行政院原子能委員會

委託研究計畫研究報告

核能儀控系統電腦化發展之安全功能認證研究

On Certification of Safety Functions in Computerized Nuclear Instrumentation and Control Systems Development

計畫編號：1002001INER027

受委託機關(構)：國立臺灣大學

計畫主持人：蔡益坤

聯絡電話：(02) 3366-1189

E-mail address：tsay@im.ntu.edu.tw

核研所聯絡人員：游原昌

報告日期：中華民國 100 年 11 月 29 日

目　錄

中文摘要

研究文獻與工業技術標準的演進顯示正規方法逐漸在電腦化系統安全功能認證上受到重視。基於核能儀控系統軟體的常見型態將會是即時多執行緒程式的考量，我們就這類軟體功能正確性驗證及執行時間分析之相關正規方法與工具作一整理與介紹。我們以一簡化但具代表性之控制器程式的驗證為範例，檢視運用以上正規方法與工具的可行性。我們進行範例程式驗證的步驟以及在各步驟中如何使用適切的方法與工具，也將是未來從事類似驗證工作者的重要參考。

關鍵詞：嵌入式系統、正規方法、霍爾邏輯、模型檢查、多執行緒程式、即時系統、安全攸關系統、靜態分析、時間量測分析、最差情況執行時間

Abstract

Research literature and the evolution of industrial standards have shown the increasing importance of formal methods in the certification of safety functions of computerized systems. Considering that nuclear instrumentation and control software will typically be real-time multithreaded programs, we review a selection of methods and tools that may be used to formally verify the functional and timing correctness of such software. We investigate as a trial case the formal verification of a simple yet representative controller program, in order to assess the adequacy of these methods and tools. The steps that we have followed and the methods and tools that we have applied should also provide a very good guidance for practitioners who will undertake similar verification tasks.

Keywords: Embedded Systems, Formal Methods, Hoare Logic, Model Checking, Multithreaded Programs, Real-Time Systems, Safety-Critical Systems, Static Analysis, Timing Analysis, WCET

壹、 計畫緣起與目的

1.1 緣起

雖然測試與模擬目前仍是電腦化系統安全功能認證所依賴的主要驗證手段，研究文獻 [Souyris et al. 2009] [Yoo, Jee, and Cha 2009] 與工業技術標準的演進 [IEC61508] [DO-178B/ED-12B]顯示正規方法（formal methods）逐漸受到重視。正規方法 [Wikipedia] [Formal Methods]泛指以嚴謹的數學及邏輯方法與工具進行軟硬體系統的需求分析、設計、開發與驗證，以確保系統正確無誤、符合規格。即使未全程使用正規方法，以嚴謹的數學及邏輯方法與工具驗證已撰寫完成的程式之正確性（即正規驗證）仍被視為確保軟體品質的最根本之道。測試與模擬雖然仍是目前驗驗軟體的主流，但是誠如計算科學先驅 Dijkstra 所言，測試僅能顯示程式有錯誤，卻無法證明程式確實沒有任何錯誤（Program testing can be used to show the presence of bugs, but never to show their absence! ） [Dijkstra 1970]。

然而，導入正規方法有相當高的技術門檻，也意味著相當高的人力與時間成本。正因為如此，正規方法目前主要仍應用於生命或安全攸關（life-critical or safety-critical）或財務攸關（financially-critical）的軟硬體系統，例如：醫療設備、飛機自動導航系統、太空船控制系統、車控系統、核電廠自動控制系統、作業系統核心、積體電路設計等。正規方法理論與應用的發展已有四十年以上的歷史，一九九零年代起便逐漸有顯著的

正規方法應用實例。歐美由於正規方法的蓬勃發展，甚至有評比各種正規方法的競賽或實驗，以瞭解不同方法的特色與適用性 [Abrial, Börger, and Langmaack 1996] [Brat et al. 2004]。與我科技水準相當的南韓在應用正規方法方面，似乎也已於幾年前開始起步 [Yoo, Jee, and Cha 2009]。

1.2 目的

基於正規方法應用於安全攸關軟硬體系統具有根本的優越性，本研究的主要目的便是在探討正規方法應用於國內自主型核能儀控系統軟體模組驗證之可行性，也希望藉此整理國內外正規方法發展安全軟體之應用實績與發展工具，以及更基礎的正規方法應用於安全軟體驗證之理論與技術。

核能儀控系統軟體的常見型態是即時多執行緒程式（real-time multithreaded programs），我們因此就這類軟體功能正確性驗證及執行時間分析之相關正規方法與工具作一整理與介紹。在程式功能正確性驗證部分，我們分別介紹利用時序邏輯模型檢查以及利用邏輯演繹推導來驗證多執行緒程式的兩種方法及相關的工具。在執行時間分析部分，我們介紹如何以靜態分析以及動態量測兩種不同方法估算單一工作或執行緒「最差情況執行時間」（Worst-Case Execution Time，WECT）；接著我們介紹，在單一執行緒執行時間已知的情形下，如何利用排程理論分析即時多執行緒程式的執行時間，相關的工具也一並介紹。

我們以一簡化但與核能儀控系統功能相近之化學反應爐溫

度控制器程式的驗證為例，檢視運用以上正規方法與工具的可行性。這個控制器程式範例具備即時與多執行緒等挑戰驗證技術的基本要素，具相當之代表性。透過實際驗證這個範例的經驗，我們更深入掌握模型檢查與演繹式方法互補的必要性，也體認到排程分析模型與即時多執行緒程式語意之關聯的確立需有更精準的方法從事。

1.3 功能正確性驗證之方法與工具

驗證軟體程式功能正確性的主要方法有兩種：模型檢查（model checking）和演繹式驗證（deductive verification）。模型檢查一般分兩階段進行，首先將系統所有可能的狀態（state）及狀態轉移建構成模型（model），並以時序邏輯（temporal logic）的表示式（formula）表述系統的性質，再運用相關演算法來自動檢查此模型是否滿足該表示式 [Clarke, Emerson, and Sistla 1986] [Clarke, Grumberg, and Peled 1999]。知名常用之工具如 SPIN [Holzmann 2003]。

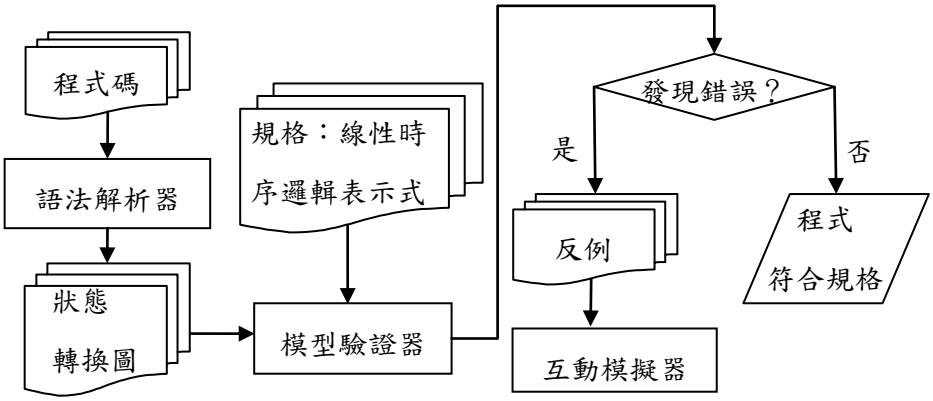


圖 1.3-1、模型檢查架構圖

演繹式程式驗證為透過演繹推導（deductive reasoning）來證明程式正確性的方法。常見建構方法如霍爾邏輯（Hoare logic）[Hoare 1969] [Apt, de Boer, and Olderog 2009]的程式邏輯理論以及陳述轉換理論（predicate transformer）。具使用者註解（program annotation）的原始碼首先轉換成一中介語言（intermediate

language），驗證條件產生器（verification condition generator）根據含有註解的中介語言程式碼產生驗證條件。有時驗證條件可自動化求解，代表工具如可滿足模組理論解題器（SMT solver）；無法自動化的驗證條件則需要驗證人員與工具互動來證明，工具如 Frama-C [Frama-C software analyzers]、VeriFast [Jacobs, Smans, and Piessens 2010]等。

我們的第一次及第二次期中報告(如附件一與附件二所示)有更多關於功能正確性驗證之方法與工具的資料。

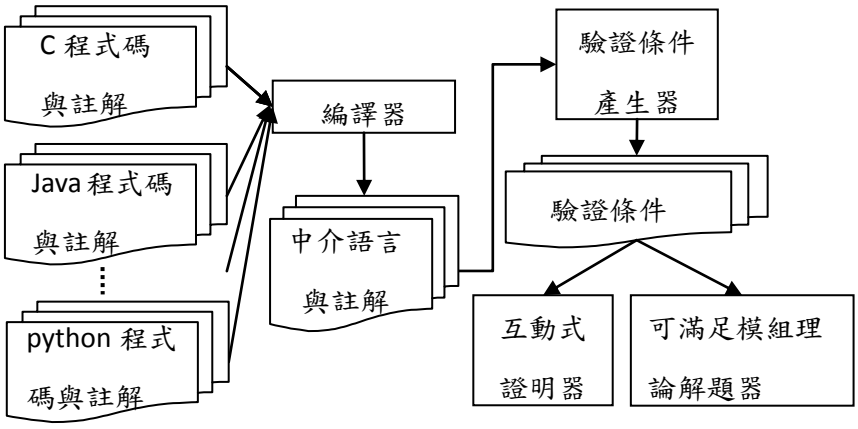


圖 1.3-2、演繹式程式驗證架構圖

1.4 時間正確性驗證之方法與工具

計算單一工作或執行緒的「最差情況執行時間」（Worst-Case Execution Time; WCET）的方法可分為兩大類：靜態分析（static analysis）和測量式（measurement-based）方法。

靜態分析方法不依賴在真正硬體或模擬器上執行程式，而是將程式碼與抽象的系統模型結合，藉此得到執行時間的上限。此方法可分多階段進行，透過數值分析（value analysis）

[Heckmann and Ferdinand 2004]決定處理器的暫存器（register）和區域變數值在每個程式點的範圍，控制流程分析（control-flow analysis）計算可能的執行路徑，處理器行為分析（processor-behavior analysis）預測記憶體、快取和管道（pipeline）等對於執行時間的影響，最後由估計計算（estimate caculation）決定 WCET 的估計值 [Wilhelm et al. 2008]。常見工具如 aiT [aiT]、Bound-T [Bound-T]、SWEET [SWEET]等。

測量式方法則以給定的輸入集合在硬體或模擬器上執行程式來測量執行時間，可用於提供實際執行時間變化的概況，但無法保證執行時間的上界。此方法可透過額外的插樁程式碼（instrumentation code）來收集時戳（timestamp）或 CPU 循環計數等方式進行測量 [Wilhelm et al. 2008]。常見工具如 RapiTime [RapiTime]等。

在所有單一執行緒執行時間已知的情形下，便可利用排程（scheduling）理論分析即時多執行緒程式的執行時間。排程可分兩階段進行，區域排程分析（local scheduling analysis）可計算每個任務的最差情況回應時間（Worst-Case Response Time; WCRT），組合系統層級分析（compositional system level analysis）可運用定點法（fixed point method）計算有反饋（feedback）之任務的 WCRT [Künzli et al. 2007]、 [Henia et al. 2005]。常見工具如 SymTA/S [SymTA/S]、RT-Druid [RT-Druid]。

我們的第一次及第二次期中報告(如附件一與附件二所示)有更多關於執行時間分析之方法與工具的資料。

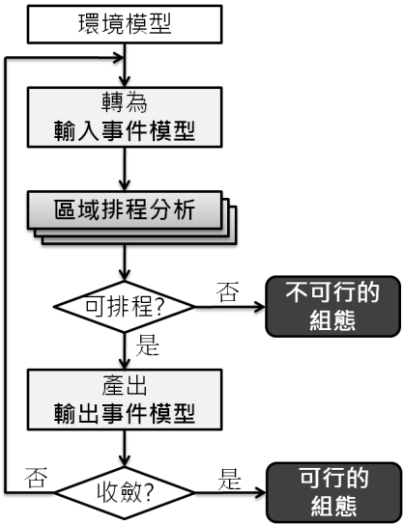


圖 1.4-1，組合系統層級分析示意圖 [Künzli et al. 2007]

1.5控制器範例及相關驗證工作

我們的驗證範例為一數位控制器，該控制器透過調整冷卻劑通過速率來控制化學反應爐的溫度。控制器以 C 語言實作，運作於與即時 POSIX 標準相容的系統，並符合工業程式撰寫標準 MISRA-C [MISRA-C 2004]。構成控制器之最基本函式如圖 1.5-1 所示，控制流程圖如圖三-2 所示，五任務皆為週期性執行之執行緒。

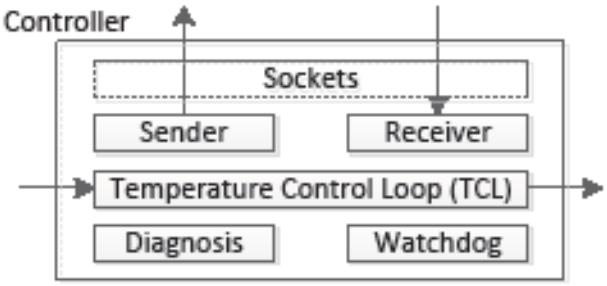


圖 1.5-1、控制器組成

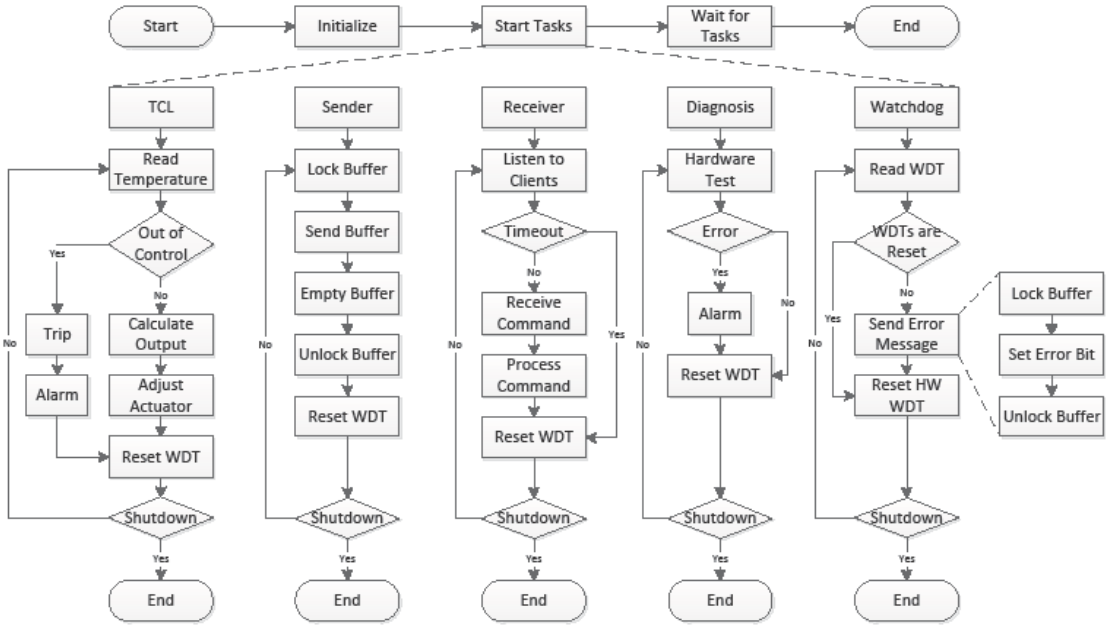


圖 1.5-2、控制器執行緒之控制流程圖

此控制器除了自我測試任務外，其他任務每 50 微秒須至少完成一次；自我測試任務則須每 8 小時至少完成一次。我們對其進行如圖 1.5-3 所示之驗證。

基本程式安全分析：透過靜態分析來確保控制器不會發生如空指標的參考行為（null-pointer dereferencing）、緩衝區溢位（buffer overflow）等程式安全原因而當機（crash）或是陷入死結（deadlock）等情形。

功能正確性驗證：驗證當最近四次取樣溫度呈遞增，且最後一次的溫度超過所設定的溫度高標時，溫度控制迴圈（Temperature Control Loop，TCL）會提高冷卻劑通過速率。當最近四次取樣溫度呈遞減，且最後一次的溫度低於所設定的溫度低標時，則會降低冷卻劑通過速率。此外亦須驗證每個任務

之計數器（counter）在每循環的最後皆會增加。

時間正確性驗證：對每一執行緒／任務作安全的 WCET 值預測，並進行排程分析來確認是否所有任務皆能在時限（其週期）內完成。

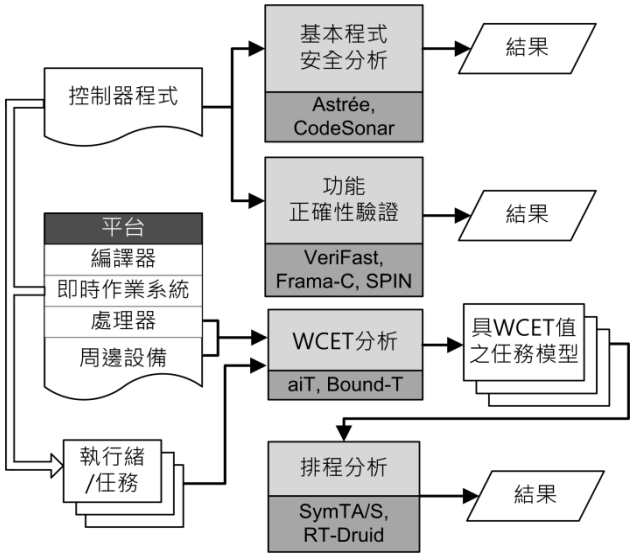


圖 1.5-3、控制器之相關驗證任務

1.6控制器功能正確性之驗證

程式的功能正確性可用兩種方法驗證，模型檢查和演繹式驗證，此章節闡述以模型檢查工具 SPIN 及演繹驗證工具 VeriFast，來對控制器範例作驗證之步驟。

(一) 模型檢查

模型檢查方法自動在抽象模型中搜索是否違反使用者定義的斷言（assertion），SPIN 模型以 PROMELA（PROcess MEta LAnguage）描述，部分屬性較演繹示方法容易表達，不可分割（atomic）的支援也較演繹式方法所使用的工具 VeriFast 佳，

且轉換成模型後為全自動，並支援 pthread 分享資源概念。下述如何在 SPIN 中建造控制器範例中 watchdog 任務的抽象模型，以及如何斷言正確性性質、處理多執行緒等。

TCL、receiver、sender、diagnosis 執行緒每循環皆會將相對應的全域檢查變數加 1，watchdog 執行緒則檢查各執行緒每循環後是否該變數值與上循環結束時不同，若相同則回報錯誤，表示該執行緒發生異常。將 watchdog 轉成 SPIN 模型時，考慮執行緒週期性觸發，增加 timer 如下。

```
active proctype timer()
{  int temp;

do
::
   TCL_c ! 1;
   receiver_c ! 1;

   TCL_c ? temp;
   receiver_c ? temp;

   watch_c ! 1;
   watch_c ? temp;

   od;
}
```

timer 用於同步執行緒，確保週期相同的 TCL、receiver 和 watchdog，在 watchdog 進行檢查時，TCL、receiver 至多循環一次。

TCL 轉換如下

```
active proctype TCL()
{
  int temp;
  do
  ::
```

```
  TCL_c ? temp;
        atomic{
            count1++;
            count1 = count1 % 2;
        }
        TCL_c ! 1;
  od;
}
```

上述模型中以 TCL_c ? temp 來等待 timer 傳送觸發訊息，並將檢查的變數值加 1 再除 2 的餘數計算設為不可分割，receiver 執行緒亦轉成相似模型。

watchdog 則轉換成如下的模型：

```
active proctype watchdog()
{
  int pre_ TCL,pre_receiver;
  pre_ TCL = -1;
  pre_ receiver = -1;
  int cur_ TCL,cur_ receiver;
  int temp;

  do
  ::
        watchdog_c ? temp;

        cur_TCL = count1;
        assert(pre_TCL != cur_TCL);
        pre_TCL = cur_TCL;

        cur_receiver = count2;
        assert(pre_receiver != cur_receiver);
        pre_receiver = cur_receiver;

        watchdog_c ! 1;
  od;
}
```

watchdog 以 watchdog_c ? temp 等待 timer 傳送觸發訊息，並用斷言 assert(pre_TCL != cur_TCL)和 assert(pre_receiver !=

cur_receiver)分別檢查各循環結束後與上循環結束時的檢查變數是否不同。若驗證通過，則代表若 TCL 和 receiver 執行緒沒有在循環後對相對應的檢查變數加 1，則 watchdog 必能檢查出異常發生。

(二) 演繹式方法

在使用演繹驗證工具前，通常需要作以下驗證步驟：改寫程式、塑模外部函數及註解程式。下述如何以 VeriFast 工具實作此三步驟。

1. 改寫程式

第一個限制為 VeriFast 只支援部分全域變數的使用，例如在迴圈不變量（loop invariants）中不可使用全域變數。解決方法為將所有全域變數置於以一 C 結構（struct）中，並以參數方式將其傳給所有執行緒。第二限制是對於非原始狀態型別（non-primitive）的堆疊上（onstack）變數，不同於 MISRA-C 中禁止堆積（heap）的使用，VeriFast 要求所有 C 結構和陣列須用 malloc 函式在堆積上創建，並以 free 函式刪除。

另一不支援的程式功能為對於一個不可分割（atomic）的變數，有單一寫入者和多讀取者。在有 GNU C 函式庫的 POSIX 系統中，int 變數可視為不可分割 [The GNU C Library manual]，因此當只有一寫入者時，不需要以互斥量（mutex）或鎖定（lock）來保護變數。控制器範例中 watchdog 的計數器便為不可分割之 int 變數。在 VeriFast 中，一執行緒對於堆積主幹（trunk）永遠有部分權限 f，f 為一介於 0（不包含）

和 1（包含）之間的實數。若一執行緒在一堆積主幹上有權限 1（預設為不設定），則可在該堆積主幹上讀取和寫入，否則此執行緒只能讀取堆積單元（cell）。當一執行緒配置（allocate）一堆積主幹，此執行緒得到在堆積主幹上的所有權限。此權限模型不支援對於不可分割變數有單一寫入者和多讀取者，因為將使完整權限值超過 1，因此我們必須在 VeriFast 中特別保護此類不可分割變數。此外，報告未提及所有限制。

2. 塑模外部函式

VeriFast 提供一套與標準 pthread 介面不同名稱和特徵（signature）的註解函式，對於未支援的系統呼叫和函式庫，我們以在前提條件（pre-condition）後置條件（post-condition）中說明規格（specification）來塑模其語意，而不提供原始碼。以下例子為對系統呼叫 open 之規格。

```
/*@ prediate on(int flags, int flag) = true ==
    ((flags & flag) == flag); @*/

/*@ predicate opened(int fd); @*/

int open(char *pathname, int flags);
    /*@ requires [?f]chars(pathname, ?cs) &*& mem
        ('\0', cs) == true &*& on(flags,
    O_RDONLY | O_WDONLY | O_RDWR); @*/
    /*@ ensures result >= -1 &*& (result == -1 ?
        true : opened(result)); @*/
     }
```

