支援字串及群集資料型態的限制式測試案例產生
Supporting String and Collection Data Types in Constraint-Based Test Case Generation

郭俊毅 林迺衛
國立中正大學 資訊工程學系
Jun-Yi Guo and Nai-Wei Lin
Email: a22890710@gmail.com, naiwei@cs.ccu.edu.tw

摘 要

我們在先前的研究中，已經開發一個支援整數型態的限制式函式層級黑箱單元測試的測試案例產生器。該黑箱測試案例產生器將使用類別圖與物件限制語言描述的不可執行的規格轉換成使用限制邏輯語言描述的可執行的規格，並透過執行限制邏輯語言程式自動產生測試案例。由於限制邏輯語言本身對非整數型態的限制求解支援度不高，該黑箱測試案例產生器對非整數型態的支援度也不高。本論文為了可以支援字串及群集資料型態的測試案例產生，有效地擴充限制邏輯語言對字串與群集資料型態的限制求解支援度。本論文並針對字串及群集資料型態的測試案例產生，做了一個初步的系統評估。

一、簡介

軟體開發中會透過各種不同的設計與開發階段，然而其過程中可能會發生各種非預期或是發生規格上的不符合需求而需要更改，而越晚發現錯誤或規格上的不符，則在修正期間所需花費的成本也會高，透過早期根據需求去制定相對應的規格文件，將需求轉換成規格文件可以提早發現不符合需求或是實際實作之後可能會不符合規格的部分，且透過規格文件去實作相對應的測試案例，讓工程師專注於各個元件的設計與程式的撰寫，同時透過測試案例去驗證實作的元件是否正確，可更早發現實作上的錯誤並去修正，減少後期才發現錯誤的可能性。其中常用於表達規格文件的為統一塑模語言[1]與物件限制語言[2]，這兩個種類。

統一塑模語言是目前被廣泛接收的視覺化模型語言，開發人員可以透過不同種類的圖去描述一個軟體的動態與靜態行為，也可描述各個類別之間的關係。同時也能表達所有類別中所包含的屬性跟其形態與裡面有何種可以執行的行為。透過其表達的屬性型態去找出該實例化何種型態的資料去對應到測試案例，但類別圖只能表現系統的靜態規格，並不能描述動態規格與對其產生複雜的限制，這部份則透過物件限制語言去彌補這部分的不足。

物件限制語言是一個以狀態為基礎的規格語言，在物件限制語言中可以透過定義類別恆定的條件、前置條件與後置條件去限制物件所需符合的狀態或定義，同時也能表達物件在其過程中的狀態是發生了何種的變化。恆定條件是用來限制與表達物件在整的生命週期中必定符合的限制條件，同時也表達出物件可能的狀態。前置條件則是用來限制與表達物件執行方法前所需滿足的條件與狀態，後置條件則是用來定義與表達物件執行後會變成如何的狀態。

透過之前的研究[3][4][5]，已開發出可將物件限制語言配合統一塑模語言轉換成可執行的限制邏輯語言的規格，而透過實際的執行可產生出相對應的測試案例，但對於可以產生的測試案例的型態卻是有相當的限制存在，其屬性的型態自動產生的測試案例中只能支援到整數的部分，對其他型態的部分並不能自行透過執行邏輯語言去自動產生，且在其中物件限制語言轉換成邏輯語言在非整數的情況下也並沒有完全的實作與實現。

邏輯語言是採用一階邏輯的行式，可用來定義項目之間的關係，程式會自動進行匹配與回溯來找出符合問題的答案。現在的限制邏輯語言系統都至少有支援在有限值域找解的部分，但對其自行提供的實例化資料的型態卻有相當的限制，必須透過另外組合才能做出求得其他不同型態資料，而找解的方法主要可分為由上而下的搜尋與分支範圍搜尋。在執行限制邏輯語言程式中，Prolog 會透過函示庫中尋找先建立好的事實與規則來找出符合的結果。本研究會使用 ECRC 所開發的 ECLiPSe[6]做為限制邏輯程式的執行平台。

限制邏輯程式的限制與使用上有一定程度的複雜與限制，造成其可能會在產生相同的結果的限制邏輯語言中可能會有不同所需時間上的執行差異，其可能差距高達十倍以上都有可能，而我們使用的 ECLiPSe 中對字串、群集的實例化與限制支援上並不完善，且在 ECLiPSe 中會形成實例化後去確認是否正確，而不是在做限制後去實例化，從而浪費更多時間的可能性更是相當的多。

