

**BỘ GIÁO DỤC VÀ ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC QUY NHƠN**

---

**ĐỒ ĐĂNG KHOA**

**ĐỘ TƯƠNG TỰ HÀNH VI CỦA CHƯƠNG TRÌNH  
VÀ THỰC NGHIỆM**

**LUẬN VĂN THẠC SĨ KHOA HỌC MÁY TÍNH**

**Bình Định – Năm 2018**

**BỘ GIÁO DỤC VÀ ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC QUY NHƠN**

---

**ĐỒ ĐĂNG KHOA**

**ĐỘ TƯƠNG TỰ HÀNH VI CỦA CHƯƠNG TRÌNH  
VÀ THỰC NGHIỆM**

**Chuyên ngành: Khoa học máy tính  
Mã số: 8 48 01 01**

**Người hướng dẫn: TS. PHẠM VĂN VIỆT**

# LỜI CAM ĐOAN

Tôi xin cam đoan: Luận văn này là công trình nghiên cứu thực sự của cá nhân, được thực hiện dưới sự hướng dẫn khoa học của TS. Phạm Văn Việt.

Các số liệu, những kết luận nghiên cứu được trình bày trong luận văn này trung thực và chưa từng được công bố dưới bất cứ hình thức nào.

Tôi xin chịu trách nhiệm về nghiên cứu của mình.

# LỜI CẢM ƠN

Tôi xin chân thành cảm ơn sự hướng dẫn, chỉ dạy và giúp đỡ tận tình của các Thầy Cô giảng dạy Sau đại học - Trường đại học Quy Nhơn.

Đặc biệt, tôi cảm ơn thầy Phạm Văn Việt, giảng viên bộ môn Công nghệ phần mềm, khoa Công nghệ thông tin, Trường Đại học Quy Nhơn đã tận tình hướng dẫn truyền đạt kiến thức và kinh nghiệm quý báu giúp tôi có đầy đủ kiến thức hoàn thành luận văn này.

Và tôi xin cảm ơn bạn bè, những người thân trong gia đình đã tin tưởng, động viên tôi trong quá trình học tập và nghiên cứu đề tài này.

Mặc dù đã tôi đã cố gắng trong việc thực hiện luận văn, song đề tài này không thể tránh khỏi những thiếu sót và chưa hoàn chỉnh. Tôi rất mong nhận được ý kiến đóng góp của quý Thầy, Cô và các bạn để đề tài được hoàn thiện hơn.

Một lần nữa, tôi xin chân thành cảm ơn!

**HỌC VIÊN**

**Đỗ Đăng Khoa**

# Mục lục

Mục lục	5
1 GIỚI THIỆU	7
2 TƯƠNG TỰ VỀ HÀNH VI GIỮA CÁC CHƯƠNG TRÌNH	8
2.1 Kiến thức cơ sở . . . . .	8
2.2 Hành vi của chương trình . . . . .	22
2.3 Một số phép đo độ tương tự hành vi . . . . .	26
3 THỰC NGHIỆM, ĐÁNH GIÁ	34
Tài liệu tham khảo	35

## DANH MỤC CÁC CHỮ VIẾT TẮT

Ký hiệu	Diễn giải
RS	<i>Random Sampling</i>
SSE	<i>Singleprogram Symbolic Execution</i>
PSE	<i>Paired-program Symbolic Execution</i>
DSE	Dynamic symbolic execution
MOOC	Massive Open Online Courses
UT	Unit Test

# Chương 1

## GIỚI THIỆU

## Chương 2

# TƯƠNG TỰ VỀ HÀNH VI GIỮA CÁC CHƯƠNG TRÌNH

Chương này tập trung làm rõ hành vi của chương trình, độ tương tự về hành vi giữa hai chương trình và các phương pháp đo độ tương tự về hành vi, sẽ được trình bày trong Phần 2.2 và 2.3. Ngoài ra, chương này còn trình bày một số kiến thức cơ sở về kiểm thử phần mềm, các phương pháp kiểm thử phần mềm, kỹ thuật sinh dữ liệu kiểm thử trong Phần 2.1.

### 2.1 Kiến thức cơ sở

Ý tưởng chính của việc đo độ tương tự về hành vi giữa hai chương trình máy tính là dựa trên tập dữ liệu vào, đếm số lượng dữ liệu ra tương ứng giống nhau giữa hai chương trình và đo tỷ lệ tương tự. Dữ liệu vào phải được chọn sao cho phủ nhiều nhất miền vào của chương trình. Về cơ bản, độ tương tự này được đo dựa trên dữ liệu các ca kiểm thử. Chương này trình bày ngắn gọn về kiểm thử phần mềm cùng việc sinh dữ liệu kiểm thử, chủ yếu tập trung vào phương pháp thực thi biểu trưng một chương trình (*DSE – Dynamic Symbolic Execution*) nhằm giúp tăng độ phủ của dữ liệu thử.

### Kiểm thử phần mềm

Hiện nay, ngành công nghiệp phần mềm giữ vai trò hết sức quan trọng, một số nước có nền công nghệ thông tin phát triển thì ngành công nghiệp phần mềm có khả năng chi phối cả nền kinh tế. Vì vậy, việc đảm bảo chất lượng phần mềm trở nên cần thiết hơn



bao giờ hết.

Quá trình phát hiện và khắc phục lỗi của phần mềm là một công việc đòi hỏi nhiều nỗ lực, công sức, phát sinh thêm nhiều chi phí trong việc phát triển phần mềm. Một sản phẩm phần mềm đạt chất lượng cao, đáp ứng được yêu cầu của người sử dụng sẽ được nhiều người biết đến, nó mang lại hiệu quả tích cực trong công việc của người sử dụng. Ngược lại, một phần mềm kém chất lượng sẽ gây thiệt hại về kinh tế, ảnh hưởng đến công việc của người sử dụng. Vì vậy, yêu cầu đặt ra đó là một sản phẩm phần mềm phải đảm bảo được sự ổn định, không phát sinh lỗi trong quá trình sử dụng.

Kiểm thử phần mềm chính là một quá trình hoặc một loạt các quy trình được thiết kế, để đảm bảo mã máy tính chỉ làm những gì nó được thiết kế và không làm bất cứ điều gì ngoài ý muốn [41]. Phần mềm phải được dự đoán và nhất quán, không gây bất ngờ cho người dùng. Đây là một bước quan trọng trong quá trình phát triển một phần mềm, giúp cho nhà phát triển phần mềm và người sử dụng thấy được hệ thống đã đáp ứng được yêu cầu đặt ra.

### **Các phương pháp kiểm thử**

Có nhiều phương pháp để kiểm thử phần mềm, trong đó hai phương pháp kiểm thử chính là *kiểm thử tĩnh* và *kiểm thử động*.

*Kiểm thử tĩnh (Static testing)* là phương pháp kiểm thử phần mềm bằng cách duyệt lại các yêu cầu và các đặc tả, mã lệnh chương trình bằng tay, thông qua việc sử dụng giấy, bút để kiểm tra tính logic từng chi tiết mà không cần chạy chương trình. Kiểu kiểm thử này thường được sử dụng bởi chuyên viên thiết kế, người viết mã lệnh chương trình. Kiểm thử tĩnh cũng có thể được tự động hóa bằng cách thực hiện kiểm tra toàn bộ hệ thống thông qua một trình thông dịch hoặc trình biên dịch, xác nhận tính hợp lệ về cú pháp của chương trình.

*Kiểm thử động (Dynamic testing)* là phương pháp kiểm thử thông qua việc thực thi

chương trình để kiểm tra trạng thái tác động của chương trình, dựa trên các ca kiểm thử xác định các đối tượng kiểm thử của chương trình. Đồng thời, kiểm thử động sẽ tiến hành kiểm tra cách thức hoạt động của mã lệnh, tức là kiểm tra phản ứng từ hệ thống với các biến thay đổi theo thời gian. Trong kiểm thử động, phần mềm phải được biên dịch và chạy, và bao gồm việc nhập các giá trị đầu vào và kiểm tra giá trị đầu ra có như mong muốn hay không.

Trong luận văn này, độ tương tự về hành vi giữa hai chương trình được đo thông qua việc thực thi hai chương trình, tức là sử dụng phương pháp kiểm thử động.

### Các chiến lược kiểm thử

Hai chiến lược kiểm thử phần mềm được sử dụng nhiều nhất đó là *kiểm thử hộp đen* và *kiểm thử hộp trắng*.

