國立臺灣師範大學資訊工程所

碩士論文

指導教授：鄭永斌 博士

Refactoring ARCATS for Maintainability and Evolvability by Design Patterns

研究生：傅心儀 撰

中華民國 九十九 年 一 月

摘要

局部性分析（Compositional Analysis）在軟體模型驗證(Model Checking)領域中被認為是該技術邁入實用化的重要關鍵技術。局部性分析之所以能夠成功，關鍵在於被檢驗的模型系統具備良好的模組架構，以階層式逐步分析並合成各個子系統，以達到分析整個系統的目的。在合成子系統的過程中並採用各種簡化技術以達到狀態空間的最小化，減緩組態爆炸的發生。

ArCats（Architecture Refactoring Compositional Analysis Tool Suite）即是依據此理論基礎所發展的局部性分析軟體驗證工具。ArCats 的合成引擎（Composition Tool）在列舉展開一個代表子系統行為的狀態空間（State Space）的過程中，以本文限制（Context Constraint）與簡化子系統（Reduction or Minimization）來降低組態爆炸（State Explosion）的問題，並在合成時一併檢測系統安全性質（Safety Property）。

在上述所提到的功能中，由於彼此間有著頻繁的合作且功能重複的特性，故以往的實作時皆 copy-paste 原有的程式碼再予以修改，以因應新功能的加入，造成系統維護和修改成本大幅提高。由於合成狀態的功能在模型驗證技術上具有核心的關鍵位置，在軟體開發的過程中，此類工具的程式碼若缺乏架構，導致無法重用與兼容，將會是後續專案開發上的阻力。隨著研究的進展，列舉狀態空間時所使用的合成規則，以及模型驗證工具欲檢驗性質的增加，缺乏彈性的實作方式僅會加深程式碼的晦澀，專案難以維護以及功能難以擴充。

本研究欲提出一個方法，將欲檢驗的性質和合成規則集中，在局部性分析列舉狀態時可隨時組合或是單獨使用這些合成規則。往後若需要分析不同型態的系統模型，亦可以輕易運用既有的合成規則，使得程式碼重覆利用，減輕專案開發的負擔。不僅可以讓程式碼更具彈性和擴充性，並期盼可以提供了更多規則的組合方式，以提供後續更深入的理論研究。

Abstract

Compositional Analysis is considered as a promising approach for scalability in the field of model checking. The success of composition analysis, however, depends on a good modularity in system architecture to verify a system in a hierarchical manner. In the hierarchical composition, there are several reduction and minimization techniques which can be applied to mitigate its state space efficiently.

ArCats (Architecture Refactoring Compositional Analysis Tool Suite) is a model checking tool which support compositional analysis. The composition tool of ArCats is capable of composing reachable state space with context constraint, on-the-fly minimization by branch bisimulation and safety property verification.

Previous implementations often use copy-paste programming to accomplish their tasks. The cost of maintenance and debugging is high when the system needs to be changed. Because the enumeration of states is a core feature that involves a lot of composition rules, it is unfortunately tangled by a lot of functionality and creates a lot of maintenance problems.

We propose an approach to refactor the whole system by centralizing the composition rules in a Rule Engine. While exploring state space the new system allows different combinations of rules, all based on a program. Our approach also keeps code readable and maintainable by refactoring. Eventually, we bring maintainability and evolvability to ArCats.

目錄

圖表目錄

表格目錄

Chapter 1　緒論

Section 1.1　研究動機

隨著軟體和硬體發展的進步，並行系統（concurrent systems）相較於以往傳統的循序程式（sequential program），可提供軟體系統更佳的效能和滿足不同需求。然而，並行系統的發展卻使得原本就不易檢驗的系統特性變得更加困難。現代電腦系統的發展規模愈形複雜龐大，且民眾依賴甚深。例如航太安全控制系統和醫療系統等關鍵性的系統（critical systems）發生錯誤，很可能會造成嚴重的金錢或人命的損失。故檢驗軟體正確性（correctness），對於這類系統是不可或缺的一個重要步驟。

模型驗證的技術，就是幫助我們發現系統設計時所可能產生的錯誤（例如deadlock），在系統開發生命週期的設計階段即予以修正，以確保後續實作時的正確性。一般來說，大型並行或分散式的程式由於系統行為都相當複雜而難以分析。因此，提供一個有效率且自動化的驗證技術是相當重要的。

模型驗證工具目前大多以總體性分析（Global Analysis）為主要方法。總體性分析的方法即是將程式執行時執行緒或程序彼此所有可能互動的路徑完全列舉出來，再逐一檢查是否違反某些性質。但總體性分析碰到的最主要問題在於，目前軟體系統規模日益龐大，在大型系統下，程式行為和程式之間的溝通也會相

對變得更複雜，進行軟體驗證時需分析的狀態數目就愈多，導致執行時實體記憶體不敷使用，此即為組態爆炸。這樣的困境使得許多以總體性分析為基礎的驗證工具，在尚未能檢驗出系統是否違反某些性質時，就因組態爆炸導致驗證失敗。

近年來已提出多種方法試圖解決這樣的難題，眾所皆知較具有潛力的解決方案即為使用局部性分析（Compositional Analysis）[20][22]來進行模型驗證。所謂局部性分析是利用系統原有的模組性（modularity）或系統本身的架構，將原本巨大的系統切割成許多小的子系統（divide and conquer）。其運作原理是一種階層式的分析，驗證工具只需逐一檢驗這些子系統，最後再將子系統分析的結果結合起來，藉以達到分析原有系統的目的。在局部性分析中，系統模組性的好壞是相當重要的關鍵

ArCats（Architecture Refactoring Compositional Analysis Tool Suite）是由Cheng[32]所率領開發的局部性分析工具，其本身具有模型架構重構的功能，解決了局部性分析最重要的問題。ArCats 目前在局部性分析的過程中，使用本文限制（Context Constraint）與簡化子系統（Reduction or Minimization） 來降低組態爆炸的問題，並可進一步分析系統是否違反安全性質（Safety Property）。

本文限制的用意在於減少在子系統中多餘的狀態（state）和移轉（transition），由於這些狀態是無法到達的（unreachable），因此愈早移除這種多餘狀態和移轉，對避免組態爆炸有愈大的助益。

在 ArCats 中採用 Branching Bisimulation Minimization 的化減技術[26]做為簡化子系統的方法。Branching Bisimulation 具有許多優於其他等價關係的特性，狀態空間經化減後可保留系統模型的必然性（liveness），演算法也有較佳的效率。

安全性質，簡單的說就是程式進行時不會進入不良的狀態。舉例來說，並行互斥（mutual exclusion）指的是一個系統資源不會同時被兩個以上的行程（process）進行存取，其所代表的就是一種安全性質。

在 ArCats 進行系統檢驗時，所有分析的基礎都是透過合成引擎執行狀態展開而來，而合成引擎在合成狀態時所使用的合成規則或遵循的步驟，會隨著我們欲檢驗性質或採用的即時化減方式而有所不同。例如在合成引擎中加入安全性質的合成規則，以檢驗系統模型是否違反了某種安全性質。或是在狀態平行合成時加入即時化減功能，以達到最小化的目的。

此外，使用不同的有限狀態自動機描述欲驗驗的系統模型，也會造成合成引擎使用合成規則的不同。目前 ArCats 所分析的 Labeled Transition System（LTS），主要以 Calculus of Communicating System（CCS）的模型為基礎。

由以上的說明可以得知，在合成引擎中，除了基本的狀態展開規則之外，影響狀態列舉時的因素還有：系統欲驗檢的性質，簡化子系統的方式，以及 LTS 型態。原有的系統針對上述功能的實作方式，我們以 Table 1 的虛擬碼說明。

CCS 狀態平行合成	1: 列舉出欲合成的 CCS 中每個狀態和行為 2: 並加入平行合成的結果

CCS 進行本文限制	1: while(進行 CCS 狀態平行合成){
	2: if (此行為未受本文限制)
	3: 此行為和產生的新狀態加入平行合成結果
	4: else{
	5: 此行為無法產生新狀態，
	6: 繼進行下一個行為的檢查
	7: }
	8: }

CCS 檢驗安全性質	1: while(進行 CCS 狀態平行合成){
	2: if(此行為產生的新狀態為不良狀態){
	3: 記錄此不良狀態，
	4: 此不良狀態會加入平行合成的結果中，
	5: 但後續不會再繼續拓展此狀態
	6: }
	7: else (此行為會產生新狀態)
	8: 將此新狀態和行為加入平行合成的結果
	9: }
	10: }

CCS 進行即時化減	1: while(進行 CCS 狀態平行合成){ 2: if (達到指定狀態數 ‖ 達到記憶體使用量 ‖ 3: 此層狀態皆已展開完畢) 4: 送至最小化引擎進行即時化減 5: }

CCS 進行本文限制
 	
與即時化減
	1: while(進行 CCS 狀態平行合成){
	2: if (此行為未受本文限制){
	3: 此行為和產生的新狀態加入平行合成結果
	4: if (達到指定狀態數 ‖ 達到記憶體使用量‖
	、 此層狀態皆已展開完畢)
	5: 送至最小化引擎進行即時化減
	6: }else{
	7: 此行為無法產生新狀態，
	8: 繼續進行下一個行為的檢查
	9: }
	10: }

Table 1 Pseudo-code of previous implement of Composition Tool

由 Table 1 的虛擬碼可知，每個功能之間有著重覆的步驟，亦即每個功能都和「CCS 狀態平行合成」有所重複，而由「CCS 進行本文限制與即時化減」的此函式，更充分顯示了此功能是以「CCS 進行本文限制」加上「CCS 進行即時化減」此兩個方式組合而成。是故在原有的程式碼中，由於程式功能重複的特性，造成每個功能（即每一個 method）中皆有 copy-paste 的情況。

以 Copy-paste 的方式實作系統的新功能，可能會使得既有程式碼產生無法預料的錯誤，導致除錯的成本提高。而維護的成本也會相對增加[14] [19]，例如我們要修改即時化減技術時，必須更動「CCS 進行即時化減」和「CCS 進行本文限制與即時化減」這兩個函式的程式碼。

除了 Copy-paste 實作方式產生的問題外，程式碼之間難以重用的情況，會造成專案開發成本提高。若我們試圖加入新的 LTS 型態進行局部性分析，例如 Communicating Sequential Process（CSP），則原有的 CCS 的功能皆無法重用，我們必須再重新實作「CSP 進行狀態平行合成」、「CSP 進行本文限制」、「CSP 進行安全性質檢查」…等全部的函式，即使 CSP 的合成規則和 CCS 有類似的實作方式，但將 CCS 的程式碼移植到 CSP 的模型中幾乎是不可能，勢必再花費心力重新撰寫，造成後續研究上時間與心力無謂的消耗。故合成狀態時所採用 LTS 的不同，也會挑戰合成引擎所能提供的彈性。

由於程式之間環環相扣的特性，在實作上往往隨著功能的增加，程式碼耦合（Coupling）的情況也變得更嚴重。合成狀態工具是模型驗證的核心技術，如果只為特定的合成規則或是 LTS 撰寫程式碼，雖能因應當務之急，但在後續的研究上，卻會產生合成規則與模型之間無法重用以及程式愈發難以維護的現象。我們希望專案程式碼可以如同理論探討一般，在未來研究中不斷演化進步。

若欲達到這樣的目標，必須提出一個良好的物件導向架構加以解決。在狀態空間的列舉過程中，合成規則之間可以互相組合，也可以因應不同的 LTS 使用。若後續有新的合成規則或是 LTS 加入時，也可輕易地套用原來的規則。並且在不改變程式既有的行為下，以最少的心力去新增並維護既有的功能，而不用重寫已存在且功能相同的程式碼，以縮短系統開發和維護的時間。

