorial on Model-Based Test Generation, Rea Kansas State University CIS 841, 1998
- [5] C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, Sep. 1979, pp. 520-529.
- [6] Joe deSpautz, Quantifying the benefits of automation, ISA Transaction, No.33, 1994, pp.107-102
- [7] Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999
- [8] *Yashwant K. Malaiya* Automatic Test Software, Computer Science Department Colorado State University, 2003
- [9] T.Y. Chen, M.F. Lau, On the divide-and-conquer approach towards test suite reduction, Information Sciences, 2003, pp.89-119
- [10] Boris Vaysburg, Luay H. Tahat, Bogdan Korel, Dependence Analysis in Reduction of Requirement Based Test Suites, ACM, 2002, pp.107-111
- [11] <http://www.softwareqatest.com>
- [12] <http://www.edp.ust.hk/math>
- [13] <http://www.mercuryinteractive.com>
- [14] Kent Beck, Extreme Programming Explained, Addison-Wesley, 2000
- [15] Software reuse, ACM Computing Surveys, June 1992, pp.24-26
- [16] E. Gamma, R Helm, R. Johnson and J. Vlissides, Elements of Reusable Object-oriented Software, Addison-Wesley, 1995
- [17] R. V. Binder, Testing Object-oriented System, Addison-Wesley Longman, 1999
- [18] T. Yamaura, How to design practical test cases, IEEE Software, 15(6), November 1998
- [19] A.K. Onoma, W-T Tsai, M. Poonawala and H. Suganuma, Regression testing in an industrial environment, Comm ACM, 41(5), May 1998

- [20]B. Beizer, Software Testing Techniques, Van Nostrand Rheinhold, 1990
- [21]Karine Arnout, Xavier Rousselot, Bertrand Meyer Automatic test case generation based on Design by Contract, ETH, June 2003
- [22]Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Automated Test Oracles for GUIs, FSE, 2000
- [23]Atif M. Memon, Mary Lou Soffa, Martha E. Pollack, Coverage Criteria for GUI Testing, FSE, 2001
- [24]Atif M. Memon, Martha E. Pollack, Mary Lou Soffa, Using a Goal-driven Approach to Generate Test Case for GUIs, ICSE, 1999
- [25]<http://www.segure.com>
- [26]<http://www.ibm.com>
- [27]<http://www.compuware.com>



# 附錄一：A GUI-based Scenario Testing Case Generation and Reduction Tool User Manual

## 1. Install

Step 1: Install Lindo API in [Driver]:\GSTCGR\dll.exe

Step 2: Install Tool in [Driver]:\GSTCGR\Setup.exe

## 2. Operation

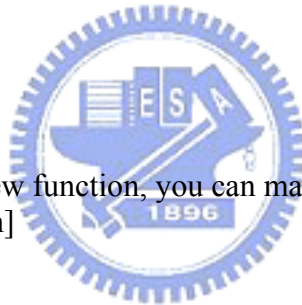
### 2.1 Single Function Mode

There are 2 kinds of function modes; one is [Single], the other is [Global]. Default is Single Function Mode.

[2.1.1 Open a new function]



Whenever you have to add a new function, you can make selection by following steps.  
[Function]→[AddNewFunction]

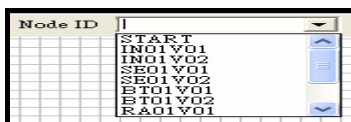


[2.1.2 Add a new screen]



You can press this button to add the picture which was captured from program user interface. It's optional for you to add or not.

[2.1.3 Select NodeId]



You can select or input Node ID by which you want, and the suggested naming rule was as follows; for example, [IN] means input while [V01] stands for the first value. Therefore, the first inputting area for node can be named as [IN01V01]. Besides, [SE] means selecting and [BT] means button; all of the naming rule can be defined at [APP\_PATH\CASEGENERATOR.INI]. However, there are 2 special nodes you have to add; one is [START], which stands for entry; the other is [END] which means to exit all nodes.

#### [2.1.4 Add a new node]



After selecting Node Id, make sure you can press this button to add a new node.

