私立東海大學

資訊工程學系研究所

碩士論文

指導教授：陳隆彬　博士

以狀態簡化技術改良分散式監測與除錯系統

Enhancing Distributed Monitoring and Debugging by using State Reduction Techniques

研 究 生：徐　明

中華民國一百年七月

摘要

分散式系統中，檢測一個給定的全局邏輯述詞是否為真，是程式監測與除錯的基礎。由於問題的排列組合性質，檢測所有的全局狀態相當耗時。本文提出了複合狀態的表示方法，將一序列的執行狀態組合起來，並且設計了一種有效率的演算法，可在線上執行狀態組合和述詞檢測。根據狀態組合技術，本論文的演算法，能產生正確的演算結果，同時通過減少所評估的狀態數目，可以提升演算法的性能。

關鍵字：分散式計算、分散式監測與除錯、全局述詞檢測

Abstract

In a distributed system, detecting whether a given logical predicate is true on the global states is fundamental for testing and debugging the program. Detecting predicates by examining all global states is time consuming due to the combinatorial nature of the problem. This paper proposes the notion of composite state of which a sequence of execution states can be combined. An efficient algorithm is also developed in this paper to perform the state combination and predicate detection in an online manner. Based on the notion of composite state, the detection result can still be correct, and the performance of the new algorithm is improved as the number of states to be evaluated is reduced.

Keywords: Distributed computing, Distributed monitoring and debugging, Global predicate detection.

目錄

摘要..I

Abstract..II

目錄..III

圖目錄..V

第一章　簡介..1

第二章　背景和定義..2

2.1　分散式計算 2

2.2　全局狀態，切割和運行....................................4

2.3　向量時鐘 6

第三章　分散式計算之監測....................................8

3.1　全局述詞評估 8

3.2　分散式計算之監測 10

第四章　分散式監測之狀態簡化................................12

4.1　支點狀態(Pivot States)..................................12

4.2　以複合狀態加強的演算法..................................13

第五章　實驗模擬..16

5.1 OMNET 模擬器..16

5.2 實驗結果..17

第六章 結論..19

參考文獻..20

圖目錄

圖 1: 切割示意圖 5

圖 2: 向量時鐘時戳更新示意圖 6

圖 3:分散式程式的時空圖及其所有全局狀態集合形成的絡.............. 9

圖 4: 複合模組 11

圖 5: FRONT PIVOT 演算法 13

圖 6: REAR PIVOT 演算法 14

圖 7：CM 演算法 15

圖 8：模擬器的執行畫面 16

圖 9：應用程序事件回報檢查者程序信息傳送模擬 17

圖 10：應用程序僅回報支點給檢查者程序 17

圖 11：前/後支點 R 與 F 取範圍比較大的那段................................ 18

第一章　簡介

分散式系統中全局述詞(global predicate) Φ 之內容是由不同程序的變數所組成，在程式的監測和除錯中，某些執行狀態必須檢測邏輯述詞是否為真。全局述詞檢測可以找出在一致全局狀態的分散式程序執行上，Φ是否為真[1]。通常Φ是用來定義分散式系統裏的不良情況，例如，Φ之真值係由全局狀態裏的本地狀態變數值所改變，通常程序之上下文值與全局述詞之真值呈現正相關傾向。例如總和全局述詞 $\Phi = x_1 + x_2 + \ldots + x_n \geq K$，當 x_i $(1 \leq i \leq n)$愈大時，Φ就愈有可能為真，本文稱此特性為單調性(monotonicity)。

本文提出了複合狀態的表示方法，將一序列的執行狀態組合起來，仍然可以產生正確的檢測結果。本文也設計了一種有效率的演算法，可在線上執行狀態組合和述詞檢測。根據此技術，透過減少所評估的狀態數目，可提升演算法的性能。本文其餘部分結構如下，第二章介紹背景和定義，第三章介紹分散式計算之監測，第四章介紹分散式監測之狀態簡化，第五章介紹實驗模擬，第六章為結論。

第二章　背景和定義

2.1　分散式計算

通常分散式計算係描述分散式程式的行為，這些分散式程式則由一群程序所執行。每一序列程序的活動則建模為執行一系列事件順序，事件可能是內部的一個程序，只改變本地的狀態，也可能涉及與另一個程序的通信，通信則經由匹配的信息事件 send(m)和 receive(m)完成。換言之，即使數個程序發送相同的數據值給同一程序，信息本身仍然是單一的。通常事件 send(m)係從一個入列(enqueue)信息 m 經由傳出通道傳輸到目的程序。而事件 receive(m)則從一個出列(dequeue)信息經由傳入通道傳輸到目的程序之行為。然而 receive(m)事件欲存在於程序 p 中，信息 m 必須到達 p，而且 p 必須宣布它願意接收信息。否則就是信息被延遲（因為程序還沒有準備好）或程序被延遲（因為信息沒有抵達）。

