THỰC THI TƯỢNG TRƯNG TRONG SINH TỰ ĐỘNG DỮ LIỆU KIỂM THỬ PHẦN MỀM

Tô Hữu Nguyên[a*], Nguyễn Hồng Tân[a], Hà Thị Thanh[a], Đỗ Thanh Mai[b]

[a]*Trường Đại học Công nghệ Thông tin và Truyền thông, Đại học Thái Nguyên, Thái Nguyên, Việt Nam*
[b]*Khoa Ngoại ngữ, Đại học Thái Nguyên, Thái Nguyên, Việt Nam*

Nhận ngày 04 tháng 01 năm 2016
Chỉnh sửa ngày 10 tháng 03 năm 2016 | Chấp nhận đăng ngày 16 tháng 03 năm 2016

Tóm tắt

Trong hoạt động kiểm thử phần mềm, các ca kiểm thử thường được tạo ra một cách thủ công, gây tốn kém về chi phí cũng như thời gian để hoàn thành công đoạn này. Thực thi tượng trưng (Symbolic execution) được biết đến là một kỹ thuật nổi tiếng với khả năng tự động sinh những bộ test case có độ bao phủ cao với các tiêu chí kiểm thử nhằm phát hiện những lỗi sâu trong các hệ thống phần mềm phức tạp. Bài báo trình bày các vấn đề tổng quan và một số kết quả của các nghiên cứu gần đây về kỹ thuật thực thi tượng trưng. Bài báo cũng đưa ra những thách thức cần giải quyết trong lĩnh vực này như: sự bùng nổ đường thực thi của chương trình, khả năng giải các ràng buộc, mô hình hóa bộ nhớ, các vấn đề về tương tranh vv.. đồng thời đưa ra một số đánh giá từ những kết quả đã công bố.

Từ khóa: Dữ liệu kiểm thử; Giải ràng buộc; Ràng buộc; Sinh dữ liệu kiểm thử; Thực thi tượng trưng; Thực thi tượng trưng động.

1. GIỚI THIỆU

Hiện nay có rất nhiều công cụ nền tảng phục vụ cho hoạt động kiểm thử phần mềm như JUnit [33] cho ngôn ngữ Java, NUnit [34], VSUnit [29] cho .NET để thực thi các ca kiểm thử mức đơn vị. Tuy nhiên, các công cụ kiểm thử này không hỗ trợ việc sinh tự động các ca kiểm thử đơn vị. Viết các ca kiểm thử là một công việc nặng nhọc và tốn nhiều công sức. Có nhiều phương pháp khác nhau hỗ trợ việc sinh tự động các ca kiểm thử (Test Case) giúp giảm chi phí và thời gian thực hiện đã được nghiên cứu và đưa ra như: Dựa trên mô hình (Model Checking), kểm thử ngẫu nhiên (Random Testing [1]). Nhưng hạn chế của nó là kiểm tra cùng một hành vi thực thi của chương trình nhiều lần với những đầu vào khác nhau và chỉ có thể kiểm tra được một số trường hợp

* Tác giả liên hệ: Email: thnguyen@ictu.edu.vn

thực thi của chương trình. Thêm vào đó, kiểm thử ngẫu nhiên khó xác định được khi nào việc kiểm thử nên được dừng lại và nó không biết tại điểm nào không gian trạng thái đã được thám hiểm hết. Để xác định khi nào việc kiểm thử dừng lại thì hệ thống kiểm thử ngẫu nhiên được kết hợp với các tiêu chuẩn an toàn [3]. Để khắc phục những hạn chế của kiểm thử ngẫu nhiên, phương pháp thực thi tương trưng xây dựng các ràng buộc trên các giá trị tượng trưng và giải các ràng buộc đó để sinh ra các giá trị đầu vào cho chương trình mà có thể bao phủ tất các dòng lệnh cũng như các nhánh thực thi của chương trình.

Ý tưởng của thực thi tượng trưng đã được đề xuất bởi King [Comm. ACM 1976], Clarke [IEEE TSE 1976] [2,10,18,21] nhưng việc hiện thực ý tưởng mới chỉ được thực hiện trong những năm gần đây qua tiến bộ đáng kể trong lý thuyết giải các ràng buộc (Constrain satisfiability) [11] và các tiếp cận mở rộng thực thi tượng trưng động (dynamic Symbolic execution) [30,16], một kỹ thuật kết hợp giữa các giá trị cụ thể và giá trị tượng trưng cho các giá trị đầu vào.

2. TỔNG QUAN VỀ KỸ THUẬT THỰC THI TƯỢNG TRƯNG

Ý tưởng chính của thực thi tượng trưng là thực thi chương trình với các giá trị tượng trưng (Symbolic value) thay vì các giá trị cụ thể (concrete value) của các tham số đầu vào kết quả là giá trị đầu ra được tính toán bởi chương trình và được biểu diễn bởi một biểu thức tượng trưng. Trong kiểm thử phần mềm, kỹ thuật thực thi tượng trưng được sử dụng để sinh dữ liệu kiểm thử cho mỗi đường thực thi khác nhau của chương trình. Ví dụ trong Hình 1 minh họa thực thi tượng trưng.

Trong quá trình thực thi tượng trưng, việc đi theo một nhánh cụ thể nào đó không phụ thuộc vào các giá trị của các tham số đầu vào. Tại tất cả các điểm rẽ nhánh tất cả các nhánh sẽ được xem xét và kiểm tra nhằm định hướng cho thực thi tiếp theo của chương trình. Với những chương trình ở dạng đơn giản có hai loại thực thi chủ yếu: đó là câu lệnh gán và câu lệnh rẽ nhánh. Tại các câu lệnh gán, giá trị tượng trưng của biến chương trình cũng như các tham số đầu vào có liên quan tới câu lệnh đó được tính toán và cập nhật lại, còn tại các điểm rẽ nhánh, chương trình sẽ điều khiển thực thi theo cả hai nhánh tương ứng đồng thời ràng buộc đường đi (path condition) tương ứng với

hai nhánh sẽ được tạo ra. Một ràng buộc là một biểu thức điều kiện tương ứng với giá trị true và ràng buộc kia tương ứng với giá trị sai của biểu thức ràng buộc. Các ràng buộc này sẽ được cập nhật vào điều kiện đường đi tương ứng với nhánh đó. Các điều kiện này sẽ được xem xét bởi một bộ giải ràng buộc để đánh giá xem đường đi tiếp theo có khả thi hay không hay nói một cách khác là kiểm tra xem có tồn tại bộ giá trị thỏa mãn ràng buộc này hay không. Nếu không thỏa mãn thì ràng buộc sẽ đánh giá là sai, khi đó thực thi tượng trưng sẽ dừng hoặc quay lui thực thi theo nhánh khả thi.