*Kiểm thử hộp đen* – *Black box* là một chiến lược kiểm thử với cách thức hoạt động chủ yếu dựa vào hướng dữ liệu inputs/outputs của chương trình, xem chương trình như là một “hộp đen”. Chiến lược kiểm thử này hoàn toàn không quan tâm về cách xử lý và cấu trúc bên trong của chương trình, nó tập trung vào tìm các trường hợp mà chương trình không thực hiện theo các đặc tả. Tuy nhiên, phương pháp kiểm thử này cũng có mặt hạn chế của nó, kiểm thử viên không biết các phần mềm cần kiểm tra thực sự được xây dựng như thế nào, cố gắng viết rất nhiều ca kiểm thử để kiểm tra một chức năng của phần mềm nhưng lẽ ra chỉ cần kiểm tra bằng một vài ca kiểm thử, hoặc một số phần của chương trình có thể bị bỏ qua không được kiểm tra.

Do vậy, kiểm thử hộp đen có ưu điểm là đánh giá khách quan, mặt khác nó lại có nhược điểm là thăm dò mù. Trong phần nghiên cứu của đề tài, kiểm thử hộp đen cũng được sử dụng như một phương pháp đo độ tương tự hành vi của các chương trình.

*Kiểm thử hộp trắng* – *White box* là một chiến lược kiểm thử khác, trái ngược hoàn toàn với kiểm thử hộp đen, còn gọi là kiểm thử hướng logic của phần mềm. Cách kiểm thử

này cho phép tạo ra dữ liệu thử nghiệm từ việc kiểm tra, khảo sát cấu trúc bên trong và kiểm thử tính logic của chương trình. Dữ liệu thử nghiệm có độ phủ lớn, đảm bảo tất cả các đường dẫn, hoặc các nhánh của chương trình được thực hiện ít nhất một lần, khắc phục được những nhược điểm thăm dò mù trong cách kiểm thử hộp đen.

## Sinh dữ liệu kiểm thử

Sinh ngẫu nhiên dữ liệu thử là một kỹ thuật kiểm thử phần mềm Black-Box, kỹ thuật này tạo ra ngẫu nhiên các giá trị đầu vào và thực thi từng giá trị đầu vào trên chương trình được kiểm thử. Kết quả đầu ra của chương trình được so sánh với các thông số kỹ thuật của phần mềm, xác định đầu ra thử nghiệm thành công hoặc không thành công [41].

Kỹ thuật sinh ngẫu nhiên dữ liệu thử không quan tâm đến hành vi và cấu trúc bên trong của chương trình, chỉ tập trung tìm kiếm những trường hợp chương trình không hoạt động theo đặc tả kỹ thuật của chương trình. Trong phương pháp này, dữ liệu thử nghiệm được tạo ngẫu nhiên từ các đặc tả kỹ thuật của phần mềm (tức là không liên quan tới hành vi và cấu trúc của chương trình).

Mã lệnh 2.1: Hàm sinh ngẫu nhiên dữ liệu thử

---

```
int myAbs(int x)
{
    if (x > 0) {
        return x;
    }
    else {
        return x;
    }
}
```

```
void testAbs(int n)
{
    for (int i = 0; i < n; i++)
    {
        int x = getRandomInput();
        int result = myAbs(x);
        assert(result >= 0);
    }
}
```

---

Mã lệnh 2.1 là một hàm sinh ngẫu nhiên dữ liệu thử, chúng ta thấy hàm *testAbs* chỉ thực hiện việc tạo giá trị đầu vào ngẫu nhiên *int x* theo đặc tả tham số đầu vào của chương trình *myAbs*, và kiểm tra kết quả đầu ra của chương trình *assert(result >= 0)*, không quan tâm hành vi và cấu trúc bên trong của hàm *myAbs*.

### Ưu điểm, hạn chế, hướng khắc phục

Sinh ngẫu nhiên dữ liệu thử có một số ưu điểm và nhược điểm như sau:

\* *Ưu điểm:*

- Đơn giản, dễ dàng sinh các đầu vào ngẫu nhiên
- Không tốn nhiều tài nguyên bộ nhớ lúc thực thi

\* *Hạn chế:*

- Một nhánh hành vi của chương trình được kiểm thử nhiều lần với nhiều đầu vào khác nhau
- Có thể một số nhánh hành vi của chương trình bị bỏ qua
- Khó xác định khi nào việc kiểm thử nên dừng lại

- Không biết dữ liệu thử có duyệt được tất cả các nhánh trong chương trình hay không

\* *Hướng khắc phục:* Để xác định khi nào việc kiểm thử dừng lại, hệ thống kiểm thử ngẫu nhiên có thể kết hợp với kỹ thuật Adequacy Criterion [68]. Kỹ thuật Adequacy Criterion là một kỹ thuật yêu cầu duyệt tất cả các nhánh của chương trình, bằng việc kết hợp này cho phép việc kiểm thử chỉ dừng lại khi tất cả các câu lệnh của chương trình được thực thi ít nhất một lần.

## Kỹ thuật Dynamic symbolic execution

Dynamic symbolic execution (DSE) là một kỹ thuật duyệt tự động tất cả các đường đi có thể của chương trình bằng cách chạy chương trình với nhiều giá trị đầu vào khác nhau để tăng độ phủ của dữ liệu thử [67].

Dựa trên các tham số đầu vào của chương trình, DSE sẽ tạo ra các giá trị đầu vào cụ thể và thực thi chương trình với các giá trị cụ thể này. Trong quá trình thực thi, DSE sẽ ghi nhận lại ràng buộc tại các nút, phủ định lại các ràng buộc này và sinh các giá trị đầu vào thỏa điều kiện ràng buộc tại các nút rẽ nhánh này. Với một giá trị đầu vào cụ thể, DSE sẽ thực thi chương trình và duyệt được một đường đi cụ thể, quá trình thực thi này sẽ lặp lại cho đến khi duyệt hết tất cả các đường đi của chương trình.

---

### Algorithm 1 DSE

---

Set $J := \emptyset$	▷ $J$ : Tập hợp các đầu vào của chương trình phân tích
loop	
Chọn đầu vào $i \notin J$	▷ Dừng lại nếu không có $i$ nào được tìm thấy
Xuất ra $i$	
Thực thi $P(i)$ ; lưu lại điều kiện đường đi $C(i)$ ; suy ra $C'(i)$	
Đặt $J := J \cup i$	
end loop	

---

```

int foo(int v) {
    return 2*v;
}

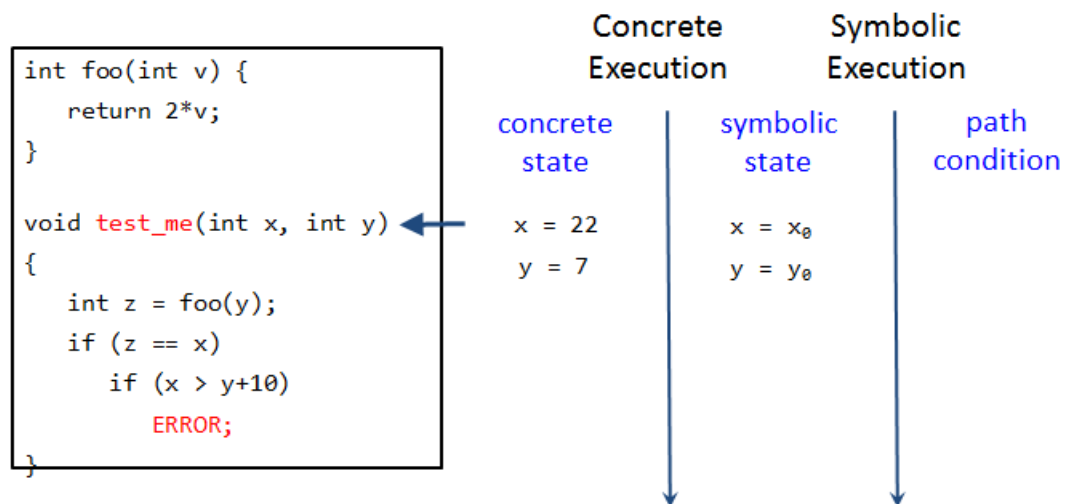
void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}

```

---

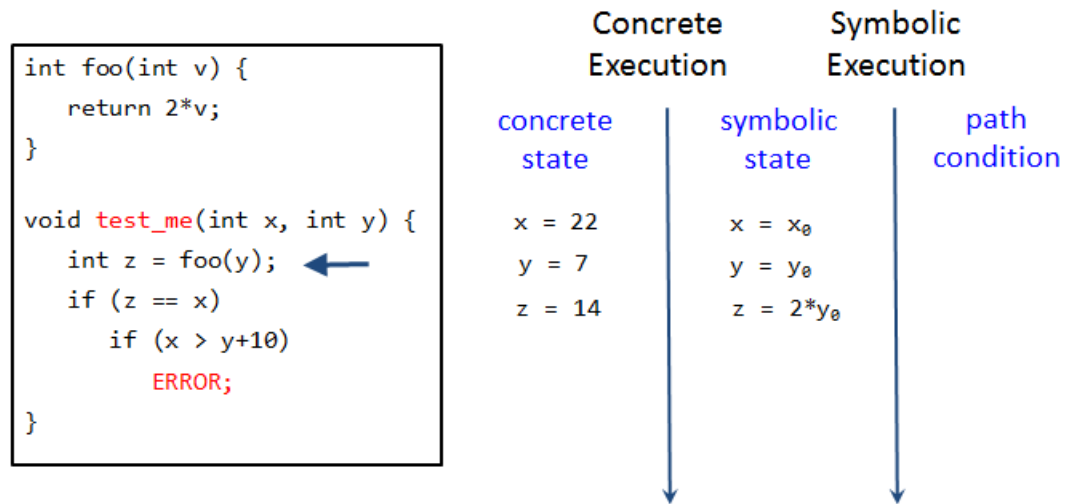
Trong mã lệnh 2.2, chúng ta xem xét hàm `test_me` với hai tham số đầu vào là `int x` và `int y`, và hàm này không có giá trị trả về. Cách thức làm việc của DSE trên hàm `test_me` như sau:

