國立臺灣師範大學
資訊工程研究所碩士論文

指導教授： 黃冠寰　博士

對於單一通道多個接收者非同步訊息傳遞程式的動態測試

Dynamic Testing for Single-Channel -Multiple-Receivers Asynchronous Message Passing Programs

研究生：　周柏宏　撰

中華民國　一百零一　年 六 月

摘要

軟體測試(software testing)是軟體工程一重要的階段，產品上市之前藉由軟體測試可以讓工程師驗證是否與當初設計的原意符合，也可以找出許多產品上的缺陷，在循序結構(sequential)的程式中做測試往往得到的結果都是可以預料的，但是在並行結構(concurrent)執行時，若程式規模較大所產生的結果就未必能夠預測。

所謂並行程式是指包含多個行程(Process)或是執行緒(thread)同時執行，並且完成所指派某些的任務，常見的並行程式測試的方法包含非決定性測試(Non-deterministic Testing)、決定性的測試(deterministic testing)、可達性測試(Reachability Testing)，前面兩者第一個主要缺點是程式執行測試所找到的交錯(Interleaving)的涵蓋率極低，第二個缺點是在於按照自訂的順序所執行，產生較繁瑣，第三個方法去除了上述兩個缺點，且能夠找到程式執行時的交錯(Interleaving)也能完全找到。

本論文主要是在以單一通道多個接收者非同步訊息傳遞為基礎的程式(Single-Channel-Multiple-Receivers Asynchronous Message Passing-Based Programs)上做動態測試，由於非同步訊息傳遞為基礎的程式可能存在潛在的競爭情況(message race)，若在設計程式時未考慮到有非決定性的行為(non-deterministic behavior)，所產生的結果可能會造成嚴重的損失，甚至導致整體的系統停擺，我

們使用的架構是採用可達性測試(Reachability Testing)，可達性測試結合了非決定性測試以及決定性測試的特性，可以根據收集來的資訊，有效的分析競爭情況，進而導出執行時各種競爭情況可能產生的執行順序，但是單純使用傳統的可達性測試在非同步訊息傳遞程式上做測試可能會有死結(Deadlock)問題，因為可達性架構所設計的情況是針對 share memory 裡的 read 和 write 事件，所以必須針對整體架構做大幅度的修改，才能方便在受測程式上做動態測試(Dynamic Testing)，且可針對欲分析的情況做紀錄特定的資訊，將分析結果收集判定是否存在錯誤情況的發生。

關鍵字：並行測試、並行程式、非決定性行為、非決定性測試、決定性測試、可達性測試、非同步訊息傳遞程式

Abstract

The Software testing is a one of phase of software engineering. This can find out defects on Products, In the sequential program testing, the results often can be predicted, but in the concurrent testing results are always unpredicted.

The concurrent program is composed by more than one Processes(or Threads) to complete the assigned tasks at the same time ,The common testing methods include Non-Deterministic Testing、Deterministic Testing and Reachability Testing , The Non-Deterministic Testing lets the Programs free running on Testing , but It may only show part of situation . Not all cases, Deterministic Testing of Concurrent Program P is carefully select a set of tests (X,S), and the forced execution determines whether S is feasible for P , But deterministic testing has additional problems to solve. One major problem is deciding which pairs of inputs and SYN-sequences to select for a concurrent program.

In this paper, I will study how to use reachability testing in dynamic testing of Single-Channel-Multiple-Receivers asynchronous message passing programs. Because the reachability is used in test shared memory read/write event. If I use the reachability testing on asynchronous message passing programs. It will be deadlock, It is necessary for architecture to make big changes. Finally, I will use a modified

version of reachability testing method on asynchronous message passing programs, and collect execution information to do analysis of whether there is message competition.

Keywords : Concurrent Testing 、Concurrent Program、Non-Deterministic behavior、Non-Deterministic Testing、Deterministic Testing、Reachability Testing、Asynchronous message passing programs

誌謝

在這裡首先要感謝我的家庭，包含我的姑姑、爸爸、奶奶，要不是他們的支持與鼓勵我可能不會那麼順利在大學畢業間攻讀碩士，家人總是在我遇到抉擇時給予我適時的建議，再來就是感謝我的碩士班指導教授黃冠寰老師，老師常常提醒我有關做研究的方法以及建議，在碩士班期間吸收到在大學時期未得到的知識，以及適時的糾正我的錯誤，這對我出去社會工作將會是非常有幫助的經驗，接著要感謝我的實驗室的學長(姊)哲生、宇程、李琪、邵彥、涵孟還有同學恆毅以及學弟震宗、思鋒、翊展、偉賢，博班學長對於實驗室的一些如器材的使用，帶領我快速熟悉實驗室的環境，宇程學長則提供我一些程式的建議，已畢業的碩班學長姊紹彥及涵孟則提供我修課方面建議，以及實驗室一些注意事項，同學恆毅因為是研究相同領域所以常常遇到一些問題時會請教他，他也不厭其煩的指導我，包含一些小細節總是能提醒我，學弟們總是在實驗室扮演著開心果的腳色，讓實驗室的氣份不至於太緊張，在這也預祝學弟們明年能夠順利畢業。

最後感謝 105 實驗室的宗賢學長，以及林順喜教授，前者給予我一些找工作上的建議，後者則是在修演算法課程當我有問題時總是不厭其煩的指導我。

周柏宏誌於

國立臺灣師範大學資訊工程研究所

民國一百零一年六月

目錄

附圖目錄

附表目錄

1. Introduction

在許多軟體尚未上市之前，必需考慮一個重要的階段，這個階段就是軟體測試(Software Testing)[17]，為何需要做軟體測試？主要原因為在軟體的發展過程中，可能因為與溝通不良造成規格不符或設計錯誤或是程式在撰寫時發生疏漏，軟體測試在軟體工程的生命週期為其中之一重要的階段，測試主要的作用希望能夠找出軟體是否與當初設計原意不符或是找出產品上的缺陷，提早在軟體上市之前修改軟體，根據研究指出產品在交付給客戶後所需修改軟體錯誤的成本約為交付前的 10 倍，軟體的品質越高之後所帶來維護的成本則會越低。

1.1 Concurrent Program

在循序(Sequential)結構的軟體上做測試,所得到的結果往往是容易預料的，即得到的結果可能只有一種，但在並行(Concurrent)架構的軟體測試中，可能產生的情況則無法預料，並行架構包含分散式系統(Distributed System)，或是在單一電腦環境下，分配多個並行程式或是執行緒同時來完成指定的任務，並行程式的出現帶來了執行效能以及生產力的增加，但卻可能存在潛在非決定性的行為(Non-deterministic behavior)[1]、[2]問題，所謂非決定性的行為是指有一並行程式(Concurrent Program)P 包含輸入 x，將 P 拿來做執行多次，可能會產生多種不同的結果，而因為並行程式存在潛在的非決定性行為將會出現多種可能的結果，而

這些結果會導致一些問題，如:相同輸入 x 代入並行程式 P 執行多次所得到的結果都如預期，但是在未來的執行中難保不會出現一次未預期的結果，對於這個情況若出現在已上市的產品上可能會影響系統整體的運作，損失難以估計，因為並行程式的不確定性使得整體難以測試。

在並行程式的測試方法中主要分成數個，第一、非決定性的測試(Non-deterministic Testing)，非決定性測試有幾個方式來測試，第一種方式為最原始的測試方法，採用讓程式自由的去運行，運行多次希望找出程式可能潛在的錯誤，但這個方法得到的結果可能為某些部分的執行情況一直出現，而某些部分的執行情況卻從未產生，也就是為了要找出程式可能潛在的錯誤下，這個方式所得到的執行情況涵蓋率不高，另一個非決定性測試的測試方法採用插入隨機延遲(Random delay)在受測程式每個程式碼的前後位置上，這樣的測試方式使得程式的執行情況涵蓋率提升，但只能在小規模的程式得到較佳的涵蓋率，若是測試在大規模的程式得到的涵蓋率也無法達到百分之百，且由於插入延遲時間的關係，導致整體的測試時間花費極大，第二、決定性測試(Deterministic Testing)，決定性測試的目的為提供一仔細選擇過的序列按照此序列要求的執行順序讓程式執行，根據執行的結果來判定是否為錯誤的情況，但也存在一個缺點就是若在小規模的程式其執行情況較容易人力掌握，若是在大規模的程式執行情況則無法靠人力掌握，第三、模組檢查(Model Checking)[3]，模組檢查工具是用於軟硬體方面的驗證，給予系統屬性(System property, 又稱作 Specification)以及系統需求

(System requirements，又稱做 Model)帶入模組檢查工具，並會得到一解答有關於驗證是否正確，第四、可達性測試(Reachability Testing)，可達性測試是結合上述兩種非決定性以及決定性測試特性讓在測試階段中先讓程式某部分同步化事件照按照指定的順序執行，接著剩餘未執行完的部分則讓它自由隨機的運作，主要優點在於能夠對並行程式所有的非決定性行為情況達到百分之百的掌握。

1.2 Dynamic Testing For Single–Channel -Multiple-Receivers Asynchronous Message Passing Programs

訊息傳遞[4]是一種溝通的形式使用在如:平行運算(parallel computing)、物件導向程式(object-oriented programming)、行程間的通訊(Interprocess communication)等..，行程(Process)或是物件(object)可以發送訊息到其他的行程，在訊息傳遞程式系統中主要考量為可靠性(Reliability)、安全性(Security)以及訊息是否能按照當初傳送的順序到達，常見的訊息傳遞系統如:Java RMI、Corba、SOAP，在訊息傳遞系統設計上提供最原始運算為基本的發送(Send)以及接收(Receive)來達到溝通的目的，根據這兩種運算主要分為兩種傳輸方式，非同步訊息傳遞(Asynchronous Message Passing)以及同步訊息傳遞(Synchronous Message Passing)，

前者運作方式發送者執行發送運算不需等待接收者接收到訊息就可繼續運作而接收運算則是必須等待接收到訊息才可繼續動作，後者則是發送者執行發送運算需等待接收者接收到訊息才能繼續執行下一步的動作，接收運算則須等待接收到訊息才能繼續動作，本篇論文旨在探討在非同步訊息傳遞且行程之間的溝通是透過一個共同的 Channel，該 Channel 接收的順序是採用先進先出(FIFO)，允許多個發送者以及接收者（Multiple Sender/Multiple Receiver）同時對 Channel 做接收(發送)，然而在這些情況下可能會有潛在的非決定性行為(Nondeterministic behavior)由於訊息競爭(message race)在同一 Channel 上，會出現未預期的結果，這些結果未必是我們所需要的，不幸的是，相關論文上僅有討論對 Multiple Sender/Single Receiver 架構的訊息傳遞程式做測試[7]、[16]，並沒有相關論文討論測試在 Single-Channel-Multiple-Receivers 訊息傳遞程式上做測試。

對於出現在非同步訊息傳遞上的非決定性行為，在論文中採用可達性測試的架構，可達性測試[9]、[10]可有效的克服非決定性行為，對於執行中的可能情況透過可達性測試所能產生的執行情況優於非決定性以及決定性測試方法，並且對其收集來的資訊作分析的動作，可得到該程式可能產生的結果以及這些結果是藉由執行那些執行情況所產生的，這些數據有利於分析判斷程式設計時是否有潛在的非決定性行為以及那些錯誤的結果所執行的情況可以在設計中避免執行到這些部分，但由於傳統的可達性測試並不適用在訊息傳遞環境下的測試，原因歸咎於可達性架構所設計的情況是針對共享記憶體(share memory)裡的 read 和 write

事件，若使用在[9]所提出的方法會出現死結(Dead lock)的情況，必須得針對訊息傳遞程式運算的特性做修改，才得以方便做測試。

本論文建構在非同步訊息傳遞程式的測試沿用舊有的可達性測試機制，像是以前綴為基礎(Prefix Based Replay)技術、競爭情況分析(Race Analysis)，並為了模擬在分散式環境下測試所以增加協調者伺服器(Coordinator Server)，協調者的目的為負責控制在執行測試時共用變數的設定，因為是在分散式環境中事件之間的執行順序必須仰賴第三方來做紀錄。最後我將我所提出的理論，利用 Java 程式語言實作他，並利用數個例子做測試來驗證理論的正確性。

本篇論文組織如下，第二章節會針對並行測試以及訊息傳遞測試上相關文獻做討論，第三章節會提到藉由修改舊有的可達性測試架構，來達到在非同步訊息傳遞程式上做測試，第四章節則會提到如何在執行中收集訊息傳遞程式同步化事件的資訊，第五章節提到將執行時收集的資訊做競爭情況的分析，採用的是競爭分析演算法，第六章節會提到利用 java 將這個測試方法實作出來並且使用其他的測試方法與我設計的方法利用測試一些例子所得到的數據來作比較分析優缺點，第七章節則是結論與未來展望。

2. Background and Related Work

並行程式（Concurrent Program）是指同時間有兩個以上的行程(Process)或是執行緒(Thread)同時運行著指定的任務，但由於有潛在的非決定性行為(Nondeterministic behavior)，使得程式難以測試以及驗證其正確性，本章節大致上將分為兩個部份來做介紹，首先將針對並行程式上的測試方法相關文獻來做探討，在來將介紹關於訊息傳遞程式上的測試方法。

2.1 Concurrent Testing

2.1.1 Nondeterministic Testing

非決定性測試(Nondeterministic Testing)[9]、[11]廣泛被應用在測試並行程式(Concurrent Program)，其執行步驟為將並行程式以及給予輸入重複執行多次，並且檢查每次執行的結果。相較於單一執行所產生的結果，經由多次執行所產生結果可得到不同的同步化事件(Synchronization Event)組合(又稱作同步化序列(Synchronization Sequence))，並且可能偵測到錯誤的出現，非決定性測試的優點如下:

(1) 當要對並行程式做非決定性測試時，不須額外增加工具或是演算法，直接讓程式運行即可，使用上較簡單方便。

(2) 整體測試所需時間較短，相較於其他測試方法。

以上優點讓非決定性測試受到大眾的喜愛,但是還是存在著主要的缺點必須克服,像是相同的同步化事件所組合而成的序列一再產生,而某些部分同步化事件所組成序列卻不曾出現,即執行多次只會出現特定結果,無法達到詳盡的測試。基於這個問題的存在目前有數個方法可以做到改善:

(1) 控制作業系統行程(Process)的排程。

(2) 插入隨機延遲敘述式(Random Delay statement)在程式特定的位置上[12]、[13]。此方法實行的步驟為:

a. 在並行程式上每個敘述式前後插入一隨機延遲時間。

b. 重複執行並行程式多次,在每回合執行完畢檢查其結果。

上述方式可以提升不同的同步化事件組合而成的序列所產生的機率,但仍然無法做到百分之百的涵蓋率,且隨機延遲時間加入可能會造成測試運行時間過久,而失去了原本非決定性測試該有的優勢。

2.1.2 Deterministic Testing

決定性測試(Deterministic Testing)[9]、[10]、[16]為另一種運用在並行程式的測試(Concurrent Testing)方法,主要步驟為 :

(1)給定一測試集合，而這個集合包含數個(X,S)，每次將會取出一個(X,S)來使用，而這個(X,S)表示 X 為輸入，S 為同步化序列(SYN-Sequence)。

(2)將這個每次取出的(X,S)強制並行程式按照(X,S)給定的資訊做決定性執行，並檢查這個 S 是否在並行程式和所給定的輸入下是可行的(feasible)。

這些步驟可以決定是否給定的輸入和同步化序列的組合(X,S)帶入程式後是可以運行並且不會有錯誤的情況發生，而將先前已執行過的(X,S)再次帶入並行程式做決定性的測試，這樣的方式叫做重播(Replay)，決定性測試可有效地快速驗證某種執行方式是否可運行，但仍存在著一些問題必須要解決：

(1) 定義同步化序列：不同的同步化事件或語言中，所需定義的格式也未必相同。

(2) 如何選擇欲測試的(X,S)：在簡單的程式中也許可以推敲出來，但在大型複雜的程式中，難以推敲每個欲測試的(X,S)。

相關文獻已有探討使用決定性測試方法使用在一些並行程式(Ada)或是以信號機(Semaphore)以及 monitor 為基礎的並行程式，更多應用決定性測試的相關文獻可參考[20]、[21]。

