國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

軟體代理人程式測試方法之研究

The Study of Software Agent Program Testing Methodology

研 究 生 ：吳 偉 聖

指 導 教 授：鍾 乾 癸 教授

中 華 民 國 九 十 三 年 六 月

軟體代理人程式測試方法之研究

The Study of Software Agent Program Testing Methodology

研 究 生：吳偉聖　　Student :Wei-Sheng Wu

指導教授：鍾乾癸　　Advisor : Chyan-Goei Chung

國 立 交 通 大 學

資 訊 工 程 學 系

碩士論文

A Thesis

Submitted to Department of Computer Science and Information Engineering College of Electrical Engineering and Computer Science National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science and Information Engineering

June 2004

Hsinchu, Taiwan, Republic of China

中華民國九十三年　六　月

軟體代理人程式測試方法之研究

研究生：吳偉聖　　指導教授：鍾乾癸

國立交通大學
資訊工程研究所

摘要

軟體代理人程式具有學習人類的行為模式、互相的合作以自動地解決繁雜的問題，以及透過行動力可以在網路上任意地點執行工作的優點，因此漸漸被應用在資料收集、資料過濾、資訊管理、電子商務、工作流程管理、網路管理及服務管理等各項領域之上，而現今雖然有軟體代理人測試環境被提出，但尚未有人依軟體代理人特性提出系統化測試方法，常造成軟體代理人測試不夠完整，而易於發生執行錯誤。

本論文根據軟體代理人的四種特性：(1)自動化，(2)學習能力，(3)行動力及(4)合作能力，以及軟體代理人間的兩種關係：(1)主從關係與(2)供需關係，擬定一套可完整測試軟體代理人行為與特性的測試順序之策略，並提出可避免重複測試的滾動式軟體代理人程式的測試方法。本研究並根據這套測試策略與方法製作一軟體代理人程式測試環境，此環境具有下列功能自動剖析軟體代理人程式、建立軟體代理人間的關係，及產生軟體代理人的測試順序的功能。

本研究分析軟體代理人的特性與行為模式來進行分析，找出軟體代理人特性對行為模式的影響，並歸納出軟體代理人間的主從關係與供需關係；藉由軟體代理人間相互影響的關係，可找出多軟體代理人系統的測試順序，而對單一軟體代理人程式的測試，經分析各項特性間交互影響後關係，提出必須先測試自動化特性，繼而測試行動力特性，最後測試合作能力，而學習力則必入自動化特性中測試，進而提出一套滾動式的軟體代理人程式測試方法，本研究的測試方法以 IBM Aglets 軟體代理人程式語言為範例，並在 IBM Aglets 軟體代理人執行環境開發測試工具。

The Study of Software Agent Program Testing Methodology

Student: Wei-Sheng Wu Advisor: Chyan-Goei Chung
Institute of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University

ABSTRACT

Software agent programs have three advantages include learning human behaviors, automatically solving complicated problems by collaboration, and executing tasks anywhere in networks by mobility. Therefore, software agent programs gradually have been applied in domains of data collection, data filtering, information management, e-commerce, task process management, network and service administration, etc. Although there are testing environments for software agents that have been suggested, nobody has proposed a systematic testing method according to the characteristics of software agents. That usually causes an incomplete testing of software agents and makes execution errors occur frequently.

This thesis is based on four characteristics of software agents: (1) Autonomy (2) Learning (3) Mobility (4) Collaboration, and two relationships between software agents: (1) master-slave (2) provider-requestor, to design a strategy that could perform a thoroughly testing on software agents behaviors and testing orders of characteristics. We also implemented a testing environment according to these testing strategies and methods. This environment has following abilities: parsing agent programs, building relationships between agents, and generating testing orders of agents.

In this thesis, the characteristics of software agents and their behavior modes are analyzed to find the influences of characteristics on behavior modes and to infer the master-slave and provider-requestor relationships among agents. Testing orders could be founded by relationships between agents. After the analysis of reciprocal effects among characteristics, we proposed that in the testing on a single agent, the autonomy should be tested first, the mobility is tested then, and the collaboration is tested finally. The testing of learning is incorporated into the testing of autonomy. A rolling-style testing method of software agents is proposed. Our testing method takes the IBM Aglets software agent language as an example, and develops testing tools under the IBM Aglets execution environment.

誌謝

首先感謝指導老師鍾乾癸教授兩年來在學識研究及做人做事上耐心地教導並不斷地給予我幫助，特別是在論文撰寫階段，更是來回的奔波且不厭其煩地指導我，不但使得我能夠完成碩士的學業，也讓我學會嚴謹的研究態度及做人處世的道理，對於未來要踏上社會的我有莫大的幫。

然後感謝實驗室鄭靜紋學姊的幫助，兩年多來也是不斷地從旁協助我作研究，不厭其煩地與我討論，並指引我正確的研究方向，使得我能夠順利的完成這篇論文，同時也要感謝實驗室的同學與學弟，不斷地給予我鼓勵與幫助，使得我能夠一路堅持下去。

最後，感謝我的父母及家人一直以來的支持與關心，僅以此成果與家人及所有關心我的師長及同學們分享。

目錄

圖目錄

表目錄

第1章　序論

1.1.研究背景與動機

軟體代理人的定義[1]是當一軟體能夠完成使用者所指派給它的任務，完成的過程就像使用者自己親自去完成的一樣時，便稱它是一個軟體代理人程式，為了達成這個目的所以軟體代理人程式必須具有四種特性[2]：自動化、學習能力、行動力及合作能力，這四種特性是傳統程式所沒有的，因此軟體代理人不論是執行的時間上、系統的彈性與資源的利用都較傳統程式佔優勢，且軟體代理人比傳統程式更有彈性、主動性以及更能貼近使用者的想法，所以軟體代理人將比傳統應用程式更適合用在資料收集、資料過濾、資訊管理、電子商務、工作流程管理、網路管理、服務管理…等應用領域[3]。

現今有關軟體代理人的研究，主要集中在軟體代理人的程式語言的開發與應用[4,5]，而關於軟體代理人程式測試的方面，大多偏重於軟體代理人模擬環境之開發[4,5]，以方便軟體代理人程式之測試，然而當一軟體代理人系統擁有多個軟體代理人，如何有系統地對此多軟體代理人系統程式進行完整測試的研究卻無人提出，對開發者而言，多軟體代理人程式之測試仍是一件困難之事。

欲探討軟體代理人程式測試方法，首先需了解軟體代理人程式與傳統程式的差異；因此本論文首先分析單一軟體代理人的行為模式與軟體代理人程式的四個特性：自動化、學習能力、行動力及合作能力，發現自動化特性為軟體代理人完成目標的核心邏輯，其正確性影響其他特性的執行結果，因此必須先測試，且只要提供足夠的資料，則自動化便可在單一軟體代理人執行環境上測試；學習能力則透過修改知識庫來改變軟體代理人的執行邏輯，因此學習能力只需測試更新知識庫的動作是否正確；當資料不足時，軟體代理人才執行行動力之動作，且行動力的動作本身並不影響其他特性的執行，因此可以跟其他特性分開來測試；而軟

體代理人間之溝通是在傳遞資料或回傳值以達到合作之目的，當測試單一軟體代理人時，必須利用 stub 來測試，根據這些發現便可擬定出軟體代理人的測試策略，因此可先測試自動化、學習能力，然後測試行動力的特性，最後再利用 stub 來測試合作能力。當多軟體代理人系統時，無法一次測試所有的軟體代理人，可透過分析軟體代理人間的合作行為模式，歸納出軟體代理人間的兩種關係：主從關係(Master-Slave)與供需關係(Provider-Requestor)，從主從關係中發現，必須達成所有 Slave 的目標，才能達成 Master 的目標；而從供需關係中發現，Requestor 委託部分工作給 Provider，可依此來決定該系統中各軟體代理人的測試順序。

本研究即根據上述，深入探討多軟體代理人系統之測試方法，以方便軟體代理人程式開發者可更系統化及效率化地測試其程式。

1.2. 章節簡介

本論文一共分成六章，除了本章外，其他章節介紹如下：

第二章介紹軟體代理人程式特性與程式結構、並介紹現有的研究成果及問題。

第三章首先分析軟體代理人的執行行為，包含單一軟體代理人系統與多軟體代理人系統的執行行為，並提出軟體代理人間關係的建立方法，進而提出軟體代理人系統的測試策略。

第四章介紹軟體代理人之測試方法，包括剖析軟體代理人程式，以建立軟體代理人測試的相關資料、軟體代理人測試順序安排方法，及提出軟體代理人程式的測試方法。

第五章總結本研究的貢獻並提出未來方向。

第2章　背景知識與相關研究

本章分成三個部分，2.1 節，介紹軟體代理人的特性及 IBM Aglets[6]軟體代理人程式的基本元件；2.2 節，介紹目前軟體代理人程式測試的研究現況與其缺失；2.3 節，提出軟體代理人程式測試的改進方向。

2.1.軟體代理人程式的簡介

軟體代理人的構想是讓代理人學習人類的行為模式，藉由代理人間互相的合作來自動地幫助人類解決繁雜的問題。本節將簡介軟體代理人程式的基本元件與其所具有的特性。

2.1.1.　軟體代理人程式的基本元件

由於近代科技的快速發展，使得過多資訊的大量湧現，讓人產生不知該如何有效率地去獲得所需要的資訊的困擾，如果能夠「讓使用者只要給予他所想要達成的工作目標，而執行的細節則完全交給一個程式去代為執行，使用者則只需等待接收完成任務後的結果，而無需擔心如何完成任務的整個過程」[3]，可幫使用者減輕不少使用電腦的負擔，因此便興起了軟體代理人 (Software Agent)[1]的構想，讓代理人學習人類的行為模式，藉由代理人間互相的合作來自動地幫助人類解決繁雜的事物。

我們把軟體代理人之定義為：當一個軟體能夠完成使用者所指派給它的任務，完成的過程就像使用者自己親自去完成的一樣時，我們就稱它是一個軟體代理人程式。而這裡所指的使用者並不一定只限於人類，它可以是一般軟體，也可以是另一個軟體代理人程式。

一個軟體代理人要能夠正確的運作，需要兩個部分的支援，第一，需要一個軟體代理人程式語言，使用者藉由軟體代理人程式語言來編寫他們所需要的軟體代理人程式；第二，需要一個讓軟體代理人執行的外部環境，這包含負責提供代

理人執行的環境(Context)與負責處理資源的代理人伺服器(Agent Server)。

軟體代理人本身是由軟體代理人程式語言構成的程式碼，軟體代理人程式語言大多是以 JAVA 為基礎，如:IBM Aglets、Objectspace Voyager、Jade 等，軟體代理人程式語言提供一套 JAVA API，包含多個 JAVA 類別跟界面，來讓使用者直接在 JAVA 的程式編輯器上編寫軟體代理人程式,使用 JAVA 為軟體代理人程式語言的基礎具有下列優點[6]：

1. 跨平台執行：JAVA 程式可在異質(Heterogeneous)環境下執行，因此軟體代理人程式具有可在網路上任意地點工作的能力。

2. 更安全的執行：JAVA 語言可用來設計對網路應用的程式，因此對於安全性需求的考量較多，如：不允許使用 Pointer 以避免記憶體覆寫(Memory Overwriting)和資料損毀(Data Corruption)的情況，所以使得軟體代理人的安全性更高。

3. 動態類別載入(Dynamic Class Loading)：這個機制供每個軟體代理人可在個自的 name space 執行，以確保所有的軟體代理人獨立性與安全性。

4. 多執行緒(Multithread)功能：每個軟體代理人具有自己的執行緒，使得軟體代理人能夠自動化地執行。

5. 物件序列化(Object Serialization)：透過這個機制可將執行時期的物件狀態儲存下來，再透過網路來傳送，使得軟體代理人具有行動能力。

6. Reflection：JAVA 程式碼執行時期的物件中可找出有關的屬性(Fields)並使用，使得軟體代理人在執行遷徙之後，能夠回復軟體代理人原新的資料狀態。

軟體代理人程式繼承原本 JAVA 程式的編寫方式，所以軟體代理人程式由下

列四種元件組成：代理人(Agent)、包裹(Package)、類別(Class)、界面(Interface)，JAVA 軟體代理人程式的組成如圖 2-1 所示，一個代理人可包含一或多個包裹，而一包裹包括一群相關類別及界面，類別及界面間的有下列三種關係[7][8][9][10]：

1. 繼承關係：繼承關係存在於類別與類別之間、界面與界面間，類別間的繼承關係只允許單一繼承；而界面間的繼承關係則允許多重繼承。

2. 實現關係：實現關係存在於類別與界面間，一個類別可以實現多個界面，一個界面也可以被多個類別實現。

3. 使用關係：使用關係存在於類別與類別間、類別與界面間。

上述關係完全與 JAVA 程式語言相同，不另作贅述。但所設計的系統不只有一個代理人時，代理人間的關係便無法再引用上述的關係，此因代理人對於外部環境的對應，是交由代理人伺服器來代為處理，代理人間不能夠直接去使用其他代理人的函式或變數，必須透過軟體代理人程式語言指定的方法來作溝通，待介紹軟體代理人外部執行環境後，再介紹代理人跟外部執行環境間的互動。

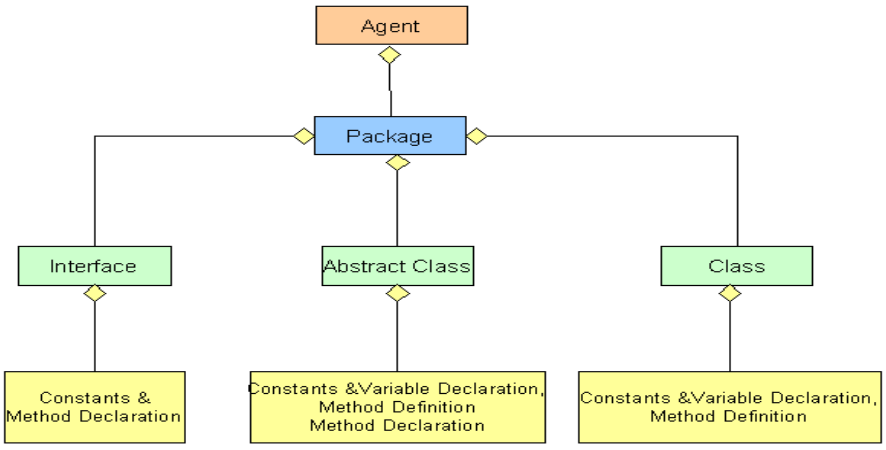

圖 2-1 軟體代理人程式中的元件

一軟體代理人除了必須在外部執行環境下才能夠執行，外部執行環境包含[6]：Proxy: Proxy 物件扮演軟體代理人代表的角色，它如同軟體代理人外層的

界面保護著軟體代理人，不讓其他的軟體代理人能夠直接的存取軟體代理人的公開函式(public method)，如果其他的軟體代理人要與其作互動的話，必須要透過 proxy 來跟該軟體代理人溝通，如圖 2- 2 所示。

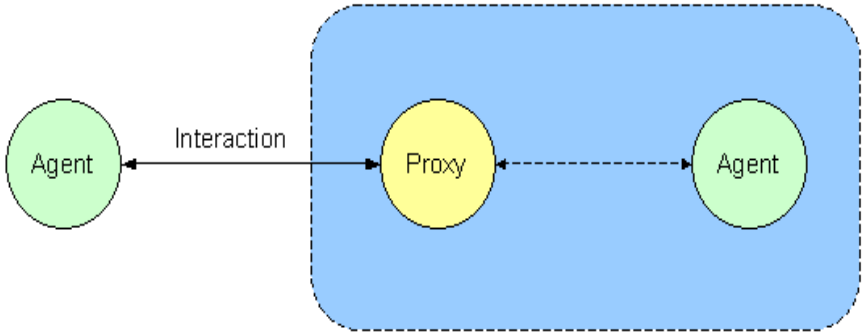

圖 2- 2 軟體代理人與 Proxy 的關係圖 Context：是軟體代理人的工作環境，它是一個固定的物件，提供方法讓軟體代理人可以在這個執行環境下運作，並隔絕外來的惡意軟體代理人的攻擊。一個主機上可有多個代理人伺服器，每個代理人伺服器又可管理一或多個 context，如圖 2- 3 所示。

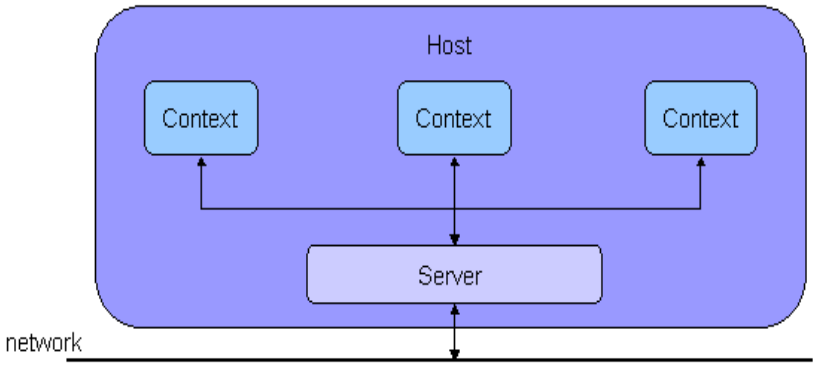

圖 2- 3 主機、代理人伺服器、context 間的關係圖

3. Identifier：每個 Identifier 對應一軟體代理人，在一軟體代理人的生命週期，它的 Identifier 是獨一無二且不能改變的。

2.1.2. 軟體代理人特性的分析

介紹軟體代理人程式的元件結構以及軟體的外部執行環境後，雖然能了解軟

體代理人的表面(程式碼)和它跟外部環境間的互動，卻無法了解軟體代理人內在的特性，這些特性才是了解使用者想利用軟體代理人來達到他們的目標的關鍵。

軟體代理人具有下列主要特性[2]：

1. 自動化(Autonomy)：軟體代理人在完成任務期間所作的動作，都是由軟體代理人自行作決定，不需要使用者的介入。
2. 合作(Collaboration)能力：軟體代理人彼此之間可以透過產生(Create)、溝通(Communication)、發送(Dispatch)等動作來共同合作完成使用者指派的任務。
3. 行動力(Mobility)：軟體代理人為了完成任務，可以遷移到它所需要用到的環境上去執行任務。
4. 反應力 (Reaction)：軟體代理人可以偵測所處環境的變化，並自行做出適當的反應動作。
5. 學習(Learning)能力：軟體代理人藉由和使用者溝通、和其他軟體代理人溝通、或經由完成過的任務之過程與結果，可以從中學習記憶某些有用的經驗，或修正某些錯誤的執行方式。所以一旦有類似或相同的任務指派時，可以依照所學得的經驗來完成任務，以增加效率。

上述特性中，反應力為自動化特性的一種表現，所以軟體代理人，最主要特性為：自動化(Autonomy)、合作能力(Collaboration)、行動力(Mobility)和學習能力(Learning)，同時具備此四特性的軟體代理人，稱為「聰明代理人(Smart Agent)」[2]，如圖 2-4 所示。

