第一章　緒論

第一節　研究動機

網路化時代來臨伴隨著軟硬體相互的演進。軟體技術從單一的循序程式(sequential programs)，慢慢的演進到多程序(multi-process)，多執行緒(multi-threading)的應用。再進一步，更可透過結合網路進行分散式的運算，或者在多 CPU 的主機上執行，以獲得更好的執行效能。這些系統都是屬於並行系統（concurrent systems)。與一般的循序程式有極大的不同。

設計 concurrent program 的主要目的是希望透過作業系統的分時(time sharing)或多個 CPU 的機制，運用多個程序(process) 或多個執行緒 (thread) 來共同解決問題。執行續或程序平行執行之後，可以增加執行效能（其中每一個 thread/process 可以被視為一個單一的 sequential program)。然而透過並行執行的 thread 或 process 並不是毫無代價的。首先，thread 與 process 彼此之間通常需要同步或訊息溝通來解決問題。同步或溝通不良的程式會產生眾所熟知的死結(deadlock)，資料競逐(race condition)，飢餓(starvation) 等等並行錯誤。很不幸的，這些並行錯誤通常難以除錯，偵測，或重現(reproducible)。使得傳統用測試來確保軟體品質的方法變得力有未逮。

目前，確保軟體品質的主要的技術仍是測試。測試是透過執行測試資料以檢測系統是否運作正常。通常，測試資料的完備與否會決定測試的品質。但是，實際上要執行完備的測試資料是不可能的，因為就算是一個普通的循序程式，他的可能執行路經通常是無限的。若是並行軟體，程式的可能路徑更呈指數性成長，這也成為測試方法力有未逮的主要原因。軟體測試有上述的理論與實務上的限制，要確保並行軟體的品質有必要求諸其他的方法。

數十年來探討如何確保並行軟體系統品質的研究也累積了許多成果（從早期的作業系統，Ada 語言的 multi-tasking，到現今的 Java multi-threads，以及

廣大的網路應用)。其中的成果有許多被發展成為軟體工具(Software engineering tools)。在這些方法當中，在理論上正好可以彌補測試上的不足而且也較具實用性的一種稱為自動化的軟體驗證技術(automatic verification)[9][10][11][12]。是一種大家較為看好的軟體技術。簡要的說，這類技術通常將一個程序或執行緒，抽象化為某種同步有限狀態機(communicating finite-state machine)。所以一個由多個程序/執行序構成的並行系統將被抽象化成為許多有限狀態機。有限狀態機與有限狀態機之間可以進行溝通以模擬程序之間的同步(process synchronization) 或跨網路的程序訊息傳遞(message passing)。通常這一群有限狀態機也被簡稱為系統模型(model)。

當模型被建構出來之後，自動驗證的分析引擎會利用窮舉的方法(exhaustive exploration)將該系統透過同步或通訊後的所有可能的狀態(reachable state)全部展開。所有可能到達的狀態稱之為狀態空間(state space)。理論上，若 process 或 thread 的抽象化過程中沒有損失重要的資訊，透過狀態空間的展開，所有可能的並行錯誤都能在狀態空間中顯現出來。展開後的狀態空間其實代表著該並行系統的所有可能達到的行為。利用該狀態空間，有許多分析的演算法，可以用來檢驗該系統的行為是否符合某些正確性性質(correctness property)。這類的方法以模型檢測(model checking)技術為代表[4][5]。

上述的技術，經由十來年的發展，都已經能自動化。自動化正是這類驗證工具吸引人的地方。但是，除了零星的業界成功應用的經驗，這類軟體驗證工具主要仍侷限於學術界。其中最主要的原因是軟體驗證工具的擴展性(scalability)。理論上，展開並行系統全部的組態空間是 PSPACE 的問題（可能狀態成指數性成長，或稱狀態爆炸)。在實際的應用當中，只要是稍微複雜的模型，很容易就耗盡所有的記憶體資源。所以就算某些模型檢測演算法（model checking) 的時間複雜度是 polynomial，一遇上並行系統，還是變成 PSPACE 的

問題。近年來，減緩狀態爆炸的方法[6] 陸續被提出來增加此類驗證工具的實用性。但是由於受限於理論上的侷限，每個方法的有效與否都具有特殊性。迄今沒有一種方法使得減緩狀態爆炸的成效，廣泛到使軟體驗證工具廣為業界所接受。

眾多減緩狀態爆炸的方法中，運用局部性分析(Compositional Analysis)是公認較具有潛力的。所謂的局部性分析就是利用系統原有的模組性或子系統架構把一個大系統分割為許多小系統(也就是所謂的 divide and conquer)，然後讓驗證工具分析子系統，最後再把子系統的分析結果結合起來達成分析原系統的目的（相對於局部性分析，前述的方法我們稱之為總體分析(global analysis)）。這樣的分析是一種階層式(hierarchical)的漸進式分析。理論上，只要能找到一個階層，裡面的每個子系統都不發生狀態爆炸，就可以分析原本總體分析(global analysis) 所不能分析的大型模型。