此「信息傳遞」的通信觀點在事件層級(level)可能是完全不同於其他更高的系統層。但在我們觀察的分散式計算層級，這些高層次的通信都簡化歸納為：在一對對程序上產生匹配的發送和接收事件。

程序 p_i 的本地歷史，在計算中是一序列事件 $h_i = e_i^1 e_i^2 \ldots$（可能無限多個）。這種程序事件的標記，稱為**典型的列舉**(canonical enumeration)，對應於本地事件執行順序之全序關係。

令 $h_i^k = e_i^1 e_i^2 \ldots e_i^k$ 代表本地歷史 h_i 中前 k 個事件，其中 e_i^1 為第 1 個事件，e_i^2 為第 2 個事件等等…, 依此類推，e_i^0 則定義為空序列。全局歷史的計算是一組集合 $H = h_1 \cup h_2 \ldots \cup h_n$，即包含其所有事件。

然而一個全局歷史並沒有指定事件之間的相對時間，在一個異步分散式系統裏，沒有全局時間框的存在，事件之計算順序可僅基於「原因和結果」的概念。我們可以定義在事件上的一個二元關係如下[2]：

1. If $e_i^k, e_i^\ell \in h_i$ and $k < \ell$, then $e_i^k \rightarrow e_i^\ell$,
2. If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$,
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

$e \rightarrow e'$，若且唯若 e 之因果關係發生在 e' 之前 — 此關係吻合了我們「原因和結果」的直觀概念，然而只有在匹配的發送接收事件下才有確定的原因和結果關係。

全局歷史中的某些事件對也可能不存在因果關係。也就是對某些事件 e 和 e' 而言，有可能不存在因果關係 $e \rightarrow e'$ 或 $e' \rightarrow e$，我們稱這樣的事件為**併發**（concurrent），以 $e||e'$ 表示之。

分散式計算形式上是一個**偏序集**（poset）定義為（$H,\rightarrow$）。所有事件都標示著它們的典型列舉，然而在通信事件中，它們也包含唯一的信息標識符。因此，每個程序的事件**全序**（total order），以及發送和接收的匹配是隱含在 H 裏。

2.2　全局狀態，切割和運行

設 σ_i^0 代表 p_i 的初始狀態，然後才執行任何事件，設 σ_i^k 表示本地程序 p_i 執行事件 e_i^k 後的立即狀態。程序的本地狀態可能包括信息例如：局部變數值，以及在各通道上信息發送和接收程序事件的順序。分散式計算的**全局狀態**(global states)為一個 n 元組的本地狀態總和：$\Sigma = (\sigma_1, \cdots, \sigma_n)$，其中每一項代表一個程序的狀態。

分散式計算的一個**切割**(cut)是其全局歷史 H 裏的一個子集，包含每一個本地歷史的初始前綴(initial prefix)。設 $C = h_1^{c_1} \cup h_2^{c_2} \ldots \cup h_n^{c_n}$，其中 $(c_1, c_2 \ldots, c_n)$ 對應於每個程序的前次事件索引，包含於切割 $(c_1, c_2 \ldots, c_n)$ 的前次事件集合 $(e_1^{c_1}, e_2^{c_2} \ldots e_n^{c_n})$ 稱為切割之**前鋒**

(frontier)。**切割($c_1, c_2\ldots,c_n$)則對應於全局狀態($\sigma_1^{c_1}, \sigma_2^{c_2}\ldots\sigma_n^{c_n}$)**，如圖 1 所示。