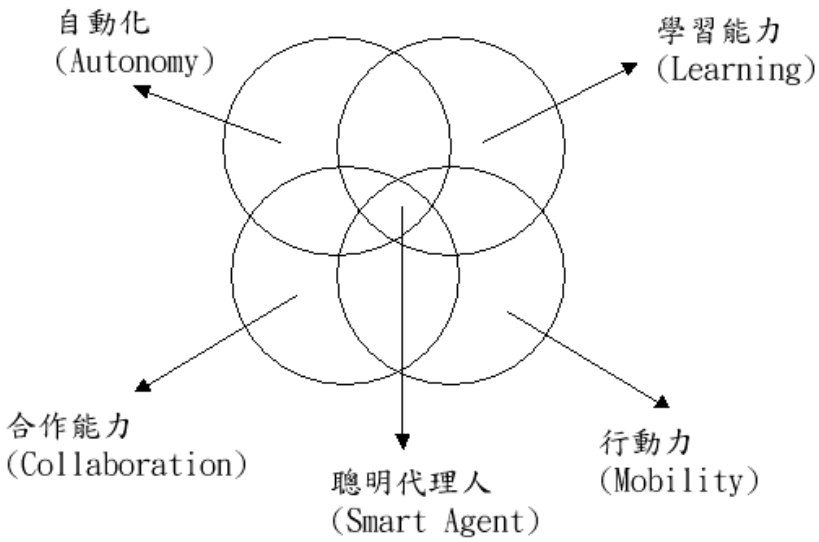

圖 2-4 軟體代理人的四大重要特性

由於軟體代理人是一獨獨立完整程式，啟動後即自動執行，程式內容是依軟體代理人的目標而設計，包含對外部環境的偵錯與因應，因此具有自動化的特性，程式內容根據不同的應用而設計，很難有固定的形式。

以下敘述其他三特性：

2.1.2.1. 合作能力(Collaboration)

軟體代理人的合作能力使得多個軟體代理人可以透過互動來完成系統的目標，軟體代理人語言提供了下列三種合作的方法：產生(create)、訊息傳遞(communication)、發送(Dispatch)[6]，供設計者設計軟體代理人系統的合作機制：

1. 產生(create)：一個軟體代理人(產生者)可因工作需要而產生另外一個軟體代理人(被產生者)來協助他的工作，產生者可利用產生的指令告知軟體代理人伺服器要被產生的軟體代理人的位址、程式碼的類別名稱以及相關的參數，代理人伺服器便可在指定的位址產生另一軟體代理人，且產生者也可得到被產生者的 proxy，因此產生者就可以透過這 proxy 去指揮被產生者執行工作，以 IBM Aglets[6]為例，Aglets 提供

createAglet 的指令，透過這個指令就可以產生它所要的軟體代理人，如圖 2-5 所示。

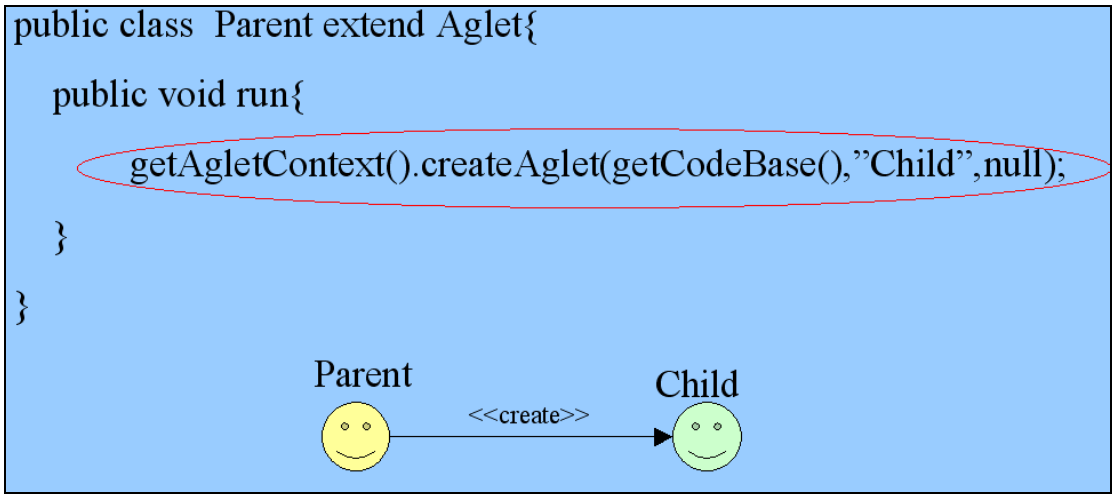

圖 2-5 軟體代理人的產生動作

2. 發送(Dispatch)：由上述的情況，當被產生者必須要到遠端執行工作時，產生者(發送者)可以透過 proxy 指揮被產生者(被發送者)執行 mobile 的動作，這樣的動作便稱為「發送(Dispatch)」，使用事件導向(event-driven)的方式，發送者會具有發送的指令，發送指令內包含了指定的移動位址，如圖 2-6 所示，被發送者端則設置一個 Listener，負責接收執行 mobile 的事件，並處理執行 mobile 的動作，以及 mobile 後的起始動作，這個部分等到下一個部分談到行動力時再詳細說明。

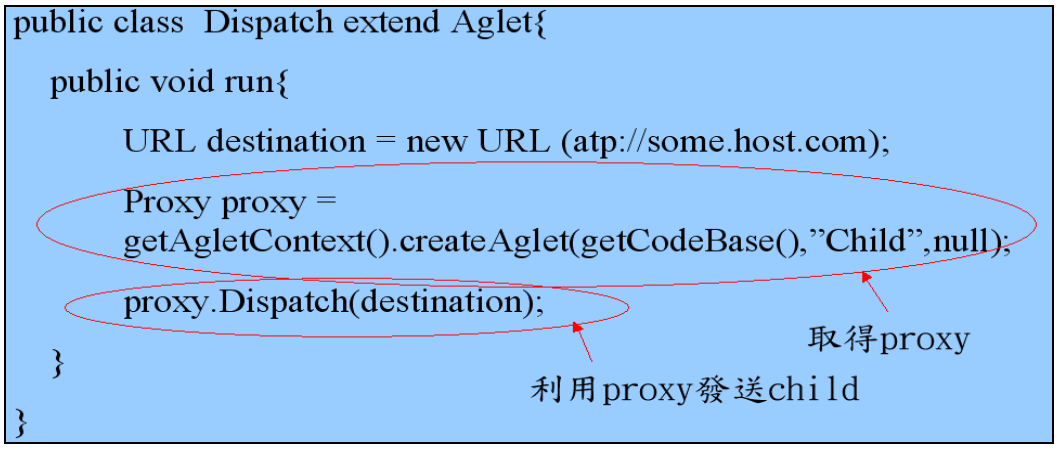

圖 2-6 發送者

3. 訊息傳遞(communication)：軟體代理人間無法像傳統的程式透過呼叫函式的方式來互動，而是透過訊息傳遞機制來互動：傳送者(Sender)和接收者(Receiver)，會透過約定好的事件來互動。當傳送者需要接收者的幫助時，需具備接收者的 proxy，取得的方式有下列四種：

   i. 透過產生接收方的軟體代理人取得。

   ii. 如果兩者都在相同代理人伺服器上的話，接收者為自己的 proxy 跟代理人伺服器設定一個 key，則傳送者便可利用相同的 key 來取得 proxy。

   iii. 由接收者先利用訊息傳遞主動將 proxy 傳送給傳送者。

   iv. 存在一個第三方的軟體代理人，傳送者可以主動跟它詢問接收者的 proxy 或第三者主動將接收者的 proxy 告知傳送者。

   另有一種訊息傳遞的方式是不需要取得 proxy，而是透過傳送者跟接收者約定好訊息，當傳送者傳送出該訊息後，約定好這個訊息的接收者就可以接收及處理，這種方式稱為群播(multicast)[6]，透過這種方式可以同時跟多個軟體代理人溝通，但必須要傳送者及接收者都在相同的代理人伺服器中才可行。

   取得 proxy 之後，傳送者會傳送包含事件的訊息給接收者，接收者收到訊息之後可以判斷屬於哪一種事件而作對應的動作，並將執行結果回傳給傳送者[6]。

   傳送者執行傳送方式有下列三種[6]：

   i. 同步傳送(Synchronous Send)：傳送者使用這個方法傳送訊息時，傳送者的執行緒會停在這個指令，直到收到接收者回傳結果或是傳送發生錯誤(Exception)，執行緒才會繼續執行，

以 IBM Aglets 為例，Aglets 提供了 sendMessage 這個指令和訊息(Message)物件，必須要取得接收者的 proxy 物件才能夠使用 sendMessage，並宣告 Reply 物件來接收回傳結果，如圖 2-7 所示。

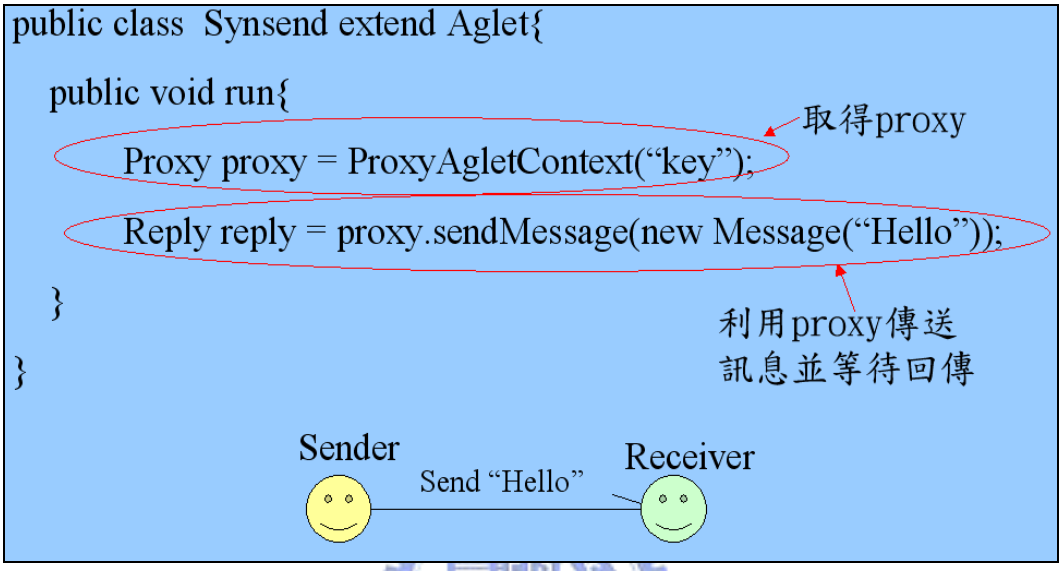

圖 2-7 同步傳送(Synchronous Send)

ii. 非同步傳送(Asynchronous Send)：當傳送者使用這種方式傳送訊息時，傳送者的執行緒不會停留在傳送的指令，可繼續執行後繼續執行指令。接收者回傳的回應(Reply)物件，是由代理人伺服器暫存起來，傳送者可再需使用回傳訊息時再詢問底層的代理人伺服器，有沒有來自接收者的回應，如果有的話，代理人伺服器就會將回應物件送給傳送者，以 IBM Aglets 為例，Aglets 提供了 sendFutureMessage 這個指令，同樣需要取得接收者的 proxy 物件才能夠使用，並提供 FutureReply 的物件，這個物件並不會馬上接收回應，等到傳送者想要詢問回傳時，才利用該物件去跟代理人伺服器取得回傳物件，如圖 2-8 所示。

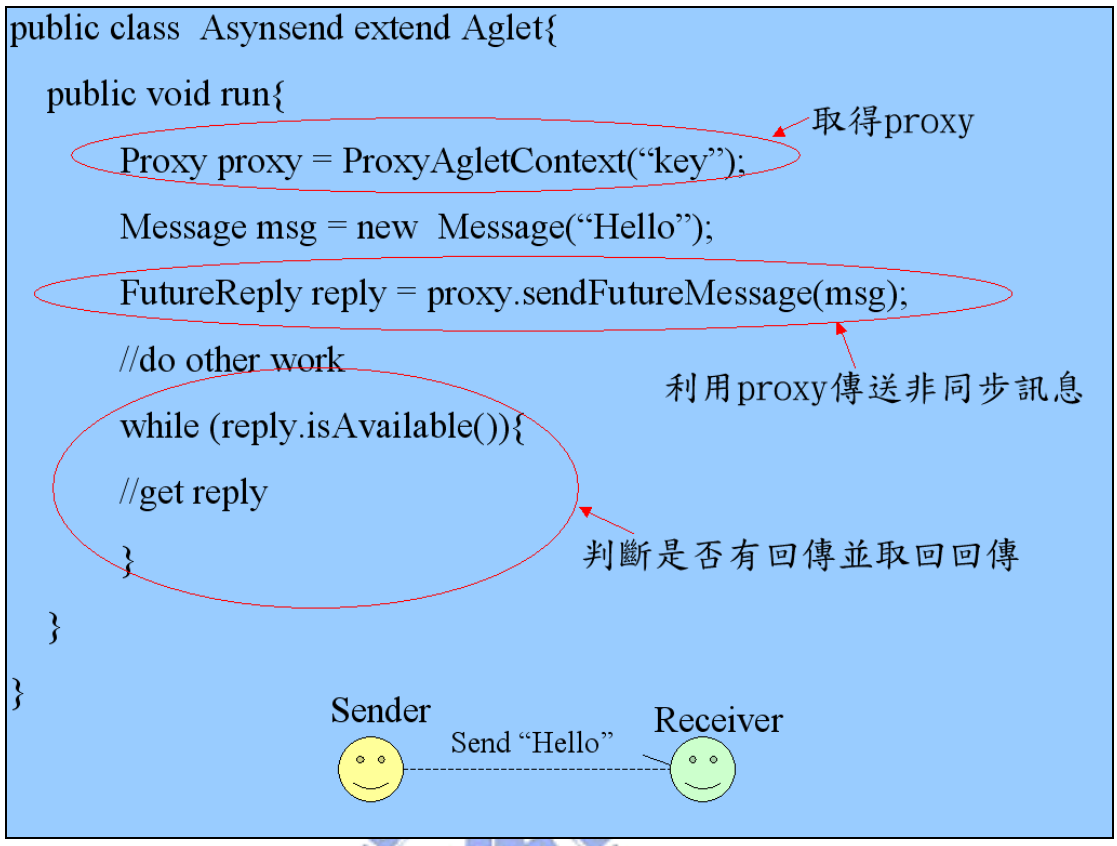

圖 2- 8 非同步傳送(Asynchronous Send)

iii. 群播傳送：當傳送者想同時傳送訊息給多個接收者時，就可以利用群播的方式來傳送，傳送者需與接收者約定好特定的訊息內容，則當傳送者利用群播的方式將訊息傳送出去後，代理人伺服器就會將這個訊息傳給所有訂閱(Subscribe)這個訊息的接收者；其等待回傳的方式，則與非同步傳送相同，傳送者送出訊息後會繼續執行，以 IBM Aglets 為例，Aglets 提供了傳送者 multicastMessage 這個指令以及 ReplySet 的物件，而接收者則可以使用 subscribeMessage 這個指令來約定訊息，傳送者則利用 FutureReply 的物件來作接收回傳的動作，如圖 2- 9、圖 2- 10 所示。

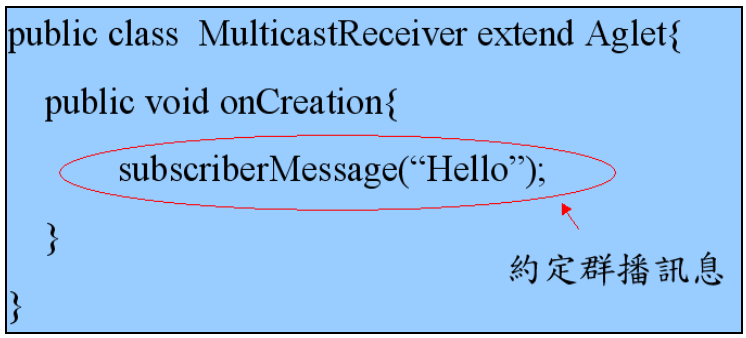

圖 2- 9 群播(Multicast)接收者

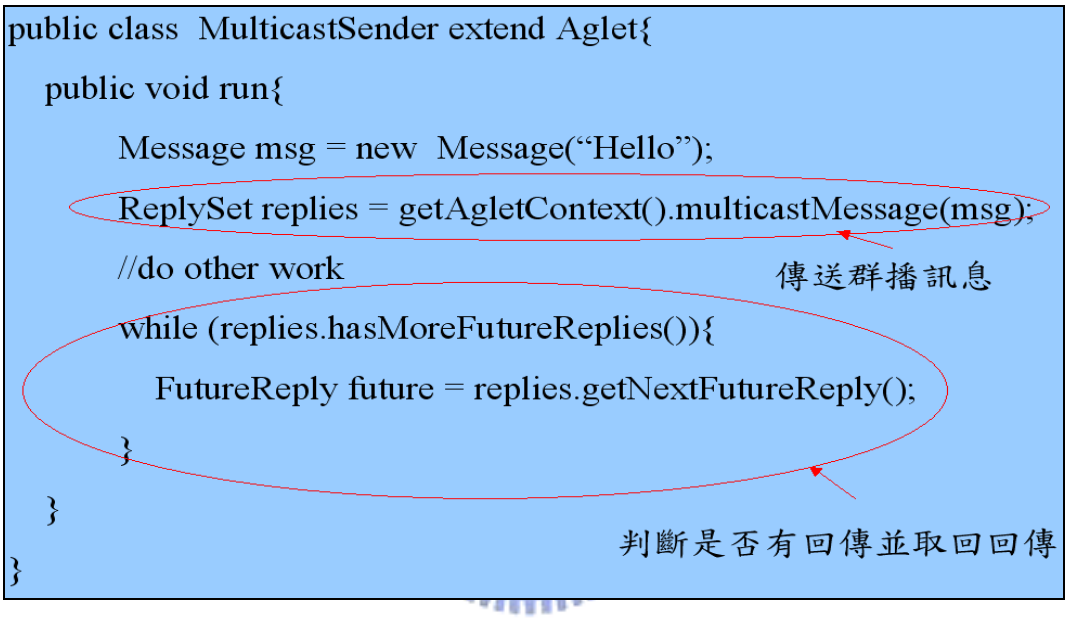

圖 2- 10 群播(Multicast)傳送者

而接收者處理訊息的方式則分為下列兩種方式[6]：

i. 循序式訊息處理(Sequential Message Handle)：所有軟體代理人都有一個專門處理外來訊息的函式「Message Handler」，當有訊息傳給此軟體代理人時，底層的代理人伺服器會先收到這個訊息，接著根據 proxy 將這個訊息當成參數去呼叫接收者的 message handler，並將訊息放在訊息序列(Message Queue)中。當軟體代理人處於等待的狀態時，Message Handler 即自動啟動執行，若有多個訊息需處理時，其預設方式是一次處理一個訊息，且採先送先處理的方式，因

此稱為「循序式訊息處理(Sequential Message Handle)」，以 IBM Aglets 為例，Aglets 提供一個名為 MessageHandle 的函式，預設為同步函式(Synchronized Function)，在同一個時間內只能被呼叫一次，因此一次只能處理一個訊息，且根據是否要回傳，利用 sendReply 的指令便可還將回傳物件送給代理人伺服器，再由代理人伺服器傳給傳送者，如圖 2- 11、圖 2- 12 所示。

```
public boolean handleMessage(Message msg){
    if (msg.sameKind("1")){
            //do something
            sendReply(someReply);          表示需要回傳結果
            return true;
    }else if (msg.sameKind("2")){
            //do something
            return true;
    }else if (msg.sameKind("3")){
            //do something
            return true;
    }
    return false;
}
```

