限制式函式層級白箱單元測試之測試案例產生器

A Constraint-Based Test Case Generator for Method-Level White-Box Unit Testing

張振鴻　　林迺衛

Cheng-Hung Chang and Nai-Wei Lin

國立中正大學 資訊工程學系

Email:cyber366454@gmail.com, naiwei@cs.ccu.edu.tw

摘要

本系統開發一個針對 Java 程式的限制式函式層級白箱測試案例產生器。限制式測試案例自動產生技術將測試案例產生問題制定為限制滿足問題。從軟體行為的敘述、軟體等價行為的分割、軟體測試覆蓋標準的滿足、到軟體等價行為所對應的限制滿足問題的敘述都將使用限制邏輯圖來表示及分析。系統先將 Java 程式原始碼轉變成限制邏輯圖，精簡的表示函式行為的限制邏輯。搭配測試覆蓋標準可以有效地將軟體行為進行等價分割。最後利用限制邏輯程式的限制求解能力，將每類等價行為所對應的限制邏輯求解後產生測試輸入，及依規格文件所對應的限制邏輯求解後產生預期輸出，最後再將測試輸入與預期輸出轉換為 Java 測試函式。

關鍵詞：限制式測試案例產生器、函式層級白箱單元測試、限制邏輯圖、限制邏輯程式

一、 簡介

軟體測試是確保軟體品質最主要的方法。軟體測試的測試案例產生技術可以分成兩類：黑箱測試(Black-box testing)及白箱測試(White-box testing)，黑箱測試根據軟體的規格來產生測試案例；白箱測試則根據已實作出來的程式產生測試案例，這兩種測試案例產生技術相輔相成。

軟體單元測試是對於軟體中最小的元件進行測試。單元測試又可以分成函式層級(method-level)與類別層級(class-level)。在函式層級中單元測試的最小元件就是函式，著重在函式執行前後之狀態驗證；類別層級的單元測試則是將類別視為最小的單元並對類別的行為進行驗證。

限制式(constraint-based)測試案例自動產生技術是一種重要的測試案例自動產生技術[3]。限制式測試案例自動產生技術將測試案例產生問題制定為限制滿足問題(constraint satisfaction problem)。從軟體行為規格的敘述、軟體等價行為的分割、軟體測試覆蓋標準的滿足、到軟體等價行為所對應的限制滿足問題的敘述會使用到限制邏輯圖和限制邏輯程式。

根據 Arnaud Gotlieb 等人提出結構性測試(白箱測試)[5]被廣泛使用在軟體的元件測試(unit-testing)，白箱測試需要 1.定義好各測試組別指令的集合，其中包括測試涵蓋標準。2.計算各組測試資料的路徑。在自動生成測試案例前，我們需要將原始程式碼轉換成限制邏輯式，爾後對這些限制邏輯式進行找解，確認是否有一條合適的控制流程路徑存在，並產生對應到的測試案例。

軟體測試要確保驗證所有的程式行為都為正確。但是實際上是不可行的，正常情況下軟體的行為數量非常龐大，假使我們須要全部測試完畢，那資源的耗費一定相當龐大。在有限的資源為前提，要最合適的測試，會面臨四個挑戰，第一個挑戰：要如何將極大量的程式

行為化分成數組等價類別。第二個挑戰：要如何使用測試覆蓋標準，畫分出合理數量的等價類別。第三個挑戰：要如何從這些等價類別的代表測試案例產生相對應的測試輸入。第四個挑戰：要如何產生與測試輸入相對應之預期輸出。

在下一節，我們會對整個系統做一個整體介紹，與大致介紹各個部分，

二、系統架構

本系統將會以白箱測試的方式、函式層級的角度對 Java 程式原始碼進行分析並自動產生測試案例。

本系統將會分成幾個組件：

- → 限制邏輯圖建構器 (CLG constructor)
- → 路徑條列器(Path enumerator)
- → 測試輸入產生器
- → 測試腳本產生器

本論文將會把系統整體行為切分為四大部分：

1. 模擬程式碼之行為並產生對應之行為限制邏輯圖。
2. 從限制邏輯圖條列出可實行路徑。
3. 產生路徑對應之測試輸入及利用程式規格產生預期輸出。
4. 整理測試輸入及預期輸出並產生測試腳本。