本研究中主要是希望設計出透過限制邏輯語言有以下的用途。透過限制邏輯語言的執行可以在並沒有明確的輸出與輸入的情況下，ECLiPSe 可以幫我們找出所有符合規格中描述的字串、群集的輸入與輸出，而每一組的測試輸入與輸出皆可以拿來當作測試資料。此外，可以更快速地透過限制找出我們所需要的資料或是表示找不到相應的解答，並且透過限制排除不可能的路徑減少需要實例化的次數。

本研究中,我們會以如圖 1 的架構產生我們所需要的測試案例,我們會先將物件限制語言與統一塑模語言透過[5]找出符合測試路徑，之後透過[4]將限制路徑上的限制敘述轉換成可執行的限制邏輯程式,之後並透過執行限制邏輯程式產生符合限制的測試資料,但原本只能對整數型態有完善的支援，因此我們會對[4]進行擴充的方式讓轉換成的限制邏輯程式可以支援到字串與群集等更多的資料型態的測試案例產生。

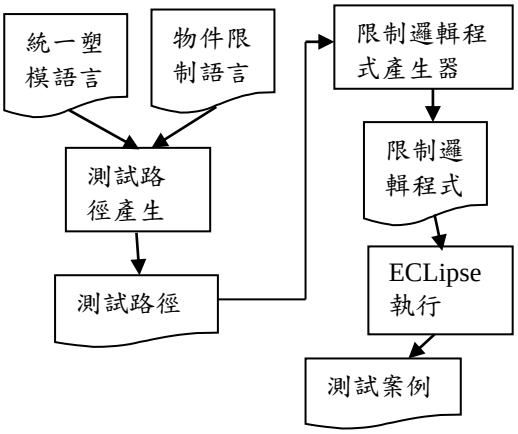

圖 1 整體流程

二、限制邏輯程式

限制邏輯程式中主要是由限制敘述所組成,限制邏輯程式同時也是限制滿足問題,我們會透過執行限制邏輯程式的方式找到符合所有限制的測試案例，如圖 2 為例，我們希望限制邏輯式找出大於 5 的整數來做測試資料，因此我們會以限制式的方式對其結果做會大於 5 的限制，並透過 ECLiPSe 提供的實例化方式，即 labeling/1 來幫我們把未知變數從一個值域變成一個特定的數值供我們之後做測試案例的資料使用,而我們會以此種的方式產生符合的測試案例。

```
test(A):-
A#>5,
labeling([A]).
```

圖 2 限制邏輯程式

本研究中使用 ECLiPSe 執行限制邏輯程式時是採用深度優先搜尋與回溯找出符合的解答,在限制邏輯程式中會因為 ECLiPSe 執行限制邏輯程式的性質而造成花費更多時間,造成不必要的實例化或是必須以別種方法才能實例化所需資料等的特性,而我們希望的執行流程也有可能與實際上並不相符合。

ECLiPSe 在執行限制邏輯程式中我們會希望以如圖 2 的流程，限制後在實例化的方式，快速的實例化出符合的測試資料,然而實際上在執行時很有可能因某種原因造成不斷的回溯重複去檢測實例化是否正確，以圖 3 是其中一例的限制邏輯程式，因 ECLiPSe 提供的 mod/3 中被除數與除數需要是實際的值才能正確執行，所以我們會透過 delay 的方式到被除數與除數有數值實才會執行 mod 的功能，因此我們會先實例 A 跟 B 為 1 的值後不斷實例化 C，因此造成了 10 次的回溯，此種情況在 ECLiPSe 中的串列跟字串也是不缺乏的情況，也是相當正常的情形，因此我們也可能在某些情況如圖 4 所示的方式,在特定情況下做上相應的限制,跟圖 3 個例子相比在只有正整數的情況會減少相應次數的回溯,而在字串與串列變數更多的情況上其情形更可能因此倍增。

```
test(A,B,C):-
[A,B,C]::1..10,
modTest(A,B,C),
labeling([A,B,C]).

delay modTest(A,B,_) if var(A);var(B).
modTest(A,B,C):-
mod(A,B,C).
```