圖 2- 11 循序式訊息處理程式碼

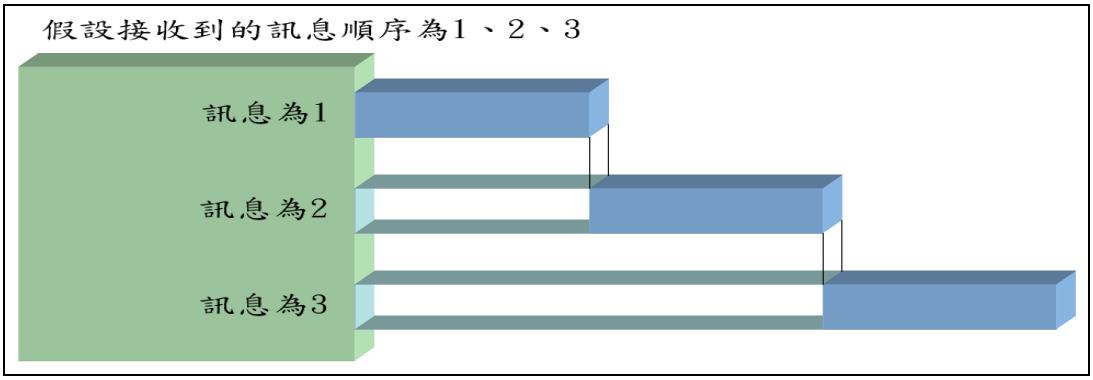

圖 2- 12 循序式訊息處理示意圖

ii. 並行式訊息處理(Parallel Message Handle)：循序式訊息處理機制一次只能處理一個訊息，當軟體代理人常接收很多訊息時，將產生處理效率不佳的問題；此時可將 Message Handler 設為非同步函式，則每個訊息都會開啟一個執行緒去處理，此稱為「並行式訊息處理(Parallel Message Handle)」，以 IBM Aglets 為例，MessageHandle 的函式內可能有多種不同的訊息處理路徑，則透過一個 exitMonitor 的指令，釋放出 lock 讓軟體代理人在執行這個訊息處理路徑，同時可以處理另一個訊息，如圖 2- 13、圖 2- 14 所示。

```
public boolean handleMessage(Message msg){
    if (msg.sameKind("1")){
            //do something
            return true;
    }else if (msg.sameKind("2")){
            //start
            getMessageManager().exitMonitor();
            //continue
            return true;
    }else if (msg.sameKind("3")){
            //do something
            return true;
    }
    return false;
}
```

表示此訊息可以與其他訊息並行處理

圖 2- 13 並行式訊息處理程式碼

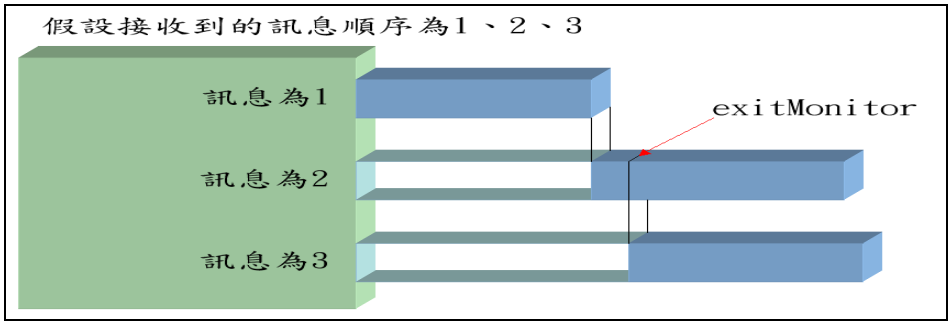

圖 2- 14 並行式訊息處理程式碼

依據上述傳送與接收訊息不同的處理方式，產生六種可能的訊息傳遞與處理方式。

綜合上面所述，可歸納出兩種軟體代理人的合作模式：(1)供需關係(Master-Slave)，軟體代理人能夠產生另一軟體代理人，並委派部分的工作由其代為執行並回報結果，來達到系統的目標。(2)供需關係(Provider-Requestor)，兩個獨立的軟體代理人，透過訊息傳遞的方式，彼此合作來達到系統的目標。則設計者透過這兩種合作模式組合來建構多軟體代理人系統的架構，讓系統中的軟體代理人發揮合作能力。

2.1.2.2.行動力(Mobility)

軟體代理人之行動力(Mobility)特性具有下列優點而被廣泛應用[3]：

1. 減少訊息傳遞的成本：軟體代理人可以直接移動到指定的地方利用當地的資源做運算，而不用像傳統的 Client-Server 架構，必須藉由一次次的訊息交換才能決定下個執行的步驟，可以節省網路頻寬的耗費並且提高執行的效率。
2. 資源利用最佳化：如果本地端的機器負荷過重時，可派出軟體代理人將本地端的部分工作轉移到遠端其他資源較多的機器去執行，如此資源的運用可以達到平均化。
3. 簡化工作統整的程序：所有軟體代理人可以分散到各地去進行任務，而藉由彼此的交換訊息與合作，再將有用的資料帶回來做運用，除了可以減少不必要的資料傳輸造成網路頻寬的浪費外，也省了很多利用近端機器處理資料的時間與空間之浪費。

軟體代理人能夠遷移的關鍵在於讓一個軟體代理人程式暫時停止執行，並將程式的狀態儲存起來，等到軟體程式遷移到另一台主機上後，被重新起動時，就可以將之前的程式狀態再度載入，接續上次暫停時的狀態繼續執行。程式執行時

的狀態分為兩個部分:資料狀態(Data State)以及執行狀態(Execution State)，資料狀態(Data State)指的是程式內各個全域變數所儲存的資料，例如在物件導向程式中的類別內部可以宣告變數，這些變數儲存的資料會隨著程式的執行而有變動。執行狀態(Execution State)指的是程式在執行時期儲存在快取記憶體(cache)以及中央處理器的暫存器(Register)的值，利用前面所提到物件序列化機制和 Reflection 機制，物件序列化可以將系統執行時期物件內部的狀態儲存成檔案，而 Reflection 機制則可以從物件狀態檔案中讀出類別內部的變數值，因此軟體代理人平台在傳送代理人程式時能夠傳送其資料狀態，當軟體代理人到達目的地後，軟體代理人需要重新執行原本的程式碼，只有上個主機執行完後的資料狀態延續下來提供執行，這種遷移的方式我們稱為「Weak Mobility」[11]，如圖 2- 15 所示，一旦軟體代理人執行遷徙的指令，則由代理人伺服器將軟體代理人

```
public class  MobileAgent extend Aglet{
   public onCreation{
          URL destination = new URL (atp://some.host.com);
          addMobilityListener(
             new Mobility Adapter(){
                public void onDispatching(MobilityEvent e){}
                public void onArrival(MobilityEvent e){
                   run();
                }
             }
          );
   }
   public void run{
          //do task
          if(目標未達成){
             Dispatch(destination);
          }
   }
```

執行遷徙之後，軟體代理人由run開始重新執行

目標未達成執行遷徙的動作

圖 2- 15 行動力

程式和資料狀態傳送到目標地點，再由目標地點的代理人伺服器重新跑起這個軟體代理人，並呼叫軟體代理人設置的 Listener 中 onArrival 的函式和提供原先的資料狀態，根據 onArrival 中的描述在目標地點開始執行工作，以圖 2- 15 為例，則軟體代理人抵達目標地點後，會重新再執行 run 的函式。

2.1.2.3. 學習能力(Learning)

軟體代理人學習能力(Learning)的定義是：軟體代理人可以學習到新的知識，並且能夠運用知識增加處理的速度，所以軟體代理人有沒有學習力特性，可從軟體代理人是否具有知識庫(Knowledge Base)且對知識庫有儲存或修改的動作來判斷，而知識庫可以分成下列兩種：

1. Rule Base：用來放置代理人的行為規則(Action Rule)的知識庫，軟體代理人由知識庫中的行為則來決定何種情況要執行對應的動作，如圖 2-16 所示，當具有學習能力時，軟體代理人可增加新的行為規則，因此可處理更多的情況或提供更多的服務，或是修改錯誤的行為規則，當軟體代理人再碰到相同情況時，並可執行正確的處理方式。

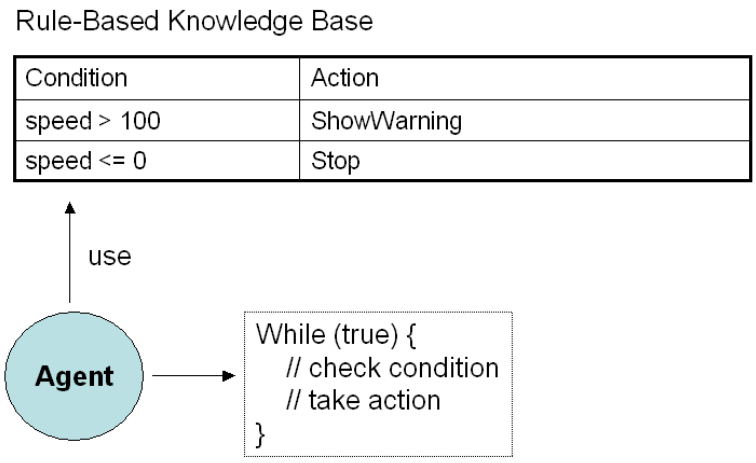

圖 2-16 Rule-Based 知識庫的形式

2. Data Base：這類型的知識庫單純用來儲存對軟體代理人執行有幫助的資料，當具有學習能力時，軟體代理人增加新資料或修改原先資料，可藉此改變執行的流程，選取最快的處理程序。

2.1.3. 與傳統物件導向程式的比較

由前面介紹可知，軟體代理人利用其特性所開發的應用程式，由下列五個方

向來跟傳統物件導向的應用程式來作比較[3]：

1. 特性：傳統物件導向的應用程式，是以功能為導向(Function-oriented)，根據功能來分割程式的元件，每個元件負責一項功能；軟體代理人的應用程式則是以目標為導向(Goal-oriented)，根據目標來決定軟體代理人應該具有哪些特性，以利用這些特性達成目標。
2. 反應能力：傳統物件導向的應用程式，必須被動等待使用者才會去執行工作，或被動等待使用者的輸入所需資訊才能完成工作；軟體代理人的應用程式，則可主動去找尋需要幫助的使用者，或利用行動力主動去找尋可以解決問題的資訊以完成目標。
3. 時間利用：傳統物件導向的應用程式，只能完成使用者先前已定義好的任務，所以即使任務分工，也會被整體中執行速度最慢的程式所拖延；軟體代理人的應用程式，透過合作能力，自主地將任務動態地分散給其它軟體代理人，或由多個代理人一起完成目標，可以節省時間並且平均工作的分擔。
4. 資源利用：傳統物件導向的應用程式，受事先所定義好的程式規範限制，只能依照使用者的認知來利用資源，程式無法自己動態更改，達到資源利用的最佳化；軟體代理人的應用程式，透過行動力，軟體代理人可遷移到別台機器上去使用該機器的資源和設備。
5. 架構彈性：傳統物件導向的應用程式，缺乏彈性，無法自動調整錯誤的執行方法，必須有使用者的介入；軟體代理人的應用程式，透過學習能力，能夠依據過去執行任務的經驗來解決近似或相同的任務。

因此軟體代理人的應用程式不論是執行的時間上、系統的彈性與資源的利用都較傳統應用程式佔優勢，且軟體代理人比傳統應用程式更有彈性、主動性以及更能貼近使用者的想法，所以軟體代理人將比傳統應用程式更適合用在資料收集、資料過濾、資訊管理、電子商務、工作流程管理、網路管理、服務管理…等應用領域之上，不但可以讓整個系統變得更強固、更具彈性之外，也能讓使用者

用最簡單的方式獲得他所想要的結果，所以可以歸納出如表 2- 1 的結果。

	傳統程式	軟體代理人
特性	以功能為導向 (Function-oriented)	以目標為導向 (Goal-oriented)
反應能力	被動式	主動式
時間利用	只能完成使用者先前已定義好的任務，所以即使任務分工，也會被整體中執行速度最慢的程式所拖延	透過合作能力，可以自主地將任務動態地分散給其它軟體代理人，或由多個代理人一起完成目標，可以節省時間並且平均工作的分擔
資源利用	受事先所定義好的程式規範限制，只能依照使用者的認知來利用資源，程式無法自己動態更改，達到資源利用的最佳化	透過行動力，可以遷移到別台機器上去使用該機器的資源和設備
架構彈性	缺乏彈性，無法自動調整錯誤的執行方法，必須有使用者的介入	透過學習能力，能夠依據過去執行任務的經驗來解決近似或相同的任務

表 2- 1 傳統程式與軟體代理人程式之比較表

2.2.相關研究

現今對軟體代理人程式測試研究並不多，且大部分的研究皆侷限於測試環境的開發，而沒有一套完整而有效的測試方法，雖然軟體代理人的應用越來越廣，但是面臨無法有效去測試這些開發的軟體代理人系統，導致現今軟體代理人系統的開發仍受到侷限，目前為止尚未有一套有效的軟體代理人的測試方法。在 2.2.1 節將介紹目前兩個較為完整的軟體代理人測試研究，並在 2.2.2 節中在提出現今軟體代理人程式測試的研究的缺失。

2.2.1. 軟體代理人程式測試研究之分析介紹

1. Test Agent[4]中首先提出一些軟體代理人系統測試的挑戰：
   i. 軟體代理人間的溝通主要是透過訊息傳遞而非函式呼叫，因此傳統的元件測試技術無法適用。
   ii. 軟體代理人是自動化且並行執行的，因此可能本身執行正確但是溝通出錯，反之亦然。
   iii. 訊息的描述通常會不正確或不完整，所以訊息的形象化是必須的。
   iv. 軟體代理人可能具有學習能力，所以成功的測試必須對於相同的測試資料會產生不同的結果。

Test Agent 提供了一套滾動式(Incrementally)軟體代理人系統的測試方法，首先針對每個單一的軟體代理人作測試，如圖 2-17 所示，軟體代理

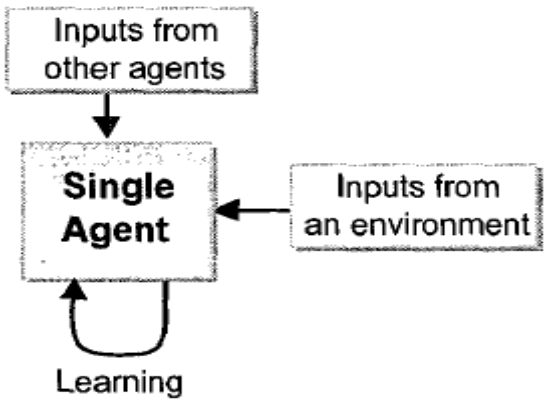

圖 2-17 測試單一的軟體代理人

人本身的功能必須透過外部的事件來觸發，這些事件可能由外部的環境、其他的軟體代理人或學習產生，首先針對來自其他軟體代理人的輸入，解決的方法是由使用者來設計一些假軟體代理人(Dummy Agent)來提供適當的輸入來測試軟體代理人，並且定義一些在開發軟體代理人時可能發生的錯誤：

i. 編寫錯誤的軟體代理人位址，以致於無法訊息傳遞。

ii. 訊息中塞入錯誤的要求，以致於接收的軟體代理人無法完成工作。

iii. 錯誤的剖析接收的訊息。

iv. 對於收到的訊息執行錯誤的處理動作。

v. 將訊息送給錯誤的軟體代理人。

vi. 沒有針對所有應該支援的訊息作對應處理的程式碼。

針對這些可能發生的錯誤來作測試；而來自外部的環境的輸入，則考量軟體代理人執行環境是簡單還是複雜，如果是簡單則直接使用執行環境來測試，反之，則必須利用模擬器(Simulator)來模擬執行的環境作測試；最後關於軟體代理人的學習能力，利用事先決定的資料餵給軟體代理人來觀察是否有獲得預期的結果，如果有學習的話，則給予一組輸入資料，便會產生新學習的行為，當遇到關鍵的情況或是無法簡單監視軟體代理人時，需要利用正規的方法作更嚴格的測試。

接下來測試軟體代理人間的社群，必須確認所有軟體代理人間的溝通工作是否與設計時相同，包含：從正確的軟體代理人接收到正確的訊息，並作正確的回應，以及與環境間的互動正確。因為軟體代理人間的社群通常是由一群程式設計師來開發，可能會產生下列幾種錯誤：

i. 軟體代理人的互動沒有適當的文件說明，導致不同的開發者開發不同的互動方式。

ii. 在訊息交換的設計上產生死結的情況。

iii. 因為傳送訊息的指令或指令的內容不正確導致錯誤的溝通。

iv. 無法一次完成所有軟體代理人的訊息之改變。

然而越大型的軟體代理人社群越難以去測試,所以能夠提供給開發者的幫助有兩個方面：

i. 可以將訊息的傳遞形象化，並且可以被重複的播放。

ii. 集中控制包含能夠開始、暫停，甚至控制軟體代理人執行的速度，讓所有軟體代理人的動作以及溝通可以被監視。

這些部分都是透過一個能夠測試並監視的 Test Agent 來完成，Test Agent 必須要能與開發者互動，藉由讓開發者觀察軟體代理人間的互動來測試、除錯，並加速開發，Test Agent 的架構如圖 2-18 所示，Test Agent 接收來自使用者或系統的規格中訊息的規格，並將它存成訊息規格的ＸＭＬ檔案，方便以後再使用，而根據 test script 來安排測試每一個軟體代理人，Test Agent 要測試軟體代理人與其間的溝通，包含：

i. 測試軟體代理人傳送與接收一個規定的訊息的能力。

ii. 測試軟體代理人可以正確處理有效跟無效的訊息。

iii. 測試社群處理所有定義的訊息之能力，並將無效的訊息給予編號註記。

iv. 維護軟體代理人社群的正式訊息規格。

v. 維護設計文件中的訊息規格。

vi. 蒐集相關的度量值(metric)包含：網路的使用、軟體代理人間的溝通等其他相關的度量值。

vii. 監視軟體代理人系統是否有潛在的錯誤和效能的問題。

Test Agent 支援兩種軟體代理人訊息處理能力的測試方式:Regression 和 Progression,Regression 測試使用在確任指定規格的軟體代理人的執行行為，且對軟體代理人的修改不會影響已經存在的訊息處理能力。而 Progression 測試則是當增加新的訊息處理能力時，支援軟體代理人開發者作開發。

