

**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
Danh sách hình vẽ	7
Listings	8
1 GIỚI THIỆU	9
1.1 Lý do chọn đề tài	9
1.2 Đối tượng, phạm vi, phương pháp nghiên cứu	10
1.3 Những nghiên cứu có liên quan	11
2 TƯƠNG TỰ VỀ HÀNH VI GIỮA CÁC CHƯƠNG TRÌNH	14
2.1 Kiến thức cơ sở	14
2.2 Hành vi của chương trình	24
2.3 Một số phép đo độ tương tự hành vi	27
3 THỰC NGHIỆM, ĐÁNH GIÁ	33
3.1 Dữ liệu thực nghiệm	33
3.2 Công cụ dùng trong thực nghiệm	34
3.3 Đánh giá kết quả thực nghiệm	37
3.4 Khả năng ứng dụng	41
3.5 Kết luận	42
Tài liệu tham khảo	44

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

Danh sách hình vẽ

2.1	DSE tạo ngẫu nhiên các tham số đầu vào của chương trình	17
2.2	DSE gán số nguyên Z bằng hàm <code>foo(y)</code>	18
2.3	DSE lưu lại điều kiện ràng buộc ($z \neq x$) của chương trình	18
2.4	DSE tính toán và giải ràng buộc $2 * y_0 == x_0$	19
2.5	DSE khởi động lại hàm <code>test_me</code>	19
2.6	DSE ghi nhận trạng thái biểu trưng của biến z	20
2.7	DSE ghi nhận điều kiện ràng buộc $2 * y_0 == x_0$	20
2.8	DSE ghi nhận ràng buộc $x_0 \leq y_0 + 10$	21
2.9	DSE thực hiện giải ràng buộc $(2 * y_0 == x_0) \text{ and } (x_0 > y_0 + 10)$	21
2.10	DSE thực thi hàm <code>test_me</code> với $x = 30$ và $y = 15$	22
2.11	DSE tạo trạng thái biểu trưng của biến z , với $z = 30$	22
2.12	Ghi nhận điều kiện ràng buộc tại điểm nhánh <code>if(z == x)</code>	23
2.13	Đường dẫn của chương trình khi thực thi với giá trị $x = 30$ và $y = 15$	23
2.14	Ví dụ sự khác biệt về hành vi	26
2.15	Ví dụ độ tương tự về hành vi	27
2.16	Ví dụ hạn chế của phép đo RS	29
2.17	Ví dụ hạn chế của phép đo SSE	31
3.1	Giao diện viết chương trình của Code Hunt	34
3.2	Quá trình biên dịch chương trình trong C#	35
3.3	Mô hình mô tả hoạt động của công cụ Pex	36
3.4	Giao diện màn hình chính của ứng dụng	37
3.5	Độ tương tự hành vi của hai chương trình sử dụng phép đo RS	39
3.6	Độ tương tự hành vi của chương trình theo phép đo SSE	40
3.7	Độ tương tự hành vi của chương trình theo phép đo PSE	41

Listings

2.1	Ví dụ minh họa cách thức hoạt động của kỹ thuật DSE	16
2.2	Sử dụng <code>switch...case</code>	25
2.3	Sử dụng <code>if...else</code>	25
2.4	Chương trình P_1	26
2.5	Chương trình P_2	26
2.6	Chương trình P_1	27
2.7	Chương trình P_2	27
2.8	Chương trình P_1	29
2.9	Chương trình P_2	29
2.10	Chương trình P_1	31
2.11	Chương trình P_2	31
3.1	Ví dụ một ca kiểm thử tham số hóa sử dụng công cụ Pex	36
3.2	Chương trình tham chiếu	37
3.3	Chương trình của sinh viên thứ nhất	38
3.4	Chương trình của sinh viên thứ hai	38
5	Mã lệnh tạo project của sinh viên	47
6	Mã lệnh tạo project chương trình tham chiếu	48
7	Mã lệnh build project của sinh viên	48
8	Mã lệnh build project chương trình tham chiếu	49
9	Mã lệnh build project chương trình hợp thành	49
10	Mã lệnh thực thi DSE trên chương trình hợp thành	49
11	Mã lệnh phép đo RS	50
12	Mã lệnh phép đo SSE	53
13	Mã lệnh phép đo PSE	55

Chương 1

GIỚI THIỆU

Chương này trình bày lý do chọn đề tài – sự cần thiết đo độ tương tự giữa các chương trình máy tính – cùng mục tiêu nghiên cứu, đối tượng và phạm vi nghiên cứu, cũng như những nghiên cứu có liên quan đến đề tài.

1.1 Lý do chọn đề tài

Hiện nay, ngành Công nghệ thông tin đang có xu hướng phát triển mạnh mẽ và học trực tuyến ngày càng trở nên phổ biến. Một số chương trình đào tạo trực tuyến nổi tiếng như Massive Open Online Courses (MOOC) [16], edX [7], Coursera [5], Udacity [20] ngày càng có nhiều người tham gia. Ngoài ra, một số chương trình hỗ trợ rèn luyện kỹ năng lập trình trực tuyến như Pex4Fun [17] hay Code Hunt [9] cũng thu hút nhiều sự quan tâm của nhiều người.

Một trong những thách thức của việc đào tạo lập trình trực tuyến là làm sao cho phép tổ chức những lớp học có quy mô lớn, trong khi đó vẫn đảm bảo chất lượng đào tạo. Thông thường, những khóa học trực tuyến phổ biến có rất nhiều học viên tham gia, hàng trăm thậm chí hàng ngàn người, nhưng số người phụ trách giảng dạy không nhiều. Sự hạn chế về số người phụ trách này có ảnh hưởng rất nhiều đến chất lượng đào tạo. Về phía người giảng dạy, để xếp hạng học viên hay cung cấp những phản hồi đối với từng lời giải của học viên đòi hỏi họ phải đọc hiểu toàn bộ mã lệnh do học viên viết. Công việc này dường như quá nặng nhọc nhưng không thể bỏ qua hoặc trì hoãn vì như vậy sẽ không thể bắt kịp tiến độ của học viên. Ngoài ra, về phía học viên, họ đến từ rất nhiều nơi trên thế giới nên thời gian học cũng khác nhau. Những lúc họ gặp khó khăn trong việc viết mã chương trình, họ có thể tìm kiếm sự trợ giúp từ bạn bè hoặc những người có kinh nghiệm, từ người dạy. Tuy nhiên, không phải lúc nào cũng có người này cũng có bên cạnh để giúp đỡ cho họ.

Thách thức trên có thể giải quyết nếu có được công cụ tự động hóa việc so sánh giải pháp của học viên đưa ra so với giải pháp của giáo viên xem chúng có tương đương không, hay

tương tự nhau ở mức độ nào. Sự tương đương giữa hai chương trình được hiểu theo nghĩa chúng cho ra cùng kết quả nếu nhận vào cùng dữ liệu. Việc chuyển đổi từ dữ liệu vào thành dữ liệu ra của một chương trình có thể xem là *hành vi* của nó. Công cụ này có thể giúp người dạy đánh giá xếp hạng giải pháp của học viên dựa vào giải pháp của mình. Ngoài ra, công cụ còn có thể giúp đánh giá được người học có tiến bộ không, giúp đưa ra những gợi ý lập trình cho học viên,...

Chúng tôi nhận thấy vấn đề cốt lõi để xây dựng công cụ này là làm sao đo được độ tương tự về hành vi giữa hai chương trình. Đó là lý do tôi chọn đề tài “*Độ tương tự hành vi của chương trình và thực nghiệm*”. Đề tài này sẽ làm rõ hành vi của chương trình, độ tương tự về hành vi của chương trình và một số phương pháp đo độ tương tự về hành vi.

1.2 Đối tượng, phạm vi, phương pháp nghiên cứu

Mục tiêu nghiên cứu

Mục tiêu nghiên cứu chính của luận văn là tìm cách đánh giá độ tương tự về hành vi giữa hai chương trình máy tính. Nó được cụ thể hóa thành những mục tiêu sau:

- Tìm hiểu sự tương tự hành vi của chương trình;
- Tìm hiểu kỹ thuật, công cụ sinh test case tự động và áp dụng để đo độ tương tự về hành vi;
- Tìm cách kết hợp các kỹ thuật đo với nhau;
- Đánh giá kết quả thực nghiệm.

Đối tượng, phạm vi nghiên cứu

Đối tượng nghiên cứu

Đối tượng nghiên cứu chính trong luận văn này gồm:

- Kỹ thuật sinh Test Case
- Các kỹ thuật đo độ tương tự hành vi
- Ứng dụng của các kỹ thuật đo độ tương tự hành vi

Phạm vi nghiên cứu

- Đo độ tương tự hành vi dựa vào Test Case
- Thực nghiệm, đánh giá trên các chương trình C#

Phương pháp nghiên cứu, thực nghiệm

Nghiên cứu lý thuyết

- Độ tương tự hành vi
- Một số kỹ thuật sinh Test Case tự động
- Kỹ thuật đo độ tương tự hành vi dựa trên Test Case
- So sánh, kết hợp các phép đo độ tương tự hành vi

Thực nghiệm

- Tiến hành cài đặt các kỹ thuật đo độ tương tự hành vi
- Thực nghiệm trên dữ liệu thực của CodeHunt, và một số dữ liệu thử khác
- Phân tích, đánh giá dựa trên kết quả thực nghiệm

1.3 Những nghiên cứu có liên quan

Đề tài này liên quan đến những nghiên cứu về xếp hạng tự động, sự tương đương giữa hai chương trình, phát hiện đạo code.

Xếp hạng tự động

Trong bài báo [1] nhóm tác giả đề xuất một phương pháp tự động so một automat hữu hạn đơn định chưa đúng của sinh viên với automat hữu hạn đơn định của giáo viên dùng làm automat tham chiếu. Cách tiếp cận này của các tác giả sử dụng các kỹ thuật đo sự khác nhau về cú pháp và ngữ nghĩa giữa hai automat. Sự khác biệt về cú pháp được đo qua khoảng cách chỉnh sửa (*syntactic edit distance*) và sự khác biệt về ngữ nghĩa được đo dựa trên các xâu vào được đoán nhận bởi automat (*accepted string inputs*). Mặc dù cách tiếp cận của nhóm tác giả bài báo và tác giả luận văn đều sử dụng khái niệm về độ tương tự về ngữ nghĩa, cách tiếp cận của bài báo dùng cho automat hữu hạn đơn định trong khi các độ đo trong luận văn dùng cho các chương trình máy tính.

Một bài báo khác liên quan đến chủ đề này là [18]. Trong đó, nhóm tác giả đề xuất phương pháp tự động xác định số sửa lỗi tối thiểu cần thực hiện trên chương trình của sinh viên (chưa đúng) sao cho khớp với giải pháp tham chiếu của giáo viên. Cách tiếp cận này cung cấp cách xử lý thông qua việc tính khoảng cách cú pháp tối thiểu (*minimal syntactic distance*) giữa chương trình sai với chương trình tham chiếu, trong khi đó luận văn tập trung vào đo độ tương tự về ngữ nghĩa của hai chương trình dựa vào các hành vi vào/ra.

Các độ đo trong luận văn có thể nhận biết độ tương tự giữa các chương trình có cấu trúc cú pháp khác nhau.

Ngoài ra, trước những nghiên cứu vừa nêu được đưa ra, trong bài báo [21], tác giả đề xuất một giải pháp đo độ tương tự về ngữ nghĩa của hai chương trình thông qua việc chuyển đổi chương trình của học viên và chương trình tham chiếu của giáo viên về một dạng chung nhưng không thay đổi ngữ nghĩa, đó là đồ thị phụ thuộc (*dependence graphs*), rồi tiến hành so sánh hai đồ thị để tính toán sự tương đồng. Thay vì so sánh các đồ thị, cách tiếp cận của đề tài này là so sánh các cặp đầu vào, đầu ra của các chương trình để tính toán các điểm tương đồng về hành vi.

Kiểm tra sự tương đương giữa hai chương trình

Có một số phương pháp kiểm tra sự tương đương về ngữ nghĩa/hành vi của các chương trình bằng cách sử dụng đồ thị phụ thuộc [2, 4], sự phụ thuộc giữa giá trị đầu vào và đầu ra [10]. Tất cả các cách tiếp cận để kiểm tra độ tương đương này đều trả về giá trị logic đúng/sai. Tuy nhiên, ngoài việc kiểm tra sự tương đương, giải pháp trong luận văn này còn có thể đo được độ tương tự giữa hai chương trình.