系統流程架構如圖 1 所示。

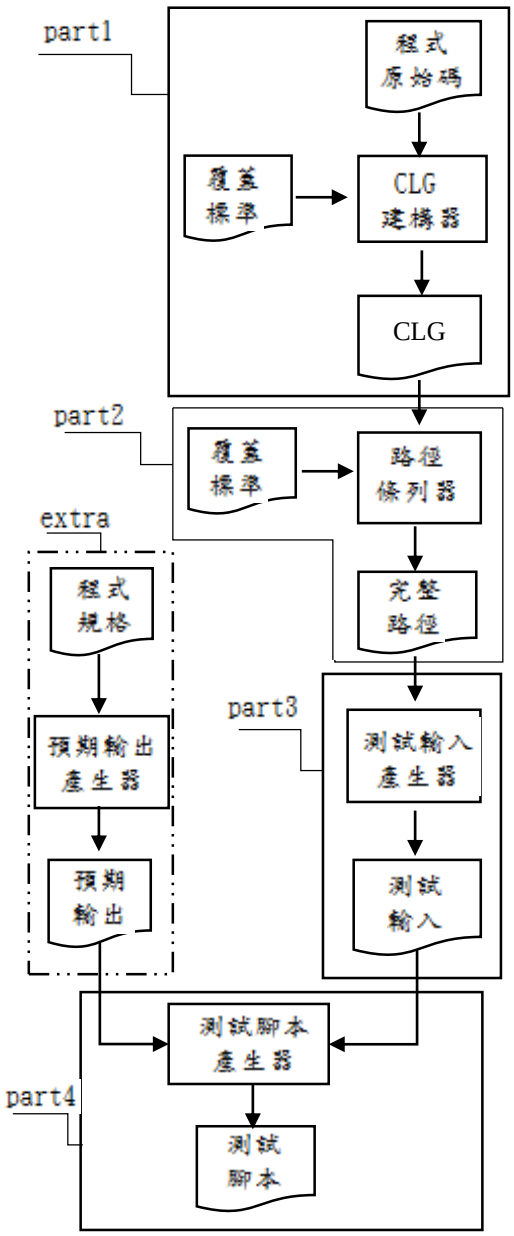


圖 1 系統架構

第一部分，會將輸入的 Java 程式原始碼進行分析，並利用設定之測試覆蓋標準建構成對應到的限制邏輯圖，以限制邏輯圖表示受測函式的程式行為。這部分將會以系統架構圖中 part1 來實現。

第二部分，一個完整的限制邏輯圖路徑就代表一個完整的程式行為，透由限制邏輯圖，將原本是無限組合的析取正規式(disjunctive normal form，DNF)的受測函式程式行為，變

成可數個數的受測函式的程式行為，我們將這些程式行為分割成一個可以被管理的等價類(equivalence classes) 集合。而系統架構中part.2 為根據限制邏輯圖，透過路徑條列器產生可實行路徑。由於各個等價類別內所有的測試案例找出錯誤的能力都相同，因此只要從每一個等價類別各挑出一個代表的測試案例，即可產生必要數量的測試案例。而必要數量的訂定會利用測試覆蓋標準(coverage criteria)做設定，進而得到最合適之測試路徑數量。

第三部分，因為每一條完整路徑都是可以視為是限制式的集合，可實行的路徑利用等價類的特性轉換出等價的限制邏輯程式(constraint logic programming - CLP)。將轉換出的限制邏輯程式輸入 part3 的測試輸入產生器，會經由限制邏輯程式的找解能力，找出各完整路徑所對應到之測試輸入。

在系統架構 extra 這邊代表是本團隊已有完成之基於限制式函式層級黑箱測試案例產生器[2, 8, 9]，會利用使用統一塑模語言(Unified Modeling Language，UML)[7]的類別圖(class diagram)與物件限制語言(Object Constraint Language，OCL)[6]作為軟體的規格，利用規格產生預期輸出。

三、限制邏輯圖建構

走訪程式碼前會有三個步驟

1.讀取檔案路徑

2.讀取檔案原始碼

3.將程式原始碼切割成抽象語法樹(Abstract Syntax Tree)。

對於抽象語法樹，這邊使用深度搜尋瀏覽(Depth-First Traversal)進行走訪各程式點，當在走訪抽象語法樹時，會同時進行限制邏輯圖(Constraint Logic Graph)的構成，在構成限制邏輯圖時會參考輸入之測試覆蓋標準(coverage criteria)，會因測試覆蓋標準的不同，產生出不同的結果。