Đầu tiên, DSE tạo hai giá trị đầu vào thử nghiệm ngẫu nhiên  $x$  và  $y$ , giả sử  $x = 22$  và  $y = 7$ . Ngoài ra, DSE sẽ theo dõi trạng thái các giá trị đầu vào thử nghiệm của chương trình với  $x$  bằng một số  $x_0$  và  $y$  bằng một số  $y_0$ .



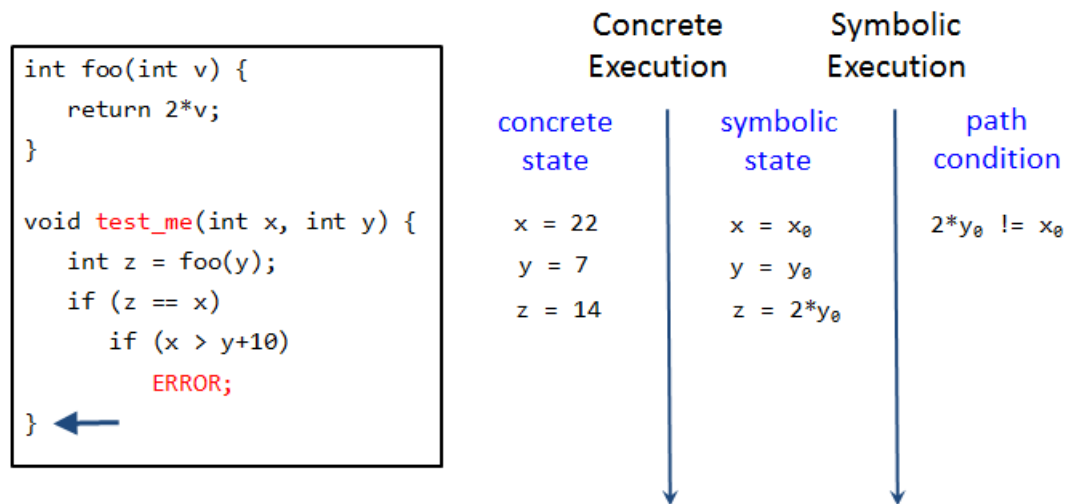
Hình 2.1: DSE khởi tạo các giá trị đầu vào

Ở dòng đầu tiên, số nguyên  $z$  được gán bằng hàm  $foo(y)$ . Điều này có nghĩa là  $z = 14$ , và ở trạng thái tương trưng, biến  $z = 2 * y_0$ .



Hình 2.2: Số nguyên  $Z$  được gán bằng hàm  $foo(y)$

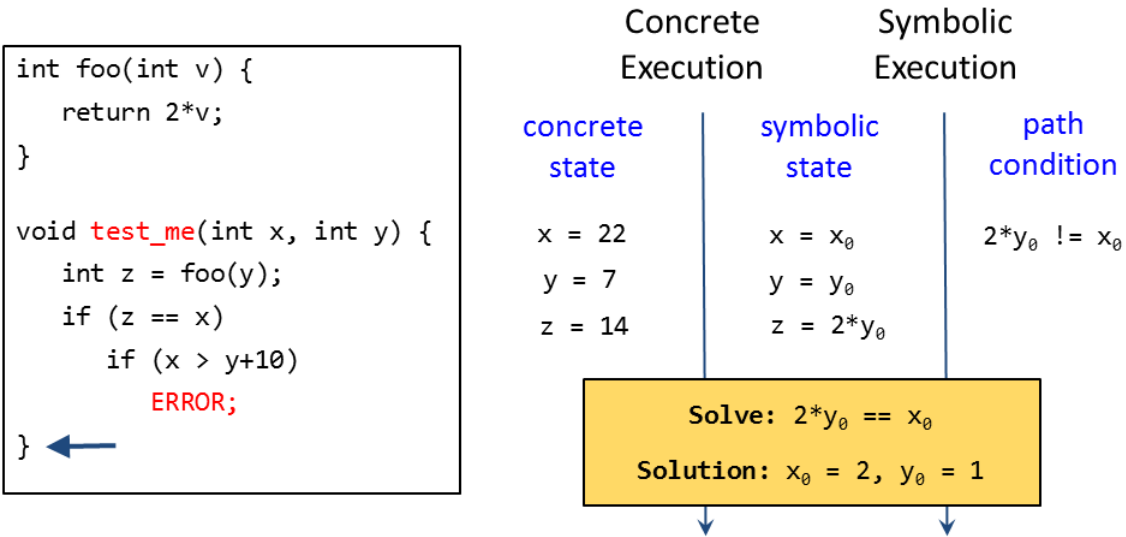
Tại nhánh  $z == x$ , DSE nhận biết giá trị của  $z$  không bằng giá trị của  $x$  và lưu trữ ràng buộc này là  $z \neq x$ , và giá trị tương trưng của đường này là:  $2 * y_0 \neq x_0$ . DSE đi theo nhánh *false* dẫn đến kết thúc chương trình.



Hình 2.3: Với điều kiện  $z == x$ , giá trị của  $z \neq x$  nên DSE chạy theo nhánh *false*

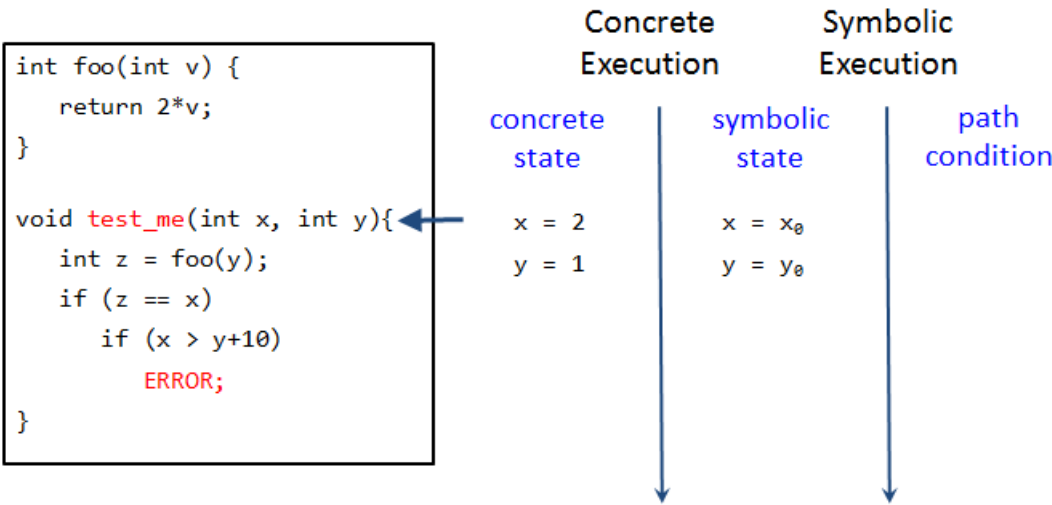
Sau khi kết thúc chương trình, DSE sẽ quay trở lại điểm nhánh gần nhất và chọn nhánh *true*. Với mục đích này, nó phủ định ràng buộc được thêm gần nhất trong điều kiện đường dẫn  $2 * y_0 \neq x_0$  thành  $2 * y_0 = x_0$ . Để thỏa mãn ràng buộc  $2 * y_0 \neq x_0$  thì hai số nguyên này sẽ là  $x_0 = 2$  và  $y_0 = 1$ .