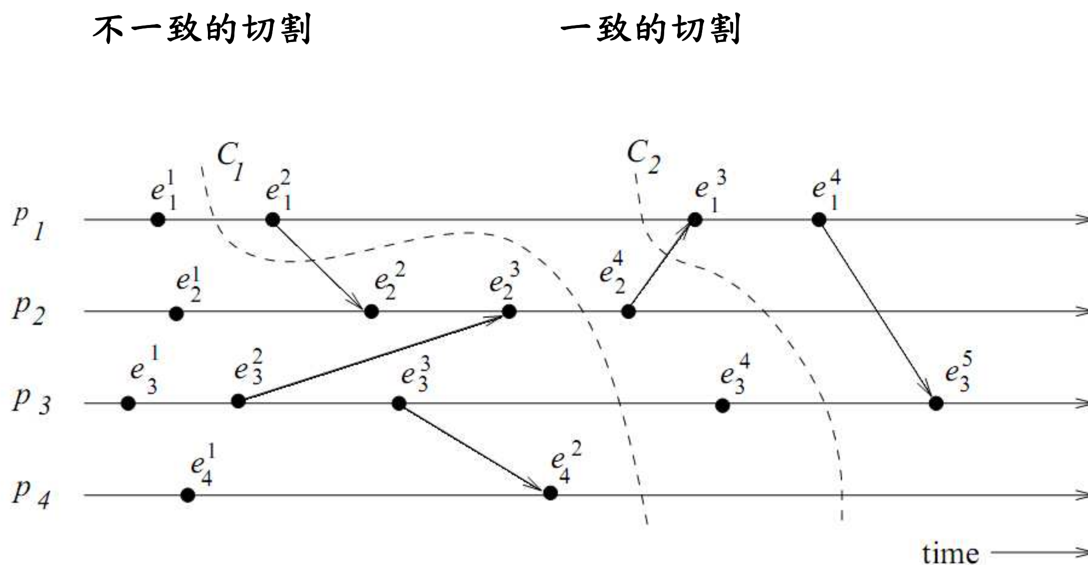

圖 1: 切割示意圖[3]

雖然分散式計算是一個偏序集，但實際執行時，所有事件，包含在不同程序之事件，存在著某種全序關係。為理解分散式系統的執行，產生了**運行**(run)的觀念。分散式計算的運行是包含全局歷史中所有事件的全序關係 R，而此全序關係與每一個本地歷史是相符一致的。換言之，對每一個程序 p_i 而言，其事件順序在 h_i 與 R 是一樣的。值得注意的是，運行不需要符合任何可能的執行，而一個分散式計算也可能會有許多運行，每一個運行則相當於一個不同的執行。

2.3 向量時鐘

程序產生的一個事件，其時戳即為該程序當時的向量時鐘值，聯繫兩個事件或兩個本地狀態的向量時戳，我們可以決定它們是否有因果關係存在[4]。

程序 P_i 含有整數向量 $VC_i[1\ldots n]$ (初始值 [0,...,0])，而其向量時鐘的時戳系統規則如下[5]：

當程序 P_i 產生一事件 (send, receive, or internal)，向量時鐘 VC_i[i]增加 (VC_i**[i]:** = VC_i**[i]+1**) 。 (4)

當程序 P_i 發送一信息，其向量時鐘 VC_i 之值 $m.VC$ 附加於信息上。(5)

當程序 P_i 接收一信息，其向量時鐘更新為 $VC_i := max(VC_i, m.VC)$。 (6)

向量時鐘的時戳系統更新如圖二所示。

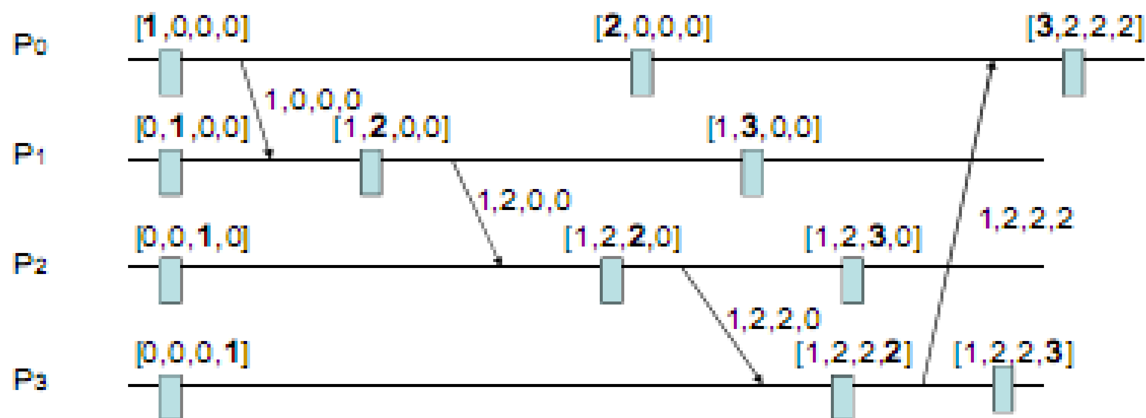

圖 2: 向量時鐘時戳更新示意圖

向量時鐘可推導一致的全局狀態，令一事件 e 之時戳為 $TS(e)$，P_i 之事件 e 與 P_j 之事件 e' 一致的條件為：

$$TS(e)[j] \le TS(e')[j] \; and \; TS(e')[i] \le TS(e)[i]$$

當兩個事件一致時，它們可屬於一致切割的前鋒，而兩個事件後之狀態則為併發。

向量時鐘可用來推導事件間發生在先(happen before)之因果關係，例如程序 P_i 產生事件 e (時戳為 $e.VC$)，程序 P_j 產生事件 f (時戳為 $f.VC$)，則其事件因果關係如下[6]：

$$\forall (e, f) : e \neq f : (e \rightarrow f) \Leftrightarrow (e.VC < f.VC),$$

第三章　分散式計算之監測

3.1　全局述詞評估(GPE :Global Predicate Evaluation)