圖 3 限制邏輯程式 多次實例化回溯

```
test(A,B,C):-
[A,B,C]::1..10,
modTest2(A,B,C),
labeling([A,B,C]).

modTest2(A,B,C):-
      (get_min(A)>=0,get_min(B)>=0->A#>=
      C,B#>C;true),modTest(A,B,C).
delay modTest(A,B,_) if var(A);var(B).
modTest(A,B,C):-
```

圖 4 限制方法的改變

其中也有 ECLiPSe 執行限制邏輯程式中的性質有關，如變數的性質、串列的性質、函式上的限制。各的性質上同時對限制與產生字串與群集可以說是造成一定的難度。

變數的性質中，主要為變數是數學邏輯上的未知數,只要確認該未知數已經沒有符合的情形就會發生回溯去找另快可能形成解的路徑,而區間限制時除特別去給予的情形,每個區間中只包含整數的部分,且在字串未知狀況下並不能做出字串中特定字元的或長度的限制。

串列的性質主要也是受變數的性質影響，對串列的大小與串列中特定的位置資料也屬於變數的性質在，而受到其影響，串列中也不易直接做出大小的限制,因此可能以加入長度變數或是在其他部分做上限制。

而函式的限制則是 ECLiPSe 中提供的函式本身的限制,在某些函式中必須要知道某些值在被給的情況下才能處理，一般我們都會用 delay 的方法去延遲,但某些情況下還是可以對其做上限制且不必等到全部都實例化完才能做，圖 3 中 mod 可以以圖 4 的方式多加上限制可能。並 ECLiPSe 對實例化產生字串與小數等並沒有直接提供。

三、字串資料型態的支援

在[4]中，已實作了將類別圖與物件限制語言的轉換，同時也對物件的關聯也做了相應的實作，

但對於實例化與限制字串並無太多的實作，其中在[7]中也有關於字串限制的相關方法，但對於如何產生與實作並無太多的實例。為了能夠更快速產生相對應的型態與資料，與更容易在限制邏輯語言下去實例化與做限制而將字串定義成別種的型態，且對各種函式的限制與實作盡量以優先做好限制再去實例化的方式實作，而不是以實例化後再去執行限制敘述是否符合，同時也對在未知明確的輸入與輸出下實例化出符合的字串。

甲、字串定義的格式

在限制邏輯語言中雖然也有定義好的字串格式存在，但對於直接透過裡面定義好的字串直接使用反而會造成不便與優先限制上的困難，因在限制邏輯語言中的變數是數學邏輯上的未知數，且直接使用裡面定義的字串不能通過限制的方式使未知的字串限制在一定的範圍內去實例化，反而造成實例化後再確認是否正確，故我們以表 1 的範例來表示字串。

表 1 字串的表式轉換

定義的形式	表示的字串
[5, 65, 65, 65, 65, 65]	“AAAAA”
[Length,A,A,A,A,A|B]	表示長度大於等於 5 的字串並且前 1 到 5 個字相同。
[2,65,_]	開頭是 A 與長度 2

如表 1，本研究中將字串定義成類似字元串列的格式，第一個位置代表的是該字串的長度，可以透過其去限制長度去決定實例化後的字串長度，進而對於各種規格與限制去更好實行。另後面的字元串列中字元則是以 ASCII 表示，透過 ASCII 轉換成整數的形式，而讓其代表的字元在限制邏輯程式中更好的對單一的字元做限制，也對於字串的實例化也更有幫助。

在限制邏輯語言中我們會透過圖 5 的函式去做格式的確立與字元的限制，會在其正確得知字串長度的情況下去實際建立成固定的長度大小的串列，不會在途中還未確認字串長的的情況下就使之長度確立與固定，且也可在其之中加入預設最大長度的限制，以防造成無限的延伸而造成跑不出任何結果。

```
creatsting([N|X]):-
      N#>=0, N#<=20,creatchars(X,N).
delay creatchars(_,N) if var(N).
creatchars([],0).
creatchars([R | Rs],N) :-
       N>0,
       N2 #= N-1,
       creatchars(Rs,N2),
       chars(R).
chars(N):-
       N:: [65..90 , 97..122].
```