Section 1.2　論文架構

本篇論文的結構如下：在第二章，我們將先簡單介紹此篇論文的研究背景、ArCats 目前的發展以及其它相關的模型驗證工具。

在第三章中針對 ArCats 的合成引擎的功能，如本文限制和安全性質做詳細的解說，並簡單介紹 ArCats 的即時化減技術。第四章的內容著重在於說明我們如何進一步地將目前現有合成引擎重構（Refactoring）成一個具有易於維護，且可演化的物件導向架構。

在第五章中，我們會將重構前架構的程式碼結構，與本篇論文實作後的成果，有關於程式碼結構與系統後續維護的部份，進行比較和評估。並在最後第六章中，針對本篇論文的研究結果做一結論，說明未來可以更進一步的研究方向。

Chapter 2　研究背景

模型驗證（Model Checking）技術常被應用在分析並行或是分散式系統，主要的步驟如下。

首先將欲分析的行程（process）或執行緒（thread），抽象化為同步有限狀態機（communicating finite state machine，CFSM）。當模型被建構出來之後，驗證工具使用列舉的方式（exhaustive exploration）將該系統透過所有可達狀態（reachable states），也就是系統所有可能執行的路徑全部展開。展開後的所有狀態亦可稱狀態空間（state space）。此時可以利用各種搜尋演算法逐一巡視所有的狀態空間，檢驗系統的行為是否符合某些性質，以確保系統開發的品質。

理論上，透過列舉的方式將整個狀態空間的展開，所有可能的錯誤都能在狀態空間中顯現出來。但這樣的做法在空間複雜度的分析上為 PSPACE 的問題，亦即狀態會成指數性成長，造成組態爆炸，當沒有足夠的記憶體進行處理時，自然無法完成後續分析。

Section 2.1　Labeled Transition System（LTS）

Labeled Transition System(LTS)是 CFSM 中一種描述方法，其中包含四個部分：狀態集合（set of states），和一個初始狀態（initial state），狀態間的轉移（set

of transitions)，並以標記(label)的方式表示溝通中的行為(set of actions including internal action τ)。

以 Figure 2-1 為例，此圖表示一個電燈開關切換的行為。初始狀態為 Off，當按下電源開關後（以 Push_button 此行為表示），狀態會由 Off 移至 On。同樣地，當電燈處於 On 的狀態下，執行 Push_button 後，又會回到 Off 的狀態。

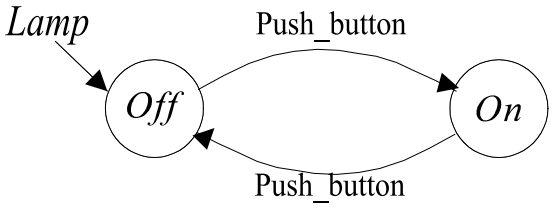


Figure 2-1 A LTS description of a Lamp switch

對於一個行程 *P* 的 LTS 正規定義可以包含四個部分$<\boldsymbol{S, A, \rightarrow, q}>$，

1. $\boldsymbol{S}$：狀態（state）的有限集合。
2. $\boldsymbol{A}$：行為（action）的集合；$\boldsymbol{A}=\boldsymbol{A}' \cup \{\tau\}$，其中 $\boldsymbol{A}'$表示行程 *P* 的溝通行為符號，而且 $\boldsymbol{A}'$不包含內部行為（internal action）τ。
3. $\rightarrow \subseteq \boldsymbol{S} \times \boldsymbol{A} \times \boldsymbol{S}$：表示透過 action 從某個狀態到另一個狀態的轉換（transition），如 $s \xrightarrow{a} s'$可表示為$(s, a, s') \in \rightarrow$。
4. $\boldsymbol{q} \in \boldsymbol{S}$：在狀態集合中的初始狀態。

Section 2.2　平行合成（Parallel Composition）

行程抽象化成 LTS 之後，可以經由平行合成獲得行程所有可能到達的狀態。平行合成操作子（parallel composition operator），以符號‘||’表示。*P* 和 *Q* 兩行程之間的平行合成，即可表示為 *P*||*Q*。在不牽涉行程之間的溝通行為下，基本合成規則有兩種：

$$\frac{P \xrightarrow{a} P'}{P \| Q \xrightarrow{a} P' \| Q}$$

$$\frac{Q \xrightarrow{b} Q'}{P \| Q \xrightarrow{b} P \| Q'}$$

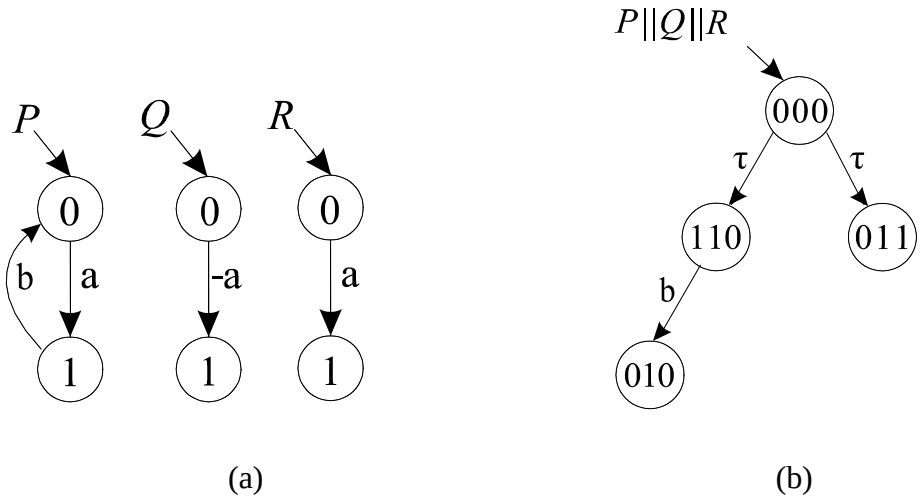
以上兩種規則表示，若 *P* 可以藉由行為 *a* 由狀態 *P* 移至 *P’*，則 *P* 和 *Q* 的平行合成會由原本的狀態 *PQ*，產生行為 *a*，到達一個新的狀態 *P’Q*。同理，若 Q 可以藉由行為 *b* 由狀態 *Q* 移至 *Q’*，則 *P* 和 *Q* 的平行合成會由原本的狀態 *PQ*，產生行為 *b*，到達一個新的狀態 *PQ’*。

在並行系統或分散式系統中，由於行程之間可以互相溝通，故我們介紹兩種溝通模式，分別為 two-way rendezvous 以及 multi-way rendezvous，以描述這類系統行程之間的互動方式。

2.2.1. Two-Way Rendezvous

在 Two-Way Rendezvous 的溝通模式中，同步是以兩個行程之間的一組配對行為來表示，由於同步是以一對為主，為了區別那兩個行為是一組配對，我們在 label 上加註符號，例如 *a* 和*–a*。若兩個行程觸發了相同的配對行為，則此兩者可完成同步並各自達到下一個狀態，否則就會在上一個狀態中停留等候另一個配對行為的觸發。

以 Figure 2-2 (a)為例，行程 *P* 的 *a* 和行程 *Q* 的 *-a* 可以是一組同步的配對行為；行程 *R* 的 *a* 和行程 *Q* 的*-a* 也可以是一組配對。但是當 *P*、*Q*、*R* 都可以觸發 *(a, -a)* 時，*P* 和 *R* 必須競爭以取得同步的權利。意即不是 *P* 可以到達狀態 *1*，就是 *R* 可以到達狀態 *1*。Figure 2-2(b)表示此三個行程平行合成後的結果。符號 *τ* 表示行程之間的內部行為，故 *a* 和 *–a* 這類的同步行為會在平行合成結果的 *P||Q||R* 中被隱藏簡化成 *τ*，不為外界所觀察得到。



(a) (b)

Figure 2-2 Parallel composition in Two-way Rendezvous

Two-way rendezvous 可以下列的數學式表示：

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{-a} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

此類的 LTS 以 CCS（Calculus of Communicating System）[24][25]為代表。CCS 溝通模式常發生於軟體中，也是本篇論文及 ArCats 工具的基礎。

2.2.2. Multi-Way Rendezvous

在 Multi-Way Rendezvous 溝通模式中，當多個行程欲進行平行合成時，一個同步的行為可能存在於兩個以上的行程裡。若某行程必須等待同步行為，例如 *a*。則該行程必須等待所有擁有行為 *a* 的行程都觸發此行為 *a* 後，才可完成同步進入下一個狀態。以 Figure 2-3(a)為例，行程 P、Q、R 在狀態 0 的時候，必須所有行程的 a 的皆被觸發，行程 P、Q、R 才能完成同步轉移至狀態 1。Figure 2-3(b)顯示三個行程平成合成的結果。

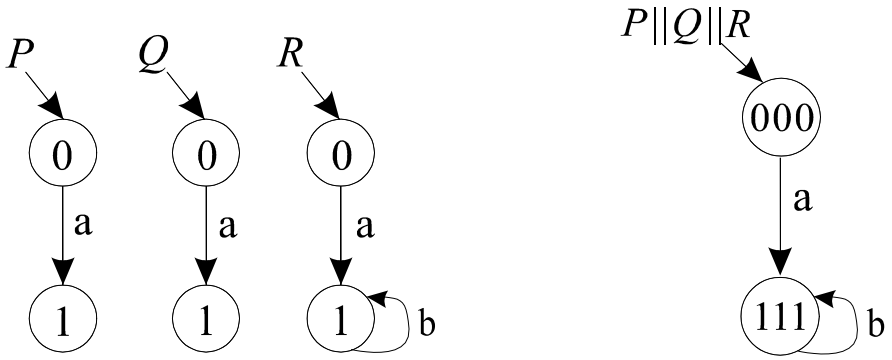


Figure 2-3 Parallel composition in Multi-way Rendezvous

Multi-way rendezvous 的可以下列數學式表示：

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'}$$

此類的 LTS 以 CSP（Communicating Sequential Process）[3]為代表。在後續的研究中，我們也預計將此種溝通模型加入 ArCats。

Section 2.3　狀態空間的展開（State Space Enumeration）

狀態空間的擴展，是所有模型驗證工具的主要重點之一。將抽象成 LTS 的行程執行時的狀態空間的展間，也就是以所有可能的執行路徑去檢驗程式是否違反了某些性質，此想法是十分直接和簡單的。在集合的運算上，就是以卡氏積（Cartesian product）運算得到所有 LTS 的狀態。

假設一個系統裡有 n 個 LTS，每個 LTS 有 m 個狀態，即計算此系統的最多產生 m^n 個狀態，這種演算法的空間複雜度是 PSPACE。一般來說，較大的系統在驗證工具展開狀態空間的過程中，很快就會耗盡系統記憶體資源而無法進行驗證。下一節簡單地介紹幾種知名的 Reduction 和 Minimization 方法。

2.3.1. Reduction：狀態空間的化減

Reduction 的演算法是為了在列舉狀態空間時，可以儘量減少狀態的產生。其中最有名的是 partial order reduction[13][17]和 symmetry reduction[2]。Partial

order reduction 的原理是利用狀態空間中有許多 state 及 transition 是由與許多不相關的事件交互運算（interleaving）得來的，而這些事件的交互運算順序若與要驗證的性質無關，那麼我們在狀態空間中展開其中一種順序即可。而 symmetry reduction 是狀態空間在展開的過程中，通常有許多 state 和 transition 彼此形成某類對稱關係，利用彼此對稱關係結合成一個狀態，以減緩組態爆炸。但狀態結合會有資訊消失（information lost）的問題。如何在驗證過程中將資訊回復是這類研究上待解決問題。