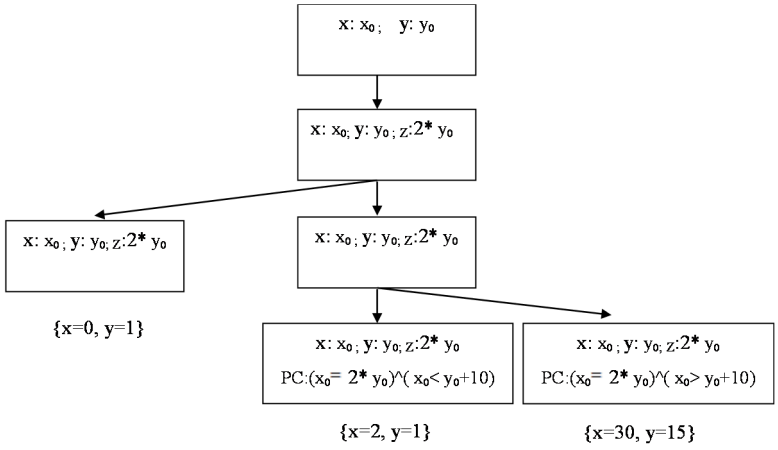
```
1   int twice (int v) {
2           return 2*v;
3   }
4
5   void testme (int x, int y) {
6           z = twice (y);
7           if (z == x) {
8                   if (x > y+10)
9                           ERROR;
10                  }
11          }
12  }
13
14  /* simple driver exercising testme() with sym inputs */
15  int main() {
16          x = sym_input();
17          y = sym_input();
18          testme(x, y);
19          return 0;
20  }
```

Hình 1. Ví dụ đơn giản cho thực thi tượng trưng

Các ràng buộc đường đi được tạo ra bằng cách thu gom các điều kiện trên đường đi tương ứng. Giải các ràng buộc này sẽ sinh ra các giá trị cụ thể cho các tham số đầu vào tương ứng với từng nhánh thực thi của chương trình.

Tất các các đường thực thi của chương trình có thể biểu diễn bởi một cấu trúc cây gọi là cây thực thi (execution tree). Hình 2 là cây thực thi tượng trưng cho hàm testme() trong Hình 1. Các cạnh của cây biểu diễn cho sự chuyển đổi từ trạng thái này sang trạng thái khác. Với ví dụ trên phương thức *testme()* trong Hình 1 sẽ có cây thực thi mô tả những đường thực thi của chương trình với các biến đầu vào là *{x=0, y=1}, {x=2, y=1} và {x=30, y=15}*, mục tiêu là sinh ra tập các ca kiểm thử thỏa mãn thực thi cho tất cả các nhánh của chương trình phụ thuộc vào giá trị tượng trưng của các tham biến đầu vào nhiều nhất có thể trong một khoảng thời gian nhất định đảm bảo khám phá

chính xác tất cả các đường thực thi bởi một lần duy nhất với những giá trị đầu vào đã cho.



Hình 2. Cây thực thi tượng trưng tương ứng với hàm testme() trong ví dụ 1

Bắt đầu thực thi tượng trưng hàm testme() bằng việc gán giá trị cho các tham số đầu vào x và y lần lượt là x_0 và y_0. Khởi tạo PC (Path Condition) nhận giá trị là True, tới câu lệnh rẽ nhánh *if(2*y_0= x_0)* hai nhánh của chương trình tương ứng đều được thực thi với các giá trị tượng trưng là x_0 và y_0. Tại đây biểu thức điều kiện rẽ nhánh là *2*y_0=x_0* và *!(2*y_0=x_0)* được bổ sung và PC theo hai nhánh khác nhau. Sau khi thực thi câu lệnh *if(2*y_0= x_0)* hàm testme() được thực thi theo nhánh mà tồn tại giá bộ giá trị *x,y* thỏa mãn. Tương tự như trên khi gặp câu lệnh *if(x_0>y_0+10)* PC theo hai nhánh tương ứng sẽ được cập nhật bổ sung là *(2*y_0=x_0)^(x_0>y_0+10)* và *(2*y_0=x_0)^(x_0<y_0+10)*. Tại mỗi điểm rẽ nhánh PC sẽ được cập nhật và một thủ tục quyết định bằng bộ xử lý ràng buộc sẽ được sử dụng để xác định xem nhánh tương ứng với PC đó có khả thi hay không để điều hướng thực thi hiện thời đi theo nhánh đó. Nếu PC được đánh giá là không khả thi, thực thi tương trưng sẽ dừng hoặc quay lui, và thực thi tượng trưng chỉ thực thi chương trình tương trưng theo nhánh mà PC được đánh giá là khả thi.

Đối với những đoạn mã chứa vòng lặp hoặc lời gọi đệ quy mà đường thực thi là vô hạn và biến điều khiển của vòng lặp hoặc lời gọi đệ quy là tượng trưng.Ví dụ trong Hình 3 sẽ là vô hạn đường thực thi với mỗi đường thực thi là 1 dãy tùy ý số lượng giá trị là true và sau cùng là false hoặc là 1 dãy vô hạn số lần lặp, khi đó PC của đường thực thi là với 1 dãy gồm n giá trị true và sau cũng là false:

```
1  void testme_inf () {
2          int sum = 0;
3          int N = sym_input();
4          while (N > 0) {
5              sum = sum + N;
6              N = sym_input();
7          }
8  }
```

Hình 3. Ví dụ cho vòng lặp vô hạn

Nơi mà mỗi N_i nhận lại giá trị tượng trưng và trạng thái cuối cùng là

$$\left(\bigwedge_{i\in[1,n]} N_i > 0\right) \wedge (N_{n+1} \leq 0)$$

$\{N \rightarrow N_{n+1}, Sum \rightarrow \sum_{i\in[1,n]} Ni\}$.