2.1.3 Reachability Testing

可達性測試(Reachability Testing)[9]、[10]、[14]、[15]、[16]為並行程式測試(Concurrent Testing)中最能有效達到百分之百涵蓋率的測試架構，最早是使用在共享記憶體(Share memory)Read、Write 事件[9]，因為共享記憶體上的共用變數若沒有經過適當保護多個行程或是執行緒會因為競爭情況而產生非決定性行為(Nondeterministic behavior)進而產生多種結果，在這裡他採用可達性測試來找出多個行程對共用變數存取的情況，之後在[14]、[15]、[16]裡另一作者重新定義一新的可達性測試可用在測試程式中包含訊息傳遞、信號機(Semaphore)、monitor 以及 lock 等會因為競爭情況存在非決定性行為的事件，在[9]之中由於其可達性

測試競爭分析演算法會產生過多的競爭變異(Race Variant)，相同作者在之後[10]提出一新競爭分析演算法，這演算法能大大減少產生多餘的競爭變異，並用來測試在客戶/伺服端(Client/Server)之間的共用資料庫存取情況。

可達性測試大致上分為三個階段，以前綴為基礎重播(Prefix-Based Replay)、競爭分析(Race Analysis)、過濾多餘重複的同步化序列(SYN-Sequence)，以前綴為基礎重播分為兩個階段，replay 以及 monitor phase，monitor 階段採用非決定性測試的特性，以自由隨機的方式讓並行程式受測同步化事件執行，replay 階段採用決定性測試的特性，讓特定事件受到控制順序執行，但單純只有以前綴為基礎並無法做到產生多組不同的情況，必須仰賴競爭情況分析所產生的結果，藉由產生的競爭變異(Race Variant)，使得下一次的以前綴為基礎的執行產生不同情況，而過濾多餘重複的同步化序列則是在每次執行完以前綴為基礎重播階段後藉由比對目前所產生的同步化序列，刪除一些重複的同步化序列，可節省不少空間，可達性測試執行主要步驟如下:

(1) 首先，以自由隨機的方式讓受測程式在 monitor phase 下執行,執行完後會產生一同步化序列(SYN-Sequence)。

(2) 接著根據步驟一所產生的同步化序列，做競爭分析，會產生零到數個競爭變異(Race Variant)，競爭變異存到一集合內。

(3) 將集合內的競爭變異帶入以前綴為基礎重播技術來執行。結果得到一同步化序列。

(4) 過濾重複的同步化序列。

重複執行二、三、四步驟直到存放競爭變異的集合為空。

可達性測試主要優點能有效地對於並行程式做完整的測試,測試過後可得到程式所有可能執行情況,以及可根據在執行過程中記錄下的資訊找出潛在的錯誤,缺點為執行時間較非決定性測試長久。

2.2 Testing of Message Passing Program

訊息傳遞[4]是一種溝通的形式使用在如:平行運算(parallel computing)、物件導向程式(object-oriented programming)、行程間的通訊(Interprocess communication)等..,因為訊息競爭(Message race),所以訊息傳遞程式存在著非決定性行為,使得在訊息傳遞程式測試上極具挑戰,訊息競爭存在主要原因為當分散各處的節點n(n>2),同時對相同目的地做發送/接收,可能因為訊息延遲等原因,得到未預期結果,以下將討論在訊息傳遞程式測試相關文獻。

[6]採用的方法為以非決定性測試(Nondeterministic Testing)方法對非同步訊息傳遞程式做測試,在[6]中提到傳統的非決定性測試方法存在著兩個主要問題,第一、如何識別插入的延遲點(delay point),傳統是以人工的方式插入,但這不是最佳解,反而較耗費時間,第二、測試何時會停止?傳統的方法無法判別,必須以人工的方式判斷,在這篇論文中他提出自動在每回合測試完記錄一些資訊,並做分析得到下一次延遲點該放入程式哪個位置,並且提供詳細的條件判斷測試何時會停止,實驗結果指出在較簡單的程式能達到百分之百的涵蓋率,而在複雜

的程式中測試也較優於傳統方法。

在[18]中，則是單純的對訊息傳遞程式可能存在著訊息競爭，提出三種競爭分析演算法分別考慮在三種不同類型的發送以及接收運算之間的同步(Asynch、FIFO、Causal)，方便用來測試以及除錯非同步訊息傳遞程式，而這些競爭分析演算法也將被用在之後新的可達性測試的論文中。

在[16]、[19]中，則是提到對非同步訊息傳遞程式測試採用方法為可達性測試(Reachability Testing)，找出執行時所有可能的發送以及接收同步化事件所組成序列(SR-Sequence)，而在[19]中針對[16]執行時的效率性做改善，修改產生競爭變異(Race Variant)演算法的部分，使得整體使用可達性測試在非同步訊息傳遞程式上更加迅速。

在[7]、[15]中，作者提出一新的可達性測試，用在測試非同步以及同步訊息傳遞程式上，因為是採用通用模組(General Model)，所以除了上述兩種同步化事件外還可測試包含訊號機、互斥鎖(lock)以及 monitor，使用時間郵戳(time stamp)的方式記錄事件發生的先後順序，並且使用競爭集合(race set)來存放可能有競爭情況的接收(receive)事件，然後再利用一競爭分析演算法產生競爭表(race table)來導出該次同步化序列的競爭變異(Race Variant)，此架構能有效的改善傳統可達性測試存在的問題，不須額外花費空間儲存同步化序列來做比對是否執行產生重複，保證同樣的序列只會出現一次。對整體效能的提升有很大的幫助。

在上述介紹的論文中，以非同步訊息傳遞程式中所滿足的特性為多個發送者

還有單一接收者（Multiple Sender /Single Receiver），訊息傳送到的 Channel 為(Port)，不幸的是，這並不是測試在 Single-Channel-Multiple-Receivers 架構下，本篇目的旨在針對 Single-Channel-Multiple-Receives 架構的訊息傳遞程式做測試，並針對可達性測試架構做修改來達成在非同步訊息傳遞程式上的測試。

3. Design on Asynchronous Message Passing Program Testing

此章節將針對在非同步訊息傳遞程式測試的設計部份作描述,測試的流程，首先對於非同步訊息傳遞測試，對其各個步驟做說明，此次測試是利用可達性測試(Reachability Testing)為基礎並加以延伸應用至分散式環境上的訊息傳遞程式作測試，再來會介紹整體測試的設計架構，測試的設計架構是建構在分散式的環境中。

3.1 Framework

在這小節將針對修改可達性測試方法測試在非同步訊息傳遞程式的設計，其中所需要執行到的步驟以及流程做大致上的介紹,對於測試在非同步訊息傳遞程式，如圖 3.1，開始會對其整體測試作一個初始化的動作，接著將初始化的所產生的 Race Variant[1]以文字檔的型式,儲存至儲存裝置中,在第一次執行可達性測試(Reachability Testing)時，首先，代入一個空的競爭變異(Race Variant)到以前綴為基礎重播(Prefix Based Replay)，由於競爭變異為空值，在以前綴為重播階段會以自由隨機的方式讓分散式環境下的訊息傳遞程式執行，執行完畢過後會產生一

[1] Race Variant : RaceVariant 為 部份的 SYN-Sequence ,即將 SYN-Sequence 做競爭情況(Race Condition) 的分析,所產生的一些同步化事件(Synchronization Event),可以使下次代入 Prefix-Based Replay 能產生新的一組同步化序列(Synchronization Sequence)。

SR-Sequence[2]如圖 3.1 步驟一，而 Prefix Based Replay 執行模式分為兩個部份:

a. 重播模式(Replay Mode)：此模式用意在控制一些事先決定好必須按照特定順序執行的 Event，能夠按照順序執行。

b. 監視模式(Monitor Mode)：此模式用意在執行完重播模式(Replay Mode) 之後，將剩餘的 Event 能夠以自由的方式執行,即不會受到控制。

針對 Prefix Based Replay 的部份,將在第四章做更進一步的討論。

第二，利用在步驟一之後所產生的 SR-Sequence，代入競爭分析器(Race Analyzer)作競爭情況(Race Condition)的分析，競爭分析器的作法是藉由改變代入 SR-Sequence 的競爭情況的出像(Outcome)來產生一組 Race Variant 集合。

第三，若每次儲存裝置內所儲存的 Race Variant 集合都還有 Race Variant，則繼續執行步驟一、二,重覆執行直到 Race Variant 集合的內容為空。

而在中間的過程中其他步驟則留到第四章 Protocol 設計的部分再作介紹。

[2] SR-Sequence：一些發送以及接收運算事件(Synchronization Event)所組成的序列 ,就叫做 SR-Sequence，用來表示該回合同步化事件所執行的情況

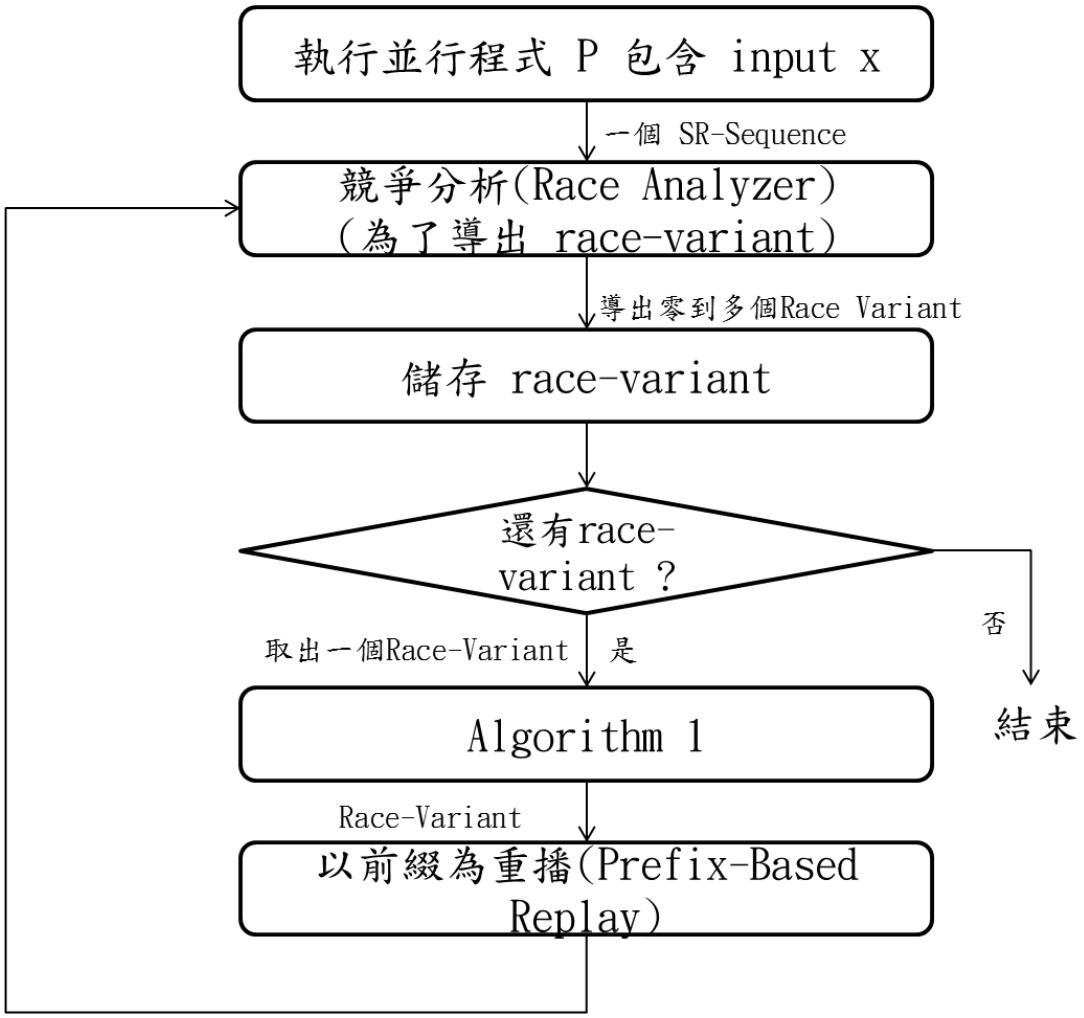


圖 3.1 Framework of Asynchronous Message Passing Testing

4. Collect Analysis Information form Execution Message-Passing Program

在這章節首先會介紹何謂以前綴為基礎重播(Prefix-Based Replay)技術，以前綴為基礎重播控制是利用在同步化事件前後插入進入(Entry)以及離開(Exit)的協議(Protocol)，主要分為兩個階段重播(Replay)模式以及監視(monitor)模式,以前綴為基礎重播的目的是為了產生該回合執行完畢所紀錄的發送以及接收事件所組成的同步化序列，然後將這些同步化序列收集起來，做為之後給予競爭分析器(Race Analyzer)做分析時重要的依據，在 4.2 章節則會提到針對非同步訊息傳遞測試發送以及接收的設計，包含使用[9]在 Share Memory 上針對測試 Read、Write 事件所設計的協議,但是若單純使用這幾個協議來測試非同步訊息傳遞程式可能會出現問題，這些問題可能導致測試無法運行，然後提到在以前綴為基礎重播測試事件時所記錄下來的資訊，有什麼目的?資訊有助於幫忙判定錯誤情況發生?以及如何針對上述所提到的問題重新設計在非同步訊息傳遞程式上的發送和接收協議?

4.1 Prefix-Based Replay

在可達性測試方法其中一個階段必須將受測程式的同步化事件執行時資訊記錄下來以供分析，此階段就稱作以前綴為基礎重播(Prefix-Based Replay)技術，在這個章節中將介紹以前綴為基礎重播的執行方式以及如何藉由協議(Protocol)達到測試目的，以前綴為基礎重播其用意是產生與前一次不同的同步化序列(SYN-Sequence),它將之前記錄所得到的同步化事件(Synchronization Event)資訊，透過競爭情況分析器(Race Analyzer)所產生的競爭變異(Race Variant)，將競爭變異分為數個部分的同步化事件(Part Of RaceVariant)並分送到各個節點，利用以前綴為基礎重播技術加以控制事件執行順序，以及記錄相關資訊可產生與前一次不同的同步化序列，此技術必須在同步化事件中插入進入(Entry)以及離開(Exit)的協議(Protocol)在其前後部分如圖 4.1，如圖 4.1 所示藉由在原始的受測程式中插入協議在欲觀察同步化事件的前後，這個協議主要會在程式執行時做以下兩階段事情。

1. 重播模式(Replay Mode)：負責將收到的 Race Variant，根據其同步化事件內部的版本號資訊，控制其執行順序，並記錄此次同步化事件的執行時的相關資訊。
2. 監視模式(Monitor Mode)：執行緊接在 Replay Mode 之後，讓整體的測試剩餘的同步化事件，以自由隨機的方式執行，在執行的過程的資訊並加以記錄起來，在此模式下可幫助產生多種不同交錯(Interleaving)的可能性。