2.3.2. Minimization：狀態空間的最小化

展開後的狀態空間可以利用 minimization 的技術，以得到一個狀態空間較小，但行為等價的系統模型，較廣為人知的是 bisimulation minimization。Milner[25] 定義了兩種 bisimulation relation，分別是 strong bisimulation 和 weak bisimulation。strong bisimulation 由於限制過於嚴格，實用性不高。weak bisimulation（又稱為 observation equivalence relation），用來判定兩個並行系統外部行為是否一致，有較高的實用價值；但利用此演算法在縮小狀態空間的過程中，會產生資訊消失的問題，例如，liveness property 會在縮小化的過程中被破壞，反而違反檢驗軟體特性的本意。Van Glabbeek[26]因而提出 branching bisimulation 技術，它比 weak bisimulation 稍微嚴格一點，但其優點在於狀態空間化減後，仍可保留系統模型的某些特性。

Section 2.4　局部性分析（Compositional Analysis）

局部性分析[12][31][27][28]是針對總體性分析所面臨的組態爆炸問題，提出試圖解決的方案。首先利用 LTS 抽象化目標系統所有行為，再給予一個適當的子系統階層架構來進行合成，在總體性分析中，系統模組性是不列入考慮的。但在局部性分析裡，系統模組性卻相當關鍵。在子系統的狀態空間展開的同時，利用 reduction 和 minimization 的技術，以求得另一個狀態數目更少但行為模式相同的 LTS 來代表此子系統。此 LTS 描述子系統的外部行為或介面行為，隱藏複雜的內部行為。我們稱這個一個 LTS 為介面行程（interface process）。

如果每一個子系統都能夠找到它們的介面行程，我們就可以利用狀態數目較少的 LTS 進行交互運算（interleaving），所得的狀態空間展開就可能可以更小。一般來說，好的模組性系統對局部性分析是有極大幫助的。不過在介面行程合成時亦需要找到一個適合的合成階層（composition hierarchy）。不好的合成階層會增加許多不必要的狀態，反而對展開不利。在理想的情境中，我們找到適當的階層並且每一個子系統的介面行程都不大，然後藉由 divide and conquer 逐步地由下往上（bottom-up）合成分析，最後達到分析整個系統的目的。

局部性分析也有一些待克服的問題。例如，有許多正確性性質（correctness property）會在 minimization 的過程中喪失，而且驗證工具在發生一個違反性質的行為時，要找到一個完整反例（counterexample）也不如在總體性分析中簡單。

Section 2.5　ArCats 與其它模型驗證工具

2.5.1. SPIN

SPIN[15][17]是由 AT&T 所開發的通訊協定與軟體驗證工具，SPIN 所開發的 PROMELA 模型語言（modeling language），是提供使用者用來描述系統模型之間的溝通行為。SPIN 將 PROMELA 讀入後，再轉換成有限狀態機的形式，進行各種分析，確認行程之間是否能正確地溝通。

SPIN 可以驗證的性質有 deadlock、safety property、liveness property、assertion、未到達的狀態（unreachable state）等。在狀態空間的列舉上，它使用 Depth First Search(DFS)演算法，窮舉所有可能的狀態，我們稱這種方式為 explicit enumeration，列舉時所需要的記憶體會和狀態的個數成正比。為了避免組態爆炸，減少記憶體使用，SPIN 亦提供了數種演算法，將狀態壓縮後再予以儲存[16]。

2.5.2. FC2Tools

FC2Tools[5][6]是 INRIA 所開發的局部性分析驗證工具，本身有專有的 FC2 格式語言。由於 FC2 格式不容易閱讀使用，所以它另備圖形編輯器 AUTOGRAPH，以供使用者建構系統模型。

FC2Tools 可以驗證的性質有 deadlock 和 liveness property 等。在狀態空間的

展開上，FC2Tools 提供兩種列舉方式以供使用者選擇。一種是直接的將 state 和 transitions 展開（即 explicit enumeration）。另一種是採用 BDD(Binary Decision Diagram) [1]技術來儲存狀態空間，以減少記憶體的消耗，我們稱此技術為 implicit enumeration。而在展開狀態的過程中，可以依使用者的需求，分別予以不同的最小化（如 strong、weak 和 branch bisimulation）。

2.5.3. ArCats

ArCats[32]是 Cheng 率領開發的局部性分析工具。使用 PROMELA 做為模型建置語言，現階段可以分析 deadlock、safety property、liveness property 等，並且提供模型重構以得到較佳的子系統架構，運用 context constraint 簡化狀態空間，使用 on-the-fly branch bisimulation minimization 技術最小化狀態空間。除了模型驗證的功能外，ArCats 另提供模擬（Simulation）的功能供使用者檢驗系統模型，避免因記錄大量的狀態空間而造成實體記憶體不夠的問題。本篇論文重構的重點在於 ArCats 合成引擎的架構改良。將程式碼耦合的部分提出為合成規則引擎（Rule Engine），原合成引擎只留下單純列舉狀態空間的功能。而使用者在使用合成引擎列舉狀態空間時，可以自由從合成規則引擎中選擇欲檢驗的性質，狀態簡化的方式，以及使用的 LTS 型態，讓整個系統架構更具彈性和擴充性。

Chapter 3　ArCats 合成規則分析與即時化減技術

本篇論文的主題為 ArCats 合成引擎的重構，而重構的關鍵在於，程式碼需要有一清晰合理的邏輯架構，以應各種可能所需要的處理程序。故我們需要先將重構的重點，意即合成引擎中合成規則的運作，做一個有系統的分析。試圖將在看似各個獨立的合成規則中，求得一個能夠彙整的邏輯結構，以達到本篇論文的目的：將合成規則具獨立性地拆解，以達到日後合成規則可重組，程式碼可重覆使用的目的。本章將以步驟性的方式，循序說明並分析合成引擎運作時所依循的合成規則，此亦為重構 ArCats 合成引擎，將其改良為合成規則引擎時，所依循的程式設計邏輯基礎。

Section 3.1 中我們將介紹基本的合成規則，如同總體性分析中將所有可能狀態全數列舉的做法。在 Section 3.2 裡，我們在列舉狀態時加入本文限制的功能，使得平行合成後的狀態數量可合理的減少。Section 3.3 介紹安全性質的檢驗實作方式。由於合成引擎與 ArCats 的最小化技術有著密切的合作關係，所以在最後 Section 3.4 我們將簡要描述 ArCats 所提供的即時化減技術具備了那些選項以供合成引擎列舉狀態時依需求使用。

Section 3.1　基本合成規則

狀態平行合成的基本規則，已於 Section 2.2 中說明。但在局部性分析中，我們會忽略子系統的內部行為，只在乎子系統和外界的溝通。而該子系統與其它子系統之間的互動，也就是可能配對的同步行為，我們必須先提出至外顯集合(export set, 簡稱 EXP)中。

export set 的產生方式如下：首先將外界其它子系統所有的 LTSs 的 action，逐一與子系統 LTSs 中的 action 所比對，如果子系統中的 LTSs 有 action 可以和外界的 LTSs 做同步的配對行為，則將此 action 記錄在 export set 中。這樣的功能在 ArCats 中，我們可以使用 *genExport* [33]自動產生出 export set，並儲存成.exp 的檔案格式供後續分析時讀入。

在 ArCats 的局部性分析中，我們不會將所有 LTSs 的 states 和 transition 全數展開，此舉形同總體性分析。我們使用合成限制規則（Restriction Rule）[25]，以下列的 Com1 數學式表示。Com1 取代了 Section 2.2 中所描述的平行合成基本規則，也就是說，只有在 export set 中的 actions，亦即會與外界產生互動的行為，我們才會將其列舉展開。

$$\text{Com 1}\quad \frac{P \xrightarrow{c} P'}{P \setminus EXP \xrightarrow{c} P' \setminus EXP} \quad (c \in EXP)$$

而由於目前 ArCats 是以 CCS based 模型進行分析，故我們使用 Two-way Rendezvous 表示法，在合成子系統時，隱藏子系統中的內部行為。

$$\text{Com 2} \quad \frac{P \xrightarrow{l} P', Q \xrightarrow{-l} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

Section 3.2 本文限制（Context Constraint）

在局部性分析中，窮舉所有子系統的狀態時，常常會出現有多餘的狀態產生。這種狀態是由於子系統被分割後各別合成所導致的情況。當子系統階層式的往上與其它子系統合成時，才會發現子系統中有些狀態是無法到達的。故在子系統的狀態展開時，這類狀態不僅佔用了記憶體空間，對驗證分析也無實質的意義。因此愈早移除這類多餘狀態和移轉，對於分析會更有效率。故 Larsen 和 Milner[20]提出可使用本文限制（Context Constraints）的技術，以減少子系統所需要擴展的狀態空間。

3.2.1. 環境介面行程(Environment Interface Process)

在 Cheung 和 Kramer[29]的研究中提到，在子系統列舉狀態時，加入環境介面行程（Environment Interface Process, EIP）一併合成，多餘的狀態將可被減化。而環境介面行程可以經由下列三個步驟產生：

（1）簡化環境行程（environment process）：在整個系統中，不包括在此子系統內的行程，皆為此子系統的環境行程。但不是每一個環境行程都會和此子系統有本文限制的關係。我們會將所有的行程與 export set 中的 actions 逐一比對，如果該環境行程有和 export set 擁有相同且兩個以上的 actions，才是和該子系統有本文限制關係的環境行程。

接著我們會從這些篩選出來的環境行程中，刪除與該子系統無關的 transitions，也就是與 export set 無關的 actions，都會被移除。

（2）將所有環境行程轉換為可決定性（deterministic）：簡化後的環境行程，必須是可決定性的，如果為不可決定性的（non- determine）則必須轉換為可決定性的 LTS。不可決定性的 LTS 是指在同一個狀態中，兩個不同的 transition，卻有著相同 label 的 actions。可決定性的 LTS 中，每個狀態中的 action 不會重覆。

（3）合成所有環境行程：經由步驟(2)所得到可決定性的環境行程，若有兩個以上，則需合成為一個。最後合成的結果即為環境介面行程。在 ArCats 中，此功能已被實作為 *genInterface* [33]，在子系統合成之前，先經由 *genInterface* 得到 EIP。然後再將 EIP 加入子系統一併進行狀態的合成，展開後所產生的新行程，會移除掉無法到達的狀態，減緩組態爆炸的發生。

3.2.2. 本文限制的合成規則

本節將以 Figure 3-1 說明如何運用 Section 3.1 中的 Com1 和 Com2，以及本文限制的合成規則（以 Com3 表示）這三種合成規則，進行具有本文限制功能的局部性分析。

Com 1 $$\frac{P \xrightarrow{c} P'}{P \setminus EXP \xrightarrow{c} P' \setminus EXP} \quad (c \in EXP)$$

Com 2 $$\frac{P \xrightarrow{l} P', Q \xrightarrow{-l} Q'}{P \,||\, Q \xrightarrow{\tau} P' \,||\, Q'}$$

Com 3 $$\frac{P \xrightarrow{l} P', EIP \xrightarrow{-l} EIP'}{P \,||\, EIP \xrightarrow{l} P' \,||\, EIP'}$$