VeriFast 註解以/*@和@*/包夾作區塊註解，或是以//@

開頭作單行註解。記號&*&表示在分離邏輯（separation logic）中的分離連接詞（separating conjunction） [Reynolds 2002]。前提條件需要 pathname 指向一最後一個可用元素包含零值（null-terminated）的字串，且此執行緒對於字串擁有權限 f（f 大於 0），flags 包含存取模組其中一個：O_RDONLY、O_WRONLY 或 O_RDWR。因為 open 運算不改變路徑名稱，僅需要非負值的權限進行讀取。後置條件確保當錯誤發生時會回傳-1。述詞（predicate）opened(result)表示檔案描述器（file descriptor）result 已關連於一開啟的檔案。為了確保在每個執行路徑中，close 運算必在一成功的 open 運算後發生，我們指定 close 運算為述詞 opened 之消耗者（consumer）。

```
int close(int fd);
//@ requires opened(fd);
//@ ensures true;
```

在 VeriFast 中，述詞被視為堆積主幹，因此當檔案描述器在一執行路徑中開啟但後未關閉時，會回報記憶體洩漏（memory leak）。

以互斥量來同步執行緒在 VeriFast 中已塑模，對於互斥函式的規格如下。

```
struct mutex *create_mutex();
//@ requires create_mutex_ghost_arg(?p) &*& p();
//@ ensures mutex(result, p);

void mutex_acquire(struct mutex *mutex);
//@ requires [?f]mutex(mutex, ?p);
//@ ensures mutex_held(mutex, p, currentThread, f) &*& p();
```

```
void mutex_release(struct mutex *mutex);
//@ requires mutex_held(mutex, ?p, currentThread, ?f)
    &*& p();
//@ ensures [f]mutex(mutex, p);

void mutex_dispose(struct mutex *mutex);
//@ requires mutex(mutex, ?p);
//@ ensures p();
```

述詞 p 代表被互斥量保護的資源之不變量 m，創建互斥量需要不變量和保護此不變量於一述詞 mutex(m, p)，此述詞可被分為多副本，各有小於 1 之權限，每副本被一執行緒所有。當一執行緒成功得到此互斥量，且在釋放互斥量前被重設，則此不變量成立。以任務 sender 的共享緩衝記憶體（shared buffer）為例，其不變量如下。

```
g->buffer |-> ?b &*& g->buffer_size |-> ?s &*& chars(b, ?cs)
    &*& malloc_block(b, s) &*& length(cs) == s
```

g 為含有所有全域變數之結構，s->f |-> ?v 表示 v 值被存於 s 結構的 f 中，chars(b, ?cs) &*& malloc_block(b, s) &*& length(cs) == s 表示 b 指向長度為 s 且型態為 char 的堆積區塊 cs。不變量中的述詞愈多，任務 sender 愈能了解緩衝記憶體的狀態。例如一安全性質（safety property）可能需要每當任務 sender 傳送帶有第 i 個錯誤位元成立之緩衝記憶體時，對應的錯誤 err 曾在之前發生，此特性可用下式設定並加入共享緩衝記憶體之不變量中。

```
nth(i, cs) != 1 || err == 1
```

但因 VeriFast 並不支援時序性質（temporal property），

因此無法設定活化性質（liveness property），像是每當錯誤 err 發生，則 sender 所送出之緩衝記憶體終將有相對應的錯誤位元設為成立。

3. 註解程式

wairOnSocket 函式接收兩參數，socket 連接的檔案描述器和 timeout 值，此函式等待從 socket 連接中傳入資料直到所設定時間的到期，並回傳等待的結果。當時間到期時回傳 0，錯誤發生時回傳-1，此函式的規格如下。

```
int waitOnSocket(int fd, int timeout)
    /*@ requires fd >= 0 && timeout >= 0; @*/
    /*@ ensures result == 0 || result == -1; @*/
```

迴圈不變量是演繹方法的重點，VeriFast 無法自動化推論足夠的迴圈不變量，因此需要明確地設定。例如任務 receiver 仰賴伺服器 socket 連接 sockfd 來接收客戶的連接，此伺服器 socket 連接在運行中不應改變，為了設定此性質，需要運用幽靈變數（ghost variable）。

```
//@ int SOCKFD = sockfd;
while(STATE == STATE_RUNNING)
/*@ invariant sockfd = SOCKFD; @*/
{
    ...
}
```

在進入 receiver while 迴圈中的週期性任務前，在註解中創建幽靈變數 SOCKFD 並設定值同 sockfd，則迴圈不變量為每次迴圈循環後 sockfd 仍然同於 SOCKETFD。

VeriFast 自動化產生驗證條件來檢查是否存在緩衝區溢

位和指標錯誤。一缺點為整數上下限固定，而非依平台而定。

以下討論控制器範例的正確性性質，當連續三次取樣溫度皆為漸增，且最後一次溫度超過高標 HIGH TEMPERATURE，則提高冷卻劑通過速率，此性質的正規描述如下：設 t_i 為第 i 個取樣時間點（$t_i < t_j$ iff i < j），temp[t]表示在時間 t 時測量的平均溫度，cool[t]表示時間 t 時的冷卻劑通過速率。對於所有連續四次取樣時間點 t_i、t_{i+1}、t_{i+2}和 t_{i+3}，若 $temp[t_i] < temp[t_{i+1}] < temp[t_{i+2}] < temp[t_{i+3}]$，且 $temp[t_{i+3}] >$ HIGH TEMPERATURE，則 $cool[t_{i+3}] > cool[t_{i+2}]$。但控制器並不記錄最近四次溫度，而是紀錄最近一次溫度和在目前時間點之前連續上升次數，若目前溫度高於前次則計數變數 pos 增加，否則重設為 0。

另一問題為目前的驗證工具通常支援在程式碼特定行列中作變數參考，例如 x@l 表示在標籤 l 的 x 值，但無法參考至一特定迴圈循環中的變數，例如 temp[ti]。解決方法為使用幽靈變數 $temp[t_i]$、$temp[t_{i+1}]$、$temp[t_{i+2}]$和 $temp[t_{i+3}]$，來證明目標性質與幽靈變數和計數變數 pos 間的關係，以下闡述如何插入幽靈變數和設定幽靈變數與其他變數間的關係。

```
int size = 4;
/*@
predicate is_zero(int e) = e == 0;

lemma int* create_ghost(int size);
```

```
    requires size > 0;
    ensures ints(result, size, ?is) &*& foreach(is, is_zero);

lemma void set_ghost(int *arr, int idx, int value);
    requires ints(arr, ?size, ?is1) &*& idx >= 0 &*& idx < size;
    ensures ints(arr, size, ?is2) &*& is2 ==
        update(idx, value, is1);

predicate consecutive(list<int> is, int size, int curr, int idx,
        nat pos) =
    switch(pos) {
        case zero: return curr == (idx-1)%size || nth(idx, is) <=
            %nth((idx-1)% size, is);
        case succ(n): return curr != (idx-1)%size &*& nth(idx,
            is) > %nth((idx -1)%size, is) &*& consecutive(is,
            size, % curr, idx-1, n);
    };
@*/

//@ int prev = 0;
//@ int curr = 0;
//@ int *temp = create_ghost(size);
while(STATE == STATE_RUNNING)
/*@ invariant ints(temp, size, ?is) &*& 0 <= prev &*& prev <
        size &*& 0 <= curr &*& curr < size @*/;
{
  //@ prev = curr;
  //@ curr = (curr + 1) % size;
  prev_temp = curr_temp;
  //@ set_ghost(temp, prev, curr_temp);
```

```
    ...
    curr_temp = temp1 + temp2;
    //@ set_ghost(temp, curr, curr_temp);

    pos = curr_temp > prev_temp ? (pos + 1) % size : 0;
    //@ assert consecutive(is, size, curr, curr, nat_of_int(pos));
    ...
}
```

輔助定理 create_ghost 用於創建幽靈變數陣列 temp 來追蹤最近四次溫度測量，set_ghost 用於更新溫度陣列的元素，遞迴述詞 consecutive 用於連結溫度陣列和記數變數 pos，來確保 pos 計算正確。

1.7控制器時間正確性之驗證

為了確保時間的正確性，首先計算每個執行緒之最差情況執行時間（WCET），接著以所算出的 WCET 計算整個控制器範例的最差狀況反應（WCRT），最後將開銷（overhead）的影響計算至 WCRT 結果中。我們假設 WCET 工具為 aiT，WCRT 工具為 SymTA/S。

(一) 計算 WCET

可分以下幾步驟來計算每個執行緒的 WCET 值，分割及編譯程式、註解程式和實行工具。

1. 分割及編譯程式

為了個別計算任務，我們將控制器程式分成六部分，每部分為獨立執行緒，分別為 main、TCL、receiver、watchdog、

sender 和 diagnosis。每部分的程式碼需含有該任務所需的所有資訊，例如須包含標頭檔（header file）和所呼叫的全域函式。在控制器範例中，main 程式碼包含其標頭檔和自身，而其他使用 error 函式來傳送錯誤訊息之執行緒，須包含在 TCL 任務中的 error 函式程式碼，以及包含在 error 函式所呼叫在 sender 任務中的 enable 函式。

除了 main 執行緒外，所有其他執行緒皆視緩衝記憶體為共享變數，鎖定機制使程式確保同一時間只會有一個執行緒存取緩衝記憶體，鎖定指令為非搶先式（non-preemptive），而其他指令為搶先式（preemptive），因此需要使用不同的排程方法，故我們將每個執行緒畫分為三部分，分別套用非搶先式、搶先式、非搶先式排程。

排程分析時需要每部分的 WCET 值，第一部分的程式碼為從 while 迴圈的開頭到鎖定指令前，第二部分為鎖定指令，第三部分包含從鎖定指令後到迴圈結束前之指令。例如 TCL 任務的第一部分為：

```
prev_temp = curr_temp;
...
if (curr_temp > MAX_TEMPERATURE) {
    trip();
}
```

第二部分為：

```
alarm(ERR_TEMPERATURE);
```

第三部分為：

```
if (pos >= 3 && curr_temp > HIGH_TEMPERATURE) {
    increase(actuator);
```

```
}
...
waitForNext(TCL_PERIOD);
```

任務 receiver 在兩分支上皆呼叫鎖定函式 error。

```
if (retval == -1)
    error(ERR_RECEIVER_SOCKET_FAILED);
else if (retval == 1) {
if ((cli_sockfd = accept(r_sockfd, (struct sockaddr *)
        &cli_addr, sizeof(cli_addr))) == -1) {
    error(ERR_RECEIVER_SOCKET_FAILED);
    ...
```

實際上 receiver 只會根據條件執行其中一分支，因此首先計算整個 receiver 的 WCET 值，從結果將能得知哪一分支有較長的執行時間，接著用註解設定使其執行該分支的條件為真，藉此將程式分成三部分。例如假設我們從結果得知 if 分支有較長的 WCET 值，則第一部分切割如下：

```
retval = waitOnSocket(r_sockfd, 1);
 if (retval == -1)
```

第二部分為

```
error(ERR_RECEIVER_SOCKET_FAILED);
```

第三部分為

```
else if (retval == 1) {
...
waitForNext(RECEIVER_PERIOD);
```

分割後，編譯每部分的程式碼來得到可執行檔。我們使用 GCC 編譯控制器（GNU 作業系統上的編譯器；GNU Compiler Collection，GCC）來編譯程式。

2. 註解程式

執行檔資訊通常不足以算出準確的 WCET 值，因此需要註解的幫助以提供工具更多資訊。例如註解所使用的編譯器可幫助 aiT 重建控制流程（control flow）。

/* ai: compiler "arm-gcc"; */

執行緒 receiver 以 while 迴圈作週期性運行，因為在迴圈之外的程式非週期性運行，而我們著重於多週期性執行緒間的時間行為，因此只計算週期性程式片段的 WCET 值。為了驗證所有任務皆能在 50 毫秒內完成一循環，我們使用註解幫助計算迴圈運行一次之 WCET 值。

```
while(STATE == STATE_RUNNING) {
/* ai: loop here begin exactly 1; */
    ...
    waitForNext(RECEIVER_PERIOD);
}
```

在計算完一次迴圈循環的 WCET 後，如分割及編譯程式章節所提，我們使用註解設定使 WCET 值較大分支的條件為真。例如若 receiver 的 if 分支有較長的執行時間，則以註解設定變數 retval 值為-1。

```
if (retval == -1)
    /* ai: instruction here is entered with
        retval=-1; */
        ...
```

當 aiT 無法靠數值分析得到暫存器（register）值時，同樣需要註解來幫助運算。例如在任務 diagnosis 中，需要註解介面（interface）的個數，因其依系統而異。

```
for (interface = interfaces; interface != NULL; interface =
        interface->ifa_next) {
    /* ai: instruction here is entered with
    interfaces=7; */
```

以上註解代表介面值在該位址永遠為 7。

(二) 計算 WCRT

在計算完所有執行緒之 WCET 值後，我們使用排程分析來得到控制器程式的 WCRT。使用工具的步驟如下：建造系統模型、詢問 WCET 和實行工具。

1. 建造系統模型

在控制器範例中，任務有三種不同的週期，任務 diagnosis 的週期最長，TCL、receiver 和 watchdog 的週期相同，而 sender 週期為上述三任務的兩倍長。優先度愈高的任務有愈短的週期，因此我們使用比率單調排程（rate-monotonic scheduling） [Liu and James 1973] [Lehoczky, Sha, and Ding 1989]。

因為任務 main 只執行一次，我們將其排除在模型外。執行緒的優先序如下：TCL = receiver = watchdog > sender > diagnosis。

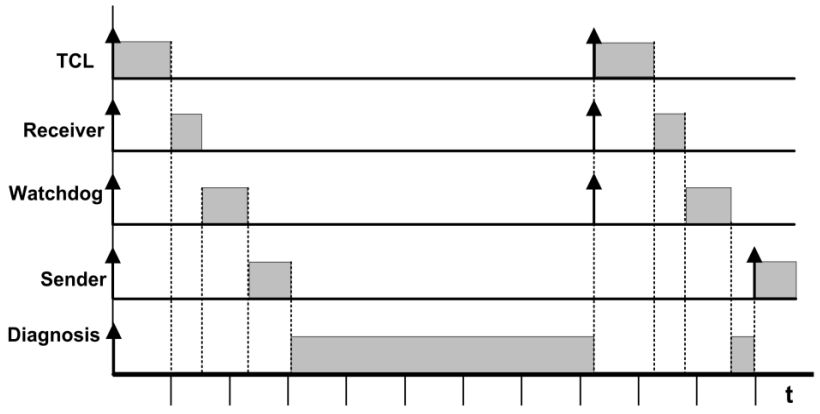


圖 1.7-1、五週期性執行緒的可能排程

圖 1.7-1 表示一無抖動（jitter）的可能執行序列，灰色方格代表執行緒一週期的執行時間，向上的箭頭表示每個週期的觸發時間。

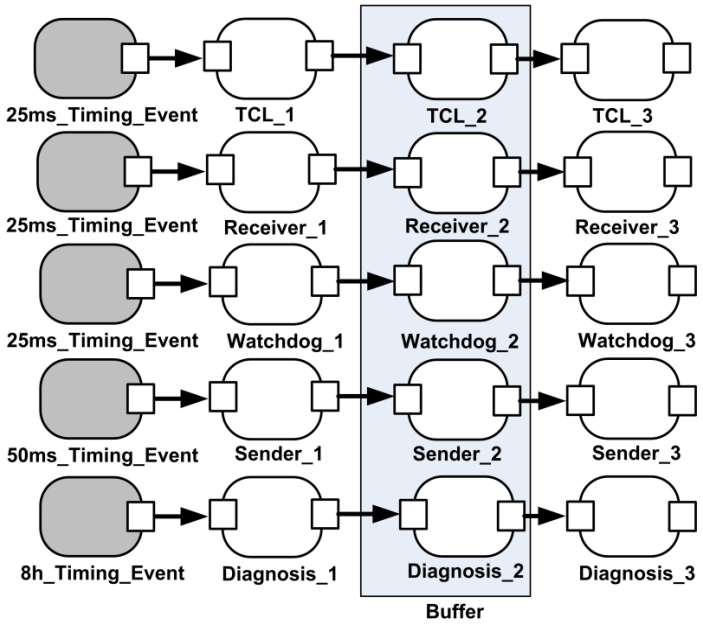


圖 1.7-2、控制器範例的系統模型

圖 1.7-2 為控制器範例的系統模型，此為示意圖而非工具畫面截圖。灰色方型代表輸入資源，也就是輸入事件模型（input event model），白色方型代表任務模型。五執行緒分別被五個不同週期性時間事件觸發。因為共享緩衝記憶體的鎖定機制，可能在鎖定指令時發生優先權倒轉（priority inversion），當低優先權的執行緒進入鎖定區域，則可阻斷（block）其他高優先權的執行緒執行。

因此我們將每個執行緒拆成三部分，任務模型名稱帶有_1 為第一部分，代表鎖定指令前的部分，帶有_3 為第三部

分，代表鎖定指令後的部分，上述兩部分皆為搶先式，因此我們套用比率單調排程。名稱帶有_2 為第二部分，代表鎖定區塊，圍著所有鎖定區塊的灰色大方框表示所有執行緒共享同樣的緩衝記憶體。鎖定區塊為非搶先式，我們套用固定優先序非搶先式排程。

2. 詢問 WCET 並實行工具

SymTA/S 提供與 aiT 的整合，在 SymTA/S 介面中，可直接運行 WCET 計算。在我們提出的系統模型中，第二部分在每個循環皆會執行，但實際上系統設定為當錯誤發生才寫入緩衝記憶體，而我們無法知道錯誤何時發生，因此計算最差情況，也就是每個循環皆會發生所有錯誤。若能驗證在此最差情況下，模型仍能滿足時間限制，則控制器程式將永遠滿足時間限制。

(三) 計算開銷

程式可能因上下文切換（context-switch）、共享快取、記憶體干擾（memory interference）等而產生開銷，因此實際週期性的執行緒帶有抖動，當優先權低的執行緒先於高者執行時，搶先便可能發生。

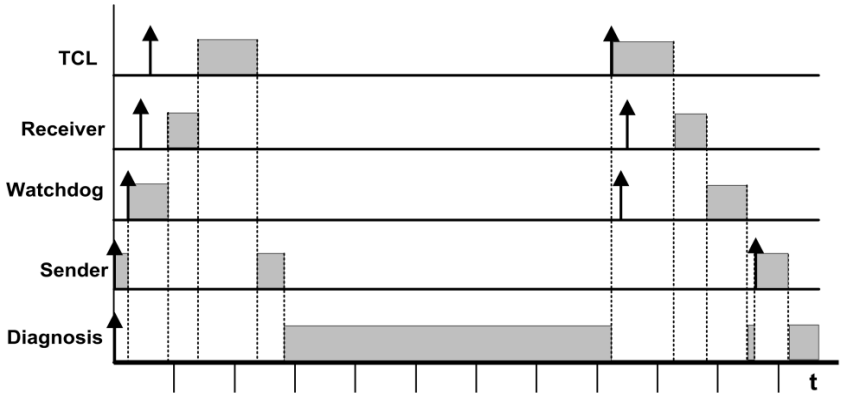


圖 1.7-3、帶抖動的五週期性執行緒的可能排程

圖 1.7-3 表示了一搶先發生的例子，若 TCL、receiver 和 watchdog 的觸發時間因抖動而延遲，sender 將會執行，但當高優先權執行緒觸發後，sender 執行權將被奪走。

使用 aiT 和 SymTA/S 工具計算開銷時，首先於 SymTA/S 介面向 aiT 要得各執行緒的 WCET 值和任務切換成本，aiT 中使用有用快取區塊（useful cache block，UCB）分析來計算任務切換成本，算出的 UCB 會存於 XTC（XML Timing Cookie）格式中 [INTERESTED project]，兩工具透過此格式進行連結。

計算出的 UCB 將以停止開銷（termination overhead）角色加至所有下個有較高優先權的任務，也就是任務的 WCRT 將加上被搶先低優先權任務的停止開銷。SymTA/S 輸出的結果可以甘特圖（Gantt chart）或數值表示。

參、 主要發現與結論

就我們所考慮的代表範例—即時多執行緒控制程式而言，目前的正規驗證方法與工具已能將許多驗證工作自動化，顯著節省人力與時間。我們進行範例程式驗證的步驟以及在各步驟中如何使用適切的方法與工具，將是未來從事類似驗證工作者的重要參考。在驗證過程中，有兩方面的發現我們認為特別值得注意：

一、 模型檢查與演繹式方法之互補

多執行緒程式功能正確性的模型檢查有相當的難度，加上即時性的需求條件後，驗證的難度更高，即便是執行緒個數固定的情況。學研界仍持續地努力研發更節省記憶空間、更有效率的方法與工具。此外，傳統上模型檢查方法針對系統的設計階段，較少考慮複雜的程式語言要素，因此往往需要人工來做程式與模型之間的轉換。相對的，演繹式的驗證方法有其廣用性，也較能因應複雜的程式語言要素，但並非全自動化，輔以模型檢查自動化的便利性是這類方法能否被實務界接受的關鍵。

二、 排程分析模型與即時多執行緒程式語意之關聯

排程分析時所假設的任務模型應依據實際即時多執行緒程式的語意，而即時多執行緒程式之語意又需依賴即時作業系統排程等行為之語意模型。然而，一般即時作業系統缺乏排程等行為之精準模型。這也是造成目前以靜態分析為主要手段的執行時間估算方法與工具，無法掌握特定即時作業系統排程行為的主因。為即時作業系統排程等行為建立精準語意模型並驗證其正確性，以做為程式執行時間分析之依據，應是根本解決之道。

正規驗證方法與工具雖未臻完美，但仍在持續進步當中，未

來所能發揮的效益應當更為顯著。我們希望以本計畫之經驗為基礎,著手進行實務上較大型且功能較複雜的自動控制程式之驗證，協助確認其正確無誤地達成應有之安全功能。

肆、 參考文獻

1. [The GNU C Library manual] The GNU C Library manual. Free Software Foundation
2. [Abrial, Börger, and Langmaack 1996] J.-R. Abrial, E. Börger, and H. Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, 1996.
3. [aiT] aiT. Available at: http://www.absint.com/ait/
4. [Apt, de Boer, and Olderog 2009] K.R. Apt, Frank S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs (Third Edition)*, Springer, 2009.
5. [Bound-T] Bound-T. Available at: http://www.bound-t.com/
6. [Brat et al. 2004] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, vol 25(2-3), pp. 167-198, 2004.
7. [Clarke, Emerson, and Sistla 1986] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, vol 8, no. 2, pp. 244-263, 1986.
8. [Clarke, Grumberg, and Peled 1999] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*, The MIT Press, 1999.
9. [Dijkstra 1970] E.W. Dijkstra. Notes on structured programming. *EWD 249*, 1970.
10. [DO-178B/ED-12B] DO-178B/ED-12B. Software Considerations in Airborne Sstems and. , RTCA/EUROCAE, .
11. [Formal Methods] Formal Methods. Available at: http://formalmethods.wikia.com/wiki/Formal_methods
12. [Frama-C software analyzers] Frama-C software analyzers. Available at: http://frama-c.com/

13. [Heckmann and Ferdinand 2004] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH, 2004. http://www.absint.com/wcet.htm.
14. [Henia et al. 2005] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis-the SymTA/S approach. *IEE Computers and Digital Techniques*, vol 152, no. 2, pp. 148-166, 2005.
15. [Hoare 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, vol 12, no. 10, pp. 576-580, 1969.
16. [Holzmann 2003] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
17. [IEC61508] IEC61508. Available at: http://www.iec.ch/functionalsafety/
18. [INTERESTED project] INTERESTED project. Available at: http://interested-ip.eu/
19. [Jacobs, Smans, and Piessens 2010] B. Jacobs, J. Smans, and F. Piessens. A Quick Tour of the VeriFast Program Verifier. *APLAS 2010*, vol LNCS, no. 6461, pp. 304-311, 2010.
20. [Künzli et al. 2007] S. Künzli, A. Hamann, R. Ernst, and L. Thiele. Combined approach to system level performance analysis of embedded systems. *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pp. 63-68, 2007.
21. [Lehoczky, Sha, and Ding 1989] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm exact characterization and. *Proceedings of Real Time Systems Symposium 1989*, pp. 166-171, 1989.
22. [Liu and James 1973] C. L. Liu and W. Layland James. Scheduling agorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, vol 20, pp. 46-61, 1973.
23. [MISRA-C 2004] MISRA-C. MISRA-C: 2004 --- Guidelines for the Use of the C Language in Critical Systems. MIRA Limited, 2004.
24. [RapiTime] RapiTime. Available at: http://www.rapitasystems.com/rapitime
25. [Reynolds 2002] J. C. Reynolds. Separation logic: a logic for shared mutable

data structures. *LICS 2002*, pp. 55-74, 2002.