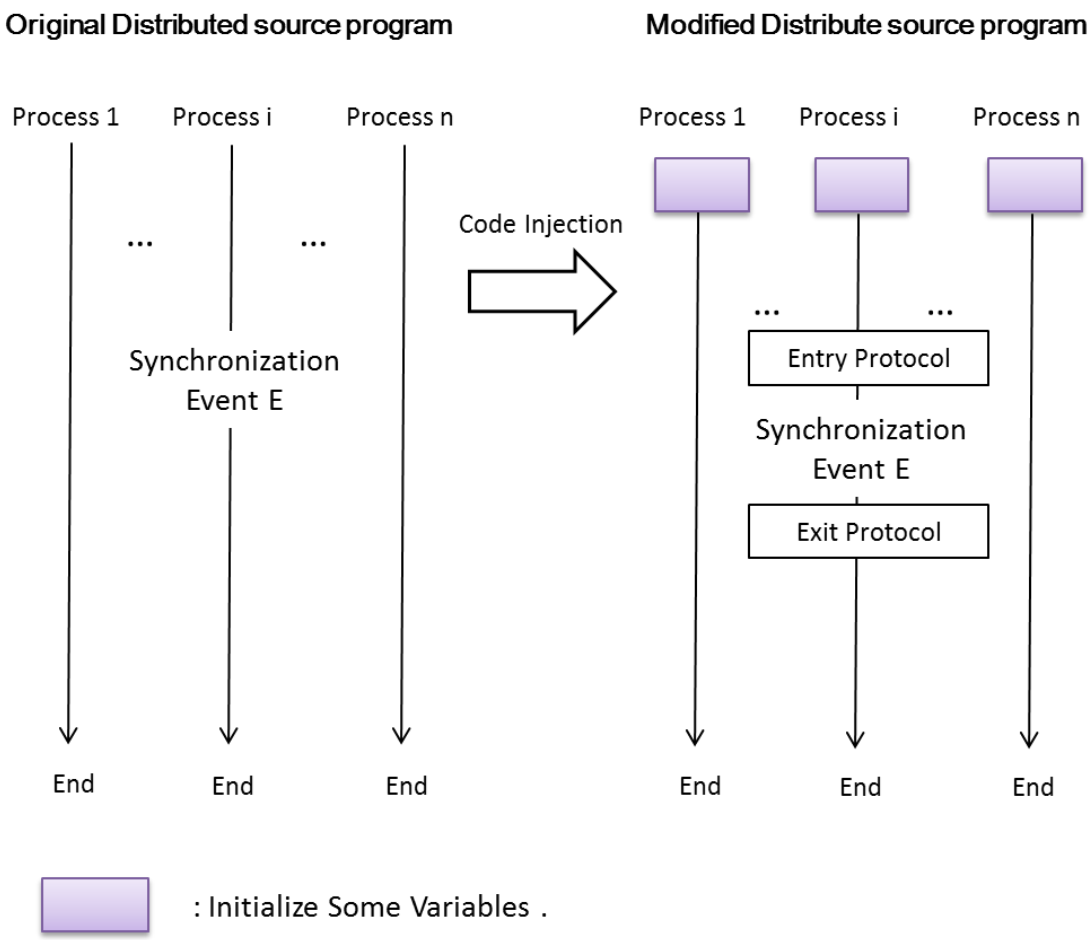


圖 4.1 Insert a Pair Protocol to Original Distributed source program

4.2. Synchronization Event Entry and Exit Protocol

4.2.1 Share Memory Protocol

在[9]提到的 Reachability Testing 使用在 Share Memory 環境下針對 Read 以及 Write 事件測試而設計的 Protocol，對於 Read、Write 進入和離開 (Entry/Exit) 協議(Protocol)設計會在此節做說明，在[9]中其主要目的是針對 Share Memory 共

用變數的存取，記錄當時執行的一些狀態資訊，而 Protocol 設計如圖 4.2 到 4.5 所示，P()以及 V() 分別表示對 semaphore 的存取(-1)以及釋放(+1),而這兩個信號機給定的初始值為 1,所以又稱做二元信號機(binary semaphore),協議(protocol)裡使用兩個信號機(一為 Semaphore_mutex，另一為 Semaphore_E),兩個信號機作用是用來保護在信號機包含的範圍(P()到 V()之間)內的運算是不可分割(atomic)藉由信號機的幫忙使得共用變數受到保護,可讓在這範圍所記錄到的資訊得以準確，mode 則表示目前執行到哪種模式(monitor 或 replay)，Current Version_U 表示目前對共用變數 U 存取到第幾個版本號，event.getVersion()表示目前在 replay mode 下欲執行 event 的版本號。

```
1   Read_Entry_Protocol(EventNode event)
2   BEGIN
3           IF (mode.equal("Replay")) THEN
4               IF (NumberOfReplayEvent == 0) THEN
5                   P(Semaphore_mutex);
6                   NumberOfProcessesFinishReplay++;
7                   V(Semaphore_mutex);
8                   mode.changeTo("Monitor");
9                   P(Semaphore_E);
10              END IF
11              ELSE
12                  while(Current_Version_U != event.getVersion());
13              END ELSE
14          END IF
15          ELSE            //Monitor Mode
16                  P(Semaphore_E);
17          END ELSE
18  END
```

圖 4.2 Read Entry Protocol

```
1   Read_Exit_Protocol()
2   BEGIN
3           CurrentVersion_U++;
4           IF (mode.equal("Monitor")) THEN
5                   Store W(U,Ver,Locctr,Value) into SYN-Sequence.
6                   /*
7                       U is a ShareVariable Name.
8                       Ver is the VersionNumber of Access ShareVariable.
9                       Locctr is a read or write event location.
10                      Value is a SharedVariable Value.
11                  */
12                      V(Semaphore_E);
13          END IF
14          ELSE
15                  NumberOfReplayEvents--;
16                  NumberOfReadReplayEvent(CurrentVersion,U)--;
17          END ELSE
18  END
```

圖 4.3 Read Exit Protocol

```
1   Write_Entry_Protocol(EventNode event)
2   BEGIN
3           IF (mode.equal("Replay")) THEN
4               IF (NumberOfReplayEvent == 0) THEN
5                   P(Semaphore_mutex);
6                   NumberOfProcessesFinishReplay++;
7                   V(Semaphore_mutex);
8                   mode.changeTo("Monitor");
9                   P(Semaphore_E);
10              END IF
11              ELSE
12                  while(CurrentVersion_U != (event.getVersion()-1));
13                  while(NumberOfReadEvent((event.getVersion()-1),U) !=0 );
14              END ELSE
15          END IF
16          ELSE                    //Monitor Mode
17                  P(Semaphore_E);
18          END ELSE
```

```
19 END
```

圖 4.4 WriteEntryProtocol

```
1   Write_Exit_Protocol()
2   BEGIN
3           CurrentVersion_U++;
4           IF (mode.equal("Monitor")) THEN
5              Store W(U,Ver,Locctr,Value) into SYN-Sequence.
6              /*
7                 U is a ShareVariable Name.
8                 Ver is the VersionNumber of Access ShareVariable.
9                 Locctr is a read or write event location.
10                Value is a SharedVariable Value.
11             */
12                V(Semaphore_E);
13          END IF
14          ELSE
15                NumberOfReplayEvents--;
16          END ELSE
17  END
```

圖 4.5 WriteExitProtocol

4.2.2 Two Version Number

此節將介紹在發送(Send)以及接收(Receive)運算進入(Entry)以及結束(Exit)協議(Protocol)中針對一個 Channel 使用兩種版本號的概念分別表達發送以及接收在 Channel 的順序，版本號(Version Number)的目的是為了用來幫助記錄事件執行的先後順序，版本號通常會放在進入以及離開協議的設計中，使用在兩方面，第一、記錄此事件在全部同步化事件中順序為何？第二、競爭變異(Race Variant)中的同步化事件附加了版本號資訊，當在執行以前綴為基礎重播時在重播

(Replay) 模式必須依照事件所附加的版本號資訊來控制事件執行的順序，在非同步訊息傳遞程式發送以及接收運算協議設計中將採用兩個版本號分別表示發送以及接送在 Channel 的順序如圖 4.6，藉由相同發送以及接收的版本號可了解(Send,Receive)序對的執行順序，以及可幫助找出潛在競爭情況而造成的錯誤，兩個版本號的使用上會在 4.2.5 節上說明，而在單純使用一個版本號來記錄事件的執行順序由於造成一些潛在的問題，這些問題會在下一節中做詳細的解說

圖 4.6 Two Version Number

4.2.3 Problems caused due to Share memory Protocol

考慮訊息傳遞程式原始的基本運算為發送(Send)以及接收(Receive)，傳送到一個中間的場所，而這個 Channel 的內容會經由發送以及接收產生變化，所以在這我們稍微修改在 Reachability Testing 裡的寫入進入協議(Write Entry Protocol)如圖 4.7 及 4.8 來代表發送(Send)以及接收(Receive)協議，插入我們所設計的例子如圖 4.9，但經過測試後發現存在了一些潛在的情況與問題，在此節將針對這些問題做描述。

1.如圖 4.7、4.8、4.9 所示，接收(Receive)運算是屬於阻擋式(Blocking)的接收，當在監視模式(monitor mode)下執行，假定為 Client1 先執行到 WriteEntryProtocol，第一次執行採取自由隨機方式讓程式進入監視(monitor)模式執行，進入之後會先

搶奪到 P(Semaphore_E)，然後減少 Semaphore 值，但此時訊息的 Channel 由於沒有訊息在裡面，導致接收運算會一直在等待，直到 Channel 有訊息存在，但此時 Client0 執行要執行發送(Send)的運算之前進入 WriteEntryProtocol 後，因為 Semaphore 的值已經被取得，需等待 Client1 執行 release 的運算來釋放 Semaphore 值，但此時 Client0 已經被卡住了，導致其他 Client 之間必須等待，形成一個 cycle，存在著死結(Deadlock)發生。

2.上述提到的論文[5]，其測試並不是在分散式的環境下，所以有很多事件的紀錄用到的共用變數的更動如：Version Number，利用多個 Thread 來做測試時僅靠一個共用物件(Object)來更新或讀取變數值，但實際在分散式環境下，行程跟行程之間是沒有關聯的，必須透過一個共用的地方來管理紀錄執行時資訊的共用變數。

3.在[9]中，寫入(Write)協議是採用一個版本號(Version)來記錄事件的執行順序，但使用一個版本號在之後所分析產生的競爭變異(Race Variant)會出現死結(DeadLock)的情況，考慮如圖 4.9，將圖 4.11 帶入圖 4.9 的簡易版本的競爭變異，圖 4.11 是藉由圖 4.10 經過 RaceAnalysis 之後所得來的(對於如何推導競爭變異將在第五章節說清楚)，在圖 4.9 的部分，Client1 會得到 S[1]的事件，將事件帶入各自的 writeEntryprotocol 後，競爭變異的事件控制部分是交由 replay mode 來執行，在執行到比較版本號的部分，由於版本號相同可以進去執行，Client1 的 receive event 執行時，並沒有訊息再 Channel 中，會一直等待在那,而其他 Client 則因為

Client1 的情況而無法繼續進行。

基於存在著以上數種情況，所以以訊息傳遞為基礎(Message Passing-Based)的程式必須針對其分散式環境下做設計以及重新定義進入(Entry)和離開(Exit)協議以達到測試的目的。

```
1  Write_Entry_Protocol(EventNode event)
2  BEGIN
3   IF (mode.equal("Replay")) THEN   //replay phase
4     IF (NumberOfReplayEvent == 0) THEN
5        P(Semaphore_mutex);
6        NumberOfProcessesFinishReplay++;
7        V(Semaphore_mutex);
8        while(NumberOfProcessFinishReplay != NumberOfProcess);
9        mode.changeTo("Monitor");
10       P(Semaphore_E);
11    END IF
12    ELSE
13       while(CurrentVersion_U != (event.getVersion()-1));
14       while(NumberOfReadEvent((event.getVersion()-1),U) !=0 );
15    END ELSE
16 END IF
17 ELSE              //Monitor phase
18       P(Semaphore_E);
19 END ELSE
20 END
```

圖 4.7 Modify WriteEntryProtocol

```
1  Write_Exit_Protocol()
2  BEGIN
3    CurrentVersion_U++;
4    IF (mode.equal("Monitor")) THEN   //Monitor phase
5      Store W(U,Ver,Locctr,Value+1) into SYN-Sequence.
6      /*
7          U is a ShareVariable Name.
8          Ver is the VersionNumber of Access ShareVariable.
```

```
9            Locctr is a read or write event location.
10           Value is a SharedVariable Value.
11       */
12       V(Semaphore_E);
13   END IF
14   ELSE          //replay phase
15       NumberOfReplayEvents--;
16   END ELSE
17 END
```

圖 4.8 Modify WriteExit Protocol

Client 0	Client 1
. . . Event event = popPartOfRVEvents(); writeEntryProtocol(event); send('a',Channel); writeExitProtocol(); . . .	String x; . . . Event event = popPartOfRVEvents(); writeEntryProtocol(event); x = receive(Channel); writeExitProtocol(); . . .
Client 2	Client 3
. . . Event event = popPartOfRVEvents(); writeEntryProtocol(event); send('b',Chanel); writeExitProtocol(); . . .	String y; . . . Event event = popPartOfRVEvents(); writeEntryProtocol(event); y = receive(Channel); writeExitProtocol(); . . .

圖 4.9 Insert Write Entry and Exit Protocol to Message Passing Example

S[0]{ W(Channel,1,501,Send,a); }	S[1]{ W(Channel,2,56,Receive,a); }
S[2]{	S[3]{

W(Channel,3,47,Send,b); }	W(Channel,4,152,Receive,b); }

圖 4.10 Simple SYN-Sequence Example

```
S[1]{
        W(null,1,null,Receive,null);
}
```

圖 4.11 Simple Race Variant Example

4.2.4 Record Synchronization Event Information

本節將介紹非同步訊息傳遞程式(Asynchronous Message Passing Program)執行時所需記錄的相關資訊，以及為何記錄這些相關的資訊，資訊可以帶給我們判斷何種情況。在並行程式(Concurrent Program)上測試時，由於存在著不確定性行為，使得並行程式難以測試，必需測試其程式碼的動態行為，此方式稱作動態測試(Dynamic Testing)，關於動態測試，必須要在程式執行時針對其執行時的資訊做紀錄，這種方式使用在很多存在不確定性行為程式的測試上如[7],[9],[10]，這些紀錄可以幫助我們判定一些情況，(如:死結(Deadlock)、程式可能執行的順序、程式可能產生的結果 等...)，在[7]中提到他採用一個通用的模組(A General Execution Model)，可以使用在 Semaphore、monitor、Asynchronous(Synchronous) Message Passing 執行時，利用一個通用的模組來做紀錄，在[9]則是只紀錄存取的讀取或寫入事件名稱、變數名稱以及這個事件執行時所存取的順序(Version Number)，因為僅需這些資訊就已足夠判定執行時事件的順序性，[10]是紀錄跟存取 Database 有關的資訊，像是存取哪個 Database，哪個 table 以及是使用那些運算用在 table 裡的哪個鍵值(Primary Key)，目的是為了找出競爭 DataBase 的可

能情況。

對於非同步訊息傳遞程式其基本原始運算為阻擋式接收(Blocking receive)以及非阻擋式發送(Non-Blocking send)兩種，針對發送以及接收的運算紀錄的資訊設計如圖 4.12

```
Receive : R(VarName,Ver,Loc,Value,RecVar,ChannelName)
Send : S(VarName,Ver,Loc,Value,ChannelName)
```

圖 4.12 非同步訊息傳遞紀錄資訊

如圖 4.12 所示，非同步訊息傳遞程式在插入協定後所需記錄的相關資訊。

- 變數名稱(Variable Name)：共用變數名稱(Variable Name)，若存在對共用變數有競爭情況的發生，則競爭圖上不同端點的同步化事件就存在一有向邊。
- 版本號(Ver)：版本號(Version Number)分為兩種:若是發送運算(Send)則表示全部的發送到相同 Channel 的發送運算之中他的執行順序為何?，而接收運算(receive)則表示全部的從相同 Channel 做接收的接收運算之中他的執行順序為何?
- 位置數(Loc)：位置數(Location count)是表示發送或是接收運算事件的所在位置。
- 運算(SROp)：運算(SROp)是紀錄著這個事件是屬於發送或是接收運算。
- 內容(Value)：內容(Value)則表示目前的發送或接收的值為何?
- Channel 名稱(ChannelName) : Channel 的名稱。
- RecVar：表示為儲存接收運算所收到訊息回傳值的變數名稱。

由於以訊息傳遞為基礎程式存在著訊息競爭(message race)，當多個節點(n，n>=2)同時對一個 Channel 做發送以及接收運算，可能因為訊息延遲或是一些情況，而造成發送(接收)到 Channel 的訊息順序會不同，進而導致產生不同的結果，針對這個情況所以紀錄了版本號、值、運算、變數名稱和位置。

4.2.5 Asynchronous Message Passing Protocol

本研究著重在於針對單一通道多個接收者非同步訊息傳遞架構程式做動態測試，對於非同步訊息傳遞，如上所示，為非阻擋式發送以及阻擋式接收，而在前面小節提到，以前綴為基礎重播(Prefix-Based Replay)技術需要插入進入(Entry)以及結束(Exit)協議(Protocol)在同步化事件前後，來達到測試的目的，而不同的情況(如 :Share Memory，Semaphore，Monitor，Message Passing 同步化事件 [7])其設定進入以及結束的協議也不盡相同，若貿然使用協議將會產生不可預期後果，此小節剩餘的部分將針對非同步訊息傳遞程式的基本訊息傳遞運算進入以及結束的協議設計做介紹，經由重新設定圖 4.4 及 4.5 的協定，將會克服前面小節所提到的情況。