Phương pháp tự động xác định những đoạn mã tương đương nhau về chức năng thông qua các phép thử ngẫu nhiên [11]. Cách tiếp cận này xem xét hai đoạn mã có tương đương nhau hay không thông qua giá trị đầu vào và đầu ra, không quan tâm đến cấu trúc và cú pháp của chúng. Độ đo RS trong luận văn này cũng tương tự với cách tiếp cận trong bài báo. Tuy nhiên, luận văn này các đưa ra hai phương pháp đo khác, dựa trên kỹ thuật thực thi biểu trưng chương trình DSE (*Dynamic Symbolic Execution*).

Phát hiện đạo code

Những nhà nghiên cứu đã đề xuất các tiếp cận tính cho việc tính toán độ tương tự giữa các biến thể của các đoạn mã lệnh để tự động nhận diện đạo code, như sử dụng cây cú pháp trừu tượng [3], đồ thị phụ thuộc của chương trình [12], các độ đo dựa trên các đơn vị cú pháp (như các lớp, hàm) [6] [13]. Các tiếp cận này tính toán độ tương tự bằng cách phân tích tĩnh các đoạn mã lệnh còn cách tiếp cận trong luận văn về việc tính độ tương tự bằng cách thực thi chương trình để sinh ra các cặp dữ liệu vào/ra, làm cơ sở để tính toán.

Tổng kết chương

Ngày nay, những chương trình đào tạo trực tuyến ngày càng có nhiều người tham gia, có thể lên đến hàng trăm, thậm chí hàng ngàn người mỗi lớp. Thách thức quan trọng là làm sao tổ chức được các lớp có quy mô lớn nhưng vẫn đảm bảo chất lượng giáo dục trong điều kiện số lượng người tham gia giảng dạy cho mỗi lớp có hạn chế. Phương pháp đo độ tương tự về hành vi giữa hai chương trình máy tính làm cơ sở để xây dựng công cụ tự

động hóa nhằm giảm thiểu công sức người dạy trong việc đánh giá xếp hạng kết quả học tập của học viên cũng như đưa ra những gợi ý hỗ trợ cho người học trong quá trình viết chương trình. Chương 1 cho thấy sự cần thiết của đề tài cũng như đối tượng, phạm vi và phương pháp nghiên cứu. Ngoài ra, chương này còn trình bày một số nghiên cứu có liên quan về việc xếp hạng tự động, kiểm tra sự tương đương, phát hiện đạo code.

Phần còn lại của luận văn được tổ chức thành hai chương. Chương 2 làm rõ về hành vi của chương trình cùng những phương pháp đo độ tương tự về hành vi. Chương 3 trình bày kết quả thực nghiệm cùng một số vấn đề liên quan.

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 và 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 khá 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 nhằm đảm bảo mã máy tính chỉ làm những gì nó được thiết kế và không làm bất cứ điều gì ngoài ý muốn [14]. 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, các đặc tả và 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 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 ngược lại 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.

Kỹ thuật Dynamic symbolic execution

Dynamic symbolic execution (DSE) là một kỹ thuật sinh dữ liệu thử bằng cách 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 được sinh ra [22].

Dựa trên kiểu dữ liệu của các tham số đầu vào của chương trình, kỹ thuật DSE sẽ tạo ra các giá trị đầu vào cụ thể và thực thi chương trình với các giá trị cụ thể vừa tạo. Trong quá trình thực thi, DSE sẽ ghi nhận lại ràng buộc tại các nút rẽ nhánh của chương trình, phủ định lại các ràng buộc này và sinh các giá trị đầu vào thỏa các điều kiện ràng buộc vừa được ghi nhận. 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. Thuật toán Algorithm 1 mô tả cách thức hoạt động tạo tập dữ liệu đầu vào thử nghiệm của kỹ thuật DSE.

Algorithm 1 Thuật toán Dynamic symbolic execution

```

Set J := ∅                                ▷ 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

```

Để hiểu rõ cách thức hoạt động tạo tập dữ liệu đầu vào thử nghiệm của kỹ thuật DSE chúng ta phân tích ví dụ trong Mã lệnh 2.1 [15], với hàm `test_me` có hai tham số đầu vào là `int x` và `int y` và hàm này không có giá trị trả về.

Mã lệnh 2.1: Ví dụ minh họa cách thức hoạt động của kỹ thuật DSE

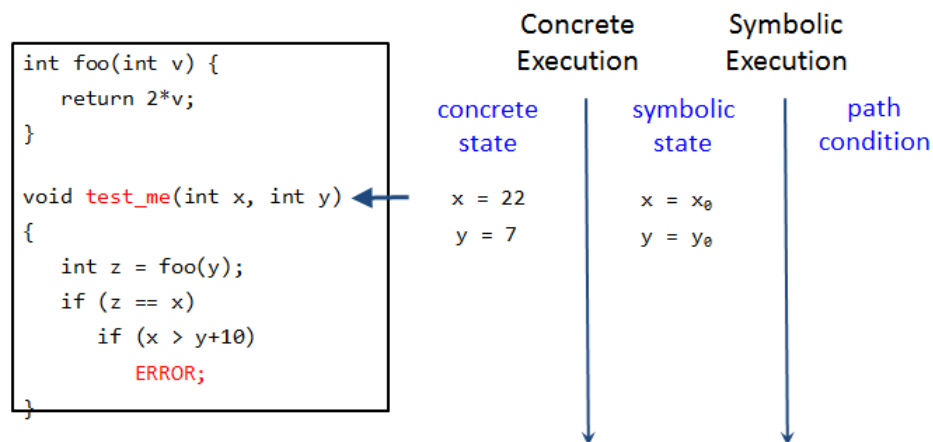
```

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;
}

```

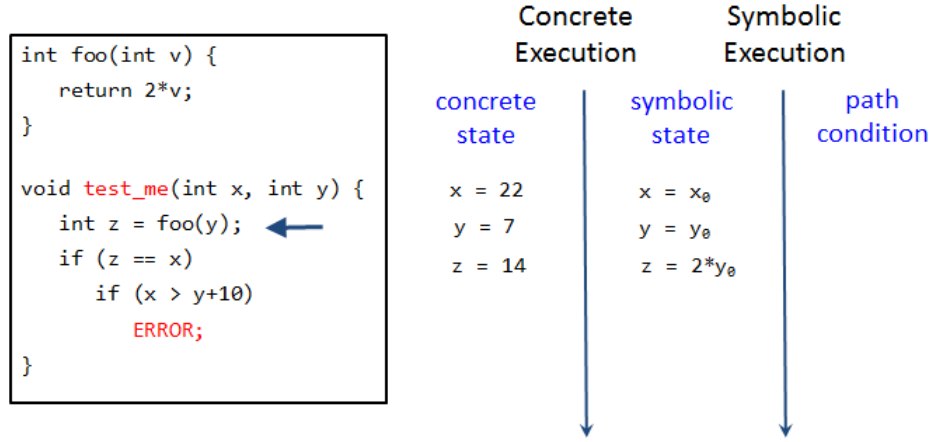
Đầu tiên, DSE tạo hai tham số đầu vào thử nghiệm (`int x`, `int y`) có giá trị ngẫu nhiên, giả sử $x = 22$ và $y = 7$. Bên cạnh đó, DSE theo dõi tham số đầu vào của chương trình bằng một giá trị biểu trưng, với x bằng một số x_0 và y bằng một số y_0 (Hình 2.1).

Hình 2.1: DSE tạo ngẫu nhiên các tham số đầu vào của chương trình

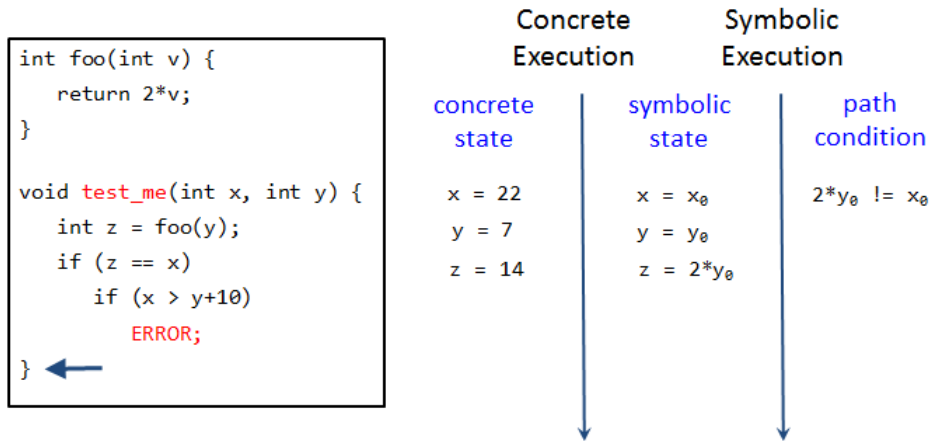


Tiếp theo, DSE thực thi dòng lệnh (`int z = foo(y)`) nghĩa là gán biến `z` bằng hàm `foo(y)`, lúc này biến $z = 14$ và trạng thái biểu trưng của biến `z` là $z = 2 * y_0$ (Hình 2.2)

Hình 2.2: DSE gán số nguyên Z bằng hàm foo(y)



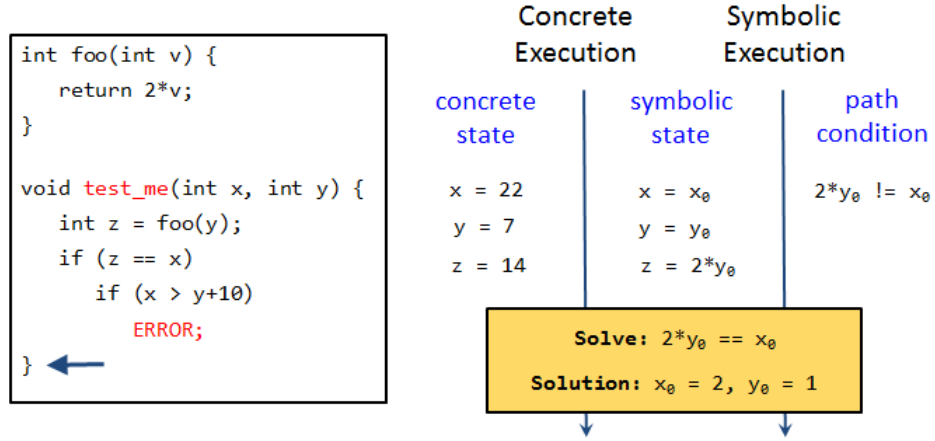
Tại câu lệnh `if(z == x)`, chúng ta thấy giá trị của `z` và `x` không bằng nhau, kết quả trả về của câu lệnh này sẽ là `false` dẫn đến kết thúc chương trình. Vì vậy, DSE sẽ lưu lại điều kiện ràng buộc ($z \neq x$) của đường dẫn này là $2 * y_0 \neq x_0$ (2.3).

Hình 2.3: DSE lưu lại điều kiện ràng buộc ($z \neq x$) của chương trình

Sau khi kết thúc chương trình, DSE quay trở lại điểm vừa lưu điều kiện ràng buộc (`if(z == x)`) để thực hiện phủ định điều kiện $2 * y_0 \neq x_0$ thành $2 * y_0 == x_0$. Sau đó DSE thực

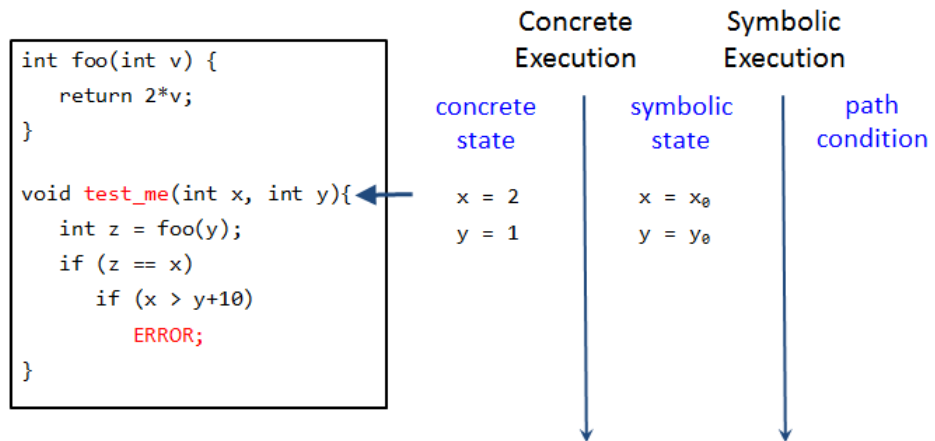
hiện tính toán và giải ràng buộc $2 * y_0 == x_0$, kết quả trả về là hai số nguyên có giá trị $x_0 = 2, y_0 = 1$ thỏa ràng buộc (Hình 2.4).

