主辦單位: CIC國家晶片系統設計中心暨

交通大學資訊工程科學研究所

九十九學年度大學校院

積體電路電腦輔助設計軟體製作競賽

競 賽 報 告

報名編號：___cada010___

組　　別：___定題A組___

Formal Property Verification

中華民國 100年 4月 23日

注意：

1. **為維護比賽公平性，請勿於報告中洩露比賽隊伍之身分。**
2. **為方便評比，敬請儘量依此格式撰寫報告。**
3. **報告限以中文或英文撰寫。**
4. **書面報告以25頁為上限，違者將取消參賽資格。字體大小以不小於11點字，行距以不小於1.5行距為原則，違者將提報評審委員會討論酌量扣分。**

一、 摘要

對於一個CNF（Conjunction Normal Form）design，一個 mutant 是其中某個變數在所有 CNF clauses 中的每個 occurrence 都被改為（positive ）/ 都被改為 （negative） / 或都被反向（negate）。

對於一個 design 的 property 則也是一些 CNF clauses，且個別與原 design AND 起來得到的 CNF 會是 Boolean satisfiable。若一個 mutant 對所有 property 都是 satisfiable，則稱此 mutant 為 live。

在此我們提供了有效率找出 live mutant 的方法，利用 mutant / property 與原 design 的特殊關係，對於極大的 input testcase 可以在數秒內就篩掉大部分的 mutant-property pairs 並得到大部分的 live mutant，而在幾分鐘內找到約 90% 的 live mutant；在這之後更應用各種加速方法（trim, Gröbner basis），結合 SAT engine 來找出剩下部分的 live-mutant。

二、 簡介

2.1 問題描述

對於一個 CNF 設計 D，若將

(1) 其中一變數的所有 occurrence 設為 positive，或

(2) 其中一變數的所有 occurrence 設為 negative，或

(3) 其中一變數的所有 occurrence 反向（negate）

且其他變數在 CNF clauses 中都保持不變，而得到一個新的 CNF，則稱這樣一個新的 CNF 為一個 「mutant」。

一個「Property」亦為一個 CNF clauses，且滿足如果和原先的設計 D 一起 conjunct (AND) 起來，能夠產生一組變數 assignment 使得整個 CNF 為 true，也就是說，可以滿足 Boolean satisfiability 特性。

在定題組 A2 的題目中，定義了 mutant 的 live 與 dead。如果一個 mutant 對於所有的 property 分別以 AND 連接後都可以有 Boolean satisfiability 特性的話，我們就稱這個mutant為 live。反之，只要有一組 property 與 mutant 配對後無法滿足Boolean satisfiability，則稱此 mutant 為 dead ，並稱此 property 為見證這個 mutant 為 dead 的「witness」。

給定一個原 design *D*，以及保證滿足這個 design 的一些 property set $P = \{ P_1, P_2, \ldots \}$ 和由這個 design 所得到的一些 mutant set $M = \{ M_1, M_2, \ldots \}$，本題的目的就是要找出所有 live 的 mutant 。

2.2 軟體功能及特性

使用 minisat 作為底層的 SAT engine，輸入 mutant 列表以及 property 列表，我們可以在一定時限內找出最多組 live 的 mutant 列表，並（可利用參數調整）可印出 mutant-property pairs solution 提供驗證。除此以外提供若干可開關的方法來進行加速。

2.3成果簡述

對於小的 input testcase (design 1)，即使不使用任何的 cut ，直接用 naïve solution 硬解所有的 mutant-property pair ，所花的時間仍不到 1 秒，這些 testcase 都能在極短的時間解完。

而對於極大的 input testcase ，使用 cut 可以在數秒內就篩掉大部分的 mutant-property pairs 並得到大部分的 alive mutant，藉由 SAT engine 的輔助，解開部分的 design-property ，可以在幾分鐘內找到約 90% 的 alive mutant；在這之後更應用各種加速方法（trim, Gröbner basis），結合 SAT-engine 來找出剩下部分的 alive-mutant。

三、 演算法

The Naïve Solution：

一個最顯然直接（卻也必定最沒效率）的做法就是直接 pairwisely 的對每個 mutant-property pair 進行驗證，也就是如原 Problem specification 所說的：

foreach Mutant $M_i \in M$

foreach Properties $\in P_j$

if (! SAT($M_i \wedge P_j$)) declare M_i to be **dead**

// witnessed by P_j

if (M_i not declared to be dead yet)

declare M_i to be **alive**.