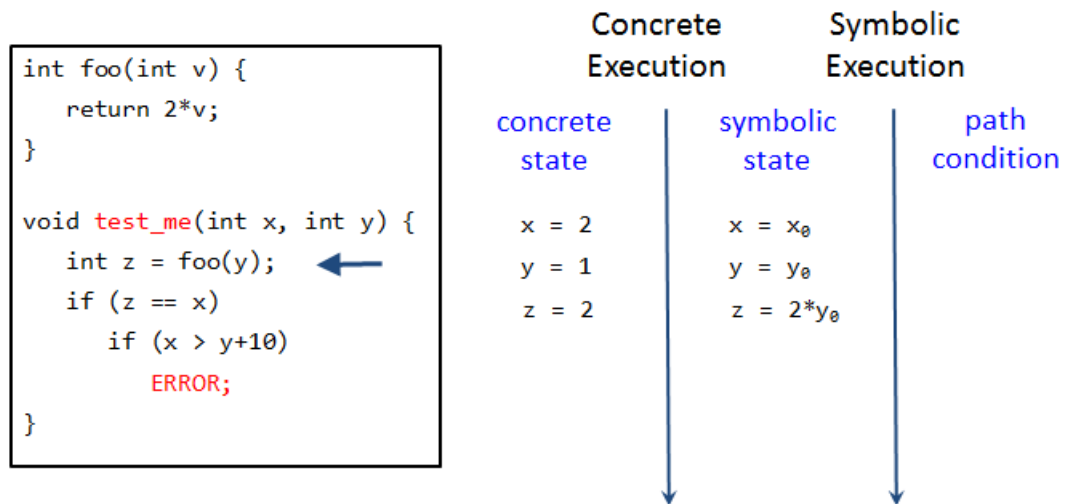
Hình 2.4: Các giá trị DSE sinh ra sau khi thực thi chương trình lần 1

Sau đó, DSE khởi động lại hàm `test_me`, lần này nó gọi các giá trị đầu vào cụ thể với giá trị:  $x = 2$  và  $y = 1$  được tạo ra bởi quá trình giải quyết ràng buộc trước đó. DSE tiếp tục theo dõi trạng thái các biến với  $x = x_0$  và  $y = y_0$ .



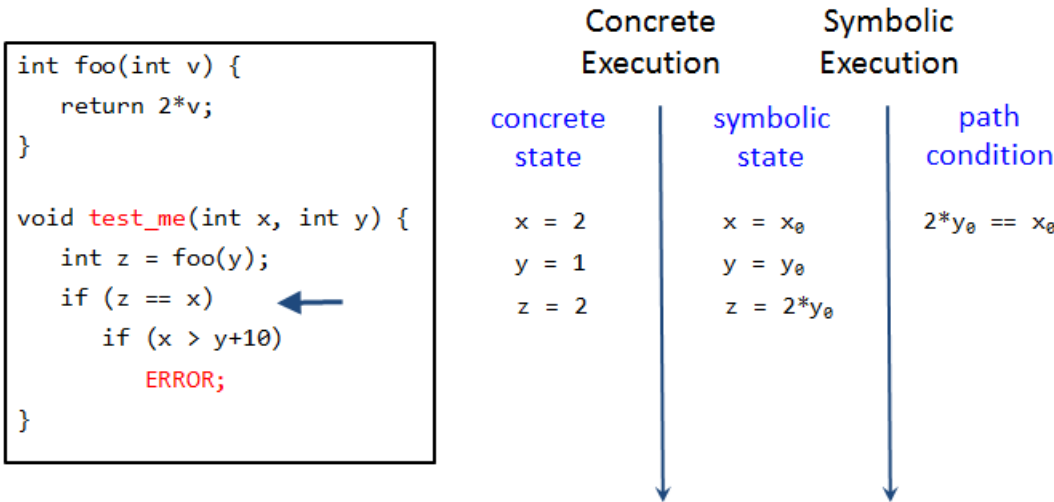
Hình 2.5: DSE khởi động lại hàm `test_me`

Sau khi thực hiện dòng đầu tiên,  $z$  có giá trị cụ thể 2 và giá trị biểu tượng  $2 * y_0$ .



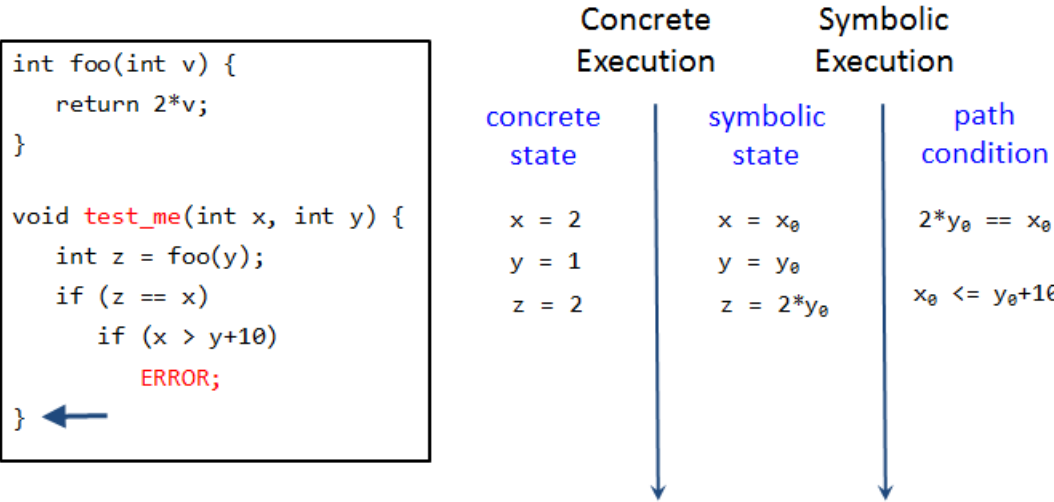
Hình 2.6: Thực hiện dòng đầu tiên  $int z = foo(y)$

Ở dòng kế tiếp, chúng ta kiểm tra tình trạng nhánh  $z == x$ . Trong trường hợp này, điều kiện là đúng vì vậy điều kiện đường dẫn của chúng ta trở thành  $2 * y_0 == x_0$ . Sau đó DSE kiểm tra dòng tiếp theo của nhánh *true*.



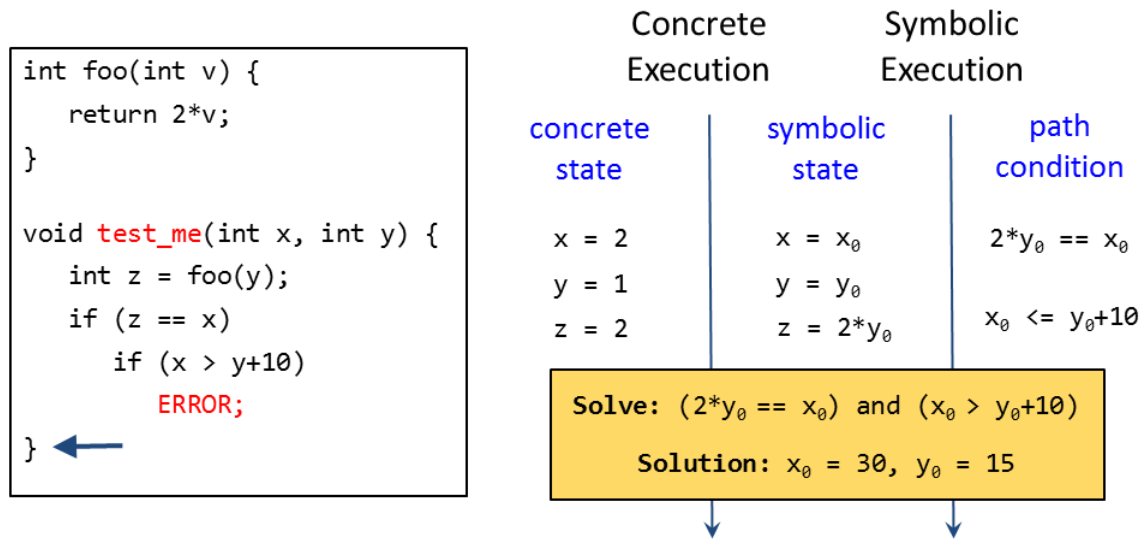
Hình 2.7: DSE thực hiện điều kiện đường dẫn *true*

Tại điểm nhánh tiếp theo,  $x$  có giá trị cụ thể là 2 và  $y + 10$  có giá trị cụ thể là 11, vì vậy DSE lấy nhánh *false*, kết thúc chương trình. Thêm ràng buộc tương trưng  $x_0 \leq y_0 + 10$  vào điều kiện đường dẫn, đây là sự phủ định của điều kiện nhánh mà DSE phát hiện là *false*.



Hình 2.8: DSE lấy nhánh *false* kết thúc chương trình

Vì DSE đã đến cuối chương trình, nó phủ nhận rằng buộc vừa được thêm gần nhất trong điều kiện đường dẫn để có được  $x_0 > y_0 + 10$ , và sau đó nó vượt qua các ràng buộc  $2 * y_0 == x_0$  và  $x_0 > y_0 + 10$ . Để thỏa những ràng buộc này, DSE trả về  $x_0 = 30$  và  $y_0 = 15$ .