基於在 4.2.3 所討論到的問題，在此必須重新設計協議，架構上仍採取 Write Protocol 設計，像是 Entry、Exit Protocol 仍採用兩階段(Replay & Monitor)執行，在執行時仍須紀錄一些相關資訊等...，針對一些特定情況的克服額外增加以及修改的主要分成三個部分:

1.一個 Channel 採用兩個版本號 ： 對於每個 Channel 採用兩個版本號分別記錄對

該 Channel 發送以及接送執行時的順序，原始的設計是採用一個版本號的設計，但由於會發生死結的情況(情況再 4.2.3 提到)，所以在這改由兩個版本號的設計。

2.檢查機制一：在 4.2.3 節所提到因為可能因為 receive event 的特性(blocking receive)，當 Channel 沒有訊息存在時，此時若讓帶有 receive event Client 先執行就會發生卡在 Semaphore 裡，進而形成死結的狀態，在此我新增額外的設計，另外增加 Channel 的訊息數狀態紀錄，當執行發送運算時，表示會發送訊息到 Channel，所以必須要增加 Channel 的訊息數，當執行接收運算時，表示會從 Channel 做接收的動作，所以必須要減少 Channel 的訊息數。

3.檢查機制二：當執行某些特定的 Race Variant 時，可能會有某些 Receive Event 因為不存在相對應的 Send Event 來匹配，使得當執行到該 receive 運算時，會有卡住的情況，進而導致整體無法運作，為了解決此問題必須額外紀錄導致卡住的 receive Event 的版本號使用演算法如圖 4.14，這個版本號將會在 Receive Entry Protocol 使用到用來讓整體測試運行不會卡住。

如圖 4.13 所示，將某次同步化序列所產生的其中之一個 Race Variant 帶入 Prefix-Based Replay 階段執行，則會產生以下情況，首先 Client 0 的 send event 以及 Client3 的 receive event 匹配，因為根據版本號的控制 Client 0 的 send event 發送的訊息會由 Client 3 的 receive event 做接收，但是當執行到第二個 receive event(Client 1)時候由於 Race Variant 裡已沒有 send event 會發送訊息到 Channel(CH1)，所以導致 receive event 會卡住，為了找出 Race Variant 裡的卡住

receive event，利用 4.14 的演算法將可以得到這個事件的版本號資訊，而這個演算法就是在 3.1 章節所提到的為了特定情況而新增的步驟紀錄會卡住的 receive event version number。

S[0]{	S[1]{	S[3]{
$W_S(In_1,1,c[0,1],"a",CH1)$	$W_R(Out_1,2,c[1,1],"a",CH1,x)$ $W_S(in_2,1,c[1,2],"c",CH2)$	$W_R(Out_1,1,c[3,1],"b",CH1,y)$
}	}	}

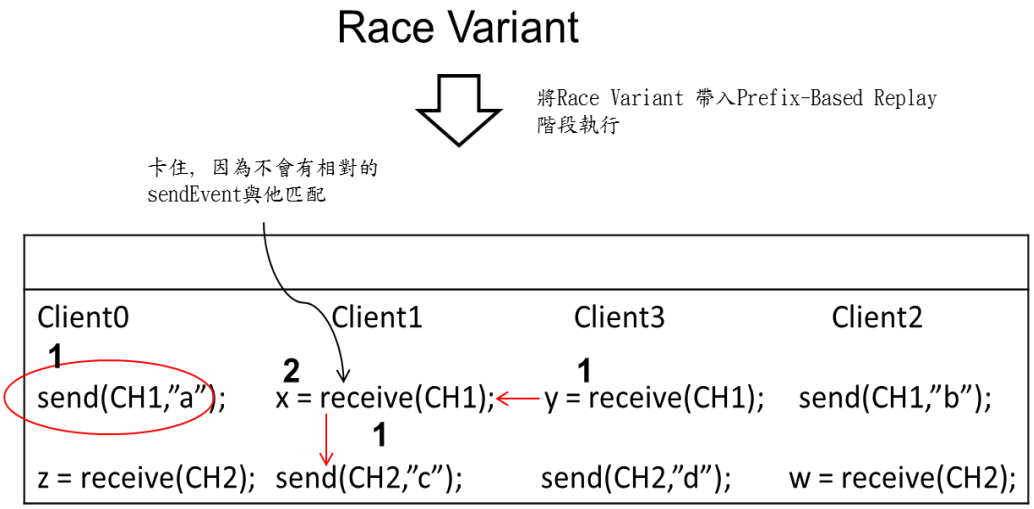


圖 4.13 影響整體測試的 Race Variant 例子

Algorithm 1

輸入 : RaceVariant **RV**

輸出:會影響整體執行的 Receive Event VersionNumber

=> Integer **BlockREVer[NumberOfClient]**

1. RV′ = RV
2. **FOR** (each RV′ Event **E1**)
   FOR(each (RV′ Event-**E1**) **E2**)
   IF((**E1**.variableName == "In_i") && (**E2**.variableName == "Out_i")) **THEN**
   IF(**E1**.Version == **E2**.Version) **THEN**
   a. 從 RV′ 上刪除 **E1**、**E2** 。
   END IF
   END IF
   ELSE IF((**E1**.variableName == "Out_i") &&
   (**E2**.variableName == "In_i")) **THEN**
   IF(**E1**.Version == **E2**.Version) **THEN**

```
                    a.從 RV′ 上刪除 E1、E2。
          END IF
      END ELSE
    END FOR
 END FOR
  FOR(each RV′ Event e)
       IF(e.variableName ==  "Out_i" ) THEN
            BlockREVer[e.clinetID] = e.Version;
       END IF
   END FOR
return BlockREVer;                                  i = Destination id
```

圖 4.14 找出會影響整體執行的 Receive Event Version Number

如圖 4.14 所示，此演算法的目的為找出會影響整體執行的 Receive Event 版本號，主要步驟為逐一比對每個 Race Variant 裡的同步化事件，若存在兩個事件分別變數名稱(VariableName)為 In_i、Out_i 或是 Out_i、In_i 的組合且版本號相同(隱含表示這兩個 Event 為 (Send/Receive Event) 發送(接收)訊息到(從)特定的 Channel(Channel i)，也就是表示這次發送訊息可以接收的到)，接著過濾掉，將過濾後的 Race Variant 判斷是否還有存在 Event 若是對 Out_i 變數做存取就把對應版本號存在對應的整數陣列(BlockREVer)，此陣列將會在 Receive Entry Protocol 派上用場以圖 4.13 的例子來說就是會過濾掉 Client0 的 Send event 以及 Client 3 的 receive Event，剩下 Client 1 為卡住的 receive event 並且記錄他對應的版本號。

如圖 4.15 到圖 4.18 為 Send/Receive Entry、Exit Protocol，在這四個協議的設計架構上仍然分成兩個階段執行(Replay、Monitor phase)，P(Semaphore_E)、V(Semaphore_E)以及 P(Semaphore_ShrVar)以及 V(Semaphore_ShrVar)這兩組為 binary Semaphore 其目的為保護該範圍執行時是不可分割的(atomic)，因為有些共

用變數若同時有多個行程或是程式存取可能會造成未預期的結果，而在 Send/Receive Entry Protocol 裡第六行的迴圈，則是判斷當目前的所有的 Client 完成所分配到的 Race Variant Event(又稱作 Replay event)，則會跳離迴圈，之後轉換到 monitor phase 來執行(Send/Receive Entry 的第八行)，否則卡在迴圈中等待其他 Client 執行完畢。

在 Send/Receive Exit Protocol 裡的第三行為前面提到的額外增加以及修改的第一點，對於每個不同 Channel 我們採用兩個版本號分別來記錄發送以及接收執行時的順序，目的是為了克服一個版本號所發生的死結情況。

在 Send/Receive Entry Protocol 第七行和 Send/Receive Exit Protocol 裡的第六行以及 Receive Entry Protocol 裡的 14~16 行，則是為了上面提到的第二個檢查機制，當分配到的 Race Variant 執行時會在若是 Receive Event 則會在第 14~16 行做個檢查判斷是否符合目前的版本號，若是，則做 NumberOfClientFinish 遞增的動作，目的是為了提早讓其他的 Client 能提早執行跳離 Entry Protocol 裡第六行的迴圈能早點執行到與卡住 receive event 匹配的 send event，而為了避免有其他的 Client 裡有包含分配到未完成 Race Variant Event 尚未執行完就執行之後屬於 monitor phase 才會執行到的 Event，增加迴圈阻擋在 Send/Receive Entry 的第七行，當某個屬於存取 In_i(Out_i)的 Race Varinat Event(又稱作 Replay Event)沒執行完則不能接下去執行 monitor phase 的 Event，避免執行時順序的錯亂。

最後在 Send/Receive Exit Protocol 裡的第五行以及 Receive Entry 裡的第九

行以及 22 行所做處理為上面所提到的第一個檢查機制，當執行發送運算時 Channel 內必定會有訊息，所以必須做訊息數的遞增，當作接收運算時 Channel 的訊息必定會減少，所以必須做訊息數的遞減，這些處理目的是為了當在執行 receive event 時 receive entry 會做一判斷，判斷目前 Channel 是否存在訊息，若無則不會讓 Client 搶到 Semaphore。

```
1    ASynSend_Entry_Protocol(EventNode event)
2    BEGIN
3        IF (mode == "Replay") THEN  //Replay mode
4          IF (NumberOfReplayEvent == 0) THEN
5              NumberOfClientFinish ++;
6              while (NumberOfClientFinish != NumberOfClient);
7              while( In_iReplayEvent != 0 );
8              mode = "Monitor";
9              P(Semaphore_E);
10         END IF
11         ELSE
12             WHILE(CurrentSendVerson_i != event.getVersion);
13             P(Semaphore_ShrVar);
14         END ELSE
15     END IF
16     ELSE  // Monitor mode
17           while( In_iReplayEvent != 0 );
18           P(Semaphore_E);
19     ELSE END                                    i = Channel id
20   END
```

圖 4.15 Asynchronous Send Entry Protocol

```
1   AsynSend_Exit_Protocol()
2   BEGIN
3     CurrentSendVersion_i ++;
4     IF(mode == "Replay") THEN
5         Channel_iMsgCount++;
6         In_iRelayEvent--;
7         Store Synchronization Event Information  S(U,Ver,L,Val,Channel)
8         /*
9           U : 共用變數名稱
10          Ver : 版本號
11          L : 紀錄同步化事件所在位置
12          Val : 執行發送以及接收運算的值
13          Channel : Channel名稱
14        */
15        NumberOfReplayEvent--;
16        V(Semaphore_ShrVar);
17    END IF
18    ELSE //monitor mode
19         Channel_iMsgCount++;
20         Store Synchronization Event Information  S(U,Ver,L,Val,Channel)
21         /*
22           U : 共用變數名稱
23           Ver : 版本號
24           L : 紀錄同步化事件所在位置
25           Val : 執行發送以及接收運算的值
26           Channel : Channel名稱
27        */
28         V(Semaphore_E);
29    ELSE END                                          i = Channel id
30  END
```

圖 4.16 Asynchronous Send Exit Protocol

```
1   ASynReceive_Entry_Protocol(EventNode event)
2   BEGIN
3       IF (mode == "Replay") THEN  //Replay mode
4          IF (NumberOfReplayEvent == 0) THEN
5              NumberOfClientFinish ++;
6              while (NumberOfClientFinish != NumberOfClient);
7              while ( Out_iReplayEvent != 0 );
8              mode  =  "Monitor";
9              While (Channel_iMsgCount  == 0);
10             P(Semaphore_E);
11         END IF
12         ELSE
13             WHILE (CurrentReceiveVersion_i != (event.getVersion()-1)));
14             IF(CurrentReceiveVersion_i== BlockREVersion[event.clientID]) THEN
15                     NumberofClientFinish++;
16              END IF
17             P(Semaphore_ShrVar);
18         END ELSE
19      END IF
20      ELSE  // Monitor mode
21            while( Out_iReplayEvent != 0 );
22            While (Channel_iMsgCount == 0);
23            P(Semaphore_E);
24      ELSE END
25  END
                                                      i = Channel id
```

圖 4.17 Asynchronous Receive Entry Protocol

```
1   AsynReceive_Exit_Protocol()
2   BEGIN
3     CurrentReceiveVersion_i ++;
4     IF(mode == "Replay") THEN //Replay mode
5       Channel_iMsgCount--;
6       Out_iRelayEvent--;
7       Store Synchronization Event Information  S(U,Ver,L,Val,Channl,RecVar)
8       /*
9         U : 共用變數名稱
10        Ver : 版本號
11        L : 紀錄同步化事件所在位置
12        Val : 執行發送以及接收運算的值
13        Channel : Channel名稱
14        RecVar :接收訊息變數名稱
15      */
16      NumberOfReplayEvent--;
17      V(Semaphore_ShrVar);
18    END IF
19    ELSE //monitor mode
20      Channel_iMsgCount--;
21      Store Synchronization Event Inforamtion S(U,Ver,L,Val,Channel,RecVar)
22      /*
23        U : 共用變數名稱
24        Ver : 版本號
25        L : 紀錄同步化事件所在位置
26        Val : 執行發送以及接收運算的值
27        Channel : Channel 名稱
28        RecVar : 接收訊息變數名稱
29      */
30      V(Semaphore_E);
31    END ELSE
32  END                                                    i = Channel id
```

圖 4.18 Asynchronous Receive Exit Protocol

利用在以前綴為基礎控制執行時所記錄到的資訊,除了可以幫助分析執行時的一些情況,還可以將這回合收集到的紀錄作競爭情況的分析,進而導出更多的可能情況,在[10]中所提到的方法,利用收集到的 T-Sequence 先轉化為競爭圖(RaceGraph),競爭圖的產生是判斷是否不同的 Client 之間是否存在者競爭(race)的情況以及根據事件發生的先後順序畫出一個有向圖,轉化出競爭圖之後,再利用演算法導出零到多個 race variant,此方法稱做 Race Analysis,產生出來的 Race Variant 可以進一步的再傳入各個節點中,藉由 Prefix-Based Replay 的技術,在執行過後可導出不同於上一組的 T-Sequence,在第五章節將針對 Race Analysis 的相關技術作介紹。

4.2.6 Prefix-Based Replay Implementation

不像在集中式環境,在分散式環境中行程之間的溝通沒有共享記憶體必須透過訊息傳遞來溝通,所以在 Prefix-Based Replay 的 Protocol 的設計上必須額外增加一協調的伺服器(Coordinator Server)來提供共用變數的存取，在此小節將對 Prefix Based Replay 的設計做介紹，如圖 4.19，如圖 4.19 所示在這用來實現訊息傳遞測試的測試是採用 JMS(Java Message Service)，JMS 為 Java 2 EE 的 API 之中的其中一部份,主要是用來讓軟體元件之間的溝通,如及時股票系統等應用...，在這我採用 JMS API 作為我實現測試的工具，在設計上如同上面所提到，進入(Entry)/離開(Exit) Protocol 的設計,需要使用到很多共用變數如 4.2.5 節中 Protocol 的設計章節所介紹到的那些變數，但由於是測試在分散式環境中，所以在測試期間所需紀錄到的變數(如版本號(Version))必須仰賴一個地方來負責協調存取，在這裡我設計了協調者伺服器，當在測試期間若需要使用到共用變數，或是一些運算如信號機(Semaphore)，就透過發送訊號要求這個協調者來達到我要的目的，圖中的 Queue 是 JMS 之中各個節點用來與協調者溝通的訊息管道。