我們可以用 GPE 來評估述詞[7]，述詞則代表分散式系統全局狀態總和的一個函數。假設單一程序 p_0 稱為「檢查者」(以下簡稱 checker)負責評估 Φ，而 p_0 可以是 $p_1,\ldots,p_n$ 其中之一，或同一分散式系統中之其他計算程序。在此特殊情況下，有一個 checker，可以將解決全局述詞 Φ 的問題簡化為 p_0 構建一個全局狀態的 Σ 計算。簡單起見，假設 checker 代表的事件執行是由外部監測，且基本上並不改變計算和事件的典型列舉。

若p_0採取主動，它發送給每個程序「狀態調查」的信息。當程序p_i收到此信息，便立即回報其本地狀態。當所有n個程序都回答了，p_0即可建構一個全局狀態的計算 $(\sigma_1,\ldots,\sigma_n)$。$p_0$程序接收到的本地歷史查詢可有效地定義為切割(即全局狀態)。

以下簡稱一致的全局狀態為全局狀態，一個執行的所有全局狀態集合形成所謂**絡**(lattice)，若系統經由執行一事件，可由一個全局狀態前進到另一個全局狀態，則絡裏面的這二個全局狀態可由**邊**(edge)所連接。絡的初始全局狀態為<0,…,0>，絡裏面從初始全局狀態開始的

路徑就是系統的一個**計算**(computation)。

程序P_i的第x個事件稱為$e_{\mathrm{i,x}}$，執行事件$e_{\mathrm{i,x}}$後(但在$e_{i,x+1}$之前)的狀態稱為$s_{i,x}$，數目x稱為$s_{i,x}$的序數(sequence number)，圖3 說明了分散式程式執行時的事件、狀態與其相對應的絡。

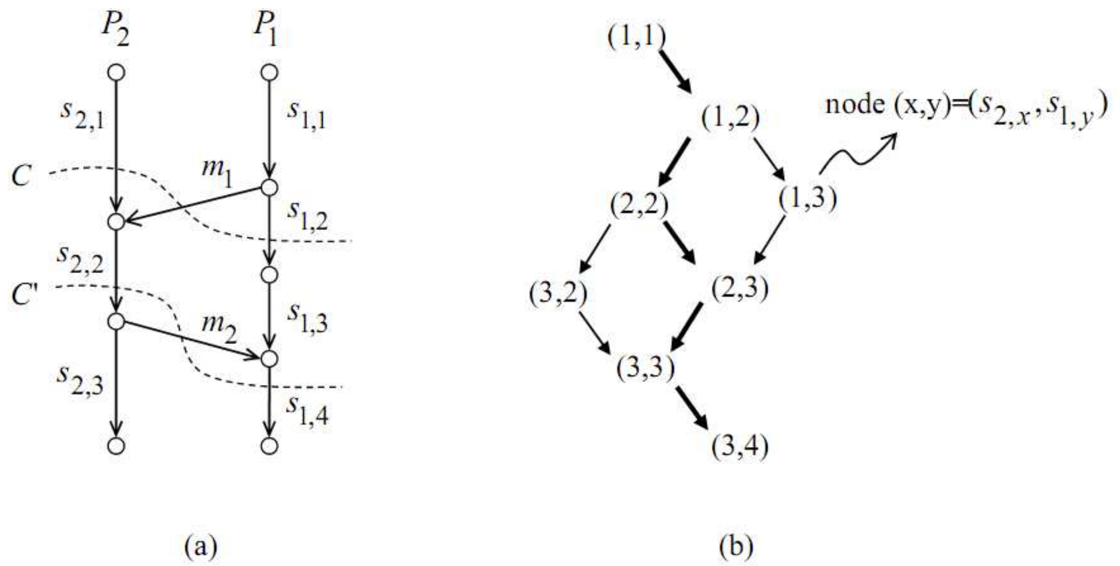

圖 3: (a)分散式程式的時空圖　　(b)其所有全局狀態集合形成的絡

分散式程式的運行可視為絡裡面的一條路徑，從初始節點(initial global state)出發到最終節點(final global state)為止。

Possibly(Φ)必須搜尋任何可能滿足Φ之狀態，需對絡作全程搜索(exhaustive search)。亦即從第0層(level 0)的初始狀態開始檢測到最後狀態，每一層都必須檢測，搜尋到可能滿足Φ之狀態為止[8]。