26. [RT-Druid] RT-Druid. Available at: http://www.evidence.eu.com/content/view/28/51/
27. [Souyris et al. 2009] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. *Proceedings of the 2nd World Congress on Formal Methods*, vol LNCS 5850, pp. 532-546, 2009.
28. [SWEET] SWEET. Available at: http://www.mrtc.mdh.se/projects/wcet/sweet.html
29. [SymTA/S] SymTA/S. Available at: http://www.symtavision.com/symtas.html
30. [Wikipedia] Wikipedia. Available at: http://en.wikipedia.org/wiki/Formal_methods
31. [Wilhelm et al. 2008] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, vol 7, no. 3, pp. 36-1, May 2008.
32. [Yoo, Jee, and Cha 2009] J. Yoo, E. Jee, and S. Cha. Formal Modeling and Verification of Safety-Critical Software. *Software, IEEE*, vol 26, no. 3, pp. 42-49, 2009.

附件一：國內外以正規化方法發展安全軟體之應用實績與開發工具

核能儀控系統電腦化發展之安全功能認證研究

On Certification of Safety Functions in Computerized Nuclear Instrumentation and Control Systems Development

第一次期中報告：**國內外以正規化方法發展安全軟體之應用實績與開發工具**

計畫編號：1002001INER027
計畫參與人員：蔡益坤、蔡明憲、林靖婕、劉啟祥
所屬單位：國立台灣大學

摘要

本第一次期中報告提供可應用於安全攸關嵌入式應用系統之驗證工具的概觀，我們主要考量的為多工暨多執行緒的即時系統。在工具選擇上，不可避免地無法窮舉列出；我們根據邏輯正確性與時間正確性這兩項檢驗目標，將這些工具分成兩大類。這些工具在原理上都基於模型檢查或靜態分析方法。若能適當地結合這些工具，對達成安全攸關即時應用系統正確性的高度信賴將有很大的助益。

關鍵詞：嵌入式系統、正規方法、模型檢查、即時系統、安全攸關系統、靜態分析、時間量測分析、最差情況執行時間（WCET）

Abstract

This first midterm report gives a survey of tools that may be used to verify multitasking, or multithreaded, real-time systems intended for safety-critical embedded applications. The selection of tools, inevitably partial, is grouped along the two dimensions of logical correctness and timing correctness. All of the tools are in some way based on model checking or static analysis methods. When adequately combined, they can greatly help achieve high confidence in the correctness of a safety-critical real-time application.

Keywords: Embedded Systems, Formal Methods, Model Checking, Real-Time Systems, Safety-Critical Systems, Static Analysis, Timing Analysis, WCET

中華民國 100 年 4 月 20 日

2

目　　錄

附圖目錄

1 背景與目的

根據我們的瞭解，電腦化核能儀控系統中常用的控制器主要為一種多功（multitasking）即時（real-time）嵌入式（embedded）系統，與用於自動操控飛機、太空船或汽車的安全攸關（safety-critical）即時嵌入式系統類似。若以高階程式語言實作，則控制器的主要軟體呈現為多執行緒（multithreaded）程式的型態。這種系統的邏輯正確性自然須仰賴程式本身的正確設計，而即時性質的確保則主要靠適當的即時作業系統的排程並搭配適當的處理器。邏輯正確性與時間正確性的驗證原理有部分共通之處，嚴謹的邏輯正確性分析往往可以協助排除不可能的系統狀態，獲致更精準的程式碼執行時間之估算。雖然這兩種正確性彼此相關，但實際進行分析驗證時，通常分別運用不同的工具完成，邏輯工具的分析驗證結果往往成為時間工具輸入的一部份。如此，能讓不同的工具各擅勝場，也有利於驗證工作的分工。

基於以上考量，我們在這第一次期中報告中，蒐集整理了一些可應用於安全攸關即時嵌入式系統的多執行緒程式邏輯驗證工具和「最差情況執行時間」（WCET）分析工具，相關的應用實績也一併簡要說明。這些工具在原理上都基於模型檢查（model checking）或靜態分析（static analysis）方法。若能適當地結合這些工具，對達成系統正確性的高度信賴將有很大的助益。

2 多執行緒程式邏輯的自動化驗證工具

若能預先確知執行緒之個數，而每一執行緒又均為有限狀態或可抽象化為有限狀態時，多執行緒程式邏輯正確性的驗證是有可能全自動化的。目前已有根據模型檢查理論所建構的實驗性驗證工具，可從事這類的驗證工作。所謂模型檢查的進行簡要來說可分為兩個階段：第一個階段是將系統所有可能的系統狀態（state）及狀態轉移圖建構出來，這就是所謂的模型（model）。第二個階段則是以時序邏輯（temporal logic）的邏輯表示式（formula）表述系統的性質並運用其相關演算法來自動檢查此模型是否滿足該表示式。時序邏輯是用來表述與事件發生順序相關之性質的邏輯，我們可以利用時序邏輯表示我們所要檢查的系統行為。如果一個系統的模型滿足一個時序邏輯表示式，則意謂著這個系統的每一個狀態改變路徑均滿足此表示式。

因此，我們可以利用以此理論為基礎實作的工具來檢查系統的行為是否有達成我們的目的，以及系統確實沒有做出我們不希望它有的行為，例如死結等等。

2.1 SPIN

SPIN（Simple Promela Interpreter）[Holzmann 2004]是一個基於模型檢查理論、開放原始碼的正規驗證工具，由當初在貝爾實驗室的 Gerard Holzmann 為首的研究團隊所開發。SPIN 在 2001 年 4 月獲得了 ACM 的 Software System Award。

此工具主要支援一種叫做 PROMELA （PROcess MEta LAnguage）的高階系統模型描述語言，以及支援以 LTL（Liner Temporal Logic；時序邏輯的一種）為表示式的正規驗證。

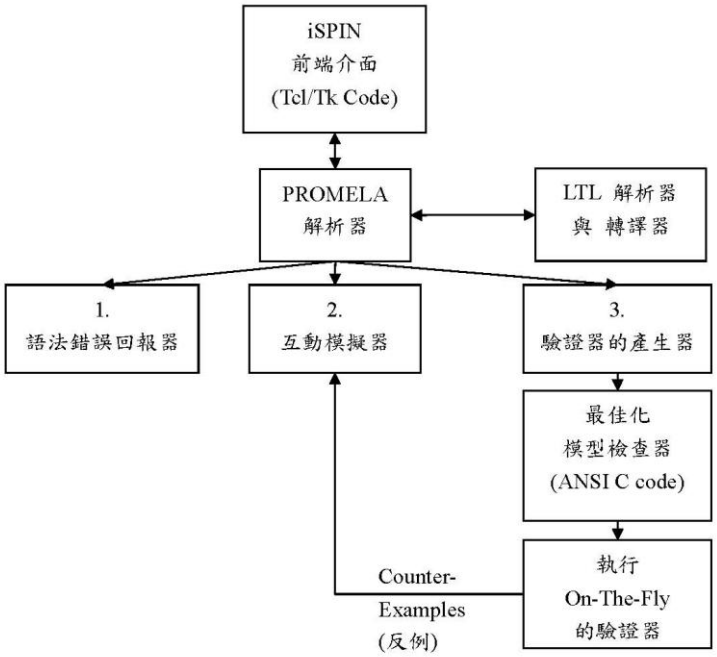


圖 2-1：SPIN 的架構圖

我們在驗證多執行緒程式時有兩種做法，第一種是把原本的 C 程式碼都轉成 PROMELA 的形式，例如把範例(1)的 Peterson 演算法 C 程式碼實作，轉化成為範例(2)的 PROMELA 格式。

範例(1)
```
static int sh_f0, sh_f1, sh_last;
    void * run_thread0 () {
        struct pa_desc d;
        d.f0 = &sh_f0;
        d.f1 = &sh_f1;
        d.last = 1;

        for (;;) {
            *(d.f0)=1;
            sh_last=d.last ;
```

```
            while (*(d.f1)==1 && (last==d.last)) {
                ; /* busy wait */
            }
            /* critical section */
            d.f0=0;
        }
    }
```

範例(2)
```
bool turn, flag[2];
    active proctype run_thread0 () {
      again:
        flag[0] = 1;
        turn = 1;
        (flag[1] == 0 || turn == 0) ->
        /* busy wait */
        /* critical section */
        flag[0] = 0;
         goto again;
    }
```

很明顯地，此方法對於直接驗證已經實作完的原始碼很不方便，因為我們必須要把原本的程式碼人工轉換成同樣行為的 PROMELA 來驗證。而這樣隱含著幾件事情：

(1) 如果人工轉換的過程發生錯誤，驗證的結果與原本程式碼的行為便會不一致。

(2) 原始碼有任何修改就要再重新轉換至 PROMELA。

第二種驗證方法是模型驅動軟體驗證（Model-Driven Software Verification） [Holzmann and Joshi 2004]，此方法是在 PROMELA 中嵌入 C 的原始碼。請參考以下範例，其中範例(4)的 PROMELA 是呼叫範例(3)中 Peterson 演算法的 C 原始碼。

範例(3)
```
static int sh_f0, sh_f1, sh_last;
    void * run_thread0 () {
        struct pa_desc d;
        d.f0 = &sh_f0;
        d.f1 = &sh_f1;
        d.last = 1;
        for (;;) {
            *(d.f0)=1;
            sh_last=d.last ;
            while (*(d.f1)==1 && (last==d.last)) {
                ; /* busy wait */
            }
            /* critical section */
```

```
                d.f0=0;
            }
        }
```

範例(4)

```
c_decl {
        extern void * run_thread0 (void *);
        extern void * run_thread1 (void *);
};
...
active proctype main() {
        init() ;
        do
        :: choose(thread0) -> c_code {run_thread0 (); }
        :: choose(thread1) -> c_code {run_thread1 (); }
        ...
        od
}
```

此方法雖然可以不用轉換原始碼，但是對於驗證的進行卻有以下幾點困難[Joshi and Zaks 2008]：

(1) 在 C 程式碼中的控制點檢查性質。
(2) 在 C 的函式中對控制流程發出中斷。
(3) 對於 interleaving 的處理不佳。

因此下一節我們將介紹另外一個可以跟 SPIN 一起運作的工具--PANCAM 來改善這些缺點。

2.2 PANCAM

PANCAM 是 2008 年在美國 NASA 噴射推進實驗室（JPL）的研究員 Rajeev Joshi 和在紐約大學的博士生 Anna Zaks 所提出的一套工具[Joshi and Zaks 2008]。此工具可以說是 SPIN 的輔助工具，並且不必對 SPIN 做任何的修改。PANCAM 可以增加 SPIN 所做不到的功能，如多核心檢查。

原生的 C 程式語言並不支援多執行緒的功能，因此必須依賴作業系統本身所提供的系統呼叫（system call）來產生執行緒。此工具目前只支援標準的 pthread 函式庫。

為了解決原本 SPIN 在驗證多執行緒程式上的問題，PANCAM 使用 LLVM（Low-Level Virtual Machine）這項技術，把 C 的原始碼轉換成 LLVM 的 Byte Code 以便將多執行緒的行為真實地抓取出來。SPIN 利用 PANCAM 提供的函式 pan_step()來得到新的狀態（state），而 PANCAM 則讓虛擬機器執行 Byte Code 並回傳新的狀態，如圖 2-2 所示。所以，SPIN 可以透過虛擬機器的狀態轉移來準確地驗證多執行緒程式。

圖 2-2：SPIN 向 PANCAM 取得狀態的示意圖

在此架構下，SPIN 本身負責：

(1) 處理狀態的搜尋

(2) 決定接下來哪一個執行緒要被執行

(3) 儲存曾經拜訪過的狀態

而 PANCAM 則負責

(1) 計算狀態的轉移

(2) 執行在執行緒中使用的指令

(3) 檢查所設定的判斷式（predicate）

2.2.1 PANCAM 使用方法

我們以 Peterson 演算法為例，下面範例(5)是 C 的原始碼

範例(5)
```
static volatile int sh_f0, sh_f1, sh_last;
        void * run_thread0 () {
            struct pa_desc d;
            d.f0 = &sh_f0;
            d.f1 = &sh_f1;
            d.last = 1;
            for (;;) {
                *(d.f0)=1;
                sh_last=d.last ;
                while (*(d.f1)==1 &&(sh_last==d.last))
                {
                    ; /* busy wait */
                }
                /* critical section */
                d.f0=0;
             }
        }
```

範例(6)
```
init() {
    c_code {
```

```
        pan_setup() ;
        pan_invariant("check exclusion") ;
        pan_run_function("init") ;
        pan_start_thread(0, "run_thread0", NULL) ;
        pan_start_thread(1, "run_thread1", NULL) ;
    };
    run thread0() ;
    run thread1()
}
proctype thread0() {
    do
    :: c_expr{pan_enabled(0)}
    -> c_code{pan_step(0);}
    od
}
proctype thread1() {
    do
    :: c_expr{pan_enabled(1)}
    -> c_code{pan_step(1);}
    od
}
```

範例(6)的前三行程式碼 **pan_setup() ;pan_invariant("check exclusion") ; pan_run_function("init") ;**是在初始化系統的狀態，而 **c_expr {pan_enabled(0)} -> c_code {pan_step(0);}**則是說如果 0 號的執行緒可以執行（pan_enabled(0)為真）則讓執行緒 0 執行，轉移下一個狀態(pan_step(0))，並回傳新的狀態給 SPIN。如此一來 SPIN 則可以透過 PANCAM 計算出所有可能的狀態。

2.2.2 使用者定義抽象化

使用者定義抽象化（User Defined Abstractions）[Joshi and Zaks 2008]，是 PANCAM 提供檢測人員在 PROMELA 的原始檔中定義那些變數在驗證時重要或不重要；那些不重要的變數在驗證時不會作為狀態比對的依據，以減少狀態數目，加速檢測時間[Holzmann and Joshi 2004]。而下面範例(7)則是使用者定義抽象化的範例：

範例(7) /* **data hiding** */

```
c_track   "&x"   "sizeof(int)"   "UnMatched";
c_track   "&y"   "sizeof(int)"   "UnMatched";
/* abstraction: */
c_track   "&sumxy"   "sizeof(unsigned char)"   "Matched";
```

c_track、UnMatched 和 Matched 是 SPIN 在 4.1 版就定義的關鍵字，而在 c_track 後第一個字

串是要處理的變數之記憶體起始位置，第二個字串是要處理的記憶體大小，第三個字串是要處理的方法，分為比對（Matched）或是不比對（UnMatched）。

2.2.3 可搶先次數上限檢查

根據文獻[Musuvathi and Qadeer 2007]指出，如果程式有任何錯誤，只需在極少的上下文切換（context-switch）次數下就可以找到，而 PANCAM 提供一個函式 pan_enabled_cb(thread id)來限制系統執行緒可搶先（preemptive）的上限次數，以達成可搶先次數上限檢查（Context-Bounded Checking），圖 2-3 是使用可搶先次數上限檢查的效果圖。

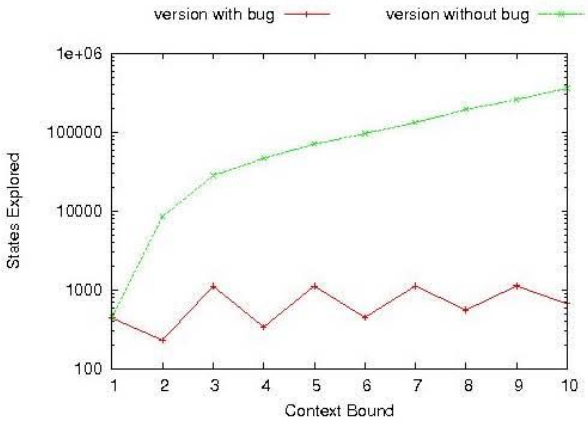


圖 2-3：可搶先次數上限檢查效能圖[Joshi and Zaks 2008]

2.3 目前限制與缺點

(1) PANCAM 與 SPIN 對於本計畫而言只能驗證系統的邏輯正確性，並無法驗證系統是否能在有限時間達成任務。
(2) 根據 PANCAM 作者所提供的資料，目前他們目前只驗證了 2800 行以下的程式。

3 驗證即時系統的 WCET 工具

3.1 WCET 簡介

一個即時系統須執行許多任務來滿足功能上的需求，每一任務通常根據不同輸入資料或環

境而有不同的執行時間。Worst-Case Execution Time（縮寫為 WCET）中文譯為「最差情況執行時間」，代表在目標處理器上執行指定程式所需的最長時間。下圖的曲線代表全部執行時間的分布，其中最短的執行時間我們稱為最佳情況執行時間（best-case execution time），而最長的執行時間即為最差情況執行時間。WCET 工具泛指可用以估算最差情況執行時間的工具。

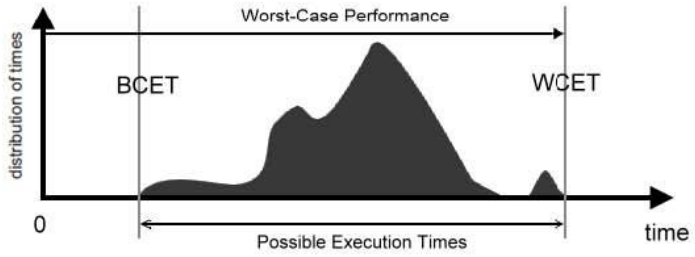


圖 3-1：WCET 示意圖

3.2 應用實績

硬性即時（hard real-time）系統需要滿足嚴格的時間限制，因此我們需要計算執行時間的上限來證明系統是否滿足所要求的時間限制，WCET 工具便可應用於此類系統中。

目前 WCET 工具被使用於檢驗攸關安全的系統，包含航空、汽車等工業，例如：空中巴士 A380 的電子飛行控制（fly-by-wire）系統。安全關鍵軟體的國際航空標準（RTCA/DO-178B），視 WCET 工具為等級 A（*註 1）程式碼的驗證工具之一[Wilhelm et al. 2008]。

許多文獻亦針對以 WCET 工具分析程式碼的實例加以探討，例如於太空上的應用[Holsti et al. 2000][Rodriguez et al. 2003]、航空工業[Thesing et al. 2003][Ferdinand et al. 2001]等。

*註 1：DO/178B 定義評估保證等級（Evaluation Assurance Level，簡稱 EAL）來判定軟體之安全等級，共分 A-E 五級，A 為最高等級。

3.3 WCET 工具介紹

3.3.1 aiT

aiT 為德國 AbsInt Angewandte Informatik 公司所開發的工具，用於從程式執行檔獲得執行時間的上限。所需的輸入包括：欲分析的程式執行檔、欲分析的任務開始位址、微處理器模型的記憶體、匯流排描述（列舉記憶體區最少及最多存取次數）及時脈。

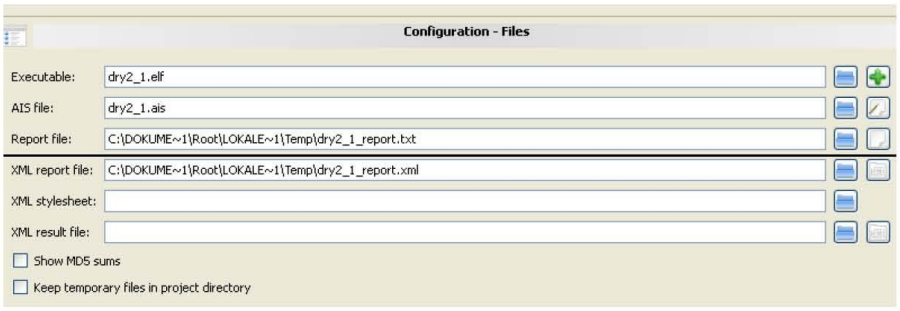


圖 3-2：aiT 實際輸入檔案介面 [aiT website]

aiT 允許使用者加上註解（annotation）來幫助工具增加計算結果的準確性，例如間接函式呼叫、迴圈上限、遞迴深度等。當資訊無法自動偵測時，便需要註解（例如動態的程式性質）提供 [aiT website]。註解可寫於原始碼中，以執行時的行資訊來對應二元位址。亦可寫於附加的參數檔案，藉絕對位址指向程式位置。另外註解可指定暫存器（register）和變數的值，有助於以變數值區別模組的分析[Wilhelm et al. 2008]。

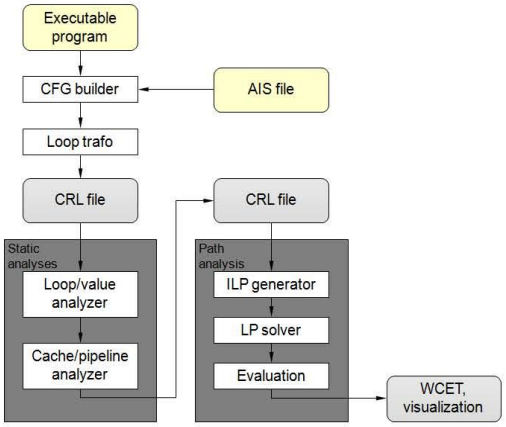


圖 3-3：aiT 結構示意圖，黃底方格表示所需輸入的檔案

以下以 WCET 工具所必須獲得的迴圈上限（loop bound）資訊作為撰寫註解的範例。aiT 利用迴圈上限分析來決定每個迴圈的執行次數上限，當無法以此自動化方法求出時，便需要使用者的註解來提供資訊。

迴圈上限註解寫於參數檔（AIS file）時，格式為「LOOP *ProgramPoint Qualifier* Max j;」。

ProgramPoint 為位址或"R" + n loop 的型式（代表在例程（routine） R 中從 1 開始數的第 n 個迴圈）。*Qualifier* 為可選填項目，可為 begin（表示為 C 語言的 while 迴圈）、end（表示為 C 語言的 do-while 迴圈）。實際註解如：「loop "_prime" + 1 loop end max 10;」，表示在例程_prime 的第一個迴圈結束點開始迴圈測試，且限制迴圈執行不超過 10 次。

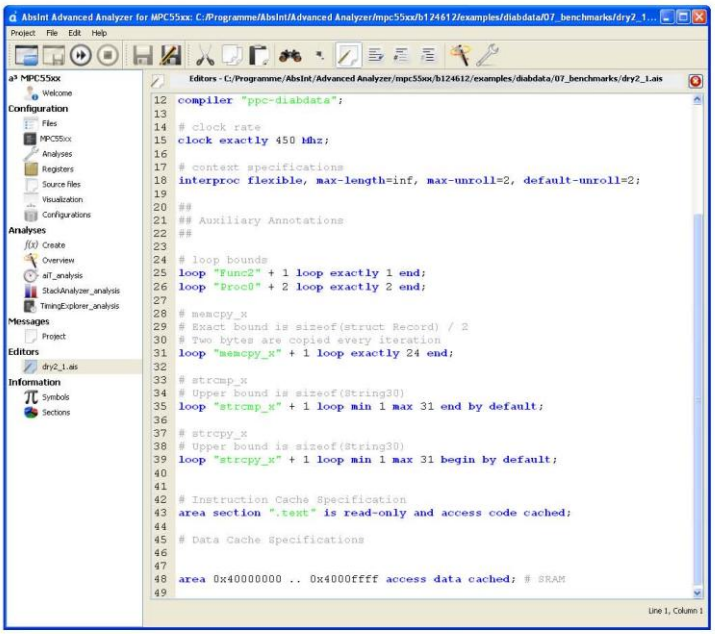


圖 3-4：aiT 實際註解參數 AIS 檔 [aiT website]