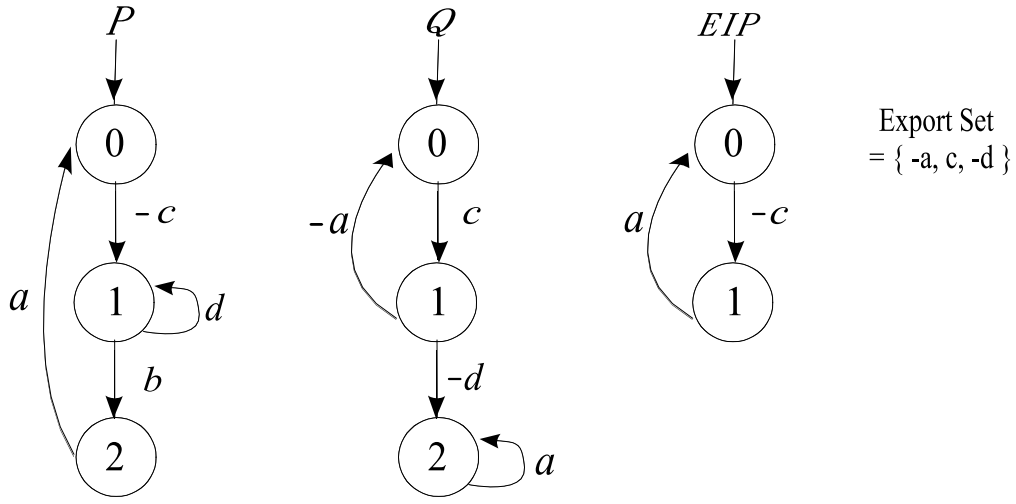


Figure 3-1 Compose processes *P* and *Q*, and *EIP*

平行合成的初始狀態為*(000)*，*(000)*狀態中的第一個 transition 上的 label 為 *-c*，我們將*-c* 這個 label 和狀態*(000)*逐一做 Com1、Com2 和 Com3 三種合成規則的檢查。由於*-c* 不存在於 export set 中，故不符 合 Com1。接著 進行 Com2 的檢查，*(000)*狀態中的下一個 transition 的 label 為 *c*，*-c* 和 *c* 是一配對行為，經由 Com2 後將此配對行為隱藏為 *τ*，所以(*P* || *Q*)的平行合成結果會由原本的狀態*(000)*，經由行為 *τ*，而到達新狀態*(110)*。我們將此新狀態*(110)*展開後，並將其存入 vector 中，等待之後繼續展開此新狀態。此時(*P* || *Q*)的結果如 Figure 3-2 表示。

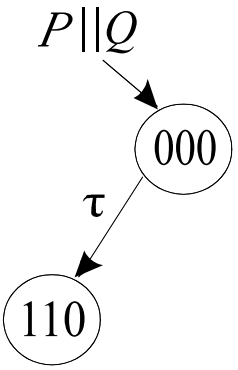


Figure 3-2 Composition of (*P* || *Q*) by Com2

*(000)*狀態的第二個 transition 的 label 為 *c*，而 *c* 存在於 export set 中，故符合 Com1。由於此例中加入了本文限制的合成規則 Com3，所以我們繼續做 Com3 的檢查，確認此處的 *c* 是否會受到本文限制而無法展開。

此時合成引擎會到 *EIP* 中檢查是否有可以產生配對行為的 label，*EIP* 目前的狀態為 *0*，狀態 *0* 有一 transition 的 label 為 *-c*，且可以藉由*-c* 由狀態 *0* 轉移至狀態 *1*，表示 *c* 這個 action 沒有受到本文限制，所以經由 Com3 的合成規則，(*P* || *Q*)

的平行合成結果會由狀態*(000)*會產生一個 label 為 *c* 的 transition 而到達新狀態 *(010)*，如 Figure 3-3 所示。

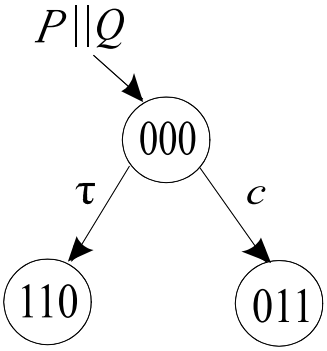


Figure 3-3 Composition of (*P* || *Q*) by Com1 and Com3

此時*(000)*狀態中的所有 transitions 都已檢查完畢，所以我們從 vector 中提出新狀態*(110)*繼續拓展，步驟皆如同上述方式，對每個狀態中的每個 transition 逐一檢查上述三種合成規則，若可以符合其中一項規則且沒有受到本文限則，即可得到新的 action 和新狀態。最後的合成狀態以 Figure 3-4 表示。

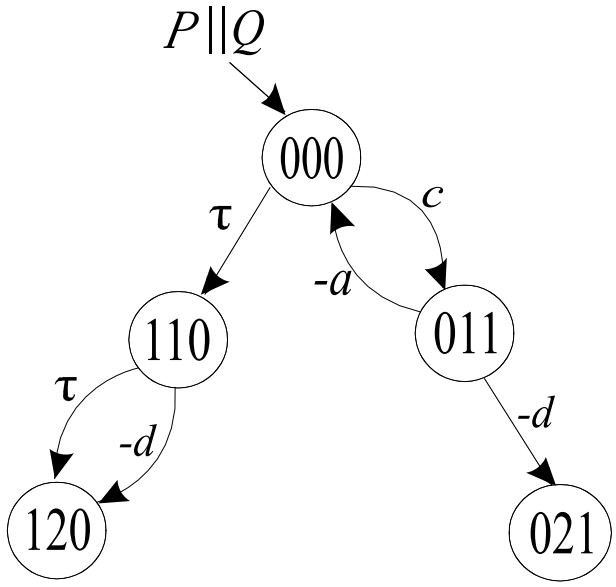


Figure 3-4 Composition of (*P* || *Q*) with Context Constraint

如果(*P* ‖ *Q*)沒有使用本文限制的合成規則，則平行合成的結果如 Figure 3-5，會產生多餘的移轉 *c* 以及*-a*，以及多餘的狀態*(10)*，以灰階表示。由於這類不可到達的狀態，對後續的子系統合成並無任何幫助，所以應使用本文限制的合成規則儘早將其移除。

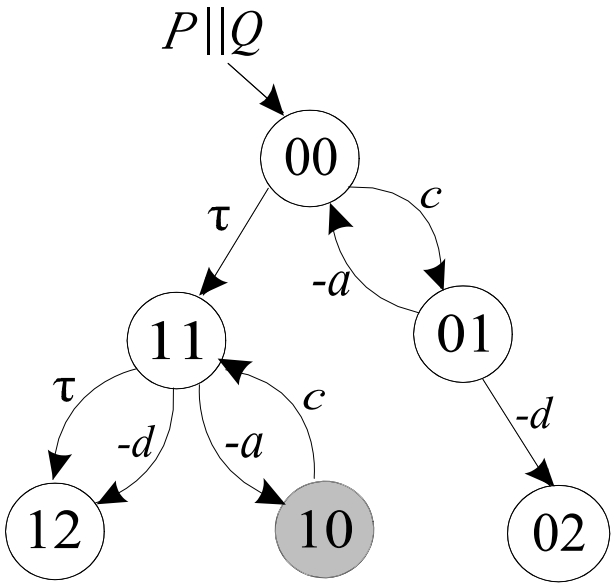


Figure 3-5 Composition of (*P* || *Q*) without Context Constraint

Section 3.3　安全性質（Safety Property）

軟體驗證工具所要檢驗的安全性質，即為檢查程式進行時，不良的狀態是否永遠不會發生。

ArCats 進行安全性質檢驗的技術[34]，採用的是 image automata 這類自動狀態機。image automata 是由 property automata 所衍生而來。一般檢驗的系統性質

稱為 property automata，如果子系統是正確執行，則執行順序會符合 property automata 所描述的特性。當子系統執行順序無法符合 property automata 時，image automata 就會進入我們不願意發生的狀態，例如 deadlock 或 mutual exclusion，我們將這樣的狀態簡稱為 π。

以 Figure 3-6 為例，當系統加入安全性質的檢查時，如果行程是正確執行，若產出的合法字串應為 *a,b,b,b…*，符合 Figure 3-6(a)所描述的特性。若產生字串為 *a,b,b,b,b,a* 時，即表示系統違反安全性質，進入不良狀態 π。

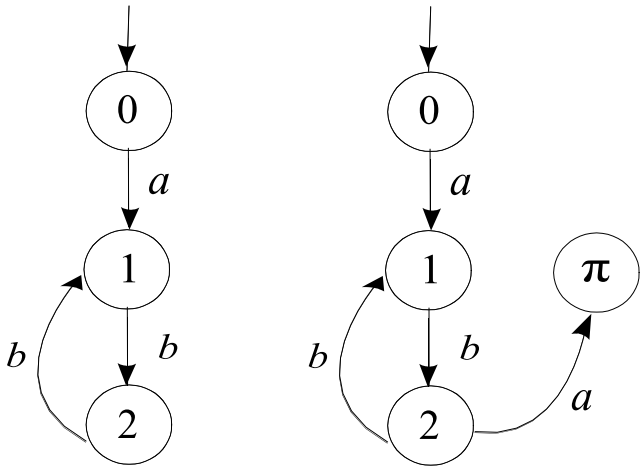


Figure 3-6 (a) Property Automata (b) Image Automata

3.3.1. 安全性質的合成規則

本節將以 Figure 3-7 說明如何運用 Section 3.1 中的 Com1 和 Com2，以及安全性質的合成規則（以 Com4 表示）這三種合成規則，進行具有檢驗安全性質功能的局部性分析，我們將檢驗安全性質的 Image Automata，以 *SP* 表示

Com 1 $$\frac{P \xrightarrow{c} P'}{P \setminus EXP \xrightarrow{c} P' \setminus EXP} \quad (c \in EXP)$$

Com 2 $$\frac{P \xrightarrow{l} P', Q \xrightarrow{-l} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

Com 4 $$\frac{P \xrightarrow{a} P', SP \xrightarrow{a} SP'}{P \parallel SP \xrightarrow{a} P' \parallel SP'}, \frac{P \xrightarrow{-a} P', SP \xrightarrow{a} SP'}{P \parallel SP \xrightarrow{-a} P' \parallel SP'} \quad P \notin \pi$$

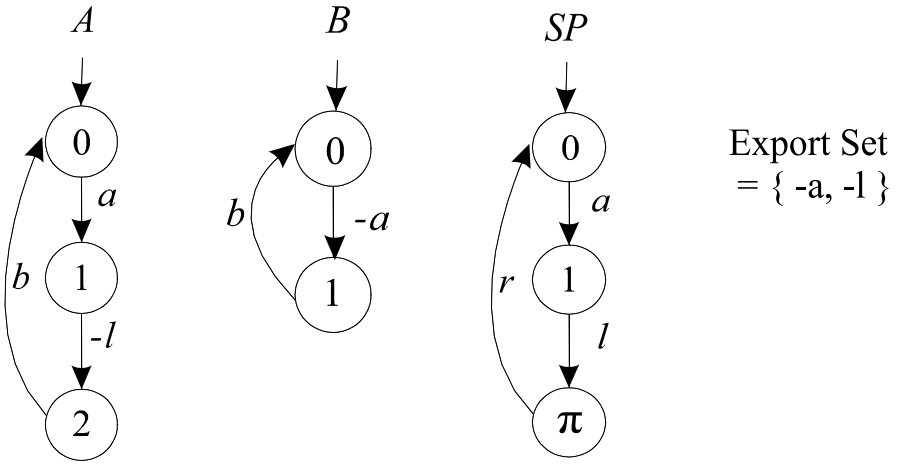


Figure 3-7 Compose processes *A* and *B*, and *SP*

平行合成的初始狀態為*(000)*，*(000)*狀態中的第一個 transition 上的 label 為 *a*，我們將 *a* 這個 label 逐一做 Com1、Com2 和 Com4 三種合成規則的檢查。由於 *a* 不存在於 export set 中，不符合 Com1，所以我們繼續檢查 Com2 的合成規則。

狀態*(000)*的第二個 transition 為*-a*，和 *a* 為一配對行為，故符合 Com2，所以我們將此內部行為以 *τ* 表示。由於此例中我們加入了安全性質的檢驗，故我們會