其中 SAT 的部分可以用 SAT-solver 來達成。然而，這樣的做法基本上忽略了 Mutant 與 Property 和原 design 的關係（Mutant 只會和原 design 相差一個 variable，property 和 design 結合必為 satisfiable），只是把這個 FPQ 的問題當作一般 SAT problem 在解。因此雖然最終的測試必定還是基於這樣的 one-to-one testing （或加上少許 incremental 之類的改進），但是可以就這方面（Mutant 和原 Design 間的關係）構思一些 cut （剪枝）來加速搜尋 / 避免解不必要的 SAT Problem；除此以外另一方面亦可對解 SAT 本身進行加速。就這兩部分我們分別都做了改進，以下分別詳述之：

Notation:

為了方便接下來演算法的描述，我們定義一個 Mutant M 可以用一個 pair $\langle x,$ type$\rangle$ 來描述，若 M 是由原 design D 把所有變數 x 的 occurrence 設為 positive，則記作 $M\langle x,+\rangle$；若是將 x 設為 negative，記作 $M\langle x,-\rangle$；若是將 x 反向（negate）則記作 $M\langle x, \neg\rangle$。

定義一個 CNF pair (*A*, *B*) 單純就是把 *A*、*B* 兩個 CNF 直接 AND 起來（conjunction）。故 (*D*, *P*) 就是原本的 CNF 和某個 property P 相 AND 得到的 CNF。

首先注意到的是，因為即便 design / mutant size 有可能非常非常大，作為驗證的 property 一般 size 遠小於 design（clause數目少），故一個 mutation variable 很有可能根本沒有在 property clauses 中出現。在這樣的情況下，我們有以下結論：

Observation 1

考慮一個 mutant-property pair (*M*, *P*)，若 Mutant 為 *M*⟨*x*,*⟩ 且 *x* 在 *P* 中沒有在任何 clause 出現，那麼這個 pair (*M*, *P*) 進行 SAT 的結果必定是 satisfiable。（其中此處 *M*⟨*x*,*⟩ 代表 type 可以是 positive / negative / negate的任一種）

Proof:

由於任一 property P 和原 design D 合起來的 pair (D, P) 都必是 satisfiable，亦即存在一對每個變數的 assignment *S*: $V \rightarrow \{0,1\}$（其中 *V* 代表 variable set，*S* 即 *V* 中每個 variable 對應到他的取值的映射），滿足 *S* 代進 (*D*, *P*) 後會是 SAT。

- 現在考慮對於一個 Mutant *M*⟨*x*,+⟩，且 *x* 不在 *P* 的任何 clause 中。則令

$$S'(v) = \begin{cases} 1, & v = x \\ S(v), & otherwise \end{cases}$$

那麼可以發現，*S'*會滿足 pair (*M*, *P*)。要證明這點，我們先考慮所有有 *x* 這個 variable 的 clause ，因為 *M* 為 positive type 且 *P* 中完全沒出現 *x*，因此所有 *x* 在 (*M*, *P*) 中的 occurrence 都為 positive *x*（即沒有¬ *x*）。又因為在 *S'* 中 *S'*(*x*) = 1，因此

凡有 x 的 clause evaluate 起來必定為 true。現在考慮剩餘（不包含 x）的 clause，這些 clause 必定和原 (D, P) 中的 clause 相同；另外，對於除了 x 以外的變數 $S'(v) = S(v)$，因此這些 clause evaluate 起來會和原先 $S(v)$ 代入 (D, P) 的結果相同，又原先 S 是 (D, P) 的一個 valid assignment，故這些剩下的 clause evaluate 結果也必定為 true。綜合以上，存在一個 $S'(x)$ 是 (M, P) 的一個 valid assignment，$M\langle x,+\rangle$ 在此情況為 alive。