註解寫於原始碼時，格式為「/* ai: specification$_1$; ... specification$_n$; */」，以註解前加 ai:表示為特殊註解。同時也可於註解中指定註解對象的位址，或以 here 表示此註解所指的位址就為註解的撰寫點。實際註解範例如下：

```
for (i=3; i*i <= n; i += 2) {
    /* ai: loop here end max 10; */
}
```

上例表示此迴圈從結束點開始迴圈測試，且限制迴圈執行不超過 10 次。

其他可註解項目包含程式的呼叫及分支（branches），格式如下：

「INSTRUCTION ProgramPoint CALLS Target$_1$, ..., Target$_n$;

INSTRUCTION ProgramPoint BRANCHES TO Target$_1$, ..., Target$_n$;」

或是限制遞迴次數，註解格式為「RECURSION "name" Bounds ;」。甚至可以設定特定基本區段（basic blocks）的執行次數，例如「flow 0x100 is max 4;」表示從位址 0x100 開始的基本區

段最多執行 4 次。亦可設定時脈，例如「clock 10200 kHz ;」。

除了上述註解以外，aiT 還可用註解指定暫存器（register）和變數的值，有助於以變數值區別模組的分析。另外可指定特定記憶體區間為只能讀取(read-only)或只能寫入(write-only)，或指定特定基本區段永不執行，或是特定條件永遠為真或否[Heckmann and Ferdinand 2004]。

aiT 輸出為目標任務的執行上限時間，並提供最差路徑等圖形化結果呈現。

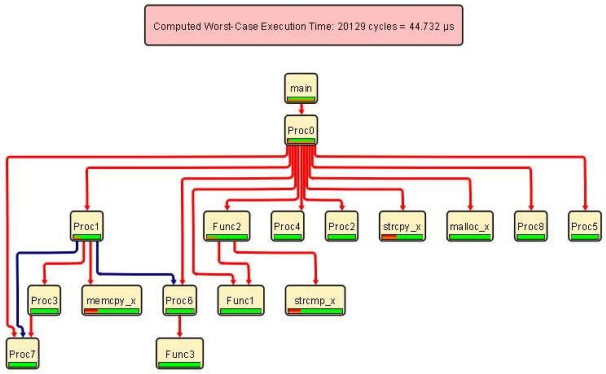


圖 3-5：aiT 實際的輸出結果，包括所計算出的最差情況執行時間及最差路徑的圖形化表示，紅色箭號為 WCET 的呼叫路徑[aiT website]

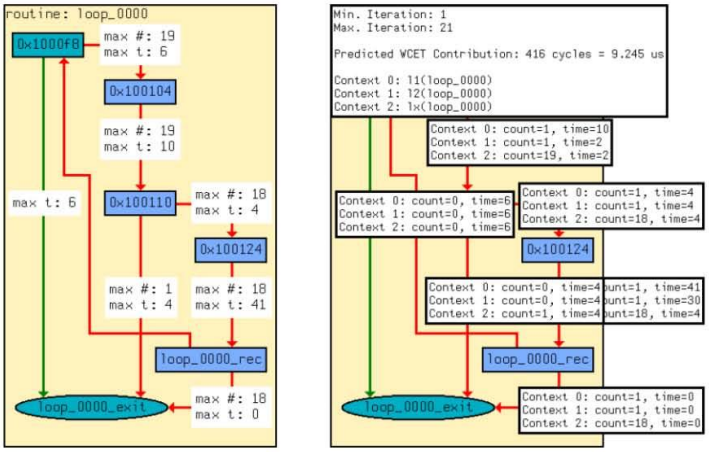


圖 3-6：aiT 實際的輸出結果，顯示一迴圈中含時間資訊的基本區段圖，左圖中#表最差狀況下走此路徑的次數，t 表最差狀況下走此路徑的時間。右圖顯示根據變數值的不同，得到不同的執行時間[aiT website]

aiT 目前支援的處理器及編譯器如下：

Processor	Compiler
ARM7	TI (Texas Instruments) ARM (ARM)
C16x/ST10	TASKING (Altium)
HCS12/STAR12, HCS12X, HCS12XE	CodeWarrior (Metrowerks/Freescale) Cosmic (Cosmic)
HC11	Cosmic (Cosmic)
LEON2, LEON3	GCC
M68020	XD Ada (EDS)
PowerPC 5xx, 603e, 55xx	DiabData (WindRiver) GHS C/C++ (Green Hills) GHS Ada (Green Hills) GCC
TMS320C3x	TI (Texas Instruments)
TriCore 1766, 1767, 1796, 1797	TASKING (Altium) HighTec (HighTec)
NEC V850E	GHS (Green Hills)

圖 3-7：aiT 所支援的處理器及編譯器[aiT website]

aiT 工具的使用有其限制，包含目標程式須為循序執行（sequential）的程式，也就是不能有執行緒(thread)、平行運算(parallelism)、外部事件等。aiT 假設沒有外部干擾，中斷(interrupt)或處理器未在預期時間內回應等問題須分開考慮；亦不支援動態資料結構，alloc 函式家族不可出現在欲分析的程式碼中；並需以符合 ANSI C 標準的 C 語言撰寫[aiT website]。

aiT 目前實際運用於空中巴士 A380 的飛行控制系統驗證[Souyris et al. 2008]，也在 Volvo 汽車嵌入式系統中顯示所計算出的 WCET 值較傳統方法精準，且能找出測量方式所忽略的情況[Sehlberg 2005]。

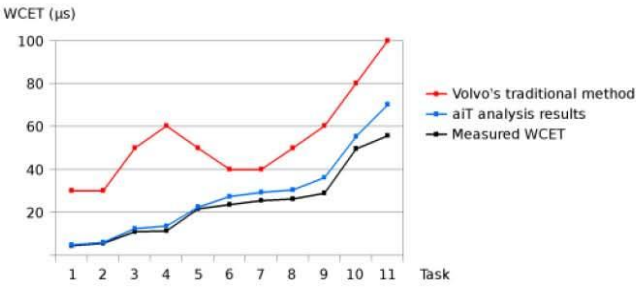


圖 3-8：Volvo 汽車嵌入式系統中使用 aiT 計算 WCET 成效圖[aiT website]

3.3.2 Bound-T

Bound-T 原本是為了芬蘭太空系統而開發的工具，目的用於驗證太空船上所用的軟體，後由 Tidorum 公司開始應用於其他領域。

Bound-T 的輸入為執行檔，通常含有除錯資訊的嵌入式符號表（embedded symbol table）。輸出包含文字檔與圖片檔，其中文字檔含有執行時間上限等資訊，而圖片檔包括函數調用關係圖（call graphs）與控制流程圖（control-flow graph）。

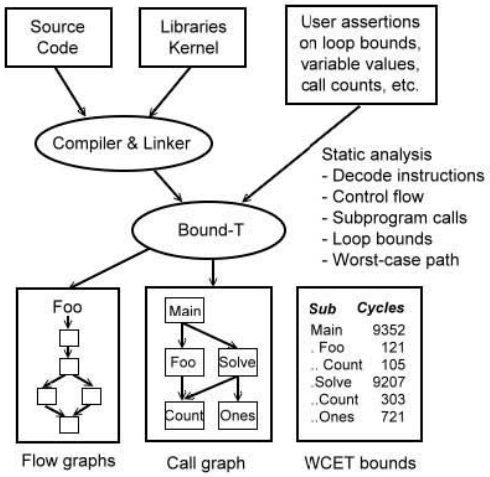


圖 3-9：Bound-T 輸入輸出示意圖

註解需寫於分開的檔案，無法寫於原始碼中。註解所指的對象可為符號名稱（symbolic name），如子程序、變數等（例如可指定變數值），或為結構性質（structural properties），如迴圈、呼叫等。工具將自動計算可計數（counter-based）迴圈的上限，其他迴圈則需使用者加以註解來幫助工具計算[Wilhelm et al. 2008]。

註解的格式為「上下文(context) + 事實（fact）」，上下文可為單一指令、呼叫、迴圈、子程式、全部程式（global context），事實可為變數值域、迴圈執行次數、呼叫執行次數、子程式的 WCET、動態呼叫（例如間接呼叫等）所可能的被呼叫者等。以下為第一個範例：

```
variable "factor" >= 20;
```

此註解表示全域變數 factor 值須大於等於 20。第二個範例如下：

```
for I in 1 .. 7 loop
    Foo (I);
    if I mod 2 = 0 then
        for J in 1 .. 5 loop
            Bar (I, J);
```

```
            end loop;
        end if;
    end loop;
```

從上述程式碼可得知第一層迴圈從初始狀態開始執行只執行 1 次(迴圈內容從 I 為 1 開始到 7，共執行 7 次，整個 for 迴圈只初始一次)，第二層 for 迴圈從初始狀態開始執行 3 次(當 I 為 2、4、6 時)，若我們假設第二層迴圈是在名為 Nested 的子程式中，註解可寫為：

```
subprogram "Nested"
    loop that is in loop starts 3 times; end loop;
end "Nested";
```

上述註解表示限制名為 Nested 的子程式，迴圈執行次數為 3 次。這樣的註解可幫助 Bound-T 更精準地計算各迴圈的執行次數，因為 Bound-T 無法自動計算第二層迴圈的開始次數，而會假設第一層迴圈的執行次數便是第二層迴圈的開始次數，如此會過度計算 WCET [Bound-T website]。

Bound-T 有其工具限制，如所分析的程式不可為遞迴，需為可化簡（reducible）的控制流程圖。此工具在迴圈上限分析時，可能找出不可到達的路徑為其邊際效應(side-effect)[Wilhelm et al. 2008]。

Bound-T 支援的處理器如下：Analog Devices ADSP21020、ARM7 TDMI、Atmel AVR（8-bit）、Intel 8051 （MCS-51®） series、Renesas H8/300, SPARC V7 / ERC32 [Bound-T website]。

3.3.3 RapiTime

RapiTime 為 pWCET （Probabilistic Worst Case Execution Time）工具的商業品質版本，此工具以測量為基礎(measurement-based)，亦即其藉由測量得知一段程式碼(通常為基本區段)的時間資訊，將測量的結果與程式結構結合，加以預測程式的最長路徑。當程式為非循序執行等情況時，採行此方法之工具，可有較好的 WCET 計算效率。

此工具不只計算出單一 WCET 值，亦提供最長路徑執行時間的機率分布，其上界便為 WCET 上界。RapiTime 目標為進階處理器上的中至大型嵌入式系統，包含汽車電子、航空電子設備和通訊產業。

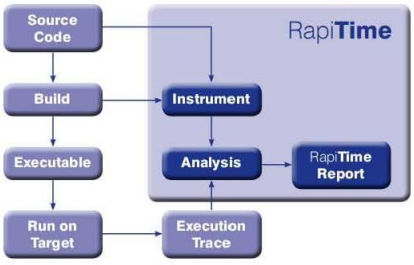


圖 3-10：RapiTime 結構示意圖[RapiTime website]

此工具的輸入可為程式原始碼（C 或 Ada）的集合或是執行檔，使用者須提供測試資料以進行測量。輸出為 HTML 的報告，內含每個函式與子函式的 WCET 預測以及實際測量的執行時間。

使用者可在程式中加註解，來引導測試設備和分析流程的運行、限制迴圈次數上限等[Wilhelm et al. 2008]。

RapiTime 可在許多即時作業系統（real-time operating systems）上使用，但需要即時作業系統的整合，將特殊事件加入在追溯（trace）中，使之有能力處理時戳環境切換（timestamp context switch）和中斷。整合即時作業系統可有以下三種方法：

(1) 可取得作業系統的原始碼時，將手動的檢測點加至環境切換和中斷例程中，這些檢測點出現在追溯可使 RapiTime 分析出環境切換所需花費的時間。

(2) 若可取得硬體支援（Nexus/ETM 等），則可用來監測中斷。

(3) 依系統的目標和追溯機制，可透過替換機制來加以監測。例如某些目標可以在中斷發生時，暫停內部計時器[RapiTime website]。

RapiTime 不仰賴處理器的模型，所以原則上可以模擬任何處理單元，其限制在於需要額外的執行追溯，而這需要程式的測試和機制以從目標系統中得到追溯。另外 RapiTime 不能分析遞迴和無法靜態分析的函式指標。

RapiTime 支援的處理器如下：Motorola processors（包括 MPC555、HCS12 等）、ARM、MIPS、NecV850[Wilhelm et al. 2008]。

3.3.4 其他工具 [Wilhelm et al. 2008]

除了上述的三個主流 WCET 工具之外，亦有許多不同應用或採用不同方法的 WCET 工具，以下是簡短的介紹。

佛羅里達州立大學（Florida State University）、北卡羅萊納州立大學（North Carolina State University）、弗曼大學（Furman University） 所發表的研究雛形，主要的應用領域在硬性即時系統和有時間限制的嵌入式能源感知（energy-aware）系統。使用者以修改過的編譯器將所要分析的任務檔案編譯，此編譯器可提供工具所需的資訊，包含迴圈執行次數、流程控制等，而工具可對每個函式和迴圈計算上下限，藉此算出程式的 WCET。此工具的限制在於必須可以靜態分析得知迴圈上限，亦不支援遞迴、指標分析與動態記憶體配置。

TU Vienna 即時系統團隊試驗了不同方法研發出幾種工具雛形，第一種工具雛形基於靜態時間分析，已被整合進 Matlab/Simulink 工具中，其分析的程式需以 wcetC 撰寫；而 wcetC 屬於 C 語言的一種，可讓使用者或資訊流分析工具對可達的執行路徑作註解。第二種工具以測量為基礎，運用遺傳理論直接產生輸入資料，用此資料測量最差狀況執行時間。第三種工具結合測量和靜態分析下得到的結果，來計算 C 程式碼的最差狀況執行時間，此工具最主要用來分析自動產生的程式碼（例如 Matlab/Simulink 模組所產生的程式碼）。

SWEET （SWEdish Execution Time tool），此工具目前移至 Mälardalen 大學發展，重點在於資訊流分析（flow analysis）。SWEET 以模組方式開發，結合不同分析方法及工具，將 WCET 分析分成三部分：資訊流分析、處理器行為分析（processor-behavior analysis）及預測

計算（estimate calculation），彼此以兩種資料結構進行溝通：含有資訊的作用域圖（scope graph）、時間量測模型（timing model）。

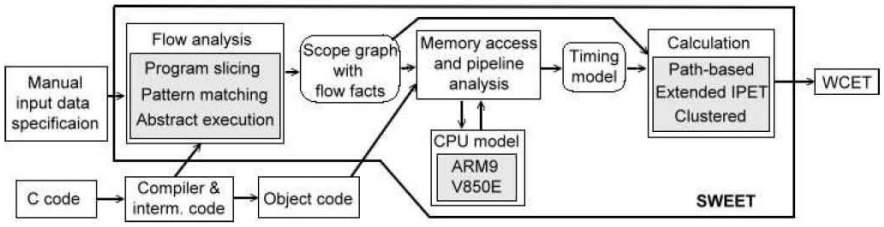


圖 3-11：SWEET 工具結構示意圖

SWEET 的資訊流分析可以處理包含指標、未結構化程式碼、遞迴等 ANSI C 程式，程式須以 SWEET 整合過的研究編譯器編譯，才可自動化資訊流分析，否則需使用者提供資訊，另外此研究編譯器目前不支援動態記憶體配置。

SymTA/P 工具可得到 C 程式在微控制器上的執行時間上下限，為 SYMbolic Timing Analysis for Processes 的縮寫。此工具用實際的目標系統，合併在原始碼層級的平台獨立路徑分析（platform-independent path analysis），以及在目標碼（object code）層級的平台相關測量方法。此種整合分析工具的優點在能簡易地重設目標於新硬體平台，執行時間可從現有的週期精確（cycle-accurate）處理器的模擬器或評估板（evaluation board）得到。

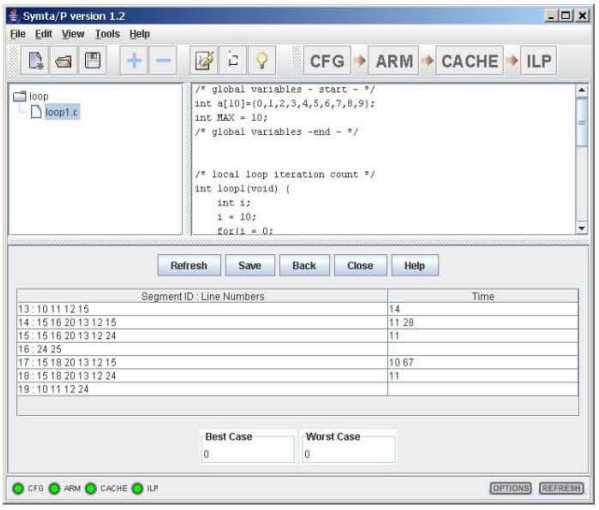


圖 3-12：SymTA/P 實際輸出結果圖[Staschulat 2005]

為了彌補管道效應（pipelining effects），SymTA/P 工具個別測量每個基本區段所增加的延遲時間，因此可能高估 WCET。另外此工具假設輸入資料涵蓋最差狀況，而不考慮指令有資料相關時的執行時間。SymTA/P 工具的測量環境將決定最終分析的精準度。

Chronos 為新加坡國立大學所發展的研究雛型，為開放原始碼的靜態 WCET 分析工具，在有複雜微架構處理器上，可判斷任務執行時間的緊密上界。其輸入為以 C 語言撰寫的程式和目標處理器的組態（configuration），有時需要使用者註解來幫助計算迴圈執行次數和增加準確性。

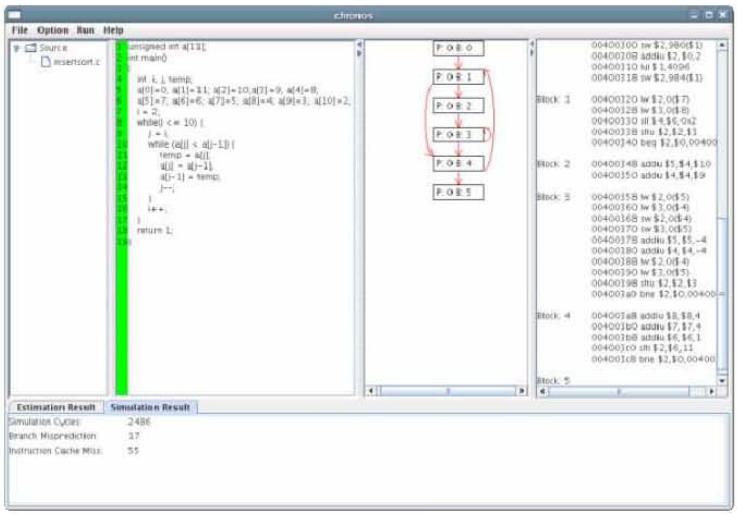


圖 3-13：Chronos 實際輸出結果圖[Chronos website]

Chronos 對程式執行檔作分析，將其拆解以產生控制流程圖，並在控制流程圖上運行處理器行為分析，此工具支援亂序管道（pipeline）、動態分支預測、指令快取，以不同功能間的互動來計算緊密的執行時間上限。目前的限制為無法分析資料快取，因為工具重點為處理器行為的分析，只能作有限的資訊流分析來計算迴圈上限，另外此工具需要使用者的回饋來得知不可能被執行的程式路徑。

4 結語

多執行緒程式邏輯正確性的自動化驗證有相當的難度，目前學界仍持續地努力尋求更好、更有效率的方法與工具。加上即時性的要求後，驗證的難度更高，勢必要結合不同的方法與工具方能得到更完整的分析驗證結果。本計畫下一階段的主要工作之一便是要探討如何做這樣的結合；此外，現有分析驗證工具的蒐集與整理仍有未盡完善之處，這也是我們希望

持續努力的地方。

造成分析高階多執行緒程式困難的主因在於，以靜態分析為主要方法的執行時間估算工具目前尚無法掌握特定即時作業系統的排程行為；而以實際量測為手段的工具則有先天的低估執行時間的可能性。為即時作業系統排程（包括 context-switch）等行為建模，以做為時間估算之依據是根本解決之道，我們將密切注意這方面的研究發展。

5 參考文獻

[aiT website] http://www.absint.com/ait/

[Bound-T website] http://www.bound-t.com/

[Chronos website] http://www.comp.nus.edu.sg/~rpembed/chronos/

[Ferdinand et al. 2001] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT '01 Proceedings of the First International Workshop on Embedded Software*, pages 469-485, 2001.

[Heckmann and Ferdinand 2004] R. Heckmann, C. Ferdinand. Worst-case execution time prediction by static program analysis. In *Proceedings of 18th International Symposium on Parallel and Distributed Processing*, pages 125–134, 2004.

[Holsti et al. 2000] N. Holsti, T. Långbacka and S. Saarinen. Worst-case execution-time analysis for digital signal processors. In *Proc. EUSIPCO 2000 Conf. (X European Signal Processing Conf.)*, 2000.

[Holsti et al. 2000] N. Holsti, T. Långbacka, and S. Saarinen. Using a worst-case execution-time tool for real-time verification of the DEBIE software. In *Proc. DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, pages 307-312, 2000.

[Holzmann 2004] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

[Holzmann and Joshi 2004] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In *11th SPIN Workshop on Model Checking of Software*, *LNCS 2989*, pages 76–91, 2004.

[Joshi and Zaks 2008] Rajeev Joshi and Anna Zaks. Verifying multi-threaded C programs with SPIN. In *15th SPIN Workshop on Model Checking of Software*, *LNCS 5156*, pages 325-342, 2008.

[Musuvathi and Qadeer 2007] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 34th ACM Symposium on Programming Languages (POPL)*, pages 446–455, 2007.

[RapiTime website] http://www.rapitasystems.com/

[Rodriguez et al. 2003] M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in calculating the WCET of a complex on-board satellite application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, pages 3-6, 2003.

[SPIN website] http://spinroot.com/

[Sandell et al. 2004] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods*, pages 146-160, 2004.

[Sehlberg 2005] D. Sehlberg. *Static WCET analysis of task-oriented code for construction vehicles*. Master's thesis, Mälardalen University, Västerås, Sweden, 2005.

[Souyris et al. 2008] J. Souyris, E. Le Pavec, G. Himbert, G. Borios, V. Jégu, R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis,* pages 21-24, 2007.

[Staschulat 2005] J. Staschulat. SYMTA/P – performance verification for complex embedded systems. *http://sourceforge.net/projects/symtap/*, 2005.

[Thesing et al. 2003] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *Proc. 2003 Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 625-632, 2003.

[Wilhelm et al. 2008] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem — Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pages 1-53, 2008.