對於抽象語法樹的轉換，我們會轉換成邊與結點的有向圖，節點主要分為四種，其一為限制節點(Constraint Node)，圖形為長方形此種節點是唯一含有限制式的抽象語法樹(Abstract Syntax Tree，AST) 的節點，在限制節點中的限制式都一定是布林式的運算式，且只有以下幾種可能：

1. 有關係運算的運算式
2. 真/假(True/False)的字符
3. 布林運算式或變數
4. 非(！)的運算式

連接用的連接節點(Connection Node)，是限制邏輯圖中唯一會連接到分支的點，圖形為菱形；限制邏輯圖的起始點，圖形為實心圓、結束點(End Node)，限制邏輯圖的結束點，圖形為實心同心圓。

利用以上的規則，我們可以將走訪後的逐漸產生限制邏輯圖，從起始的開始節點，每當碰上運算式會加上一個限制節點，碰上判斷式分支及迴圈執行條件將會添上連結節點，抵達程式結束點會以結束結點作結尾，整張限制邏輯圖是以有向箭頭繪出。

而在產生限制邏輯圖之前，會利用之前所設定的測試覆蓋標準，對各判斷進行合適的覆蓋標準轉換，由選擇的覆蓋標準評估是否已經產生足夠數量的可視完整路徑，我們提供了兩種覆蓋標準，一為 All Decision / Condition Coverage Criterion (AD/CC)，另一為 Multiple Condition Coverage Criterion(MCC)，兩個覆蓋標準產生的限制邏輯圖並不同，所以造成在對應到結構模型覆蓋 (structural model coverage)中的控制流程為主的覆蓋標準(control-flow oriented coverage criteria)的能力會有所不同，故以兩個覆蓋標準的能力作為命名。我們會在以下展現兩個覆蓋標準產生的限制邏輯圖，並且以會產生分支的運算式分開做討論。

碰上了 or 運算式我們會以測試覆蓋標準所定

義之決策點及條件點之真假值組合，進行足以分辨之限制邏輯圖，接著會展示 or 運算式在不同測試覆蓋標準下，限制邏輯圖的差別。

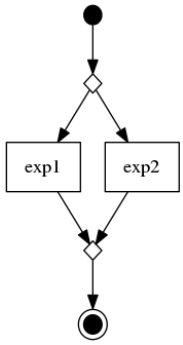
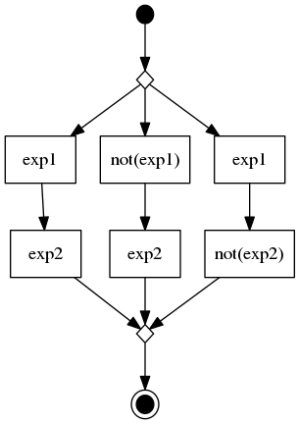
<table>
<tr><td colspan="2">exp1 || exp2</td></tr>
<tr><td>限制邏輯圖
(AD/CC)</td><td>限制邏輯圖(MCC)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

圖 2 or 運算式

兩種覆蓋標準產生的限制邏輯圖不一樣，其實就是對於 or 產生的限制邏輯圖的要求與想法不一樣，以 AD/CC 的觀點來看，運算式 A or 運算式 B，即是運算式 A 或者運算式 B 至少其中一個為真，or 運算式就為真，故根據限制邏輯圖的角度來看，可將運算式 A 與運算式 B 視為分開的兩條路徑上的節點，若運算式 A 與運算式 B 都為假，那不管走哪一條路徑都會失敗。而以 MCC 的觀點來說，上面那張圖的情況不能產生(運算式 A or 運算式 B)為真的全部情況，MCC 希望可以將全部的情況都展現出來。

接著以圖 3 、圖 4 Triangle Class 的 Category() method 進行限制邏輯圖的生成，測試覆蓋標準會使用 AD/CC。會產生如圖 5 的完整限制邏輯圖。

```
public Triangle{
   private int a,b,c;
   public Triangle(int sa,int sb,int sc);
   public String category();
}
```

圖 3 Triangle class

```
public String category(){
        String tc;
        if(a==b){
                if(b==c){tc="Equilateral";}
                else{tc=Isoscale;}
        }else{
                if(a==c){tc=Isoscale;}
                else{
                   if(b==c){tc=Isoscale;}
                   else {Scalence;}
                }
        }
        return tc;
}
```

圖 4 Triangle::Category()