Hình 2.4: DSE tính toán và giải ràng buộc $2 * y_0 == x_0$



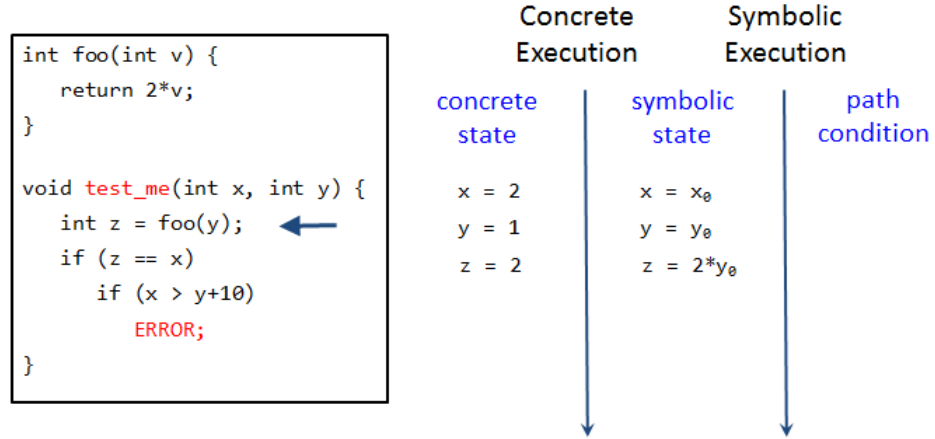
Sau đó, DSE khởi động lại hàm `test_me` với các giá trị đầu vào cụ thể $x = 2, y = 1$ vừa tạo ra trước đó và tiếp tục theo dõi trạng thái biểu trưng các giá trị đầu vào với $x = x_0$ và $y = y_0$ (Hình 2.5).

Hình 2.5: DSE khởi động lại hàm `test_me`



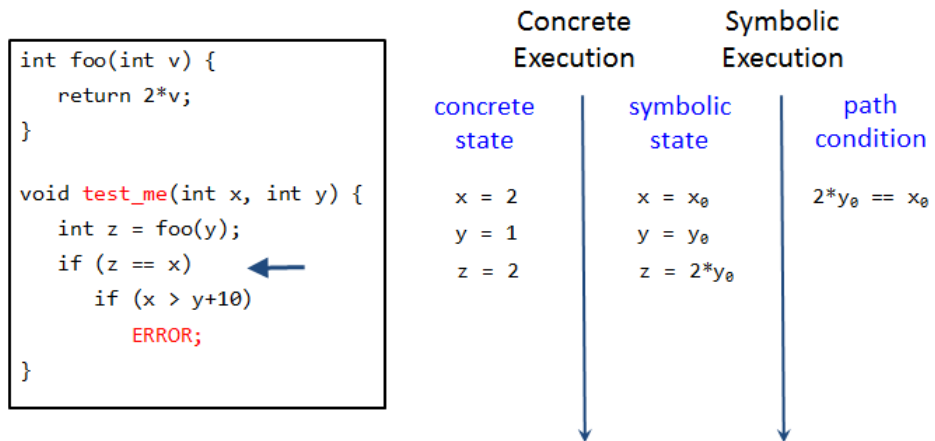
Tại câu lệnh gán biến $z = \text{foo}(y)$, lúc này biến z sẽ có giá trị bằng 2 và DSE thực hiện lưu trạng thái biểu trưng của biến z là $z = 2 * y_0$ (Hình 2.6).

Hình 2.6: DSE ghi nhận trạng thái biểu trưng của biến z



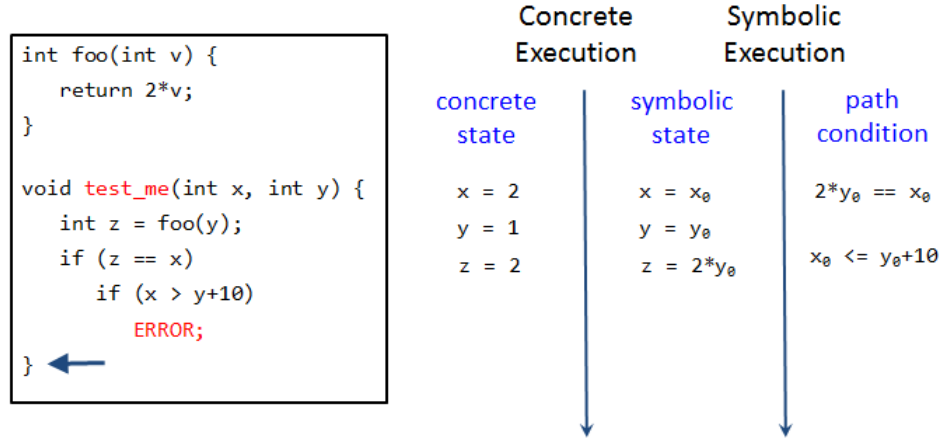
Tại nút rẽ nhánh $\text{if}(z == x)$, chúng ta thấy giá trị của biến $z == x$ thỏa điều kiện nên chương trình tiếp tục thực thi theo nhánh điều kiện **true** và DSE ghi nhận rằng buộc $2 * y_0 == x_0$ cho nút nhánh này (Hình 2.7).

Hình 2.7: DSE ghi nhận điều kiện ràng buộc $2 * y_0 == x_0$



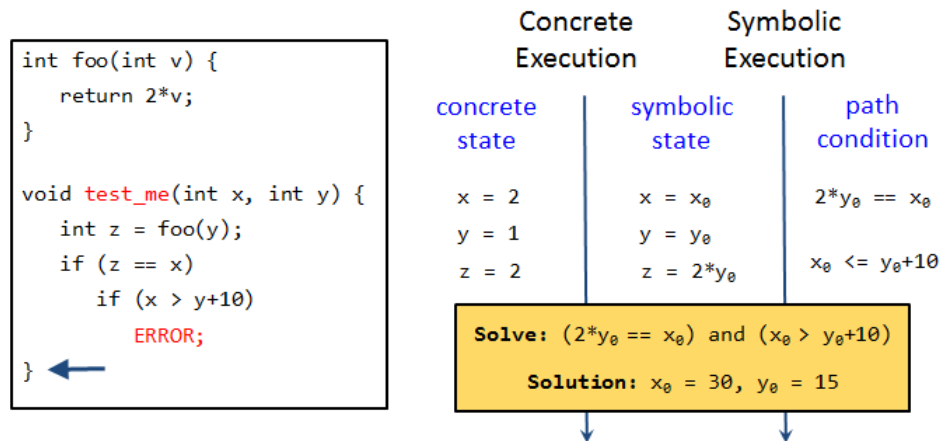
Tại nút rẽ nhánh tiếp theo $\text{if}(x > y+10)$, giá trị biến x bằng 2, và $(y + 10)$ bằng 11 nên chương trình sẽ thực thi theo nhánh false để kết thúc chương trình. Tại đây, DSE ghi nhận lại ràng buộc $x_0 \leq y_0 + 10$ cho nút rẽ nhánh $\text{if}(x > y+10)$ (Hình 2.8).

Hình 2.8: DSE ghi nhận ràng buộc $x_0 \leq y_0 + 10$



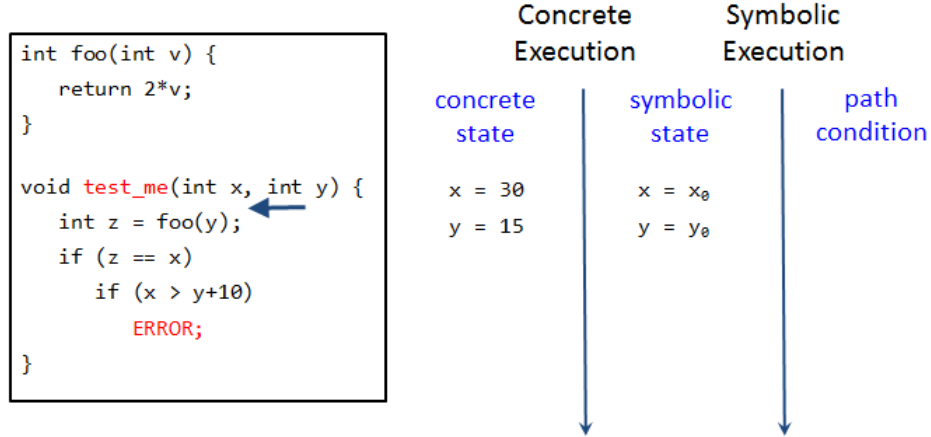
Vì đã đến cuối chương trình nên DSE quay lại nút nhánh gần nhất $\text{if}(x > y+10)$ và thực hiện phủ định lại ràng buộc $x_0 \leq y_0 + 10$ thành $x_0 > y_0 + 10$. Sau đó, DSE thực hiện giải ràng buộc $(2 * y_0 == x_0) \text{and} (x_0 > y_0 + 10)$ và tạo ra một giá trị đầu vào thỏa ràng buộc với $x_0 = 30$ và $y_0 = 15$ (Hình 2.9)

Hình 2.9: DSE thực hiện giải ràng buộc $(2 * y_0 == x_0) \text{and} (x_0 > y_0 + 10)$



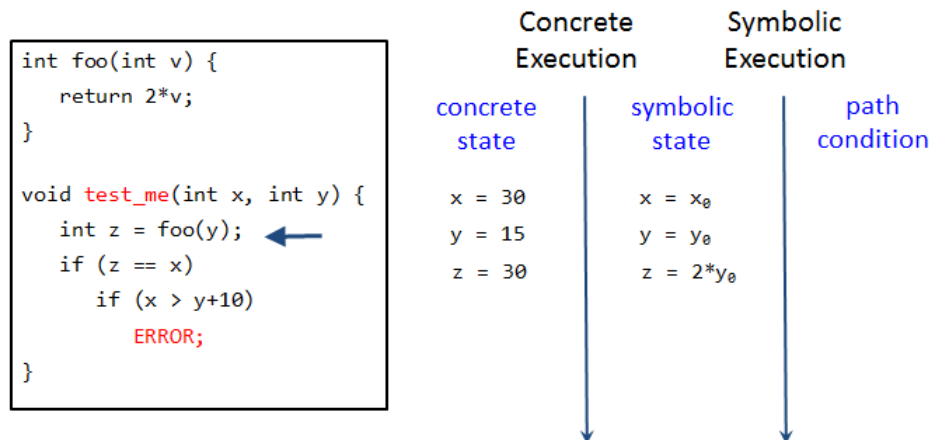
Sau khi có giá trị đầu vào mới, DSE thực thi hàm `test_me` một lần nữa với các giá trị đầu vào cụ thể $x = 30$ và $y = 15$ và tạo các trạng thái biểu trưng cho các biến đầu vào lần lượt $x = x_0$ và $y = y_0$ (Hình 2.10)

Hình 2.10: DSE thực thi hàm `test_me` với $x = 30$ và $y = 15$



Tại câu lệnh `z = foo(y)`, giá trị của biến `z` lúc này bằng 30 và trạng thái biểu trưng của biến `z` được khởi tạo $z = 2 * y_0$ như những lần trước đó (Hình 2.11)

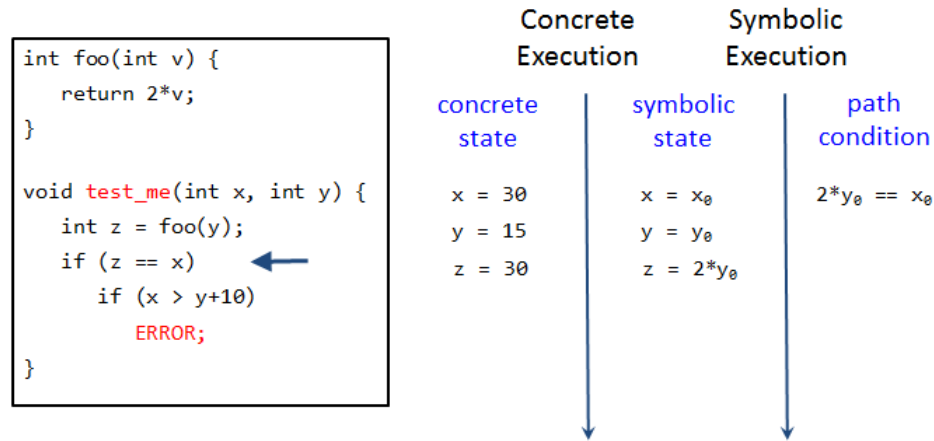
Hình 2.11: DSE tạo trạng thái biểu trưng của biến `z`, với $z = 30$



Tại điểm nhánh `if (z == x)`, chúng ta thấy giá trị của biến `z` bằng giá trị của biến `x` nên

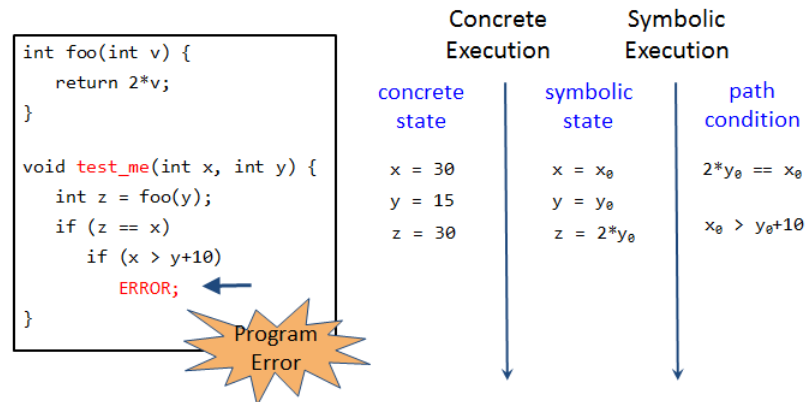
chương trình thực thi theo nhánh **true** và DSE ghi nhận ràng buộc $2 * y_0 == x_0$ cho điểm nhánh này (Hình 2.12)

Hình 2.12: Ghi nhận điều kiện ràng buộc tại điểm nhánh `if(z == x)`



Tại điểm nhánh tiếp theo `if(x > y+10)`, giá trị của biến $x = 30$, biến $y = 15$ nên $(x > y + 10)$ thỏa điều kiện đường dẫn, chương trình sẽ thực thi theo nhánh **true** dẫn đến kết quả **ERROR**, lúc này DSE thực hiện ghi nhận lại ràng buộc $x_0 > y_0 + 10$ cho nút rẽ nhánh `if(x > y+10)`. Vì chương trình thực thi đến **ERROR** và kết thúc chương trình nên chúng ta đã xác định được giá trị đầu vào cụ thể làm cho chương trình thực thi theo nhánh này với $x = 30$ và $y = 15$ (Hình 2.13).