圖 5　創建字串的限制敘述

乙、物件限制語言轉換成限制邏輯程式

對於限制邏輯程式中，我們希望的流程是在限制的狀態下實例化，而不是去實例化後才去做確認是否符合，而在原本的限制邏輯語言函式中大多都是用來實例化後檢查是否符合，與因格式改變的關係而另外實作了於限制邏輯程式中的函式。而我們實做了的函式與對應如表 2 所示。

表 2 物件限制語言對應的限制邏輯程式

物件限制語言	限制邏輯程式
+ 與 concat(s:String)	ocl_string_concat
size()	ocl_string_size
subString(lower:Integer ,upper:Integer)	ocl_string_substring
toUpperCase()	ocl_string_toUpperCase
toLowerCase()	ocl_string_toLowerCase
indexOf(s:String)	ocl_string_indexOf
equalsIgnoreCase(s:String)	ocl_string_equalsIgnoreCase
at(i:Integer)	ocl_string_at
< , > , = , >= , <=, <>	compareTo

圖 6 與圖 7 中是用來實作在物件限制語言中 substring 的函式轉換到到限制邏輯語言中的例子圖 6 中前半部分先行透過字串的長度做限制，其之後在得知並確認長度與位置的狀況下會立刻對字串作相對應的限制，在還未知長度或位置的狀況下並不會做限制，以減少造成在不正確的情況下先行實例化到錯誤位置的可能性。

```
ocl_string_substring([LengthX|StringX], Position,
Z, [LengthResult,StringResult]):-
       Position#>=1,
       LengthResult#=Z-Position+1,
       LengthX#>=Position,
       LengthX#>=Z,
       LengthResult#>=0,
       test_sub_B(StringX,StringResult,Position,L
engthResult).
```

圖 6 ocl_string_substring 的前半實作

```
delay test_sub_B(_,_,Position,LengthResult) if
var(Position);var(LengthResult).
test_sub_B([A|X],[B|Y],Position,LengthResult):
-
       (Position > 1 -> N#=Position-1,
       test_sub_B(X,[B|Y],N,LengthResult)
       ;(LengthResult #> 1 ->
       W#=LengthResult-1,A=B,
       test_sub_B(X,Y,Position,W);Y=[],A=B)).
```

圖 7 ocl_string_substring 的後半實作

```
ocl_string_concat ([A|X],[B|Y],[C|Z]):-
       C#=A+B,
       concat_B([A|X],[B|Y],[C|Z]).
delay concat_B([A|_],_,_) if var(A).
concat_B([A|X],[_|Y],[_|Z]):-
       length(X,A),
       append(X,Y,Z).
```

圖 8 ocl_string_concat 的實作

圖 8 中是用於字串合併的限制邏輯程式語言，對於限制邏輯語言中會在其確認到接在前面字串的長度確認的情況下才會開始做限制，避免造成合併後的字串中第二個字串所在的位置不符而造成回溯，但對於字串長度的限制會先去實行。

```
ocl_string_toUpperCase([A|X],[B|Y]):-
        A#=B,
        toUpperCase_B(X,Y).

delay toUpperCase_B(X,Y) if var(X),var(Y).
toUpperCase_B([],[]).
toUpperCase_B([A|X],[B|Y]):-
(A::97..122,B#=A-32;A::65..90,B#=A),toUpper
Case_B(X,Y).
```

圖 9 ocl_string_toUpperCase 的實作

```
ocl_string_toLowerCase([A|X],[B|Y]):-
        A#=B,
        toLowerCase_B(X,Y).

delay toLowerCase_B(X,Y) if var(X),var(Y).
toLowerCase_B([],[]).
toLowerCase_B([A|X],[B|Y]):-
(A::97..122,B#=A;A::65..90,B#=A+32),
toLowerCase_B(X,Y).
```

圖 10 ocl_string_toLowerCase 的實作

圖 9,10 中，是對於字串大小寫變換的函式，而後面對於在其字串加上限制開始的條件是在確認到該字串位置本身一定還有字的情況下才會開始執行做限制，如不確定的情況下則不會，以防造成多增加不必要的字元可能。

```
ocl_string_equalsIgnoreCase([A|X],[B|Y],Result):-
        (Result::0..1),
        (Result#=1,LengthX#=LengthY,
        equalsIgnoreCase_B(StringX,StringY,1))
        ;(Result#=0,(LengthX#\=LengthY;
        equalsIgnoreCase_B(StringX,StringY,0))).
```