Possibly(Φ)之評估可由監測程序執行，每一程序在執行一個事件之後，便發送事件之時戳，以及相關於Φ的本地變數值給監測程序。為了發現Φ是否成立，監測程序需造訪(或測試)所有的全局狀態，並利用造訪得到的變數值來評估Φ。

3.2 分散式計算之監測

本文以個別的檢查者程序(checker process)來收集應用程序回報的向量時鐘，以執行程式分析。checker 程序藉由收集得來的向量時鐘來建構絡。分析程式時，checker 程序橫越絡裏面的節點，並呼叫檢測模組(detection module)，例如除錯程式或軟體驗證工具。但由於絡結構的基數組合性質，要橫越絡裏面的所有節點(即檢測所有的全局狀態)，便極難處理。

本文討論之複合模組，在檢測模組分析前，以複合狀態的表示方法，將區間內連續的執行狀態組合起來，而檢測模組仍然可以產生正確的檢測結果，如圖四所示。

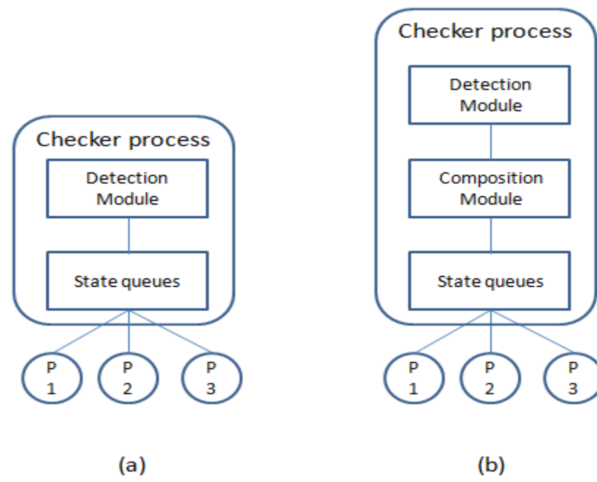

圖 4：複合模組(Composition Module)

(a)直接呼叫檢測模組分析 (b)先以複合模組實行狀態簡化後分析

本文也以一種有效率的演算法，可以在線上執行狀態組合和述詞檢測。根據此技術，減少所評估的狀態數目，可提升演算法的性能。

第四章　分散式監測之狀態簡化

4.1　支點狀態(Pivot States)

對全局述詞 $\Phi = x1 + x2 + \ldots + xn \geq K$，當 xi $(1 \leq i \leq n)$ 愈大時，Φ 就愈有可能為真(單調性)。許多分散式系統裏使用令牌(token)來代表資源數量及使用權，檢測系統令牌的總量是否符合全局述詞非常實用，大部份關於令牌的算術運算的全局述詞都滿足這種單調性質。

利用單調性，本文將數個本地狀態組合成為複合狀態 (composite state)，以區間內最大值為支點(pivot)，代表此區間內全局述詞最有可能成立的情況。當一個區間內之最大值使得 Φ 不成立時，區間內其他值亦使Φ不成立。因此，整個區間只需驗算支點即可。

支點是一個複合狀態中具有最大值的狀態。分為以下兩類：

- 前支點(Front Pivot)：一段連續狀態(Si,begin ,⋯, Si,end)，不含有 send 事件，而且第一個狀態 Si,begin 具有最大值，則 Si,begin 稱為前支點。
- 後支點(Rear Pivot)：一段連續狀態(Si,begin ,⋯, Si,end)，不含有 receive 事件，而且最後狀態 Si,end 具有最大值，則 Si,end 稱為後支點。

4.2 以複合狀態加強的演算法

本節討論 CM (Composition Module) 演算法來決定狀態序列的較佳合併方式，CM 演算法如圖 5, 6, 7。演算法中使用 $S_{i,x}$ 代表程序 P_i 的第 x 個狀態，以 $d(S_{i,x})$ 代表 $S_{i,x}$ 的變數值 (亦即全局述詞 Φ 的變數 x_i 在狀態 $S_{i,x}$ 的值)。