Hình 2.13: Đường dẫn của chương trình khi thực thi với giá trị $x = 30$ và $y = 15$



Sau 3 lần chạy chương trình, kỹ thuật DSE tạo ra tập các giá trị đầu vào [22, 7], [2, 1], [30, 15] có độ phủ cao. Khi thực thi chương trình với tập giá trị đầu vào này tất cả các nhánh trong chương trình đều được thực thi.

Một số công cụ áp 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 như Pex [19] và SAGE [8]... 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 khác ở Bảng 2.1.

Bảng 2.1: Một số công cụ áp dụng kỹ thuật DSE

Tên Công cụ	Ngôn ngữ	Url
KLEE	LLVM	klee.github.io/
JPf	Java	babelfish.arc.nasa.gov/trac/jpf
jCUTE	Java	github.com/osl/jcute
Janala2	Java	github.com/ksen007/janala2
JBSE	Java	github.com/pietrobraione/jbse
KeY	Java	www.key-project.org/
Mayhem	Binary	forallsecure.com/mayhem.html
Otter	C	bitbucket.org/khooy/otter/overview
Rubyx	Ruby	www.cs.umd.edu/~avik/papers/ssarorwa.pdf
Pex	.NET Framework	research.microsoft.com/en-us/projects/pex/
Jalangi2	JavaScript	github.com/Samsung/jalangi2
Kite	LLVM	www.cs.ubc.ca/labs/isd/Projects/Kite/
pysymemu	x86-64 / Native	github.com/feliam/pysymemu/
Triton	x86 and x86-64	triton.quarkslab.com
BE-PUM	x86	https://github.com/NMHai/BE-PUM

2.2 Hành vi của chương trình

Trong luận văn này, hành vi của một chương trình là hành động biến đổi một dữ liệu đầu vào thành đầu ra tương ứng. Chương trình được xem như một hộp đen. Hành vi biến đổi của chương trình từ đầu vào sang đầu ra là hành động một bước (*one step action/big step*). Nghĩa là ta không quan tâm đến sự thay đổi của dữ liệu qua từng bước nhỏ khi thực hiện mỗi câu lệnh trong chương trình mà chỉ xem xét sự thay đổi qua một bước lớn, xem tất cả các câu lệnh là một câu lệnh ghép.

Để hình thức hóa hành vi của một chương trình và sử dụng trong những phần tiếp theo, ta tìm hiểu các định nghĩa hình thức liên quan đến việc thực thi một chương trình, sự

tương đương về hành vi giữa hai chương trình, sự khác biệt về hành vi và độ tương tự về hành vi giữa hai chương trình.

Thực thi chương trình

Việc thực thi một chương trình có thể xem là sự tương ứng mỗi giá trị vào với một giá trị ra, được nêu ra trong Định nghĩa 1.

Định nghĩa 1 (Thực thi chương trình) 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 chương trình P trên i ta thu được giá trị đầu ra tương ứng $o \in O, o = exec(P, i)$.

Tương đương về hành vi

Hai chương trình được gọi là tương đương với nhau về hành vi nếu chúng biến đổi cùng một giá trị vào thành cùng một giá trị ra đối với mọi giá trị trên miền vào. Xét ví dụ hai chương trình trong Mã lệnh 2.2 và 2.3.

Mã lệnh 2.2: Sử dụng `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 2.3: Sử dụng `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 2.2 và 2.3 có cùng tham số đầu vào x kiểu `int`, cùng tính toán và trả về giá trị y phụ thuộc vào x theo cách: nếu $x = 1$ thì trả về $y + 4$, nếu $x = 2$ thì trả về $2y$, còn không thì trả về giá trị bình phương của y .

Rõ ràng hành vi của hai mã lệnh chương trình trên là tương đương vì chúng trả về giá trị y giống nhau với cùng mỗi giá trị vào kiểu số nguyên x , mặc dù chúng có cấu trúc khác nhau (Mã lệnh 2.2 sử dụng cấu trúc `switch...case`, Mã lệnh 2.3 sử dụng cấu trúc

`if...else` để kiểm tra giá trị đầu vào x). Sự tương đương về hành vi của hai chương trình được hình thức hóa trong Định nghĩa 2.

Định nghĩa 2 (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 $exec(P_1, I) = exec(P_2, I)$.

Khác biệt về hành vi

Dựa vào Định nghĩa 2 ta có thể suy ra hai chương trình có hành vi khác nhau nếu chúng có miền giá trị đầu vào khác nhau, hoặc có một vài giá trị vào làm cho thực thi của chúng khác nhau. Trong phần này ta chỉ xét những chương trình có cùng miền giá trị vào. Ví dụ trong Hình 2.14 minh họa điều đó.

Hình 2.14: Ví dụ sự khác biệt về hành vi

Mã lệnh 2.4: Chương trình P_1	Mã lệnh 2.5: Chương trình P_2
<pre>using System; public class Program { public static int F1(int x) { return x - 10; } }</pre>	<pre>using System; public class Program { public static int F2(int x) { return x + 10; } }</pre>

Trong Hình 2.14, hàm $F1$ và $F2$ có cùng miền giá trị đầu vào kiểu `int`. Với mỗi giá trị vào x , giá trị trả về của hàm $F1$ là $x - 10$ và của $F2$ là $x + 10$. Rõ ràng hai hàm này có hành vi khác nhau. Mô tả hình thức về sự khác biệt hành vi giữa hai chương trình được nêu ra trong Định nghĩa 3.

Định nghĩa 3 (Sự khác biệt 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 xem là có sự khác biệt về hành vi khi và chỉ khi thực thi của chúng khác nhau trên một vài giá trị đầu vào $i \in I$, ký hiệu là $exec(P_1, I) \neq exec(P_2, I)$.

Độ tương tự về hành vi

Trong trường hợp hai chương trình không tương đương nhau về hành vi, nghĩa là có sự khác biệt về hành vi giữa chúng, thì ta cần biết mức độ tương tự giữa chúng là bao nhiêu.

Thông tin này khá hữu ích, nó giúp cho người dạy có thể đánh giá xếp hạng được giải pháp của người học dựa vào giải pháp của mình đưa ra, hoặc có thể biết được mức độ hoàn thiện của giải pháp do người học đưa ra. Xét độ tương tự về hành vi của hai chương trình P_1 và P_2 được cho lần lượt trong Mã lệnh 2.6 và 2.7. Chúng ta dễ dàng thấy được hai hàm $F1$ và $F2$ là không tương đương nhau trên toàn bộ miền giá trị vào `int` mà chỉ tương đương trên miền các giá trị số nguyên $E = [0, 100] \cup \{-1\}$. Từ đó, ta có thể tính được độ tương tự về hành vi của hai hàm này là $|E|/|\text{int}|$, trong đó $|\text{int}|$ là kích thước miền giá trị vào của hai hàm, `int`.

Hình 2.15: Ví dụ độ tương tự về hành vi

Mã lệnh 2.6: Chương trình P_1

```
using System;
public class Program
{
    public static int F1(int x)
    {
        if (x >= 0 && x <= 100)
            return x + 10;
        else
            return x;
    }
}
```

Mã lệnh 2.7: Chương trình P_2

```
using System;
public class Program
{
    public static int F2(int x)
    {
        if (x >= 0 && x <= 100)
            return x + 10;
        else
            return -1;
    }
}
```

Mô tả hình thức cho độ tương tự về hành vi giữa hai chương trình được nêu ra trong Định nghĩa 4. Trong định nghĩa này, I_s là tập lớn nhất các giá trị vào mà ở đó hai chương trình tương đương về hành vi.

Định nghĩa 4 (Độ tương tự về hành vi) Cho P_1 và P_2 là hai chương trình có cùng miền giá trị đầu vào I . Gọi $I_s \subseteq I$ là tập con của I sao cho $\text{exec}(P_1, I_s) = \text{exec}(P_2, I_s)$ và $\forall j \in I \setminus I_s, \text{exec}(P_1, j) \neq \text{exec}(P_2, j)$. Khi đó, độ tương tự về hành vi giữa P_1 và P_2 là $|I_s|/|I|$.

2.3 Một số phép đo độ tương tự hành vi

Theo Định nghĩa 4, để đo độ tương tự về hành vi của hai chương trình ta cần xác định:

1. Số phần tử miền vào của hai chương trình
2. Số phần tử trên miền vào (cực đại) mà ở đó hai chương trình thực thi giống nhau.

Để thấy, việc thứ 1 tương đối đơn giản, chỉ cần cho thực thi hai chương trình và so sánh kết quả. Khó khăn nằm ở công việc 2. Với những chương trình có miền vào nhỏ hoặc hữu hạn thì có thể thực hiện việc này bằng cách vét cạn tất cả các khả năng. Tuy nhiên, với những chương trình có miền vào cực lớn hoặc vô hạn thì việc vét cạn như vậy hầu như bất khả thi. Để giải quyết vấn đề này, chúng ta cần đến kỹ thuật sinh mẫu thử tự động sao cho đảm bảo bao phủ miền vào nhiều nhất. Luận văn này áp dụng kỹ thuật thực thi biểu trưng đối với chương trình (dynamic symbolic execution – DSE) để giải quyết khó khăn này (Xem Mục 2.1).

Phần còn lại của mục này trình bày ba phương pháp đo độ tương tự về hành vi giữa hai chương trình: lấy mẫu ngẫu nhiên, áp dụng DSE trên đơn chương trình tham chiếu, áp dụng DSE trên chương trình hợp thành từ chương trình giải pháp (*do người học cung cấp*) và chương trình tham chiếu (*đưa ra bởi người dạy*).

Lấy mẫu ngẫu nhiên

Phương pháp đơn giản nhất để đo độ tương tự về hành vi giữa hai chương trình là lấy ngẫu nhiên một số giá trị trên miền vào (*Random Sampling – RS*) và tính số lượng giá trị mà ở đó hai chương trình thực thi giống nhau, từ đó tính mức độ tương tự. Chúng ta gọi phép đo độ tương tự theo phương pháp này là *Phép đo RS*, được mô tả hình thức trong Định nghĩa 5.

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 sao cho $exec(P_1, I_a) = exec(P_2, I_a)$ và $\forall j \in I_s \setminus I_a$, thì $exec(P_1, j) \neq exec(P_2, j)$. Độ tương tự về hành vi giữa hai chương trình theo phép đo RS được định nghĩa là $M_{RS}(P_1, P_2) = |I_a| / |I_s|$.

Từ Định nghĩa 5, chúng tôi xây dựng thuật toán để tính độ tương tự theo phép đo RS như mô tả trong Algorithm 2, trong đó $I_s = Random(I)$ là thao tác chọn ngẫu nhiên một số giá trị vào I_s trong miền I . Trong phần thực nghiệm ở Chương 3, chúng tôi chọn số lượng giá trị ngẫu nhiên là 200, nghĩa là $|I_s| = 200$.