圖 11 ocl_string_equalsIgnoreCase 前半的實作

```
delay equalsIgnoreCase_B(X,Y,_) if
var(X),var(Y).
equalsIgnoreCase_B([],[],1).
equalsIgnoreCase_B([A|X],[B|Y],1):-
        A::[65..90,97..122],
        B::[65..90,97..122],
        (A#=B;A#=B-32;A#=B+32),
        equalsIgnoreCase_B(X,Y,1).
equalsIgnoreCase_B([A|X],[B|Y],0):-
        A::[65..90,97..122],
        B::[65..90,97..122],
        ((A#\=B,A#\=B-32,A#\=B+32);
        (A#=B;A#=B-32;A#=B+32),
        equalsIgnoreCase_B(X,Y,0)).
```

圖 12 ocl_string_equalsIgnoreCase 後半的實作

圖 11 與圖 12 是對字串本身不管大小寫相等的函式，對於各種不同的回傳結果會產生不同的限制，在圖 12 中可看出對於相同與不同中限制方式一，在 false 情況下只要有一個字元或長度不同就行，但對於 true 的情況下必須要全部符合所以限制的情況也會跟者不同。

圖 13 中 size 的對應則是直接提取第一個位置的長度即可而在 indexOf, at 則是排除某些 substring 與不同的情況下而沿用其部分。

圖 14 與圖 15 是對字串的比較的函式，這裡是採用類似於 java 字串比較的功能，最後根據 Result 的限制來表示字串的比較大小結果，其中 Result 等於 0 時是相等，非 0 時是不相等，大於 0 時代表前面字串大於後面，小於 0 則相反，如此做出各種比較，這邊則是分成大於小於兩種情況特別分離寫出，且在未知長度的情況下並不會執行個別字元的限制部分。

```
ocl_string_size([Length|_], Result) :-
        Result#=Length.

ocl_string_at(X, Position,
[LengthResult|StringResult]):-
        LengthResult = 1,
        ocl_string_substring(X, Position, Position,
        [LengthResult|StringResult]).

ocl_string_indexOf([LengthX|StringX],
[LengthY|StringY], Result):-
        Result#>=0,
        (LengthX==0 ->
        Result=0;(LengthY==0->Result=1;
        ((Z#=LengthY+Result-1,ocl_string_substri
        ng( [LengthX|StringX], Result, Z,
        [LengthY|StringY]));Result=0))).
```

圖 13 size()、indexOf`(s)、at(i) 的實作

```
compareTo([A|X],[B|Y],Result):-
(Result#=0,X=Y,A=B);
(Result#>0,
compare_larger_length([A|X],[B|Y],Result))
;(Result#<0,
compare_small_length([A|X],[B|Y],Result))
```

圖 14 compareTo 前半的實作

```
delay
compare_larger_length([X|_],[Y|_],Result)      if
var(Y);var(X).
compare_larger_length([A|X],[B|Y],Result):-
        (A>B->Result#=A-B;
        compare_larger_word(X, Y,Result)).
compare_larger_word([X|Xs],[Y|Ys],Result):-
        (X#>Y,Result#=X-Y);
(Y#=X,compare_larger_word(Xs,Ys,Result)).

delay compare_small_length([X|_],[Y|_],Result)
if var(X);var(Y).
compare_small_length([A|X],[B|Y],Result):-
        (B>A->Result#=A-B;
        compare_small_word(X,Y,Result)).
compare_small_word([X|Xs],[Y|Ys],Result):-
        (Y#>X,Result#=X-Y);
(Y#=X,compare_small_word(Xs,Ys,Result)).
```

圖 15 compareTo 後半的實作

丙、字串拆解成字元串列原因與效率的比較

在 ECLiPSe 執行限制邏輯程式中字串本身並不能做單一字元的限制，進而很容易變成在限制單一字元時字串全部字元與長度都會被固定或字串在實例化時並沒有被限制，而在實例化後才去執行限制敘述是否相符判是否需重新實例化，但拆成字元串列格式就可以做單一字元的限制，如 A~Z 字元組成的長度 5 個字串，我們想要的結果只有開頭為 Z 的字串，則在原本最多需實例化 26^5 的次數，而限制過後的只需要 26^4 就能達成，省下不少的花費時間，且在開頭就確定開頭一定不是 Z 時就可以立即跳出該區段。