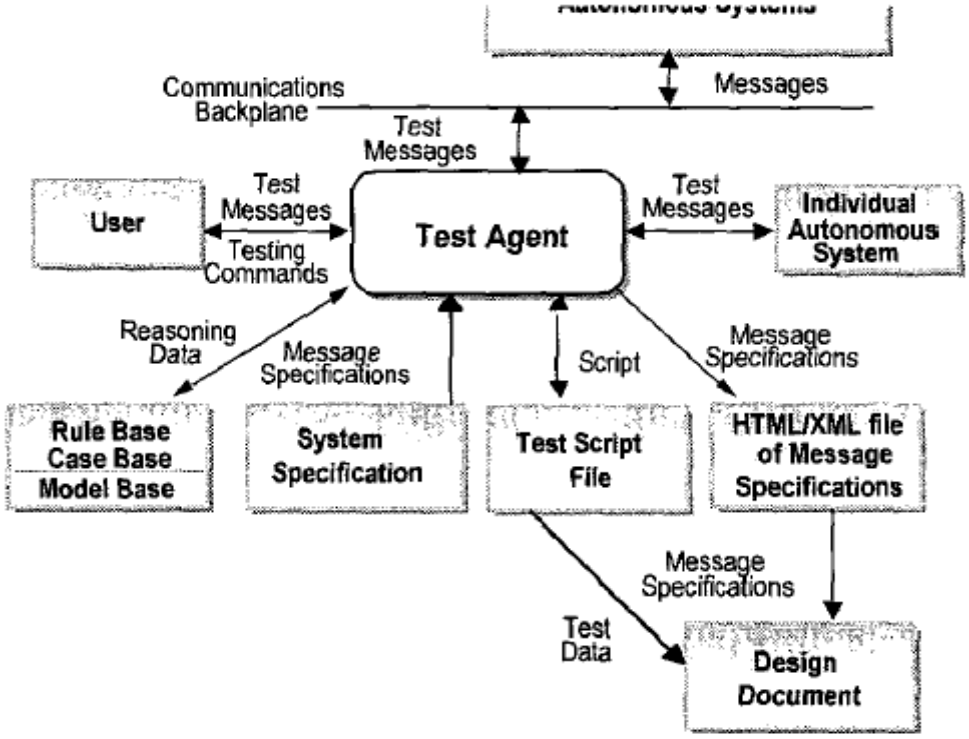

圖 2- 18 Test Agent 的架構

所以整體來說，使用者提供 Test Agent 欲測試的軟體代理人的相關資料，包含訊息規格等，由 Test Agent 代為執行測試的動作並蒐集相關的資料提供使用者判斷軟體代理人是否有錯誤，而測試的方法為先測試單一軟體代理人的功能，在測試整個軟體代理人社群內所有的溝通的情況。

2. JAMES[5](JAVA Based agent modeling environment for simulation)

針對 Mole 的軟體代理人程式語言，提供測試軟體代理人系統一個虛擬環境的測試平台，同時在虛擬的環境中可以設定一個 Representative 來表示欲測試的軟體代理人，並建立兩者間的關聯，利用 Representative 來讓使用者作軟體代理人的測試，因此 JAMES 必須具有兩個特性：

i. 具有讓測試的軟體代理人如同設定的 Representative 一樣地執行，且隨意切換實際的執行環境與虛擬的測試環境。

ii. 軟體代理人是測試設定不可或缺的一部分，軟體代理人應該被了解且可以控制的，而不是如同黑箱般的與測試環境互動。

JAMES 利用”Plug & Test”的方式來執行軟體代理人的測試，一個欲測試的軟體代理人，必須要在 JAMES 中設定一個 Representative，如圖 2-19 所示，Representative 被描述成一個 time-triggered 自動程式，稱為 atomic model，包含 Heart beat 負責控制時間依據設定來觸發事件讓軟體代理人執行，以及提供對應事件的函式，Satellites 則提供一個抽象的觀點來描述軟體代理人的狀態跟行為，如圖 2-20 所示，並且提供 Peripheral port 讓 Representative 跟軟體代理人可互動的連結，藉由輸入的 port 來根據 Satellites 流程送出觸發的事件來讓軟體代理人產生對應的測試動作，而輸出的 port 來得到執行之後的結果，以讓使用者判斷是否執行正確。

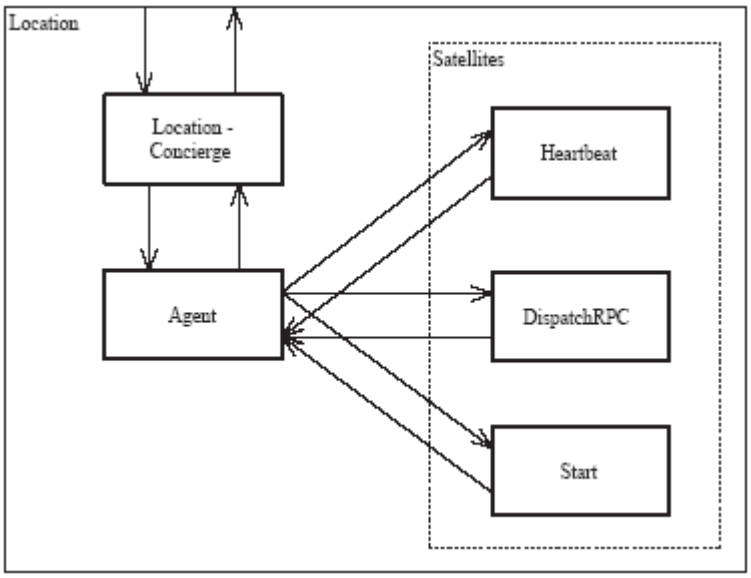

圖 2-19 軟體代理人與 Representative 的互動

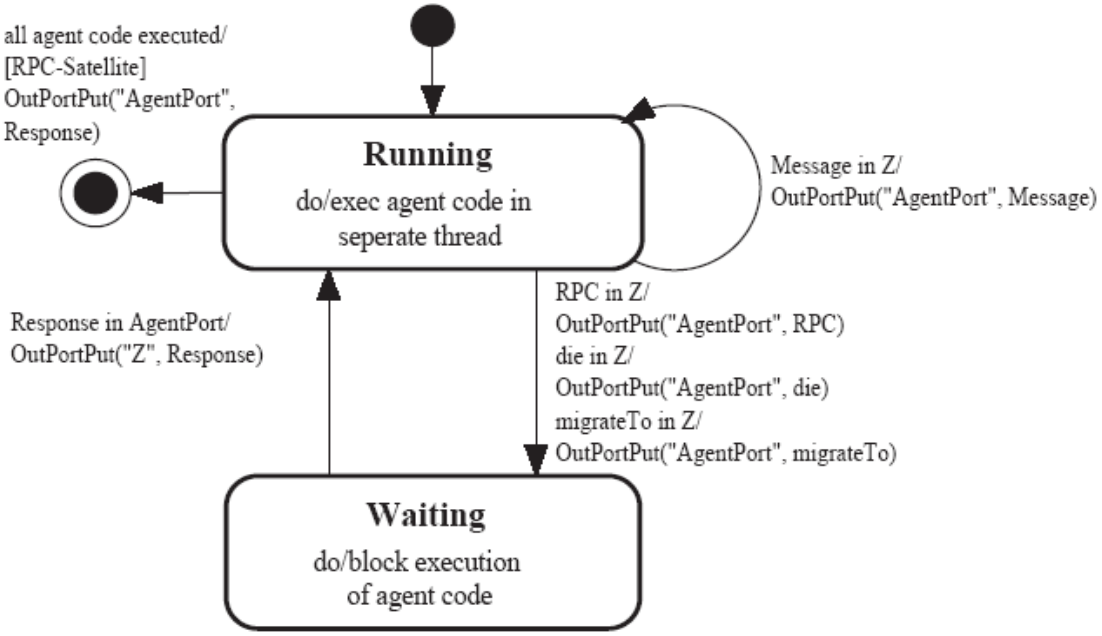

圖 2-20 Satellite

2.2.2. 現今軟體代理人程式測試的缺失

前面已經介紹以及分析了兩個現行的軟體代理人的測試研究，可以得知現今軟體代理人測試有下列的缺失：

1. 大部分的研究都著重在測試平台的研究，如：JAMES，或只針對小部分的特性來提出測試方法，如：Test Agent 只針對訊息傳遞和學習能力，而缺乏一套系統化的軟體代理人測試方法。

2. 缺乏對軟體代理人間的合作關係的研究，在軟體代理人系統中，所有軟體代理人間的關係並非單純只是溝通，而是為了合作完成某個目標而設計的，因此軟體代理人間的合作行為勢必會影響軟體代理人的測試方法。

3. 對於合作能力與學習能力的測試並不完整，在 Test Agent 中合作能力只提出訊息傳遞，但實際上軟體代理人的合作能力不單只有訊息傳遞；而對於學習能力的測試方法，Test Agent 中只將其放在未來的工作中並未提出真正的測試方法。

4. 沒有針對行動力的特性提出測試的方法，行動力的特性對於軟體代理人應用非常重要，因此軟體代理人的測試方法應將考量行動力的特性列入考量。

5. 所有的軟體代理人的測試系統都屬於被動性，即必須透過使用者自己來設計 test case、安排測試順序等，現有的軟體代理人系統只負責被動的執行部分。

2.3.軟體代理人程式測試的改進

由前一節所提到關於現今的軟體代理人介紹、分析，並提出它們的缺失，得知現今軟體代理人程式測試的研究中，最缺乏的便是一套系統化的軟體代理人程式的測試方法，則現今軟體代理人程式的測試應由下列幾點來作改進方向：

1. 建構出完整的軟體代理人間的關係，並由關係中提出正確並有效率的軟體代理人的測試順序。
2. 針對單一軟體代理人，根據特性間關係，建構一套滾動方式(Incremental approach)的特性順序，讓單一軟體代理人的測試更加簡單且有效率。
3. 針對軟體代理人具有的特性提出更完整且有效率的測試方法，包含：自動化、合作能力、行動力、學習能力。
4. 建構一套軟體代理人測試系統，利用提出的軟體代理人系統的測試方法，主動地協助測試者由軟體代理人系統的程式碼中找出軟體代理人間的關係並安排測試順序，且根據特性安排所有單一軟體代理人的可能的測試路徑。

根據以上這些軟體代理人程式測試方法的需求，必須由軟體代理人系統的行為去分析，來歸納出能達到上述需求的軟體代理人測試方法，並架構一個能夠支援這套軟體代理人測試方法的測試系統。

第3章　軟體代理人系統的分析與測試策略

軟體代理人具有自動化、學習能力、合作能力與行動力之特性，一般物件導向軟體的測試方法並未考慮到這些特性。本章針對軟體代理人的特性來分析現有測試方法的適用性，進而提出適用軟體代理人特性之測試方式，首先在 3.1 節中從軟體代理人系統的架構來探討軟體代理人間的關係並依這些關係對測試順序的影響，進而在 3.2 節，針對軟體代理人的特性提出測試策略。

3.1.軟體代理人系統的執行行為之分析

軟體代理人系統設計時，設計者首先會先訂定這套系統之目標(Goal)，依照這個目標來決定軟體代理人系統的架構，如果系統的目標單純且簡單時，設計者只需要設計一個軟體代理人即可達成系統的目標；如果系統的目標複雜，則可由數個軟體代理人合作達成系統的目標，因此可先分析單一軟體代理人之執行行為，再分析多軟體代理人系統之執行行為。

3.1.1.　單一軟體代理人系統執行行為分析

在單一軟體代理人系統中不可能發生與其他軟體代理人有互動合作行為，因此不具有合作能力的特性，其執行行為分析如下：

1. 如果軟體代理人只具有自動化特性，則此軟體代理人只能在固定的執行環境下執行它的任務，所需的資料來源可以來自使用者、軟體代理人自己的知識庫及本地端的資源，軟體代理人只能利用本身之執行邏輯來達成系統的目標，並將執行的結果回報給使用者，由第二章之介紹可知，其程式架構與一般物件導向的軟體程式相同，可直接引用一般物件導向軟體的測試方法。
2. 當軟體代理人執行任務所需的資料必須要在遠端的機器取得或者是必須要提供遠端的使用者服務時，軟體代理人必須具有行動力的特性才能

夠達成任務，具有行動力的軟體代理人有兩種可能完成目標之行為模式：

1. Goal-Oriented：軟體代理人在一執行環境執行後，依據是否已達成目標來決定是否要執行遷徙(遷徙)的動作，執行的步驟如下：

   i. 軟體代理人在起始時，會有兩種不同的情況產生：

   - 如果資料本身就不在本地端或是要提供服務的對象不會在本地端時，在軟體代理人起始時，就會執行一次遷徙的動作，遷徙到第一執行的位置。
   - 反之，則軟體代理人會先在本地端執行任務，並判斷目標是否完成，如果完成，則回報結果給使用者；如果沒有完成，則遷徙到下一個執行地點。

   ii. 當軟體代理人需要遷徙時，軟體代理人要先取得設計者編排好的 itinerary，依序執行遷徙的指令，讓軟體代理人取得 itinerary 的方式有三種：

   - 軟體代理人在產生時將 itinerary 當成參數輸入給軟體代理人。
   - itinerary 放入軟體代理人的知識庫，軟體代理人需要遷徙的時候，自行去知識庫取得。
   - itinerary 直接編寫在軟體代理人的程式碼中。

   iii. 取得了 itinerary 之後，就可以執行遷徙的指令，由於軟體代理人是採用 Weak Mobility 的方式來完成遷徙的動作，當軟體代理人遷徙到目標 destination 後，先回復之前的資料狀態繼續執行。

   iv. 軟體代理人一旦達成目標之後，就不需要再執行遷徙的動作；否則，繼續遷徙到下一個 destination，直到目標達成或是 itinerary 中沒有其他 destination 可以遷徙為止。

2. Destination-Oriented：軟體代理人在 itinerary 中每個 destination 上的工作都為軟體代理人的目標的部分目標(Partial-Goal)，軟體代理人必須依 itinerary 指定的 destination 去執行所有工作後，才能判斷目標是否達成，因此在執行的步驟上是先根據 itinerary 中是否還有 destination 來決定是否要執行遷徙動作，等到所有的 destination 都執行完成後，軟體代理人才能夠判斷是否達成目標。

因此 Goal-Oriented 和 Destination-Oriented 的差別只在於軟體代理人是否要將所有的 destination 都走完，無論軟體代理人怎麼遷徙，它所負責達成目標的部分(自動化)並不會因此而有所改變，雖然 destination 不同但是提供軟體代理人執行的環境都是固定的。

因為軟體代理人採用的是 Weak Mobility 的方式，執行遷徙的動作之後，軟體代理人需要重新執行原本的程式碼，而非接續原本執行路徑的下一個步驟，因此所有執行的遷徙的動作，必須在軟體代理人程式執行路徑的最後一個步驟，否則永遠不會執行到遷徙動作後面的步驟。

如果不在本地端先執行就遷徙的情況，則起始的函式必須有遷徙的動作；當 run 函式執行後還需判斷是否遷徙，因此仍需在 run 函式執行路徑的最後一個步驟必須為遷徙動作。

從上述分析可知，自動化跟行動力兩者間的關聯在於資料或資源的取得，當要測試具有自動化和行動力的軟體代理人時，只要可以先提供足夠的資料或資源，就可以先測試自動化的特性是否正確，然後再測試行動力的特性是否正確，接下來證明自動化與行動力分開測試，還是可以正確的測試軟體代理人：

由**圖 3-1** 來說明，具有行動力的軟體代理人的行為如下，軟體代理人在起始的位置執行一些工作後，判斷是否需要遷徙，需要則遷徙到下一個

destination，繼續執行工作，直到不需要遷徙為止，如果是先遷徙才執行的 run 函式時，context1 的工作為軟體代理人的起始動作，所以依然成立。

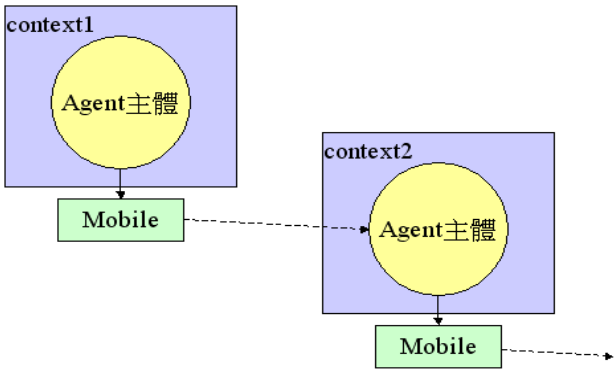

圖 3- 1 尚未開始執行

如果我們能夠證明軟體代理人的自動化在所有執行路徑都正確，則表示如圖 3- 2 虛線框起所表示的情況結果是對的。

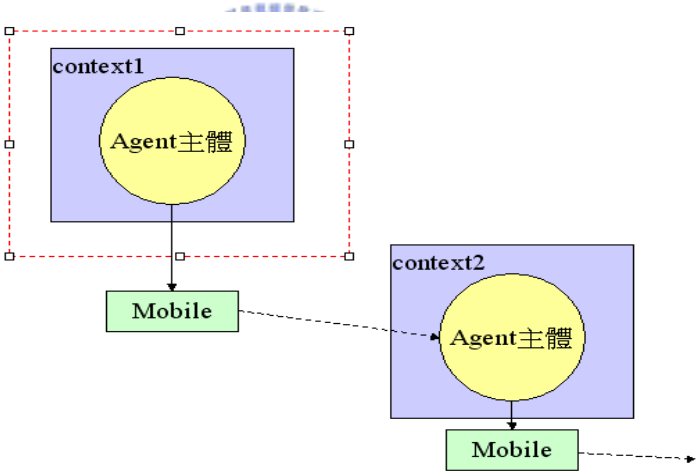

圖 3- 2 執行完 context1 工作

當產生遷徙至 context2 時，如圖 3- 3 這個虛線框起所表示，

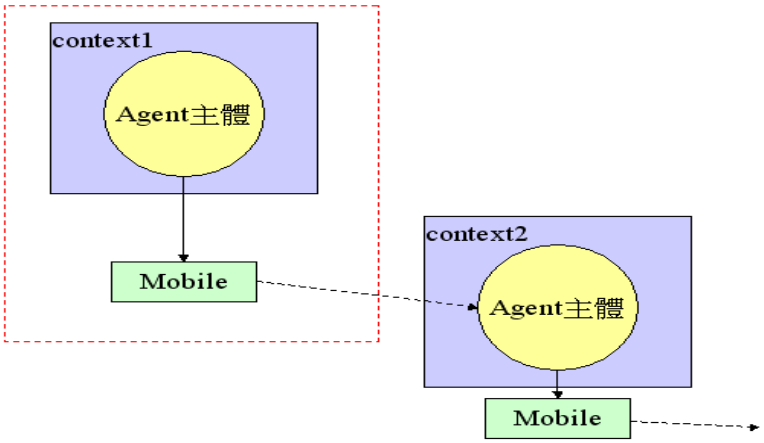

圖 3- 3 執行遷徙到 context2 的動作

在 context1 已經測試了所有可能的執行情況都正確，由於執行路徑均相同，因此可推論在 context2 上自動化的執行也是對的，則如圖 3- 4 這個虛線框起所表示的情況

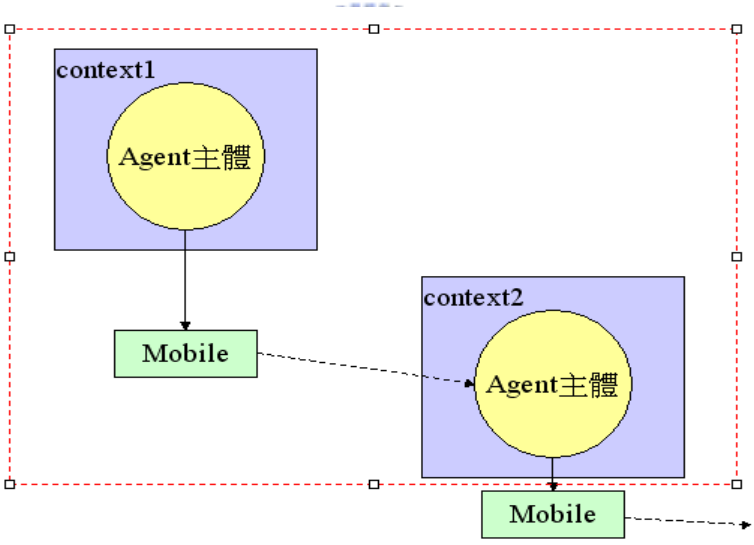

圖 3- 4 執行完 context2 工作

同樣的我們已經測試了所有可產生遷徙的情況，遷徙的動作都是對的，則可推論 context2 的執行完後作遷徙的動作也會是對了，則依此類推下去，我們可以利用分開測試自動化跟行動力來驗證，軟體代理人是否是正確的。