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 $I_s = Random(I)$

▷ I_s : Tập con ngẫu nhiên của I

Set $I_a = \emptyset$

for $i \in I_s$ **do**

if $(exec(P_1, i) = exec(P_2, i))$ **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 để tính độ tương tự về hành vi giữa hai chương trình. Vì không phải phân tích từng câu lệnh của chương trình mà chỉ quan tâm đến kiểu của miền giá trị vào để chọn một số giá trị ngẫu nhiên nên tốc độ xử lý tương đối nhanh và ít tốn tài nguyên hệ thống. Tuy nhiên, nhược điểm của phương pháp này là độ bao phủ của dữ liệu sinh ngẫu nhiên không cao. Chúng ta phân tích Mã lệnh 2.8 và 2.9 để thấy được hạn chế của phép đo **RS**.

Hình 2.16: Ví dụ hạn chế của phép đo RS

Mã lệnh 2.8: Chương trình P_1

```
public static int Y1(string x)
{
    if (x == "XYZ")
        return 0;
    if (x == "ABC")
        return 1;
    return -1;
}
```

Mã lệnh 2.9: Chương trình P_2

```
public static int Y2(string x)
{
    if (x == "ABC")
        return 1
    return -1;
}
```

Hai chương trình P_1 và P_2 có cùng tham số vào x kiểu **string**, cấu trúc mã lệnh hai chương trình gần giống nhau. Chương trình P_1 khác với chương trình P_2 khi có một điểm rẽ nhánh `if (x == "XYZ") return 0;`. Với điểm rẽ nhánh này, khả năng phép đo **RS** lấy ngẫu nhiên giá trị vào x có giá trị bằng XYZ trong miền vào kiểu **string** là rất thấp. Do đó chương trình P_1 có thể sẽ không thực thi nhánh `if (x == "XYZ") return 0`. Dữ liệu vào được lấy ngẫu nhiên trong phép đo RS trên miền vào của hai chương trình có thể sẽ không bao phủ hết các nhánh của chương trình.

DSE trên chương trình tham chiếu

Nhược điểm của phép đo RS được khắc phục bằng cách áp dụng kỹ thuật DSE để tăng độ bao phủ của tập dữ liệu thử. Phép đo áp dụng DSE trên chương trình tham chiếu, sau đây gọi tắt là SSE (*Single program Symbolic Execution*), chỉ cần dựa vào chương trình tham chiếu để xác định tập dữ liệu thử. Sự khác nhau giữa SSE và RS là SSE cần phân tích các câu lệnh của chương trình trong khi đó RS chỉ quan tâm đến kiểu dữ liệu vào. Mô tả hình thức cho phép đo SSE được nêu ra trong Định nghĩa 6.

Định nghĩa 6 (Phép đo SSE) Cho P_1 và P_2 là hai chương trình có cùng miền giá trị đầu vào I , trong đó P_1 là chương trình tham chiếu. Gọi I_s là tập các giá trị đầu vào được tạo bởi DSE trên P_1 , và $I_a \subseteq I_s$ là tập con lớn nhất của I_s sao cho $exec(P_1, I_a) = exec(P_2, I_a)$ và $\forall j \in I_s \setminus I_a, exec(P_1, j) \neq exec(P_2, j)$. Độ tương tự về hành vi giữa hai chương trình dùng phép đo SSE được định nghĩa là $M_{SSE}(P_1, P_2) = |I_a| / |I_s|$.

Thuật toán để đo độ tương tự dùng phép đo SSE được mô tả trong Algorithm 3, trong đó P_1 là chương trình tham chiếu và $DSE(P_1)$ là hành động sinh dữ liệu thử cho chương trình P_1 áp dụng kỹ thuật DSE.

Algorithm 3 Phép đo SSE

P_1, P_2 : hai chương trình cần đo tương tự, P_1 là chương trình tham chiếu
 I : Miền giá trị đầu vào của P_1, P_2
 Set $I_s = DSE(P_1)$ $\triangleright I_s$: Tập đầu vào của P_1 theo DSE
 Set $I_a = \emptyset$
for ($i \in I_s$) **do**
 if ($exec(P_1, i) = exec(P_2, i)$) **then**
 $I_a = I_a \cup i$
 $M_{SSE}(P_1, P_2) = |I_a| / |I_s|$.

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**.

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. Nhưng 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 bao phủ được hết các hành vi của chương trình cần phân tích vì 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 đi của chương trình. Chúng ta xem xét và phân tích 2 đoạn Mã lệnh 2.10 và 2.11 để thấy được hạn chế của phép đo SSE.

Hai đoạn Mã lệnh 2.10 và Mã lệnh 2.11 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 $(0, 1, 2)$. Tập giá trị đầu vào thử nghiệm do phép đo **SSE** tạo ra thiếu giá trị đầu vào 2 để có thể phủ hết các đường đi của chương trình P_2 .

Hình 2.17: Ví dụ hạn chế của phép đo SSE

Mã lệnh 2.10: 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 2.11: 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;
}

```

DSE trên chương trình hợp thành

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, ta xây dựng một chương trình hợp thành kết hợp cả hai, được mô tả trong Định nghĩa 7.

Định nghĩa 7 (Chương trình hợp thành) Cho P_1 và P_2 là hai chương trình có cùng miền vào I , trong đó P_1 là chương trình tham chiếu. Hợp thành của hai chương trình là một chương trình mới $P_c = P_1 \oplus P_2$ có dạng $\text{assert}(\text{exec}(P_1, I) = \text{exec}(P_2, I))$, trong đó $\text{assert}(\dots)$ là hàm đánh giá nhận vào một biểu thức điều kiện và đánh giá xem biểu thức đó đúng hay sai. Ký hiệu $\text{exec}(P_c, i) = \top$ để chỉ chương trình hợp thành P_c thỏa hàm đánh giá, cũng có nghĩa là hai chương trình thành phần thực thi giống nhau trên đầu vào i .

Phép đo độ tương tự về hành vi giữa hai chương trình áp dụng kỹ thuật DSE trên chương trình hợp thành, từ đây gọi là phép đo PSE (*Paired program Symbolic Execution*), giải quyết giới hạn của phép đo SSE bằng cách tạo một chương trình hợp thành giữa chương trình cần phân tích với chương trình tham chiếu. Dựa trên chương trình hợp thành, ta 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.

Thuật toán mô tả cách đo độ tương tự về hành vi giữa hai chương trình theo phép đo PSE được mô tả trong Algorithm 4.

Algorithm 4 Phép đo PSE

P_1, P_2 : hai chương trình cần đo tương tự, P_1 là chương trình tham chiếu
 I : Miền giá trị đầu vào của P_1, P_2
 $P_3 = P_1 \oplus P_2$
 Set $I_s = DSE(P_3)$ $\triangleright I_s$: Tập đầu vào của P_3 theo DSE
 Set $I_a = \emptyset$
for $i \in I_s$ **do**
 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 hợp thành. Tập dữ liệu thử nghiệm có khả năng thực thi hết các nhánh đường đi của chương trình tham chiếu và chương trình cần phân tích. 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 hợp thành nên quá trình xử lý sẽ tốn thời gian và tài nguyên của hệ thống hơn so với phép đo **SSE** khi chỉ khám phá đường dẫn của chương trình tham chiếu.

Tổng kết chương

Chương này tập trung trình bày những định nghĩa liên quan đến hành vi của chương trình cùng các phép đo độ tương tự về hành vi của hai chương trình. Ngoài ra, chương này còn cung cấp một số kiến thức cơ sở cho việc tìm hiểu những nội dung vừa nêu, cụ thể là kiến thức về kiểm thử phần mềm, đặc biệt là kỹ thuật DSE.

Thực thi của một chương trình là một ánh xạ tương ứng mỗi giá trị vào với một giá trị ra. Hành vi của một chương trình được xem xét trong luận văn này là một hành động một bước, chuyển một giá trị vào thành một giá trị đầu ra. Khi hai chương trình thực thi giống nhau trên mọi dữ liệu vào thì chúng được xem là tương đương nhau về hành vi. Ngược lại, nếu tồn tại một số giá trị vào làm cho chúng thực thi khác nhau thì được xem là khác biệt về hành vi. Khi hai chương trình không tương đương về hành vi, ta cần biết chúng tương tự nhau ở mức độ nào thông qua độ tương tự về hành vi. Độ tương tự trên có thể được đo bằng một trong 3 phép đo: RS, SSE và PSE, trong đó hai phép đo sau áp dụng kỹ thuật DSE để tăng độ phủ của tập dữ liệu thử.

Chương 3

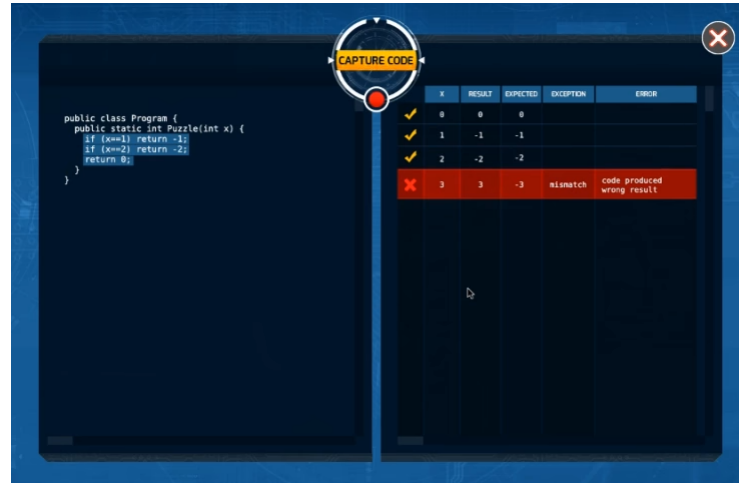
THỰC NGHIỆM, ĐÁNH GIÁ

Chương này, trình bày một số nội dung về dữ liệu và những công cụ được sử dụng trong quá trình thực nghiệm, đánh giá kết quả thực nghiệm đã làm được cũng như khả năng ứng dụng và hướng phát triển của đề tài trong tương lai. Qua đó đưa ra những nhận xét, đánh giá và kết luận chung cho đề tài.

3.1 Dữ liệu thực nghiệm

Để tiến hành thực nghiệm những nội dung nghiên cứu lý thuyết trong đề tài, tôi đã sử dụng hai nguồn dữ liệu chính để làm thực nghiệm. Một nguồn là dữ liệu của trò chơi Code Hunt, nguồn dữ liệu còn lại do tôi tự xây dựng.

Code Hunt là một game về lập trình, được sử dụng cho các cuộc thi viết mã và thực hành các kỹ năng lập trình do Microsoft phát triển. Code Hunt dựa trên công cụ Pex, ứng dụng kỹ thuật DSE khám phá các nhánh đường đi của chương trình để suy ra giá trị đầu vào có độ phủ cao. Dữ liệu trong trò chơi Code Hunt là một tập hợp những chương trình con, tương ứng mỗi chương trình con là những câu hỏi được trình bày như một bài kiểm tra. Người chơi phải chọn câu hỏi và trả lời mã câu hỏi bằng cách viết một đoạn mã sao cho kết quả trùng với kết quả cử câu hỏi. Hiện nay Code Hunt đã được hơn 350.000 người chơi sử dụng tính đến tháng 8 năm 2016 [9]. Dữ liệu từ các cuộc thi gần đây đã được công khai, cho phép những người quan tâm đến Code Hunt tải về để phân tích và nghiên cứu trong cộng đồng giáo dục.



Hình 3.1: Giao diện viết chương trình của Code Hunt

Tập dữ liệu Code Hunt là một tập dữ liệu chứa các chương trình do sinh viên trên toàn thế giới viết khi tham gia trò chơi, với 250 người sử dụng, 24 câu hỏi và khoảng 13.000 chương trình được sinh viên thực hiện trên 2 ngôn ngữ là Java và C#. Để có thể sử dụng tập dữ liệu Code Hunt trong thực nghiệm, tôi đã thực hiện kiểm tra và loại bỏ một số chương trình bị lỗi và không phù hợp với đề tài.

3.2 Công cụ dùng trong thực nghiệm

Trong quá trình làm thực nghiệm, tôi đã sử dụng một số công cụ để tạo một ứng dụng minh họa cho các phép đo độ tương tự hành vi của chương trình.

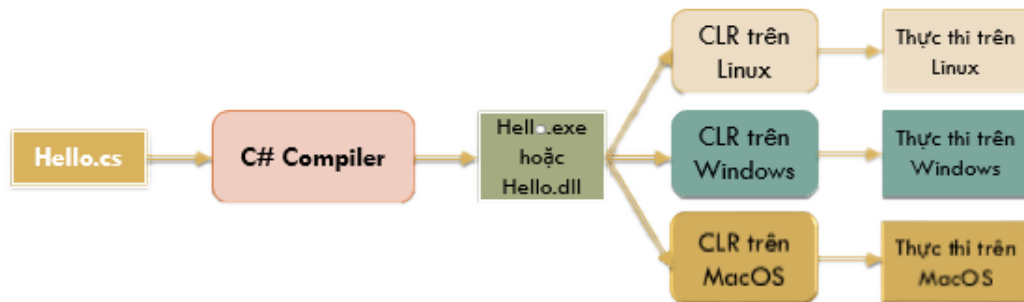
Microsoft Visual studio

Microsoft Visual Studio là một môi trường phát triển tích hợp từ Microsoft. Nó được sử dụng để phát triển chương trình máy tính cho Microsoft Windows, cũng như các trang web, các ứng dụng web và các dịch vụ web. Microsoft Visual Studio sử dụng nền tảng phát triển phần mềm của Microsoft như Windows API, Windows Forms, Windows Presentation Foundation, Windows Store và Microsoft Silverlight.

Visual Studio hỗ trợ nhiều ngôn ngữ lập trình khác nhau và cho phép trình biên tập mã và gỡ lỗi để hỗ trợ (mức độ khác nhau) hầu như mọi ngôn ngữ lập trình. Các ngôn ngữ tích hợp gồm có C, C++ và C++/CLI (thông qua Visual C++), VB.NET (thông qua Visual Basic.NET), C# (thông qua Visual C#) và F# (như của Visual Studio 2010). Hỗ trợ cho các ngôn ngữ khác như J++/J#, Python và Ruby thông qua dịch vụ cài đặt riêng rẽ. Nó cũng hỗ trợ XML/XSLT, HTML/XHTML, JavaScript và CSS.

Trong luận văn này, tôi sử dụng chủ yếu là ngôn ngữ lập trình C# để viết ứng dụng minh họa. Ngôn ngữ lập trình này là một ngôn ngữ lập trình hiện đại, được phát triển bởi Anders Hejlsberg cùng nhóm phát triển .Net Framework của Microsoft và được phê duyệt bởi European Computer Manufacturers Association (ECMA) và International Standards Organization (ISO). Mã nguồn C# là các tập tin *.cs được trình biên dịch Compiler biên dịch thành các file *.dll hoặc *.exe, sau đó các file này được các hệ thống thông dịch CLR trên điều hành thông dịch qua mã máy và dùng kỹ thuật JIT (just-in-time) để tăng tốc độ. Hình 3.2 mô tả quá trình biên dịch một file *.cs.

Hình 3.2: Quá trình biên dịch chương trình trong C#



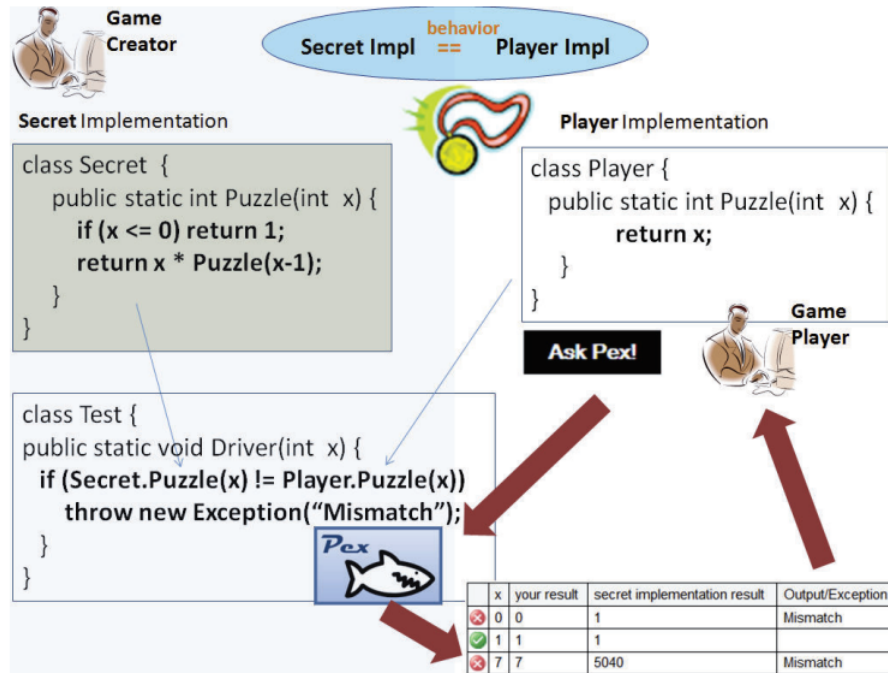
C# có thể tạo ra được nhiều loại ứng dụng, trong đó có 3 kiểu phổ biến được nhiều người sử dụng nhất đó là: Console, Window và ứng dụng Web.

Công cụ sinh dữ liệu thử Pex

Giới thiệu

Công cụ Pex là một sản phẩm của Microsoft được xây dựng và phát triển dựa trên kỹ thuật DSE. Trong Visual Studio, Pex đã được tích hợp như một Add-in, có thể tạo ra các test case kết hợp với các bộ kiểm thử khác nhau như NUnit và MSTest. Về bản chất công cụ Pex là một công cụ hỗ trợ kỹ thuật kiểm thử hộp trắng (White-box) khi nhận lại các ràng buộc tại các nút rẽ nhánh của chương trình, phủ định lại các ràng buộc này và sinh ra các đầu vào có thể phủ hết các nhánh của chương trình. Hình 3.3 mô tả mô hình hoạt động và ứng dụng của công cụ Pex.

Hình 3.3: Mô hình mô tả hoạt động của công cụ Pex



Cũng như với Unit Test, chúng ta có thể sử dụng công cụ Pex để viết các lớp kiểm thử với nhiều ca kiểm thử tham số hóa khác nhau, nhưng Pex chỉ thực thi được một ca kiểm thử tham số hóa trong mỗi lần chạy. Ví dụ một ca kiểm thử tham số hóa như Hình 3.1.

Mã lệnh 3.1: Ví dụ một ca kiểm thử tham số hóa sử dụng công cụ Pex