四、群集資料型態的支援

對於物件限制語言中，單一物件的變數不一定只有一個，且同一資料型態更不一定只有幾個，大量的資料處理也是相當常見，因此對於大量相同的資料型態以群集來表示也是不缺乏的。利用群集等的資料型態也可以將物件的關聯限制與表達融入物件限制語言中，因此可只透過物件限制語言表達出關聯上的限制，同時也能將關聯融入物件屬性裡面表達出來。且對單一型態大量資料的物件屬性表達也更方便，對限制邏輯程式與物件限制語言運算的操作上有一定的便利性。

甲、群集定義的格式

在限制邏輯程式群集的定義格式中，字串定義成字元串列格式時，可以說是與群集相同或類似的，所以採取類似於表 1 中字串的格式，並因物件的群集裡所需放的資料不同而放入不同的東西，但對於格式是相同統一個，對於放入的東西如表 3 所示，其中第一位置也是統一放上該串列的長度的部分，而對於後面的資料依據資料型態不同而有差異，而其中物件的型態時我們會採用如表 4 一樣的方式表示物件，串列第一個會表示他是一個物件，串列第二個會表示他的類別名字，第三個是他的 Oid，第四個之後則是所包含的屬性，而在物件屬性的群集裡面我們只會放入 Oid 部分代表被放入的物件是何者，並不會整的物件放入其中。

同時我們也希望將這些形式的的資料放入物件的屬性中直接表達物件之間的關聯與該物件之中有所含有的串列資料。最後透過將其提取出來轉換成測試資料。

表 3 群集資料的表示方法

資料型態	定義的形式	表示的意思
整數	[2,63,-1]	整數群集 {63,-1}
字串	[2,[1,65], [2,65,66]]	字串群集{'A',"AB"}
物件	[2,1,2]	代表特定物件中代表的 Oid 為 1.2 的物件有關聯

表 4 物件的表示方式

[uml_obj, ClsName, Oid, Attrs…]
[uml_obj, "Date", 1,2012,12,31]

乙、物件限制語言轉換成限制邏輯程式

對於被轉換成限制邏輯程式的串列中，都有者大致與字串形式相等的問題，對於某些資料在未確認的情形下不能保證其大致結果，但可以對其在盡可能的情況下做上相應的限制或是在盡可能的情況下做相對應的限制，而其中圖 16 可以說是其中的例子，也是從之前研究中的程式擷取出來的一段函式，其更改並不多，但也有可能因這情況省下很多時間。

```
delay ocl_set_equals(X, _, _) if nonground(X).
delay ocl_set_equals(_, Y, _) if nonground(Y).
ocl_set_equals(Set1, Set2, Result) :-
    list_to_ord_set(Set1, Aux1),
    list_to_ord_set(Set2, Aux2),
    ( Aux1 = Aux2 -> Result = 1; Result = 0 ).


delay test_equals(X,Y,Z) if
(nonground(X);noground(Y)),var(Z).
test_equals(Set1, Set2, Result):-
      (Z==1 ->X=Y; ocl_set_equals(X,Y,Z))
```

圖 16 ocl_set_equals 增加限制的可能

以圖 16 中的 ocl_set_equals 為例，在 Result 為確認是 1 即 true 的情況下我們可以先將兩個變數本身去做相等的限制，因他們的結果最後根據傳入的 Result 已經被限制除了回溯外已不再有其他可能，代表者兩者本身一定要相等才符合限制而不用再仔細確認裡面資料，但對於 Result 是 false 的情況下則還是必須做延遲上的等待，因在數學邏輯未知數上表示相等的同時也表示兩者的值也必定會相等，但在未知數沒有表示相等的情形下，其最後的值並不一定表示不相等，所以必須在完全或部分實例化已不相等的情況下才能確認其不相等，而不用再執行限制敘述檢查其最後值是否不相等。而也可能從已經實例化部分的資訊中確認其不相等，因此表示其不相等，但不能從只有部分實例化的資訊中表示他們已經相等的確定。

丙、群集優先限制比較與可能

在限制邏輯程式中有些情況的限制下在相同函式中不同情況下有各種不同限制的可能。而對於之中限制是否優先執行也有不同的影響，而其中有些情形的場合如確認是否相等的情況，在我們知道相等的情形就可以做相等的限制，而不相等的情形則不能做直接做限制，透過如此減少不必要的實例化以減少求解的所需時間，同時對於將關聯加入屬性的方式可以更明確的直接表達物件與何物件有直接的關聯。