這樣分開測試的好處為(i)可以避免重複測試相同的執行路徑，以減輕測試者的負擔，(ii)只需要在一個測試平臺即可測試自動化的所有行為。

3. 當軟體代理人透過行動力的遷徙來幫助自己達成目標後，軟體代理人可以得到新的資料或是經驗，為了下次再碰到相同或類似的情況時，可以選取最快的的處理方式，因此軟體代理人必須具有學習能力，要具有學習能力則必須要擁有知識庫，而學習能力就是將從執行中學習某些有用的經驗增添或修正某些行為規則(Action Rule)，因此學習能力是透過寫入知識庫的動作來達成，學習能力本身對於軟體代理人的影響在於當下次軟體代理人碰到相同或類似的情況時，軟體代理人將選擇最佳執行的執行路徑，而不是增加一條執行的路徑，只要確保寫入知識庫不發生錯誤，便不會造成執行該條路徑時發生錯誤，所以在測試學習能力不需要在多測試執行路徑，只需要測試含有學習(寫入知識庫)的路徑是否正確即可，測試軟體代理人學習能力是否正確，包含取得有用的資料(自動化)，處理這些資料(自動化)，最後是寫入知識庫的動作(學習能力)，因此學習能力不會影響自動化特性的正確與否，但是自動化卻會影響學習能力的正確與否，所以測試的時候先確定自動化是否正確，再測試學習能力的動作，而學習能力的測試可以在單一的平台上測試完成，所以先測試完自動化和學習能力，再來測試行動力。

因此在單一的軟體代人時，採用滾動方式(Incremental approach)依序測試軟體代理人的自動化、學習能力以及行動力。

3.1.2. 多軟體代理人系統

當軟體代理人系統的目標及任務複雜時，可以透過多個軟體代理人的合作來共同完成系統的目標，多軟體代理人有兩種產生方式，(1)一軟體代理人會產生一個或多個子軟體代理人來分擔工作，將部份工作分給子軟體代理人，並委派或指揮這些子軟體代理人去處理，所有子軟體代理人的目標都屬於父軟體代理人的子目標(Subgoal)，當所有子軟體代理人完成自己的目標，則父軟體代理人就可

以達成其目標，如圖 3-5 所示，則稱這樣的軟體代理人間的關係為“主從關係”(Master-Slave)，產生者為 Master，而被產生者為 Slave；(2)設計者將系統的目標分割成數個獨立的子目標，一個子目標由一個軟體代理人負責，每一個軟體代理人就如同一個子系統，且彼此間會透過溝通的動作來互相合作，藉以得到所需的資料完成自己的目標，如圖 3-6 所示，則稱這樣的軟體代理人間的關係為“供需關係”(Provider-Requestor)，主動提出要求的為 Requestor，而被動提供訊息的為 Provider。

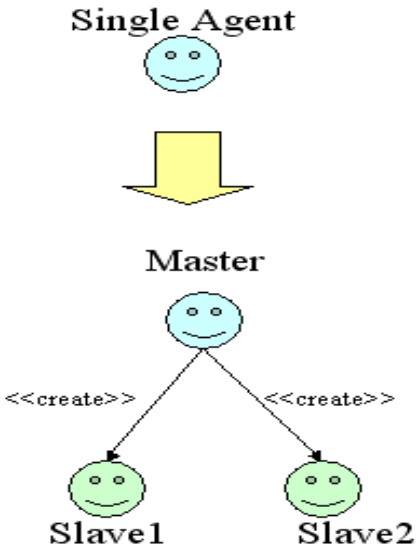

圖 3-5 主從關係

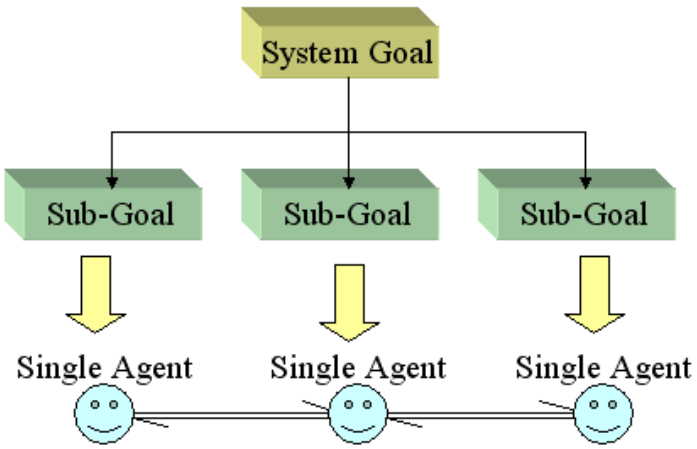

圖 3-6 供需關係

從測試的角度來看多軟體代理人系統的架構，則須先決定所有軟體代理人的

測試順序，再採合滾動方式(Incremental approach)依序測試每個軟體代理人，以減少系統測試的複雜度及提升偵錯效率;測試順序的決定取決於軟體代理人間的關係，有必要先了解軟體代理人間的關係組合各種可能的組合情況，才能擬定測試的順序，再來討論合作能力對每個軟體代理人測試的影響。

多軟體代理人系統架構的“主從關係”可能的情況，有下列幾種：

1. 如果只有兩個軟體代理人時，一個軟體代理人為 Master，而另一個軟體代理人則為 Slave，兩者間的執行步驟為：
   i. Master 產生(Create)Slave。
   ii. 如果 Master 與 Slave 均在同一個代理人伺服器上工作，則 Master 透過訊息傳遞(Communication)的方式指揮 Slave 工作，如果不在同一代理人伺服器工作的話，則 Master 透過發送(Dispatch)的動作將 Slave 送到指定的地點執行任務。
   iii. Slave 在執行任務過程中發生問題時，透過訊息傳遞的方式與 Master 溝通。
   iv. Slave 完成它的工作後，需透過訊息傳遞的方式回報結果給 Master。
   v. Master 根據 Slave 回報的結果來達成自己的目標。

在這種情況中，因為 Slave 的目標為 Master 的子目標，所以我們必須先測試 Slave 是否能正確完成目標，才能夠測試 Master 是否能夠正確完成目標，從而證實系統的目標可以被正確的達成。

2. 如果有三個以上的軟體代理人時，有下列兩種情況：
   vi. 一個 Master 產生多個 Slave 來協助它的完成目標，如圖 3-5 所示，首先我們必須先考量一個問題，這些 Slave 是否皆來自同一份程式碼，如果是的話我們將這些 Slave 視為是同一個類型(Type)的軟體代理人，相同類型的

Slave 具有相同的行為模式，因此相同類型的 Slave 測試時只需要測試一次即可，而所有的 Slave 都是 Master 的一個子目標，所以所有的 Slave 都測試過後，才測試 Master。而 Slave 間的測試順序，如果彼此間沒有供需關係，則可任意決定所有 Slave 的測試順序，如果彼此間有供需關係的話，先測試沒有供需關係的 Slave，再根據其他 Slave 間的供需關係來決定測試的順序，順序決定的方式在後面討論供需關係時再說明。

vii. 根據 i 的情況，當每個 Slave 無法由單一個軟體代理人來完成目標時，會另外產生數個 Slave 來協助它完成目標，因此形成多層的主從關係的樹狀架構，圖 3-7 所示，這種情況下，Slave1 和 Slave2

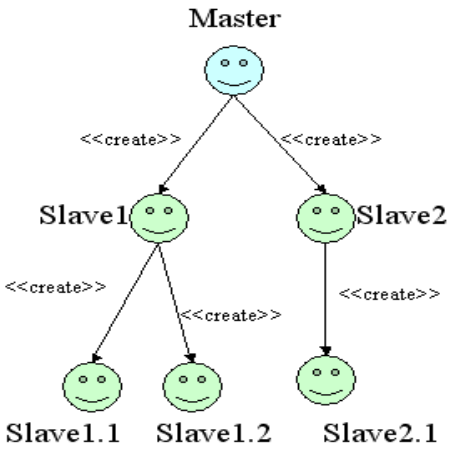

圖 3-7 多層的主從關係

都為 Master 的子目標，而 Slave1.1 和 Slave1.2 為 Slave1 的子目標，Slave2.1 為 Slave2 的子目標，子目標必須先測，才能測主目標，依序遞迴測試，若兩個個體間存在供需關係，則透過供需關係來決定個體間的測試順序，順序決定的方式在後面討論供需關係時再說明。

多軟體代理人系統架構的“供需關係”可能的情況,可分二個軟體代理人間的有供需關係或三個以上軟體代理人有供需關係來討論：

1. 如果只有兩個軟體代理人時，可能的情況有兩種：

   - 兩個軟體代理人間只有單向的供需關係，則一個軟體代理人為 Provider，而另一個軟體代理人則為 Requestor，兩者間的執行模式又可為兩種：

     - 當有一方(Requestor)需要另一方(Provider)的資料或服務時，如：Requestor 跟 Provider 要求 Provider 知識庫的 rule 來完成任務，執行步驟為：

       a Requestor 必須透過訊息傳遞的方式送出訊息給 Provider。

       b Provider 必須等到處理完自己的工作後，才能處理接收到的訊息。

       c Provider 接收到訊息後，會根據訊息選取正確的處理方式，處理完之後回傳結果給 Requestor。

     - 當有一方(Requestor)要求另一方(Provider)執行某個任務，執行步驟為：

       a Requestor 必須透過訊息傳遞的方式送出訊息給 Provider。

       b Provider 必須等到處理完自己的工作後，才能處理接收到的訊息。

       c Provider 接收到訊息後，會根據訊息選取正確的處理方式，不須回傳結果給 Requestor。

從上述可以發現，兩種行為的模式差別在於 Provider 是否要回傳結果給 Requestor，因此可以視為 Requestor 將一部分的工作交由 Provider 來處理，則測試時必須先驗證 Provider 是否正確，然後才能夠驗證 Requestor 是否正確。

- 如果兩者間具有雙向的供需關係時，可用圖 3-8 來說明，實線表示

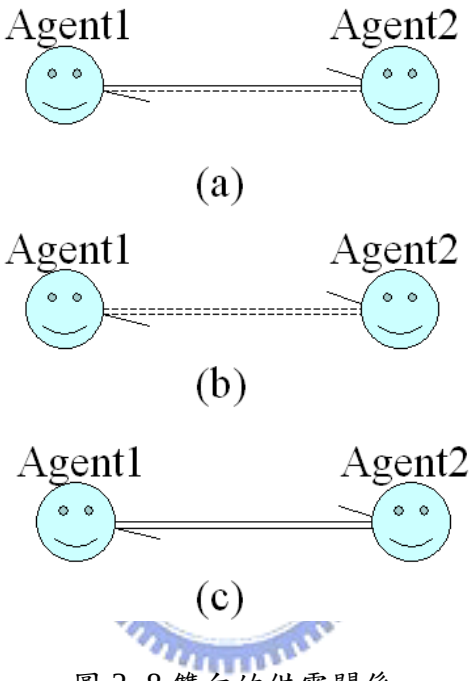

圖 3-8 雙向的供需關係

要回傳結果的供需關係，虛線表示不需要回傳結果的供需關係，在圖 3-8(a)，因為代理人 2 往代理人 1 的方向為不需要回傳的結果，表示這個供需關係對代理人 2 的本身執行沒有影響，所以可先測試代理人 2 再測試代理人 1。在圖 3-8(b)或(c)中，無法直接從兩邊的供需關係決定軟體代理人的測試順序，必須觀察彼此訊息傳遞的動作是否為循序式，討論如下：

- 情況(b)，軟體代理人 1 與軟體代理人 2 之循序式的訊息傳遞可分成兩種情況來討論：
  - 如圖 3-9 所示，軟體代理人 1 從 run 中主動去傳遞訊息給軟體

代理人 2，軟體代理人 2 再從 handleMessage 的函式中傳遞訊息給軟體代理人 1，而產生循序式的傳遞訊息，這種情況是由第一個傳遞訊息動作來觸發兩者的供需關係，即第一傳遞訊息動作來決定兩個軟體代理人的供需關係，軟體代理人 1 為 Requestor 而軟體代理人 2 為 Provider，因此軟體代理人 1 要比軟體代理人 2 先測試。

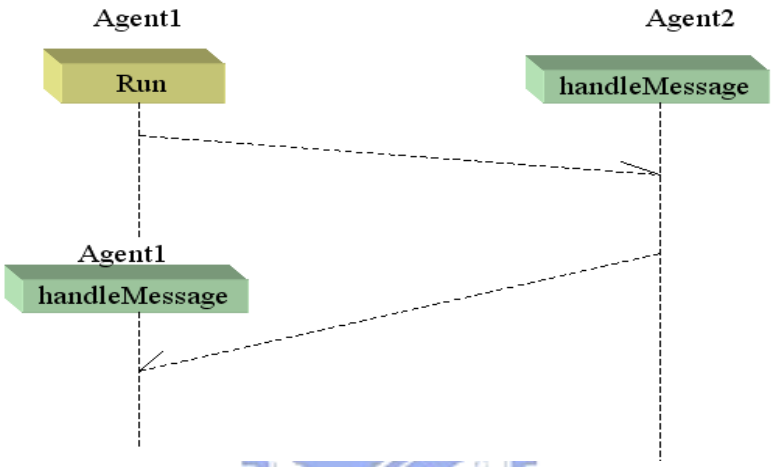

圖 3- 9 非同步的循序式訊息傳遞(A)

◆ 另外一種情況則是，兩者間的傳遞訊息的動作都在 handleMessage 的函式中，表示兩者間的傳遞訊息是被動的，另有第三方的軟體代理人觸發兩者間傳遞訊息，以圖 3- 10 為

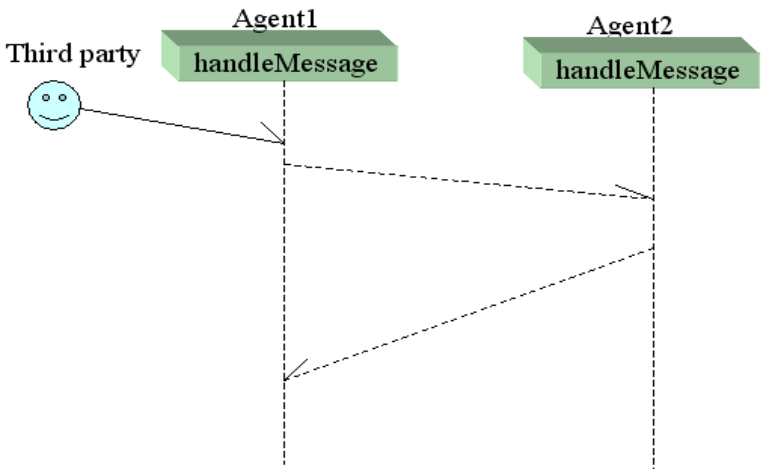

圖 3- 10 非同步的循序式訊息傳遞(B)

例，第三方觸發，軟體代理人 1 先送出傳遞訊息的動作，可視為軟體代理人的第一個傳遞訊息動作來觸發兩者的供需關係，即第一傳遞訊息動作來決定兩個軟體代理人的供需關係，軟體代理人 1 為 Requestor 而軟體代理人 2 為 Provider，因此軟體代理人 1 要比軟體代理人 2 先測試。

除此之外，皆不為循序式的訊息傳遞，則無法簡化為單向的供需關係，因此只能亂數選取一個軟體代理人出來利用 stub 先來測試它，再來測試另外一個軟體代理人。

- 情況(c)，軟體代理人 1 與軟體代理人 2 之循序式的訊息傳遞可分成兩種情況來討論：

  - 如圖 3- 11 所示，且等待軟體代理人 2 回傳結果，因此軟體代理

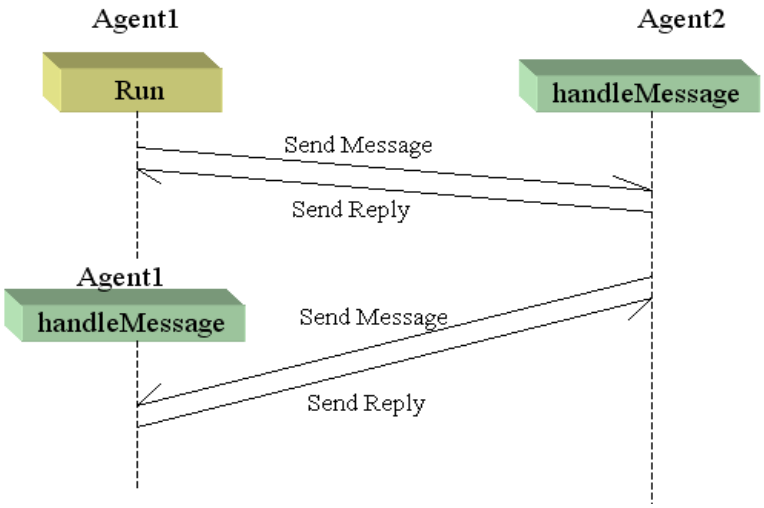

圖 3- 11 同同步的循序式訊息傳遞(A)

人 1 從 run 中主動去傳遞訊息給軟體代理人 2，軟體代理人 2 必須要先回傳結果給軟體代理人 1，才能再從 handleMessage 的函式中傳遞訊息給軟體代理人 1，而產生循序式的傳遞訊息，否則軟體代理人 1 會因為等候軟體代理人 2 的回傳，而形

成死結。此種情況，如圖 3-11 所示，是軟體代理人 1 先送出傳遞訊息的動作，可將軟體代理人 1 為 Requestor 而軟體代理人 2 為 Provider，則軟體代理人 2 要比軟體代理人 1 先測試。

- 如圖 3- 12 所示，兩者間的傳遞訊息的動作都在 handleMessage 的函式中，表示兩者間的傳遞訊息是被動的，另有一第三方的軟體代理人觸發兩者間傳遞訊息，軟體代理人 1 先送出傳遞訊息的動作，軟體代理人 2 必須要先回傳結果給軟體代理人 1。才能再從 handleMessage 的函式中傳遞訊息給軟體代理人 1，而產生循序式的傳遞訊息，因此軟體代理人 1 為 Requestor 而軟體代理人 2 為 Provider，則軟體代理人 2 要比軟體代理人 1 先測試。