不過，局部性分析對有複雜介面子系統之系統，卻不能在分解與組合過程中防止組態爆炸，因此引進 Refactoring 方法 [7][8]。Refactoring 是自動化地透過一些等價的轉換(equivalence transformations) 來打破系統舊有的模組性 (as-built modularity) 以創造出新的系統架構。使得局部性分析在新的系統架構底下能夠發揮 divide and conquer 的優點。在[7][8]中，針對一些代表性的系統，經由 refactoring 轉換之後的新架構，局部性分析可以充分利用 divide and conquer 的方法，使得可分析的系統大小，突破以往的限制。

Refactoring 方法分析子系統之行為，分解重組行為以產生新的子系統，使得這些新的子系統組合起來對外只有簡單的界面，因而有效的縮減系統可到達之狀態，進而防止組態爆炸。但使用 Refactoring 轉移系統時，必需確保所轉成的新系統與原系統是等價的。

等價關係有不同的層次，在不同的應用方面使用不同的等價關係，我們希望的等價關係是可以取代的等價關係(congruence)，因此我們建立一套公設化代數結構，從這個代數結構架構一程序代數模型，證明 Refactoring 轉移成的新系統與原系統是具有取代的等價關係。

第二節　簡介

局部性分析能夠避免組態爆炸，主要原因為它在分解與組合的過程中，以簡單的介面程序(interface process)取代一個子系統，隱藏了子系統內元件的狀態，因此在接下來一步一步的組合過程中，可以避免可到達狀態的指數性成長。但對於已架構好的系統，它的元件間又有複雜的介面，若在不打破元件結構的前提下，是不可能找到簡單的介面程序，像這類的系統就不能應用局部性分析。

Cheng and Young 在 [7][8] 中介紹的 refactoring 轉換可以突破上述的限制。而且轉換過程中必須維持一個等價關係。在[7][8] 中，他們借用了 weak bisimulation 做為他們的等價關係。不過，weak bisimulation 是一種比較弱的等價關係，無法保留重要的特性像 liveness。所以，我們的目標是使用程序代數(process algebra)，驗證經由 refactoring 轉移後的系統與原系統，具有更強的等價關係，不用怕重要的特性會在 refactoring 轉換的過程中喪失。因此以程序代數的方式，分別將使用 refactoring 轉移前後的系統，以代數表示式來表達。原系統的代數表示式代表規格，轉移後的系統代表實作的模型，然後證明實作的模型滿足原系統的規格。

要以代數的方式來模擬程序的行為，就必需將組成程序的元素，分別抽象化成為代數的常數及函數、將程序間互相運作而產生相同的行為轉成公設，所以公設變成規範程序間相同的行為。在不同的應用方面，我們所在意程序的行為並不一樣，就會有不同的公設，譬如說如果我們只在乎外界的觀察，那麼程序 a(b+c) 與程序 ab+ac 就代表相同的程序，因此 x(y+z)= xy+xz 就可以被納入成為一個公設。當然公設不是隨便就可以任意自定的，必需符合一致性及不會互相矛盾。

有了公設我們還需要定義行為執行的關係(action relation)，即在不同的程序組合下(用運算子組合)，執行行為產生的結果，也就是成為另一個程序。現在我們可以說程序是否相等都是由公設決定的，或應用公設推論的結果，因為除了公設定義的相等外，我們並未定義任何東西使得兩個程序會相等，這樣我們稱

這些公設為 complete 公設。

當我們將更複雜的程序納入程序代數模擬的對象，例如遞迴等式代表的程序、含 τ 的程序等等，若只是採用公設定義的相等就限制了應用的範圍，因此必需另外定義一些視兩個程序為相等的關係，所以許多不同的 bisimulation 被定義出來，分別適用於不同的方面的應用。不論如何定義兩個程序視為相等的關係，它都必需是等價的關係，而且不能違反公設定義的相等，非等價關係幾乎不能應用，違反公設則必需作修改。因不能違反公設，所以程序都符合公設的定義，但公設已不能推論出程序還滿足那些等式。

過去的程序代數會有不同的 bisimulation 關係，皆因如何處理 τ 與其他行為合併執行的結果，因此會產生不同的 τ 定律，τ 定律又會形成新的公設。

第三節　論文結構

在第二章我們介紹一些會用到的背景知識：公設系統、代數(Algebra)的簡介與應用、項轉移系統(Term Rewriting System)、等價關係的簡介、局部性分析(compositional analysis)的簡介。第三章介紹程序代數(Process Algebra)之建構步驟，從最簡單的基本程序代數介紹到並行式的程序代數，這一章完全以 closed term 為對象來介紹，雖然程序代數的規格越來越豐富，但還是可以化成各種簡單的基本程序代數。第四章以第三章的程序代數規格為基礎分別加入定義程序的遞迴等式，形成各種不同的程序代數模型。第五章實作一個程序代數模型，證明 Refactoring 轉移成的新系統與原系統是具有取代的等價關係。第六章討論與未來發展方向應用。