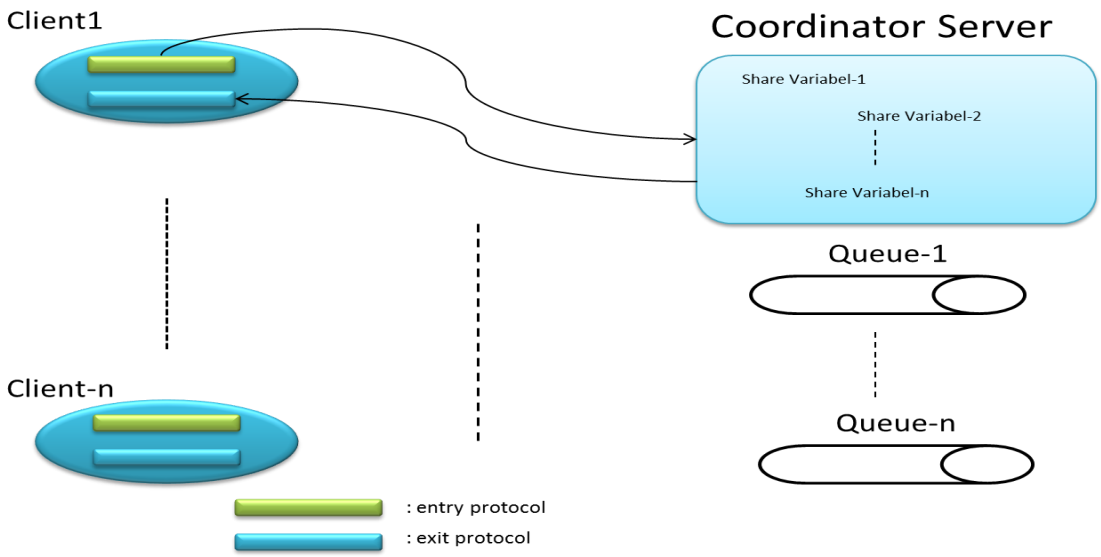


圖 4.19 Prefix Based Replay Design

5. Race Analysis from Collect Information

此章節著重在介紹競爭分析器所執行的過程,首先先介紹什麼是競爭圖(Race Graph)，如何將收集來 SR 序列轉化成競爭圖，轉化成競爭圖之後，再來利用改良版的競爭分析演算法來產生競爭變異,針對以上介紹會使用一簡單圖例方便了解整體運作。

並行程式(Concurrent Program)在執行時若沒有對共用變數或共用資源給定保護的機制的話，當發生競爭情況(race condition)的時候可能造成產生的結果與事先設定不符的情況，而競爭情況是指多個行程(Process)或是執行緒(Thread)同時對一個共用變數或是共用資源做存取，可能因為系統排程的次序不同，而產生不同的結果，在以非同步訊息傳遞為基礎程式中(Asynchronous Message Passing-Based Program)，若沒有對其競爭情況做設計考量則可能發生錯誤的訊息傳遞，以非同步訊息傳遞為基礎程式中的測試來說，測試架構上其中一個階段會針對訊息競爭情況做分析，經由競爭分析器(Race Analyzer)分析的結果，可分析出訊息對 Channel 的競爭情況，會產生零個到數個的競爭變異(Race Variant)，將競爭變異帶入第四章節所提到以前綴為基礎重播(Prefix-Based Replay)技術後，進而導出不同於此次的執行狀況。

5.1 Race Graph

本節將介紹利用一演算法導出可用來表達非同步訊息傳遞程式同步化事件競爭關係的競爭圖，競爭圖(Race Graph)常見於用來分析目前是否存在潛在競爭情況(Race Condition)[10]，它是一有向圖(direct graph)，圖中的每個點表示同步化事件(如:message-passing 的 send 和 receive，semaphore 的 p 以及 v 等…)，而邊用來表示事件發生的先後順序(happen before)，競爭圖最早使用在[10]之中提到用競爭圖拿來分析多個客戶端(Client)同時存取數個資料庫(Data Base)所存在潛在競爭情況，在此我們將他使用在對以非同步訊息傳遞為基礎程式潛在競爭情況分析上，競爭圖產生的演算法如圖 5.1，如圖 5.1 所示首先第一步是將目前 SR 序列的 SR-event 各自轉化為一個點，第二步是將所屬相同 client 的 SR-event，藉由邊來表示事件執行順序，第三步是判斷不同 client 之間是否存在潛在的競爭以及比較版本號來表示不同 client 之間事件的先後關係。以下將提供一簡單轉換例子說明。

演算法: 產生 SR-Sequence 對應的 Race Graph
輸入 : SR-Sequence
輸出 : 競爭圖(Race Graph) G
(1) 將每個 SR-Sequence 裡的 SR-Event 作為競爭圖 G 的一點。
(2) 對相同節點的 SR-Event 根據其程式執行事件的順序加上一有向邊。
(3) 若不同節點的 SR-Event 可能同時對相同共用變數執行運算，假設為 SRx、SRy 兩點，且 SRx 的版本號小於 SRy，則加上一有向邊(SRx,SRy) 在圖 G 上。
[註] : SR-Event 為 non-blocking send 或是 blocking receive 事件。

圖 5.1 Algorithm : Generate the race graph of a SRW-sequence

Client0	Client1
S[0]{ @ W(In_1,2,c[0,1],send,'a',CH1,monitor) W(Out_2,2,c[0,2],receive,'d','z',CH2,monitor) }	S[1]{ @ W(Out_1,2,c[1,1],receive,'c','x',CH1,monitor) W(In_2,2,c[1,2],send,'c',CH2,monitor) }
Client2	Client3
S[2]{ @ W(In_1,1,c[2,1],send,'b',CH1,monitor) W(Out_2,1,c[2,2],receive,'c','w',CH2,monitor) }	S[3]{ @ W(Out_1,1,c[3,1],receive,'b','y',CH1,monitor) W(In_2,1,c[3,2],send,'d',CH2,monitor) }

圖 5.2 同步化序列(SR-Sequence)例子

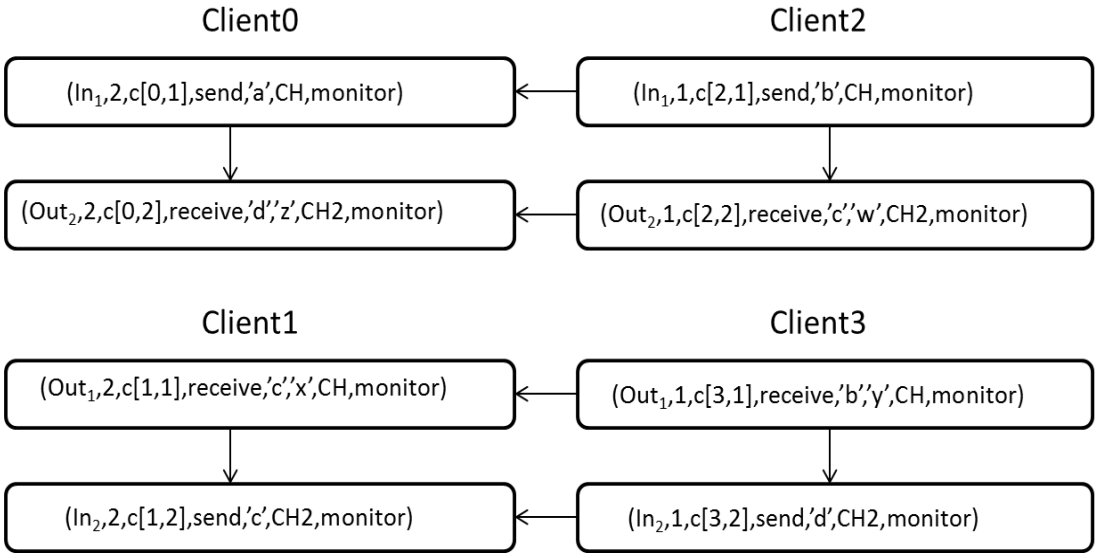


圖 5.3 race graph of a SR-Sequence

如圖 5.1、圖 5.2 所示，圖 5.3 是將圖 5.2 透過圖 5.1 演算法所產生的競爭圖，首先按照步驟一指示將各個事件轉化成節點，接著畫出相同 Client 之間的事件順序關係，如 Client0 的第一個 event 所在的行號比第二個 event 小，故存在一線段從第一個 event 到第二個 event，在來執行步驟三由於 client 0 和 client 2 可能對相同共用變數有存取的動作(在這看成為對 In_1 變數競爭，隱含表示兩個 send event 發送訊息到 Channel(CH))，可能發生潛在的訊息競爭(message race)，而 Client 0 執行 send event 的順序比 Client 2 還早(Version : 1<2)，故加上一有向邊從 Client 0 到 Client2 ,同理方式,Client 1 以及 Client 3 執行第三步驟(Version : 2>1)

故將邊連上從 Client3 的 node 到 Client1。

[註]: 圖 5.3 所得到的資訊額外增加 monitor 這部分沒有在 4.2.4 提到，主要原因是配合例子因為在競爭分析演算法階段會需要用到，如何分辨是屬於 monitor phase 執行的 event? 如圖 5.2 所示位於@符號之後的 event 就是 monitor phase 執行的 event。

競爭圖的產生可以幫忙在之後的競爭分析演算法產生多組競爭變異(Race Variant)，競爭分析演算法是藉由將事件之中存在潛在的競爭情況將兩事件所代表點之間的邊做轉向，在刪除一些不必要的事件所得到事件集合。此演算法將在下一節作介紹，並提供利用此演算法所推導的例子當解說。

5.2 Race Analysis Algorithm

在這裡，我將修改使用在[10]所提出的方法，將對以非同步訊息傳遞為基礎程式的同步化事件產生的競爭圖，利用修改過後的演算法導出競爭變異，這個競爭變異可使在下一回合的帶入以前綴重播為基礎技術(Prefix- Based Replay)產生不同的同步化序列。首先提出相關的推導競爭變異(Race Variant)的演算法技術為[9],但是存在著會產生過多多餘的競爭變異缺點,所以在[10]之中提出了一個新的方法利用所產生的競爭圖，帶入此演算法，即可產生出零到多個競爭變異，可大幅度減少產生多餘的競爭變異點。

如圖 5.5 所示，圖 5.5 演算法表示如何由 SR 序列所轉換而成競爭圖藉由分析邊來導出競爭變異在此我們使用圖 5.3 的競爭圖當作例子並由圖 5.5 競爭分析

演算法產生整體推導結果，結果顯示在圖 5.7，在介紹競爭演算法之前，首先先提到 in-edge，in-edge 是指若某點存在一有向邊指向它，而 out-edge 是指某點存在一邊連向其他點對該點來說那一個邊就叫做 out-edge。如圖 5.4 所示以點 E 為例，虛線部分則表示點 E 的 in-edge，而點線部分則表示點 E 的 out-edge。而圖 5.6 的部分提到將會針對圖 5.3 的某一競爭變異的推導的過程介紹，剩餘的推導步驟則同介紹方法。僅需照演算法步驟推導即可產生圖 5.7 完整結果。

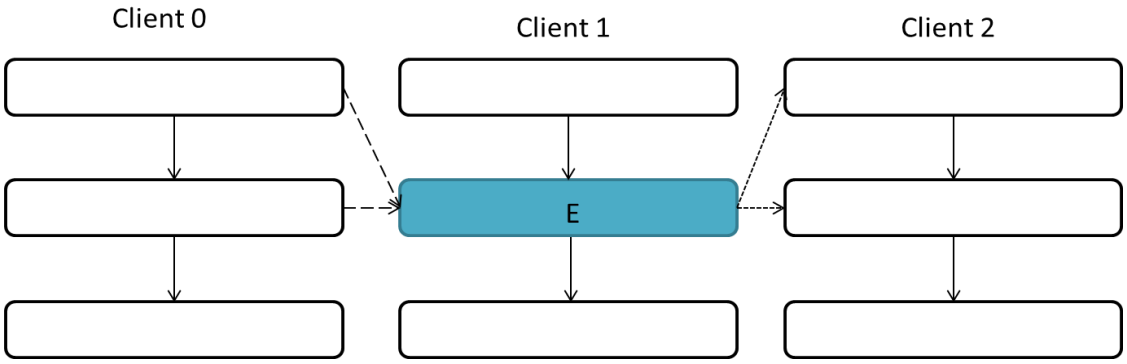


圖 5.4 in-edge and out-edge

在此將只介紹 Race Variant 1 的推導過程，剩餘的部分按照演算法步驟推導即可，如圖 5.6 所示，圖 5.6 為圖 5.3 簡化圖，Client 0 的 event 1 存在一個 in-edge(如虛線處)e1，由於只有一個邊所以所產生對應的二元表示式(binary representation)為[b1]，接著由於 b1 為 1，所以表示需要作反轉 e1 的動作(如步驟一)，在來則是將沒有發生在 event 1 之前的節點過濾掉(如步驟二)，而 Client 1 以及 Client 3 所相連的點及邊，因沒有與 Client 0 以及 Client 2 的事件相連，且又是在 monitor phase 執行所以必須作刪除的動作，最後得到的結果如圖 5.6 的 Race Variant 1 所示。競爭變異結果將以競爭圖來呈現。

Race Analysis Algorithm： 分析同步化序列所轉換的競爭圖來導出 race

variant。

輸入 : SR-Sequence 所導出的競爭圖(Race Graph) (G(SR))

輸出 : Race-Variant Set

(1) RV-Set = {ø}

(2) 令 SRE-Set 為 G(SR) 裡面的 SR-Event 所組成的集合，且集合內的 Event 至少包含一個 in-edge 和 在 monitor mode。

(3) 對每個在 SRE-Set 裡的每個 SR-Event **E**, k = **E** 的 in-edge 數. (假定為 (e1,e2,e3,……,ek))

For i=1 to 2^k -1

假定[b1,b2,……,bk]是 i 的二進位表示法。

令 RG' = RG(SR)

For j = 1 to k

IF b_j = 1 **THEN** 反轉對應的邊 ej

END For

IF RG' 不存在迴圈 **THEN**

1.移除發生在 E 之後且為 monitor mode 的點。

2.清除節點不必要的資訊。

3.RV-Set = RV-Set U (經由前面步驟所產生之 RG')