Trên thực tế cần phải thêm các tiêu chí giới hạn cho việc tìm kiếm như thời gian thực thi, giới hạn chiều sâu của đường thực thi hoặc số lượng các vòng lặp lồng nhau.

Nhược điểm chính của thực thi tượng trưng truyền thống là không thể sinh dữ liệu test nếu không giải được biểu thức ràng buộc đường thực thi. Ví dụ nếu ta thay hàm twice() ở hình 1 bằng hàm twice() ở Hình 4 sau khi thực hiện câu lệnh dòng 7 thì thực thi tương trưng sẽ cho ra hai ràng buộc mới bổ sung $x_0 \neq y_0 * y_0 \% 50$ và $x_0 = y_0 * y_0 \% 50$ và khi bộ giải ràng buộc không giải được ràng buộc trên thực thi tượng trưng sẽ thất bại trong việc sinh các dữ liệu kiểm thử cho chương trình sau khi đã được sửa đổi.

```
1  int twice (int v) {
2          return (v*v) % 50;
3  }
```

Hình 4. Ví dụ cho hàm twice() là phi tuyến

Trong phần này, chúng tôi sẽ trình bày trong phần hai về các kỹ thuật thực thi tượng trưng hiện đại mà có ít nhất một vài dữ liệu kiểm thử giải quyết được vấn đề trên

3. CÁC KỸ THUẬT THỰC THI TƯỢNG TRƯNG HIỆN ĐẠI

Một trong những yếu tố quan trọng của kỹ thuật thực thi tượng trưng hiện đại là khả năng kết hợp giữa giá trị cụ thể và giá trị tượng trưng. Chúng tôi sẽ trình bày dưới đây hai kỹ thuật quan trọng và đánh giá những ưu điểm mà các kỹ thuật này cung cấp.

3.1. Concolic testing

Directed Automated Random Testing (DART) [13] thực hiện kỹ thuật tượng trưng động (là kỹ thuật phân tích chương trình động). Thực thi tượng trưng động chính là sự kết hợp giữa thực thi cụ thể và thực thi tượng trưng. Trong thực thi tượng trưng động, chương trình được thực thi nhiều lần với những giá trị khác nhau của tham số đầu vào.

Bắt đầu bằng việc chọn những giá trị tùy ý cho các tham số đầu vào và thực thi chương trình với những giá trị cụ thể đó chương trình sẽ được thực thi theo một đường đi xác định. DART thực thi chương trình với các giá trị cụ thể của tham số đầu vào và thu gom các ràng buộc trong quá trình thực thi theo đường đi mà sự thực thi cụ thể này đi theo, đồng thời suy ra các ràng buộc mới từ những ràng buộc đã thu gom được.

Tại các câu lệnh rẽ nhánh, biểu thức điều kiện rẽ nhánh sẽ được đánh giá theo các giá trị cụ thể của các tham số đầu vào. Nếu biểu thức điều kiện rẽ nhánh nhận giá trị là True thì biểu thức của điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của PC và được ghi nhớ, đồng thời phủ định của điều kiện rẽ nhánh sẽ được sinh ra và được thêm vào một PC tương ứng với nhánh còn lại mà sự thực thi cụ thể đó không đi theo. Một bộ xử lý ràng buộc (Constraint Solver) sẽ được sử dụng để giải quyết các ràng buộc mới sinh ra này để sinh ra các giá trị cụ thể của tham số đầu vào. Trong trường hợp ngược lại biểu thức phủ định của điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của PC tương ứng với nhánh mà sự thực thi hiện thời đang đi theo và được ghi nhớ. Đồng thời điều kiện rẽ nhánh sẽ được sinh ra và thêm vào PC tương ứng với nhánh còn lại mà sự thực thi hiện thời không đi theo. Các giá trị mới được sinh ra của các tham số đầu vào sẽ tiếp tục được thực thi và quá trình này sẽ được lặp lại cho tới khi chương trình được thực thi theo tất cả các đường đi. Do các chương trình được thực thi với những giá trị cụ thể nên có thể thấy rằng tất cả các đường đi phân tích được trong quá trình thực thi tượng trưng động đều là các đường đi khả thi.

Với ví dụ trong Hình 1, Concolic testing sẽ sinh ra một vài giá trị ngẫu nhiên cho tham số đầu vào, giả sử là *{x=22,y=7}* và thực thi chương trình với cả với các giá trị tượng trưng và cụ thể. Thực thi với giá trị cụ thể sẽ theo nhánh else tại dòng 7 và

thực thi tượng trưng sẽ bổ sung ràng buộc $(x_0 \neq 2y_0)$ theo đường thực thi của các giá trị cụ thể, đồng thời Concolic testing sẽ lấy phủ định trong ràng buộc đó và giải ràng buộc tương ứng của biểu thức $x_0=2*y_0$ để sinh ra dữ liệu tương ứng là *{x=2,y=1}*. Đầu vào mới này sẽ bắt buộc chương trình thực thi theo nhánh khác với các giá trị cụ thể là {x=22, y=7}, tại câu lệnh rẽ nhánh ở dòng 8, giống như trên chương trình thực thi với giá trị {x=2,y=1} sẽ thực thi theo nhánh thỏa mãn ràng buộc $(x_0=2*y_0)\&(x_0 \leq y_0+10)$ lấy phủ định của biểu thức $(x_0 \leq y_0+10)$ và sẽ giải ràng buộc tương ứng là $(x_0=2*y_0)\&(x_0>y_0+10)$ để sinh ra ràng buộc tương ứng là *{x=30, y=15}*. Chương trình sẽ gặp câu lệnh *ERROR* với đầu vào mới này, sau lần thực thi lần thứ 3 này Concolic testing sẽ báo rằng tất cả các đường thực thi của chương trình đã được khám phá và dừng thực hiện. Chú ý rằng trong ví dụ này concolic testing đã khám phá tất cả các đường thực thi của chương trình và sử dụng chiến thuật tìm kiếm theo chiều sâu, tuy nhiên cũng có thể áp dụng chiến thuật khác cho việc tìm kiếm được đề cập ở mục 4.1.

3.2. Execution Generated Testing (EGT)

Phương pháp tiếp cận EGT thực hiện mở rộng của cả hai công cụ EXE [8] và phương pháp KLLE [4]. Nó hoạt động dựa trên nguyên lý tạo ra sự khác biệt giữa giá trị cụ thể và trạng thái tượng trưng của chương trình.