到 *SP* 中去檢查此步驟是否會到達不良狀態 π。

此時 SP 中狀態為 *0*，且在此狀態上有一個 transition 的 label 為 *a*，和我們欲檢查的行為 *a* 相同。*SP* 可藉由 *a* 由狀態 *0* 到達狀態 *1*。所以我們可以運用 Com4 的規則，使得(*A* || *B*)的平行合成結果會從狀態*(000)*，經由 transition 的 label τ，到達新狀態*(111)*，如 Figure 3-8。由於此新狀態不會產生不良狀態 π，所以我們將此新狀態*(111)*加入 vector 中，以待後續繼續拓展。如果產生的新狀態包括 π，我們會顯示在(*A* || *B*)的平行合成結果中，但此狀態不會再加入 vector，因為包括 π 的狀態我們就不會再繼續展開。

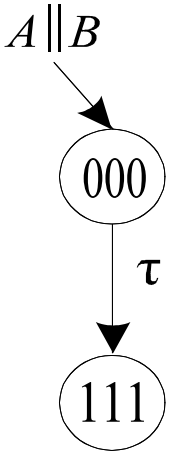


Figure 3-8 Composition of (*A* || *B*) by Com2 and Com4

由於狀態*(000)*尚未有 transition 沒檢查完畢，所以我們繼續檢查第二個 transition 上的 label *-a*。*-a* 存在於 export set 中，符合 Com1。於是我們繼續進行安全性質合成規則 Com4 的檢查，此時 *SP* 的狀態為 *0*，且狀態 *0* 上有一個 transition

的 label 為 *a*，可以使狀態 *0* 到達狀態 *1*，且狀態不包含 π。所以經由 Com4 的合成規則，(*A* || *B*)的平行合成結果會從狀態*(000)*，經由 transition 的 label -*a*，到達一新狀態*(011)*，如 Figure 3-9 所示。

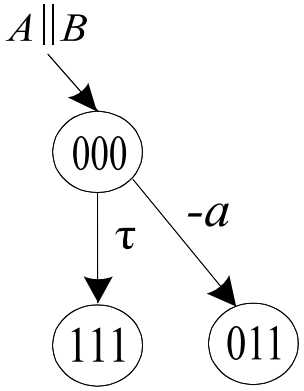


Figure 3-9 Composition of (*A* || *B*) by Com1 and Com4

利用上述三項合成規則反覆檢查每一個狀態的 transition 後，(*A* || *B*)的最後平行合成結果如 Figure 3-10 所示，其中有一狀態*(21π)*，表示該子系統的執行順序會進入不良狀態，違反安全性質。

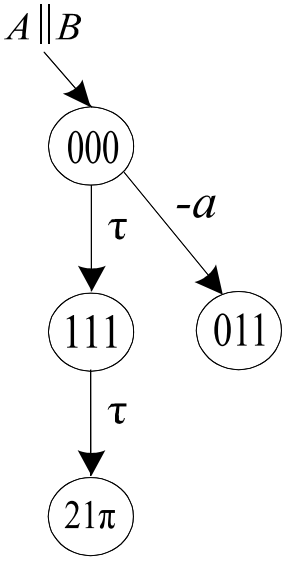


Figure 3-10 Composition of (*A* || *B*) with checking Safety Property

Section 3.4　即時化減技術

在上述許多說明中，如隱藏內部行為，以階層式的概念驗證子系統以及使用本文限制消除多餘狀態等等，都是局部性分析為了避免可能的組態爆炸問題所提出的解決方案。然而，局部性分析若是系統的階層架構不良，或是切割後的子系統仍舊太大，還是可能讓局部性分析產生組態爆炸導致失效。

ArCats 採用 Branching Bisimulation Minimization[26][18]的化減技術，以得到一個組態空間較小但表現行為等價的系統模型。然而，局部性分析在建構子系統階段就可能會面臨組態爆炸的問題，如果等到子系統建置完畢後再進行化減，將無法發生最小化的功用即已分析失敗。故 ArCats 提出即時化減技術—On-the-fly Branching Bisimulation Minimization[18]，在子系統狀態展開時，根據使用者所選定的參數進行即時化減，讓內部行為所佔用的記憶體空間，在狀態展開的過程中即可被釋放出來，減少記憶體的使用，以提高局部分析的可行性。

故 ArCats 即時化減技術的實作與合成引擎有著緊密的結合，ArCats 提供三種即時化減的參數，以供使用者在合成狀態時依需求選擇。

第一種是根據使用者所給定的狀態數目。例如使用者指定的狀態數為 10 個，當合成引擎中的合成的狀態數目已達到 10 個，就會進行即時化減；如果合成的狀態數目尚未超過指定的數目，合成引擎將繼續進行狀態的合成。

第二種是根據記憶體使用量的百分比來決定是否即時化減。如果在記憶體的使用仍十分充裕的情況下進行化減，只會增加分析時的 overhead，所以使用者可以自訂使用的百分比，記憶體的使用到達此百分比時才執行最小化。若使用者不指定百分比，系統預設值為 98%。

由於目前 ArCats 狀態的展開是以 Breadth-First 實作，故第三種選擇即是依照 Breadth-First 每一層展開的狀態，執行即時化減。

ArCats 的即時化減技術雖與合成規則無關，亦非此篇論文進行重構的內容，但由於即時化減的功能與合成引擎有頻繁的合作，每當合成引擎列舉狀態達到一定的程度（例如狀態數目已達到 10 個），就會將合成中的結果透過 method 的呼叫以進行即時化減。故後續我們實作合成規則引擎時，也會與此部份牽連甚深，故在此節簡單地予以介紹。

Chapter 4　重構 ArCats 局部性分析工具

本章的架構如下，我們在 Section 4.1 中說明重構前的合成引擎實作方式，並對於後續系統繼續開發所面臨的問題作一總結。在 Section 4.2 中，我們會針對先前架構的缺失提出解決方案，並實作合成規則引擎且對改進最小化引擎，以解決原先合成引擎中造成程式碼耦合，難以擴充重用的問題。由於我們的重構目標是希望 ArCats 局部性分析工具能擁有演化性的架構，以因應後續研究中，不同的合成規則及 LTS 的加入，所以在 Section 4.3 此節中，我們將以重構完成的架構圖說明如何達到我們重構的目標。

Section 4.1　重構前的合成引擎架構

ArCats 的合成引擎提供了狀態平行合成、安全性質檢驗、本文限制與即時化減技術等幾種複合的功能，使用者可照需求自行選定參數予以設定，參數如 Table 2 表示。我們並以 Figure 4-1 簡單地說明重構前的合成引擎中，程式呼叫的決策流程。

-c	平行合成時加入 EIP 以進行本文限制
-s	平行合成時加入 SP 以進行安全性質的檢驗
-b1	平行合成時會根據指定的狀態數目進行即時化減
-b2	平行合成時會根據指定的記憶體用量%進行即時化減
-b3	平行合成時會根據 BFS 所展開的每一層進行即時化減

Table 2 The parameter of composition tool

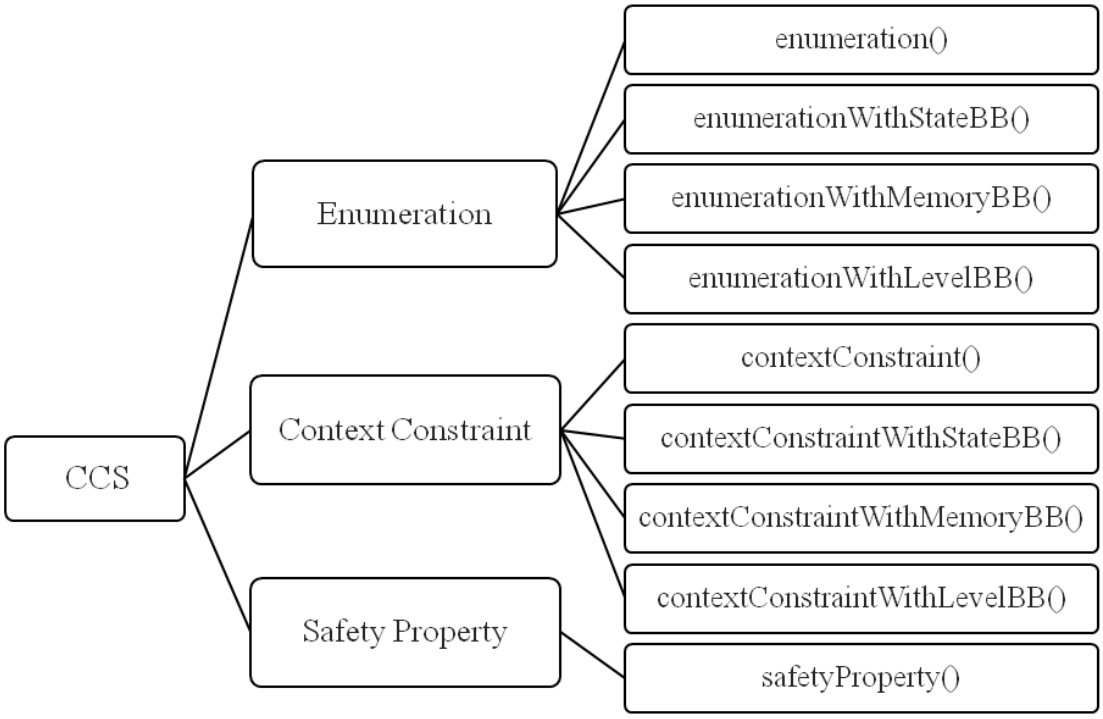


Figure 4-1 The call graph of composition tool before refactoring

Figure 4-1 樹狀似的 call graph 的程式呼叫步驟說明如下：第一層表示系統判斷讀入的 LTS 型態是否為 CCS。若為 CCS，再檢查使用者是否設定 Context Constraint 或是 Safety Property 的參數，如果有，則依參數決定要呼叫 Context Constraint 或是 Safety Property 的 method，若皆未設定兩者的參數，表示只需單純進行平行合成狀態的展開，此時會去呼叫 Enumeration 的 method。

確認完合成方式後，最後 再依使用者給定的參數判斷是否要執行即時化減技術，若使用者未指定任何即時化減的參數，意即合成時不需最小化。以使用者輸入的參數為 compose A.ccs B.ccs –c i1.ccs –b3 為例，表示使用者選擇在進行($A \| B$)的平行合成時加入本文限制的檢查，使用的 EIP 名稱為 i1.ccs。在平行合成時，

合成引擎會隨著 Breadth-First 展開每一層狀態，進行即時化減。所以依照此例所給定的參數，程式經由巢狀 if-else 的條件述句分析後，最後會呼叫 ContextConstraintWithLevelBB() 此 method，以進行局部性分析。

由以上敘述以及 Figure 4-1 的示意圖可知，由於要經過層層判斷方能確認使用那一個 method 進行局部性分析，程式中充滿許多 if-else 的條件判斷式，以達成逐步確認的效果。所以我們可以對重構前的架構做一總結如下：

（1）非物件導向架構，程式難以擴充或重用

在非物件導向的架構下，最大問題在於程式碼無法重覆利用。假設我們要在檢驗 Safety Property 時加入即時化減的功能，我們必須實作三個新的 method：分別為 SafetyProperyWithStateBB()、SafetyProperyWithMemoryBB() 以及 SafetyProperyWithLevelBB()。若將規模更進一步擴充，在 ArCats 中新增 CSP based 的 LTS 進行局部性分析，則原有的 Context Constraint 和 Safety Property 的合成規則程式碼更如同虛設，必須將 Figure 4-1 中九個 method 全數重寫。