- 對於 Mutant $M\langle x,-\rangle$ 的情形與 $M\langle x,+\rangle$ 雷同，唯在 S' 中是把 x 的取值都設為 0 (false)。M 在此情況也必為 alive，以下就不贅述。
- 考慮最後一種情況若 mutant 是 $M\langle x, \neg\rangle$，那麼同樣假設對於 pair (D, P) 的一個 valid assignment 是 S，那麼定義 S' 為：

$$S'(v) = \begin{cases} \neg S(v), & v = x \\ S(v), & otherwise \end{cases}$$

即在 S' 中 x 的取值恰與 S 中相反，對於其餘變數則不變。那麼可以發現，S' 會滿足 pair (M, P)。要證明這點，同樣先考慮有 x 的 clause。注意到由於 property clause 中都沒有 x，且 design clause 中 x 的 occurrence 都被 negate，因此 (M, P) 中的這些 clause evaluate 起來必定與原先一樣（為true）。剩下的 clause 則同之前所述，結果不變依然為 true。故對於 mutant $M\langle x,!\rangle$ 亦保證存在 assignment $S'(x)$ 會滿足 pair (M, P)。

由此可知，無論 M 的 mutant type 為何，只要mutated variable x 沒有在某 property P 中出現，那麼 mutant-priority pair (M, P) 必定為 satisfiable （且我們不需知道實際上 S 和 S' 是什麼！）。■

接下來注意到題目給的 Property 對於原 design 都是 satisfiable，對 mutant 則並不保證是 SAT。考慮到 satisfiable solution 通常利用 SAT-solver 可以比較快的得到（比起一個 UNSAT 的 CNF），我們或許可以先解出原 design property pair (D, P) 的一個 assignment S，來嘗試藉此得到其他 mutant 對於此同樣 property 的解。對此我們有以下結論：

Observation 2

對於一個 property P 與原 design D，令 pair (D, P) 的一組 valid assignment 為 S（需由 SAT engine 解得）。考慮對於一 mutant M：

1) 若 mutant 為 $M\langle x,+\rangle$ 且 $S(x) = 1$，則 pair (M, P) 必為 SAT。

2) 若 mutant 為 $M\langle x,-\rangle$ 且 $S(x) = 0$，則 pair (M, P) 必為 SAT。

Proof:

我們首先考慮第一種情形，即 mutant 為 $M\langle x,+\rangle$ 且 $S(x) = 1$。我們可以直接令 $S' = S$，則 S' 必定可以滿足 pair (M, P)，原因如下。考慮所有 property clause，由於 property clause 中的 x 是不會被改動到的（其他 variable 當然也不會），因此所有 (M, P) 中 property clause 用 S' 的 evaluate 結果必定與原先相同（為 true）。再考慮 mutant clause，其中 x 和 $\neg x$ 都被 x 所取代，而 $S'(x) = 1$，故任何有 x 的 mutant clause 都會 evaluate 為 true。其他 clause 結果不變（為 true）。故可知 $S'(x)$ 滿足 pair (M, P)，mutant M 在這情況下為 live。

第二種情形，即 mutant 為 $M\langle x,-\rangle$ 且 $S(x) = 0$，與第一種情形類似，唯 assignment S'(x) 改為 0，其他不變，在此就不贅述。

綜合以上，可得到結論對於 pair (D, P) 得到的 assignment S，若一 mutant 為 $M\langle x,+\rangle$ 且 $S(x) = 1$，或者 mutant 為 $M\langle x,-\rangle$ 且 $S(x) = 0$，則該 mutant 必為 live。■

稍加分析可知道，若在 assignment 中一個 variable 為 0/1 的機率為各半，而 mutant type 是 positive / negative / negate 的機率為各 1/3（即 mutant type 是隨機產生）的前提下，則利用這個結論得到的 cut 可以在約 (2/3) × (1/2) = 1/3 的情形都有效（可以利用 *S* 直接確認該個 mutant-property pair 為SAT且可以直接用 *S* 得到一組 assignment）。也就是說，利用這個 cut 大約可以預期將待解的 mutant-property set 縮減為 2/3。