EGT trộn lẫn giá trị cụ thể và thực thi tượng trưng bằng cách kiểm tra động trước mọi toán hạng. Nếu các giá trị liên quan đó hoàn toàn là các giá trị cụ thể thì toán hạng đó sẽ thực thi như chương trình gốc. Còn nếu có ít nhất một giá trị là tượng trưng thì toán hạng sẽ thực hiện thực thi tượng trưng và cập nhật điều kiện ràng buộc đường đi cho đường thực thi hiện thời.

Ví dụ trong chương trình ở Hình 1 lệnh trên dòng 17 được thay đổi là *y=10,* sau dòng 6 chỉ cần gọi phương thức twice() với tham biến trên và cho giá trị cụ thể là 20. Lời gọi đó sẽ thực thi như là với chương trình gốc. Do vậy tại dòng 7 câu lệnh rẽ nhánh sẽ là $if(x_0=20)$ và thực thi tượng trưng sẽ cho phép thực hiện theo hai nhánh if và else tương ứng với ràng buộc là $x_0=20$ và $x_0 \neq 20$. Khi gặp câu lệnh tại dòng 8, biểu thức ràng buộc tương ứng với nhánh này *là* $(x_0=20)\&(x_0>20)$. Lúc này bộ giải ràng buộc sẽ đánh giá là đường thực thi không khả thi và thực thi tượng trưng sẽ quay lui.

Concolic và EGT là hai đại diện cho kỹ thuật thực thi hiện đại và tiến bộ chính là khả năng trộn giá trị cụ thể với thực thi tượng trưng và được gọi chung là kỹ thuật thực thi tượng trưng động (Dynamic symbolic execution).

3.3. Tính tương đối và đầy đủ trong thực thi tượng trưng động

Một trong những lợi thế chính của phương pháp kết hợp là *"tương đối"* do phải tương tác với các mã nguồn bên ngoài hoặc không thể đưa ra các giá trị thỏa mãn các ràng buộc của bộ giải ràng buộc và có thể được đơn giản hóa do sử dụng các giá trị cụ thể.

Ví dụ các chương trình thực tế luôn phải tương tác với thế giới từ bên ngoài như là: sử dụng các thư viện ngoài mà nó không hỗ trợ thực thi tượng trưng hoặc sử dụng các lời gọi từ hệ điều hành. Nếu tất cả các đối số được truyền cho lời gọi hàm như vậy đều là các giá trị cụ thể thì lời gọi hàm có thể được thực hiện một cách đơn giản với các giá trị cụ thể như là thực thi với chương trình gốc. Tuy nhiên, ngay cả khi nếu một vài toán hạng là tượng trưng thì thực thi tượng trưng động sẽ sử dụng một trong những giá trị cụ thể của toán hạng tương ứng đó.

EGT thực hiện đơn giản hóa ràng buộc chỉ để kiểm tra sự thỏa mãn tại đường đi hiện tại (EGT có thể được tối ưu hóa thông qua ví dụ sẽ được đề cập trong mục 4.2). Trong khi concolic sử dụng ngay các giá trị cụ thể tại thời điểm thực thi của chương trình cho đầu vào từ thực thi concolic hiện thời. Bên cạnh những mã nguồn bên ngoài, sự thiếu chính xác (tương đối) của thực thi tượng trưng còn thể hện ở một vài điểm như là không giải quyết được một số toán tử (dấu phảy động) hoặc những hàm phức tạp không giải được bởi bộ giải ràng buộc, việc sử dụng các giá trị cụ thể cho phép thực thi tượng trưng động khôi phục lại những thiếu sót và chi phí của nó là có thể thiếu một vài đường thực thi. Do vậy EGT mất đi tính đầy đủ.

Để làm rõ điều này chúng tôi sẽ minh họa hành vi của Concolic testing với ví dụ thực hiện hàm *twice()* mà giá trị trả về là phi tuyến *(v*v)%50* (xem trong Hình 4) . Giả sử rằng concolic bắt đầu thực hiện với giá trị ngẫu nhiên cho tham số đầu vào là *{x=22, y=7}.* Với giá trị này *concolic* sẽ sinh ra ràng buộc đường đi là: $x_0 \neq y_0 * y_0 \% 50$. Nếu giả

sử rằng bộ giải ràng buộc không giải được với với ràng buộc phi tuyến thì Concolic testing sẽ thất bại trong việc sinh dữ liệu đầu vào cho đường thực thi tiếp theo. Chúng ta có trường hợp tương tự khi mã nguồn cho hàm *twice()* là không xác định (do bên thứ 3 cung cấp và là mã nguồn đóng hoặc là lời gọi từ hệ thống). Trong trường hợp đó ràng buộc đường đi sẽ là $x_0=twice(y_0)$ và twice() là hàm không xác định (interpreted function). Concolic testing giải quyết tình huống này bằng cách thay thế một vài giá trị tượng trưng bằng giá trị cụ thể. Kết quả là ràng buộc sẽ trở lên đơn giản và có thể giải được bằng bộ giải hiện thời. Với ví dụ trên, Concolic testing sẽ thay thế y_0 bởi một giá trị cụ thể là 7 nó sẽ đơn giản hóa ràng buộc là $x_0=49$. Khi giải ràng buộc này, Concolic testing sẽ có kết quả là $\{x=49, y=8\}$ để thực thi những đường đi chưa khám phá trước đó.

Chú ý rằng khả năng đơn giản hóa ràng buộc do sử dụng các giá trị cụ thể giúp sinh các dữ liệu test cho những đường thực thi của chương trình mà thực thi tượng tưng bị cản trở. Nhưng, sự đơn giản hóa đó sẽ đi kèm với với một số thiếu sót nhất định, Nó có thể mất đi tính đầy đủ có nghĩa là nó có thể không sinh được dữ liệu kiểm thử cho một vài đường thực thi. Tuy nhiên, rõ ràng là nó phù hợp cho việc thay thế hoặc bỏ qua một vài đường thực thực thi không hỗ trợ hay đối với các hàm mở rộng từ bên ngoài khi gặp phải.