END IF

END For

圖 5.5 Race Analysis Algorithm

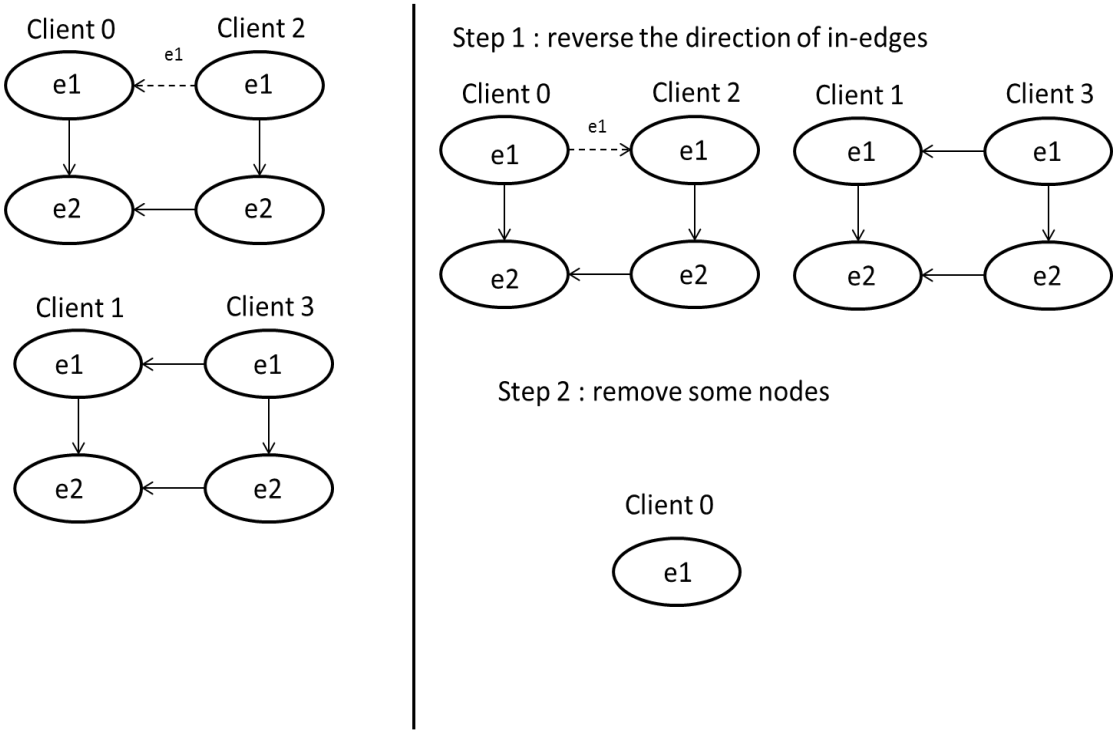


圖 5.6 An Process to illustrate Race Analysis Algorithm

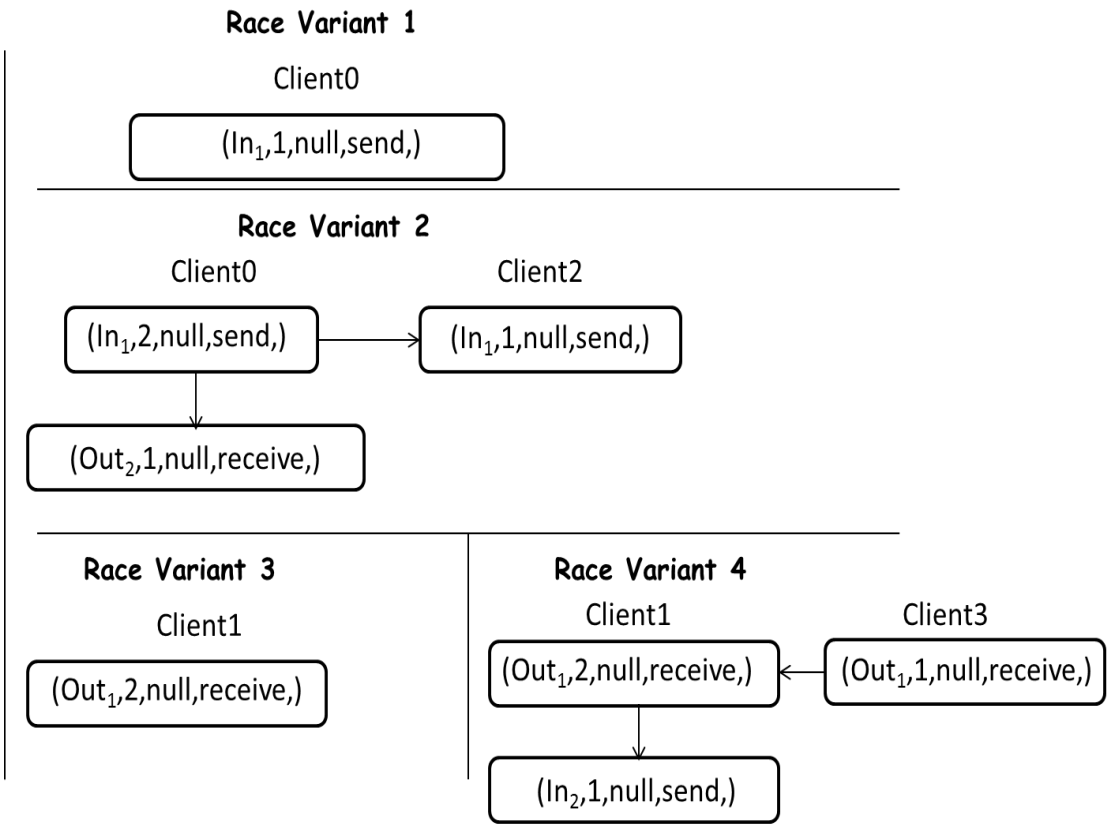


圖 5.7 Race Graphs of the race variants derived by Race Analysis Algorithm

6. Implementation and Experimental result

此章節將針對如何利用程式語言來實作在非同步訊息傳遞上的測試，主要分成兩個部分，首先會先介紹在實作上的架構這部分是使用到 Java 語言(J2SE)以及 JMS API(J2EE)的部分以及架構上設計的展現，再來，第二部分會提到將實作出來的測試架構使用在所給予的例子所得到的結果，在這部分我將會使用三種測試方法，一種是本篇論文使用到的可達性測試(Reachability Testing)架構，另外兩個分別為隨機延遲測試(Random Delay Testing)以及非決定性的測試(Non-Deterministic Testing)，比較這三種測試方法測試在我的例子上，根據實驗結果來評估三種方法的優缺點。

6.1 Implementation

在本篇論文所提到非同步訊息傳遞程式測試的實作上使用 Java 語言，Java 語言是屬於高階程式語言，優點為具有大量已建立好的函式庫提供使用者開發，缺點為包含大量的函式庫導至應用程式過於龐大，相較於 C 較不適合用在嵌入式應用的開發，Java 用途非常廣，包含企業方面的應用(屬於 J2EE)，以及手機應用程式的開發(Android)等… ，在實作上我採用 Java 2 Platform Standard Edition(J2SE 1.6.0_26)這是 Java 提供給使用者來開發的套件，以及 Java 2

Platform Enterprise Edition(J2EE)裡的 Java Message Service(JMS) API 主要是用來實現軟體元件之間的溝通，實際的應用方面如及時股票系統等應用...，如圖 6.1 所示, 圖 6.1 的部分展現了設計的架構,主要分成三個部分主程式(Main Program)、受測端(Client)以及協調者伺服器(Coordinator Server)，此架構是建置在分散式環境下，也就是各個訊息傳遞程式分處不同地方且彼此不了解對方的位置僅透過 Channel 當成一個中介場所來溝通，由主程式為執行的開端，對應程式的部分如圖 6.2，然後再經由透過控制器(Controller)負責控制整體測試的運作描述如下：

1.以前綴為基礎重播控制(Prefix Based Replay): 負責將每回合要代入執行的 Race Variant 分成數個部份(Part of RaceVariant)，送到各個需要做訊息傳遞程式測試的地方，而這些受測端需要在測試之前將程式中欲測試的同步化事件插入進入(Entry)以及離開(Exit)協議(Protocol)，最後在這個階段會將該回合記錄同步化事件所得到的資訊回傳到主程式上。

2. 競爭分析控制(Race Analyzer):負責將 SR-Sequence 代入 Race Analyzer 做分析並將產生出的零到數個可行的 Race Variant，儲存起來。

3. 負責判斷是否所有的 Race Variant 已經執行完畢。

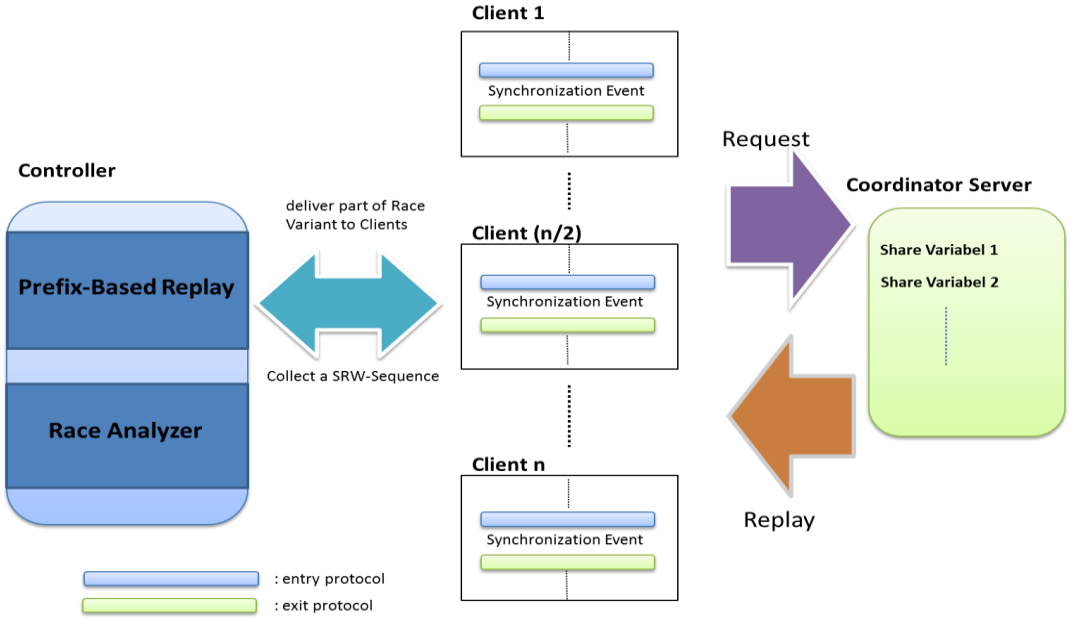


圖 6.1 Reachability Testing of Asynchronous Message Passing Program Architecture

```
controller.Initialize();
while(TotalRVs > 0 )
{
        NumberOfPerform ++;
        controller.ExecutePrefixBasedReplay();
        controller.ExecuteRaceAnalyzer();
        TotalRVs = ro.ReadFileValue("D:\\Dropbox\\For_Study\\NTNU
(Laptop)\\Java_Code\\ExperimentData\\TempInfo\\TotalRVs.txt");
        controller.SharedVarReSet();
}
```

圖 6.2 A Part of Main Program

而受測程式的部分如同前面以前綴為基礎重播控制的部分提到必須插入進入(Entry)以及離開(Exit)協議在受測程式上如圖 6.3 所示紅字的部分為插入進入(Entry)以及離開(Exit)協議在受測事件中以達到測試的目的。

```
while(Repeat!=false)
{
        blocker.Waiting();
       System.out.println("----------Start Testing["+NumberOfTest+"] ----------");
        protocol.ReSet();
        protocol.SetNumberOfReplayEvent(CommonVar.GetReplayEvents().size());
```

```
        if(CommonVar.GetReplayEvents().size() != 0)
        {
                Event = CommonVar.GetReplayEvents().popEvnt();
        }
        protocol.SendEntry(Event);
        Client0.SendMessage(ChannelName, "a");
      EventRecord = protocol.SendExit(NumberOfClientPerform, 1000,
      "a",ChannelName);
        System.out.println("NumberOfClientFinish    : " +transportData.S
endReplyRequest(Cid, CommonStr.GetNOCFIncreaseOp(),
CommonStr.GetNumberOfClientFinishQ()));
        transportData.SendObjectMessage(EventRecord,CommonStr.GetPB
REventNodeQ());
        Client0.SendClinetFinishSignal(transportData,CommonStr.GetPBREventNodeQ());
        NumberOfClientPerform = 1;
        NumberOfTest++;
}
```

圖 6.3 A Part of Test Program

最後一個部份是協調者伺服器(Coordinator Server)，他是處理共用變數的控制、運算、分配等…，當測試分散式環境下訊息傳遞程式時，多個分散在不同端點的程式，必須常常依賴共用變數，使用情況，如:使用到協定(Protocol)時，程式內部會使用到像是信號機(Semaphore)，必須要參考到一份共用的變數資料，如果變數設計為各端點都掌握一份區域變數的話，可能造成多個端點都可以同時執行 P()運算，這樣的設計會造成同時進入臨界區(Critical Section)，而失去了原本訊號機(Semaphore)是用來做互斥存取(Mutual Exclusion)的用途了，令一情況，像是在做非同步訊息傳遞測試中，會使用到共用變數，叫做版本號(Version Number)，版本號的用途是為了使當各端點同時在執行同步化事件(Synchronization Event)時，用來做記錄事件執行的先後關係，若版本號沒有使用

共用變數而是使用區域變數，則無法了解事件執行先後順序，這對測試的判斷影響極大。

6.2 Experimental Result

在此章節會呈現利用三種測試方法所測試三個例子，並且從實驗數據中比較其中的優缺點，如下三個例子如圖 6.5、6.6、6.7，其實驗結果如表 6.1

Client 0	Client 1
public static void main(…) { . . Send(“a”,CH1); . . }	Public static void main(…) { String x; . . x = receive(CH1); . }
Client 2	Client 3
Public static void main(…) { . . Send(“b”,CH1); . . }	Public static void main(…) { String y; . . y = receive(CH1); . }

圖 6.4 Example One

Client0	Client1
. send(‘a’,CH1); x = receive(CH2); .	. x = receive(CH1); send(‘c’,CH2); .
Client2	Client3
. send(‘a’,CH1);	. x = receive(CH1);

x = receive(CH2); .	send('c',CH2); .

圖 6.5 Example Two

Client0	Client1
. send("a",CH1);	. String x ; x = receive(CH1); If(x=="a") { send("c",CH2); }
Client2	Client3
String z; . . send("b",CH1); z = receive(CH2); . .	. . String y; y = receive(CH1); If(y==a){ send("d",CH2); }

圖 6.6 Example Three

Client 0	Client 1
public static void main(…) { . send("c",CH1); y = receive(CH2); . . }	public static void main(…) { . x = receive(CH1); send("d",CH2); w = receive(CH1); . . }
Client 2	Client 3
public static void main(…) { . . send("a",CH1); z = receive(CH2); .	public static void main(…) { . send("b",CH1); send("f",CH2); u = receive(CH1); .

. }	. }

圖 6.7 Example four

Information Program	Number of Send Event	Number of Receive Event	Number of Client	Number Of Channel	得到不同的可行性序列/總共執行次數	執行所花時間
Example One	2	2	4	1	4/5	26 s
Example T*wo	4	4	4	2	16/33	280 s
Example Three	4	2	4	2	4/5	34.65 s
Example Four	5	5	4	2	60/133	1302.8s

表 6-1 Experimental Data

接著將呈現以另外兩種測試方法，來測試上述例子，第一種方法是使用非決定性測試(Non-Deterministic Testing)，簡單來說，非決定性測試測試方式就是沒有加上任何額外的東西，讓程式自由的運行，期望產生程式所有可能運行的情況，第二個方法是使用隨機延遲測試(Random Delay Testing)，隨機延遲測試是在程式的敘述式之間插入延遲點(Delay Point)如圖 6.7，這個延遲點會延遲暫停一些時間，期望讓程式運行時能產生更多不同的情況，使用上述兩種測試架構分別測試不同執行次數，將測試 10 個回合所得結果，呈現在下列表中，而在隨機延遲測試方法中，因經過測試過後因無法得知哪個區間的時間可以得到最佳的效果,故任意選擇幾個區間做為測試的標準如表 6.5、6.6，

```
.
Delay Statement;
Statement 1;
Delay Statement;
```

Statement2;

Delay Statement;

Statement3;

Delay Statement;

.

.

圖 6.8 Insert Delay Statement to Test Program

執行次數 / 回合數	6	10	15
1th	4	4	4
2th	2	4	4
3th	3	3	4
4th	3	4	4
5th	2	3	3
6th	2	3	4
7th	3	4	4
8th	3	4	4
9th	3	3	3
10th	3	3	4
Average Feasible SR-Sequence	2.8	3.5	3.8
Average Time	24s	38s	60s

表 6-2 Non-Deterministic Testing of Example One

執行次數 / 回合數	33	40	100
1th	4	4	4
2th	4	4	6
3th	4	4	4
4th	4	4	4
5th	4	4	5
6th	4	4	4
7th	4	4	4
8th	4	4	6

9th	4	4	4
10th	4	4	4
Average Feasible SR-Sequence	4	4	4.6
Average Time	203s	248s	545s

表 6-3 Non-Deterministic Testing of Example Two

執行次數 / 回合數	5	10	15
1th	2	4	4
2th	2	3	4
3th	3	4	4
4th	4	4	4
5th	3	4	4
6th	3	4	4
7th	3	4	4
8th	3	4	4
9th	3	3	4
10th	3	4	4
Average Feasible SR-Sequence	2.9	3.8	4
Average Time	25.2 s	51.2 s	73.3 s

表 6-4 Non-Deterministic Testing of Example Three

執行次數 / 回合數	133	160	250
1th	7	6	6
2th	6	6	6
3th	6	6	6
4th	6	6	6
5th	6	6	6
6th	6	6	6
7th	7	6	7

8th	6	6	6
9th	6	6	6
10th	6	7	6
Average Feasible SR-Sequence	6.2	6.1	6.1
Average Time	842.39S	977.475s	1722.09s

表 6-5 Non-Deterministic Testing of Example Three

執行次數 / 回合數	5	10	15
1th	4	4	4
2th	3	3	4
3th	4	4	4
4th	4	3	4
5th	2	4	3
6th	2	2	4
7th	4	4	4
8th	4	3	4
9th	3	3	4
10th	3	4	3
Average Feasible SR-Sequence	3.3	3.4	3.8
Average Time	50.7s	103s	137.1s

表 6-6 Random_Delay_Testing of Example One(0~1s)

執行次數 / 回合數	6	10	15
1th	4	3	4
2th	4	4	4
3th	4	4	4
4th	3	4	4
5th	4	4	4
6th	4	3	4

7th	3	4	4
8th	3	4	4
9th	4	3	4
10th	3	3	4
Average Feasible SR-Sequence	3.66	3.66	4
Average Time	70s	130s	190s