五、效能評估

在這的章節中，會展示出我們將字串以拆成串列的形式做處理與沒有拆成字串做處理的差別，也會展示出用物件限制語言寫出的限制字串的測試案例與其透過我們程式產生出來的結果。

表 5 字串與字元串列產生時間的比較

	字串	字元串列
"JEFG"	0.02s	<0.01s
"JEFGA"	0.06s	<0.01s
"JEFGAG"	0.23s	<0.01s
"JEFGAGE"	2.14s	<0.01s
"JEFGAGEE"	21.2s	<0.01s
失敗狀況	21.4s	<0.01s

```
creatsting(X):-
[A]::2..8,
creatchars(A,N),
labeling([A|N]),
string_list(X,N).

delay creatchars(N,_) if var(N).
creatchars(0,[]).
creatchars(N, [R | Rs]) :-
        N > 0,
        N1 is N - 1,
        creatchars(N1,Rs),
        chars(R).

chars(N):-
        N#>=65,N#=<74.
```

圖 17 以原本型態產生字串的函式

```
creatsting([N|X]):-
N#>=0,N#<=8
creatchars(X,N).

delay creatchars(_,N) if var(N).
creatchars([],0).
creatchars([R | Rs],N) :-
        N#>0,
        N2 #= N-1,
        creatchars(Rs,N2),
        chars(R).

chars(N):-
        N::[65..90,97..122].

test(A):-
creatsting(A),
flatten([A], L),
selectlist(is_solver_type, L, ToBeLabeling),
labeling(ToBeLabeling),
flatten([A], L2),
selectlist(is_solver_type, L2, ToBeLabeling2),
labeling(ToBeLabeling2).
```

圖 18 以字元串列表示字串產生函式

表 5 中是根據圖 17 與圖 18 的程式分別測試出來的結果部分，我們會設定其字串最長長度為 8 以防止無限延伸，實際測試圖 18 程式時會先將字串轉換成符合的格式做測試，而對於這樣測試方法相當與產生符合全部的字串，而最後則是測試不在我們範圍內的字元輸入發現所需時間，而測試的結果發現在原本的字串定義下會比此種方式多上相當多的產生時間。

表 6~9 中是根據物件限制語言內建函式轉換成的限制邏輯程式的測試結果，第一列為所代表的物件限制語言函式，第二列為限制邏輯程式，第一與第二列中以相同變數名稱的方式表示物件限制語言的參數與回傳轉換到限制邏輯程式的相對位置，第三列後左方則是針對不同情況所給予的限制，右方則是通過實例化後得到的結果之一或是判斷是否符合限制邏輯。

表 6 concat 測試結果

物件限制語言	C=A.concat(B)
限制邏輯程式	ocl_string_concat(A,B,C)
ocl_string_concat(A,B,C)	A=[0],B=[0],C=[0]
ocl_string_concat([1,65],B,C)	B=[0],C=[1,65]
ocl_string_concat(A,B,[2,66,67])	A=[0],B=[2,66,67]
ocl_string_concat([1,65],[1,66], [2,66,67])	fail
ocl_string_concat([1,65],[1,66], [2,65,66])	true

表 7 equalsIgnoreCase 測試結果

物件限制語言	Result=A.equalsIgnoreCase (B)
限制邏輯程式	ocl_string_equalsIgnoreCase (A,B,Result)
ocl_string_equalsIgnoreCase (A,B,Result)	A=[0],B=[0],Result=1
ocl_string_concat([1,65],B,C)	B=[1,65], Result=1
ocl_string_concat(A,B,0)	A=[0],B=[1,65]
ocl_string_concat([1,65],[1,66], 0)	true
ocl_string_concat([1,65],[1,66], 1)	fail

表 8 toUpeerCase 測試結果

物件限制語言	Result=A.toUpeerCase()
限制邏輯程式	ocl_string_toUpeerCase(A,Result)
ocl_string_toUpeerCase(A,Result)	A=[1,65], Result=[1.65]
ocl_string_toUpeerCase([1,97],Result)	Result=[1,65]
ocl_string_concat([1,65],[1,98])	fail
ocl_string_concat([1,65],[1,65])	true