4. NHỮNG THÁCH THỨC VÀ MỘT VÀI GIẢI PHÁP

Trong phần này chúng ta sẽ thảo luận về những thách thức quan trọng trong thực thi tượng trưng và một vài giải pháp thú vị cho việc giải quyết các thách thức đặt ra.

4.1. Bùng nổ đường đi

Một trong những thách thức quan trọng của thực thi tượng trưng là nó sẽ sinh ra một số lượng lớn các đường thực thi mặc dù chương trình là nhỏ và thường là hàm mũ trong số các câu lệnh rẽ nhánh tĩnh của mã nguồn. Kết quả là trong một khoảng thời gian định trước nó sẽ quyết định khám phá con đường đầu tiên sao cho phù hợp nhất.

Trước hết, lưu ý rằng thực thi tượng trưng đã ngầm lọc ra tất cả các đường thực thi không phụ thuộc vào các biến tượng trưng đầu vào và những đường dẫn không khả

thi (Infeasiblle path) trong quá trình giải các ràng buộc. Mặc dù như vậy, sự bùng nổ đường dẫn vẫn là một trong những thách thức lớn nhất đối với thực thi tượng trưng. Hiện nay có 2 cách tiếp cận chính để giải quyết vấn đề này là ưu tiên tìm kiếm kinh nghiệm (Heuristic search) và sử dụng kỹ thuật phân tích tính đúng đắn của chương trình trong quá trình phân tích được trình bày dưới đây.

4.1.1. Tìm kiếm kinh nghiệm (Heuristic search)

Cơ chế quan được sử dụng trong các công cụ thực thi tượng trưng để ưu tiên đường dẫn khai phá là sử dụng tìm kiếm kinh nghiệm. Hầu hết Heuristic đặt trọng tâm vào vấn đề bao phủ dòng lệnh và bao phủ rẽ nhánh. Nhưng chúng cũng có thể được sử dụng để tối ưu hóa các tiêu chí mong muốn khác.

Một cách tiếp cận đặc biệt hiệu quả là sử dụng đồ thị luồng điều khiển (Control Flow Graph). Để điều hướng đường gần nhất từ những đường dẫn đã khám phá [4,8], một cách tương tự trong EXE [8] là ưu tiên những câu lệnh đã thực thi mà có số lần chạy là ít nhất. Một ví dụ khác với tìm kiếm Heuristic dựa trên khám phá ngẫu nhiên cũng đã được chứng minh tính hiệu quả [4,8]. Ý tưởng chính của phương pháp này là khi bắt đầu chương trình với mỗi câu lệnh rẽ nhánh của thực thi tượng trưng trong các nhánh khả thi nó sẽ chọn ngẫu nhiên 1 nhánh để khám phá.

Một tiếp cận thành công khác là xen vào kiểm thử ngẫu nhiên và khám phá tượng trưng (symbolic exploration). Phương pháp này kết hợp khả năng của kiểm thử ngẫu nhiên để nhanh chóng đến những trạng thái thực thi sâu của chương trình. Với sức mạnh của thực thi tượng trưng việc kết hợp được khai phá triệt để những trạng thái trong một vùng lân cận.

Gần đây hơn nữa thực thi tượng trưng còn được kết hợp với kỹ thuật tìm kiếm tiến hóa(evolutionnary seach). Trong đó hàm mục tiêu được sử dụng để định hướng không gian đầu vào của của các biến [3,19,22,27]. Ví dụ với công cụ Austin [22] sử dụng tìm kiếm tiến hóa trong tìm kiếm đường thực thi. Trong đó nó sử dụng một hàm mục tiêu phát triển tìm kiếm của không gian dữ liệu kiểm thử đầu vào. Với thực thi

tượng trưng động khai thác tốt được cả hai lĩnh vực trên, hiệu quả của kiểm thử phần mềm phụ thuộc vào chất lượng của hàm mục tiêu.

Một vài tiếp cận hứa hẹn khác [3,19,27] khai thác thông tin trạng thái thực thi của giá trị cụ thể và thực thi tượng trưng từ thực thi tượng trưng động và phân tích tĩnh để cải tiến hàm mục tiêu cho kết quả sinh dữ liệu kiểm thử tốt hơn. Kiểm thử đột biến nới các bộ test đầy đủ được đánh giá bằng cách kiểm tra khả năng xác định khác nhau của đột biến trong chương trình cũng đã kết hợp thành công với thực thi tượng trưng động [20].

Nhìn chung các phương pháp mới trong việc kết hợp thực thi tượng trưng và kỹ thuật tìm kiếm kinh nghiệm đã cho những kết quả hứa hẹn và chúng tôi tin rằng những tiến bộ hơn nữa trong lĩnh vực này có thể đóng vai trò quan trọng trong việc giải quyết sự bùng nổ đường dẫn trong thực thi tượng trưng

4.1.2. Kỹ thuật phân tích tính đúng đắn của chương trình

Một cách tiếp cận quan trọng khác cho vấn đề bùng nổ đường dẫn từ phân tích chương trình và kiểm chứng phần mềm nhằm giảm độ phức tạp của đường dẫn thực thi chương trình đó là sử dụng thông qua phân tích tính đúng đắn của chương trình. Cách tiếp cận đơn giản được sử dụng để giảm số lượng đường thực thi chương trình là gộp lại theo hướng tĩnh sử dụng biểu diễn bởi cấu trúc lựa chọn, sau đó được truyền trực tiếp cho bộ giải [13]. Tiếp cận này có thể có hiệu quả trong nhiều trường hợp nhưng nó không phù hợp khi truyền độ phức tạp cho bộ giải. Nội dung này sẽ được trình bày ở phần sau.

Các kỹ thuật kết hợp (Compositional) cải tiến thực thi tượng trưng bằng cách sử dụng bộ nhớ đệm và sử dụng lại phân tích của những hàm ở mức thấp trong những tính toán tiếp theo. Ý tưởng chính của kỹ thuật này được thể hiện trong các tài liệu [14,15] là tính toán, tóm tắt hàm đối với mỗi hàm kiểm thử được mô tả về điều kiện trước và giá trị đầu ra của hàm và sử dụng lại sự tóm tắt này đối với những hàm ở mức cao.