（2）重複的程式碼

Figure 4-1 樹狀式的 call graph 的呼叫方式，顯示原有的合成引擎中有許多重複的程式碼散落於各個 method 中。重複的程式碼表示 method 之間有相同的程式結構，亦會造成程式碼的耦合。耦合的程式碼不僅難以維護，在增

加了新的功能後，很可能還會使得既有程式碼出現錯誤，增加 debug 的負擔。舉例來說，當我們要修改某一項功能時（例如即時化減技術），由 Figure 4-1 的示意圖中可知，我們必須更動到六個 method。而程式碼需要更改的地方愈多，程式出現錯誤的可能就愈高。而且修改程式碼時很容易使得既有的功能，如 Enumeration 或 Context Constraint 受到影響，產生無法預料的錯誤。

（3）重複的邏輯指定

在確定要使用那一個 method 之前，所有 method 都要經過相同的邏輯篩選程序，首先確認是否為指定的 LTS 型態，再確認合成規則，最後確認即時化減的方式。當我們新加入 method 或修改原有功能時，除了實作既有的內容外，還需要到專案多處進行 if-else 的條件式的修正與補進，只為了確保程式能夠正確的呼叫。因此過多的條件判斷式不僅使得程式碼難以閱讀，要維護的 method 和成本也會增加。我們希望可以將相關的邏輯處理集中，閱讀程式碼時就無須在各種條件判斷句之間來回切換，造成理解上的阻礙。

Section 4.2　重構後的合成引擎架構

在 Chapter 3 中，我們已詳細說明各個合成規則以及合成時的步驟與順序。在本節中，我們將依循這樣的程式邏輯實作出合成規則引擎。並針對合成引擎先前架構所面臨的問題，提出解決方案。

4.2.1. 消除重複的程式碼：合成引擎的改良

由 Figure 4-1 中可以知道，程式間一個很大的問題在於，許多 method 之間的程式結構其實大多相同，也就是系統中其實有許多重複的程式碼，分散出現在不同的 method 中。而在各別的 method 裡，只是在某些地方增加了些許不同的功能。以沖調咖啡的例子來說，在調味咖啡的步驟中，A 口味的咖啡要加一匙奶精兩匙糖，B 口味的咖啡要加一匙奶精但不加糖，泡兩種咖啡的步驟幾乎相同，差別只在於最後加幾匙糖的步驟。

在重構前的合成引擎中，我們也可以發現這樣的關係。例如 enumeration()和 enumerationWithStateBB()這兩個 method，後者的程式結構幾乎和前者完全相同，也就是說，兩者都使用 Com1 和 Com2（詳見 Chapter 3）進行狀態的展開；不同處只是在於 enumerationWithStateBB()在合成狀態數到達使用者所指定的數目時，合成引擎會交由最小化引擎進行即時化減。在 contextContraint()中，也僅是在基本的合成規則，即 Com1 和 Com2 中，加入了本文限制的合成規則 Com3，所以在程式結構上，contextConstraint()也和 enumeration()有著諸多類似之處。

有許多方式可以消除程式中重複的程式碼[19][23]。在本研究中，我們引入 *Template Method Design Pattern*[8][23]於重構後的架構中。我們在父類別中定義完整的演算法基礎步驟，並組織成適當的 abstract method，細節則留給各個子類別，

依照個別的目的去填補或改寫。

Template Method Design Pattern 的架構提供的好處在於，演算法的不變之處只需要撰寫一次，其它會變動之處留待子類別完成。而子類別中重複的行為，也可以交由父類別的 method 去實作。由此可知，*Template method* 除了可以消除重複的程式碼，亦是一種相當實用的程式碼再利用方式。

依照這樣的 design pattern，我們設計了一個名為 Compose 的類別。此類別也將用來取代原有的合成引擎，新的合成引擎只留下原有單純狀態展開的功能，並由 enumeration()這個 method 實作。其餘可能變動的部份，如合成規則的使用和即時化減的方式，我們交由 Rule（即重構後的合成規則引擎）和 Minimize（即原有的即時化減技術）這兩個繼承架構執行。當合成引擎中狀態展開時需要進行合成規則的檢查，或是即時化減的技術時；enumeration()會分別去呼叫 Rule 和 Minimize 的物件以執行合成規則的檢查和最小化。Compose 的類別圖如 Figure 4-2 所示。而 Rule 和 Minimize 的實作方式，我們將在下節詳細說明。

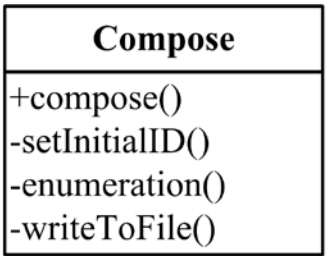


Figure 4-2 Class diagram of Compose

4.2.2. 多型取代條件式：

合成規則引擎實作和最小化引擎的改良

對於合成規則的運用方式，重構前的合成引擎使用許多 if-else 的條件式輔助判斷，關於此點，我們可以使用物件導向的多型架構予以解決[23]。

使用多型意味著必須有一繼承架構，所以我們必須在各個看似獨立的合成規則之間，歸納出相似的部分，才能將重複的程式碼提昇至父類別（superclass），各個子類別（subclass）再依據其特性的不同，分別進行不同的實作。

由於各種不同的合成規則的操作，背後的理論探討和要求的條件皆不盡相同，乍看之下，的確難以提鍊出這些合成規則中的共有特質。倘若我們先不著重各個合成規則運作時的細節，我們可以歸納出一個結論，也就是所有的合成規則之間，其實都有著相同的邏輯行為。若把每個合成規則視為一個 black box，合成規則的實作方式被隱藏在此 black box 中，而每一個合成規則的輸入，都會是一個既有的 state 和 transition；每一個合成規則的輸出，都會是一個新的 state 和 transition，如 Figure 4-3 所示。

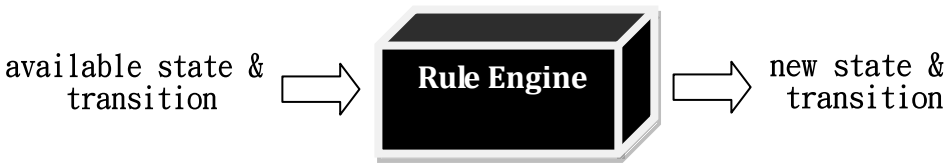


Figure 4-3 The operation of Rule Engine

也就是說，當我們輸入一個 state 和 transition 交由合成規則進行分析時，該合成規則會依照其本身理論的運作方式，提供給平行合成的結果可能的新 state 和新 transition，或是顯示這次檢驗的 state 和 transition 對平行合成的結果不會造成影響，也就是無法產生新的 state。

所以我們可以在看似毫無關聯的合成規則中，提出邏輯相同的部分，制定成為父類別的一個 abstract method。此 abstract method 我們命名為 doRuleCheck()，如同 Figure 4-3 中 black box 操作的概念，我們將子類別的合成規則，隱藏在對 doRuleCheck()的不同實作中，而唯一的共通點，就是 doRuleCheck()的產出必為一新 state 和新 transition。

我們以 Figure 4-4 說明重構後新加入的合成規則引擎繼承架構。由名為 Rule 的 class 做為父類別，在 Chapter 3 中描述的不同的合成規則，例如 Context Constraint，就建立成子類別去繼承 Rule。子類別會分別實作父類別的 abstract method，當呼叫該物件的 doRuleCheck()時，就表示要進行該子類別的合成規則檢查。往後若要加入新的合成規則，甚至是 CSP based LTS 的 multi-way rendezvous 的溝通模式，都可以直接在此繼承架構下，新增一個新的子類別，再實作 doRuleCheck()此函式即可。這樣的設計不僅不會影響到原有合成規則的程式碼，新的 LTS 模型也可以很容易地運用現有的合成規則，而 Context Constraint 和 Safety Property，而不用每次引入一個新的 LTS 模型，所有合成規則就要全部重新撰寫，解決之前架構中程式碼無法重用的問題。

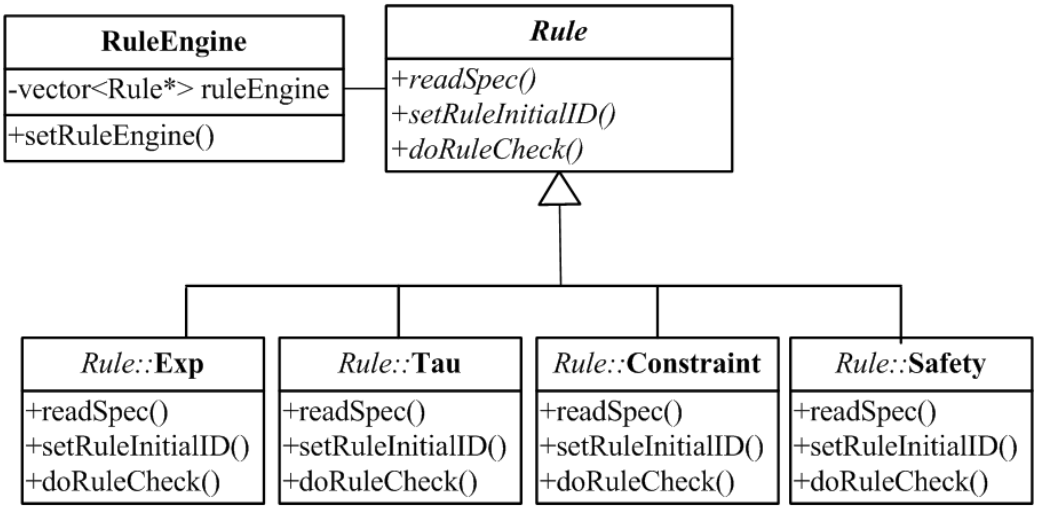


Figure 4-4 The inheritance structure of Rule Engine

當合成引擎中 enumeration()進行狀態展開時，會使用一個 for 迴圈，逐一檢查 RuleEngine 中所儲存的 Rule 物件，並會呼叫 Rule 物件中的 doRuleCheck()，將每個展開 state 和 transition 做為參數傳入進行合成規則的檢查。當每個 RuleEngine 中的 Rule 物件都檢查完成後，doRuleCheck()會 return 新的 state 和 transition 給合成引擎中的 enumeration()。enumeration()得到新的 state 和 transition 後，會將新的 state 和 transition 加入 LTS 平行合成後的結果。如果此時 enumeration()中當有狀態未展開完畢，enumeration()會繼續將 state 和 transition 交由合成規則引擎進行 doRuleCheck()分析，這樣的步驟會一直反覆進行，直到所有的狀態都全部處理完成。

對於即時化減的技術，我們也採取相同的方式，以多型架構取代條件式的判斷。即時化減的繼承架構可以提供合成引擎的 enumeration()在列舉狀態時可以使用 Minimize 物件以執行最小化。最小化引擎架構我們以 Figure 4-5 表示。