表 9 toLowerCase 測試結果

物件限制語言	Result=A. toLowerCase ()
限制邏輯程式	ocl_string_toLower Case (A,Result)
ocl_string_toUpeerCase(A,Result)	A=[1,65], Result=[1.97]
ocl_string_toUpeerCase([1,97],Result)	Result=[1,97]
ocl_string_concat([1,65], [1,98])	fail
ocl_string_concat([1,97], [1,97])	true

我們也寫了一個類別 JavaStringTest，該類別包函一個屬性叫 word 是字串型態，也包含兩個 java 函式 startwith(String)與 endwith(String)，我們也將此兩函式分別寫出物件限制語言如圖 19、圖 20，我們之後透過程式找出各有兩條主要分別為 result 是 true 與 false 的測試路徑，之後將測試路徑轉換成限制邏輯程式，最後執行限制邏輯程式得到以表 10 方式表示的測試資料分別為表 11、表 12，其中執行時間也同樣是在 0.01 秒內。

```
context JavaStringTest :: startwith(s:String) :
Boolean
post:
let e=self.word@pre.substring(1,s.size()) in
if e=s then
        result=true
else
        result=false
endif
```

圖 19 startwith 的物件限制語言

```
context JavaStringTest :: endwith(s:String) :
Boolean
post:
        let b=self.word@pre.size() in
        let c=b-s.size()+1 in
        let e=self.word@pre.substring(c,b) in
        if e=s then
                result=true
        else
                result=false
        endif
```

圖 20 endwith 的物件限制語言

表 10 資料表示格式

[回傳值,[物件執行前狀態, 物件執行後狀態], [參數傳入前狀態,參數傳入後狀態]]

表 11 startwith 產生測試資料

第一筆
[1, [(uml_obj, JavaStringTest, 1, [2, 65, 65]), (uml_obj, JavaStringTest, 1, [2, 65, 65])], [[2, 65, 65], [2, 65, 65]]]
第二筆
[0, [(uml_obj, JavaStringTest, 1, [2, 66, 65]), (uml_obj, JavaStringTest, 1, [2, 66, 65])], [[2, 65, 65], [2, 65, 65]]]

表 12 endwith 產生測試資料

第一筆
[1, [(uml_obj, JavaStringTest, 1, [2, 65, 65]), (uml_obj, JavaStringTest, 1, [2, 65, 65])], [[2, 65, 65], [2, 65, 65]]]
第二筆
[0, [(uml_obj, JavaStringTest, 1, [2, 66, 65]), (uml_obj, JavaStringTest, 1, [2, 66, 65])], [[2, 65, 65], [2, 65, 65]]]

六、結論

在研究中，基於利用限制邏輯語言中產生字串的方式做了研究，在最初的時候原本是想以直接字串的方式去做限制、比較、實例化等，但對其結果卻不甚滿意而做不斷的嘗試，更從中得出如何有效率的削減在實例化時不需要產生的資料，對其結果有更大的影響與可能，且對在限制邏輯語言中本身字串即為單一變數有很大的不便利性，進而嘗試以不同方式表示字串時就以字元串列的方式去表達字串，而其中字串以字元串列的表達方式與運作更能將其延伸到群集等的部分，字串拆成字元串列後如同多個整數變數可以個別同時做限制與相等，比起原本以字串的方式容易與快速的限制，進而大量削減不必要的實例化，而加快執行速度達成有效率的在限定時間之類快速的產生所需的字串測試資料，同時我們也讓物件限制語言中的群集轉換成限制邏輯程式，使限制邏輯程式執行後的輸出可以用串列的方式在物件的屬性中表達他有多少的測試資料，同時也能用串列的方式表達他與何物件有相應的關聯存在，而對其後的一般程式中的陣列支援更進一步。

參考文獻

[1] Object Management Group, Unified Modeling Language Infrastructure, Version 2.4.1, 2011.

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

[3] 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.

[4] 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.

[5] 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.

[6] K. R. Apt and M. G. Wallace, The ECLiPSe Constraint Programming System, Available: http://eclipseclp.org/. [Accessed Jun, 2015].

[7] H. Belhaouari and F. Peschanski, "Automated Generation of Test Cases from Contract-Oriented Specifications: A Csp-Based

Approach," Proceedings of 11th IEEE International Symposim on High Assurance Systems Engineering. IEEE, pp. 219-228, 2008.