Sinh dữ liệu kiểm thử lười (Lazy test generation)[23]. Nó là một cách tiếp cận liên quan để tránh sự lặp lại trong khai phá đường dẫn bằng cách tự động cắt tỉa các đường dẫn dư thừa trong quá trình khám phá đường dẫn. Kỹ thuật này dựa trên ý tưởng nếu các đường dẫn đạt đến cùng một điểm và cùng có ràng buộc tương tự với đường dẫn trước đó thì sẽ được thực thi tương tự đồng thời đường dẫn đó sẽ bị loại bỏ.

4.2. Giải các ràng buộc

Mặc dù đã có những tiến bộ đáng kể trong kỹ thuật thực hiện giải các ràng buộc trong những năm gần đây, giúp cho thực thi tượng trưng trở nên thực tế hơn, nhưng việc giải các ràng buộc vẫn là cản trở đáng kể của thực thi tượng trưng. Nó thường chiếm nhiều thời gian trong quá trình thực hiện và là một trong những lý do chính khiến thực thi tương trưng không mở rộng được với một số loại chương trình mà mã nguồn của nó sinh ra với yêu cầu rất lớn trong việc giải các ràng buộc.

Kết quả là cần thiết phải thực hiện tối ưu khi giải các ràng buộc được sinh ra trong quá trình thực thi tượng trưng trong những chương trình thực tế. Chúng tôi trình bày dưới đây hai phương pháp tối ưu được sử dụng bởi các công cụ hiện tại.

4.2.1. Loại bỏ ràng buộc không liên quan

Phần lớn các yêu cầu của thực thi tượng trưng đặt ra là xác định tính khả thi của câu lệnh rẽ nhánh. Ví dụ: Concolic testing là một loại biến thể của thực thi tượng trưng. Một nhánh của ràng buộc đường dẫn hiện thời là phủ định của nó và tập các kết quả của ràng buộc sẽ kiểm tra tính thỏa mãn để xác định nếu chương trình có thể rẽ sang một nhánh khác của ràng buộc tương ứng với phủ định của ràng buộc hay không.

Một chú ý quan trọng là trong trường hợp chung nhất, nhánh của chương trình phụ thuộc vào một số lượng nhỏ các tham biến của chương trình trên một số lượng nhỏ các ràng buộc của điều kiện đường đi. Do vậy một cách tối ưu hóa hiệu quả là loại bỏ khỏi ràng buộc đường đi những ràng buộc không liên quan trong quyết định đến kết quả của đường dẫn hiện thời. Ví dụ: Xem xét ràng buộc của một đường đi hiện thời của thực thi là: *(x+y>0)^(z>0)^(y<12)^(z-x=0). G*iả sử chúng ta muốn sinh các giá trị đầu vào

mới từ giải ràng buộc $(x+y>0)\wedge(z>0)\& \neg(y<12)$ trong đó $\neg(y<12)$ là phủ định của nhánh điều kiện có tính khả thi mà đang thử xây dựng trong ràng buộc trên. Ta có thể an toàn loại bỏ *z* vì ràng buộc này không ảnh hưởng đến kết quả của nhánh $\neg(y<12)$. Giải pháp này sẽ giảm các thiết lập ràng buộc và sẽ cho những giá trị *x* và *y* mới. Chúng ta sử dụng giá trị *z* từ thực thi hiện thời để sinh các dữ liệu đầu vào mới.

Thực tế hơn, thuật toán sẽ tính toán tính bắc cầu của tất cả các ràng buộc trong các phủ định của ràng buộc phụ thuộc bằng cách xem xét sự chia sẻ bất kì các biến giữa chúng trong những trường hợp mở rộng. Trong giải quyết với tham chiếu con trỏ và chỉ số mảng đã được trình bày chi tiết trong các tài liệu [8,12,26].

4.2.2. Giải ràng buộc theo phương pháp gia tăng (Increamental Solving)

Một trong những tính chất quan trọng của tập các ràng buộc được tạo ra trong quá trình thực thi tượng trưng là nó mô tả biểu diễn cho một tập cách nhánh tĩnh từ mã nguồn của chương trình. Vì lý do này nhiều đường thực thi tượng trưng có thể có các ràng buộc tương tự và do vậy cũng có các giải pháp tương tự cho các nhánh này. Xuất phát từ thực tế này ta có thể khai thác nhằm cải thiện tốc độ của bộ giải ràng buộc và sử dụng lại kết quả tương tự với những truy vấn trước đó như đã thực hiện thành công trên vài công cụ CUTE[8,26] và KLEE minh họa cho vấn đề trên chúng tôi minh họa một thuật toán là phương pháp dùng bộ nhớ đệm sử dụng bởi KLEE.

Trong KLEE tất cả các kết quả truy vấn được lưu vào vùng nhớ đệm ánh xạ tập các ràng buộc và các giá trị cụ thể tìm được tương ứng với ràng buộc đó (trong trường hợp đặc biệt không có lời giải hoặc không thỏa mãn tương ứng với một kí hiệu đặc biệt). Chẳng hạn một ánh xạ trong bộ nhớ đệm có thể như sau $(x+y<10)\wedge(x>5) \rightarrow \{x=6,y=3\}$, sử dụng những ánh xạ này KLEE có thể cho kết quả nhánh chóng với một số yêu cầu tương tự liên quan đến tập cha của tập con ràng buộc đã lưu trong bộ nhớ đệm.

Hơn nữa nếu tập con của tập các ràng buộc đã lưu trong vùng nhớ đệm gặp phải KLEE có thể nhanh chóng kiểm tra nếu lời giải cho vùng nhớ đệm có hiệu lực bằng cách thêm những giá trị này vào tập cha, ví dụ: KLEE nhanh chóng kiểm tra bộ giá trị

{x=6,y=3} vẫn thỏa mãn truy vấn *(x+y<10)^(x>5)^(y≥0)* là tập cha của tập (x+y<10)^(x>5) KLEE khai thác khía cạnh này trong cài đặt bằng cách thêm vào các ràng buộc mở rộng nhưng không mất đi giá trị của lời giải hiện có.