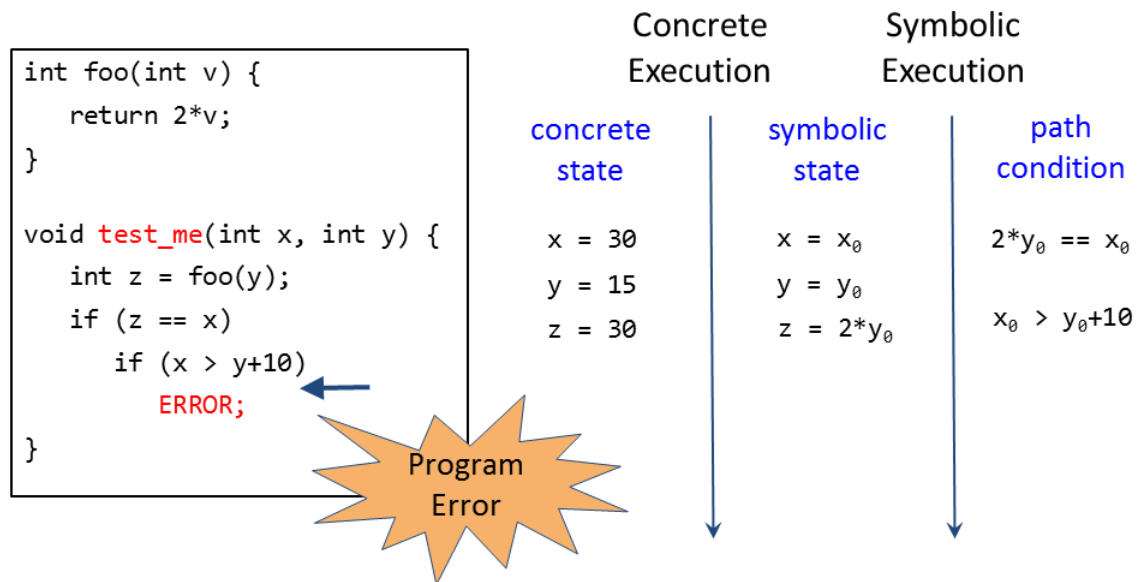


Hình 2.9: Các giá trị DSE sinh ra sau khi thực thi chương trình lần 2

Bây giờ, DSE chạy hàm `test_me` một lần nữa, lần này với các đầu vào  $x = 30$  và  $y = 15$ . Trạng thái biểu tượng các biến bắt đầu  $x = x_0$  và  $y = y_0$ .  $z$  được gán giá trị cụ thể là 30, trong khi giá trị tượng trưng của nó là  $2 * y_0$  như những lần chạy trước.

Khi tới điều kiện rẽ nhánh  $z == x$ , DSE nhận thấy đây là điều kiện *true*, vì vậy DSE thêm điều kiện tượng trưng  $2 * y_0 == x_0$ .

Sau đó tại điểm nhánh tiếp theo, với  $x > y + 10$  vì vậy DSE thêm ràng buộc tượng trưng mới  $x_0 > y_0 + 10$ . Nhánh này dẫn đến *ERROR*, tại thời điểm đó chúng ta đã xác định được đầu vào cụ thể làm cho chương trình dẫn đến *ERROR* là:  $x = 30$  và  $y = 15$ .



Hình 2.10: Các giá trị DSE sinh ra sau khi thực thi chương trình lần 3

Kết quả, sau 3 lần chạy chương trình, DSE tạo ra được các cặp giá trị đầu vào có thể duyệt hết các nhánh của chương trình test\_me đó là:  $[22, 7]$ ,  $[2, 1]$ ,  $[30, 15]$

## Một số công cụ ứng dụng DSE

Trên thế giới hiện có nhiều công cụ sử dụng kỹ thuật DSE để giải quyết các ràng buộc và tạo ra các giá trị đầu vào có độ phủ cao như Pex [58] và SAGE [22]... và những công cụ này được phát triển để có thể chạy được trên nhiều nền tảng khác nhau. Chúng ta có thể tham khảo một số công cụ sau đây:

Tên Công cụ	Ngôn ngữ	Url
KLEE	LLVM	<a href="http://klee.github.io/">klee.github.io/</a>
JPF	Java	<a href="http://babelfish.arc.nasa.gov/trac/jpf">babelfish.arc.nasa.gov/trac/jpf</a>
jCUTE	Java	<a href="https://github.com/osl/jcute">github.com/osl/jcute</a>
janala2	Java	<a href="https://github.com/ksen007/janala2">github.com/ksen007/janala2</a>
JBSE	Java	<a href="https://github.com/pietrobraione/jbse">github.com/pietrobraione/jbse</a>
KeY	Java	<a href="http://www.key-project.org/">www.key-project.org/</a>
Mayhem	Binary	<a href="http://forallsecure.com/mayhem.html">forallsecure.com/mayhem.html</a>
Otter	C	<a href="http://bitbucket.org/khooy/otter/overview">bitbucket.org/khooy/otter/overview</a>
Rubyx	Ruby	<a href="http://www.cs.umd.edu/~avik/papers/ssarorwa.pdf">www.cs.umd.edu/~avik/papers/ssarorwa.pdf</a>
Pex	.NET Framework	<a href="http://research.microsoft.com/en-us/projects/pex/">research.microsoft.com/en-us/projects/pex/</a>
Jalangi2	JavaScript	<a href="https://github.com/Samsung/jalangi2">github.com/Samsung/jalangi2</a>
Kite	LLVM	<a href="http://www.cs.ubc.ca/labs/isd/Projects/Kite/">www.cs.ubc.ca/labs/isd/Projects/Kite/</a>
pysymemu	x86-64 / Native	<a href="https://github.com/feliam/pysymemu/">github.com/feliam/pysymemu/</a>
Triton	x86 and x86-64	<a href="http://triton.quarkslab.com">triton.quarkslab.com</a>
BE-PUM	x86	<a href="https://github.com/NMHai/BE-PUM">https://github.com/NMHai/BE-PUM</a>

## 2.2 Hành vi của chương trình

Để định lượng hai chương trình tương tự nhau, chúng ta nghiên cứu một số định nghĩa liên quan đến hành vi của hai chương trình như sau:

### Thực thi chương trình

**Định nghĩa 1** Cho  $P$  là một chương trình,  $I$  là tập hợp các trị đầu vào của  $P$  và  $O$  là tập hợp các giá trị đầu ra của  $P$ . Thực thi chương trình  $P$  là ánh xạ  $exec : P \times I \rightarrow O$ . Với giá trị đầu vào  $i \in I$ , sau khi thực thi  $P$  trên  $i$  ta có giá trị đầu ra tương ứng  $o \in O$  và ký hiệu  $o = exec(P, i)$ .

## Độ tương đương về hành vi (Behavioral Equivalence)

Dựa trên định nghĩa về thực thi chương trình, chúng ta tìm hiểu thế nào là độ tương đương về hành vi giữa hai chương trình thông qua hai chương trình minh họa sau:

Mã lệnh 2.3: Switch...Case

```
public static int TinhY(int x)
{
    y = 0;
    switch (x)
    {
        case 1: y += 4; break;
        case 2: y *= 2; break;
        default: y = y * y;
    }
    return y;
}
```

Mã lệnh 2.4: If...Else

```
public static int TinhY(int x)
{
    y = 0;
    if (x == 1)
        y += 4;
    else if (x == 2)
        y *= 2;
    else y = y * y;
    return y;
}
```

Mã lệnh 2.3 và 2.4 có tham số đầu vào cùng kiểu giá trị *int*. Mã lệnh 2.3 sử dụng cấu trúc *switch...case*, mã lệnh 2.4 sử dụng cấu trúc *if...else* để kiểm tra giá trị đầu vào *x*. Mặc dù cú pháp sử dụng trong hai chương trình là khác nhau nhưng cách thức xử lý trả về kết quả *y* là như nhau. Từ đó, chúng ta có thể định nghĩa thế nào là độ tương đương hành vi giữa hai chương trình như sau:

**Định nghĩa 2 (Độ tương đương về hành vi)** Cho  $P_1$  và  $P_2$  là hai chương trình có cùng miền các giá trị đầu vào  $I$ . Hai chương trình này được gọi là tương đương khi và chỉ khi thực thi của chúng giống nhau trên mọi giá trị đầu vào trên  $I$ , ký hiệu là  $exec(P_1, I) = exec(P_2, I)$ .