附件二：正規方法應用於安全軟體驗證之理論與技術

核能儀控系統電腦化發展之安全功能認證研究

On Certification of Safety Functions in Computerized Nuclear Instrumentation and Control Systems Development

第二次期中報告：正規方法應用於安全軟體驗證之理論與技術

計畫編號：1002001INER027
計畫參與人員：蔡益坤、蔡明憲、林靖婕、劉啟祥
所屬單位：國立台灣大學

摘要

我們就並行即時安全軟體邏輯正確性驗證及執行時間分析之相關正規方法作一整理與介紹。在邏輯正確性驗證部分，我們介紹利用程式註記輔以定理證明與決定程序以及利用時序邏輯模型檢查來驗證多執行緒程式的兩種方法。在執行時間分析部分，我們介紹如何以靜態分析以及動態執行與量測兩種方法估算單一工作或執行緒「最差情況執行時間」；接著我們介紹，在單一執行緒執行時間已知的情形下，如何利用排程理論分析即時多執行緒程式的執行時間。

關鍵詞：並行性、嵌入式系統、正規方法、霍爾邏輯、模型檢查、即時系統、安全攸關系統、靜態分析、時間量測分析、最差情況執行時間

Abstract

We review the theory and technology of formal methods as applied in verifying the logical and timing correctness of concurrent real-time safety-critical software. For the verification of logical correctness, we review deductive methods based on program annotation and assisted by theorem provers and decision procedures and temporal-logic model checking methods for multi-threaded programs. For timing analysis, we review methods based on static analysis and dynamic execution and measurement for estimating the worst-case execution time of a single task or thread. We then review how, given the execution times of individual threads, scheduling theory may be applied in analyzing the running time of a multi-threaded program.

Keywords: Concurrency, Embedded Systems, Formal Methods, Hoare Logic, Model Checking, Real-Time Systems, Safety-Critical Systems, Static Analysis, Timing Analysis, WCET

中華民國 100 年 7 月 6 日

目　錄

附圖目錄

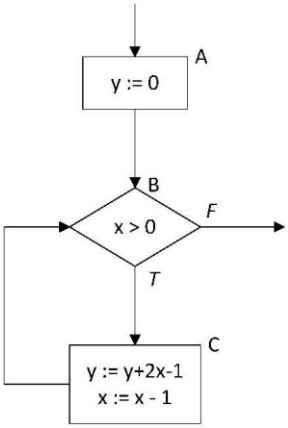
1 背景與目的

在第一次期中報告中，我們蒐集整理了一些可應用於安全攸關（safety-critical）即時嵌入式系統（real-time embedded systems）之邏輯正確性驗證及執行時間分析的工具。這些工具在原理上都基於正規程式分析與驗證方法，包括靜態分析（static analysis）及模型檢查（model checking）等。在這第二次期中報告中，我們就相關正規方法理論與技術作一有系統之整理與介紹。

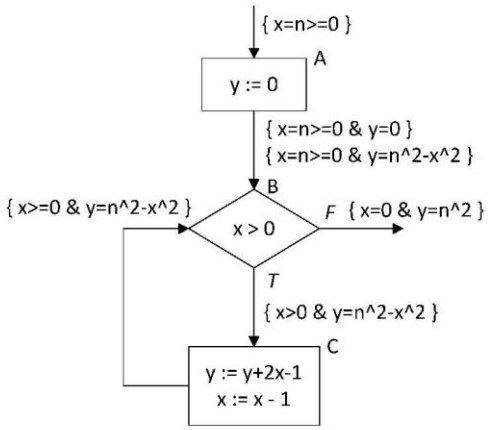
在程式邏輯正確性驗證部分，我們分別介紹利用程式註記（program annotation）輔以定理證明（theorem proving）與決定程序（decision procedures）以及利用時序邏輯（temporal logic）模型檢查來驗證多執行緒程式的兩種方法。在執行時間分析部分，我們介紹如何以靜態分析以及動態執行與量測兩種不同方法估算單一工作或執行緒「最差情況執行時間」（Worst-Case Execution Time）；接著我們介紹，在單一執行緒執行時間已知的情形下，如何利用排程（scheduling）理論分析即時多執行緒程式的執行時間。我們也同時說明邏輯及執行時間兩種正確性彼此之間的關聯。

2 基本概念

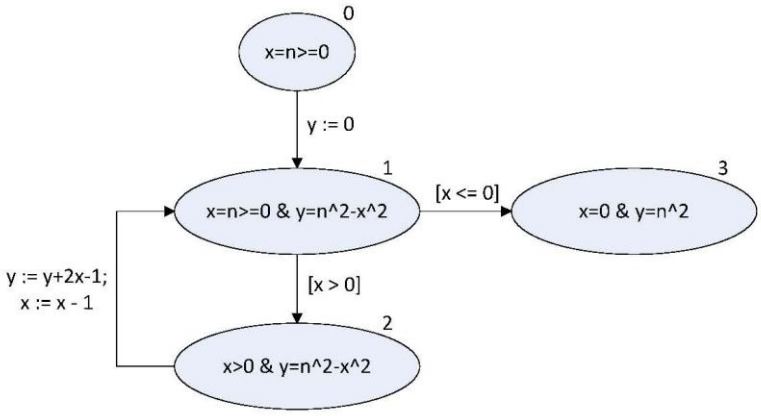
正規程式分析與驗證的目的在於精準且廣義地推斷程式執行時的動態行為，所謂廣義是指這樣的推斷能適用於很多不同的情境，特別是不同的初始化情形。其中最基本的觀念是程式在任兩個相鄰指令或判斷式之間有明確的狀態。以底下代表某一程式的流程圖為例，在 A 與 B 之間變數 y 的值為 0。雖然我們無從判斷在這時 x 的值為何，但 x 必等於其在 y := 0 執行前的值。



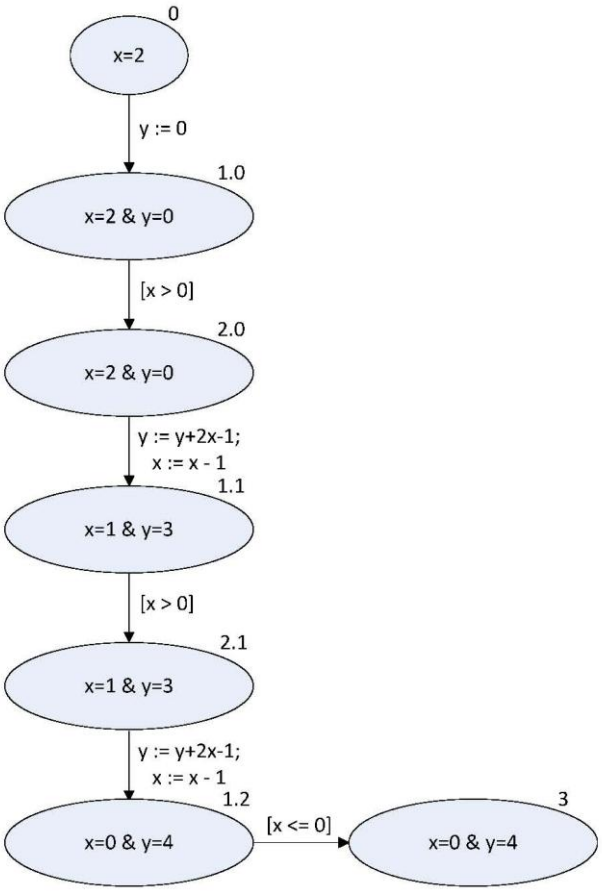
程式的狀態隨指令的執行而不斷改變，但程式變數之間通常存在著極簡單且不變的關係，讓我們可以推斷程式完成的工作究竟為何（畢竟正常的程式不會漫無目的計算）。以上的程式究竟在計算什麼呢？我們可以透過對程式狀態的註記來「彰顯」程式的目的，如下圖所示。這些註記透露一件重要的事：如果程式開始執行時，x 的值為一大於或等於零的整數 n，則程式結束（跳離迴圈）時，y 的值為 n 的平方。當然，註記本身要正確，以上的推斷才有意義；這通常並不難判斷。我們只要針對每一個指令，觀察程式在滿足該指令前方之註記的任何狀態下，執行該指令完畢後的狀態，是否滿足該指令後方之註記的條件。



我們也可以把以上對於程式狀態的理解和自動機理論關聯在一起。以橢圓圈來表示程式的狀態、箭號表示判斷式或指令，我們可以把註記完整的流程圖，想像成一個抽象的有限狀態機，如下圖；之所以稱為抽象，是因為一個橢圓圈其實很可能同時代表很多不同的程式狀態。



這個抽象的有限狀態機在 x 有特定初始值時（如 x=2），便成為名符其實的有限狀態機，如下圖。這項程式與自動機的關聯性，讓我們可以運用自動機的理論來驗證程式的正確性。


0
x=2
y := 0
1.0
x=2 & y=0
[x > 0]
2.0
x=2 & y=0
y := y+2x-1;
x := x - 1
1.1
x=1 & y=3
[x > 0]
2.1
x=1 & y=3
y := y+2x-1;
x := x - 1
1.2
x=0 & y=4
[x <= 0]
3
x=0 & y=4

7

3 驗證並行程式邏輯正確性之架構與方法

在並行程式（concurrent program）中，若能預先確知執行緒（thread）或程序（process）之個數，而每一執行緒或程序又均為有限狀態或可抽象化為有限狀態時，並行程式的驗證是有可能全自動化的。

以下在章節3.1中介紹根據模型檢查（model checking）理論為基礎的驗證架構與方法，在章節3.2中介紹以演繹式程式驗證（deductive program verification）理論為基礎的驗證架構與方法。

3.1 模型檢查

模型檢查是一種常見用於硬體驗證和軟體驗證的方法。所謂模型檢查的進行，簡要來說可分為兩個階段：第一個階段是將系統所有可能的系統狀態（state）及狀態轉移圖建構出來，這就是所謂的模型（model）。第二個階段則是以時序邏輯（temporal logic）的邏輯表示式（formula）表述系統的性質並運用其相關演算法來自動檢查此模型是否滿足該表示式 [Clarke, Emerson and Sistla 1986]。模型檢查與時序邏輯的理論很多部分都是由 E.M. Clarke 與 E.A. Emerson 所提出的，而這兩人也是 2007 年圖靈獎（Turing Award）的得主。

在章節3.1.1將會先討論時序邏輯與ω自動機，之後再於章節3.1.2討論詳細的架構與方法，並於章節3.1.3討論基於pthread之並行程式驗證。

3.1.1 時序邏輯與ω自動機

根據計算理論，程式的運行可以視為一種狀態的轉移（改變），一連串的狀態可以稱為計算（computing） [Clarke, Grumberg and Peled 1999]。一個並行程式的狀態為各個變數當前的值，以及各個執行緒當前所在的程式碼行數，因此一個並行程式的語意可以視為一個包含該程式各種可能計算的集合。而時序邏輯中的線性時序邏輯（linear temporal logic）和 ω 自動機（ω-automata）都可用來描述某個計算集合，是以可用較能為人所理解的線性時序邏輯作為系統規格（specifications）的表述，並透過 ω 自動機來辨認符合或違反規格的計算。

3.1.1.1 線性時序邏輯

表 3-1列出常見的線性時序邏輯表示式，其中p和q是描述系統狀態的述語（predicate），例如當系統變數有x與y時，述語可為「x = 100」、「x + y > 10」、「x > 0 且y < 0」等。

表示式	語意
○p	下一步 p 要成立
□p	p 永遠成立
◇p	p 之後終究會成立
p U q	q 終究會成立，在 q 成立前，p 會一直成立
□(p→◇q)	永遠 p 成立後 q 終究會成立
□◇p	p 會無窮地成立
◇□p	p 終究會一直成立

表 3-1、常見的線性時序邏輯表示式

舉例來說，在互斥存取（mutual exclusion）的演算法規格中，可能有兩個性質必須被滿足：

1. 一個臨界區域（critical section）同時間不可以有兩個以上的執行緒進入。
2. 如果有一個執行緒一直想要進去某臨界區域存取共享資料（share data），則終究可以一直進去。

假設系統只有兩個執行緒，則性質 1 可以表示為$\square(\sim(crit_1 \wedge crit_2))$，其中 $crit_i$ 是表示執行緒 i 在臨界區域裡面的一個述語，~表示否定，∧運算子用來表示左右兩個述語必須同時滿足。而性質 2 可以用$(\square\diamond enter_1 \rightarrow \square\diamond crit_1) \wedge (\square\diamond enter_2 \rightarrow \square\diamond crit_2)$來表示，其中 $enter_i$ 為表述執行緒 i 想要進去臨界區域這件事情。

檢查一個無窮（infinite）的計算是否滿足一個線性時序邏輯表示式的方式如下：將該線性時序邏輯表示式轉換成等價的ω自動機，以該無窮的計算作為輸入字串給予ω自動機，自動機則會回答滿足或不滿足（見章節3.1.1.2）。如果此計算滿足此自動機，則此計算也就符合我們所要確認的規格；同樣地，也可以產生時序邏輯的反命題來確認此計算是否違背所訂定的規格。

3.1.1.2 ω自動機

ω 自動機有許多種類，以下將以 Büchi 自動機為例。一個 Büchi 自動機定義為 $A = (Q, \Sigma, \delta, q_0, Acc)$，其中：

1. Q 為所有的狀態。
2. Σ 為所有的輸入符號。
3. δ：$Q \times \Sigma \times Q$ 為狀態轉換關係，若(q_i, w_j, q_k)屬於 δ，則表示在狀態 q_i 遇到輸入符號 w_j 時，可以轉換到狀態 q_k。每一個屬於 δ 的(q_i, w_j, q_k)都稱作一個狀態轉換（transition）。
4. q_0 為初始狀態，屬於 Q。

5. Acc 為接受條件，是 Q 的子集合。

一個無窮字串可寫作$\sigma = w_0w_1w_2w_3...$，其中w_i屬於 Σ。一個無窮字串$\sigma = w_0w_1w_2w_3...$在自動機 A 上可以有一個以上的計算 $q_0q_1q_2q_3...$，其中(q_i, w_i, q_{i+1})屬於 δ。一個自動機 A 的計算是可接受的條件為：該計算出現無窮多次的狀態跟 Acc 有交集；而一個字串被自動機 A 接受的條件為：該字串在自動機 A 上至少有一個可接受的計算。所有可以被自動機A接受的字串集合稱作此自動機的語言，以L(A)來表示。

下圖 3-1是與□(p → ◇q)等價的一個Büchi自動機，此自動機的

- ✧ 狀態 $Q = \{q_0, q_1\}$，
- ✧ 輸入符號 $\Sigma = \{p\ q, p\ \sim q, \sim p\ q, \sim p\ \sim q\}$，
- ✧ 狀態轉換關係 $\delta = \{(q_0, p\ q, q_0), (q_0, p\ \sim q, q_1), (q_0, \sim p\ q, q_0), (q_0, \sim p\ \sim q, q_0), (q_1, p\ q, q_0), (q_1, p\ \sim q, q_1), (q_1, \sim p\ q, q_0), (q_1, \sim p\ \sim q, q_1)\}$，
- ✧ 接受條件 $Acc = \{q_0\}$。

無窮字串$\{(p\ \sim q)(\sim p\ \sim q)(\sim p\ q)\}^{\omega}(\{w\}^{\omega}$表示w會無窮地出現)會被此自動機所接受，因為此字串在該自動機上有一個可接受的計算：$\{q_0q_1q_1\}^{\omega}$，其中出現無窮多次的狀態為$\{q_0\}$，與 Acc 之交集不為空集合。然而字串$(p\ \sim q)\{(\sim p\ \sim q)\}^{\omega}$不會被此自動機所接受，因為此字串在該自動機上沒有一個計算可以滿足接受條件。

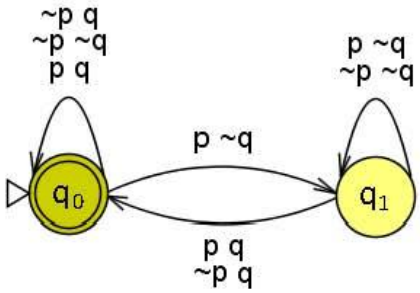


圖 3-1、與□(p→◇q)等價的一個 Büchi 自動機 [Büchi Store]

3.1.2 架構與方法

在章節3.1.1中已知可利用線性時序邏輯來制定規格並透過ω自動機來檢查計算是否滿足規格。當面對程式碼時，須先把程式碼轉換成狀態轉移圖，也就是模型，再根據規格對模型做出全盤的檢查和分析 [Clarke, Grumberg and Peled 1999]，其架構如圖 3-2。

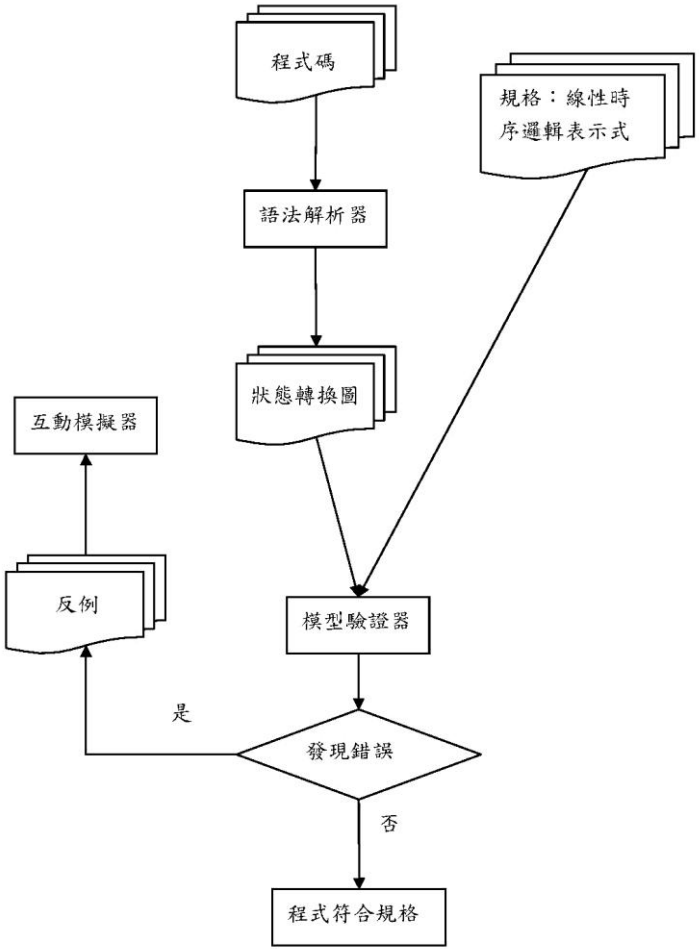


圖 3-2、模型檢查架構圖

一般檢查一個模型 M 是否滿足一個時序邏輯表示式 F 的方法如下：產生與規格 F 等價的 ω 自動機 A_F，以及該自動機的補集 $A_{\sim F}$，再將其補集能辨認的計算集合 $L(A_{\sim F})$與模型 M 能辨認的計算集合 L(M)作交集。如果交集結果不為空集合（$L(M) \cap L(A_{\sim F}) \neq \emptyset$），則該模型有行為違反規格，如圖 3-4 所示；反之，結果若為空集合則表示該模型沒有違反規格，如圖 3-3 所示。

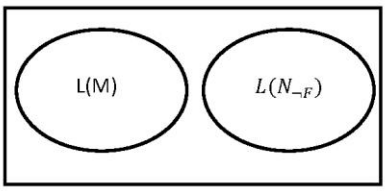


圖 3-4、模型不違反規格

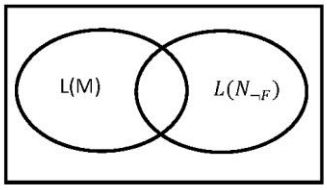


圖 3-3、兩者有交集，模型違反規格

由於實務上往往程式都是非常龐大與複雜，如果一次就把所有的狀態都計算出來，會有狀態爆炸（state explosion）的情況，這樣對於驗證是很沒有效率的。所謂的狀態爆炸是指因為要驗證的程式非常龐大，所以狀態轉移圖有很多狀態，以至於所需的記憶體空間遠大於機器所配置的記憶體容量，導致無法完成模型檢查。為了避免一次產生所有狀態而導致狀態爆炸，可以採取一種漸進式（on-the-fly）的狀態探索方法來檢查，其架構如圖 3-5所示。

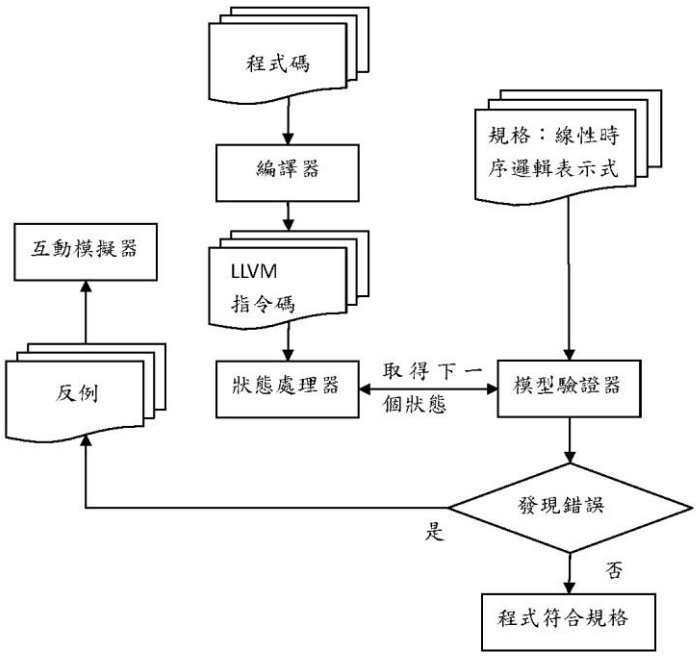


圖 3-5、漸進式的模型檢查架構圖

在漸進式的架構中使用深度優先搜尋（DFS）來探索整個模型是否有滿足規格。並沒有一次就把所有的狀態都計算（轉換）出來，而是由模型驗證器一邊驗證一邊主動向狀態處理器取得下一個狀態，當發現錯誤則可馬上停止，如此會比較有效率。此外在處理狀態的部分，原始程式碼編譯成 LLVM 的指令碼 [LLVM]，之後由狀態處理器執行指令碼並取得狀態，如此一來可以真實地取得並行程式的狀態。

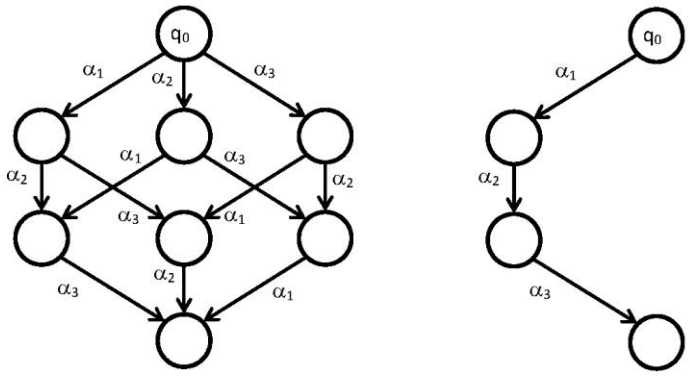
使用漸進式模型檢查方法必須要維護下列資料：

1. 記住從初始狀態到現在狀態的路徑以供 DFS 探索模型時使用。
2. 計算現在的狀態可以到達的下一個狀態。
3. 利用雜湊（hashing）記住曾經到過的狀態。

然而使用這個架構與方法還是有可能產生狀態爆炸的狀況。以下在章節3.1.2.1、章節3.1.2.2跟章節3.1.2.3介紹目前常見的改進方法。

3.1.2.1 部份模型簡化

部分模型簡化（partial order reduction）在減少需要檢查的狀態與路徑，提高模型檢查的可行性。其主要精神在於為每個自動機狀態 q 產生一個充足集合（ample set），這個充足集合是從狀態 q 開始的所有狀態轉換之子集，而充足集合保證透過深度搜尋進行模型檢查時，可以不探索所有從 q 開始的狀態轉換，而是只探索 q 的充足集合，同時不會影響模型檢查的正確性，如此以減少深度搜尋所需要探索的狀態。