4.3. Mô hình hóa bộ nhớ

Độ chính xác của các câu lệnh của chương trình khi chuyển sang các ràng buộc tượng trưng có ảnh hưởng đáng kể đến độ bao phủ khi thực hiện thực thi tượng trưng. Ví dụ khi sử dụng mô hình hóa bộ nhớ mà xấp xỉ để thiết lập độ rộng cho tham biến kiểu Interger có thể có hiệu quả hơn trong thực tế nhưng mặt khác kết quả lại thiếu chính xác trong phân tích mã nguồn. Nó phụ thuộc vào khoảng lựa chọn giá trị tương ứng như lỗi tràn bộ nhớ toán học, cũng có thể gây ra thiếu đường dẫn trong thực thi tượng trưng, hoặc khai phá những đường đi không khả thi vv..

Một ví dụ khác là mô hình hóa bộ nhớ với con trỏ. Trong một phạm vi cần giải quyết, DART chỉ được thực hiện với các giá trị cụ thể, hoặc hệ thống CUTE và CREST chỉ hỗ trợ và thực hiện giải tốt với ràng buộc kiểu đẳng thức và bất đẳng thức trên kiểu dữ liệu con trỏ. Trong các ngôn ngữ khác nhau giống như EXE [8] hoặc gần đây hơn là KLEE [4] và SAGE [12] thực hiện mô hình hóa con trỏ sử dụng khái niệm liên quan đến mảng, lựa chọn và cập nhật các thành phần bởi lời giải giống như với STP [13] và Z3 [11].

Việc cân bằng giữa độ chính xác và khả năng mở rộng có thể xác định được những phương diện khi phân tích mã nguồn và sự thực hiện chính xác giữa các bộ giải ràng buộc khác nhau. Ngoài ra chú ý rằng thực thi tượng trưng động cho phép điều chỉnh cả khả năng mở rộng và sự chính xác bởi các tùy biến đối với các giá trị cụ thể trong biểu thức tượng trưng.

4.4. Giải quyết vấn đề tương tranh

Một lượng lớn các chương trình trong thực tế thường sảy ra các vấn đề về tương tranh bởi vấn đề không xác định là vốn có của chương trình và kiểm thử chúng là rất khó khăn, mặc dù với thách thức như vậy nhưng Concolic testing đã sử dụng hiệu quả

kiểm thử những chương trình tương tranh bao gồm những chương trình có đầu vào phức tạp [24,25] hoặc là đối với những chương trình phân tán [6].

5. MỘT SỐ CÔNG CỤ TRONG THỰC THI TƯỢNG TRƯNG ĐỘNG

Thực thi tượng trưng động đã được cài đặt bởi một số công cụ trong các phòng thí nghiệm và các viện nghiên cứu [1,7,8,16,17,26,28] hỗ trợ nhiều ngôn ngữ lập trình như C, C++, java và thực hiện nhiều mô hình bộ nhớ khác nhau .Với mục tiêu là có thể áp dụng được với nhiều kiểu ứng dụng khác nhau và sử dụng một vài bội giải ràng buộc khác nhau, chúng tôi sẽ giới thiệu dưới đây một vài công cụ liên quan đến các phần lý thuyết đã trình bày ở trên.

5.1. SPF Symbolic JavaPathFinder

SPF [34] kết hợp thực thi tượng trưng và kiểm chứng mô hình và thực hiện giải các ràng buộc để sinh các dữ liệu test, SPF hỗ trợ thực thi tượng trưng động và thực thi trên các tệp bytecode (.class) và có thể áp dụng đối với mô hình cho các ngôn ngữ mô hình hóa. SPF có thể thực hiện trên các kiểu dữ liệu nguyên, kiểu thực, con trỏ, mảng, một số toán tử trên kiểu xâu (đang phát triển và hoàn thiện) và sử dụng kết hợp một số bộ giải ràng buộc như: Choco, IASolve, CVC3 thực hiện giải các ràng buộc tuyến tính, phi tuyến tính trên các kiểu số nguyên, số thực, kiểu tham chiếu (con trỏ) và kiểu xâu.

5.2. DART, CUTE và CREST

DART là công cụ đầu tiên kết hợp thực thi tượng trưng động với kiểm thử ngẫu nhiên và kỹ thuật kiểm thử mô hình (Model Checking) với mục tiêu là thực thi một cách có hệ thống tất các các đường thực thi của chương trình cho mỗi loại lỗi khác nhau. DART được cài đặt đầu tiên tại phòng thí nghiệm Bell để thực hiện kiểm thử trên ngôn ngữ C và đã là nguồn cảm hứng cho nhiều sự mở rộng và công cụ sau này và JCUTE cho ngôn ngữ java là một mở rộng của CUTE (Concolic unit Testing Engine) để giải quyết vấn đề đa luồng với các cấu trúc dữ liệu động sử dụng toán tử con trỏ trong các chương trình đa luồng. CUTE kết hợp Concolic và một phần thực thi tượng trưng động thiên về sinh ra một cách có hệ thống cả dữ liệu kiểm thử và lịch biểu của luồng. CUTE và JCUTE được phát triển tại đại học Illinois nhằm hỗ trợ các ngôn ngữ C và Java. Cả

hai công cụ hỗ trợ thực hiện cho một vài phần mềm mã nguồn mở bao gồm cả thư viện java.util của SUN JDK.

5.3. EXE và KLEE

EXE [8] là công cụ thực thi tượng trưng cho ngôn ngữ C được thiết kết để kiểm tra một cách toàn diện cho những phần mềm phức tạp với trong tâm là mã nguồn hệ thống để giải quyết sự phức tạp của mã hệ thống. EXE mô hình hóa bộ nhớ ở mức độ chính xác tới mức bit. EXE cung cấp tốc độ để nhanh chóng giải các ràng buộc được sinh ra bởi mã nguồn. Trong thực tế thông qua sự kết hợp tối ưu hóa ở mức thấp các thành phần SPT [8,13] và một loạt các thành phần ở mức cao như bộ nhớ đệm và loại bỏ các ràng buộc không liên quan.

KLEE [4] là một thiết kế lại của EXE được xây dựng lại trên nền biên dịch LLVM. Giống như EXE thực hiện trộn lẫn các giá trị cụ thể và giá trị tượng trưng mà mô hình hóa bộ nhớ với mức độ chính xác ở mức Bit. KLEE sử dụng nhiều tối ưu hóa bộ giải ràng buộc và sử dụng kỹ thuật tìm kiếm Heuristic để đạt được độ bao phủ cao của mã nguồn.