最後還有一個並不是很有用（發生機率較低），但驗證的 cost 很小故還是值得一試的 cut：

Observation 3

對於 mutant-property pair (*M, P*)：

1) 若 mutant 為 $M\langle x,+\rangle$ 且 *P* 中所有 *x* 的 occurrence 都為 positive *x*，則 pair (*M, P*) 必為 SAT。
2) 若 mutant 為 $M\langle x,-\rangle$ 且 *P* 中所有 *x* 的 occurrence 都為 negative *x*（$\neg x$），則 pair (*M, P*) 必為 SAT。

Proof:

Trivial。只要分別 assign *x* 為 1 / 0 即可。■

以上為 cut 的部分，在真正大量用 SAT engine 開始解 CNF 之前，可以先用這些 observation 來去除許多不需要測試就可以知道結果的 mutant-property pair（詳見之後實作部分的 first-pass ~ third-pass），再開始對每個 pair 進行 SAT-solve。

但在 SAT-solve 部分，同樣有許多方法可以嘗試改進 SAT-solve 的速度，以下分別描述之：

Method 1: "trim"

首先是觀察到 property / design CNF 中常常會有一些 correlated variable，亦即有兩變數 u 和 v 滿足 $u = v$ 或 $u = \neg v$，此時若可以成功偵測到這樣的關係並且把他們事先處理掉（亦即把所有 v / $\neg v$ 的 occurrence 都換為 u / $\neg u$），理論上再進行 SAT-solve 會對於解 SAT 的速度有幫助，因為常理而言SAT的運行速度與變數數目有關。

當然「所有」這樣的關係未必可以簡單的從 CNF clauses中得到，我們處理的方式如下。因為 CNF clauses 中有許多少於兩個 variables 的 clause，一個 variable 的 clause 就相當於直接 designate 某個變數為 true / false，在除去這些 clause 之後，我們著重於想辦法用剩下恰有兩個 variable 的 clause（以下稱為 bi-clause）來得到兩個變數的關係。

一個 bi-clause $(u + v)$ 代表的其實是一個 imply 關係，即$\neg u \Rightarrow v$ 和 $\neg v \Rightarrow u$。考慮若同時有兩個 bi-clause $(u + v)$ 和 $(\neg u + \neg v)$，則有：$\neg u \Leftrightarrow v$ 且 $\neg v \Leftrightarrow u$，也就是 $v = \neg u$！（同理如果有兩個 bi-clause $(u + \neg v)$ 和 $(\neg u + v)$ 也同樣可以簡化）。因此若可以有效率的找出這樣的關聯 bi-clause 並得到變數間的 correlation，理論上就可以幫助我們縮解 SAT-solve 所需時間。

以下大略描述如何實作上述的作法。

Algorithm for method 1 (trim)

最簡單的方法就是對任兩個 clause 去 pairwisely 看他們是不是關聯 bi-clause，而更新時則每發現一個關聯 bi-clause (u, v) 就把所有 v 的 occurrence 都換為用 u 表示。但由於實際上的 clause 數和 variable 數都非常多，這樣的作法時間複雜度可以高達$O(|\text{clause}|^2$ +

|variable|^2)，因此需要一個比較有效率的作法。

首先第一步是先除去所有 single-variable clause，做法很簡單，一個 single-variable-clause (v) 就代表 v 的 assignment 必定要是 true，反之則是 false。在做完這步後，所有min-sized clause 都是 bi-clause。

接下來要找出關聯 clause，我們可以用一個 set 來記錄有出現過的 clause (u, v)，並對於每個 clause 去 set 裡面尋找他的關聯 clause ($\neg u$, $\neg v$)，若找到我們就可以 deduce relation v = $\neg u$。

此時我們需要更新所有 v / $\neg v$ 的 occurrence 為 u，但為了效率並不允許我們直接這樣做。因此這部分可用 disjoint set 的 union / find 技巧來達成。

以上整個流程進行一次稱為一次「trim iteration」。時間複雜度約為 O(N(lg$N$ + α))，其中 N 為 input size，即 N = O(|clause| + |variable|)，α 為 disjoint set 操作所帶的 inverse-Ackermann function，在任何合理 input size (小於10^{80})下都不超過 4，可視為極小常數。