表 6-7 Random_Delay_Testing of Example One(0~2s)

執行次數 / 回合數	5	10	15
1th	3	3	3
2th	3	2	4
3th	3	3	2
4th	2	4	4
5th	2	3	4
6th	3	4	4
7th	4	3	4
8th	3	4	4
9th	3	4	4
10th	3	3	4
Average Feasible SR-Sequence	2.9	3.3	3.7
Average Time	87.4s	114.1s	166.4s

表 6-8 Random_Delay_Testing of Example One(0~3s)

執行次數 / 回合數	33	40	100
1th	8	7	8
2th	6	6	10
3th	6	5	7
4th	5	6	10
5th	6	5	9
6th	7	6	8
7th	6	7	10

8th	7	7	12
9th	6	6	10
10th	6	6	11
Average Feasible SR-Sequence	6.3	6.1	9.5
Average Time	452.46s	541.17s	1392.27s

表 6-9 Random_Delay_Testing of Example Two(0~1s)

執行次數 / 回合數	33	40	100
1th	8	9	8
2th	8	6	9
3th	5	6	10
4th	6	7	9
5th	7	7	13
6th	6	8	8
7th	9	7	10
8th	8	8	12
9th	7	7	10
10th	8	7	11
Average Feasible SR-Sequence	7.2	7.2	10
Average Time	760.04s	873.07s	2213.81s

表 6-10 Random_Delay_Testing of Example Two(0~2s)

執行次數 / 回合數	33	40	100
1th	7	7	8
2th	8	8	10
3th	6	8	9
4th	7	7	11
5th	8	9	12
6th	5	7	11
7th	9	8	11
8th	7	7	11

9th	8	8	11
10th	7	8	9
Average Feasible SR-Sequence	7.2	7.7	10.3
Average Time	943.33s	1267.27s	3177.83s

表 6-11 Random_Delay_Testing of Example Two(0~3s)

執行次數 / 回合數	33	40	100
1th	8	10	12
2th	7	7	11
3th	7	10	12
4th	10	9	9
5th	6	8	12
6th	7	6	10
7th	6	6	11
8th	8	7	10
9th	5	8	10
10th	8	7	11
Average Feasible SR-Sequence	7.2	7.8	10.9
Average Time	1379s	1639s	4045s

表 6-12 Random_Delay_Testing of Example Two(0~4s)

執行次數 / 回合數	5	10	15
1th	3	4	4
2th	3	3	4
3th	3	4	4
4th	4	4	4
5th	3	4	4
6th	2	4	4
7th	3	4	4
8th	3	4	4
9th	3	3	4

10th	2	4	4
Average Feasible SR-Sequence	2.9	3.8	4
Average Time	61.9 s	127.3 s	184.4 s

表 6-13 Random_Delay_Testing of Example Three(0~1s)

執行次數 / 回合數	5	10	15
1th	3	4	4
2th	2	4	4
3th	3	4	4
4th	4	3	3
5th	3	4	4
6th	4	4	4
7th	2	4	4
8th	2	4	4
9th	3	4	4
10th	3	4	4
Average Feasible SR-Sequence	2.9	3.9	3.9
Average Time	93.98s	194.31s	289.31S

表 6-14 Random_Delay_Testing of Example Three(0~2s)

執行次數 / 回合數	5	10	15
1th	3	3	4
2th	4	4	4
3th	4	3	4
4th	4	4	4
5th	3	4	4
6th	4	3	4
7th	3	4	4
8th	4	3	4
9th	3	4	4
10th	3	3	4

Average Feasible SR-Sequence	3.5	3.5	4
Average Time	128.89s	273.87s	387.42s

表 6-15 Random_Delay_Testing of Example Three(0~3s)

執行次數 / 回合數	133	160
1th	11	11
2th	10	14
3th	13	16
4th	10	11
5th	11	13
6th	10	12
7th	13	12
8th	12	11
9th	10	12
10th	12	12
Average Feasible SR-Sequence	11.2	12.4
Average Time	1528.56s	1876.99s

表 6-16 Random_Delay_Testing of Example Four(0~0.5s)

執行次數 / 回合數	133	160
1th	14	16
2th	16	16
3th	11	15
4th	14	18
5th	16	17
6th	15	17
7th	14	14
8th	12	16
9th	17	17
10th	17	19
Average	14.6	16.9

Feasible SR-Sequence		
Average Time	2198.67s	2548.13s

表 6-17 Random_Delay_Testing of Example Four(0~1s)

執行次數 / 回合數	133	160
1th	16	13
2th	18	14
3th	15	13
4th	15	19
5th	18	16
6th	15	17
7th	15	16
8th	17	17
9th	15	15
10th	19	15
Average Feasible SR-Sequence	16.3	15.5
Average Time	2906.33s	3872.9s

表 6-18 Random_Delay_Testing of Example Four(0~1.5s)

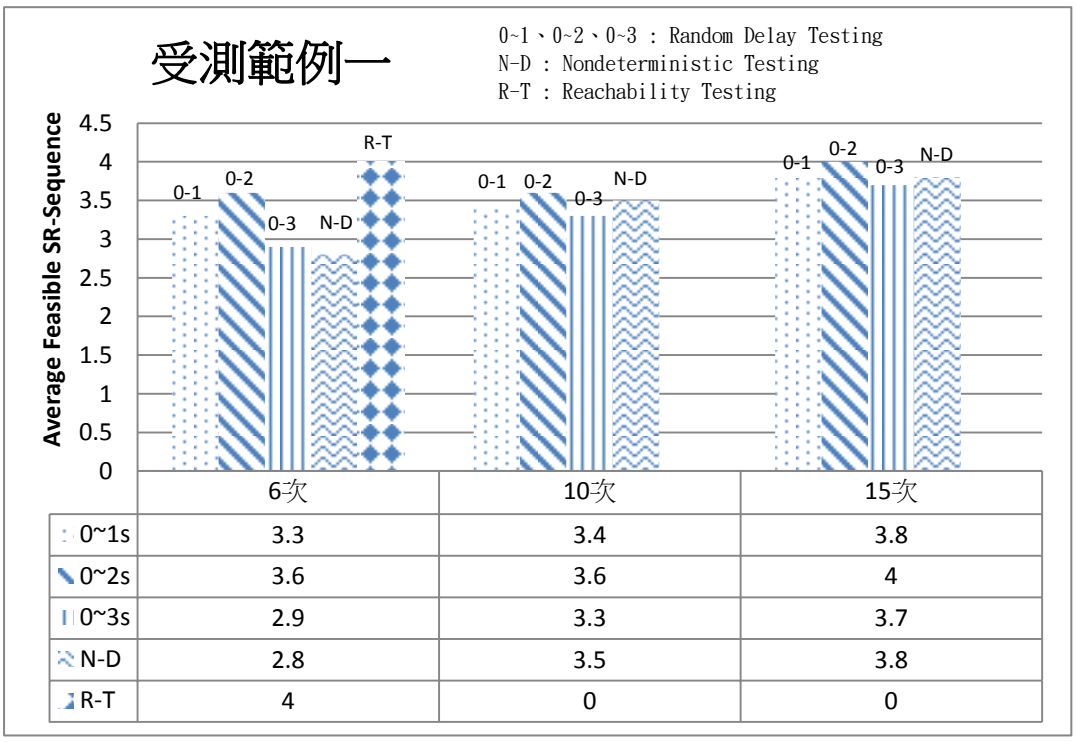
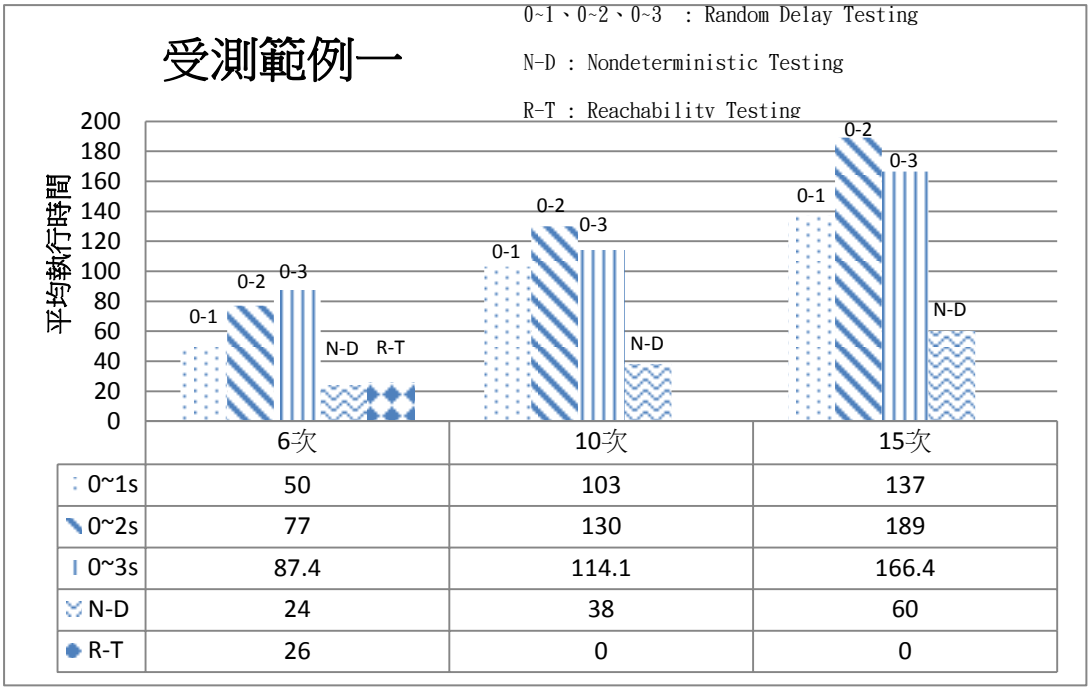
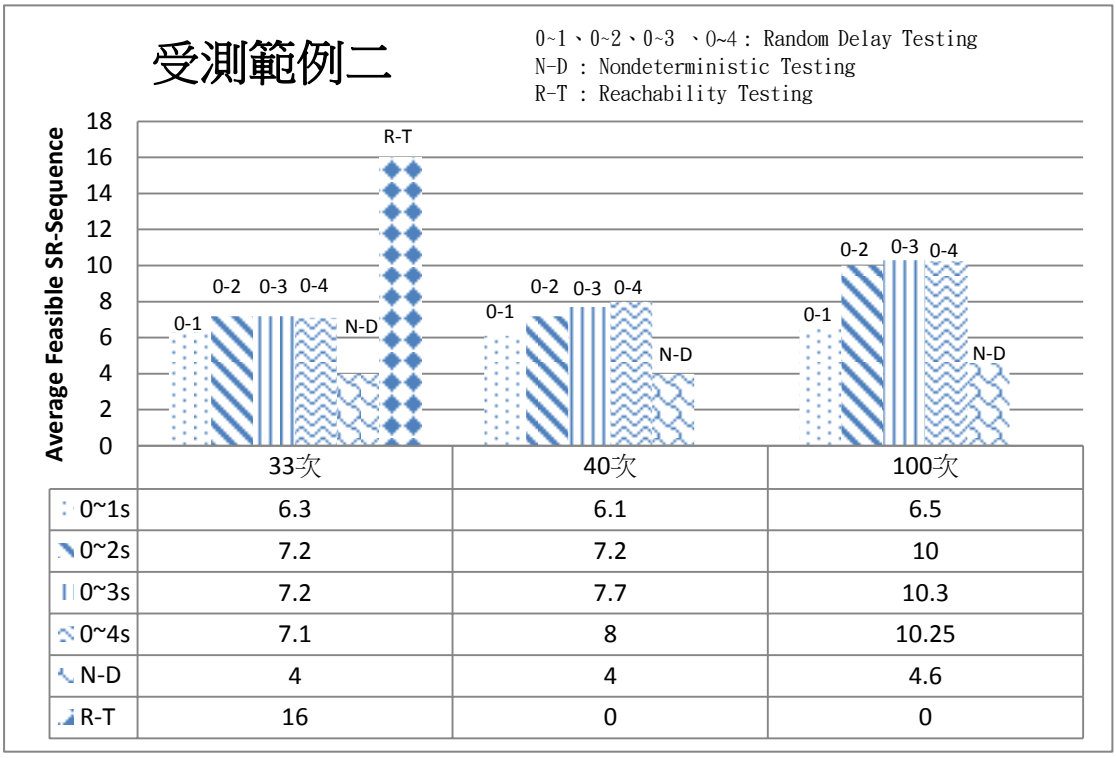


圖 6.9 執行受測範例一在不同測試方法所得平均可行性序列數



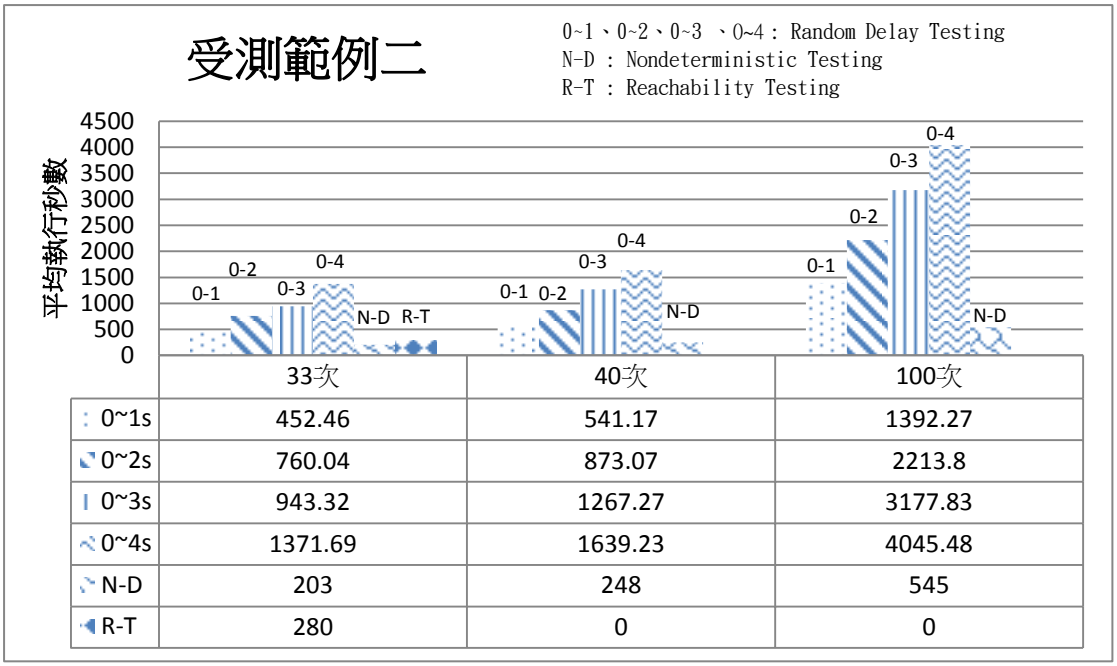
	6次	10次	15次
0~1s	50	103	137
0~2s	77	130	189
0~3s	87.4	114.1	166.4
N-D	24	38	60
R-T	26	0	0