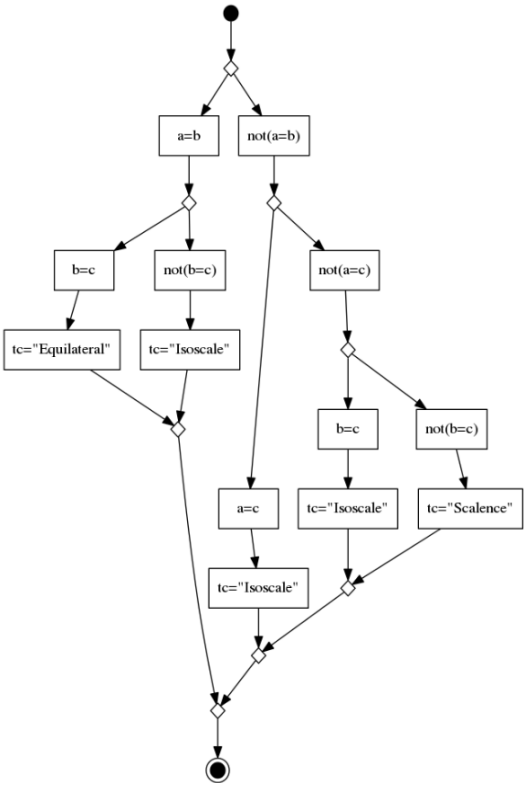


圖 5 Triangle::Category()的 CLG

四、函式等價行為分割

爾後，程式碼分析之後的抽象語法樹走訪完畢後，會得到一組完整的限制邏輯圖，得到的限制邏輯圖會進行路徑切割，將各路徑切割成各別的限制式集合。

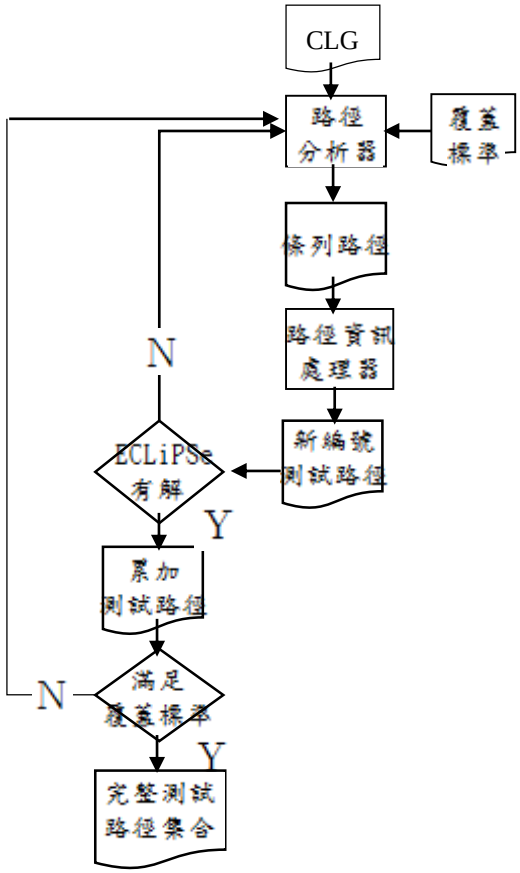


圖 6 part3 內部結構

使用路徑產生器(圖 6)分割的同時，會使用覆蓋標準來確保需要測試的路徑的組合，會採用廣度優先瀏覽(Breadth First Traversal)的演算法找尋與產生路徑，故會優先選擇還未被覆蓋到的路徑。在路徑產生器使用到的覆蓋標準會查看是否將限制邏輯圖的邊都覆蓋過一次，稱之為 All Branch Coverage Criterion。

挑出一條完整的路徑後，再轉換之前仍須做一些處理：由於限制邏輯程式的每個變數都只能有一個值，我們將此路徑上的相同變數收集起來，再次給予各個變數編號。這些限制邏輯程式就是各測試路徑將對應到的限制邏輯程式。每條路徑都是限制式的集合，一條完整的路徑並不一定代表這路徑一定能夠產生測試案例，所以必須辨別出是否為可實行路徑。需要經過可實行路徑(feasible path)的分析，判斷是否為可實行路徑。最後再使用限制邏輯程式找解器(constraint logic programming solver)的 ECLiPSe，若能找到符合限制邏輯程式的限制式的測試資料，就代表這條路徑是可實行路徑。當分析出的路徑以滿足測試覆蓋標準或是又碰到被找尋過之不可實行路徑，這時將視為滿足測試覆蓋標準。

圖 7 是從圖 5 萃取出之一條完整路徑，確認該完整路徑是可實行路徑後，會將該路徑轉變成限制邏輯程式(圖 8、圖 9)，當我們對限制邏輯找解程式(ECLiPSe)做查詢：

輸入

testTriangleCategory(S_pre,Arg,Result,S_post).