值得注意的是因為經過一次之後「trim iteration」後，可能會出現更多的 single-variable-clause / bi-clause，因此我們可以不斷重複進行 trim iteration 直到 input CNF 無法再繼續簡化下去。

Method 2: Clause Learning with Gröbner Basis

有論文[1]指出，可以用 Gröbner Basis 來做 Clause Learning ，我們有試著在 minisat 中加入 Gröbner Basis 的演算法來取代部分的 Clause Learning。不過，我們所實作的 Gröbner Basis 的效率不足，並沒有縮短總執行時間。因此在最後的版本中，並沒有使用 Gröbner Basis 做 Clause Learning。

---

[1] **Christoph Zengler and Wolfgang Küchlin:**
Extending Clause Learning of SAT Solvers with Boolean Gröbner Bases

四、 實作

4.1 資料結構

在內部的儲存中，我們定義了幾個 class 來儲存CNF：

Clauses

是用來儲存所有的CNF的底層資料結構，利用二維陣列來儲存CNF。其中一個維度代表CNF中clause的個數，另一個維度則和單一clause中variable的個數有關。

Assignment

儲存每個變數及其所對應的值，用來承接SAT engine的輸出結果。

Mutant

用來儲存每個mutant的資訊。包含mutant variable以及mutant 方式等。另外也有維護一個property的列表，代表著尚未決定的mutant-property pair。在mutant資料結構中，我們並沒有儲存真正的CNF資訊，而是統一由model產生mutant的CNF。

Model

繼承自clauses，用來儲存原本的design。另外也提供函式產生mutant的CNF。

Property

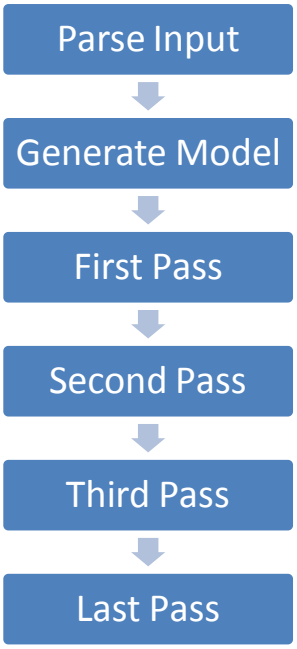
同樣繼承自clauses，儲存property的CNF資訊。若曾經有用SAT engine解過model + property，其assignment結果也會cache在此資料結構中。

此外，我們也有介面與SAT Engine溝通：

SatAgent

提供統一的API讓程式內部使用。無論使用哪一套SAT Engine，都可以只修改SatAgent內的adapter便可以使其運作。現行版本的SAT engine 為 minisat 2.2.0。

4.2 程式流程



Parse Input

在這個過程中，我們藉由讀入輸入的檔案，把 property 以自有的資料結構儲存。同時也會對 mutant 作處理，然而所有的 mutant 具有很大的相似性，因此不需要一一讀入儲存 mutant 的 CNF（如 desing2_lite 中的所有 mutant file 加起來可是有約 2GB!），只需將 mutant variable 以及 mutant type 記錄下來，之後由原 model CNF 和 mutant variable 和 type 即可得到 model CNF。

Generate Model

由於之後的動作會利用到原始 model 和 property 一定有 Boolean satisfiability 的特性，我們必須找出原始 model 的 CNF。透過兩個 mutant 的 CNF 我們可以反解找出原本的 model CNF。

First Pass

利用先前 **Observation 1** 的觀念，對於所有的 mutant-property pair，檢查是否有 mutant variable 不在 property 出現的情形。如果情形發

生，可以知道該 mutant-property 的 boolean satisfiability 必定滿足，便可將該property 從 mutant 的 property list 中移除。在檢查完成後，如果mutant 的 property list 沒有剩餘 property 時，則宣告該 mutant 為live。

Second Pass

利用 Observation 3 的觀念，若有 mutant type 是 positive 同時該 mutant variable 在 property 都是以 positive 的情況出現，或者 mutant type是negative同時該mutant variable在property都是以negative的情況出現時，則判定該mutant-property pair為Boolean satisfiable。