## Sự khác biệt về hành vi (Behavioral Difference)

Để tìm hiểu sự khác biệt về hành vi của hai chương trình, chúng ta tìm hiểu hai mã lệnh sau:

Mã lệnh 2.5: Switch...Case

```
using System;
public class Program
{
    public static int P1(int x)
    {
        return x - 10;
    }
}
```

Mã lệnh 2.6: If...Else

```
using System;
public class Program
{
    public static int P2(int x)
    {
        return x + 10;
    }
}
```

Mã lệnh 2.5 và Mã lệnh 2.6 của Chương trình  $P_1$  và  $P_2$ , cả hai Chương trình có miền giá trị đầu vào cùng kiểu *int*, giá trị trả về của Chương trình  $P_1$  là  $x - 10$ , giá trị trả về của Chương trình  $P_2$  là  $x + 10$ . Với mọi giá trị của  $x$  được thực thi trên cả hai chương trình  $P_1$  và  $P_2$  kết quả trả về sẽ không giống nhau. Mặc dù cả hai Chương trình  $P_1$  và  $P_2$  có miền giá trị đầu vào như nhau, nhưng hành vi của hai Chương trình hoàn toàn khác nhau. Qua đó, chúng ta có thể định nghĩa sự khác biệt về hành vi như sau:

**Định nghĩa 3 (Sự khác biệt hành vi)** Cho  $P_1$  và  $P_2$  là hai chương trình có cùng một miền các giá trị đầu vào  $I$ . Hai chương trình này được xem là có sự khác biệt về hành vi khi và chỉ khi thực thi của chúng khác nhau trên mọi giá trị đầu vào  $I$ , ký hiệu là  $exec(P_1, I) \neq exec(P_2, I)$ .

## Độ tương tự hành vi (Behavioral Similarity)

Để hiểu thế nào là tương tự hành vi, chúng ta phân tích hai Mã lệnh sau:



Mã lệnh 2.7: Chương trình  $P_1$ 

```

using System;
public class Program
{
    public static int P1(int x)
    {
        if (x >= 0 && x <= 100)
            return x + 10;
        else
            return x;
    }
}

```

Mã lệnh 2.8: Chương trình  $P_2$ 

```

using System;
public class Program
{
    public static int P2(int x)
    {
        if (x >= 0 && x <= 100)
            return x + 10;
        else
            return -1;
    }
}

```

Với Mã lệnh 2.7 và Mã lệnh 2.8 của hai Chương trình  $P_1$  và  $P_2$ , chúng ta thấy cả hai Chương trình có giá trị đầu vào cùng kiểu dữ liệu là *int*, nếu giá trị đầu vào của biến  $x$  nằm trong khoảng 0 đến 100 thì giá trị trả về của cả hai Chương trình đều bằng nhau là  $x + 10$ . Ngược lại, giá trị đầu vào của biến  $x$  nằm ngoài khoảng 0 đến 100, thì giá trị trả về của Chương trình  $P_1$  là  $x$  và giá trị trả về của Chương trình  $P_2$  là  $-1$ . Hai Chương trình  $P_1$  và  $P_2$  tuy có kiểu dữ liệu đầu vào như nhau nhưng kết quả đầu ra có thể giống nhau hoặc khác nhau tùy theo giá trị đầu vào của biến  $x$ . Dựa trên kết quả phân tích, chúng ta định nghĩa độ tương tự hành vi của Chương trình như sau:

**Định nghĩa 4 (Độ tương tự hành vi)** Cho  $P_1$  và  $P_2$  là hai chương trình có cùng miền giá trị đầu vào  $I$ , và  $I_s$  là tập con của  $I$ . Hai Chương trình được xem là tương tự hành vi khi thực thi chúng giống nhau trên mọi giá trị đầu vào  $I_s$ , ký hiệu  $exec(P_1, I_s) = exec(P_2, I_s)$  và khác nhau  $\forall j \in I \setminus I_s$ , ký hiệu  $exec(P_1, j) \neq exec(P_2, j)$

## 2.3 Một số phép đo độ tương tự hành vi

Để đo độ tương tự về hành vi giữa hai chương trình, chúng ta có thể chạy từng giá trị đầu vào trong miền giá trị đầu vào của hai chương trình. Tỷ lệ giữa số lượng đầu vào thử nghiệm khi thực thi trên cả hai chương trình cho kết quả đầu ra giống nhau trên tổng số lượng đầu vào được thử nghiệm là độ tương tự hành vi giữa hai chương trình. Dựa trên cách tính tỷ lệ kết quả đầu ra của hai chương trình, chúng ta có một số phép đo độ tương tự hành vi như sau:

### Phép đo lấy mẫu ngẫu nhiên (RS)

Kỹ thuật của phép đo **RS** là thực hiện lấy mẫu ngẫu nhiên giá trị đầu vào trên miền giá trị đầu vào của hai chương trình. Thực thi cả hai chương trình trên từng giá trị đầu vào, tiến hành so sánh giá trị kết quả đầu ra của cả hai chương trình. Tỷ lệ giữa tổng số mẫu đầu vào khi thực thi những mẫu này hai chương trình cho kết quả đầu ra có giá trị giống nhau, trên tổng số mẫu đầu vào được thử nghiệm là kết quả cho phép đo RS. Từ đó, chúng ta có định nghĩa phép đo **RS** như sau:

**Định nghĩa 5 (Phép đo RS)** Cho  $P_1$  và  $P_2$  là hai chương trình có cùng miền giá trị đầu vào  $I$ ,  $I_s$  là tập con ngẫu nhiên của  $I$ ,  $I_a$  là tập con  $I_s$ . Hai chương trình thực thi giống nhau với  $\forall i \in I_a$ , ký hiệu  $exec(P_1, i) = exec(P_2, i)$  và hai chương trình thực thi khác nhau với  $\forall j \in I_s \setminus I_a$ , ký hiệu  $exec(P_1, j) \neq exec(P_2, j)$ . Chỉ số phép đo **RS** được định nghĩa là  $M_{RS}(P_1, P_2) = |I_a| / |I_s|$ .

**Algorithm 2** Phép đo RS

---

$P_1, P_2$  : Là hai chương trình cần đo độ tương tự  
 $I$ : Miền giá trị đầu vào của  $P_1, P_2$   
Set  $I_s = \text{Random}(I)$  ▷  $I_s$ : Tập con ngẫu nhiên của  $I$   
Set  $I_a = \emptyset$   
**while**  $i \in I_s$  **do** ▷ Vòng lặp dừng lại khi  $\forall i \in I_s$  đã được thực thi  
    **if** ( $\text{exec}(P_1, i) = \text{exec}(P_2, i)$ ) **then**  
         $I_a = I_a \cup i$   
 $M_{RS}(P_1, P_2) = |I_a| / |I_s|$ .

---

Phép đo **RS** là một phép đo đơn giản và hiệu quả để tính độ tương tự của hành vi. Khi miền giá trị của tham số đầu vào rất lớn hoặc vô hạn, phép đo **RS** thực hiện lấy mẫu ngẫu nhiên để tính toán độ tương tự của hành vi, kết quả đầu ra tương đối tốt và hợp lý so với hành vi thực tế của chương trình. Phép đo **RS** xử lý, tính toán độ tương tự hành vi dưới dạng hộp đen và không phân tích chương trình để tạo thử nghiệm, nên tốc độ xử lý nhanh và chiếm ít tài nguyên. Mặc khác, phép đo **RS** không phân tích chương trình để tạo đầu vào thử nghiệm nên phép đo **RS** có thể bỏ qua một vài tham số đầu vào thử nghiệm có thể được sử dụng để thực thi một số nhánh khác nhau giữa hai chương trình. Vì vậy, phép đo **RS** không phân biệt được các chương trình có một số hành vi khác nhau. Chúng ta phân tích Mã lệnh 3.7 và 3.8 sau để thấy được hạn chế của phép đo **RS**:

Mã lệnh 2.9: Chương trình  $P_1$ 


---

```
public static int Y(string x)
{
    if (x == "XYZ")
        return 0;
    if (x == "ABC")
        return 1;
    return -1;
}
```

---

Mã lệnh 2.10: Chương trình  $P_2$ 


---

```
public static int Y(string x)
{
    if (x == "ABC")
        return 1
    return -1;
}
```

---

Chúng ta thấy đoạn Mã lệnh 2.9 và 2.10 của hai Chương trình  $P_1$  và  $P_2$  có cùng miền giá trị đầu vào là *string* $x$ , cấu trúc mã lệnh hai chương trình gần như nhau. Nhưng Chương trình  $P_1$  khác với Chương trình  $P_2$  đó là sẽ trả kết quả về 0 nếu tham số đầu vào có giá trị là XYZ. Tỷ lệ phép đo **RS** lấy ngẫu nhiên giá trị đầu vào  $x$  trên miền giá trị đầu vào của hai chương trình có giá trị bằng XYZ là rất thấp, vì vậy khả năng câu lệnh  $if(x == "XYZ")return 0;$  của Chương trình  $P_1$  có thể sẽ không được thực thi nên kết quả của phép đo **RS** sẽ ở mức tương đối so với hành vi thực tế của chương trình.