(a) 簡化前　　(b) 簡化後

圖 3-6、部分模型簡化之示意

部分模型簡化的原理簡單敘述如下：假設程式有三個執行緒，分別以三個自

動機來表示，這三個執行緒個別執行三個動作α_1、α_2、α_3，整個程式則為此三個自動機的合成，包含這三個動作各種先後順序的組合，如圖 3-6(a)所示。對於這樣的一個系統，會需要檢查 8 個狀態，共六個路徑。倘若所要驗證的規格與這三個動作的先後順序無關，則不論程式執行動作的順序為$\alpha_1\alpha_2\alpha_3$、$\alpha_1\alpha_3\alpha_2$或$\alpha_3\alpha_2\alpha_1$，只要其中一種順序滿足該規格，則所有順序也都會滿足規格。所以狀態q_0的充足集合可以是$\{\alpha_1\}$，而簡化後的模型如圖 3-6(b)所示，只剩下四個狀態與一個路徑。

3.1.2.2 抽象化

抽象化（abstraction）是一種驗證時常用的手法，目標是簡化系統模型，並以簡化後的系統模型來進行模型檢驗，例如統一以述語x > 0 來代表x的值確實大於 0 的所有狀態。一般而言，抽象化會對原有系統模型進行高估（over-approximation），亦即確保抽象化的系統模型包含所有原有系統模型的行為。在這樣的情況下，有可能產生誤報的情況，這是因為太過於簡化系統模型，以至於簡化的模型具有一個不在原有模型並且違反規格的計算。然而這可以透過一個反例回饋抽象細緻化（counterexample-guided abstraction refinement）的機制來細緻化簡化的模型，並以此細緻化的模型（仍為原有系統模型的簡化）繼續進行模型檢驗 [Clarke et al. 2000]。此機制如圖 3-7所示。

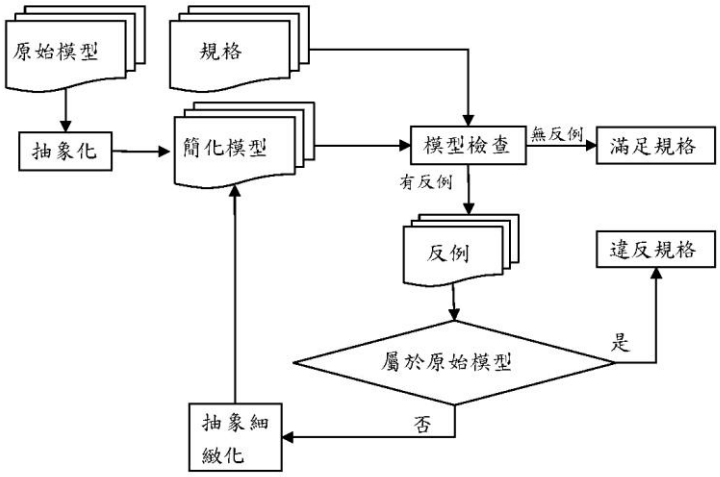


圖 3-7、反例回報抽象細緻化流程

另外有一種抽象化的實作方法是將與檢查性質無關的變數在驗證時不會作為狀態比對的依據，因為有些變數對於驗證的正確性是沒有影響的，以減少狀態數目，加速檢測時間 [Holzmann and Joshi 2004]。

3.1.2.3 可搶先次數上限檢查

根據文獻 [Musuvathi and Qadeer 2007]指出，在並行程式中，如果程式有任何錯誤，只需在極少的上下文切換（context-switch）次數下就可以找到。在 [Zaks and Joshi 2008]所提出的實作工具pancam中有實作此方法，該工具的實驗結果(圖3-8)也可以證明該方法是有效的。

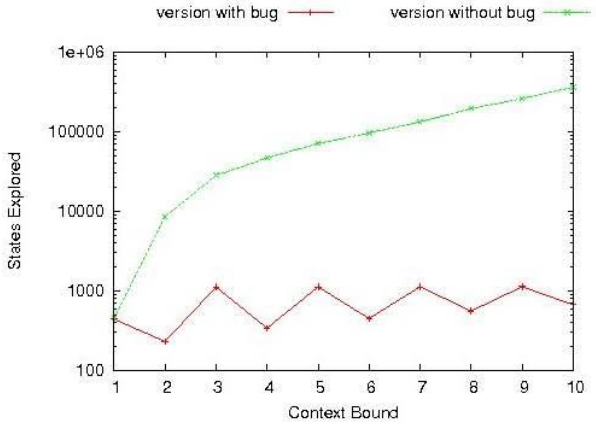


圖 3-8、在程式有錯誤的情況下，可以把狀態個數控制在一定範圍內 [Zaks and Joshi 2008]

3.1.3 基於pthread之並行程式驗證

C 語言本身並沒有執行緒的功能，若要使用多執行緒，則需依賴第三方函式庫，例如廣泛用於類 UNIX 作業系統的 POSIX Thread 函式庫（pthread）以及微軟作業系統的 Win32 Thread。此外，除了利用上述時態邏輯來表示並驗證程式運算的正確性，程式的並行性本身也有一些性質需要被檢驗，例如死結（deadlock）。

在多執行緒的並行程式中，共享變數可以被各執行緒存取，為了避免存取過程中共享變數被其他執行緒所改變，pthread 函式庫提供上鎖（lock）與解鎖（unlock）的函式：pthread_mutex_lock、pthread_mutex_unlock。這兩個函式都需要一個互斥器（mutex）變數：pthread_mutex_t，當一個互斥器被一個執行緒上鎖之後，就不能被其他執行緒上鎖。若一個執行緒要鎖上一個已經上鎖的互斥

器，則需等待該互斥器解鎖之後才能執行上鎖。如果所有的執行緒都在等待一個互斥器解鎖，則造成死結，系統將無止盡地原地等待。同樣地，如果執行緒有循環等待解鎖的關係，也會造成死結。例如執行緒 A 等待執行緒 B 對互斥器 M_1 解鎖，而執行緒 B 也在等待執行緒 A 對互斥器 M_2 解鎖。

為了方便各執行緒間進行同步，pthread 函式庫提供條件等待（conditional waiting）之函式：pthread_cond_wait，其輸入為一個條件等待變數 pthread_cond_t 以及一個互斥器變數。一個執行緒可以等待一個條件變數，當執行緒等待時，該執行緒不會執行任何指令。當其他執行緒對此條件變數發出信號或廣播時，等待之執行緒才會被喚醒。互斥器在此的作用是防止多個執行緒同時等待同一個條件變數。與互斥器類似，條件等待也有可能造成執行緒的死結。

本章節以 [Cordeiro and Fischer 2011]為基礎，介紹基於 pthread 之程式並行性驗證，包含互斥器與條件等待之塑模，以及死結之偵測。

3.1.3.1 互斥器上鎖運算之塑模

在一個最簡單的互斥器模型中，可以把一個互斥器變數看成一個可為 0 或 1 的整數變數，1 表示該互斥器已被上鎖，0 表示未被上鎖。當執行緒欲上鎖一個互斥器時，互斥器有鎖上以及未鎖兩個狀態，然而在已鎖上的狀態下，執行緒需持續等待直到該互斥器被解鎖。所以在分析驗證時只需要考慮一個狀況：執行緒上鎖一個互斥器時，該互斥器一定在未鎖狀態。上鎖函式 pthread_mutex_lock 的語意可以表示為以下程式碼：

```
int pthread_mutex_lock (pthread_mutex_t *m) {
  atomic { assume(*m == 0); *m = 1 }
}
```

其中 atomic{●}表示大括號中的指令可以連續執行完而不會被其他執行緒中斷，assume(e)則表示此時假設 e 為真，完全不考慮 e 不成立的狀況。而解鎖函式 pthread_mutex_unlock 則可表示為：

```
int pthread_mutex_unlock (pthread_mutex_t *m) {
  atomic { assume(*m == 1); *m = 0 }
}
```

上述簡單模型雖然可以表達上鎖與解鎖動作的語意，但卻無法偵測是否有死結發生。為了偵測死結，首先把執行緒的狀態分成以下五種：

1. 會合狀態（join state）：執行緒正在等待其他執行緒結束。
2. 鎖定狀態（lock state）：執行緒正在等待某個互斥器變數被解鎖。
3. 等待狀態（wait state）：執行緒正在等待某個條件變數的信號或廣播。
4. 離開狀態（exit state）：執行緒執行完畢。

5. 自由狀態（free state）：執行緒可以自由執行其指令。

其中除離開狀態以外的四個狀態都是執行緒正在執行的狀態，而會合狀態、鎖定狀態與等待狀態都是執行緒被凍結的狀態。所以當被凍結的執行緒個數等於正在執行的執行緒個數時，死結產生。考量執行緒狀態後，一個較完善的 pthread_mutex_lock 模型如下：

```
01 int pthread_mutex_lock (pthread_mutex_t *m) {
02 extern uint trds_in_run, c_lock = 0;
03   atomic {
04     unlocked = (mutex_lock_field(*m) == 0);
05     if (unlocked) mutex_lock_field(*m) = 1;
06     else c_lock = c_lock + 1;
07   }
08   atomic {
09     if (mutex_lock_field(*m) == 0)
10       c_lock = c_lock - 1;
11     if (!unlocked) {
12       deadlock_mutex = (c_lock < trds_in_run);
13       assert(deadlock_mutex);
14       assume(!deadlock_mutex);
15     }
16   }
17 }
```

其中 trds_in_run 表示正在執行的執行緒個數，c_lock 表示因為等待互斥器變數 m 解鎖而被凍結的執行緒個數，mutex_lock_field 可以將互斥器狀態讀取出來。3-7 行程式碼主要在讀取互斥器狀態，如果互斥器在未鎖狀態，則將之上鎖；反之則將因互斥器 m 而凍結的執行緒數目加一。11-14 行程式碼則利用第 13 行的檢核點來確保被凍結的執行緒個數必定小於正在執行的執行緒個數，若此檢核點通過模型檢查的驗證，可保證程式不會有死結發生。

3.1.3.2 條件等待之塑模

條件等待函式 pthread_cond_wait 的模型與 pthread_mutex_lock 類似，同樣會記錄因某條件變數而被凍結的執行緒個數。並設置檢查點以確保被凍結的執行緒個數一定小於正在執行的執行緒個數。其詳細模型如下：

```
01 int pthread_cond_wait (pthread_cond_t *c,
02                       pthread_mutex_t *m) {
03 extern uint trds_in_run, c_wait = 0;
04   atomic {
05     cond_lock_field(*c) = 1;
06     assert(mutex_lock_field(*m));
07     mutex_lock_field(*m) = 0;
08     c_wait = c_wait + 1;
09   }
10   atomic {
11     deadlock_wait = (c_wait < trds_in_run);
12     assert(deadlock_wait);
13     assume(!deadlock_wait
14            || cond_lock_field(*c) == 0);
15     c_wait = c_wait - 1;
16   }
17   mutex_lock_field(*m) = 1;
18   return 0;
19 }
```

其中 cond_lock_field 可取出條件變數中的互斥器變數。4-9 行程式碼所執行的動作如下：將條件變數 c 上鎖，確保互斥器 m 已被鎖上並將其解鎖，將等待條件變數 c 的執行緒個數加一。11-12 行程式碼設置檢核點以確保不會發生死結，14 行程式碼以條件變數 c 的解鎖來模擬信號與廣播，所以在第 15 行程式碼將等待條件變數 c 的執行緒個數減一。最後在第 17 行程式碼將互斥器 m 鎖上。

3.2 演繹式程式驗證

演繹式程式驗證（deductive program verification）為一種透過演繹推導（deductive reasoning）用來證明程式之正確性的方法。所謂的演繹推導是根據一群推導規則（inference rule），從一群公理（axioms）推導到要被證明的命題。

根據霍爾邏輯（Hoare logic）的邏輯理論以及陳述轉換理論（predicate transformer）所建構的驗證方法為一常見的演繹式驗證方法。以下在章節3.2.1先介紹霍爾邏輯的推導法則跟使用方式，於章節3.2.2介紹演繹式程式驗證的架構，於章節3.2.3介紹以演繹推導驗證並行程式的一些概念與手法。

3.2.1 霍爾邏輯

為了明確地表述「程式假設的環境」以及「程式的正確輸出」為何，可以使用兩個邏輯命題：前置條件（pre-condition）與後置條件（post-condition），兩者都是使用一階邏輯（first-order logic）來表述。前置條件用來表述進入該程式前，系統環境必須滿足的狀態；後置條件則用來表述離開該程式後的系統狀態[Hoare 1969]。

程式的正確性可以分成兩種：部份正確性（partial correctness）與完全正確性（total correctness） [Hoare 1969]。一個程式或程式片段 S 的部份正確性可表述為一個霍爾三元組（Hoare triple），記作{P} S {Q}，意思是在前置條件 P 滿足的狀況下，若 S 執行完畢，則系統狀態滿足後置條件 Q；而完全正確性可表述為[P] S [Q]，意思是在前置條件 P 滿足的狀況下，S 最終必然會執行完畢，而且系統狀態在 S 執行完畢時會滿足後置條件 Q。

舉例來說，下列程式片段 S：

```
y := 2;
z := x + y;
```

如果將 P 定為 x = 1，Q 定為 z = 3，則{P} S {Q}為真，因為當程式起始時 x 等於 1，程式執行後 y 等於 2，z 會被指定為 3。底下列出部份的霍爾邏輯公理與推導法則。

1. 公理（Assignment Axiom）：

$$\overline{\{P[E/x]\}\ x := E\ \{P\}}$$

例：

$$\{x+1=43\}\ y := x+1\ \{y=43\}$$
$$\{x+1\leq N\}\ x := x+1\ \{x\leq N\}$$

2. 合成規則（composition rule）：

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

3. 條件規則（conditional rule）：

$$\frac{\{B\wedge P\}\ S\ \{Q\}\ ,\ \{\neg B\wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

4. 迴圈規則（while rule）：

$$\frac{\{P\wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B\wedge P\}}$$

其中 P 為迴圈不變量（loop invariant）

例：迴圈不變量為 $\{x \geq 0 \wedge y > 0\}$

$\{x \geq 0 \wedge y > 0\}$
while $x \geq y$ **do**
$\{x \geq 0 \wedge y > 0 \wedge x \geq y\}$
$x := x - y$
od
$\{x \geq 0 \wedge y > 0 \wedge x \ngeq y\}$
// or
$\{x \geq 0 \wedge y > 0 \wedge x < y\}$

3.2.2 架構與方法

在本章節將介紹一個演繹式程式驗證常見的驗證方法，該方法使用中介語言（intermediate language）、驗證條件產生器（verification condition generation）、互動式證明器（interactive prover）跟可滿足模組理論問題解決器（satisfiability modulo theory solver 或SMT solver）。在此架構中，使用者必須要對程式碼下註解（annotation），以註解來表述程式片段的前置或後置條件，亦為要驗證的規格。其架構如圖 3-9、演繹式程式驗證架構圖：

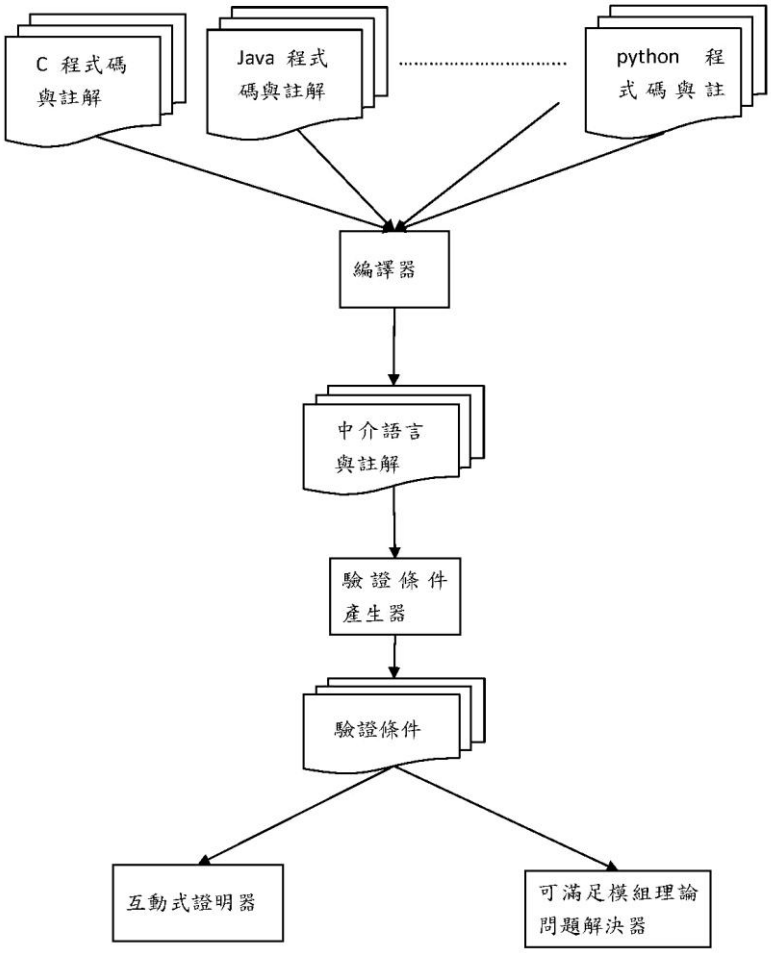


圖 3-9、演繹式程式驗證架構圖

由於原始碼可能包含各種不同的程式語言，所以這些具有使用者註解的原始碼首先會被轉換成一個中介語言（如 Boogie [Boogie]、CIL [CIL]。），其目的是讓驗證條件產生器只需要針對一種語言來設計，如此一來除了可以驗證不同語言的程式碼，也可以不用維護不同語言的條件產生器。例如法國實作出的 Frama-C 驗證工具就是將程式碼編譯成 CIL（C intermediate language），再針對 CIL 驗證 [Frama-C]，微軟的 VCC 也是使用這種架構 [Cohen et al. 2009] [VCC]。

21

驗證條件產生器的功能是根據含有註解的中介語言程式碼產生驗證條件，並保證若所有驗證條件都可被證明為真，表示原始程式碼滿足使用者所給的註解（規格）。而驗證條件產生器的原理是根據陳述轉換理論和最弱前置條件（weakest pre-condiction），以此產生驗證程式符合註解所需的條件 [Dijkstra 1975]。

有註解的程式碼	驗證條件
$\{P_1\}$ $\{Q_6\}$ S_1 $\{Q_5\}$ … $\{Q_4\}$ $(Q_4 = wp(S_i, P_2))$ S_i $\{\{P_2\}\}$ $\{Q_3\}$ S_{i+1} $\{Q_2\}$ … $\{Q_1\}$ $(Q_1 = wp(S_n, P_3))$ S_n $\{P_3\}$	1. $\forall v.P_2 \rightarrow Q_3$ 2. $\forall v.P_1 \rightarrow Q_6$

表 3-2、驗證條件產生過程

以表 3-2為例，其中S_1, S_i, S_{i+1}, S_n為程式片段；P_1、P_2、P_3為使用者給予的程式註解，也就是程式要滿足的條件；Q_1, Q_2, Q_3, Q_4, Q_5, Q_6為驗證條件產生器根據程式註解與程式指令所產生的最弱前置條件。一個程式片段S執行完後要滿足後置條件P的最弱前置條件可以記作wp(S, P)，意思是指執行S前最少要滿足這樣的條件，如此執行S後才能滿足後置條件P。表 3-2中驗證條件產生器一路從P_3往回產生前置條件，當遇到程式註解P_2時，由於在S_{i+1}前使得整個程式最終滿足P_3的最弱前置條件為Q_3，而使用者要求S_i的後置條件為P_2，因此最終是否滿足P_3的一個驗證條件為：P_2是否隱含Q_3；當推導到P_1時，則可產生另一個驗證條件。所以只要證明這兩個驗證條件，就能證明整個程式滿足使用者註解 [Dijkstra 1975]。

有時候驗證條件可以自動化求解，無法自動化的驗證條件則需要驗證人員與工具互動來證明。可滿足模組理論問題解決器(SMT solver)就是自動化證明工具的一類，不同的可滿足模組理論問題解決器能夠解決的問題領域不一樣，如 Mini smt可以解非線性運算的問題 [MiniSmt]，Z3可以解線性運算跟陣列的問題 [Z3]。所以在面對不同問題時可以使用不同的問題解決器。在互動證明方面，則可使用

互動證明輔助工具，讓證明的過程更加嚴謹，例如 Coq [Coq proof assistant]、Isabelle/HOL [Nipkow, Paulson and Wenzel 2002]等。

3.2.3 以演繹推導驗證並行程式

在章節3.2.1與章節3.2.2分別介紹了霍爾邏輯的基本概念與演繹推導式驗證程式的基本架構，本章節將主要以VCC [VCC] [MSR-TR-2009-15 2009] [Cohen et al. 2009] [Ernie Cohen 2010]為基礎，介紹以演繹推導驗證並行程式的一些概念與手法。

3.2.3.1 物件不變量

在並行程式中，除了程式片段本身必須滿足使用者給定規格以外，由於一個變數或物件（以下以「物件」統稱）有可能被兩個以上的執行緒所讀取或寫入，而執行緒的這些讀取與寫入動作可能會互相干擾，導致預期之外的結果，因此在並行程式中確保物件維持某種一致性是相當重要的。

物件的一致性可以用不變量來表示，所謂不變量是指不會改變的一個值，在演繹推導方式中，常以一階邏輯或其他邏輯式子來表示物件不會改變的性質。例如對一個排序串列物件 l 而言，其物件不變量可為：

$$\forall i, j.\ 0 \le i \le j < size(l) \rightarrow acc(l, i) \le acc(l, j)$$

其中 acc(l,i)表示串列 l 在位置 i 的值，而這個式子所描述的就是串列 l 是由小到大排序好的，亦即對於所有兩個串列位置 i 與 j，若位置 i 在位置 j 之前，則串列 l 在位置 i 的值不會大於在位置 j 的值。

為了要確保物件不變量能夠被並行程式所維持，驗證條件產生器必須在每一行程式碼都為每一個物件產生一個驗證條件，確保其不變量之成立。然而這樣會導致大量的驗證條件被產生，拖累驗證的效率。此外，某些物件的不變量可能會在函式中暫時被破壞，但是在函式結束前不變量會再度被確立。例如一個左右平衡的二元搜尋樹資料結構：AVL樹，一個加入元素到AVL樹的函式avl_add可能會因為新元素的加入而暫時破壞AVL樹左右的平衡，但是函式在元素加入後會透過子樹的旋轉重新平衡此AVL樹。針對這個問題以及驗證的效率，可以透過章節3.2.3.3所介紹的物件開放與封閉來改善。

```
function avl_add(AVLTree t, Element e) {
   insert e to t;
   rebalance t;
}
```

3.2.3.2 擁有關係

如果一個物件可以同時被兩個以上的執行緒所更改，則可能發生競賽情況（race condition），根據各個執行緒的執行順序，此物件將會有不同的變化。此外若將各執行緒的執行順序與相互影響考慮進來，則驗證過程會更加複雜。因此一般在並行程式的驗證中，都會確保物件在一段時間內只會被一個執行緒所更改。

擁有概念的產生即是為了確保物件只會被其擁有者所更動，一個物件同時只會有一個擁有者，而一個物件可能擁有數個物件，因此物件的擁有關係便形成一個樹林（forest），如圖 3-10，其中一個點代表一個物件，而一個點A指向點B表示物件B擁有物件A。物件的擁有關係可以是變動的，擁有權的轉移可透過上鎖（lock）或其他機制來達成。