Third Pass

利用Observation 2的觀念，在判斷之前，須要先把與此mutant-property pair所對應的 (D, P) 用SAT engine找出一組解。倘若mutant variable在這組解的assignment是true，且mutant type是positive，則可判斷此mutant-property pair 一定是Boolean satisfiable。在mutant type是negative的情況亦類似，唯negate的情形在這裡沒有辦法判斷。

Last Pass

在這裡剩下的mutant-property pair必須真正利用SAT engine得知是否有Boolean satisfiability。在這裡可以利用先前提到的method 1與method 2來加速SAT engine的運行。不過，我們希望可以優先找到 satisfiable 的 mutant-property pair ，所以，在解各 mutant-property pair時，會加上 restart limit ，並慢慢增加 restart limit。

五、 實驗結果

5.1 工作平台及程式語言

我們以 C++ 程式語言撰寫，並以G++編譯程式編譯，運行在GNU/Linux作業系統中。

5.2 測試檔輸出

對於 design1 測試檔案，我們找到38組live mutant，和參考答案一樣。

對於 design2_lite 測試檔案，我們在一秒內找到195組 live mutant，兩分鐘內找到260組live mutant，一小時可以找到290組live mutant。

5.3 時間及記憶體使用量

以design2_lite為例，我們可以在不同的運行時間得到不同的live mutant組數，一般來說在五分鐘之內就能得到大部分的live mutant列表。剩下的mutant由於難度的關係，需要大量的時間才有機會解出來。

利用/proc/[pid]/status的資訊，我們可以找到process的記憶體使用量。整個程式執行中最大的記憶體使用量可以由Vmpeak參數得到，為512MB。