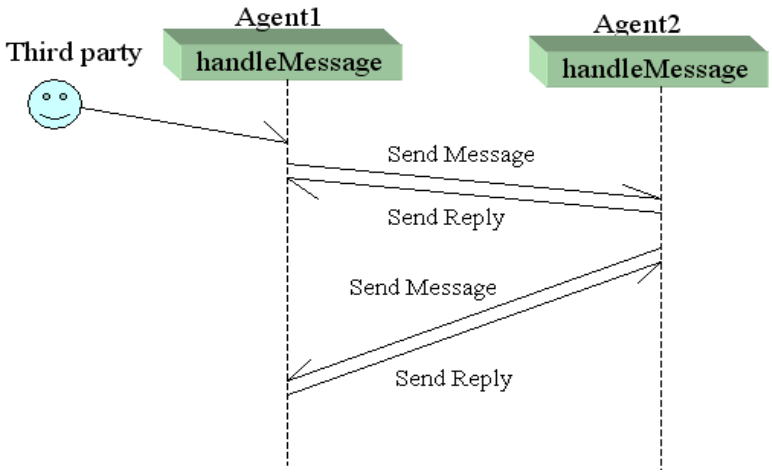

圖 3- 12 同步的循序式訊息傳遞(B)

除此之外，皆不為循序式的訊息傳遞，無法簡化為單向的供需關係，因此只能先警告測試者有可發生死結，然後亂數選取一個軟體代理人出來利用 stub 先來測試它，再來測試另外一個軟體代理人。

2. 如果有三個以上的軟體代理人時，可能有兩種不同的情況：

- 主從關係中，Slave 間有供需關係(如圖 3- 13 所示)，即 Slave1 和 Slave2 間有供需的關係。

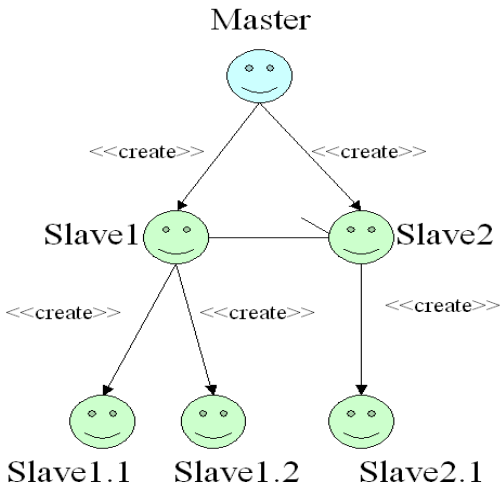

圖 3- 13 Slave 間有供需關係

- 軟體代理人系統由多個子系統組成，每個子系統可能由單一的軟體代理人或是具有主從關係的多個軟體代理人組成，如圖 3- 14 所示。

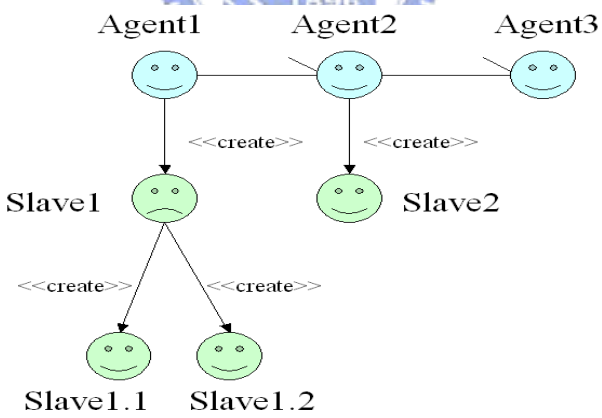

圖 3- 14 多個軟體代理子系統

如前主從關係所述，可先將具有主從關係的相關軟體代理人視為一個體，再依供需關係決定個體間的測試順序。但個體間有循環的供需關係時(如圖 3- 15 所示)，則無法由供需關係決定測試的順序，則可以先測試訊息傳遞次數最少軟體代理人，並利用 stub 測試該

軟體代理人，以減少測試者編寫 stub 的負擔；而後檢查剩下的個體是否還有循環的情況，如果還有則繼續利用相同方法選出一個軟體代理人，直到不在產生循環的供需關係為止。

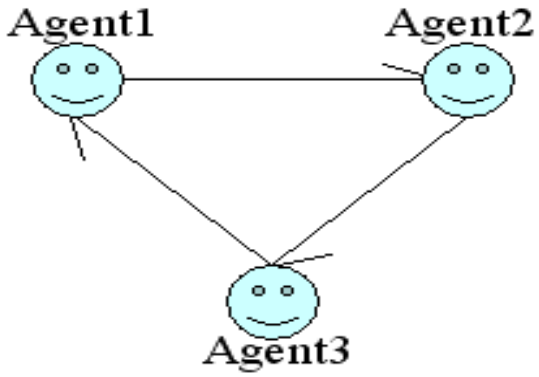

圖 3- 15 循環的供需關係

- 還有一種情況是一軟體代理人可同時對多軟體代理人傳遞訊息，我們稱這樣的動作為「群播(Multicast)」(如圖 3- 16 所示)，軟體代理人 1 群播 message 給軟體代理人 2、軟體代理人 3、軟體代理人 4，則測試的順序為先測試 Provider(即軟體代理人 2、軟體代理人 3、軟體代理人 4)再測試 Requestor(軟體代理人 1)。

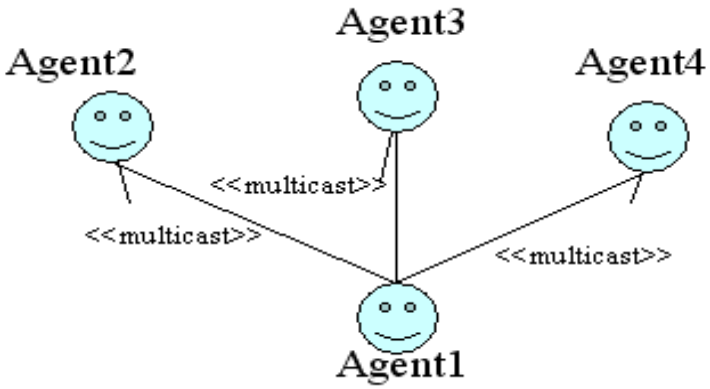

圖 3- 16 群播的供需關係

前面我們已經討論完如何從軟體代理人間關係找出每個軟體代理人的測試順序，得到測試順序之後，便要依序測試這些軟體代理人，多軟體代理人系統中

與單一軟體代理人系統的軟體代理人最大差異在於具有合作能力的特性，所以討論合作能力的測試以及對於其他特性的影響，如此便可以同樣利用同單一軟體代理人時滾動方式來測試。

1. 合作能力的測試，主要利用測試的順序和配合合作能力的動作來測試，利用順序方面所指的是，如：Master-Slave 中，測試的順序是先測試 Slave 再測試 Master，只要我們測試 Slave 正確，則便可以利用 Slave 來測試 Master 的合作能力，而 Slave 的合作能力只能先利用 stub 來測試，對於 Provider-Requestor 亦同；而配合合作能力的動作則是指，在第二章中我們提到過軟體代理人的合作能力之動作(產生、訊息傳遞、發送)，根據動作作對應的測試，來測試每個動作是否正確，因此我們可以知道要測試合作能力必須先掌握兩個關鍵：合作能力執行的對象以及執行的動作。

2. 因為只要提供正確的資料便可直接測試自動化的特性是否正確，且自動化的正確與否會影響到 Receiver 回傳的結果，因此先測試自動化，再測試合作能力。

3. 同樣因為合作能力會跟軟體代理人取得資料的方法是否正確有關，根據在單一軟體代理人中提過的，則學習能力不會影響合作能力的正確與否，但是合作能力卻會影響學習能力的正確與否，而合作能力又會跟自動化一起測試，所以測試軟體代理人的時候先確定合作能力跟自動化是否正確，再測試學習能力的動作。

4. 在執行合作能力的動作時，彼此需要互動的軟體代理人必定是處於固定的狀態，所以我們只要準備對應的軟體代理人或 stub 便可以在單一的測試平台上測試合作能力，因此可將軟體代理人合作能力與行動力分開測試。

綜合上述的結果，在測試多軟體代理人系統的軟體代理人時，會先將合作能力與自動化一起測試，然後測試學習能力，最後再測試行動力，利用這種滾動方式的測試方法來測試軟體代理人。

3.1.3. 軟體代理人間關係的建立

討論完如何從軟體代理人間的關係中來找出軟體代理人的測試順序，接下來討論如何從軟體代理人的程式碼靜態剖析出軟體代理人的關係，分別針對主從關係與供需關係來討論：

1. 主從關係：根據主從關係的行為模式可知，主從關係中 Master 必定會有產生 Slave 的動作，因此如果軟體代理人程式碼中具有產生的指令，則可以判斷該軟體代理人為 Master，而由指令內的參數可以知道為 Slave 的軟體代理人，便可建立軟體代理人間的主從關係。

2. 供需關係：供需關係需透過訊息傳遞的合作方式來完成，因此可利用訊息傳遞的動作來找出軟體代理人的供需關係，因為軟體代理人利用 Proxy 來傳遞訊息，而非直接呼叫函式的方式，所以需藉由 Proxy 來找出對應的訊息傳遞對象，下列根據軟體代理人可能取得 Proxy 的方式來討論：

   i. 如圖 3-17 所示，Master 分別產生 Slave1 和 Slave2，且 Slave1

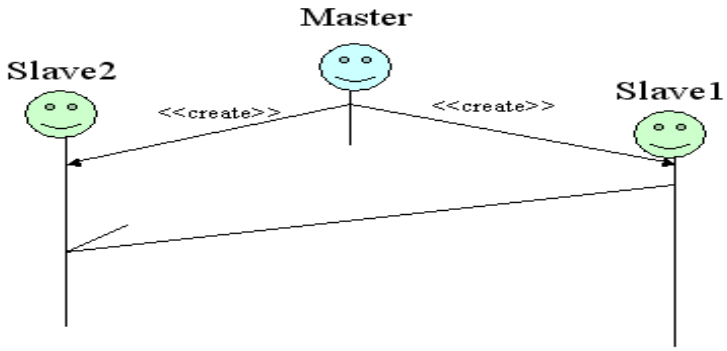

圖 3-17 由 Master 取得 Proxy

對 Slave2 作傳遞訊息的動作，我們可以 Master 產生 Slave1 的動作得知，Master 將 Slave2 的 Proxy 在產生 Slave1 時利用參數輸入給 Slave1，則建立 Slave1 跟 Slave2 的供需關係，Slave1 為 Requestor、Slave2 為 Provider。

ii. 如圖 3-18 所示，Requestor 對 Provider 作傳遞訊息的動作，

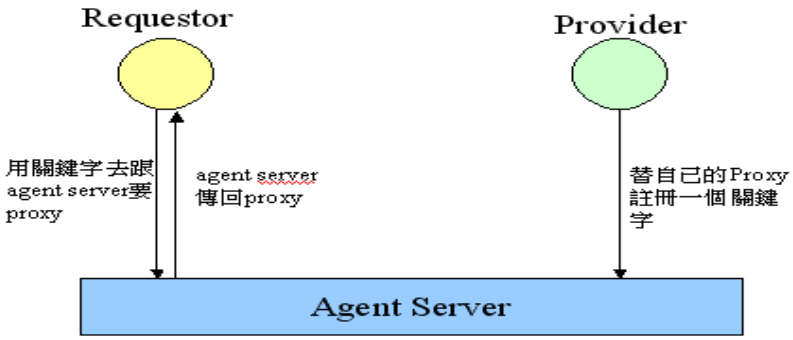

圖 3-18 利用 Key 取得 Proxy

Provider 為自己的 Proxy 跟代理人伺服器設定一個 key，Requestor 則利用這個 key 去跟代理人伺服器取得 Proxy，如果兩者的 key 皆為明碼的話，可以透過比對 key 來建立兩者間供需關係。

iii. 如圖 3-19 所示，軟體代理人 1 跟第三方要軟體代理人 2 的 Proxy

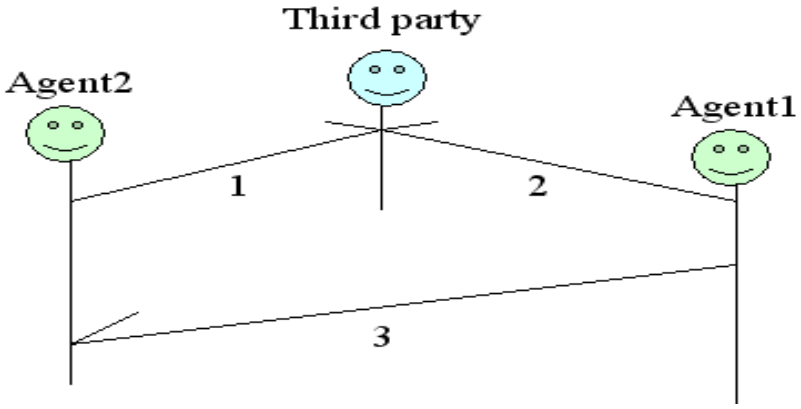

圖 3-19 跟第三方取得 Proxy

去作傳遞訊息，第三方會從跟它註冊的 Proxy 中選出正確的 Proxy 傳回給軟體代理人 1，所以軟體代理人 1 得到的 Proxy 可能為所有將自己 Proxy 傳給第三方的軟體代理人其中之一，因此建立軟體代理人 1 跟這些軟體代理人的供需關係。

iv. 如圖 3- 16 所示，軟體代理人 1 利用群播的方式來傳遞訊息，則利用彼此間約定的訊息來判斷，如果訊息是明碼，可知軟體代理人 1 散播訊息給哪些軟體代理人，建立軟體代理人 1 跟這些軟體代理人的供需關係。

v. 從上述的方法可建立大部分軟體代理人的供需關係，但仍有下列三種情況無法正確建立出軟體代理人間的供需關係：

a 如圖 3- 18 所示，如果兩者約定的 key 不屬於明碼的話，則無法由 Proxy 找出正確對應的軟體代理人，因此先警告測試者，再將所有可能的 set-get 的組合取得的 Proxy 都建立軟體代理人間的供需關係。

b 如圖 3- 16 所示，在群播的情況下，彼此間約定的訊息不是明碼的話，則無法由約定的訊息找出正確對應的軟體代理人，因此先警告測試者，再將所有可能的 multicast-subscriber 的組合都需建立軟體代理人間的供需關係。

c 如果得到 Proxy 的方式是接收別人利用訊息傳送過來的話，則無法得知軟體代理人利用這個 Proxy 來跟哪個軟體代理人溝通，只能發出警告無法找出 Proxy 對應的軟體代理人。

vi. 當兩個軟體代理人間具有雙向的供需關係時，透過前面提過的方法找出是否為循環式的供需關係，以簡化彼此間的供需關係。

3.2. 軟體代理人系統的測試策略

根據對軟體代理人系統分析的結果，軟體代理人系統的測試包含兩個部分，決定軟體代理人的測試順序與軟體代理人特性的測試，本節就是針對這兩個部分提出完整的軟體代理人測試策略。

若軟體代理人系統只具有單一軟體代理人，則直接測試該軟體代理人即可。若為多軟體代理人系統，首先必須利用根據軟體代理人間的主從關係找出相同主從架構的軟體代理人們，將這些軟體代理人包成一個成份，可能分成下列三種情況：

1. 若軟體代理人系統只具有一個成份，且 Slave 間沒有供需關係，如圖 3- 20 所示，則直接測試該成分即可，根據主從的關係，由最底層的 Slave 先測，再往上一層測試，直到測試到最上層的 Master 為止；若 Slave 與 Master 間具有訊息傳遞或發送的動作，則 Slave 先利用 stub 來測試，Master 則直接利用測試完的 Slave 來測試。

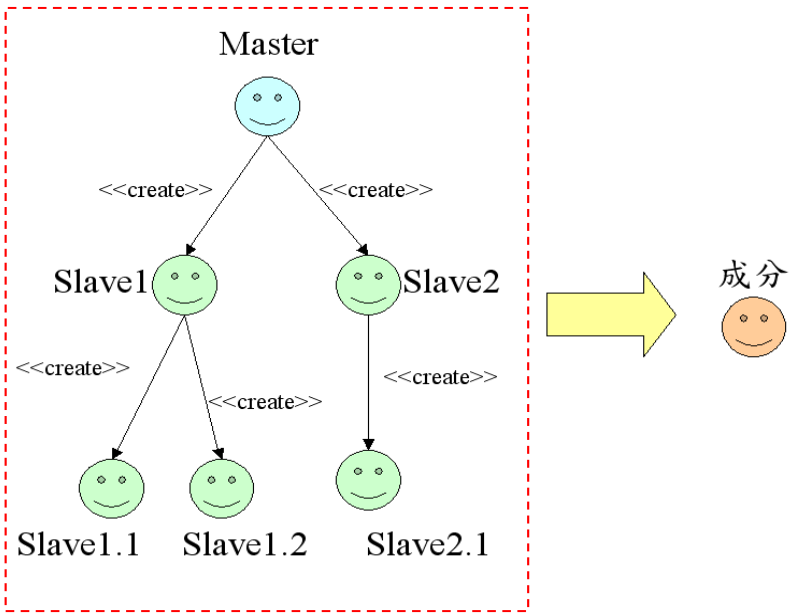

圖 3- 20 單一成份的軟體代理人系統

2. 若軟體代理人系統只具有一個成份，但 Slave 間具有供需關係，則再根據 Slave 間的主從關係找出相同主從架構的 Slave 們，分別包成數個小

成份，先測試獨立的小成份內的軟體代理人，再根據小成份間的供需關係，扮演 Provider 的小成份先測，再測扮演 Requestor 的小成分，小成份內的軟體代理人仍利用主從關係來決定測試順序，且 Provider 的訊息傳遞需利用 stub 來測試，而 Requestor 則可直接利用測試完的 Provider 來協助訊息傳遞的測試。若小成分內的 Slave 仍具有供需關係，則依據遞迴處理，直到不再具有供需關係為止。

3. 軟體代理人系統具有多個成份(如圖 3- 21 所示)，成份間會具有供需關係，先根據成份間的供需關係來決定測試順序，扮演 Provider 的成份先測，再測扮演 Requestor 的成分，且 Provider 的訊息傳遞需利用 stub 來測試，而 Requestor 則可直接利用測試完的 Provider 來協助訊息傳遞的測試，而成份內的測試順序，根據 Slave 間是否具有供需關係，來決定利用上述 1 或 2 的方法來決定。

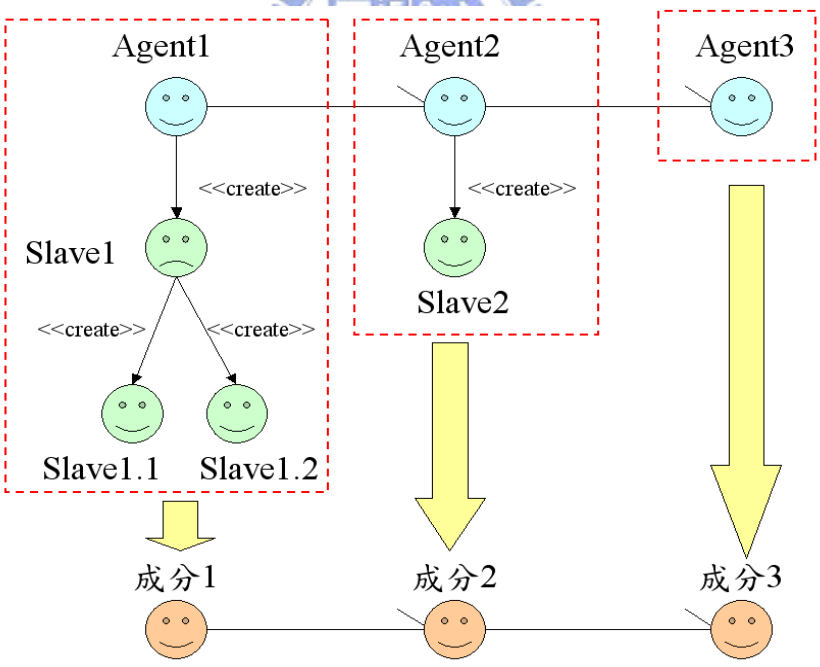

圖 3- 21 多成份的軟體代理人系統

總合上述的策略，可得到軟體代理人的測試順序的安排原則是(1)Slave 較 Master 先測，(2)Provider 較 Requestor 先測。這樣的安排原則有以下兩個優點：(1)減輕撰寫 stub 的成本，因為受測的軟體代理人的 Slave 以及需要傳遞訊息的