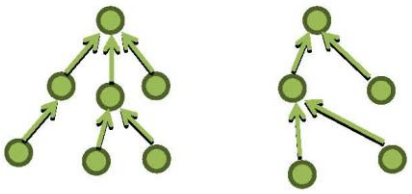


圖 3-10、物件擁有關係圖

以前述加入元素至 AVL 樹的函式 avl_add 為例，可以在更動 AVL 樹前確保該樹的擁有者是執行此函數的執行緒，使得在加入元素的過程中不會被其他執行緒

```
function avl_add(AVLTree t, Element e) {
   { owner(t) = me }
   insert e to t;
   rebalance t;
}
```

所干擾。而這可以透過驗證條件 owner(t) = me 來達成，其中 owner(t)表示物件 t 的擁有者，而 me 則代表執行此函式的執行緒。

3.2.3.3 開放與封閉

對於每一個物件，可以額外賦予開放與封閉的性質。若一個物件是開放的，表示此物件可以自由地被更動；反之，若一個物件是封閉的，則此物件不能被更動，其不變量必須被滿足。如此一來，驗證條件產生器只需要在物件開放與封閉的時間點確保物件不變量成立即可。而使用者則必須在程式適當的地方告知驗證條件產生器何時物件是開放或封閉的。

在 VCC 中，可以透過 wrap(o)來執行以下動作：

- 確立物件 o 的擁有者是當前的執行緒。
- 確立物件 o 是封閉的，其不變量亦必須成立。
- 將物件 o 設成開放。

另外可透過 unwrap(o)來執行以下動作：

- 確立物件 o 的擁有者是當前的執行緒。
- 確立物件 o 是開放的。
- 確立物件 o 的不變量成立。
- 將物件 o 設成封閉。

以 avl_add 函式為例，可以在加入元素前安插幽靈程式碼（ghost code）unwrap(t)，並在函式最後安插幽靈程式碼 wrap(t)，使得驗證條件產生器會針對這兩個程式點產生驗證條件，確保物件 t 的不變量在這兩個程式點成立，同時物

```
function avl_add(AVLTree t, Element e) {
  unwrap(t);
  insert e to t;
  rebalance t;
  wrap(t);
}
```

件 t 只會被當前執行緒所擁有並進行更動。這裡所稱幽靈程式碼是指不影響原有程式碼之執行，但與程式規格相關的程式碼。與幽靈程式類似的還有幽靈物件，意指不屬於原始程式而與規格相關的物件，例如物件的開放與封閉性可以透過賦予物件一個幽靈布林變數來表示。

3.2.3.4 接受性

考量一個物件 o，若其不變量只和 o 本身有關，則其不變量只會在物件 o 開放時因為擁有者改變物件內容而暫時被破壞，並在物件封閉時重新建立不變量。然而若物件 o 的不變量牽涉到其他物件，則更改其他物件的同時，很有可能破壞物件 o 的不變量，所以物件不變量的確立不能只有在物件開放與封閉時進行。為

了避免這個問題，可以使用物件不變量的接受性（admissibility）。

對於一個物件o而言，其物件不變量的接受性是指：當程式觸發一個動作時，若此動作可維持所有其他物件的不變量，則該動作不會破壞物件o之不變量。若物件o的不變量只牽涉到物件o本身，則顯而易見地這個不變量滿足接受性。在VCC中，驗證條件產生器會產生驗證條件以確保所有一般物件的不變量都滿足接受性，如此一來，只要再確保物件開放與封閉時其不變量之成立即可。

3.2.3.5 物件分享

先前所提到的一些概念與手法都是針對物件被更動時能夠確立不變量，然而當物件被其他物件所讀取而非寫入時，則可有比較少的限制。假設物件o的擁有者將物件o分享出來，由於物件的讀取者在讀取物件o時可能假設其不變量成立，所以：

- 當物件o有可能被讀取時，物件o的擁有者不能更改物件o，亦即不能開放物件o。
- 物件o的讀取者只能在物件o是封閉時才能讀取。

為了確保以上兩點，可以透過識別物件（handle）的發送與回收。每一個物件都會帶有一些識別物件，並記錄自己發出識別物件的數量。當物件o要讀取物件p時，會向物件p索取其識別物件，並於讀取完畢時歸還。而當物件o的擁有者想要更改物件o時，會先檢查物件o所發出的識別物件數量是否為0。這些識別物件的發送、回收以及數量計算都可以透過安插額外的幽靈程式碼與幽靈物件來達成，並由驗證產生器產生驗證條件來確保物件識別機制的正確性。

4 執行時間分析之方法與理論

在章節 4.1 中，介紹如何以靜態分析以及動態執行量測兩種不同方法估算單一工作或執行緒「最差情況執行時間」(Worst-Case Execution Time)；在章節 4.2 中，介紹在單一執行緒執行時間已知的情形下，如何利用排程（scheduling）理論分析即時多執行緒程式的執行時間。

4.1 計算WCET之方法與理論

計算單一工作或執行緒的「最差情況執行時間」的方法可分為兩大類：靜態分析和測量式（measurement-based）方法，以下簡介各方法所運用的技術和理論。

4.1.1 靜態分析方法

此類方法不依賴在真正硬體或模擬器上運行程式，而是將程式碼與抽象的系統模型結合，藉此得到執行時間的上限。圖 4.1 表示靜態時間分析工具的核心內容和資訊流。

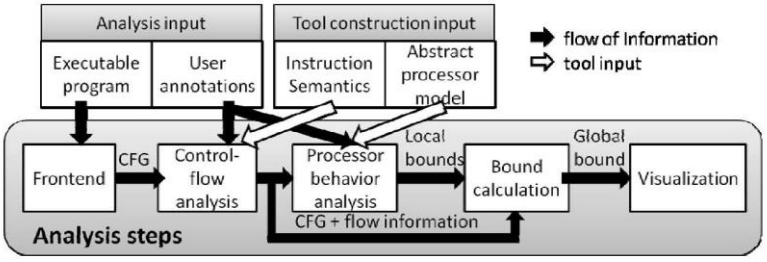


圖 4-1、靜態時間分析工具的核心內容和資訊流 [Wilhelm et al. 2008]

4.1.1.1 數值分析（Value Analysis） [Heckmann and Ferdinand 2004]

數值分析為靜態的程式分析，用來決定處理器的暫存器（register）和區域變數值在每個程式點的範圍，分析結果可以用來決定可能的間接記憶體存取，對於快取分析和迴圈界限分析來說是重要資訊。通常無法決定確切的數值，而是找到安全的最低與最高界限，也就是保證含有確切數值的區間。

數值分析使用抽象解釋理論（abstract interpretation）的框架：一抽象狀態將暫存器對應到可能值的區間，因此所有處理器指令皆需塑模成抽象狀態的運算元，包含簡單的數學運算元像是加或乘，甚至是複雜的尋址模式像是比例索引的暫存器間接尋址（register indirect with scaled index）等也需作轉換，來近似有效地址的記憶體參考。

以下以 add 指令為例，D_{abs} 為抽象的暫存器或記憶格（memory cell），[l, u] 代表一 D_{abs} 所對應的區間，l 為最低數值而 u 為最高數值。add 指令將兩 D_{abs} 相加後產出新的 D_{abs}，新的 D_{abs} 的數值區間落在原本兩 D_{abs} 最低數值加總（$l_1 + l_2$）和最高數值加總（$u_1 + u_2$）之間，並且因為實際的暫存器或記憶格大小為有限的，所以須作溢位的檢查，若溢位發生則新的 D_{abs} 的數值區間為未知 [Ferdinand et al. 2001]。

$$\text{add}: D_{abs} \times D_{abs} \rightarrow D_{abs},$$

$$[l_1,u_1] + [l_2,u_2] = \begin{cases} [l_1 + l_2, u_1 + u_2], & \text{若不可能溢位} \\ \text{未知}, & \text{若可能溢位} \end{cases}$$

任何資料快取行為分析的方法為了判斷記憶體存取的發生位置，皆需要知道有效的資料記憶體位址，而有效的位址只能在執行時得到，但透過數值分析，我們在每個程式點計算處理器中暫存器（register）和區域變數值的範圍，靜態地決定有效位址。

4.1.1.2 控制流程分析（Control-Flow Analysis） [Wilhelm et al. 2008]

控制流程分析的目的為收集可能的執行路徑，因為程式最終將停止是被保證的，所以路徑的集合的個數為有限個。而一般我們不計算路徑的真正集合，而是計算包含此真正集合的超集（superset），此超集將是安全的近似值，且集合愈小愈好。

控制流程分析的輸入包含程式的表示，例如函數調用關係圖（call graphs）、控制流程圖和可能的額外資訊，如輸入資料的範圍、迴圈的執行次數限制，可靠事先的數值分析判斷或由使用者提供。輸出的結果可視為程式動態行為的限制，包含哪些函式將會被呼叫、條件間的關係和路徑的可行性等。

自動化的控制流程分析有許多方法，部分專門用於特定的程式架構，也會因所分析的程式類型而有不同，例如原始碼、中間碼（intermediate code）或是機器碼（machine code）。通常在原始碼作控制流程分析較在機器碼上簡單，但很難將結果對應到機器碼的程式，因為編譯時可能因程式碼的最佳化或是連結（linking）而改變控制流程的結構。

控制流程分析的輸出為一註解語法樹（annotated syntax tree）和控制流程圖的轉移（transition）資訊集合，前者用於結構式（structure-based）方法，詳見

章節 4.1.1.4（d）；後者用於隱式（implicit）路徑列舉方法時轉成系統限制，詳見章節 4.1.1.4（c）。

4.1.1.3 處理器行為分析（Processor-Behavior Analysis）[Wilhelm et al. 2008]

處理器包含許多元件來使執行時間上下文相關（context-dependent），像是記憶體、快取、管道（pipeline）和分支預測。單個指令的執行時間與其執行歷程相關，為了精準地找出執行時間邊界，需要分析對於所有到達該指令路徑的處理器元件的佔有狀態。處理器行為分析決定佔有狀態的不變量（invariants）。

工具須將處理器的周圍相關條件納入考量才夠完整，像是完整的記憶體層級、匯流排、外圍裝置等，因此「硬體子系統行為分析（hardware-subsystem behavior analysis）」可能是更恰當的名稱。此分析對象為連結的可執行檔，因其含所有所需資訊，以處理器的抽象模型為基礎，加上考量保守計算的實際硬體時間，代表其預測的執行時間高於在實體處理器上可觀察到時間限制。

產生抽象處理器模型的複雜度主要由所用處理器的種類決定。對於 8 位元和 16 位元的處理器，產生時間模型是簡易但仍費時的。影響處理器行為分析的複雜因素包含因為參數值而使指令有不同執行時間，以及因為記憶體區存取次數而使資料有不同的參考時間。對於更先進的 16 位元（advanced 16bit）或 32 位元處理器，其擁有簡單的管道和快取，因為沒有不規則的執行順序，我們可以分別分析不同的硬體功能。而造成分析複雜的因素除了和 8、16 位元處理器相似外，還包含因快取命中（cache hit）與否而有不同存取時間、指令間不同的管道重疊（overlap）。而更先進的處理器擁有許多性能增強功能可互相影響，使執行順序不規則，因此建立時間模型非常複雜。

一般情況下，一個指令的執行時間上限由執行到此指令時的處理器狀態決定，而處理器狀態資訊由分析到此指令的潛在執行歷程決定。某個指令可能只在特定狀態下可執行，因此可能有不同的執行時間，影響計算的精準度。例如若一迴圈執行次數為 100 次，而最差情況下，某一段內容只會執行一次，則過度計算其他次的執行時間。因此我們可根據資訊流的上下文分開考慮所屬的執行歷史，資訊流上下文以透過迴圈和呼叫控制哪些路徑可到達該指令來表示。當處理器執行狀態未知時，以保守假設或去探索所有可能性。

處理器行為分析中大部分方法運用資料流分析（data flow analysis），一基於抽象解釋理論的靜態程式分析技術，這些方法用來計算在各程式點處理器執行狀態的不變量，若對於每個程式點存在同一不變量，表示到此程式點的所有的執行路徑皆滿足該不變量。若有不同路徑皆能到達某一基本區段（basic block），則在該區段程式點可能有不同的不變量，因此須計算多個不變量，每個不變量滿足一執行路徑的集合，聯集所有集合則產生到達此程式點的所有執行路徑集合，我們有時稱此種集合為調用上下文（calling context），也可單稱上下文。而不變量表

示了快取內容、功能單元的佔用、處理器序列和分支預測單元狀態的靜態知識，其中快取內容的知識將用來分類記憶體的存取是否為確切的快取命中。而管道序列和功能單元佔用的知識將用來排除管道延遲（pipeline stalls）。

4.1.1.4 估計計算（Estimiate Caculation） [Wilhelm et al. 2008]

估計計算的目的是決定 WCET 的估計值，動態方法可能因只計算了全部的子集合而低估 WCET 值，而將各程式片段的執行時間加總又可能高估 WCET 值。靜態方法則根據前階段產生的資訊流和時間資訊，計算程式的執行時間上限，通常稱為界限計算（bound calculation），可分為三大類方法：結構式（structure-based）、路徑式（path-based）和運用隱式路徑列舉的技術（implicit path enumeration technique，IPET）。

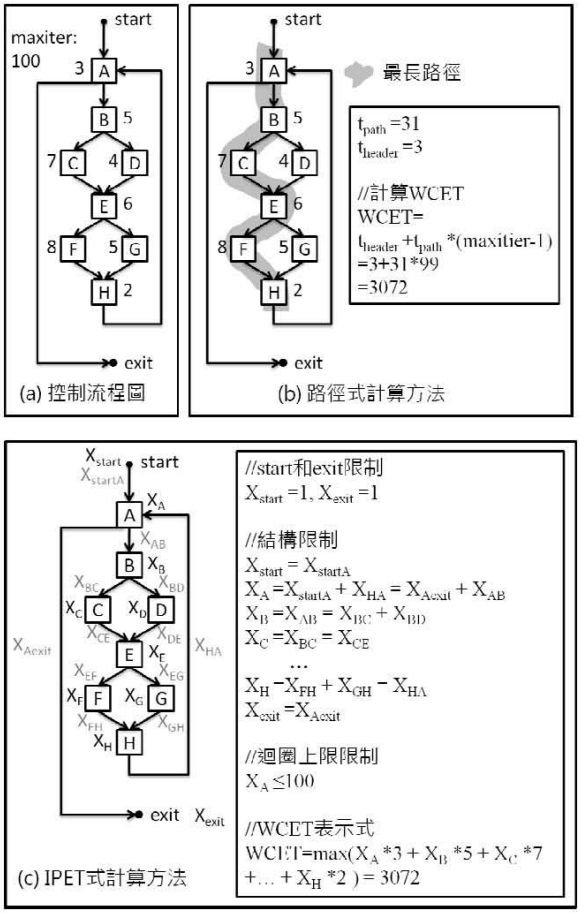
圖 4-2表示三種不同的方法：路徑式方法、IPET方法、結構式方法，其中圖 4-2 (a)為控制流程圖，在節點上有時間資訊，並含迴圈上限資訊，下述各方法皆以此圖進行計算。圖 4-2 (b)為路徑式方法，此方法計算不同路徑的執行時間上限，尋找所有路徑中最長的執行時間來決定程式的執行時間上限。特點為可以明確表示可能的執行路徑，在單一迴圈中此方法相當合乎常情，但當資訊流橫跨巢狀迴圈層級時便會發生問題，路徑數量將隨著分支點數量而指數成長，因此可能需要啟發式搜尋法（heuristic search method）來改善效率。

圖 4-2 (c)為IPET方法，此方法將程式的資訊流和基本區段執行時間上界合併為算數限制式的集合，給定每個基本區段和程式資訊流邊（program flow edge）一時間係數t_{entity}，用來表示該個體每次執行時對於整體執行時間的貢獻上界，並用變數x_{entity}來表示該個體執行次數。加總執行時間和執行次數的相乘來決定上界$\sum_{i\in entities} x_i * t_i$，執行次數變數受程式結構和可能資訊流的限制影響。IPET的結果為時間上界和最差情況下每個個體的執行次數。

圖中顯示以 IPET 方法產生的限制和規則，start 和 exit 限制表示程式必開始和結束一次。structural 限制表示可能的程式資訊流，代表進入一基本區段的次數必相同於離開該基本區段的次數。loop bound 限制表示迴圈開頭節點 A 可以執行的次數。IPET 方法運用整數線性規劃（integer linear programming，ILP）或是限制規劃（constraint programming，CP）技巧，因此複雜度隨程式大小可能指數成長，並且因為資訊流的事實將轉成限制，限制系統大小隨著資訊流的事實數成長。

圖 4-2 (d)為結構式方法，圖中表示此方法如何根據語法樹和給定的轉換規則（transformation rules）進行計算，節點的集合折疊成單個節點，同時對此新節點產生時間資訊，因為不同的資訊流上下文可能導致不同的執行時間，因此須考慮不同的資訊流上下文來增加準確度，而資訊流上下文需要轉換語法樹來反應不同的上下文，大多數較有利的轉換可簡單地在語法樹上表示，例如迴圈展開（loop unrolling）。結構式方法的問題為非所有的控制流程皆可用語法樹表示，此

方法假設在來源（source code）結構和不輕易接受程式碼最佳化的目標程式（binary code）間，有非常簡單的對應關係，因此一般不可能如同IPET般可以增加額外的資訊流。



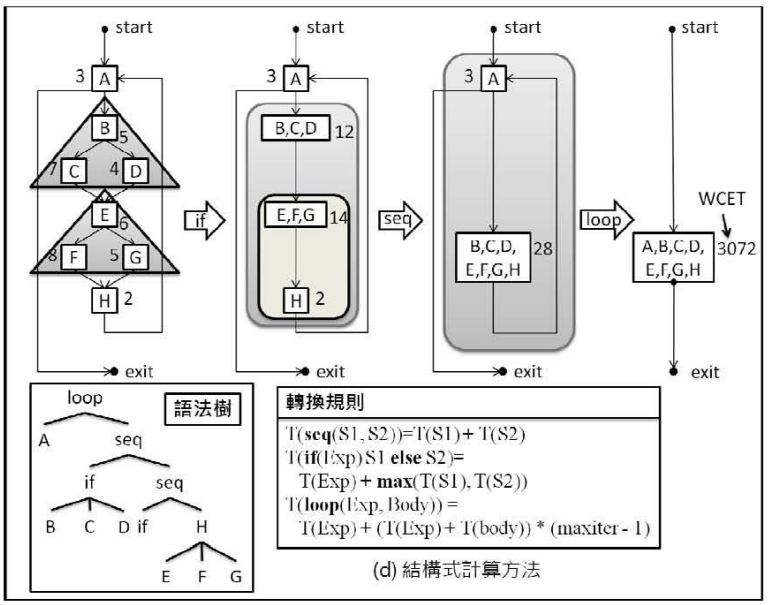


圖 4-2、3 種估計計算方法示意圖 [Wilhelm et al. 2008]

4.1.1.5 符號模擬（Symbolic Simulation） [Wilhelm et al. 2008]

符號模擬為另一種以處理器的抽象模型來模擬程式執行的靜態方法，此模擬不需輸入資料，因此模擬器需要能處理部分未知的執行狀態。此方法結合資訊流分析、處理器行為預測和界限計算在一整合階段，其問題為分析時間正比於程式實際執行時間，因此可能導致分析時間非常長。

4.1.2 測量式方法(Measurement-Based Method）[Wilhelm et al. 2008]

此種方法以給定的輸入集合在硬體或模擬器上執行程式來測量執行時間，可用於提供實際執行時間變化的概況，亦可用於驗證靜態分析方法的結果。此方法對所有可能執行路徑的子集合進行測量，產出執行時間的估計或是分布，但因為無法保證子集含有最差情況，所以並非執行時間的上界。

此方法也可測量程式碼片段（通常為控制流程圖的基本區段）的執行時間，再以界限計算來分析，產生 WCET 或 BCET 的預測，也就是以測量取代靜態方法中的處理器行為分析。當以控制流程分析找尋所有可能路徑時，結合所測量的時

間作界限計算，此結果會包含所有可能路徑，但若測量的基本區段時間不安全的話，也會導致最後預測的執行時間不安全。

此方法的另一問題為只使用了所有可能上下文的子集，解決方法為進行更多測試來測量更多上下文，或是對於每個測量程式碼片段設定最差情況初始狀態。前者降低不安全的預測結果出現機率但非完全避免，對所有路徑測試的代價高且經常難以作到。後者若能決定最差情況初始狀態便能使預測結果為安全的，但對於複雜的處理器來說，得到最差情況初始狀態是困難且近乎不可能的。目前測量式方法的工具可對簡單時間行為的處理器計算執行時間界限，但有上述問題的複雜處理器則尚未支援。

進行測量有許多方法，最簡單的方法是透過額外的插樁程式碼（instrumentation code）來收集時戳（timestamp）或 CPU 循環計數，而混合硬體/軟體的插樁技術需要外部硬體來收集輕量插樁程式碼的時間。使用邏輯分析器進行全透明（非侵入性）的測量機制是可行的，另外硬體追蹤機制如 NEXUS 標準和 ARM 用的 ETM 追蹤機制皆為非侵入性，但不一定產生精確的計時。例如 Nexus 將輸出存在緩衝區（buffer）中，當事件離開緩衝區時才產生時戳，造成中間存在延遲。

4.2 排程方法與理論

當多個任務（task）共享資源時，為了避免資源需求的衝突，需要排程來決定任務執行、得到資源的順序。以下簡介排程的方法及理論。

4.2.1 區域排程分析（Local Scheduling Analysis） [Künzli et al. 2007]、 [Henia et al. 2005]

任務由事件（event）來觸發，事件像是定時器（timer）到期、內部或外部的中斷（interrupt）和連串、接續的任務等。排程分析將個別的事件觸發抽象為事件流(event stream)，根據事件流可系統化的產生最差狀況的腳本(scenarios)，藉此計算任務的最差情況回應時間（worst-case response time），也就是從任務觸發到結束的時間差。

圖 4-3表示兩任務共享兩資源的模型，R1、R2 為任務，Src1、Src2 為資源，E1、E2、E3、E4 為事件，此模型記錄了事件可能的觸發順序，稱為事件模型(event model)。

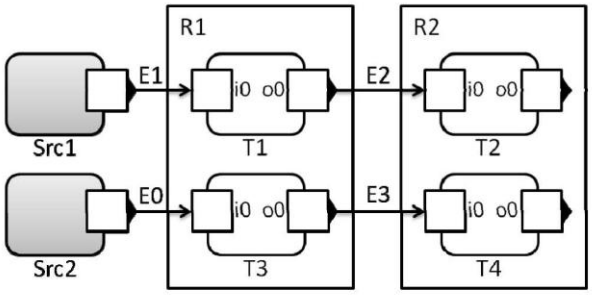


圖 4-3、兩任務共享兩資源的模型 [Henia et al. 2005]

事件模型以參數描述，例如有抖動（jitter）的週期事件以 2 個參數（P, J）表示，P為週期性發生事件的週期，J為可能因發生抖動而偏離原週期的差距，我們稱此模型為具抖動之週期事件模型（periodic with jitter event model）。當（P, J）為（4, 1）時，如圖 4-4表示，灰底的方格表示長度為 1 的抖動間隔，在週期為 4 的事件上不斷重複抖動。此圖亦表示了滿足事件模型的事件序列，每一事件必須在灰色的方格內發生，圖中向上指之箭號即表示一可能的事件發生序列。