5.4 結果分析

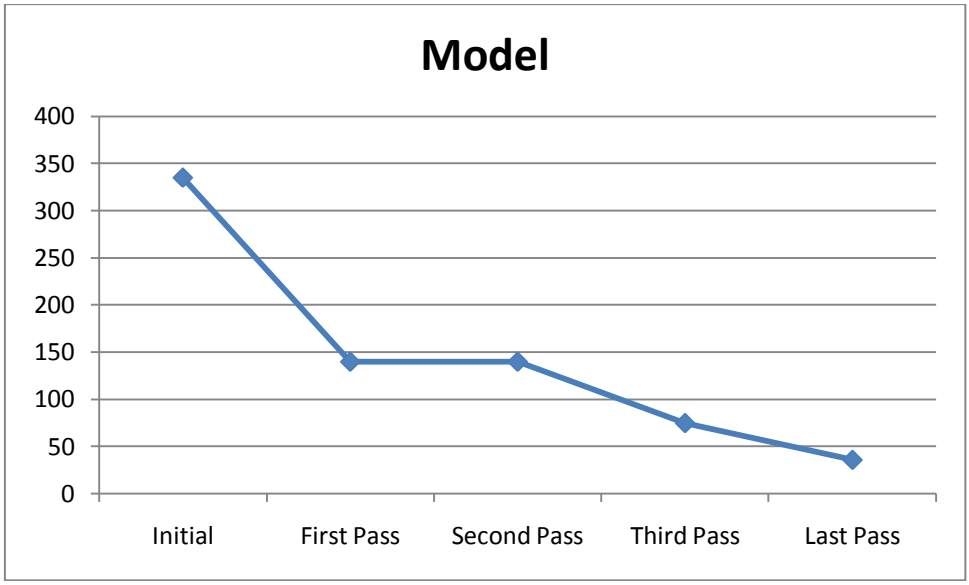


Figure 1

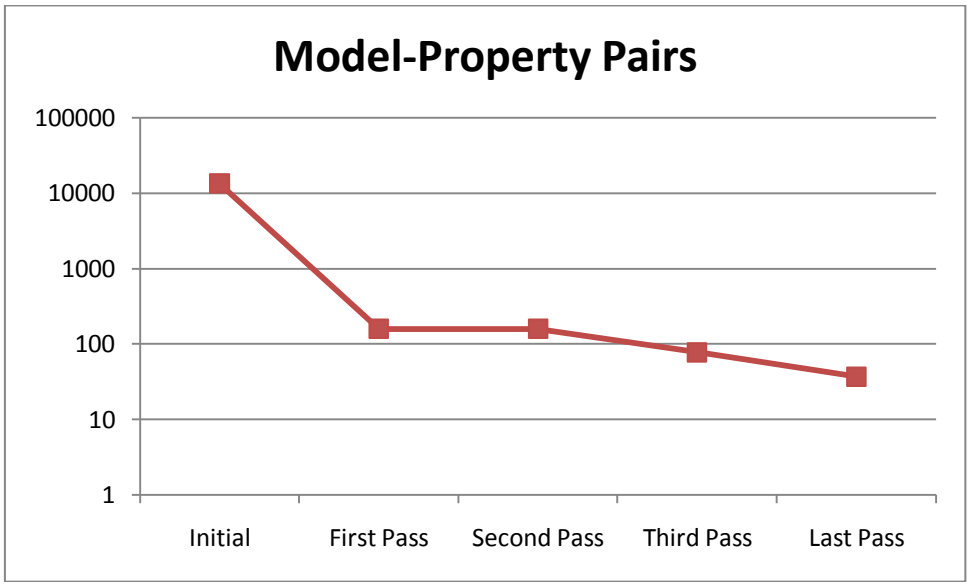


Figure 2

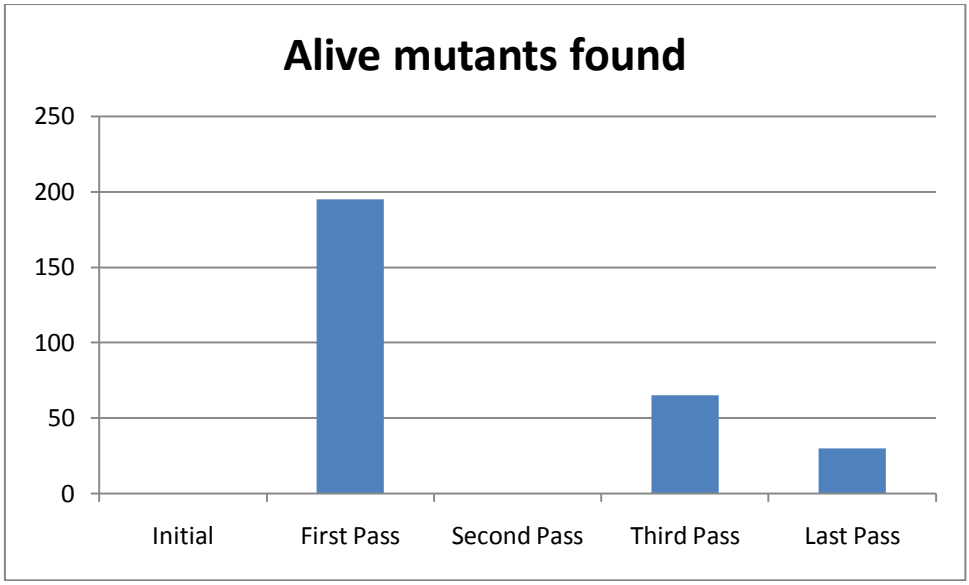


Figure 3

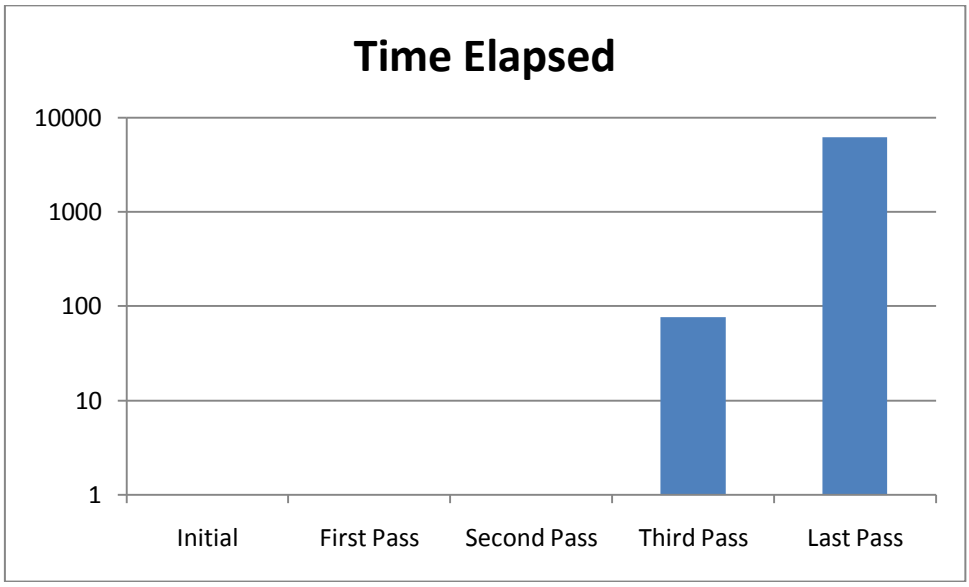


Figure 4

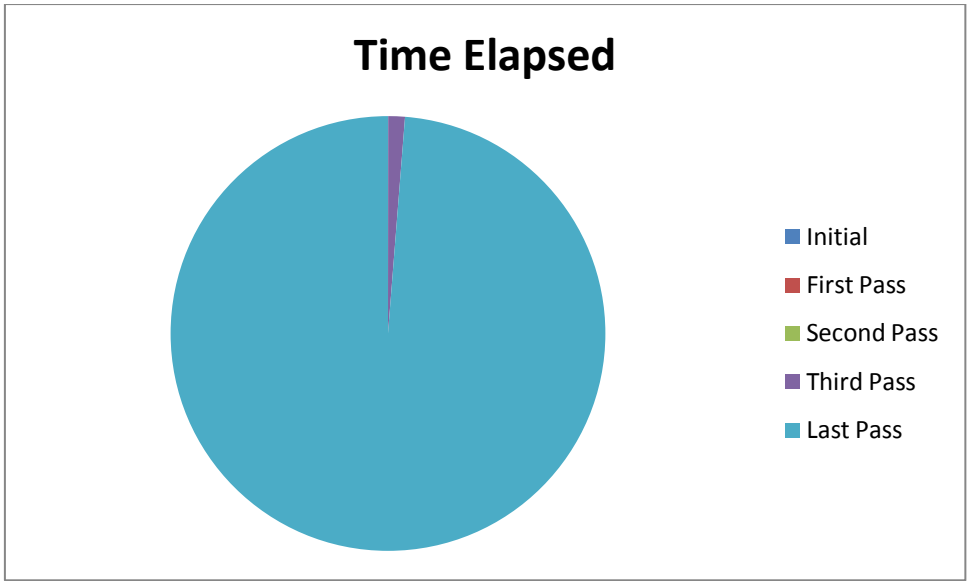


Figure 5

從 Figure 1 我們可以看到，在 First Pass 中，大部分 (195個) Mutant 都被證明是 alive ，剩下的 mutant 中，而且，所剩的大多是一對一的 mutant-property pair。

同時請注意到 Figure 2 的 y軸是 log scale，換句話說，mutant-property pair 的數量在 First Pass 中減少了兩個數量級。而剩下的 mutant 中，大約有一半會在 Third Pass 中被刪除（證明是 alive）。

在 Figure 4、Figure 5中，我們可以看到程式在不同的部分所花的時間。First Pass 和 Second Pass 都不用解 SAT ，所需時間是線性的。而 Third Pass 則需要解部分的 SAT ，所花的時間明顯的上升了。在 Last Pass 中，我們必須要解所有剩下的 mutant-property pair，所花的時間快速的上升，由 Figure 5 可以看到，幾乎所有的時間都花在 Last Pass 中。

六、 參考資料

- minisat 2.2.0: https://github.com/niklasso/minisat
- Extending Clause Learning of SAT Solvers with Boolean Gröbner Bases

附錄: 使用手冊

A. 如何編譯程式

Makefile 已經寫好並附在程式碼當中。先用

```
$ make cleanall
```

清除已存在的二進位檔。再執行

```
$ make
```

指令來進行編譯的動作，此動作會自動編譯所需要的SAT engine。

B. 如何執行程式

產生的執行檔為fpq，可以用下列指令格式執行：

```
$ ./fpq [mutant_file] [property_file] [output]
```