將會得到

S_pre=[1,1,1],Arg=[],

Result“Equilateral”,S_post=[1,1,1]

這邊的 S_pre=[1,1,1]和 Arg=[]就是我們要找的測試輸入。最後測試案例的產生將會利用這組測試輸入做為一組代表。

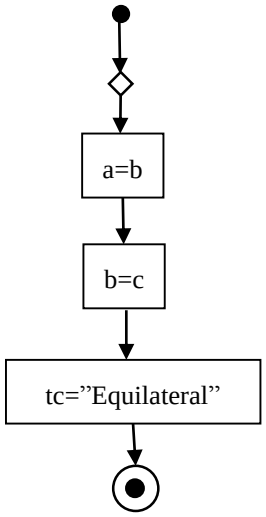


圖 7 Triangle::Category()的第一條完整路徑

```
triangleCategory([A_pre,B_pre,C_pre],[],Result,[A,B,C]):-
        %Constrints
        A_pre #= B_pre, B_pre #= C_pre,
        Tc = “Equilateral”, Result = Tc,
        %State Updates
        A #= A_pre, B #= B_pre, C #= C_pre.
```

圖 8 Triangle::Category()第一路徑的限制邏輯

程式

```
%include constraint solving library
:- lib(ic).
testTriangleCategory(S_pre,Arg,Result,S_post):-
        %Domains of variables
        [A_pre, B_pre, C_pre] :: -32768..32767,
        [A, B, C] :: -32768..32767
        %Constraints on variables
        S_pre = [A_pre, B_pre, C_pre] ,
        S_post = [A, B, C] ,
        triangleCategory(S_pre,Arg,Result,S_post),
        %Solving constraints
        labeling(A_pre, B_pre, C_pre, A, B, C).
```

圖 9 Triangle::Category()第一路徑的限制邏輯程式找解程式內容

五、測試案例產生

可實行路徑之限制邏輯程式代表這是函式某一路徑行為，這時使用限制邏輯程式找解器便可以替我們找出符合該測試路徑上所有限制式之測試資料，這些測試資料便是所求之測試輸入。

而預期輸出的產生，會使用本團隊過去所研發之限制式黑箱測試案例產生器產生，會利用規格文件進行分析，將無法執行的(non-Executable)規格文件轉變成可執行的(Executable)規格文件，進而產生出程式規格所描述之行為對應之預期輸出，最後我們會將所找出由程式內容產生之測試輸入及規格文件產生之預期輸出進行測試案例腳本的產生。

這邊規格文件是使用 OCL/UML 來訂定軟體規格，以同樣 Triangle::Category()做為例子。如圖 10，此為 Triangle::Category() 的 OCL 規格，在這訂定函式的行為。再交由限制式黑箱測試案例產生器，得到可執行的規格文件(圖 11、圖 12)。這邊可執行規格文件會用 CLP 來表示。

```
context Triangle::category() : String
post:
    if  sideA@pre = sideB@pre then
        if sideB@pre = sideC@pre then
            result = 'Equilateral'
        else
            result = 'Isosceles'
        endif
    else
        if sideA@pre = sideC@pre then
            result = 'Isosceles'
        else
            if sideB@pre=sideC@pre then
                result = 'Isosceles'
            else
                result = 'Scalene'
            endif
        endif
    endif
```

圖 10 Triangle::Category() 的 OCL 規格文件

```
triangleCategory([A_pre,B_pre,C_pre],[],Result,[A,B,C]):-
        %Constrints
        A_pre #= B_pre, B_pre #= C_pre,
        Tc = “Equilateral”, Result = Tc,
        %State Updates
        A #= A_pre, B #= B_pre, C #= C_pre.
```

圖 11 Triangle::Category() 的可執行規格文件

```
%include constraint solving library
:- lib(ic).
testTriangleCategory(S_pre,Arg,Result,S_post):-
        %Domains of variables
        [A_pre, B_pre, C_pre] :: -32768..32767,
        [A, B, C] :: -32768..32767
        %Constraints on variables
        S_pre = [A_pre, B_pre, C_pre] ,
        S_post = [A, B, C] ,
        triangleCategory(S_pre,Arg,Result,S_post),
        %Solving constraints
        labeling(A_pre, B_pre, C_pre, A, B, C).
```

圖 12 Triangle::Category() 的可執行規格文件找解程式內容

轉變成可執行的規格文件，我們一樣會使用限制邏輯找解器(ECLiPSe)進行找解

輸入

testTriangleCategory(S_pre,Arg,Result,S_post).