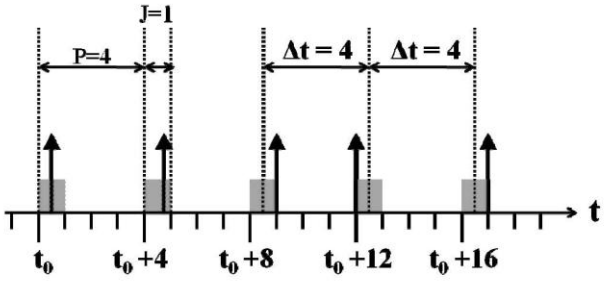


圖 4-4、滿足（P, J）=（4, 1）的事件模型 [Henia et al. 2005]

事件模型亦可以兩事件函式$\eta^u(\Delta t)$、$\eta^l(\Delta t)$表示，事件上界函式（upper event function）$\eta^u(\Delta t)$為在長度為Δt的區間中，事件發生次數的上限，事件下界函式（lower event function）$\eta^l(\Delta t)$為在長度為Δt的區間中，事件必須發生次數的下限。事件函式為單位高度的分段常數函式（piecewise constant step functions），圖 4-5表示（P, J）為（4, 1）的事件函式圖型，事件上界函式使用較小數值的點（白點），事件下界函式使用較大數值的點（黑點）作計算，在長度為Δt的區間中，事件發生的次數將在兩函式所限制的範圍內。

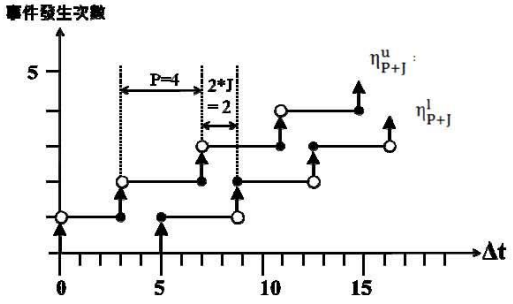


圖 4-5、事件上界函式$\eta^u(\Delta t)$和事件下界函式$\eta^l(\Delta t)$示意圖 [Henia et al. 2005]

以η^u_{P+J}、η^l_{P+J}函式表示含抖動的週期事件模型：

$$\eta^u_{P+J} = \left\lceil \frac{\Delta t + J}{P} \right\rceil$$

$$\eta^l_{P+J} = \max\left(0, \left\lfloor \frac{\Delta t - J}{P} \right\rfloor\right)$$

以圖 4-5為例，考慮Δt為 4 時，從事件上界函式我們可知在任何長度為 4 的區間中，事件最多發生 2 次，像是圖 4-4中$\Delta t + 8.5$到$\Delta t + 12.5$區間。從事件下界函式則得知在任何長度為4的區間中，事件最少發生0次，也就是可以不發生，像是$\Delta t + 12.5$到$\Delta t + 16.5$區間。

我們以距離函式$\delta^{min}(N \geq 2)$和$\delta^{max}(N \geq 2)$表示在一事件流中，連續事件數大於 2 的最小與最大距離。

$$\delta^{min}(N \geq 2) = \max\{0, (N-1) * P - J\}$$

$$\delta^{max}(N \geq 2) = (N-1) * P + J$$

例如在一含抖動的週期事件模型中，(P, J) 為 (4, 1)，則兩事件的最小距離為 3 時間單位，最大距離為 5 時間單位。

當事件為偶發（sporadic）時，事件下界函式$\eta^l(\Delta t)$永遠為 0，最大距離函式$\delta^{max}(N \geq 2)$為無窮。

利用以上的事件模型與函式，我們可用以下等式計算一任務之最差情況回應時間： [Jersak. 2004]

$$r_i = C_i + \sum_{\forall j \in hp(i)} \eta^u_j(r_i) * C_j$$

其中r_i為任務 i 之最差情況回應時間，任務 j 為執行優先序高於 i 之任務，hp(i)為一包含所有執行優先序高於 i 之任務集合，C_i與C_j為任務 i 和 j 之 WCET 值，$\eta^u_j(r_i)$為在長度為r_i的區間中，任務 j 的事件發生次數上界。此等式計算任務 i 被其他執行優先序較高任務中斷次數最多的情況，因此為最差情況回應時間。另外此等式忽略上下文切換（context-switch）之開銷（overhead），

且當 r_i 大於兩觸發事件的最小距離時此值無效。

此等式的計算方法如下。首先假設$r_i = C_i$，計算在r_i時間內任務 i 可被中斷的時間長度，將所有可能中斷的時間長度加至r_i，接著再次計算r_i時間內可能被中斷的時間長度，直到r_i不再增加或是r_i超過任務 i 兩觸發事件的最小距離時停止，若停止時r_i未超過任務 i 兩觸發事件的最小距離，則r_i為任務 i 之最差情況回應時間。

以圖 4-6為例來計算任務C之最差情況回應時間：

步驟 1. 設$r_i = 50$，帶入$r_i = C_i + \sum_{\forall j \in hp(i)} \eta_j^u(r_i) * C_j$的等號右邊，得到

$$r_i = 50 + \sum_{\forall j \in hp(i)} \eta_j^u(50) * C_j$$

$$\Rightarrow r_i = 50 + \left\lceil \frac{50+0}{100} \right\rceil * 30 + \left\lceil \frac{50+20}{100} \right\rceil * 30 = 110$$

任務 i 兩觸發事件的最小距離為：

$\max\{0, (2-1) * P - J\} = \max\{0, 1000 - 0\} = 1000$

$r_i = 110$ 並無超過最小距離，可繼續計算。

步驟 2. $r_i = 50 + \sum_{\forall j \in hp(i)} \eta_j^u(110) * C_j$

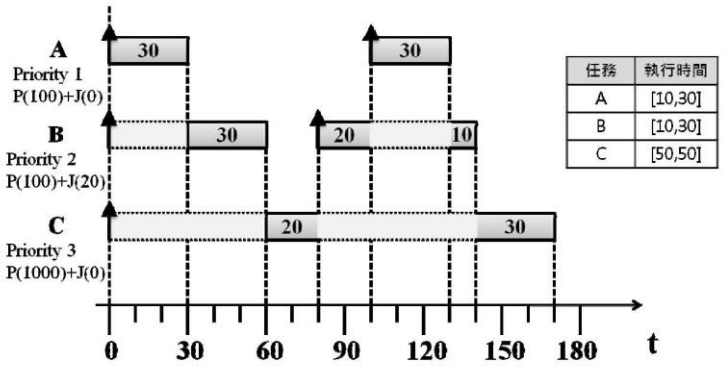
$$\Rightarrow r_i = 50 + \left\lceil \frac{110+0}{100} \right\rceil * 30 + \left\lceil \frac{110+20}{100} \right\rceil * 30 = 170$$

同樣$r_i = 170$ 並無超過最小距離，可繼續計算。

步驟 3. $r_i = 50 + \sum_{\forall j \in hp(i)} \eta_j^u(170) * C_j$

$$\Rightarrow r_i = 50 + \left\lceil \frac{170+0}{100} \right\rceil * 30 + \left\lceil \frac{170+20}{100} \right\rceil * 30 = 170$$

r_i與步驟 2 計算結果相同，停止運算，得到任務 C 的最差情況回應時間為 170 時間單位。



任務	執行時間
A	[10,30]
B	[10,30]
C	[50,50]

圖 4-6、任務 C 最差情況回應時間的發生情況 [Henia et al. 2005]

4.2.2 組合系統層級分析（Compositional System Level Analysis）[Künzli et al. 2007]、 [Henia et al. 2005]

在區域排程分析中，每個任務得到最差情況回應時間後，將輸出一事件模型，作為接續任務之觸發事件模型。輸出事件模型的週期與輸入該任務的觸發事件模型相同，抖動則要加上最長和最短回應時間差，如下式：

$$J_{out} = J_{act} + (t_{resp,max} - t_{resp,min})$$

J_{out}為輸入事件模型之抖動，J_{act}為的觸發事件模型之抖動，$t_{resp,max}$為最差情況回應時間，$t_{resp,min}$為最佳情況回應時間。

以圖 4-3為例，觸發R1 任務的觸發事件模型皆可得到，因此可藉由區域排程分析得到最差情況回應時間T1 和T3 以及輸出事件模型，此輸出事件模型為T2 和T4 的觸發事件模型，再由區域排程分析得到T2 和T4。

然而並非所有的情況皆能如上所述順利運作，像是圖 4-6的例子，一開始只有T1 和T3 的觸發事件模型為可計算的，此時系統分析無法運行，因為一任務所需觸發事件模型的資源不被滿足，計算R1 的回應時間時需要R2 的回應時間，反之亦然，我們稱此問題為相關排程循環（cyclic scheduling dependency）。

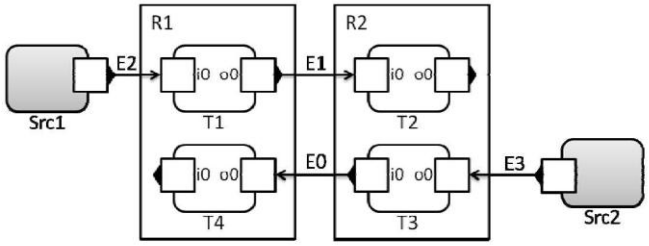


圖 4-7、兩任務共享兩資源且存在循環的模型 [Henia et al. 2005]

解決方法為初始時對所有系統路徑傳遞所有的外部事件模型，直到每個任務的觸發事件模型皆可計算。因為排程無法改變事件模型週期，且只增加事件模型抖動，而大的抖動區間包含小的抖動區間，最小抖動的假設安全，因此此方法是安全的。

在傳遞外部事件模型後，可運行全域系統分析，首先對各資源進行區域排程分析得到輸出事件模型，當所有輸出事件模型傳遞完畢，檢查是否有觸發事件模型資訊需要更新，若有代表輸出事件模型將可能被改變，因此重複區域排程分析。若所有模型在傳遞後皆不需要更新，表示達到收斂狀態，最後計算的最差情況執行時間為有效的。或是滿足中止條件，例如違反排序限制等，停止系統層級分析。

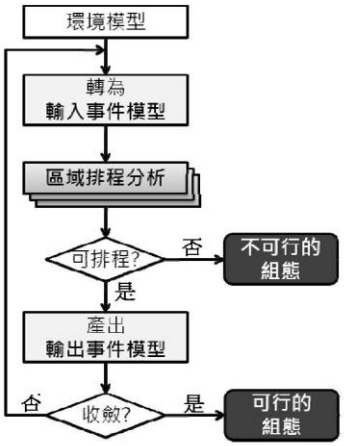


圖 4-8，組合系統層級分析示意圖 [Künzli et al. 2007]

4.2.3 處理多輸入之任務 [Jersak. 2004]

上面所述計算適用於任務只有單一個觸發事件時，當任務具多個輸入時，我們以布林函式將所有的輸入事件轉為一觸發事件模型。在此限制不可用輸入來使觸發無效，也就是不允許否定（negation），因此只需考慮 AND 與 OR 的情況，以下對此分開討論。

AND-觸發為當每個輸入皆發生輸入事件後，任務被觸發。如圖 4-9，當任務 A、B與C皆傳輸輸出事件模型後，任務K才被觸發。而在一次觸發所需的所有輸入皆抵達前，輸入資料須在緩衝記憶體等待，稱為AND-記憶體緩衝（AND-buffering ）。

為了確保 AND-緩衝記憶體的大小，所有輸入事件模型週期須相同，而觸發事件模型之週期便等同於所有輸入事件模型週期。而在任一時間區間 Δt 中，AND-觸發的發生次數 $\eta^u_{AND} = \max\{\eta^u_i\}$，$\eta^l_{AND} = \min\{\eta^l_i\}$，i 為所有輸入任務。抖動 $J_{AND} = \max\{J_i\}; i = 1 \ldots k$，k 為輸入任務個數。

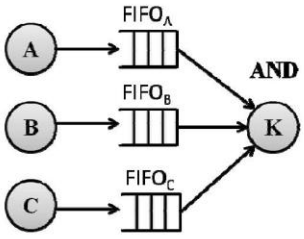


圖 4-9、AND-觸發示意圖 [Henia et al. 2005]

OR-觸發為當任一輸入事件發生，任務皆被觸發，其不需要緩衝記憶體，因為不需要等待其他輸入事件。我們以兩輸入之任務為例，設輸入的事件模型分別為(P, J)=(4, 2)與(3, 2)時，可分別建出圖 4-10之左圖與右圖。

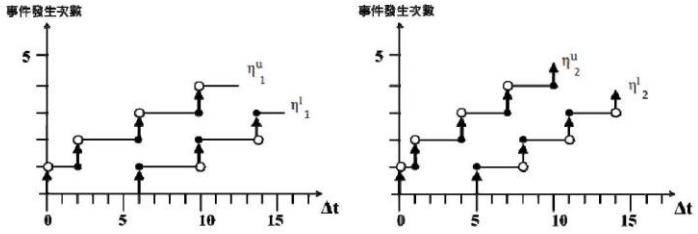


圖 4-10、事件上下界函式示意圖，左為(P, J)=(4, 2)，右為(P, J)=(3, 2) [Henia et al. 2005]

OR-觸發之事件上下界函式為所有輸入的加總：$\eta^{u}_{OR} = \sum_{i=1}^{n} \eta^{u}_{i}$ ，$\eta^{l}_{OR} = \sum_{i=1}^{n} \eta^{l}_{i}$ ，以圖 4-11 表示。

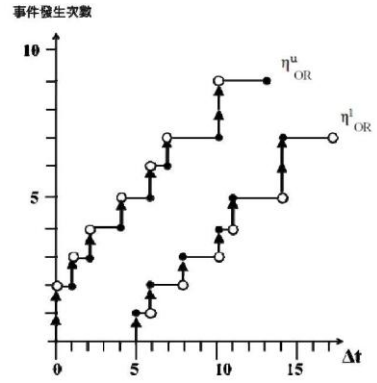


圖 4-11、兩觸發事件作 OR 合併之事件上下界函式示意圖 [Henia et al. 2005]

然而因為不規則的發展，我們無法直接以週期抖動模型描述，因此需要作保守的近似，如圖 4-12，虛線為真實事件函式，實線為近似後可用週期抖動模型描述之事件函式，近似後之上界大於原始上界，近似後下界小於原始下界。

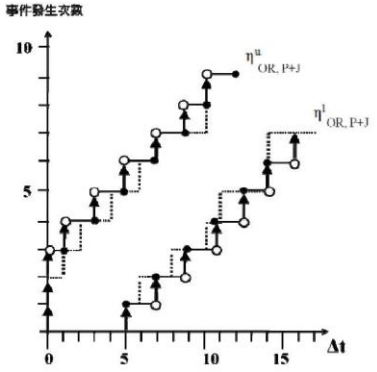


圖 4-12、近似後之 OR-觸發上下界函式示意圖 [Henia et al. 2005]

我們用以下方法來進行近似。針對週期作以下式計算：

$$P_{OR} = \frac{LCM(P_i)}{\sum_{i=1}^{n} \frac{LCM(P_i)}{P_i}} = \frac{1}{\sum_{i=1}^{n} \frac{1}{P_i}}$$

其中 LCM(Pi)為所有輸入事件模型週期之最小公倍數，此式在假設所有輸入事件抖動皆為 0 的情況下，以 LCM(Pi)除以在 LCM(Pi)下發生的輸入事件次數

合。

接著我們針對抖動作作近似，從$\eta_{OR}^{u}=\sum_{i=1}^{n}\eta_{i}^{u}$可得$\left\lceil\frac{\Delta t+J_{OR}}{P_{OR}}\right\rceil\geq\sum_{i=1}^{n}\left\lceil\frac{\Delta t+J_{i}}{P_{i}}\right\rceil$，而因等式兩邊皆為分段的連續，以下式分段計算不等式。

$$\left\lceil\frac{\Delta t+J_{OR,j}}{P_{OR}}\right\rceil\geq k_j;\Delta t_j<\Delta t\leq\Delta t_{j+1},k_j\in\mathbb{N}$$

進行簡化：

$$\lim_{\epsilon\to+0}\left\lceil\frac{\Delta t+\epsilon+J_{OR,j}}{P_{OR}}\right\rceil\geq k_j;k_j\in\mathbb{N}$$

$$\Leftrightarrow\lim_{\epsilon\to+0}\frac{\Delta t+\epsilon+J_{OR,j}}{P_{OR}}\geq k_j-1$$

$$\Leftrightarrow\lim_{\epsilon\to+0}(J_{OR,j}+\epsilon)\geq(k_j-1)*P_{OR}-\Delta t_j$$

$$\Leftrightarrow J_{OR,j}\geq(k_j-1)*P_{OR}-\Delta t_j$$

以兩輸入事件模型(P, J)=(4, 2)與(3, 2)為例，

$$P_{OR}=\frac{1}{\sum_{i=1}^{n}\frac{1}{P_i}}=\frac{1}{\frac{1}{4}+\frac{1}{3}}=\frac{12}{7}$$

$$\left\lceil\frac{\Delta t+J_{OR}}{\frac{12}{7}}\right\rceil\geq\left\lceil\frac{\Delta t+2}{4}\right\rceil+\left\lceil\frac{\Delta t+2}{3}\right\rceil$$

$$\left\lceil\frac{\Delta t+J_{OR,0}}{\frac{12}{7}}\right\rceil\geq 2;0<\Delta t\leq 1$$

$$\Leftrightarrow J_{OR,0}\geq 1*\frac{12}{7}-0=\frac{12}{7}$$

Δt 範圍	k_j	$J_{OR,j}$
$0<\Delta t\leq 1$	2	$J_{OR,0}\geq 1*\frac{12}{7}-0=\frac{12}{7}$
$1<\Delta t\leq 2$	3	$J_{OR,1}\geq 2*\frac{12}{7}-1=\frac{17}{7}$
$2<\Delta t\leq 4$	4	$J_{OR,2}\geq 3*\frac{12}{7}-2=\frac{22}{7}$
$4<\Delta t\leq 6$	5	$J_{OR,3}\geq 4*\frac{12}{7}-4=\frac{20}{7}$
$6<\Delta t\leq 7$	6	$J_{OR,4}\geq 5*\frac{12}{7}-6=\frac{18}{7}$

$7 < \Delta t \leq 10$	7	$J_{OR,5} \geq 6 * \frac{12}{7} - 7 = \frac{23}{7}$
$10 < \Delta t \leq 13$	9	$J_{OR,6} \geq 8 * \frac{12}{7} - 10 = \frac{26}{7}$
$13 < \Delta t \leq 14$	10	$J_{OR,7} \geq 9 * \frac{12}{7} - 13 = \frac{17}{7}$

經過Δt=LCM(P1,P2)=4 *3=12 開始重複，即可停止計算。從計算結果中選出最大者即為 OR-觸發之抖動，在此為$J_{OR} = \frac{26}{7}$。

5 結語

如我們在前一次期中報告結語中所指出，多執行緒程式邏輯正確性的自動化驗證有相當的難度，目前學界仍持續地努力尋求更好、更有效率的方法。加上即時性的要求後，驗證的難度更高，勢必要結合不同的方法方能得到更完整的分析驗證結果；我們目前仍持續探討如何做這樣的結合。另外，演譯式的驗證方法有其廣用性，但並非全自動化，是否能被實務界接受仍有待努力的推廣。

此外，造成高階多執行緒程式分析困難的主因在於，以靜態分析為主要方法的執行時間估算工具目前尚無法掌握特定即時作業系統的排程行為；而以實際量測為手段的工具則有先天的低估執行時間的可能性。為即時作業系統排程等行為建模，以做為時間估算之依據是根本解決之道；我們仍持續密切注意這方面的研究發展。

6 參考文獻

[Boogie] Boogie. Available at: http://research.microsoft.com/en-us/projects/boogie/

[Büchi Store] Büchi Store. Available at: http://buchi.im.ntu.edu.tw/

[CIL] CIL. Available at: http://www.eecs.berkeley.edu/~necula/cil/index.html

[Clarke, Emerson and Sistla 1986] E. M. Clarke, E. A. Emerson and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, vol 8, no. 2, pp. 244-263, 1986.

[Clarke et al. 2000] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-guided abstraction refinement. In Proceedings of the 12th International Conference on Computer Aided Verification, LNCS 1855, pp.154-169, Chicago, IL, USA, 2000.

[Clarke, Grumberg and Peled 1999] E. M. Clarke, O. Grumberg and D. A. Peled, *Model checking*, The MIT Press, 1999.

[Cohen et al. 2009] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte and S. Tobies. VCC: a practical system for verifying concurrent C. In Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, LNCS 5674, pp.23-42, Munich, Germany, 2009.

[Coq proof assistant] Coq proof assistant. Available at: http://coq.inria.fr/

[Cordeiro and Fischer 2011] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In Proceedings of the 33rd International Conference on Software Engineering, pp.331-340, Waikiki, Honolulu , HI, USA, 2011.

[Dijkstra 1975] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, vol 18, no. 8, pp. 453-457, 1975.

[Ernie Cohen 2010] Michal Moskal, Wolfram Schulte, Stephan Tobies Ernie Cohen. Local verification of global invariants in concurrent programs. In Proceedings of the 22nd International Conference on Computer Aided Verification, LNCS 6174, pp.480-494, Edinburgh, UK, 2010.

[Ferdinand et al. 2001] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In Proceedings of the 1st International Workshop on Embedded Software, LNCS 2211, pp.469-485, Tahoe City, CA, USA, 2001.

[Frama-C] Frama-C. Available at: http://frama-c.com/

[Heckmann and Ferdinand 2004] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH, 2004. http://www.absint.com/wcet.htm.

[Henia et al. 2005] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter and R. Ernst. System level performance analysis-the SymTA/S approach. *IEE Computers and Digital Techniques*, vol 152, no. 2, pp. 148-166, 2005.

[Hoare 1969] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, vol 12, no. 10, pp. 576-580, 1969.

[Holzmann and Joshi 2004] G. J. Holzmann and R. Joshi. Model-driven software verification. In Proceedings of the 11th International SPIN Workshop on Model Checking of Software, LNCS 2989, pp.76-91, Barcelona, Spain, 2004.

[Jersak. 2004] M. Jersak. Compositional performance analysis for complex embedded applications. , PhD thesis, Technical University of Braunschweig, 2004.

[Künzli et al. 2007] S. Künzli, A. Hamann, R. Ernst and L. Thiele. Combined approach to system level performance analysis of embedded systems. In Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, pp.63-68, Salzburg, Austria, 2007.

[LLVM] LLVM. Available at: http://llvm.org/

[MiniSmt] MiniSmt. Available at: http://cl-informatik.uibk.ac.at/software/minismt/

[MSR-TR-2009-15 2009] A practical verification methodology for concurrent programs. Technical report, Microsoft Research, MSR-TR-2009-15, 2009.

[Musuvathi and Qadeer 2007] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM SIGPLAN Notices*, vol 42, no. 6, pp. 446-455, 2007.

[Nipkow, Paulson and Wenzel 2002] T. Nipkow, L. C. Paulson and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, Springer, 2002.

[RTDruid] RTDruid. Available at: http://www.evidence.eu.com/content/view/28/51/

[SymTA/S] SymTA/S. Available at: http://www.symtavision.com/symtas.html

[VCC] VCC. Available at: http://research.microsoft.com/en-us/projects/vcc/

[Wilhelm et al. 2008] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, vol 7, no. 3, pp. 36-1, May 2008.

[Z3] Z3. Available at: http://research.microsoft.com/en-us/um/redmond/projects/z3/

[Zaks and Joshi 2008] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In Proceedings of the 15th International SPIN Workshop on Model

Checking Software, LNCS 5156, pp.325-342, Los Angeles, CA, USA, 2008.