### Phép đo tương trưng trên một chương trình Single Program Symbol Execution (SSE)

Phép đo **SSE** là một phép đo dựa trên số lượng các nhánh đường đi của Chương trình mẫu, mỗi nhánh đường đi của Chương trình mẫu được xem là một hành vi của chương trình. Nếu chọn một giá trị đầu vào thử nghiệm cho một nhánh đường đi trong Chương trình thì các giá trị đầu vào thử nghiệm này sẽ khám phá hết các hành vi trong Chương trình mẫu. Do vậy, số phần tử trong tập các giá trị đầu vào thử nghiệm của phép đo **SSE** sẽ nhỏ hơn tập các giá trị đầu vào thử nghiệm được chọn theo phương pháp lấy ngẫu nhiên giá trị đầu vào.

Để tính độ tương tự hành vi của hai chương trình với phép đo **SSE**, chúng ta chọn Chương trình mẫu làm Chương trình tham chiếu và áp dụng kỹ thuật **DSE** để tạo ra các đầu vào thử nghiệm dựa trên Chương trình tham chiếu. Sau đó thực thi cả hai chương trình dựa trên các giá trị đầu vào thử nghiệm. Tỷ lệ số lượng các kết quả đầu ra giống nhau của cả hai chương trình trên tổng số các giá trị đầu vào thử nghiệm của Chương trình tham chiếu là kết quả của phép đo **SSE**. Qua đó, chúng ta có định nghĩa phép đo **SSE** như sau:

**Định nghĩa 6** Cho  $P_1$  và  $P_2$  là hai chương trình có cùng miền giá trị đầu vào  $I$ , Chương trình  $P_1$  là Chương trình tham chiếu,  $I_s$  là tập các giá trị đầu vào được tạo bởi DSE trên chương trình  $P_1$ , và  $I_a$  là tập con  $I_s$ . Hai Chương trình thực thi giống nhau với  $\forall i \in I_a$ , ký hiệu  $exec(P_1, i) = exec(P_2, i)$  và hai Chương trình thực thi khác nhau với  $\forall j \in I_s \setminus I_a$ , ký hiệu  $exec(P_1, j) \neq exec(P_2, j)$ . Chỉ số phép đo **SSE** được định nghĩa là  $M_{SSE}(P_1, P_2) = |I_a| / |I_s|$ .

---

**Algorithm 3** Phép đo SSE
 

---

$P_1, P_2$  : Là hai chương trình cần đo tương tự

$I$ : Miền giá trị đầu vào của  $P_1, P_2$

$P_1$ : Là Chương trình tham chiếu

Set  $I_s = DSE(P_1)$

▷  $I_s$ : Tập đầu vào của  $P_1$  theo DSE

Set  $I_a = \emptyset$

**while** ( $i \in I_s$ ) **do**

▷ Vòng lặp dừng lại khi  $\forall i \in I_s$  đã được thực thi

**if** ( $exec(P_1, i) = exec(P_2, i)$ ) **then**

$I_a = I_a \cup i$

$M_{SSE}(P_1, P_2) = |I_a| / |I_s|$ .

---

Ngược lại với phép đo RS, phép đo SSE khám phá những đường đi khả thi khác nhau trong chương trình tham chiếu để tạo dữ liệu đầu vào của chương trình. Do đó, các đầu vào thử nghiệm này sẽ thực thi hết các đường đi của chương trình tham chiếu và có khả năng phát hiện được những chương trình cần tính có những hành vi khác so với Chương trình tham chiếu. Những phép đo SSE vẫn còn hạn chế, đó là phép đo SSE không xem xét đường đi của Chương trình cần phân tích để tạo các giá trị đầu vào thử

nghiệm mà chỉ dựa vào các đầu vào thử nghiệm được phân tích từ Chương trình tham chiếu. Các đầu vào thử nghiệm này không nắm bắt được hết các hành vi của Chương trình cần phân tích, Chương trình cần phân tích có thể sẽ có những hành vi khác so với Chương trình tham chiếu. Một số chương trình có thể có những vòng lặp vô hạn phụ thuộc vào giá trị đầu vào nên SSE không thể liệt kê được tất cả các đường dẫn của chương trình. Chúng ta xem xét và phân tích 2 đoạn Mã lệnh 3.9 và 3.10 để thấy được hạn chế của phép đo SSE như sau:

Mã lệnh 2.11: Chương trình  $P_1$ 

```
public static int sol(int x)
{
    int y = 0;
    switch (x)
    {
        case 1:
            y += 4;
            break;
        default:
            y = x - 100;
            break;
    }
    return y;
}
```

Mã lệnh 2.12: Chương trình  $P_2$ 

```
public static int sub(int x)
{
    int y = 0;
    if (x == 1)
        return y += 4;
    if (x == 2)
        return y *= 4;
    else
        return y = x - 100;
}
```

Hai đoạn Mã lệnh 2.11 và Mã lệnh 2.12 của hai Chương trình  $P_1$  và  $P_2$ , chọn Chương trình  $P_1$  làm Chương trình tham chiếu, sử dụng kỹ thuật **DSE** để phân tích chương trình  $P_1$  ta được tập các giá trị đầu vào thử nghiệm là  $(0, 1)$ . Trong khi đó, phân tích Chương trình  $P_2$  chúng ta được tập các giá trị đầu vào thử nghiệm của Chương trình  $P_2$  là  $(0, 1, 2)$ . Do đó, chúng ta thấy tập giá trị đầu vào thử nghiệm do phép đo **SSE** tạo ra

thiếu giá trị đầu thử nghiệm 2 để có thể thực thi hết các đường đi của Chương trình  $P_2$ .

## Kỹ thuật thực thi chương trình kết hợp Paired Program Symbolic Execution (PSE)

Để giải quyết giới hạn của phép đo SSE khi tạo ra tập các giá trị đầu vào thử nghiệm không thực thi hết các các đi của Chương trình cần phân tích. Phép đo PSE giải quyết giới hạn của phép đo SSE bằng cách tạo một Chương trình kết hợp giữa Chương trình cần phân tích với Chương trình tham chiếu. Dựa trên Chương trình kết hợp sử dụng kỹ thuật DSE để tạo ra đầu vào thử nghiệm cho cả hai chương trình, các đầu vào thử nghiệm này bao gồm các đầu vào thử nghiệm đúng và không đúng. Các đầu vào thử nghiệm đúng là những giá trị khi thực thi trên cả hai chương trình sẽ cho kết quả đầu ra như nhau, ngược lại các đầu vào thử nghiệm không đúng là những giá trị khi thực thi trên cả hai chương trình sẽ cho kết quả khác nhau. Do đó, phép đo PSE được tính bằng tỷ lệ các giá trị đầu vào thử nghiệm đúng trên tổng số các giá trị đầu vào được thử nghiệm.

Mã lệnh 2.13: Chương trình kết hợp PSE

---

```
public int P_3 (int number)
{
    if(Program1(args) == Program2(args))
        return 1;
    return 0;
}
```

---

Dựa trên cách thức hoạt động của phép đo PSE, chúng ta có định nghĩa phép đo PSE như sau:

**Định nghĩa 7 (Phép đo PSE)** Cho  $P_1$  và  $P_2$  là hai Chương trình có cùng miền giá trị đầu vào

I.  $P_3$  là Chương trình kết hợp của  $P_1$  và  $P_2$ , ký hiệu ( $exec(P_1, I) = exec(P_2, I)$ ).  $I_s$  là tập các giá trị đầu vào được tạo bởi DSE từ Chương trình  $P_3$ ,  $I_a$  là tập con  $I_s$ . Hai Chương trình  $P_1$  và  $P_2$  thực thi giống nhau khi Chương trình  $P_3$  thực thi cho giá trị đúng với  $\forall i \in I_a$ , ký hiệu  $exec(P_3, i) = T$ , và  $\nexists j \in I_s \setminus I_a$  thực thi Chương trình  $P_3$  cho giá trị đúng. Chỉ số phép đo PSE được định nghĩa là  $M_{PSE}(P_1, P_2) = |I_a| / |I_s|$ .

---

**Algorithm 4** Phép đo PSE
 

---

$P_1, P_2$ : Là hai chương trình cần đo tương tự  
 $I$ : Miền giá trị đầu vào của  $P_1, P_2$   
 $P_3$ : Là Chương trình kết hợp của  $P_1, P_2$ , ký hiệu ( $exec(P_1, I) = exec(P_2, I)$ )  
 Set  $I_s = DSE(P_3)$   $\triangleright I_s$ : Tập đầu vào của  $P_3$  theo DSE  
 Set  $I_a = \emptyset$   
**while**  $i \in I_s$  **do**  $\triangleright$  Vòng lặp dừng lại khi  $\forall i \in I_s$  đã được thực thi  
     **if** ( $exec(P_1, i) = exec(P_2, i)$ ) **then**  
          $I_a = I_a \cup i$   
 $M_{PSE}(P_1, P_2) = |I_a| / |I_s|$ .