Provider 皆已事先測試過，所以可以直接拿來協助測試，除非遇到循環關係，(2)同時將單一軟體代理人測試與軟體代理人間的整合測試作完。

接下來針對軟體代理人特性，來討論軟體代理人的測試策略，分成下列六點來討論：

1. 因為在測試單一軟體代理人時，合作能力必須透過測試者設計 stub 才能夠測試，因此將合作能力放在最後測試。
2. 只要能在單一的機器上提供足夠的資料或資源，就可測試自動化是否正確，且可利用已經測試正確的對應軟體代理人或 stub，便可測試合作能力是否是正確，學習能力則可透過軟體代理人對知識庫是否正確執行寫入的動作來測試，因此將軟體代理人的行動力與其它的特性(自動化、合作能力、學習能力)分開來測試，先在單一機器上測試完軟體代理人的自動化、學習能力，再來測試行動力的特性。
3. 自動化的測試方法直接利用一般物件導向的測試方法即可。
4. 學習能力：只有軟體代理人程式碼的情況下，無法判斷學習能力的動作，因此必須由驗證是否正確的寫入知識庫，來測試軟體代理人的學習能力。
5. 行動力：如果軟體代理人的 itinerary 為直接寫入在軟體代理人程式碼中，則測試行動力時，直接執行並比對是否正確的執行遷徙的動作；否則軟體代理人的 itinerary 為變動的參數，由測試者提供 itinerary，根據提供的 itinerary 來驗證是否正確的執行遷徙的動作。
6. 合作能力的測試，依據三種不同的合作能力提出對應的測試策略：
   i. 產生：只有 Master 需測試產生的能力，除了測試是否能夠正確的產生軟體代理人之外，還要了解產生的軟體代理人之種類及數目，因為 Master 與 Slave 之間通常會透過產生取得的 Proxy，來執行訊息傳遞或發送的合作動作。

ii. 發送：發送的動作發生於 Master 跟 Slave 之間，可藉由 Proxy 來找出對應的被發送者，則 Slave 必須先測完本身的特性，再透過 stub 來測試被發送的行為，而 Master 則直接利用已測試完 Slave 來驗證發送的指令是否正確。

iii. 訊息傳遞：分成傳送者跟接收者來討論。

a. 傳送者根據傳送的動作可分成三種來討論：

vii. 同步傳送(Synchronous Send)針對下列兩種可能的情況來測試：

i. 給予正確的 Proxy 的情況。

ii. 給予錯誤的 Proxy，以致於無法訊息傳遞或將訊息送給錯誤的軟體代理人的情況。

viii. 非同步傳送(Asynchronous Send)：除了要測試上述的兩種測試情況，還必須測試等待回傳時，軟體代理人是否能夠繼續正確執行工作。

ix. 群播(Multicast)：若此傳送者對多接收者，則先一一測試每個傳遞訊息的組合，測試方法與非同步傳送相同，再測試同時與所有接收者作訊息傳遞的動作是否正確。

b. 接收者根據處理訊息的方式分成兩種

i. 循序式訊息處理(Sequential Message Handle)：需測試下列兩種情況：

甲、測試 Message Handler 中所有處理訊息可能的執行路徑

乙、是否有應該支援的訊息，卻在 Message Handler 中沒有對應處理的路徑。

ii. 並行式訊息處理(Parallel Message Handle)，其中並行式

訊息處理除了測試各種訊息處理的執行路徑之外，還要測試所有可能的並行路徑的情況，以圖 3- 22 為例，並行執行的路徑

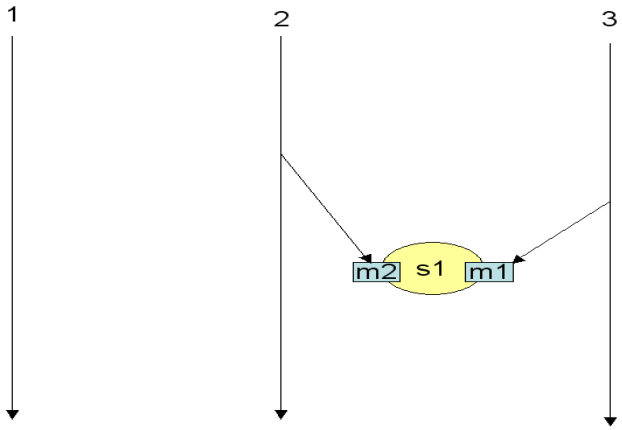

圖 3- 22 並行式訊息處理

為處理訊息為 2 的路徑，根據不同的接收訊息的情況，若訊息為 1、2、3，則 1 與 2 的路徑會發生並行的情況，如圖 3- 23(A)；若訊息為 3、2、1，則 2 與 3 的路徑會發生並行的情況，如圖 3- 23(B)，除此之外還要考量是否有呼叫共用物件[7][8][9][10]，則實際上

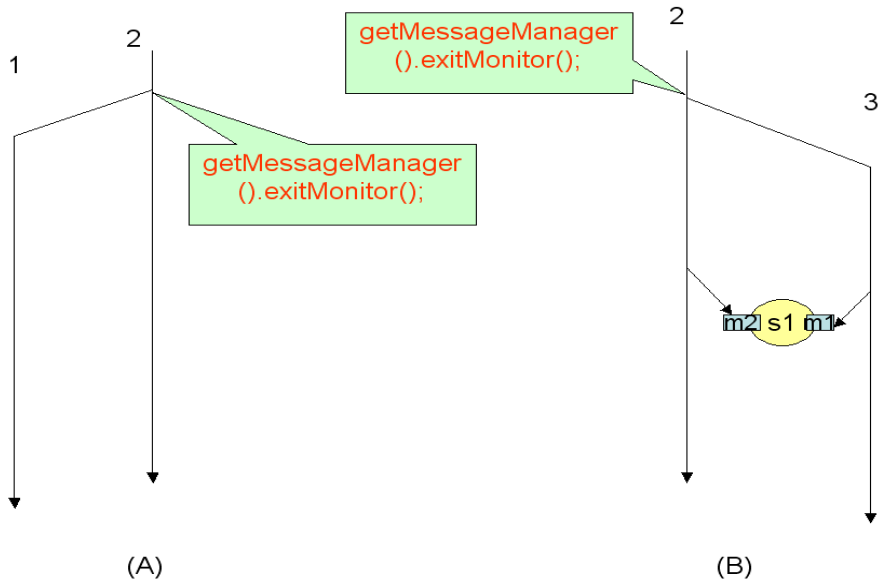

圖 3- 23 不同的並行情況

只需要測試 2 與 3 並行的情況，下面針對這樣的情況提出完整的產生出需測試的並行組合之方法：

(1) 根據 exitMoinitor 的指令，將”handleMessage”內的所有路徑分成”multithread”跟”singlethread”兩個集合。

(2) 建立”multithread”中所有路徑間共用物件的關係。

(3) 找出所有具有共用物件關係的”multithread”路徑，根據找出的順序當作是訊息的接收順序。取出”singlethread”中的一條路徑，找出與它有共用物件關係的”multithread”路徑，根據找出的順序當作是訊息的接收順序，而”singlethread”的路徑則為最後接收的訊息。

(5) 重複步驟 4~5，直到所有”singlethread”中的路徑都產生出所有需測試的訊息的接收順序。

第4章　軟體代理人系統的測試方法

依據上一章所提軟體代理人系統的測試策略，在本章進一步提出一套系統化的軟體代理人系統測試方法，軟體代理人系統測試方法可以分成：軟體代理人測試順序與軟體代理人程式之測試，在 4.1 節介紹軟體代理人程式前置作業分析已得到軟體代理人間的關係及特性，進而在 4.2 節，提出軟體代理人測試順序安排方法，4.3 節中，提出軟體代理人程式測試方法。

4.1. 前置作業

欲決定各軟體代理人間的測試順序及軟體代理人內程式之測試路徑安排，必須先蒐集足夠的資訊後，因此必須先剖析每一個軟體代理人的程式碼，並取得且儲存下列的資料：

1. 軟體代理人只有一核心的類別，由此類別來負責整個軟體代理人的邏輯運作，以 IBM Aglets 為例，核心類別會繼承 Aglet 類別[6]，如**圖 4-1** 所示，因此可將該類別的名稱視為軟體代理人的名稱。

```
public class Example extends Aglet{}
```

圖 4-1 核心類別接下來必須從軟體代理人程式碼找出該軟體代理人所具有的特性，軟體代理人之特性與程式碼之關係性討論如下：

i. 行動力：以 IBM Aglets 為例，如圖 4-2 所示，程式碼中有 Dispatch 指令，表示軟體代理人具有行動力的特性，將其程式碼位置與遷徙地點(變數或常數)存入行動力資料表中，如表 4-1 所示。

程式碼位置	遷徙地點

表 4-1 行動力資料表

```
public class  Mobile extend Aglet{
    public onCreation{
        URL destination = new URL (atp://some.host.com);
    }
    public void run{
        //do task
        if(目標未達成){
            Dispatch(destination);
        }
    }
}
```

圖 4-2 行動力程式範例

ii. 合作能力，可以分成三個合作的方式來討論：

a. 產生子軟體代理人：如圖 4-3 所示，以 IBM Aglets 之 createAglet

```
public class Creator extends Aglet{
    public void run() {
        AgletProxy proxy =
            getAgletContext().createAglet(getCodeBase(),"Child",null);
    }
}
```

圖 4-3 產生者的程式範例

指令執行時將產生軟體代理人，它的三個參數，分別代表被產生者程式碼所在路徑、被產生者核心類別名稱、輸入給被產生者的資料，而從被產生者核心類別名稱代表被產生的軟體代理人名稱，產生動作代表軟體代理人間的主從關係。由此指令，可將被產生的對應軟體代理人名稱、程式碼的位置以及輸入給被產生者的資料存入產生資料表中，如表 4-2 所示，另外，藉由產生的指令可以得到被產生軟體代理人的 Proxy，因此可以

將 Proxy 名稱與所對應的軟體代理人名稱存入 Proxy-軟體代理人對應資料表中，如表 4-3 所示。

程式碼位置	被產生的軟體代理人名稱	被產生者的輸入資料

表 4-2 產生資料表

Proxy名稱	對應軟體代理人名稱

表 4-3 Proxy-軟體代理人對應資料表

b. 訊息傳遞分成兩個部分：

（1）訊息傳遞的動作，可分成傳送跟接收：

甲、傳送：

- 同步傳送：圖 4-4 所示為 sendMessage 同步傳送的指令，根據左邊是否回應(Reply)物件可來判斷是否會回傳結果，可將動作指令(sendMessage)、回傳物件名稱(reply)、Proxy 名稱(proxy)、訊息(Hello)，以及程式碼位置存入傳送資料表，如表 4-4 所示。

```
public class  Synsend extend Aglet{
   public void run{
         Reply reply = proxy.sendMessage(new Message("Hello"));
   }
}
```

圖 4-4 同步傳送的程式範例

傳送的動作	回傳的物件名稱	Proxy名稱	訊息	程式碼位置

表 4-4 傳送資料表

- 非同步傳送:圖 4-5 所示為 sendFutureMessage 非同步傳送指令，根據左邊是否(FutureReply)物件可來判斷是否會回傳結果，可將動作指令(sendFutureMessage)、回傳

物件名稱(reply)、Proxy 名稱(proxy)、訊息(Hello)，以及程式碼位置存入傳送資料表。

```
public class  Asynsend extend Aglet{
   public void run{
        Message msg = new  Message("Hello");
        FutureReply reply = proxy.sendFutureMessage(msg);
   }
}
```

圖 4-5 非同步傳送的程式範例

- 群播：如圖 4-6 所示之 multicastMessage

```
//Agent A
public class A extends Aglet{
        public void run() {
            getAgletContext().multicastMessage(new Message("Hello"));
        }
}
//Agent B
public class B extends Aglet{
        public void onCreation(){
            subscribeMessage("Hello");
        }
}
```

圖 4-6 群播的傳送者與接收者程式範例

群播指令，根據左邊是否有宣告非同步回應(FutureReply)物件來判斷是否會回傳結果，若沒有的話，將動作指令(sendFutureMessage)、回傳物件名稱(reply)、message(Hello)，以及程式碼位置存入傳送資料表。

乙、接收：如圖 4-7 所示，根據 exitMonitor，可找出 handleMessage 函式內的那些路徑為並行式的處理路

徑，將所有的 message 以及是否並行存入接收資料表，如表 4- 5 所示。

```
public class Receiver extends aglet {
        public boolean handleMessage(Message msg) {
                if (msg.sameKind (“1”)) {
                        //do something…
                        getMessageManager().exitMonitor();
                        msg.sendReply(someReply);
                        return true;
                }else if (msg.sameKind (“2”)) {
                        //do something…
                        return true;
                }else if (msg.sameKind (“3”)) {
                        //do something…
                        return true;
                }
                        return false;
                }
        }
}
```

圖 4- 7 並行式訊息處理的程式範例

處理的訊息	並行式(True/False)

表 4- 5 接收資料表

（2）建立訊息傳遞的管道的方式，可分成兩種：

甲、利用 Proxy 來建立，只要用到有宣告 Proxy 的指令，將該 Proxy 的名稱加入 Proxy-軟體代理人對應資料表，由取得 Proxy 的方法來討論：

- 由產生其他軟體代理人得來的 Proxy，前面討論產生時已經說明，如何加入 Proxy-軟體代理人資料表。
- 如圖 4- 8 所示，由 proxy 的物件宣告可知是由起始物件取得的，表示由產生者在產生時利用參數輸入，但單只有剖析該軟體代理人無法得知 Proxy 對應的軟體代理

人，則先建立該軟體代理人的起始 Proxy 資料表，如表 4- 6 所示，將自己的軟體代理人名稱以及 Proxy 名稱存入其中。

```
public class  Example extend Aglet{
    public void onCreation(Object init){
        proxy = (AgletProxy)init;
    }
}
```

圖 4- 8 起始 Proxy 的程式範例

軟體代理人名稱	Proxy名稱

表 4- 6 起始 Proxy 資料表

- 如圖 4- 9(A)所示，由 proxy 的物件宣告的等式中可知是由利用 getProperty 和 key 來取得，並透過建立參數的宣告資料表，比對出 key 為變數或常數，將 key(參數或常數)、proxy 的名稱存入該軟體代理人的 get 資料表中，如表 4- 7 所示；或是如圖 4- 9(B)所示，利用 setProperty 和 key 來註冊自己的 Proxy，並透過建立參數的宣告資料表，比對出 key 為變數或常數，將 key(參數或常數)存入該軟體代理人的 set 資料表中，如表 4- 8 所示。

key	Proxy名稱

表 4- 7get 資料表

key

表 4- 8set 資料表

```
(A)
public class  Example extend Aglet{
   public void onCreation(Object init){
         AgletProxy proxy =
         (AgletProxy)getAgletContext().getProperty("keyword");
   }
}
(B)
public class  Example extend Aglet{
   public void onCreation(Object init){
         getAgletContext().setProperty("keyword", getProxy());
   }
}
```

圖 4-9 get-set 的程式範例

➢ 透過訊息傳遞得來，只針對利用 sendMessage 去要 proxy 的方式來處理，如圖 4-10 所示，則建立 ask 資料表，將被要求的 Proxy 名稱，與回傳 Proxy 的名稱存入 ask 資料表，如表 4-9 所示。

```
public class  Synsend extend Aglet{
   public void run{
         AgletProxy proxy  = proxy.sendMessage(new Message("Hello"));
   }
}
```

圖 4-10 要求回傳 Proxy 的程式範例

被要求的Proxy名稱	回傳的Proxy名稱

表 4-9 ask 資料表

乙、利用訊息來建立

➢ 只有 multicastMessage 可以利用訊息來建立訊息傳

遞的管道，而 multicastMessage 的指令皆已儲存在軟體代理人的傳送資料表中，所以只需建立訂閱資料表，如表 4- 10 所示，來紀錄軟體代理人程式碼中所訂閱的訊息(變數或常數)。

訊息

表 4- 10 訂閱資料表

(3) 發送：可找出透過 Proxy 執行 dispath 的指令，且可知發送指令都是由產生者發送給被產生者，因此可以得知被發送的軟體代理人，所以建立發送資料表，如表 4- 11 所示，來儲存被發送軟體代理人名稱、發送地點(變數或常數)，以及程式碼位置。

被發送的軟體代理人名稱	發送地點	程式碼位置

表 4- 11 發送資料表

3. 因為學習能力並非軟體代理人的標準指令，所以無法從程式碼的剖析中得知軟體代理人是否具有學習能力，當剖析完一個軟體代理人後，則詢問測試者，該軟體代理人是否具有學習能力以及它的知識庫名稱，並將知識庫名稱儲存在學習能力資料表中，如表 4- 12 所示。