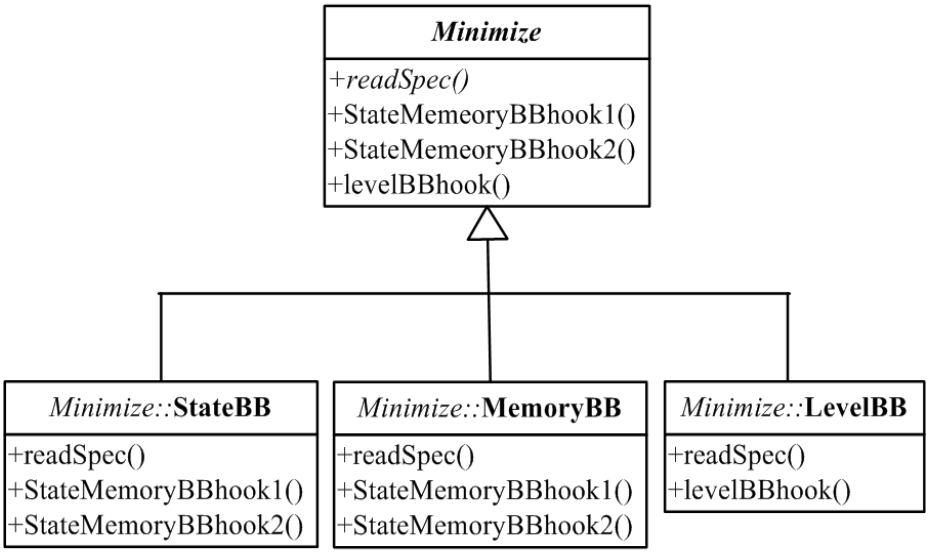


Figure 4-5 The inheritance structure of Minimize Engine

在此須要特別注意的是，由於在 enumeration()的演算法步驟中，不同的即時化減技術可能出現在演算法執行時的不同階段中，所以在 Compose 這個類別的實作中，我們使用了 hook method[8][21]的技術解決此問題，以 Table 3 中的第五行為例。

```
1:  void Compose::compose(){
2:
3:     setInitialID();
4:     enumeration();
5:     minimize->state&memoryBBhook2();
6:     writeToFile();
7:  }
```

Table 3 Template method in Compose class

Hook method 的用意在於，父類別可以用此掌控子類別可能有的擴充方式，父類別會去呼叫 hook method，而子類別可以挑選自己需要的 hook method 進行覆寫。Table 3 第五行程式碼 minimize->state&memoryBBhook2()，表示 Compose 物件會在此呼叫 Minimize 物件中的 state&memoryBBhook2()函式，由於在 Minimize 中的只有 StateBB 和 MemoryBB 物件會在此階段進行即時化減，所以只有這兩個子類別會進行 state&memoryBBhook2()這個 hook method 的覆寫。而 LevelBB 由於在此階段不需要做任何事，所以在 LevelBB 子類別的定義中，此 hook method 會是一個空的函式。

這樣的好處同樣是為我們帶來往後要繼續擴充的便利性，如果未來即時化減方式的選項增加，需要在展開狀態的過程中不同的時間點執行即時化減，我們只要在父類別中增加一個 hook method，再令此新增的 Minimize 子類別去實作此 hook method 的內容即可，其它既有的 Minimize 物件的程式碼完全不會變更到，當然也不用擔心會產生意料之外的程式錯誤。

在此，先前未重構的架構中所有的條件判斷式的問題，我們都以物件導向的多型方式再加上 *Template Method Design Pattern* 予以解決，在 Table 4 中，我們對於重構前後的函數呼叫方式，做一簡單的比較。

重構前的架構：
 	
使用 if-else
	1: if(withContextConstraint){
	2: if(stateBB)
	3: contextConstraintWithStateBB();
	4: else if(memoryBB){
	5: contextConstraintWithMemoryBB();
	6: else if(levelBB)
	7: contextConstraintWithLevelBB();
	8: else
	9: contextConstraint();
	10: }
	11: else if(withSafetyProperty){
	12: safetyProperty();
	13: }
	14: else{
	15: if(stateBB)
	16: enumerationWithStateBB();
	17: else if(memoryBB){
	18: enumerationWithMemoryBB();
	19: else if(levelBB)
	20: enumerationWithLevelBB();
	21: else
	22: enumeration();
	23: }

重構後的架構：
 	
使用多型
	//in RuleEngine.cpp
	1: if(LTStype==CCS){
	2: ruleEngine.push_back(new Exp);
	3: ruleEngine.push_back(new Tau);
	4: if(ContextConstraint)
	5: ruleEngine.push_back(new Constraint);
	6: if(SafetyProperty)
	7: ruleEngine.push_back(new Safety);
	
	// in Compose.cpp
	8: if(stateBB)
	9: minimize = new StateBB;
	10: else if(memoryBB)
	11: minimize = new MemoryBB;
	12: else if(levelBB)
	13: minimize = new LevelBB;

Table 4 The compare of two architectures

Section 4.3 重構後的架構圖

由於專案的發展往往是在不同階段，由不同的開發人員完成。所以系統架構的可維護性在專案開發的過程中顯得相當重要。我們以 Figure 4-6 說明重構後的物件導向架構。Composition 類別是保留原系統中合成引擎狀態平行合成的功能，讓演算法不變的地方只要在 Composition 類別中撰寫一次即可，而其餘可能

變動部分交由合成規則引擎與最小化引擎處理；Rule 類別的繼承架構代表著由原本合成引擎所獨立出來的合成規則引擎，可因應未來不同合成規則與不同型態 LTS 的變動需求；Minimize 類別的繼承架構表示即時化減技術提供給合成引擎不同最小化方式的選項，往後若有不同的即時化減方式，只要新增至此架構中的子類別即可，使得系統更具備彈性和擴充性。

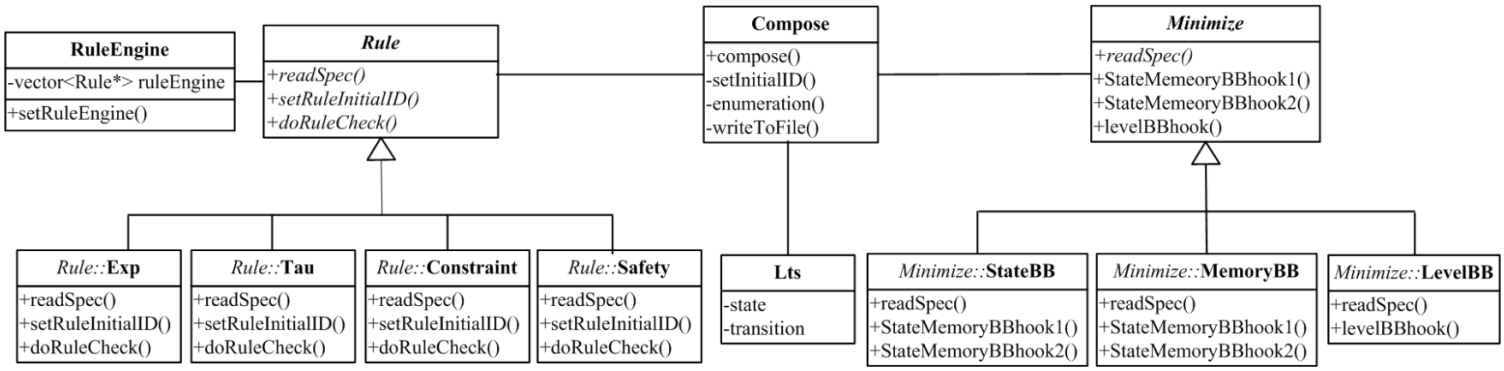


Figure 4-6 The system architecture after refactoring

Chapter 5　評估

局部性分析工具所使用到合成規則和欲檢驗的性質可以靈活運用，並且能因應未來分析模型的不同，達到程式碼重用的目的，乃是本篇論文進行系統重構的重點。在本章中我們會對重構後的系統逐一審核評估，並說明重構後的架構如何達成我們所要求的目標[30]。

Section 5.1　減少不必要的程式碼

由於我們在重構時希望可以減少系統中重複的程式碼，在 Table 5 的「減少%」此列中，我們可以發現重構後的系統，較原有的系統程式碼檔案大小，共減少了 38.3%；而程式碼總行數，較原系統的程式碼總行數，共減少了 26.5%。由此可知，重構後的系統架構不僅刪除了不必要的程式碼，還提高了程式碼的可讀性，使得程式的流程架構更清楚易懂，並且不影響原有系統中所具備的功能。

程式碼	功能	檔案大小 (bytes)	總行數
重構前系統	CCS Enumeration CCS Enumeration with StateBB CCS Enumeration with MemoryBB CCS Enumeration with LevelBB CCS Context Constraint CCS Context Constraint with StateBB CCS Context Constraint with MemoryBB CCS Context Constraint with LevelBB Safety Property	225,932	6548
重構後系統		139,286	4758
減少%		**38.3%**	**26.5%**

Table 5 The compare of codes

Section 5.2　彈性和擴充性

重構後的系統所帶來的好處不僅只有整理原有的程式碼,將其變得更清晰易懂。更重要的在於，重構後對於系統的彈性和擴充性提高，對於後續系統的開發與維護將會產生更明顯的助益。

5.2.1. 合成規則的組合與加入即時化減技術

若我們要在原有的系統進行 CCS 安全性質的檢驗時，加入一併執行即時化減的功能。在假設 CCS 檢驗安全性質時進行即時化減技術，不會影響原有 CCS 安全性質與即時化減理論上的運用的前提下。重構前的系統架構在新增此項功能時，等於要新增一個相對應的函式，以供主程式呼叫。並且以 copy-paste 的方式，在既有的 SafetyProperty()函式中，進行即時化減功能的修改。而這些功能的新增與修改，必須藉由 EnumerationWithStateBB()、EnumerationWithMemoryBB()和 EnumerationWithLevelBB()，這三個函式中有關即時化減技術的程式碼。才能實作出三個新的函式 SafetyPropertyWithStateBB()、SafetyPropertyWithMemoryBB()和 SafetyPropertyWithLevelBB()的新增，以供主程式呼叫。

由於原系統新增功能時必須以 copy-paste 的方式撰寫，所以我們以原有的函式 SafetyProperty()、EnumerationWithStateBB()、EnumerationWithMemoryBB()和

EnumerationWithLevelBB()這四個函式中既有的程式碼做估算，重構前的系統若欲新增檢驗安全性質時，同時進行即時化減的功能，必需再額外撰寫約 500 行的程式碼。

而在重構後的系統中，由於合成規則和即時化減技術選項是以組合的方式，分別由 Rule Engine 和 Minimization Engine 各別運作。如果要新增檢驗安全性質時，同時進行即時化減的功能，我們只要同時給定安全性質和即時化減技術的參數，不用再新增任何程式碼，系統即可立即進行分析。所以新增此功能所需的額外的程式碼是 0 行。以上的比較的結果我們以 Table 6 說明。

程式碼	新增功能	新增程式碼行數
重構前系統	CCS Safety Property with StateBB CCS Safety Property with LevelBB CCS Safety Property with MemoryBB	約 500 行
重構後系統		約 0 行

Table 6 The compare of adding minimization option in composition rule

在系統進行分析時，若我們希望加入同時進行本文限制和安全性質的檢驗，也就是兩個合成規則之間的組合。同樣地，我們假設兩個合成規則同時運作時，和該合成規則單獨運作時，所持有的理論邏輯是相同的。那麼在重構前的系統中，我們必須將原有 ContextConstraint()和 SafetyProperty()兩支函式的程式碼以 copy-paste 的方式修改，綜合成一支具有複合合成功能的函式，以供主程式呼叫。我們也依既有的程式碼進行估算，可以得知如果要在重構前的系統中新增

ContextConstarintWithSafetyPropery()的函式，必需再新增加約 500 行的程式碼。

對於重構後的系統而言，如果要新增同時進行本文限制和安全性質的檢驗，在理論不會因而改變的情況下，我們也無需新增任何程式碼，直接讓使用者設定參數，以原有的程式碼運作即可。事實上，重構後的系統只有在組合合成規則之後，會造成理論運作時細節的改變，才需要對程式碼做適當的修改或處理。而重構前的系統中，無論合成規則之間的組合，在理論上的操作會不會有變化，都必須要撰寫新的程式碼因應，這是由於原有的架構無法使得程式碼可以重覆利用，所造成後續系統開發上的負擔，以上的比較我們以 Table 7 說明。

程式碼	新增功能	新增程式碼行數
重構前系統	CCS ContextConstraintWithSafetyProperty	約 500 行
重構後系統		約 0 行