#### [2.1.5 Delete a Node]



When you want to delete a node, you must press this button to enter deletion mode, and then every node you click will be deleted until you press this button again to disable deletion mode.

#### [2.1.6 Add a new Line]



When you want to add a new line, you must press this button, and then press “from” node and “to” node. It means that the user will operate from “from” node first and then to “to” node. Press this button again will disable the function of adding new line.

#### [2.1.7 Remove Line]



You can press this button and click the “from” node and “to” node to remove the line between those 2 nodes.

#### [2.1.7 Generate Test Case]



You can generate all testing case by pressing this button.

#### [2.1.8 Optimal Test Case]



You can get the optimal testing case by pressing this button; however, before doing this, you must generate all testing cases first.



[2.1.9 Line switch]



You can switch the line shown on the screen or not by pressing this button.

[2.1.10 Save]



You can save all data on the screen.

[2.1.11 Load]



You can load the file which you saved before.

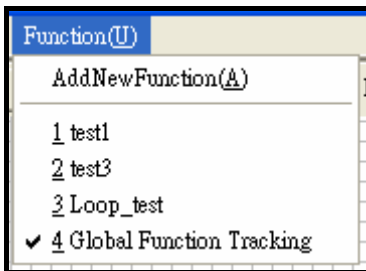


## 2.2 Global Function Mode



You can enable global function mode by selecting [Global Mode]

You can switch function by clicking [Function] of function list.

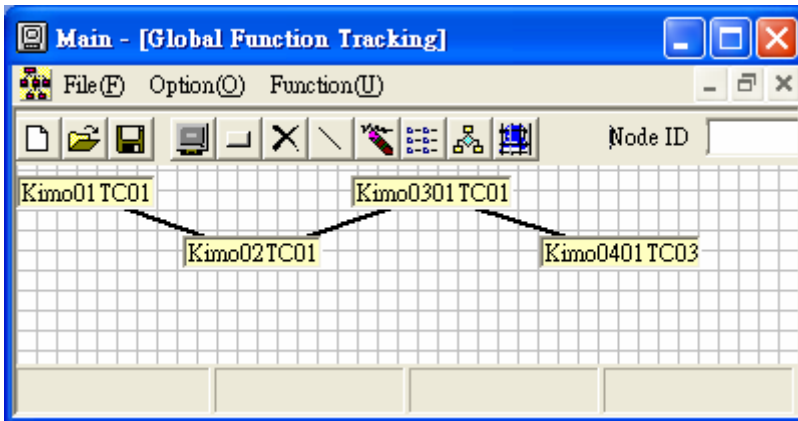


Step 1: Select the first screen you will run; then click the node and follow the sequences that you may do as usual. When clicking the node in the program, the next available nodes you can click will be shown as green color. If there is no green node, it means that the case in this screen is finished, and a new node will be added in function [Global Function Tracking]. Then go to step 2.

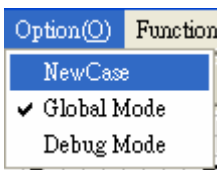


Step 2: After you finish one screen, you can switch to next screen and follow the sequences you run the program as usual.

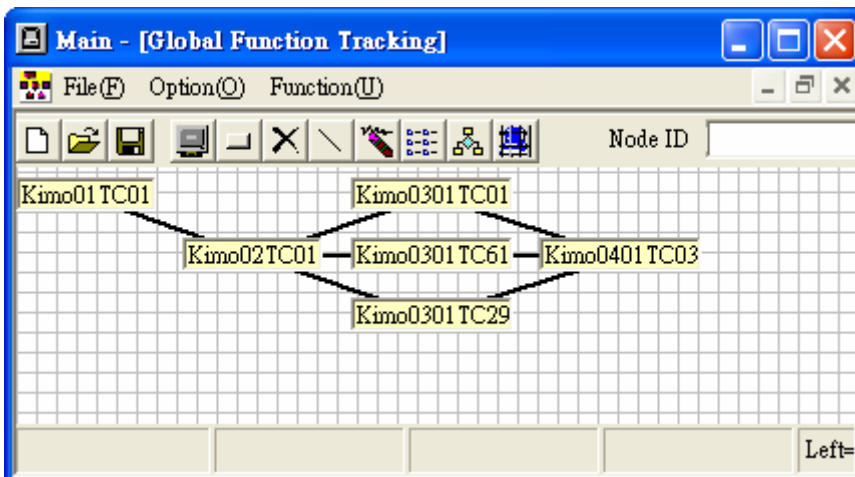
Step 3: After finishing all screen, it means that you have already created a new testing case. And you will see a path graphic in function [Global Function Tracking].