圖 5 的 Procedure F() 是尋找程序 P_i 從第 x 個狀態開始的 F-composition state 以及 F-pivot。做法是依序檢視第 x、x+1、x+2…狀態，直到遇到 send 事件為止。

procedureF

```
fend = pivot = x; //initialize index
last ? the index of last state in process Pi;

while (fend < last) and (e_{i,fend+1} is not a send event) do
    fend ?  fend + 1;
    if d(s_{i,pivot}) = d(s_{i,fend}) then
        pivot ?  fend; //update pivot
    end if
end while
```

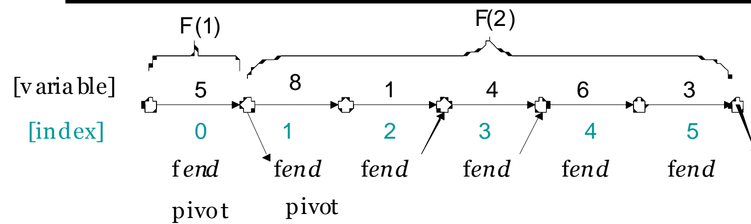

圖 5： Front Pivot 演算法

圖 6 的 Procedure R()是尋找 R-composition state，與 Procedure F()類似，直到遇到 receive 事件為止。

procedureR

```
rend = pivot = x; //initialize index
last ?  the index of last state in process Pi;

while (rend < last) and (e_{i,rend+1} is not a receive event) do
    rend ?   rend + 1;
    if d(s_{i,pivot}) = d(s_{i,rend}) then
        pivot ?   rend; //update pivot
    end if
end while
```

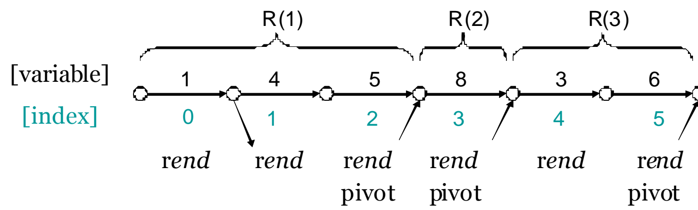

圖　6 : Rear Pivot 演算法

給定一段 $S_{i,begin}$ 到 $S_{i,end}$ 之間的狀態序列，此序列可以有前或後兩種複合方式。我們同時檢測兩種方式，並選取較長的狀態序列組合以增加壓縮率，如圖 7 所示。

Algorithm 3 CompositionModule($s_{i,begin}$, $s_{i,last}$)

```
1: Input: a sequence of states s_{i,start}..s_{i,last} in process P_i
2: Output: a list of intervals and pivot states that represents the states s_{i,start}..s_{i,last}
3:
4: pivotList ← {}
5: x ← start
6: while x ≤ last do
7:    /* find F-interval and R-interval starting from s_{i,x} */
8:    (s_{i,fbegin}, s_{i,fend}) ← f(s_{i,x})
9:    (s_{i,rbegin}, s_{i,rend}) ← r(s_{i,x})
10:
11:   /*choose the longer interval*/
12:   if fend ≥ rend then
13:      Add (s_{i,fbegin}, s_{i,fend}) to pivotList.
14:      x ← fend + 1
15:   else
16:      Add (s_{i,rbegin}, s_{i,rend}) to pivotList.
17:      x ← rend + 1
18:   end if
19: end while
20: return pivotList;
```

圖 7：CM 演算法

第五章　實驗模擬

5.1　OMNET 模擬器

為驗證複合模組演算法(以下簡稱 CM 演算法)是有效率的方法，我們使用 Omnet++網路模擬程式進行實驗[9]。在 Omnet++架構下一個模擬程式可以結合許多模組，其中網路描述模組代表網路結構，omnetpp.ini 為組態配置檔整合所有系統參數。程式撰寫完成後經由編譯、連結產生執行檔。Omnet 程式的執行結果，如圖 8 所示。