知識庫名稱

表 4- 12 學習能力資料表

4. 建立軟體代理人 Node 的資料結構，如表 4- 13 所示，包含：

ii. 軟體代理人名稱為軟體代理人的核心類別名稱。

iii. 主從位元為 0 或 1，起始值皆為 0，一旦成為其他軟體代理人的 Slave 則變為 1，所以只有主從關係架構最上層的軟體代理人會為 0。

iv. 行動力：為一個連結，負責連結到行動力資料表。

v.學習能力：為一個連結，負責連結到學習能力資料表。

vi. 合作能力：可分成四種連結:產生、發送、傳送以及接收，分別連接到對應的資料表。

vii. Proxy：負責儲存所有取得的Proxy相關的資料，可分成六種連結：起始、get、set、ask、訂閱以及Proxy，分別連接到對應的資料表。

viii. Slave List:存放與被該軟體代理人產生的Slave的Node之連結(Slave Link)，若有多個則產生對應的連結數目。

ix. Provider List:存放被該軟體代理人傳送的Provider的Node之連結(Provider Link)，若有多個則產生對應的連結數目。

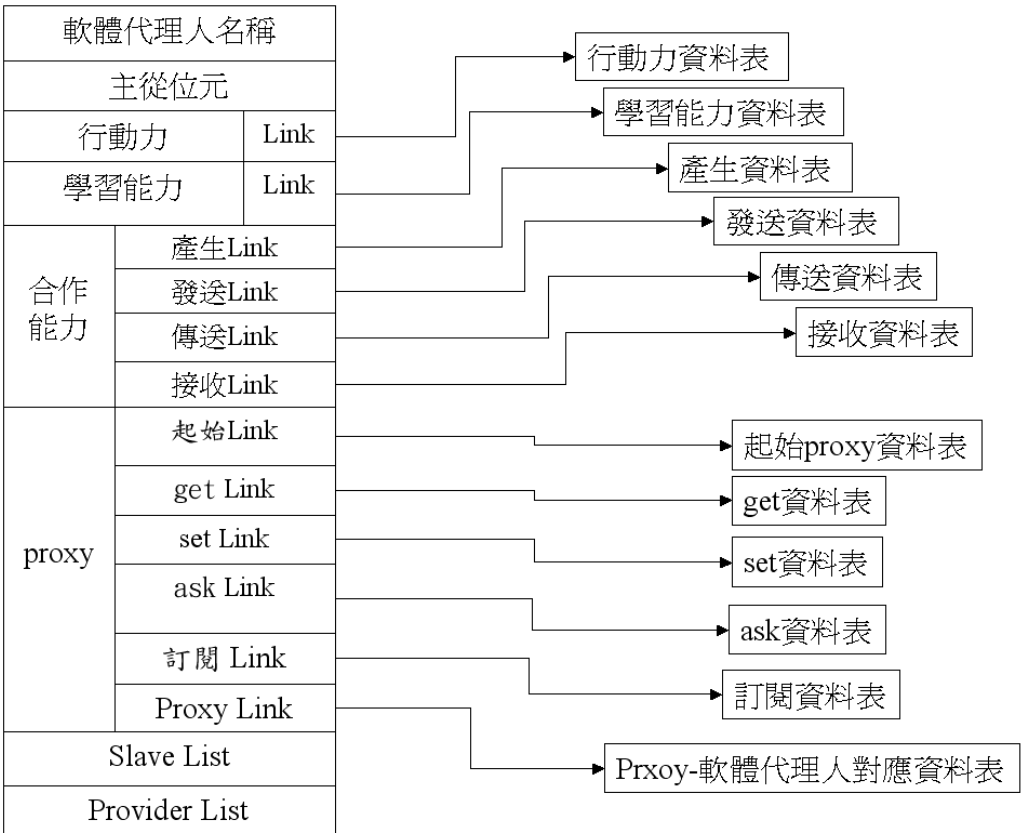

表4-13 軟體代理人的資料結構

5. 根據軟體代理人是否具有行動力、學習能力、合作能力的資料連結，可判斷軟體代理人具有哪些特性。

6. 因為除了產生以外，其他取得Proxy的方式皆無法只單由剖析單一軟體

代理人程式就可得知 Proxy 對應的軟體代理人，所以等到所有軟體代理人程式皆剖析完，依照下列方法透過其他的軟體代理人來找出 Proxy 對應的軟體代理人：

i. 若起始 Proxy 資料表不為空，則取出它的軟體代理人名稱，與所有其他軟體代理人的產生資料表比對，找出它的產生者，取出該筆資料輸入參數的 Proxy 名稱，再去利用該 Proxy 名稱與其 Proxy-軟體代理人對應資料表比較，找出對應軟體代理人名稱，則將起始資料表的 proxy 名稱以及找出的對應軟體代理人名稱存入自己的 Proxy-軟體代理人對應資料表。

ii. 若 get 資料表不為空，則取出 key，若 key 為常數，則與所有其他軟體代理人的 set 資料表中 key 比對，將 key 相同的該軟體代理人的名稱，與該 key 的 Proxy 名稱，存入自己的 Proxy-軟體代理人對應資料表；若 key 為變數，則將所有 set 資料表不為空的軟體代理人名稱，與該 key 的 Proxy 名稱，存入自己的 Proxy-軟體代理人對應資料表。

iii. 若 ask 資料表不為空，則取出被要求的 Proxy 名稱，與自己的 Proxy-軟體代理人對應資料表，找出所有對應這個 Proxy 名稱的軟體代理人名稱，依據軟體代理人名稱取出從它的 Proxy-軟體代理人對應資料表中，取出它已知所有的軟體代理人名稱，與回傳的 proxy 名稱，存入自己的 Proxy-軟體代理人對應資料表。

iv. 完成後，若還有 Proxy-軟體代理人對應資料表中，若還有 Proxy 沒有對應的軟體代理人名稱，則警告使用者該 Proxy 可能沒有對應的軟體代理人，由使用者來完成缺少的部分。

7. 根據產生的資料表，取出被產生的軟體代理人名稱，建立 Slave Link，並更改 Slave 的主從位元為 1，依序建立完所有軟體代理人的 Slave

Link，則產生樹狀的主從關係架構，由有多少個主從位元為 0 的 Node，便可知道有多少個主從關係架構。

8. 根據傳送的資料表，取出對應的 Proxy 名稱，來與 Proxy-軟體代理人對應資料表比對，找出所有對應這個 Proxy 名稱的軟體代理人，來建立 Provider Link。藉由每個軟體代理人的傳送資料表取出群播的訊息，與其他軟體代理人的訂閱資料表比對訊息，若訊息為常數則找出所有訊息相同的軟體代理人，來建立 Provider Link，若訊息為變數，則找出所有具有訂閱訊息的軟體代理人，來建立 Provider Link。

10. 若有多個主從位元為 0 的 Node，則檢查它們的主從關係架構內 Slave 間是否具有供需關係，若有則建立兩個 Node 間的 Provider Link。

4.2.軟體代理人的測試順序

軟體代理人的測試順序如下：

1　若只有一個主從位元為 0 的 Node 且 Node 的 Slave Link 為 Null，則可判斷該系統只有單一個軟體代理人，所以直接測試該軟體代理人即可。

2　若只有一個主從位元為 0 的 Node 且 Node 的 Slave Link 不為 Null，則判斷該系統只有單一個主從關係架構，演算法如下：

```
Stack order
Test_Order (vertex v){
    push (v) into order
    選取所有的 Slaves of v into V  //V 為一個 list
    E = all Provider-Requestor relation with vertices in V
    while (V is not Empty){
        pick v1 with the least indegree
        remove all Provider-Requestor relation with v1 in E
        remove v1 from V
        Test_Order(v1)
    }
}
```

3　若有多個主從位元為 0 的 Node，則判斷該系統有多個主從關係架構，則先找出所有位元為 0 的 Node 間的供需關係，每次選取供需關係中 indegree 最小的，然後呼叫 2 的演算法來決定該主從關係架構的測試順序。

4.3.軟體代理人程式的測試方法

依據第三章所討論的測試策略，提出一套滾動式的軟體代理人程式測試方法，則依據軟體代理人所具有的特性，根據下列幾種情況，分別提出特性測試順序:

1. 若只有一個主從位元為 0 的 Node 且 Node 的 Slave Link 為 Null，則可判斷該系統只有單一個軟體代理人，這種情況下不會具有合作能力，所以軟體代理人特性的測試順序為先測試自動化，再測試學習能力，最後測試行動力。
2. 若只有一個主從位元為 0 的 Node 且 Node 的 Slave Link 不為 Null，判斷該系統只有單一個主從關係架構，前一節提出的方法，建立軟體代理人的測試順序，依序取出測試，則可能碰到情況如下：
   i. 若 Node 的 Slave Link 為 Null，則表示在測試最底層的軟體代理人時，若不具有合作能力，則同單一軟體代理人的測試；若具有合作能力，則先測試自動化，再測試學習能力，然後測試行動力，最後利用 stub 來測試合作能力。
   ii. 否則 Node 的 Slave Link 不為 Null，則表示它是某些軟體代理人的 Master，則先測試自己的特性，測試順循序為先測試自動化，再測試學習能力，然後測試行動力，最後利用 stub 來測試合作能力；接著根據 slave 間的供需關係，依序抓出 indegree 最小的 Slave，根據該 Slave 的 Provider Link，測試在相同主從架構下與其他 Slave 間的合作能力，若出現雙向的供需關係，則利用第三章的方法先檢查是否為循序式的訊息傳遞來處理，最後測試所有 Master-Slave 間的合作能力。
3. 若有多個主從位元為 0 的 Node，則利用 2 的方法先測試完每個主從架構，當所有的主從架構都測試完後，則根據主從位元為 0 的 Node 間的

供需關係，依序抓出 indegree 最小的 Master，找出為它的 Provider 之其他主從架構來測試合作能力，若出現雙向的供需關係，則利用第三章的方法先檢查是否為循序式的訊息傳遞來處理。

下列提出軟體代理人的特性的測試方法：

1. 自動化：先來討論軟體代理人的程式架構，如圖 4-11 所示，分別針對這些部分來介紹：

   i. onCreation 為軟體代理人的起始函式。

   ii. run 為軟體代理人的執行函式，自動化的執行邏輯都在該函式中。

   iii. run 執行完後，則軟體代理人會等待下一步指令。

   iv. 如果有其他軟體代理人傳送訊息過來，由 wait 轉換到 handleMessage 函式來負責處理接收訊息。

   v. 軟體代理人在執行遷徙的指令前，由 wait 轉換到 onDispatching 來處理。

   vi. 軟體代理人在遷徙到目的地後，則由 onDispatching 轉換到 onArrival 來起始軟體代理人繼續執行它的工作。

   vii. 軟體代理人在結束前，則由 wait 轉換到 onDisposing 來處理動作。

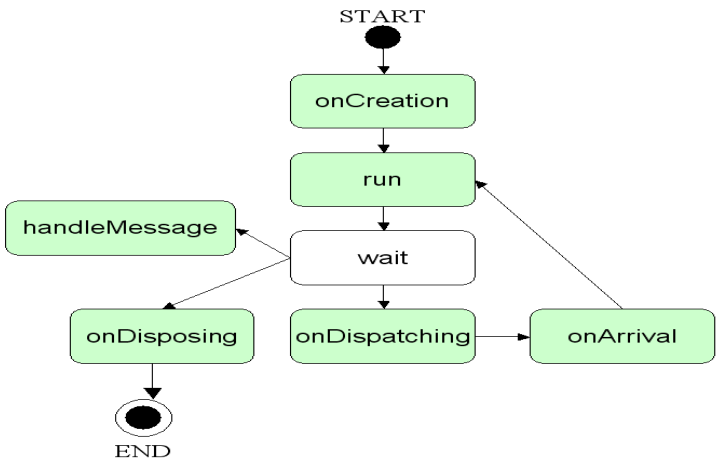

圖 4-11 軟體代理人的程式架構

因此可以得知，若要測試軟體代理人的自動化，則只需測試軟體代理人程式中的 onCreation 與 run 兩個函式，而要單獨只執行這兩個函式必須先將軟體代理人其它的特性排除，若軟體代理人具有行動力的特性，則可由前置作業已經建立好的行動力資料表，根據程式碼位置先取消掉這些特性的指令，且透過訊息傳遞來取得的資料，改由 stub 來輸入所需資料，因此利用前製作業建立好的傳送資料表，找出指令所在的位置，依下列的方法，根據傳送方式來編寫對應的 stub：

i. 同步傳送：必須將程式碼改寫成，如圖 4- 12 所示，讓 proxy 指向測試者設計的 stub，讓測試者先由 stub 輸入所需資料，來測試自動化。

```
public class  Synsend extend Aglet{
   public void run{
        AgletProxy proxy=
        getAgletContext().creatAglet(getCodeBase(),"stub", null);
        Reply reply  = proxy.sendMessage(new Message("Hello"));
   }
}
```

圖 4- 12 修改後同步傳送的程式

ii. 非同步傳送：必須將程式碼改寫成，如圖 4- 13 所示，讓 proxy 指向測試者設計的 stub，讓測試者先由 stub 輸入所需資料，來測試自動化。

```
public class  Asynsend extend Aglet{
   public void run{
        Message msg = new  Message("Hello");
        AgletProxy proxy=
        getAgletContext().creatAglet(getCodeBase(),"stub", null);
        FutureReply reply = proxy.sendFutureMessage(msg);
   }
}
```

圖 4- 13 修改後非同步傳送的程式

iii. 群播：測試者編寫一個訂閱該訊息的 stub，如圖 4- 14 所示，由 stub 輸入所需資料，來測試自動化。

```
public class stub extends Aglet{
        public void onCreation(){
            subscribeMessage("Hello");
        }
        public boolean handleMessage(Message msg) {
            if (msg.sameKind ("Hello")) {
                //do something…
                msg.sendReply(someReply);
                return true;
            }
                return false;
        }
}
```

圖 4- 14 對應群播的 stub 程式

除了上述的三種情況，其餘的合作能力指令，則利用前置作業已建立好的產生資料表、發送資料表與傳送資料表，找出特性指令所在程式碼的位置先取消掉其功能，則便可在軟體代理人執行環境上，測試軟體代理人的 onCreation 與 run 函式，利用 JAVA 測試方法來產生測試的路徑，只要提供充足的資料便可測試所有測試路徑，以判斷自動化是否正確。

2. 學習能力:因為無法找出學習能力的指令動作，所以若軟體代理人有寫入知識庫的動作，則比較寫入知識庫的結果是否與預期的結果相同，來判斷學習能力是否正確。

3. 行動力:在 onCreation 與 run 函式中找出具有行動力特性的函式路徑，一一去測試每一條函式路徑，而根據 itinerary 是否為常數來判斷，執行遷徙的次數以及必須設置的測試環境數，若為常數則依照指定的位址設置足夠資源，比較是否遷徙到正確的地點，以及在遷徙位置上的工作是否正確執行，否則，必須由測試者輸入 itinerary 給軟體代理人，並

會在遷徙到的地點設置足夠的資源，比較是否遷徙到正確的地點，以及在遷徙位置上的工作是否正確執行。

4. 合作能力：先利用 stub 來測試所有的合作能力的指令動作，傳送的動作，必須在所有的函式中找出傳送的動作，並將其 Proxy 指向 stub，而接收的動作，則需設計測試 stub 程式，如圖 4- 15 所示，一一測試 handleMessage 中所有的函式路徑，若具有並行式訊息處理的函式路徑，則利用第三章所提的方法來產生所有可能的並行處理路徑的組合。

```
public class stub extend aglet{
   public void run(){
      try{
            AgletProxy proxy =
getAgletContext().creatAglet(getCodeBase(),“Receiver”, null);
         try{
            Reply reply = proxy.sendMessage(new Message(“1”));
            proxy.sendMessage(new Message(“2”));
            proxy.sendMessage(new Message(“3”));
         }catch(NotHandledException e){}
      }catch (Exception e){}
   }
}
```

圖 4- 15 測試接收的 stub 程式

第5章　結論

軟體代理人具有自動化、學習能力、行動力及合作能力這四種特性，因此軟體代理人可模擬人類的行為模式，藉由代理人間互相的合作來自動地幫助人類解決繁雜的問題，而透過行動力使軟體代理人程式更具有：(1)減少訊息傳遞的成本，(2)資源利用最佳化，(3)簡化工作統整的程序等優點。雖然傳統物件導向的程式結構元件與軟體代理人程式相同，但軟體代理人比傳統應用程式更有彈性、主動性以及更能貼近使用者的想法，相對的，軟體代理人程式的測試也因此比傳統程式語言所開發的軟體更困難。

目前對於軟體代理人程式測試的研究，大部份均針對軟體代理人測試模擬環境提出研究成果，然而當一軟體代理人系統擁有多個軟體代理人，如何有系統地對此多軟體代理人系統程式進行完整測試的研究卻無人提出，對開發者而言，多軟體代理人程式之測試仍是一件困難之事。

因此，本研究分析軟體代理人的特性與行為模式來進行分析，找出軟體代理人特性對行為模式的影響，並歸納出軟體代理人間的主從關係與供需關係；藉由軟體代理人間相互影響的關係，可找出多軟體代理人系統的測試順序，而對單一軟體代理人程式的測試，經分析各項特性間交互影響後關係，提出必須先測試自動化特性，繼而測試行動力特性，最後測試合作能力，而學習力則必入自動化特性中測試，進而提出一套滾動式的軟體代理人程式測試方法，並採用將行動力與其他特性分開測試的策略，先在單一軟體代理人執行環境上分別測試軟體代理人的自動化與學習能力的特性，再測試軟體代理人的行動力，以減少執行重複的測試動作，以及測試設備的設置，最後測試軟體代理人間的合作能力，藉由這樣一步一步的測試軟體代理人的每項特性，使得測試者更能系統化地確認軟體代理人程式的正確性，進而提升軟體代理人程式的開發效率。

本研究之成果包含：

(1) 歸納出軟體代理人間的主從關係與供需關係:透過這兩種關係可以建立出各軟體代理人系統的關聯性,並利用這兩種關係建立軟體代理人間測試順序，使得多軟體代理人系統的測試上更有效率且具系統化。

(2) 提出系統化的一軟體代理人程式測試方法：針對軟體代理人的每項特性，提出對應的測試方法。以及測試的步驟，使得軟體代理人程式的測試能夠採用滾動式的方式，先測試自動化特性，在測試行動特性，最後才測試合作能力，使測試者能夠更容易去發現軟體代理人程式的錯誤。

未來的研究方向為

(1)本論文在測試的執行環境上，是直接使用 IBM Aglets 的執行環境，雖完成大部分功能，但限於時間，仍無法使用，實有必要予以完成，以驗證所提方法的可行。

(2)目前異質軟體代理人的合作研究越來越多，透過 ACL(Agent Communication Language)的方式,使得不同軟體代理人程式語言的軟體代理人程式可以互相合作，可將這樣的情況考量進來，以進行更完善的測試。

參考文獻

1. J. M. Bradshaw, "An Introduction to Software Agents," AAAIPress/The MIT Press, 1997
2. H. S. Nwana. " Software Agents: An Overview," Knowledge Engineering Review, 11(3), pp.1-40, Sep 1996.
3. 許嘉容, "利用行動代理人的互助式學習之網路管理系統", 國立交通大學資訊工程研究所, 碩士論文, 2001
4. Christopher Rouff, "A Test Agent for Testing Agents and their Communities" In Proceedings of the IEEE Aerospace Conference, 2002.
5. Adelinde M. Uhrmacher, Bernd G. Kullick, "PLUG AND TEST" - SOFTWARE AGENTS IN VIRTUAL ENVIRONMENTS", Proceedings of the 2000 Winter Simulation Conference.
6. D. B. Lange and M. Oshima, "Programming and Deploying Java™ Mobile Agents with Aglets™", Addison Wesley Longman, Inc., August 1998
7. 李正國、林浩澄、鍾乾癸，"Java 程式測試方法"，第八屆物件導向技術及應用研討會，38-44 頁
8. 林浩澄，"Java 程式測試方法與 Java 程式測試環境之研究"，碩士論文，國立交通大學，1997
9. 陳家豪，"並行 Java 程式不確定執行行為之測試方法"，碩士論文，國立交通大學，1999
10. 李正國，"最佳測試個案選取問題之研究"，博士論文，國立交通大學，2000
11. M. Baldi, S. Gai, G. Picco and P. d. Torino, "Exploiting Code Mobility in Decentralized and Flexible Network Management," Proceedings of First International Workshop on Mobile Agent (MA' 97), Berlin, Germany, April 1997.
12. Roger S. Pressman, "Software Engineering - A Practitioner's Approach, 4th edition", McGraw-Hill, 1997
13. P. Morreale, "Agents on the Move", IEEE Spectrum, Vol. 35, p34-41, April 1998.
14. G. P. Picco, "Understanding, Evaluating, Formalizing, and Exploiting Code Mobility," PhD thesis, DIPARTIMENTO DI AUTOMATICA E INFORMATICA, 1998.
15. C. Ghezzi, G. Vigna and P. D. Milano, "Mobile Code Paradigms and

Technologies: A Case Study," Proceedings of First International Workshop on Mobile Agent (MA'97), Berlin, Germany, April 1997.
16. Y. Labrou, T. Finin and Y. Peng, "The interoperability problem: bringing together mobile agents and agent communication languages," Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences, pp. 10, Jan 1999.