Step 4: If you would like to create more testing cases, you can select [Option] in function list after creating a case; it means that you will start creating another new test case. Then repeat the steps from 1 to 3. You do not need to start from first screen every time. You can skip the same path and run different path only.

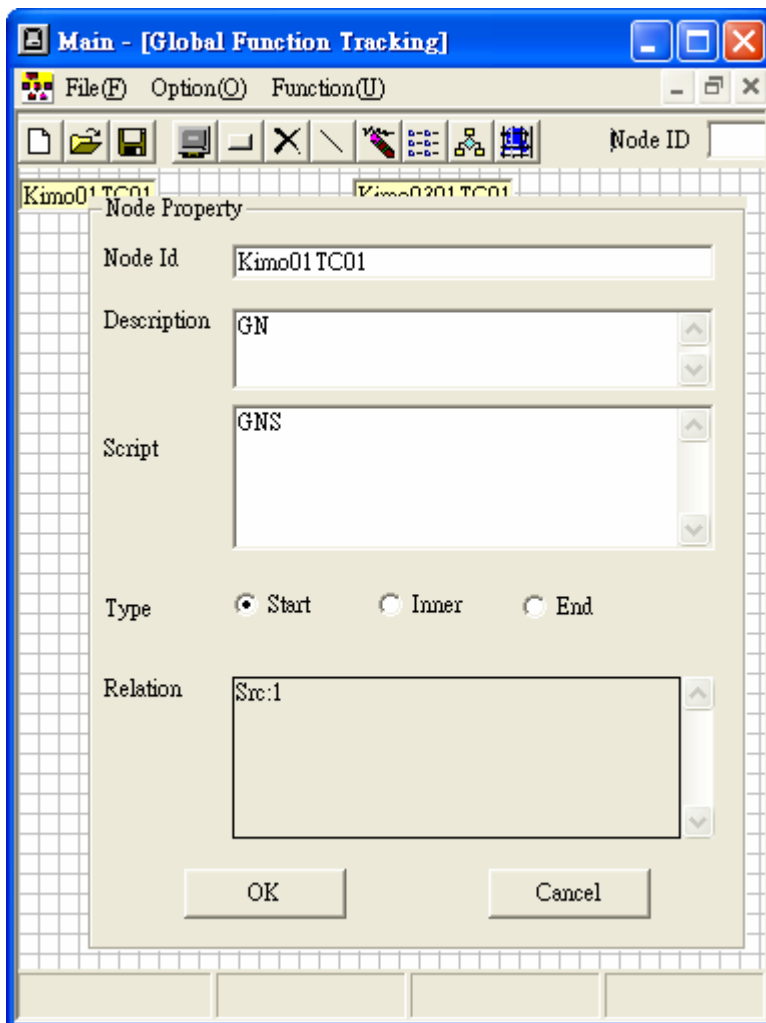


Step 5: After creating all testing cases you want, you will see the diagram similar to the below one.



Before some more testing cases generated for you automatically, you still need to set [Start] and [End] node by moving the cursor to the node, clicking mouse right button to select [Type] as [Start] for start node, [End] for end node, and then press [OK] button. Besides, you can add 2 new nodes by selecting [START] and [END] from Node Id list. And then link [START]

node to all of the entry nodes while the entire exit nodes link to [END] node.



Step 6: After setting you can press [Generate Test Case] button to generate more testing case. And press [Optimal Test Case] to get the optimal solution for testing the system.