Một trong những cải tiến quan trọng của KLEE hơn EXE là khả năng lưu trữ lớn hơn những trạng thái tượng trưng thực hiện bởi khai thác sự chia sẻ giữa các trạng thái và đối tượng. Một cải tiến quan trọng nữa là nâng cao khả năng giải quyết vấn đề tương tác với môi trường bên ngoài hoặc được cung cấp bởi thiết kế mô hình để khám phá tất cả những tương tác hợp lệ với thế giới bên ngoài. Những tính năng này của EXE và KLEE đã rất thành công khi sử dụng để kiểm thử một số lượng lớn các phần mềm hệ thống khác nhau bao gồm những hệ thống mạng máy chủ, tệp tin hệ thống, thiết bị điều khiển và các thư viện chuẩn.

Trong nghành công nghệ phần mềm cũng bắt đầu áp dụng thực thi tượng trưng ví dụ hãng Microsoft đã phát triển SAGE [12] là công cụ thực thi tượng trưng động và nó đã phát hiện ra một số lỗi quan trọng trong các ứng dụng lớn của hệ điều hành Windows như là bộ xử lý ảnh và giải mã tập tin [17]. Cũng như PEX [38] là công cụ thực thi tượng trưng cho mã nguồn .NET đã được tích hợp trong add-in của Visual

Studio. Một vài công ty lớn như NASA [4] IBM [1] và Fujitsu cũng đã phát triển công nghệ này để áp dụng cho những phần mềm của họ.

Trong khi sự tác động của thực thi tượng trưng vẫn còn hạn chế, các nghiên cứu theo dòng cải tiến trong lĩnh vực này sẽ làm gia tăng khả năng áp dụng và đưa vào thực tế.

6. KẾT LUẬN

Thực thi tượng trưng đã trở thành kỹ thuật hiệu quả trong kiểm thử chương trình, nó cung cấp một phương pháp tự động sinh dữ liệu kiểm thử để tìm ra những lỗi của phần mềm ở các cấp độ khác nhau. Những công cụ hiện có đã cung cấp các kỹ thuật hiệu quả trong kiểm thử và tìm được những lỗi trong nhiều dạng phần mềm khác nhau như: phần mềm điều khiển hệ thống mạng, hệ điều hành đến những ứng dụng ở mức cao.

Bài báo đã trình bày một cách khái quát về thực thi tượng trưng và các công cụ hiện nay. Đóng góp chính của bài báo là: Đánh giá ưu nhược điểm của mỗi kỹ thuật đồng thời đưa ra một số thách thức mà thực thi tượng trưng đang gặp phải. Bài báo cũng đưa ra một vài giải pháp khắc phục các khó khăn trên.

Trong thời gian tới, dựa trên kết quả nghiên cứu đã đạt được, nhóm nghiên cứu có thể đưa ra đề xuất cho các kỹ thuật kiểm thử mới có độ chính xác, tính đầu đủ và hiệu quả hơn.

TÀI LIỆU THAM KHẢO

[1] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. SIGPLAN Not., 10:234–245, (1975).

[2] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In ASE’11, (2011).

[3] H. Zhu, P. Hall, J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys, 29 (4). ISSN 0360-0300*, December, pp. 366–427, (1997).

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI’08, Dec (2008).

[5] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In ASE'08, Sept. (2008).

[6] D. Bird and C. Munoz, "Automatic Generation of Random Self- Checking Test Cases," IBM Systems Journal, vol. 22, no. 3, pp. 229–245, (1983).

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In SPIN'05, Aug (2005).

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In CCS'06, Oct–Nov 2006. An extended version appeared in ACM TISSEC 12:2, (2008).

[9] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In EuroSys'11, Apr (2011).

[10] L. A. Clarke. A program testing system. In Proc. of the 1976 annual conference, pages 488–491, (1976).

[11] L. De Moura and N. Bjorner. Satisfiability modulo theories: introduction and applications. Commun. ACM, 54:69–77, Sept. (2011).

[12] B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In ISSTA'09, (2009).

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In CAV'07, July (2007).

[14] P. Godefroid. Compositional dynamic test generation. In POPL'07, Jan. (2007).

[15] P. Godefroid. Higher-order test generation. In PLDI'11, (2011).

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In PLDI'05, June (2005).

[17] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In NDSS'08, Feb. (2008).

[18] W. Howden. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering, 3(4):266–278, (1977).

[19] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In ASE'08, (2008).

[20] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng., 37(5):649–678, (2011).

[21] J. C. King. Symbolic execution and program testing. Commun. ACM, 9:385–394, July (1976).

[22] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. J. Systems and Software, 83(12):2379–91, (2010).

[23] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, Mar (2007).

[24] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In FASE'06, (2006).

[25] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In CAV'06, (2006).

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In ESEC/FSE'05, Sep (2005).

[27] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitnessguided path exploration in dynamic symbolic execution. In DSN'09, pages 359–368, (2009).

[28] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In TAP'08, Apr (2008).

[29] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolicexecution. SIGPLAN Not., 10:234–245, (1975).

[30] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI'08, Dec (2008).Web site

[31] http://msdn2.microsoft.com/vs2008/

[32] http://www.junit.org/

[33] http://www.nunit.org/

[34] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

SYMBOLIC EXECUTION IN AUTOMATICALLY GENERATIONOF DATA OF SOFTWARE TESTING

To Huu Nguyen[a], Nguyen Hong Tan[a], Ha Thi Thanh[a], Do Thanh Mai[b]

[a]Information and Communication Technology University, Thainguyen University, Thainguyen, Vietnam
[b]School of Foreign Languages, Thainguyen University, Thainguyen, Vietnam
[*]Corresponding author: thnguyen@ictu.edu.vn

Article history
Received: January 04th, 2016
Received in revised form: March 10th, 2016
Accepted: March 16th, 2016

Abstract

In software testing area, all test cases are often generated manually. It time-consumming and costly to complete such task. Symbolic execution is a well-known technique for automatically making the test cases that cover almost every testing criteria in order to deeply discover some errors in complex software systems.

In this paper, we present some general points and several results in recent researches about symbolic execution technique. This paper also shows different challenges that need setting in this field such as the explosion of execution paths in a program, the ability of constraint solving, memory modelling or concurrent problems etc. The evaluation of published results is given in this paper as well.

Keywords: Constraint solving; Dynamic symbolic execution; Symbolic execution; Testing data; Testing data generation.