```
[PexMethod]
public void AddSpec(ArrayList list, object element)
{
    // assumptions
    PexAssume.IsTrue(list != null);
    // method sequence
    int len = list.Count;
    list.Add(element);
    // assertions
    Assert.IsTrue(list[len] == element);
}
```

3.3 Đánh giá kết quả thực nghiệm

Trong nội dung Chương 2 của đề tài, tôi đã trình bày một số lý thuyết về kiểm thử phần mềm, kỹ thuật sinh dữ liệu thử nghiệm, và các phép đo độ tương tự hành vi của chương trình **RS**, **SSE**, **PSE**. Áp dụng những lý thuyết trên, tôi thực nghiệm bằng cách xây dựng một ứng dụng để mô tả quá trình hoạt động của các phép đo, và đánh giá kết quả các phép đo. Kết quả thực nghiệm đạt được như Hình 3.4

Hình 3.4: Giao diện màn hình chính của ứng dụng



Hai nút đầu tiên, nút **Upload Solution** cho phép chúng ta upload file code chương trình tham chiếu, nút **Upload Users** cho phép chúng ta upload files cần tính độ tương tự. Bên dưới là các nút đo hành vi của chương trình, và các nút tính toán độ tương tự hành vi của chương trình theo các phép đo **RS**, **SSE** và **PSE**.

Để mô tả hoạt động của ứng dụng, tôi thực hiện chọn một đoạn mã lệnh mẫu làm chương trình tham chiếu với Mã lệnh 3.2 và hai chương trình cần tính độ tương tự với Mã lệnh 3.3 và Mã lệnh 3.4.

Mã lệnh 3.2: Chương trình tham chiếu

```
using System;
public class Program {
    public static int Puzzle(int x) {
        int y = 0;
        switch (x)
        {
```

```
        case 1: y += 4; break;
        case 2: y += 8; break;
        default: y = x - 100; break;
    }
    return y;
}
}
```

Mã lệnh 3.3: Chương trình của sinh viên thứ nhất

```
using System;
public class Program
{
    public static int Puzzle(int x)
    {
        int y = 0;
        if (x == 1) return y += 4;
        if (x == 2) return y *= 4;
        else return y = x - 100;
    }
}
```

Mã lệnh 3.4: Chương trình của sinh viên thứ hai

```
using System;
public class Program
{
    public static int Puzzle(int x)
    {
        int y = 0;
        if (x == 1) return y += 4;
        if (x == 2) return y *= 4;
        if (x == 3) return y *= 6;
        else return y = x - 100;
    }
}
```

Độ tương tự hành vi của chương trình theo phép đo RS

Dựa vào kết quả độ đo RS ở Hình 3.5, chúng ta có một tập các giá trị đầu vào với 200 phần tử được sinh ngẫu nhiên từ miền đầu vào của các chương trình. Kết quả độ tương tự hành vi hai chương trình của sinh viên với chương trình tham chiếu theo phép đo RS

lần lượt đều bằng 1, Với kết quả này, hai chương trình của sinh viên được xem là tương đương với chương trình tham chiếu.

Hình 3.5: Độ tương tự hành vi của hai chương trình sử dụng phép đo RS

KẾT QUẢ ĐỘ ĐO RS

Các giá trị đầu vào:

```

-1738776733 -560279829 -1015401832 397634579 -1729671826 763875409 -
1343618017 1859928678 1063545981 2082497361 -1085986565 -1057940699
1277641286 8540767 2039318814 398950736 -815602898 1433576148 1545564632
1483945513 -135291489 1933723976 -400025913 -906358606 2109802556 -
386679545 -820996490 615443091 -405638781 -1931757824 -1259301738
695348146 1074479106 1754984105 -1419503816 1375838032 1966251023 -
791642588 -1814689400 658870012 196429550 -802226029 -825780770
1594265037 -46730950 -1160747492 352124921 -906573942 -1395887281
2030805838 885517704 1185540148 923390076 -1370860017 742337985

```

	Students	Pass	All	RS
▶	sub1	200	200	1
	sub2	200	200	1
*				

Trong khi đó, phân tích mã lệnh chương trình của hai sinh viên chúng ta thấy có những hành vi khác so với chương trình tham chiếu, với sinh viên thứ nhất có hành vi *if* ($x == 2$) *return* $y * = 4$; và mã lệnh của sinh viên thứ hai có hành vi *if* ($x == 2$) *return* $y * = 4$; và *if* ($x == 3$) *return* $y * = 6$;

Vì vậy, kết quả độ tương tự hành vi chương trình của hai sinh viên với chương trình tham chiếu theo phép đo **RS** chỉ cho kết quả ở mức tương đối, giá trị đầu vào thử nghiệm được lấy ngẫu nhiên từ miền vào của các chương trình không phủ hết đường đi trong chương trình của hai sinh viên và chương trình tham chiếu.

Độ tương tự hành vi của chương trình theo phép đo SSE

Phân tích chương trình tham chiếu với kỹ thuật DSE chúng ta có tập các đầu vào thử nghiệm là (0, 1, 2). Khi thực thi chương trình của hai sinh viên và chương trình tham chiếu

với tập các đầu vào này, kết quả đầu ra chương trình của hai sinh viên có 2 kết quả trên tổng số 3 kết quả giống với chương trình tham chiếu, kết quả là 0,66 như Hình 3.6. Chúng ta thấy kết quả này đã thay đổi so với kết quả của phép đo RS (kết quả phép đo RS bằng 1). Tập các đầu vào thử nghiệm được tạo ra bởi kỹ thuật DSE có độ phủ cao, có thể phủ hết các nhánh của chương trình tham chiếu. Do vậy phép đo SSE cho kết quả chính xác hơn phép đo RS.

Hình 3.6: Độ tương tự hành vi của chương trình theo phép đo SSE

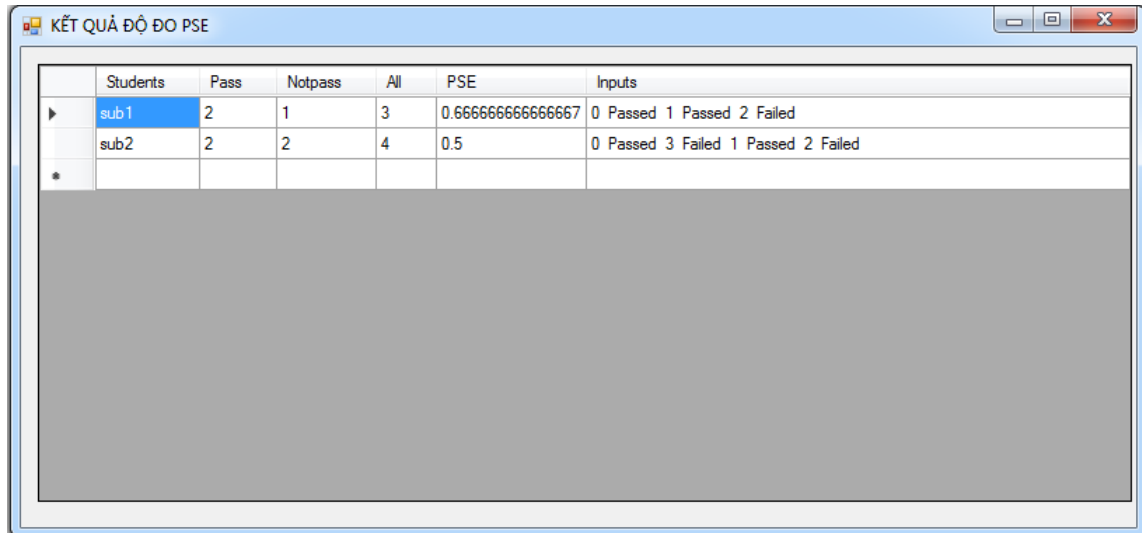
	Students	Pass	All	SSE
▶	sub1	2	3	0.6666666666666667
	sub2	2	3	0.6666666666666667
*				

Tuy nhiên, phép đo SSE lại không quan tâm đến hành vi của chương trình cần tính để tạo ra các đầu vào thử nghiệm có khả năng phủ hết các nhánh đường đi trong chương trình của sinh viên. Chúng ta thấy, các dữ liệu đầu vào trong trường hợp này không phủ hết được nhánh (`if(x==3) return y*=6;`) trong chương trình của sinh viên thứ hai.

Độ tương tự hành vi của chương trình theo phép đo PSE

Dựa trên kết quả Hình 3.7, chúng ta thấy kết quả độ tương tự hành vi chương trình của sinh viên thứ nhất với chương trình tham chiếu theo phép đo PSE giống với kết quả của phép đo SSE (*cả hai phép đo đều bằng 0.66*) và tập dữ liệu vào được tạo bởi DSE trên chương trình hợp thành của hai chương trình (0, 1, 2) cũng không thay đổi so với phép đo SSE.

Hình 3.7: Độ tương tự hành vi của chương trình theo phép đo PSE



	Students	Pass	Notpass	All	PSE	Inputs
▶	sub 1	2	1	3	0.666666666666667	0 Passed 1 Passed 2 Failed
	sub2	2	2	4	0.5	0 Passed 3 Failed 1 Passed 2 Failed
*						

Nhưng độ tương tự hành vi chương trình của sinh viên thứ hai với chương trình tham chiếu theo phép đo PSE đã thay đổi bằng 0.5, kết quả này nhỏ hơn so với kết quả 0.66 của phép đo SSE. Phép đo PSE cũng đã tạo ra tập dữ liệu vào dựa trên chương trình hợp thành với 4 phần tử là (0, 3, 1, 2) nhiều hơn tập đầu vào do phép đo SSE tạo ra 1 phần tử.

Dựa trên chương trình hợp thành, DSE đã tạo ra tập đầu vào có độ phủ cao, có thể thực thi hết các nhánh của chương trình tham chiếu và chương trình của sinh viên thứ hai. Kết quả độ tương tự hành vi của chương trình với chương trình tham chiếu theo phép đo PSE cho kết quả tương đối chính xác.

3.4 Khả năng ứng dụng

Các kỹ thuật đo độ tương tự trình bày trong đề tài này có thể áp dụng được trong nhiều lĩnh vực, nhưng chủ yếu tập trung vào giáo dục, chương trình đào tạo lập trình viên, hay đào tạo kỹ sư phần mềm... Một số ứng dụng thực tế có thể phát triển trong tương lai như:

Đánh giá tiến bộ trong lập trình: Theo dõi sự tiến bộ trong học tập là một việc quan trọng, mà ngay cả với giảng viên và sinh viên. Có nhiều tiêu chí đánh giá sự tiến bộ trong học tập của sinh viên, trong đó tiêu chí về điểm số là một trong những tiêu chí cơ bản nhất. Một bảng điểm thống kê điểm số, thành tích học tập của sinh viên sẽ thể hiện được sự tiến bộ của sinh viên trong học tập. Một ứng dụng hỗ trợ chấm điểm, lưu trữ, thống kê và đánh giá điểm số của sinh viên là sẽ là một công cụ hỗ trợ đắc lực cho giảng viên trong công tác quản lý của mình. Nếu số liệu thống kê kết quả các bài kiểm tra của sinh viên ngày càng cao, chứng tỏ sinh viên nắm được nội dung và kiến thức của chương trình

đào tạo, và kết quả tốt sẽ là một động lực giúp cho sinh viên thêm tự tin, đam mê công việc học tập của mình. Ngược lại, nếu một sinh viên có điểm số ngày càng thấp đi, chứng tỏ sinh viên đang có vấn đề trong kiến thức của của mình, lúc này tốt nhất sinh viên nên dừng lại và kiểm tra xem vấn đề mình đang gặp phải.

Xếp hạng tự động: Công việc chấm điểm, phân loại và xếp hạng các bài kiểm tra của sinh viên cũng là một công việc tốn không ít công sức của giảng viên. Để giảm bớt gánh nặng cho giảng viên, chúng ta có thể sử dụng kết quả các phép đo trên từng bài tập của sinh viên như một phương pháp hỗ trợ công việc chấm điểm của từng sinh viên. Sự giống nhau về hành vi giữa chương trình của sinh viên và chương trình tham chiếu có thể là một yếu tố để phân loại sinh viên. Độ tương tự càng cao thì điểm số càng cao, các chỉ số này dựa hoàn toàn trên ngữ nghĩa của chương trình. Cách tiếp cận này giải quyết được các giới hạn trong trường hợp chương trình của sinh viên giống với chương trình tham chiếu, nhưng khác nhau về ngữ nghĩa. Các kết quả trong việc xếp hạng tự động sẽ giúp tiết kiệm được thời gian và giảng viên có thể đưa ra giải pháp giúp những sinh viên có điểm số thấp khắc phục được hạn chế đang gặp phải.

Gợi ý giải pháp lập trình: Thông thường, sinh viên viết code mới thực hiện chạy chương trình, lúc này sinh viên mới biết được kết quả đoạn code vừa thực hiện. Để hỗ trợ sinh viên viết code được tốt hơn, nếu như có một công cụ hỗ trợ kiểm tra theo thời gian thực và gửi thông báo lỗi nếu sinh viên viết code sai cú pháp hoặc chương trình bị lỗi không thể thực thi được. Ngoài ra, công cụ sẽ gợi ý giải pháp lập trình cho sinh viên bằng hình thức tự động tính toán thông báo kết quả các tham số đầu vào và đầu ra của chương trình so với chương trình được tham chiếu, đưa ra các số liệu về độ tương tự hành vi của chương trình.

3.5 Kết luận

Qua quá trình nghiên cứu đề tài, có thể thấy rằng việc sử dụng một công cụ hỗ trợ để đo độ tương tự hành vi chương trình phục vụ trong việc dạy và học tại các trường đào tạo lập trình viên là rất cần thiết. Công cụ hỗ trợ không chỉ giúp giảng viên tiết kiệm được thời gian trong việc đọc và đánh giá các đoạn mã lệnh do sinh viên viết. Bên cạnh đó, công cụ còn hỗ trợ cho giảng viên trong việc đánh giá xếp loại học lực của sinh viên thông qua kết quả độ tương tự hành vi của chương trình sinh viên với chương trình tham chiếu. Sinh viên cũng có thể sử dụng công cụ này trong việc thực hành kỹ năng lập trình và giải các bài tập do giảng viên đưa ra.

Một công cụ hỗ trợ để đo độ tương tự hành vi chương trình cần tính với chương trình tham chiếu phải đảm bảo thực thi nhanh và có kết quả chính xác. Tuy nhiên, các phép đo độ tương tự hành vi của chương trình được nghiên cứu trong đề tài đều có ưu điểm và hạn chế riêng. Phép đo RS có ưu điểm là đánh giá khách quan, chương trình có tốc độ xử lý nhanh, ít tốn tài nguyên máy tính, nhưng phép đo RS có hạn chế khi kết quả phép đo chỉ ở mức độ tương đối vì phép đo không quan tâm đến hành vi của chương trình, tập

giá trị đầu vào được lấy ngẫu nhiên từ miền vào của chương trình có độ phủ không cao, không phủ được tất cả các nhánh của chương trình. Phép đo SSE sử dụng kỹ thuật DSE để sinh ra tập các giá trị đầu vào của chương trình, các giá trị trong tập đầu vào này có khả năng phủ tất cả nhánh của chương trình tham chiếu, vì vậy phép đo SSE cho kết quả độ tương tự hành vi của chương trình chính xác hơn phép đo RS. Tuy nhiên, phép đo SSE vẫn có mặc hạn chế khi không quan tâm đến hành vi của chương trình cần tính, tập đầu vào của phép đo SSE không phủ được tất cả các nhánh của chương trình cần phân tích. Để giải quyết hạn chế của phép đo SSE, phép đo PSE sử dụng kỹ thuật DSE để tạo tập giá trị đầu vào trên chương trình hợp thành của chương trình cần tính và chương trình tham chiếu. Tập các giá trị đầu vào này có khả năng phủ tất cả các nhánh đường đi trên cả hai chương trình. Kết quả của phép đo PSE tương đối chính xác và tốt hơn kết quả của phép đo SSE, nhưng phép đo PSE lại tốn nhiều tài nguyên máy tính và thời gian thực thi lâu hơn phép đo SSE.

Từ các kết quả đã đạt được, tương lai đề tài có thể phát triển thành một ứng dụng hoàn thiện, bằng cách cải tiến các kỹ thuật đo để kết quả các phép đo được chính xác hơn và nhạy hơn. Bổ sung thêm một số chức năng như phân loại, xếp hạng tự động đánh giá kết quả học tập của sinh viên... Ngoài ra, để ứng dụng được nhiều người quan tâm hơn thì ứng dụng phải đơn giản, thân thiện và phải hỗ trợ một số ngôn ngữ lập trình phổ biến như Java, C++...

Trong quá trình nghiên cứu những kiến thức cơ sở của đề tài và tiến hành là thực nghiệm, bản thân tôi cũng gặp rất nhiều khó khăn như: Lượng kiến thức cơ sở cần phải nghiên cứu tương đối nhiều; bản thân thiếu kinh nghiệm trong việc thực hiện các đề tài... Tuy nhiên, nhờ sự động viên và giúp đỡ của bạn bè, gia đình, cùng với sự tận giúp đỡ của giáo viên hướng dẫn tôi đã hoàn thành luận văn, đáp ứng được các yêu cầu đã đề ra.

Tài liệu tham khảo

- [1] 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.
- [2] 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.
- [3] 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.
- [4] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 41–50. IEEE, 1992.
- [5] Coursera. <https://www.coursera.org/>.
- [6] 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.
- [7] EdX. <https://www.edx.org/>.
- [8] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [9] Code Hunt. <https://www.microsoft.com/en-us/research/project/code-hunt/>.
- [10] 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.
- [11] 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.

- [12] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.
- [13] 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.
- [14] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.
- [15] Mayur Naik. <http://www.cis.upenn.edu/~mhnaik/edu/cis700/index.html>.
- [16] Coursera MOOC on Software Engineering for SaaS. <https://www.coursera.org/course/saas>.
- [17] Pex4Fun. <https://pexforfun.com/>.
- [18] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.
- [19] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.
- [20] Udacity. <http://www.udacity.com/>.
- [21] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. volume 49, pages 99–107. Elsevier, 2007.
- [22] 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.

Số: 2991/QĐ-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 Trưởng 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 đúng 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) Trưởng phòng Đào tạo sau đại học, Hành chính - Tổng hợp, Kế hoạch - Tài chính, Trưởng 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 b/c);
- Như Điều 3;
- Lưu: VT, ĐTSĐH.



PGS.TS. Nguyễn Đình Hiền

MỘT SỐ MÃ LỆNH QUAN TRỌNG

Mã lệnh 5: Mã lệnh tạo project của sinh viên