將會得到

S_pre=[1,1,1],Arg=[],

Result“Equilateral”,S_post=[1,1,1]

這邊的 Result“Equilateral”,S_post=[1,1,1] 就是我們對應到規格文件之預期輸出。

最後我們將測試輸入及預期輸出做統整，進行完整測試案例的產生，最終可以產生如圖 13 之測試案例。

```
public void testCategory(){
@Test
Triangle obj1 = new Triangle(1,1,1);
assertEquals(“Equilateral”, obj1.category());
@Test
Triangle obj1 = new Triangle(2,2,1);
assertEquals(“Isosceles”, obj1.category());
@Test
Triangle obj1 = new Triangle(2,1,2);
assertEquals(“Isosceles”, obj1.category());
@Test
Triangle obj1 = new Triangle(1,2,2);
assertEquals(“Isosceles”, obj1.category());
@Test
Triangle obj1 = new Triangle(2,3,4);
assertEquals(“Scalene”, obj1.category());
}
```

圖 13 category() 完整測試案例

六、結論

本論文實作基於限制式函式層級白箱測試案例產生器。利用基於限制式方式(constraint-based approach)將自動測試案例產生問題簡化為限制滿足問題(constraint satisfaction problem)，從 Java 程式原始碼，將程式行為畫分成多個等價類別。藉由測試覆蓋標準分析成合適的限制邏輯圖，再從限制邏輯圖轉換成限制邏輯運算式之完整路徑，測試輸入將會使用限制式完整路徑產生的限制邏輯程式求得。藉由自動化的白箱測試，減少人工測試的偏差及誤判，增加測試的可信度和標準性。

七、相關技術與研究

限制式的測試案例產生研究

基於限制式的測試(Constraint-Based Testing - CBT)被廣泛使用到各種不同的測試技術與方法中。DeMillo 和 Offutt 是將基於限制式的研究用在測試上的先驅[3]，他們提出了基於限制式測試，自動產生測試輸入用在基於錯誤的測試(fault-based testing)

A. Gotlieb, B. Botella 和 M. Rueher 提出了第一個從 C 程式轉換程靜態單賦值形式(static single assignment form)[4]，所有變數都只會被賦予最多一次的值，接著可以透過這個方式有系統的取得控制流程圖中每一條不同執行路徑的限制式，進行符號執行(symbolic execution)。最後，透過限制找解器(constraint solvers)自動產生所有可實行路徑的測試輸入。

參考文獻

[1] K. Apt and M. Wallace, The ECLiPSe Constraint Programming System, Available: http://eclipseclp.org/.

[2] C.-K. Chang and N.-W. Lin, “A Constraint-Based Framework for Test Case Generation in Method-Level Black-Box Unit Testing,” Journal of Information Science and Engineering, 32(2), pp.365-387, 2016.

[3] R. A. DeMillo and A. J. Offutt, “Constraint-Based Automatic Test Data Generation,” IEEE Transactions on Software Engineering, 17(9), pp. 900-910, 1991.

[4] A. Gotlieb, B. Botella and M. Rueher, “Automatic Test Data Generation Using Constraint Solving Techniques,” Proceedings of the 1998 ACM Symposium on Software Testing and Analysis, pp. 53-62, 1998.

[5] A. Gotlieb, B. Botella and M. Rueher, “A CLP Framework for Computing Structural Test Data,” Proceedings of First International Conference on

Computational Logic, pp, 399-413, 2000.

[6] O. M. Group, Object Constraint Language Specification, Version 2.0, 2006.

[7] O. M. Group, OMG Unified Modeling Language, Version 2.5, 2012.

[8] C.-Y. Huang, C.-K. Chang and N.-W. Lin, “Test Case Generation Based on Constraint Logic Graph,” proceedings of Taiwan Conference on Software Engineering, 2015.

[9] Y.-T. Lan C.-K. Chang and N.-W. Lin, “Automatic Coversion from UML/OCL Specification to CLP Specification”. proceedings of Taiwan Conference on Software Engineering, 2015.