圖 6.10 執行受測範例一在不同測試方法所得平均執行時間



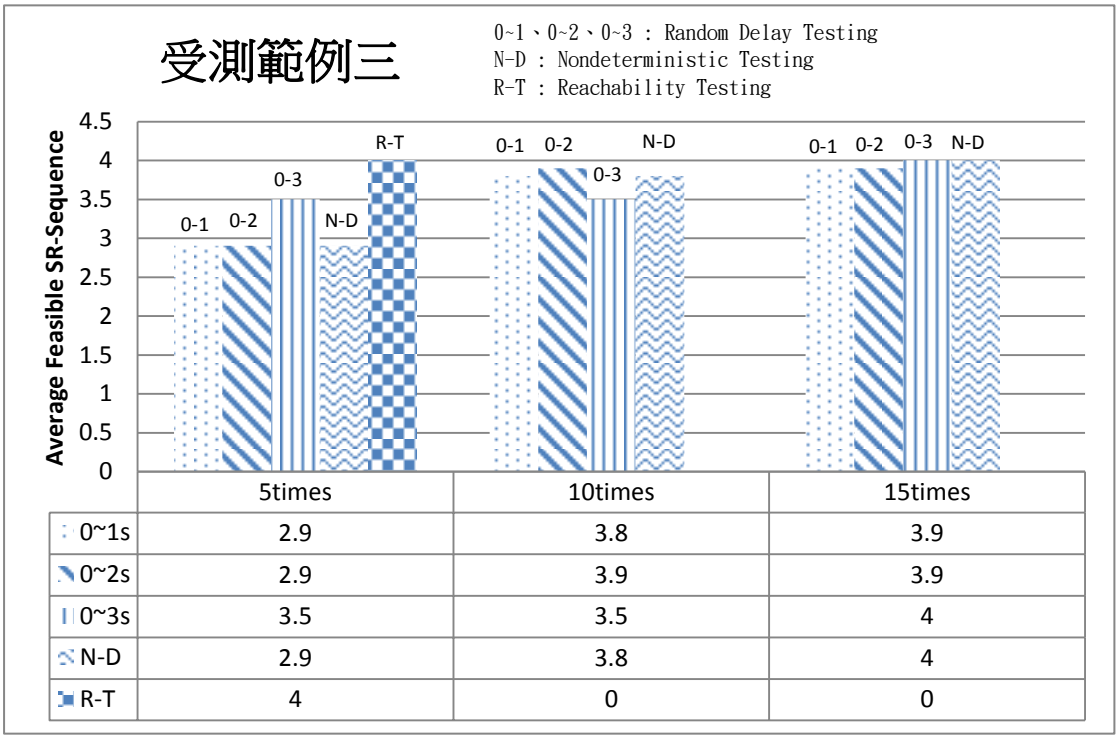
	33次	40次	100次
0~1s	6.3	6.1	6.5
0~2s	7.2	7.2	10
0~3s	7.2	7.7	10.3
0~4s	7.1	8	10.25
N-D	4	4	4.6
R-T	16	0	0

圖 6.11 執行受測範例二在不同測試方法所得平均可行性序列數



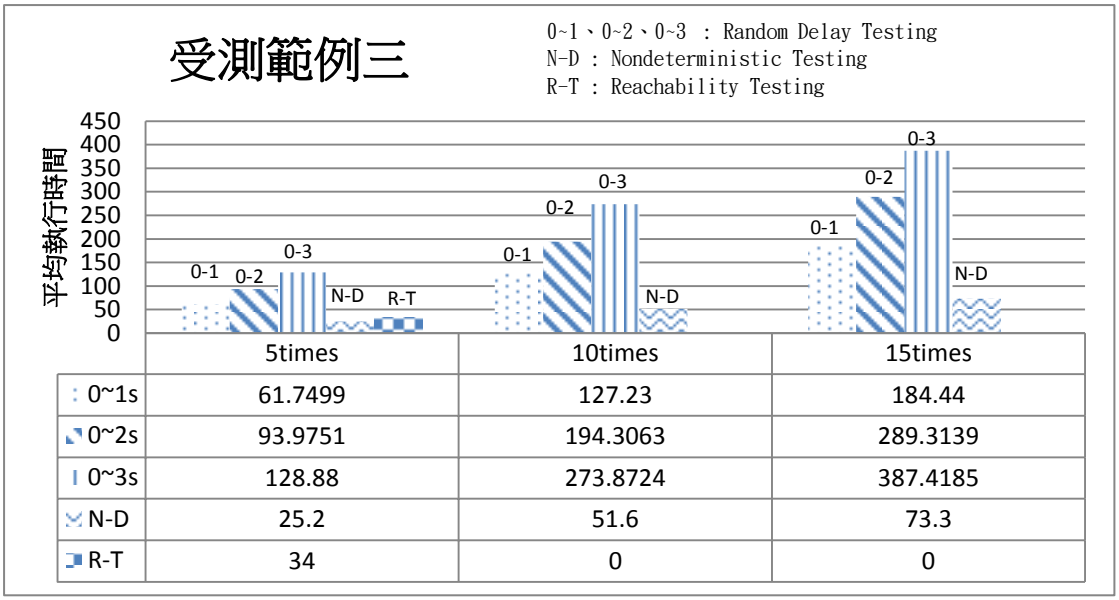
	33次	40次	100次
0~1s	452.46	541.17	1392.27
0~2s	760.04	873.07	2213.8
0~3s	943.32	1267.27	3177.83
0~4s	1371.69	1639.23	4045.48
N-D	203	248	545
R-T	280	0	0

圖 6.12 執行受測範例一在不同測試方法所得平均執行時間



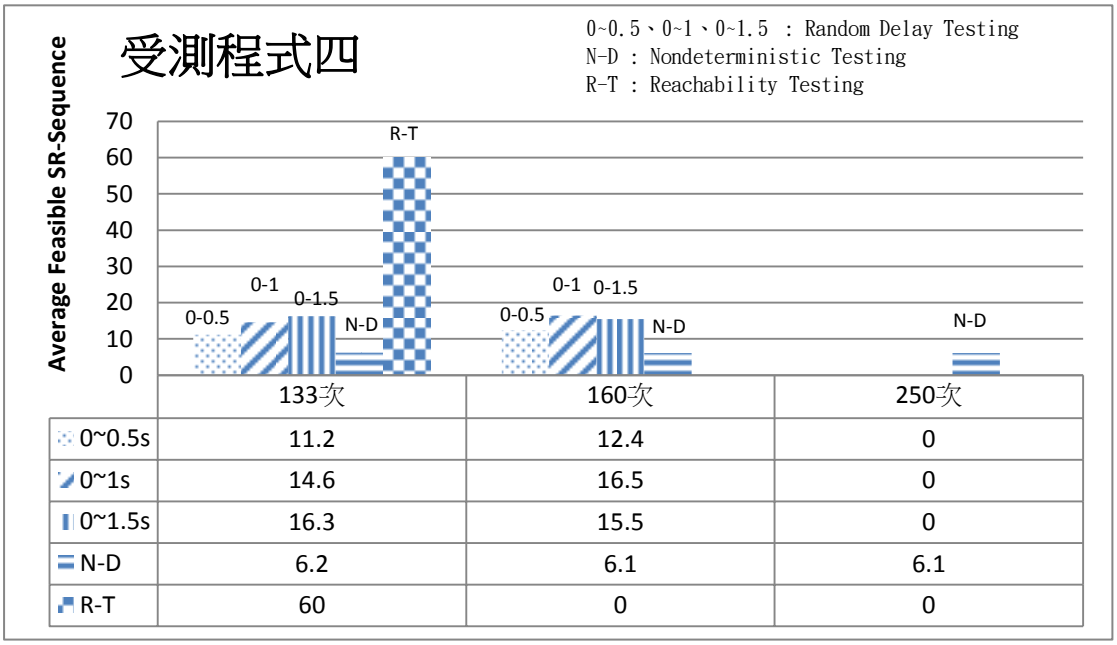
	5times	10times	15times
0~1s	2.9	3.8	3.9
0~2s	2.9	3.9	3.9
0~3s	3.5	3.5	4
N-D	2.9	3.8	4
R-T	4	0	0

圖 6.13 執行受測範例三在不同測試方法所得平均可行性序列數



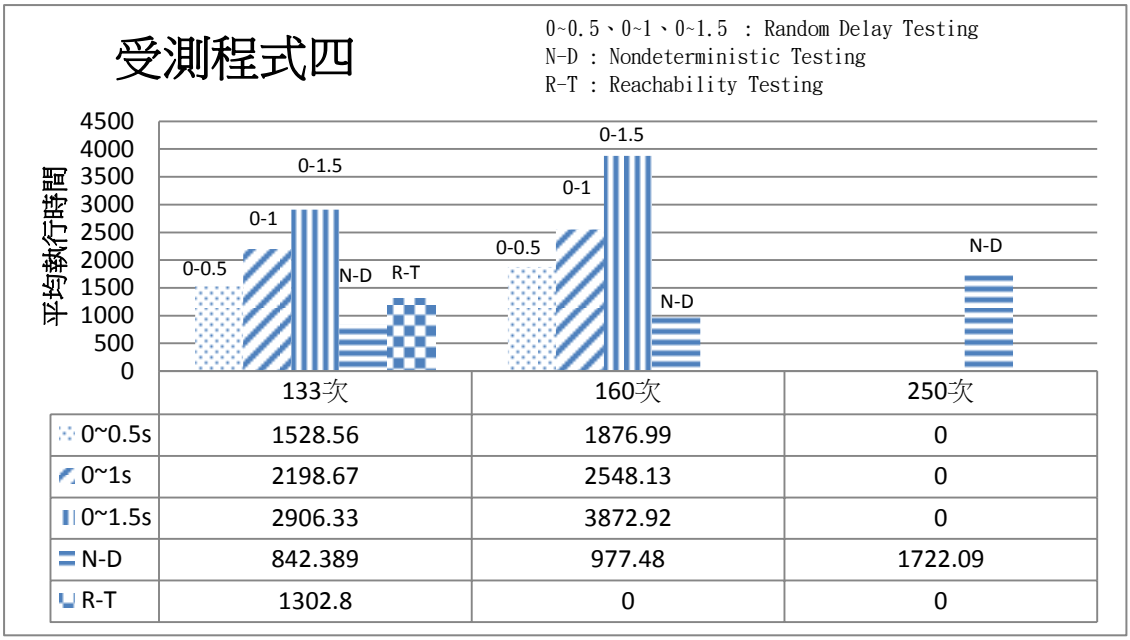
	5times	10times	15times
0~1s	61.7499	127.23	184.44
0~2s	93.9751	194.3063	289.3139
0~3s	128.88	273.8724	387.4185
N-D	25.2	51.6	73.3
R-T	34	0	0

圖 6.14 執行受測範例三在不同測試方法所得平均執行時間



	133次	160次	250次
0~0.5s	11.2	12.4	0
0~1s	14.6	16.5	0
0~1.5s	16.3	15.5	0
N-D	6.2	6.1	6.1
R-T	60	0	0

圖 6.15 執行受測範例四在不同測試方法所得平均可行性序列數



	133次	160次	250次
0~0.5s	1528.56	1876.99	0
0~1s	2198.67	2548.13	0
0~1.5s	2906.33	3872.92	0
N-D	842.389	977.48	1722.09
R-T	1302.8	0	0

圖 6.16 執行受測範例四在不同測試方法所得平均執行時間

根據上述圖表所得知利用可達性測試(Reachability Testing)架構來測試所得到的可行性序列(Feasible Sequence)數是最佳的，相較於其他兩個測試方法，使用測試在例子 1、2 和 3 上，由上面圖表可以觀察到當在與可達性測試測試回合數相同時，非決定性測試以及隨機延遲測試所得到的效果不及可達性測試，在非決定測試方法中當次數提升到原始可達性測試的兩倍次數甚至超過在某些較小的例子上所得到的效果可與可達性測試相同，但可能必須付出多過可達性測試的二到三倍的時間，在無形之中已經失去非決定性測試的優勢了，而在隨機延遲測試方法上所得到的數據中可得知這是一個需要犧牲時間來換取增加可行性序列的方法，在相同次數下(6、33、5)時間是三種方法之中花費最多，雖然比非決定測試所找到序列還多，但當在較多序列的例子上(example_two)效果仍不及可達性測試，在隨機秒數的遞增雖然可以得到可行性序列增加，但所花費的時間已經遠遠超過可達性測試數倍時間，相對也已經失去它的優勢了。

7. Conclusions and Future Work

軟體測試是一門艱深的學問，尤其在並行程式上的測試，傳統採用如非決定性的測試以及決定性的測試作為主要的測試，但前者的測試涵蓋率太低，而後者必須自己推導測試情況，可達性測試是結合前述兩種方法的特點，並且有效的克服並行程式所帶來的非決定性行為，在本篇論文中針對了在單一通道多個接收者非同步訊息傳遞程式上的競爭情況問題作探討，並且使用可達性測試架構為基礎在非同步訊息傳遞程式上的測試，由於原始的可達性架構並非使用在測試非同步訊息傳遞上，所以會發生一些狀況，所以必須針對原始架構作大幅度的修改，在論文中我們重新設計針對非阻擋式發送以及阻擋式接收的進入(Entry)以及離開(Exit)協議做設計，並且利用一協調者伺服器來協調插入在各測試端同步化事件協議之間所需共用變數的存取。並利用執行時的資訊提供給競爭分析器做競爭分析產生競爭變異，在利用這競爭變異提供給受測端來控制事件的執行重複以上步驟以達到測試的目的。

本篇論文中僅針對非同步訊息傳遞程式測試設計，在未來，第一、希望能補足同步訊息傳遞測試，所謂同步訊息傳遞是指發送運算為阻擋式的發送意即發送之後必須等待對方回應表示他收到訊息，而接收運算的部分為阻擋式的接收意即必須等待 Channel 有訊息才能做接收針對上述兩種運算做進入(Entry)以及離開(Exit)協議的設計，才能方便使用可達性測試。

Reference

[1] Charles E. Mcdowell and David P. Helmold, “Debugging Concurrent Programs,” ACM Computing Surveys, Volume 21, Issue 4, December 1989.

[2] K.C. Tai and Richard H. Carver, “Testing of Distributed Programs,” Chapter 33 in Parallel and Distributed Computing Handbook, Editor: A. Y. Zomaya, McGraw-Hill, 1996.

[3] E.Clarke,O.Grumberg,and D. Peled ,”Model Checking”. The MIT Press,1999.

[4] Wikipedia “Message-Passing”, http://en.wikipedia.org/wiki/Message_passing

[5] Abraham Silberschatz, Peter Baer Galvin and Greg Gagne,”Operating System Concepts ” , John Wiley & Sons,ISBN: 0471725951,7th edition(2006)

[6] Yu Lei and Eric Wong , “A Novel Framework for Non-deterministic Testing of Message-Passing Program”,pp. 66 - 75,HASE 2005. Oct. 2005

[7] Yu Lei and Richard H. Carver, “Reachability Testing of Concurrent Programs,” IEEE Transaction on Software Engineering, Volume 32, Number 6, pp. 382-403, June 2006.

[8] Che-Sheng Lin and Gwan-Hwan Hwang ,“State-cover Testing for Nondeterministic Concurrent Programs with an infite number of Synchronization Sequence”

[9] Gwan-Hwan Hwang , Kuo-Chung Tai and Ting-Lu Huang, “Rechability Testing : An Approach to Testing Concurrent Software” Software Engineering Conference,pp246-255, 7-9 Dec 1994

[10] Gwan-Hwan Hwang , Sheng-Jen Chang and Huey-Der Chu, “Technology for Testing Nondeterministic Client/Server Database Applications”, IEEE Transactions on Software Engineering,pp59-77, April 2004

[11] J Lei , “Non-deterministic Testing of Concurrent Programs”, ISSRE 2003

[12] Jason Gait, “A probe effect in concurrent programs”, Software: Practice and

Experience, Volume 16, Issue 3, pages 225–233, March 1986

[13] D. Heimbold and D. Luckham, “Debugging Ada Tasking Programs”, Journal IEEE Software archive, Volume 2 Issue 2, March 1985

[14] Yu Lei and Richard Carver, “Reachability Testing of Semaphore-based Programs”, COMPSAC '04 Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01,2004

[15] Yu Lei and Richard Carver, “A New Algorithm for Reachability Testing of Concurrent Programs”, ISSRE’05 , 2005

[16] Kuo-Chung Tai ,” Rechability Tesing of Asynchronous Message Passing Program”, Software Engineering for Parallel and Distributed Systems,pp50-61., 1997

[17] TOSHIAKI KUROKAWA and MASATO SHINAGAWA , “Technical Trends and Challenges of Software Testing”, pp34-45. 2008

[18] Kuo-Chung Tai ,” Race analysis of traces of asynchronous message-passing programs”, International Conference on Distributed Computing Systems, 1997.

[19] Yu Lei and Kuo-Chung Tai, ” Effcient Reachability Testing of Asynchronous Message-Passing Program”, ICECCS '02 Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, 2002.

[20] RICHARD H. CARVER and KLJO-CHLJNG TAI , “Replay and testing for concurrent programs”, pp66-74. Volume: 8, Issue: 2, March 1991

[21] Kuo-Chung Tai , Richard H. Carver and Evelyn E. Obaid ,“Debugging concurrent Ada programs by deterministic execution”, IEEE Transactions on Software Engineering, Volume 17 Issue 1, January 1991