Table 7 The compare of combination of composition rules

5.2.2. 新增 LTS 型態

除了在合成規則中新增即時化減的功能，或是合成規則之間的組合之外，在之前的章節中我們也有提到，ArCats 的局部性分析工具希望在使用者建立系統模型時，可以提供更多彈性，所以未來希望加入 CSP 的合成規則，以進一步分析 CSP 的 LTS 模型。

我們除了在系統中加入 CSP 的基本合成規則，使得 CSP 可以具有平行合成，也就是 enumeration 的功能之外。我們並希望 CSP 可以具有如同 CCS 相同的分析功能：本文限制、安全性質檢驗、即時化減技術，以及前述這些合成規則之間可能的組合。

我們假設 CCS 和 CSP 在本文限制、安全性質和即時化減技術等理論上的運作基本上都是相同的。在重構前的系統中，原有的 CCS 本文限制、安全性質和即時化減技術的程式碼，完全無法直接讓 CSP 此新的 LTS 型態重覆使用，必需要將全部的程式碼重新撰寫。而以既有 CCS 程式碼所能達到的功能來估算，加入 CSP 後所需要重新撰寫的程式碼約有 2000 行以上。

而在重構後的系統加入 CSP 的合成規則，由於假設本文限制和安全性質的理論操作都相同，所以程式碼可以不用修改。僅需新增 CSP 中有關模型溝通方式的合成規則，也就是在 Section 3.1 中所介紹的 Com2。因為 CSP 在並行系統的溝通上是 multi-way rendezvous，CCS 是以 two-way rendezvous 的方式，所以 CSP 和 CCS 在基本的合成規則上僅在 Com2 的運作上有所區別。所以在重構後的系統中，我們只需增加 CSP 有關 Com2 的規則的程式碼即可，其餘的程式碼皆無需更動，新增的額外程式碼約在 100 行以內即可完成。我們以 Table 8 說明重構前後的系統架構，加入 CSP 後的比較。

程式碼	新增功能	新增程式碼行數
重構前系統	CSP Enumeration CSP Enumeration with StateBB CSP Enumeration with MemoryBB CSP Enumeration with LevelBB CSP Context Constraint CSP Context Constraint with StateBB	約 2000 行
重構後系統	CSP Context Constraint with MemoryBB CSP Context Constraint with LevelBB CSP Safety Property CSP Safety Property with StateBB CSP Safety Property with MemoryBB CSP Safety Property with LevelBB	約 100 行

Table 8 The compare of adding new LTS type (CSP)

Section 5.3 程式執行效能

我們以電梯系統[4]為例，說明重構前與重構後的系統，在執行時所花費的時間和記憶體使用量。由 Table 9 中可知，無論電梯的台數為何，重構後的系統相較於重構前的系統，在執行時都需要花費較多時間與記憶體。

電梯數	2		3		4	
效能	時間 (sec)	記憶體 (bytes)	時間 (sec)	記憶體 (bytes)	時間 (sec)	記憶體 (bytes)
重構前系統	0.578	4,660K	6.328	23,720K	58.156	182,440K
重構後系統	0.687	4,860K	7.671	28,396K	71.171	194,816K

Table 9 The compare of executing time and used memory

雖然重構後的系統效能未能提昇，但由於本研究進行重構時的理論基礎在於，建立一個合成規則引擎，好讓後續研究中的合成規則可以輕易地加入，並運用現有的程式碼，以降低未來實作上的負擔。而目前重構後的架構，也確實達到了我們在理論上的實作要求。

關於系統效能改進的部份，可做為未來再次進行重構的實作方向。未來再次重構的重點，可朝向效能的最佳化，例如記憶體的使用量降低等此類目標著手。在未來工作中，我們建議改善狀態的資料儲存方式，可以降低記憶體的使用。而程式改良後的架構中，對於資料結構的重整，只需集中於修改一部份的程式碼。若是重構前的架構，由於資料結構的改變，會涉及所有函式的變動，所以需要對所有的程式碼做修改，工程相當浩大。所以重構雖然會使得系統執行時間變長，但重構也將使得軟體的效率最佳化更容易進行[23]。

Chapter 6　結論與未來展望

Section 6.1　結論

重構所帶來的好處，除了消除不必要的程式碼，提高架構的彈性與維護性之外，亦可以加速後續專案開發的速度。本論文中原先的重構目標，撰寫一具有彈性的局部性分析工具，所以我們將邏輯變動的部份，設計成 Rule 和 Minimize 兩個繼承架構去實作。然而，在先前的設計中，每個局部性分析所具備的功能皆以獨立的 method 實作，除了先前章節中提到大量 copy-paste 的問題，參數以傳值的方式傳遞，使得每一個 method 和儲存狀態的資料結構耦合程度非常高。往後若要針對資料結構加以改良，以減少狀態儲存時佔有記憶體的空間，由於所有的 method 都和狀態的資料結構有所牽連，所以我們必須修改系統中所有的 methods 一以符合新的資料結構型態。

而重構後的架構由 Figure 4-6 可知，真正和儲存狀態的資料結構（即 Lts 類別）有直接相關的類別僅有 Compose。往後若需修改儲存狀態的資料結構，只需修正 Compose 類別中的程式碼即可。由於此效果並非我們原本重構前所預期，可以說是重構後所帶來的意外驚喜，再次證明隨著系統架構的改良，可減輕後續開發時無謂的負擔。

Section 6.2　未來展望

在本篇論文中，我們將 ArCats 局部性分析工具，重構為一具有演化性和擴充性的物件導向架構系統，最終的用意就是希望在往後增加合成規則和進行不同 LTS 模型分析時，程式的撰寫能夠不僅容易擴充，並大幅降低維護和 debug 的成本，加速系統開發的速度。

目前 ArCats 是以 CCS 的 LTS 為主要分析模型，未來我們希望加入 CSP -based 的 LTS 模型，使得 ArCats 局部性分析工具所分析的模型可以更多元化。另外我們也預計加入其它合成規則於重構後的局部性分析工具中，例如 Liveness Property 的檢驗。

此外，我們也希望藉由這樣富有彈性的架構，可以藉由各個合成規則間的組合，提供未來理論研究時的實作，例如在檢驗安全性質時，是否可以一併進行本文限制的化減，或是使用即時化減技術進行狀態的最小化，以真正發揮局部性分析的優點。

在合成引擎未來的改進方向中，我們也可以針對每個狀態的資料結構儲存方式加以改良，例如使用資料壓縮技術。在狀態展開使用各種技術予以簡化的同時，也可以從狀態佔有記憶體的空間著手，試圖達到最小化或是更為精簡，以降低局部性分析面臨組態爆炸此最大難題的可能。

Reference

[1] Bryant, R. E. *Graph-based algorithms for Boolean function manipulations.* IEEE Transactions on Computers, Vol. C-35, No. 8, pp677-691, 1986.

[2] Clarke, E. M., Grumberg, O. and Peled, Doron A. *Model Checking*, MIT press, 2000.

[3] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[4] D.J. Richardson, S.L. Aha, and T.O. O'Malley. *Specification-based test oracles for reactive system*. In Proceedings of the Fourteenth International Conference on Software Engineering, pages 105-118, Melbourne, Australia, May 1992.

[5] E. Madelaine and R. de Simone. *The FC2Tools User Manual*, http://www-sop.inria.fr/meije/verification/, 1999, INRIA.

[6] E. Madelaine and R. de Simone. *Fc2 format Reference Manual*, http://www-sop.inria.fr/meije/verification/, 1999, INRIA.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. *Automatic verification of finite-state concurrent system using temporal logic*. ACM Transactions on Programming Languages and Systems,8（2）:244-263,April 1986.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading, Mass. 1995.

[9] G. J. Holzmann, *The model checker SPIN*. IEEE Transaction on Software Engineering, Volume 23 , Issue 5 : 279-295,May 1997 .

[10] G. R. Andrews, *Concurrent Programming – Principles and Practice*. The Benjamin/Cummings Publishing Company Ltd., 1991.

[11] Gian-Luigi Ferrari, Stefania Gnesi, Isti-C.N.R.Pisa, Marco Pistore. *A model-checking Verification Environment for Mobile Processes*. ACM 2003.

[12] Graf, S. and Steffen, B. *Compositional Minimization of Finite State Systems*, pages 186-196, CAV 1990.

[13] Godefroid, P. and Wolper, P. *A Partial Approach to Model Checking*. LICS 1991, pp.406-415.

[14] G. G. Koni-N'Sapu. Supremo, *A scenario based approach for refactoring duplicated code in object oriented systems*. Diploma thesis, University of Bern, June 2001.

[15] Holzmann, G. J. *The model checker SPIN*. IEEE Transaction on Software Engineering, Volume 23, Issue 5 : 279-295, May 1997.

[16] Holzmann, G. J. *State Compression in SPIN: Recursive Indexing and Compression Training Runs*, In Proceedings of Third International SPIN Workshop .1997.

[17] Holzmann, G. J. *The SPIN Model Checker: Primer and Reference Manual*. Pearson Educational, 2003.

[18] Hong-Yi Wang master thesis. *On-the-fly Bisimulation Minimization in Compositional Analysis*. NTNU 2005.

[19] Kim, M., Bergman, L, Lau, T., and Notkin, D. *An Ethnographic Stud of Copy and Paste Programming Practices in OOPL*. International Symposium on Empirical Software Engineering, 2004.

[20] Larsen. K. and Milner, R. *Verifying a protocol using relativized bisimulation*. In proceedings of the 14th International Colloquium on Automata language and Programming**.** 1987.

[21] L. Tokuda and D.S. Batory, *Evolving Object-Oriented Designs with Refactorings*, Automated Software Engineering, 8, 89–120, 2001

[22] Malhotra J. Smolka, S. A. Giacalone and Shapir R. *A tool for hierarchical design and simulation of concurrent systems*. In Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems, 1988.

[23] Martin. Fowler. *Refactoring: Improving The Design Of Existing Code*. Addison-Wesley, 1999.

[24] R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Clifffs, N.J., 1989.

[25] R. Milner. *A Calculus of Communicating System.* Vol92 of Lecture Notes in Computer Science. Springer-Verlag, New York, 1980.

[26] R.J. Van Glabbeek and W. Peter Weijland, *Branching Time and Abstraction in Bisimulation Semantics (extended abstract).* In information Processing 89, G. Ritter, ed., North-Holland, 1989, pages 613-618.

[27] S.C. Cheung, D. Giannakopoulou, J. Kramer. *Verification of Liveness Properties using Compositional Reachability Analysis*, In 5th ACM SIGSOFT Symposium on Foundations of Software Engineering, pages 227-243, Zurich, Switzerland, Sep, 1997.

[28] S.C. Cheung and J. Kramer. *Checking Safety Properties using Compositional Reachability Analysis*, ACM Transactions on Software Engineering and Methodology, Jan, 1999, vol. 8, page 49-78.

[29] Shing Chi Cheung and Jeff Kramer. *Context Constraints for Compositional Reachability Analysis*. ACM 1996.

[30] T. Mens and T. Tourw´e. *A survey of software refactoring*. IEEE Transactions on Software Engineering, 30(2):126–139, Feb. 2004.

[31] Yeh, W.-J. and Young, M. *Compositional Reachability Analysis Using Process Algebra*, TAV4, pages 49-59, 1991.

[32] Yung-Pin Cheng. The ArCats reference. http://www.csie.ntnu.edu.tw/~ypc/ArCats.htm, 2007

[33] Yu-ju Cheng master thesis. *Explicit Compositional State-Space Enumeration with Context Constraint & Counter Examples by Hierarchical Tracing*. NTNU 2005.

[34] Zhi-Wen Hsu master thesis. *Compositional Model Checking Safety and Liveness Properties in ARCATS.* NTNU 2006.