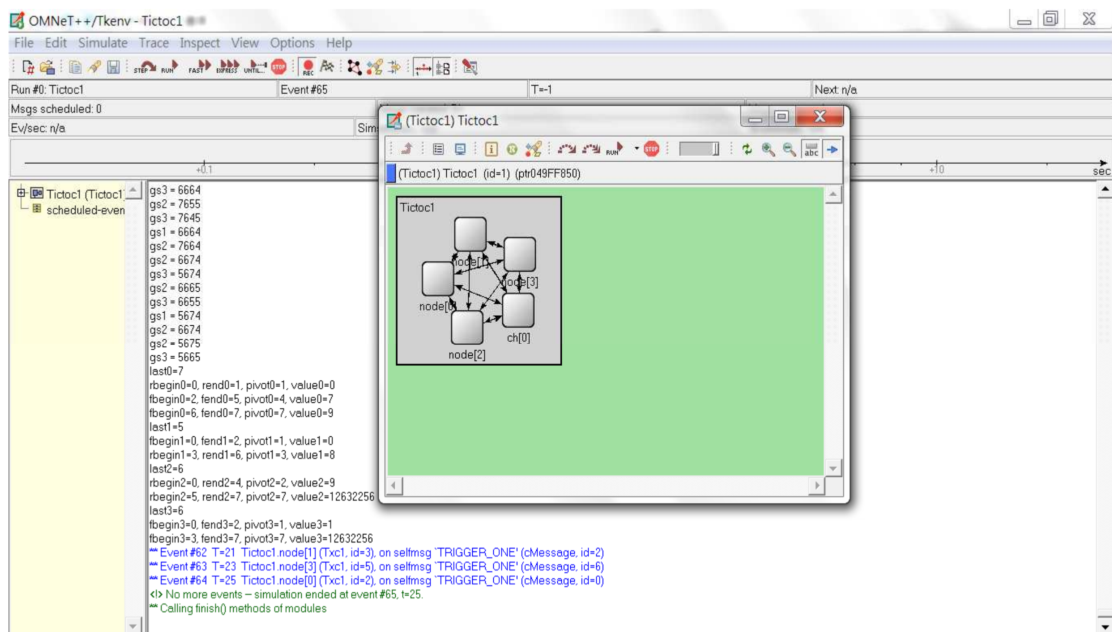

圖 8：模擬器的執行畫面

5.2　實驗結果

應用程序 N0~N3 報告給檢查者程序 Checker 的向量時戳、變數、事件類型、佇列信息內容以及絡節點如圖 9、10 所示。

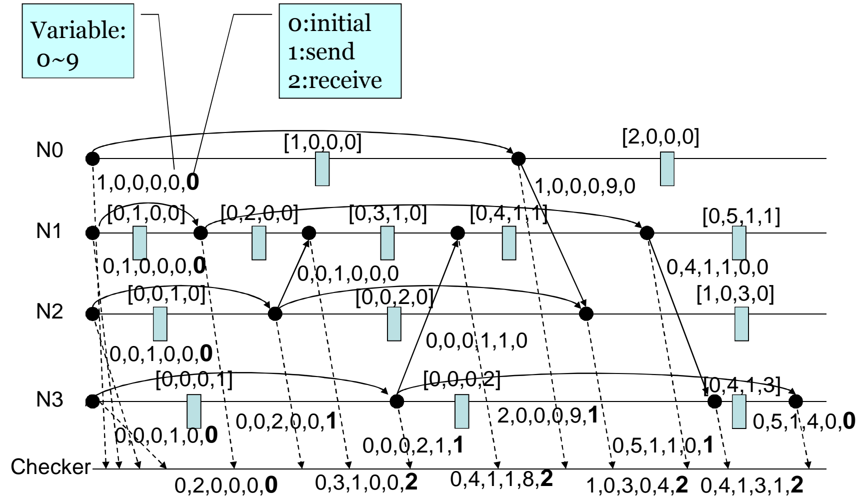

圖 9：應用程序事件回報檢查者程序信息傳送模擬

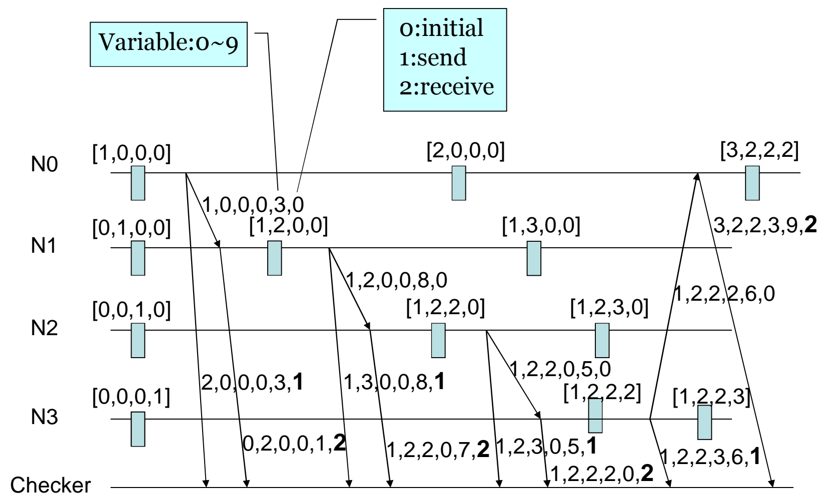

圖 10：應用程序僅回報支點給檢查者程序

CM 演算法實驗求得之支點及佇列信息內容如圖 11 所示。

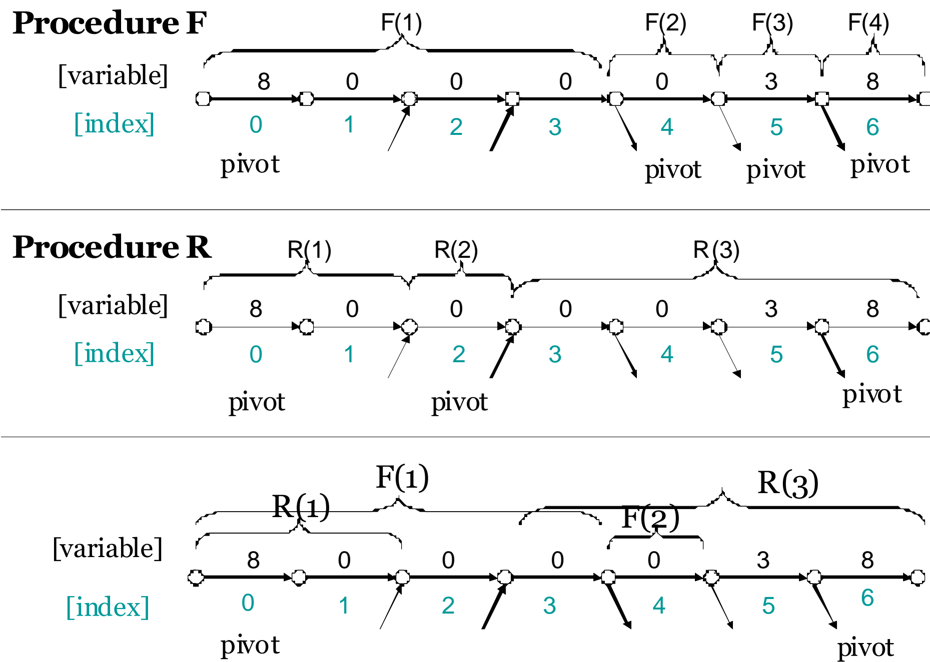

圖 11：前/後支點 R 與 F 取範圍比較大的那段

第六章　結論

S. Alagar和S. Venkatesan[10]使用區間來增加執行步驟的顆粒性，將執行步驟由一個程序中的一個事件，增加為一序列事件(區間)，其演算法由檢查每一個全局區間，取代檢查每一個全局狀態檢查點。

本文以單調性為基礎，將數個狀態組合成為複合狀態，更進而由檢查每一個全局區間的支點，來取代檢查每一個全局區間，除了減少全局狀態「絡」的基數以外，利用單調性的特徵，可不需從頭開始橫越絡裏面的所有節點，以更有效率的方式去查訪應用程式的絡，來檢測全局述詞。

大部份關於令牌的算術運算全局述詞都滿足此單調性質，傳統的集中檢測方式，應用程序必須回報其所有執行狀態給除錯程序，本文提出的複合狀態表示方法，將一序列執行狀態組合起來，並設計了一種有效率的演算法，可在線上執行狀態組合和述詞檢測，並依然能產生正確的結果。本文提出的演算法，對應用程序狀態進行壓縮，能有效地減少其回報信息數量，並進而提升演算法的性能。

參考文獻

1. R. Cooper and K. Marzullo. Consistent detection of global predicates. Sigplan Notices, pages 167-174, 1991.
2. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.
3. Ajay D. Kshemkalyani and Mukesh Singhal. DISTRIBUTED COMPUTING PRINCIPLES, ALGORITHMS, and SYSTEMS. January 28, 2007, pp. 99.
4. C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In Eleventh Australian Computer Science Conference, pages 55–66, University of Queensland, February 1988.
5. R. Baldoni M. Raynal, Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems, IEEE Distributed Systems on-line, formerly IEEE Concurrency. Vol. 3, No. 2, February 2002.
6. Fidge C., Logical Time in Distributed Computing Systems. IEEE Computer 24(8), 28–33(1991).
7. Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In ACM/ONR Workshop on Parallel and Distributed Debugging, pages 163–173, Santa Cruz, California, May 1991.

8. Ajay D. Kshemkalyani and Mukesh Singhal. DISTRIBUTED COMPUTING PRINCIPLES, ALGORITHMS, and SYSTEMS. January 28, 2007, pp. 415.

9. A. Varga. Using the omnet++ discrete event simulation system in education. IEEE Transactions on Education, 42(4), 1999.

10. Sridhar Alagar and Subbarayan Venkatesan, Techniques to Tackle State Explosion in Global Predicate Detection. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 27, NO. 8, AUGUST 2001, pp. 709-710.