---

Phép đo PSE đã cải thiện được hạn chế của phép đo SSE khi dữ liệu thử nghiệm được tạo ra từ trên Chương trình kết hợp, tập dữ liệu thử nghiệm có khả năng thực thi hết các nhánh đường đi của Chương trình tham chiếu và Chương trình cần tính. Tuy nhiên, phép đo PSE cũng có hạn chế trong quá trình xử lý các vòng lặp lớn hoặc vô hạn. Để giảm bớt hạn chế này, chúng ta có thể giới hạn miền đầu vào hoặc đếm số vòng lặp của các Chương trình. Ngoài ra, phép đo PSE khám phá đường dẫn của Chương trình kết hợp nên quá trình xử lý sẽ tốn thời gian và tài nguyên hơn so với phép đo SSE khi chỉ khám phá đường dẫn của Chương trình tham chiếu.

## Tổng kết chương

Nội dung chính được trình bày trong chương này bao gồm những kiến thức cơ sở về kiểm thử phần mềm, kỹ thuật sinh dữ liệu kiểm, kỹ thuật Dynamic symbolic execution. Một số định nghĩa về thực thi chương trình, độ tương tự hành vi, sự khác biệt về hành vi độ tương tự về hành vi của chương trình. Mô tả thuật toán và định nghĩa 3 kỹ thuật



đo RS, SSE, PSE cũng như trình bày những ưu điểm, nhược điểm và hướng khắc phục của 3 kỹ thuật đo.

## **Chương 3**

# **THỰC NGHIỆM, ĐÁNH GIÁ**

# Tài liệu tham khảo

- [1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 153–160. ACM, 2010.
- [2] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. Automated grading of dfa constructions. In *IJCAI*, volume 13, pages 1976–1982, 2013.
- [3] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
- [4] Samuel Bates and Susan Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 384–396. ACM, 1993.
- [5] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.
- [6] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 41–50. IEEE, 1992.
- [7] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the sixth conference on Computer systems*, pages 183–198. ACM, 2011.
- [8] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing

- in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.
- [9] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
- [10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.
- [11] Coursera. <https://www.coursera.org/>.
- [12] Yingnong Dang, Dongmei Zhang, Song Ge, Chengyun Chu, Yingjun Qiu, and Tao Xie. Xiao: Tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 369–378. ACM, 2012.
- [13] John A Darringer and James C King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.
- [14] Peli de Halleux and Nikolai Tillmann. Parameterized test patterns for effective testing with pex. volume 21. Citeseer, 2008.
- [15] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
- [16] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4, 2005.
- [17] EdX. <https://www.edx.org/>.
- [18] Daniel Galin. Software quality assurance: from theory to implementation. Pearson Education India, 2004.

- [19] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.
- [20] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. volume 17, pages 1284–1288. IEEE, 1991.
- [21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [23] Jan B Hext and JW Winings. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275, 1969.
- [24] Code Hunt. <https://www.microsoft.com/en-us/research/project/code-hunt/>.
- [25] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93. ACM, 2010.
- [26] Julia Isong. Developing an automated program checkers. In *Journal of Computing Sciences in Colleges*, volume 16, pages 218–224. Consortium for Computing Sciences in Colleges, 2001.
- [27] Daniel Jackson, David A Ladd, et al. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, volume 94, pages 243–252, 1994.
- [28] David Jackson and Michelle Usher. Grading student programs using assyst. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM, 1997.

- [29] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009.
- [30] Edward L Jones. Grading student programs-a software testing approach. *Journal of Computing Sciences in Colleges*, 16(2):185–192, 2001.
- [31] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [32] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.
- [33] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [34] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.
- [35] Daniel Kroening and Ofer Strichman. Decision procedures for propositional logic. In *Decision Procedures*, pages 27–58. Springer, 2016.
- [36] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, volume 12, pages 712–717. Springer, 2012.
- [37] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. Measuring code behavioral similarity for programming and software engineering education.

- In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 501–510. ACM, 2016.
- [38] John J. Marciniak, editor. *Encyclopedia of Software Engineering (Vol. 1 A-N)*. Wiley-Interscience, New York, NY, USA, 1994.
- [39] Ettore Merlo, Giuliano Antoniol, Massimiliano Di Penta, and Vincenzo Fabio Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 412–416. IEEE, 2004.
- [40] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.
- [41] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [42] Coursera MOOC on Software Engineering for SaaS. <https://www.coursera.org/course/saas>.
- [43] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.
- [44] Laura Pappano. The year of the mooc. *The New York Times*, 2(12):2012, 2012.
- [45] Corina S Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.
- [46] Corina S Păsăreanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In *International SPIN Workshop on Model Checking of Software*, pages 164–181. Springer, 2004.

- [47] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.
- [48] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.
- [49] Pex4Fun. <https://pexforfun.com/>.
- [50] Roger S Pressman. Software engineering: a practitioner’s approach. Palgrave Macmillan, 2005.
- [51] Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990):3, 1990.
- [52] Graham HB Roberts and Janet LM Verbyla. An online programming assessment tool. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 69–75. Australian Computer Society, Inc., 2003.
- [53] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.
- [54] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.
- [55] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: Detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 702–714. ACM, 2016.



- [56] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.
- [57] Kunal Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 407–410. IEEE Computer Society, 2008.
- [58] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.
- [59] Nikolai Tillmann, Jonathan De Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 385–396. ACM, 2014.
- [60] Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. Teaching and learning programming and software engineering via interactive gaming. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1117–1126. IEEE Press, 2013.
- [61] Udacity. <http://www.udacity.com/>.
- [62] Andrew Walenstein, Mohammad El-Ramly, James R Cordy, William S Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.
- [63] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. volume 49, pages 99–107. Elsevier, 2007.
- [64] James A Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.

- [65] Wikipedia. <https://en.wikipedia.org>.
- [66] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 246–256. IEEE, 2013.
- [67] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.
- [68] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. volume 29, pages 366–427. ACM, 1997.

Số: 299/LQĐ-ĐHQN

Bình Định, ngày 25 tháng 12 năm 2017

**QUYẾT ĐỊNH**

**Về việc giao đề tài và cử người hướng dẫn luận văn thạc sĩ**

**HIỆU TRƯỞNG TRƯỜNG ĐẠI HỌC QUY NHƠN**

Căn cứ Quyết định số 221/2003/QĐ-TTg ngày 30/10/2003 của Thủ tướng Chính phủ về việc đổi tên Trường Đại học sư phạm Quy Nhơn thành Trường Đại học Quy Nhơn;

Căn cứ nhiệm vụ và quyền hạn của Hiệu trưởng Trường đại học được quy định tại Điều 11 "Điều lệ trường đại học" ban hành kèm theo Quyết định số 70/2014/QĐ-TTg ngày 10/12/2014 của Thủ tướng Chính phủ;

Căn cứ Thông tư số 15/2014/TT-BGDĐT ngày 15/5/2014 về việc ban hành Quy chế đào tạo trình độ thạc sĩ của Bộ trưởng Bộ Giáo dục và Đào tạo và Quyết định số 5508/QĐ-ĐHQN ngày 12/11/2015 của Hiệu trưởng về việc ban hành Quy định đào tạo trình độ thạc sĩ của Trường Đại học Quy Nhơn;

Căn cứ Quyết định số 1867/QĐ-ĐHQN ngày 20/10/2016 về việc công nhận học viên của khóa đào tạo trình độ thạc sĩ 2016-2018 chuyên ngành Khoa học máy tính của Hiệu trưởng Trường Đại học Quy Nhơn;

Xét đề nghị của Trường phòng Đào tạo sau đại học,

**QUYẾT ĐỊNH:**

**Điều 1.** Giao đề tài luận văn thạc sĩ: **Độ tương tự hành vi của chương trình và thực nghiệm - chuyên ngành Khoa học máy tính**, mã số: 60480101, cho học viên **Đỗ Đăng Khoa** - Khóa 19 (2016-2018) và cử TS. **Phạm Văn Việt** - Trường Đại học Quy Nhơn, là người hướng dẫn luận văn thạc sĩ.

**Điều 2.** Học viên và người hướng dẫn thực hiện đồng nhiệm vụ và được hưởng quyền lợi theo các quy chế, quy định đào tạo trình độ thạc sĩ hiện hành.

**Điều 3.** Các ông (bà) Trường phòng Đào tạo sau đại học, Hành chính - Tổng hợp, Kế hoạch - Tài chính, Trường khoa Công nghệ thông tin, người hướng dẫn và học viên có tên tại Điều 1 chịu trách nhiệm thi hành Quyết định này./.

Nơi nhận: 

- Hiệu trưởng (để k/c);
- Như Điều 3;
- Lưu: VT, ĐTSĐH.



PGS.TS. Nguyễn Đình Hiền

# MỘT SỐ MÃ LỆNH QUAN TRỌNG