```
public static void MakeProjects(string topDir) {
    string[] references = { "Microsoft.Pex.Framework",
                            "Microsoft.Pex.Framework.Settings",
                            "System.Text.RegularExpressions" };
    foreach (string taskDir in Directory.GetDirectories(topDir)) {
        foreach (var studentDir in Directory.GetDirectories(taskDir)) {
            if (studentDir.EndsWith("secret_project"))
                continue;
            foreach (var file in Directory.GetFiles(studentDir)) {
                if (file.EndsWith(".cs")) {
                    string fileName = file.Substring(file.LastIndexOf("\\") + 1);
                    string projectName = "project" + fileName.Substring(0,
                        fileName.Length - 3);
                    string projectDir = studentDir + "\\" + projectName;
                    Directory.CreateDirectory(projectDir);
                    CreateProjectFile(projectDir + "\\" + projectName +
                        ".csproj", fileName);
                    string propertyDir = projectDir + "\\Properties";
                    Directory.CreateDirectory(propertyDir);
                    CreateAssemblyFile(propertyDir + "\\AssemblyInfo.cs",
                        projectName);
                    string newFile = projectDir + "\\" + fileName;
                    File.Copy(file, newFile, true);
                    AddNameSpace(newFile, "Submission");
                    AddUsingStatements(newFile, references);
                    //string[] classes={"Program"};
                    string[] methods = { "Puzzle" };
                    AddPexAttribute(newFile, null, methods);
                }
            }
        }
    }
}
```

```

    }
  }
}

```

Mã lệnh 6: Mã lệnh tạo project chương trình tham chiếu

```

public static void MakeSecretProjects(string topDir) {
    string[] references = { "Microsoft.Pex.Framework",
                           "Microsoft.Pex.Framework.Settings",
                           "System.Text.RegularExpressions"};
    foreach (string taskDir in Directory.GetDirectories(topDir)) {
        string[] files = Directory.GetFiles(taskDir);
        string secretFile = null;
        foreach (var file in files) {
            if (file.EndsWith("solution.cs")){
                secretFile = file;
                break;
            }
        }
        if (secretFile == null) {
            throw new Exception("secret implementation not found");
        }
        string fileName = secretFile.Substring(secretFile.LastIndexOf("\\") + 1);
        string projectName = "secret_project";
        string projectDir = taskDir + "\\ " + projectName;
        Directory.CreateDirectory(projectDir);
        CreateProjectFile(projectDir + "\\ " + projectName + ".csproj", fileName);
        string propertyDir = projectDir + "\\Properties";
        Directory.CreateDirectory(propertyDir);
        CreateAssemblyFile(propertyDir + "\\AssemblyInfo.cs", projectName);
        string newFile = projectDir + "\\ " + fileName;
        File.Copy(secretFile, newFile, true);
        AddNameSpace(newFile, "Solution");
        AddUsingStatements(newFile, references);
        string[] classes = { "Program" };
        string[] methods = { "Puzzle" };
        AddPexAttribute(newFile, classes, methods);
    }
}

```

Mã lệnh 7: Mã lệnh build project của sinh viên

```

public static void BuildProjects(string topDir, bool rebuild){
    foreach (var taskDir in Directory.GetDirectories(topDir)){
        foreach (var studentDir in Directory.GetDirectories(taskDir)){

```

```
        if (studentDir.EndsWith("secret_project"))
            continue;
        foreach (var projectDir in Directory.GetDirectories(studentDir)){
            if (!projectDir.Contains("meta_project")){
                BuildSingleProject(projectDir, rebuild);
            }
        }
    }
}
```

Mã lệnh 8: Mã lệnh build project chương trình tham chiếu

```
public static void BuildSecretProjects(string topDir, bool rebuild){
    foreach (string taskDir in Directory.GetDirectories(topDir)){
        string secretDir = null;
        foreach (var dir in Directory.GetDirectories(taskDir)){
            if (dir.EndsWith("secret_project")){
                secretDir = dir;
                break;
            }
        }
        if (secretDir == null)
            throw new Exception("secret project not found");
        BuildSingleProject(secretDir, rebuild);
    }
}
```

Mã lệnh 9: Mã lệnh build project chương trình hợp thành

```
public static void BuildMetaProjects(string topDir, bool rebuild){
    foreach (var taskDir in Directory.GetDirectories(topDir)){
        foreach (var studentDir in Directory.GetDirectories(taskDir)){
            if (studentDir.EndsWith("secret_project"))
                continue;
            foreach (var projectDir in Directory.GetDirectories(studentDir)){
                if (projectDir.Contains("meta_project")){
                    BuildSingleProject(projectDir, rebuild);
                }
            }
        }
    }
}
```

 Mã lệnh 10: Mã lệnh thực thi DSE trên chương trình hợp thành

```

public static void RunPexOnMetaProjects(string topDir){
    foreach (var taskDir in Directory.GetDirectories(topDir)){
        foreach (var studentDir in Directory.GetDirectories(taskDir)){
            if (studentDir.EndsWith(@"\secret_project"))
                continue;
            foreach (var metaDir in Directory.GetDirectories(studentDir)){
                if (metaDir.Contains("meta_project")){
                    string reportDir = metaDir + @"\bin\Debug\reports";
                    if (Directory.Exists(reportDir)){
                        DeleteDirectory(reportDir);
                    }
                    string assemblyName = metaDir.Substring(metaDir.LastIndexOf('\') +
                        1);
                    string assemblyFile = metaDir + @"\bin\Debug\"+assemblyName+".dll";
                    if (!File.Exists(assemblyFile)) {
                        continue;
                    }
                    string[] methods = { "Check" };
                    CommandExecutor.ExecuteCommand(
                        CommandGenerator.GenerateRunPexCommand(assemblyFile,
                            "MetaProject", "MetaProgram", methods));
                }
            }
        }
    }
}

```

 Mã lệnh 11: Mã lệnh phép đo RS

```

public static void ComputeMetric3(string topDir) {
    foreach (var taskDir in Directory.GetDirectories(topDir)) {
        //Console.WriteLine(taskDir);
        List<Test> tests = Serializer.DeserializeTests(taskDir
            + @"\secret_project\RandomTests.xml");
        MethodInfo secretMethod = Utility.GetMethodDefinition(
            Utility.GetAssemblyForProject(taskDir + @"\secret_project")
            , "Program", "Puzzle");
        foreach (var test in tests) {
            try {
                test.TestOutput = secretMethod.Invoke(null,
                    test.TestInputs.ToArray());
            }
            catch (Exception e) {
            }
        }
    }
}

```

```
        test.TestOutput = e;
    }
}
foreach (var studentDir in Directory.GetDirectories(taskDir)) {
    StringBuilder sb = new StringBuilder();
    sb.AppendLine("projectNo\t#match\t#all\tmetric3");
    if (studentDir.EndsWith("secret_project"))
        continue;
    foreach (var projectDir in Directory.GetDirectories(studentDir)) {
        double match = 0;
        double metric = 0;
        if (!projectDir.Contains("meta_project")) {
            string projectNo = projectDir.Substring(projectDir
                .LastIndexOf("project") + 7);
            MethodInfo method = Utility.GetMethodDefinition(
                Utility.GetAssemblyForProject(projectDir), "Program", "Puzzle");
            foreach (var test in tests) {
                object result;
                try {
                    result = method.Invoke(null, test.TestInputs.ToArray());
                }
                catch (Exception e) {
                    result = e;
                }
                if (result is Exception) {
                    if (test.TestOutput is Exception) {
                        string type1 = ((Exception)result).InnerException
                            .GetType().ToString();
                        string type2 = ((Exception)test.TestOutput)
                            .InnerException.GetType().ToString();
                        if (type1 == type2) {
                            match++;
                        }
                    }
                }
            }
        }
        else {
            if (test.TestOutput is Exception)
                continue;
            if (result == null){
                if (test.TestOutput == null)
                    match++;
            }
            else if (result is Int32){
                if ((int)result == (int)test.TestOutput)
```

```
        match++;
    }
    else if (result is Double){
        if ((double)result == (double)test.TestOuput)
            match++;
    }
    else if (result is String){
        if ((string)result == (string)test.TestOuput)
            match++;
    }
    else if (result is Byte){
        if ((byte)result == (byte)test.TestOuput)
            match++;
    }
    else if (result is Char){
        if ((char)result == (char)test.TestOuput)
            match++;
    }
    else if (result is Double){
        if ((double)result == (double)test.TestOuput)
            match++;
    }
    else if (result is Boolean){
        if ((bool)result == (bool)test.TestOuput)
            match++;
    }
    else if (result is Int32[]){
        int[] array1 = (int[])result;
        int[] array2 = (int[])test.TestOuput;
        if (array1.Length == array2.Length){
            bool equal = true;
            for (int i = 0; i < array1.Length; i++){
                if (array1[i] != array2[i]){
                    equal = false;
                    break;
                }
            }
            if (equal)
                match++;
        }
    }
    else {
        throw new Exception("Not handled return type at "
            + projectDir);
    }
}
```

```
        }
    }
}
metric = match / tests.Count;
sb.AppendLine(projectNo + "\t" + match + "\t"
+ tests.Count + "\t" + metric);
}
}
File.WriteAllText(studentDir + @"\Metric3.txt", sb.ToString());
}
}
```

Mã lệnh 12: Mã lệnh phép đo SSE

```
public static void ComputeMetric2(string topDir) {
    foreach (var taskDir in Directory.GetDirectories(topDir)){
        List<Test> tests = Serializer.DeserializeTests(taskDir
+ @"\secret_project\PexTests.xml");
        MethodInfo secretMethod = Utility.GetMethodDefinition(
Utility.GetAssemblyForProject(taskDir + @"\secret_project")
, "Program", "Puzzle");
        foreach (var test in tests){
            try{
                test.TestOutput = secretMethod.Invoke
                (null, test.TestInputs.ToArray());
            }
            catch (Exception e){
                test.TestOutput = e;
            }
        }
    }
    foreach (var studentDir in Directory.GetDirectories(taskDir)){
        StringBuilder sb = new StringBuilder();
        sb.AppendLine("projectNo\t#match\t#all\tmetric2");
        if (studentDir.EndsWith("secret_project"))
            continue;
        foreach (var projectDir in Directory.GetDirectories(studentDir)){
            double match = 0;
            double metric = 0;
            if (!projectDir.Contains("meta_project")){
                string projectNo = projectDir.Substring(projectDir
.LastIndexOf("project") + 7);
                MethodInfo method = Utility.GetMethodDefinition(
Utility.GetAssemblyForProject(projectDir), "Program", "Puzzle");
                foreach (var test in tests){
```

```
object result;
try {
    result = method.Invoke(null, test.TestInputs.ToArray());
}
catch (Exception e){
    result = e;
}
if (result is Exception){
    if (test.TestOutput is Exception){
        string type1 = ((Exception)result).InnerException.GetType()
            .ToString();
        string type2 = ((Exception)test.TestOutput).InnerException
            .GetType().ToString();
        if (type1 == type2){
            match++;
        }
    }
}
else {
    if (test.TestOutput is Exception)
        continue;
    if (result == null){
        if (test.TestOutput == null)
            match++;
    }
    else if (result is Int32){
        if ((int)result == (int)test.TestOutput)
            match++;
    }
    else if (result is Double){
        if ((double)result == (double)test.TestOutput)
            match++;
    }
    else if (result is String){
        if ((string)result == (string)test.TestOutput)
            match++;
    }
    else if (result is Byte){
        if ((byte)result == (byte)test.TestOutput)
            match++;
    }
    else if (result is Char){
        if ((char)result == (char)test.TestOutput)
            match++;
    }
}
```

```

    }
    else if (result is Double){
        if ((double)result == (double)test.TestOutput)
            match++;
    }
    else if (result is Boolean){
        if ((bool)result == (bool)test.TestOutput)
            match++;
    }
    else if (result is Int32[]){
        int[] array1 = (int[])result;
        int[] array2 = (int[])test.TestOutput;
        if (array1.Length == array2.Length){
            bool equal = true;
            for (int i = 0; i < array1.Length; i++){
                if (array1[i] != array2[i]){
                    equal = false;
                    break;
                }
            }
            if (equal)
                match++;
        }
    }
    else{
        throw new Exception("Not handled return type at "
            + projectDir);
    }
}

}
metric = match / tests.Count;
sb.AppendLine(projectNo + "\t" + match + "\t" + tests.Count
    + "\t" + metric);
}
}
File.WriteAllText(studentDir + @"\Metric2.txt", sb.ToString());
}
}
}

```

Mã lệnh 13: Mã lệnh phép đo PSE

```

public static void ComputeMetric1(string topDir)
{
    foreach (var taskDir in Directory.GetDirectories(topDir))

```

```
{
    foreach (var studentDir in Directory.GetDirectories(taskDir))
    {
        StringBuilder sb = new StringBuilder();
        sb.AppendLine("projectNo\t#pass\t#notpass\t#all\tmetric1");
        if (studentDir.EndsWith("secret_project"))
            continue;
        foreach (var projectDir in Directory.GetDirectories(studentDir))
        {
            double pass = 0;
            double notPass = 0;
            double metric = 0;
            if (projectDir.Contains("meta_project"))
            {
                string projectNo = projectDir.Substring(projectDir
                    .LastIndexOf("meta_project")+12);
                List<Test> tests = Serializer.DeserializeTests
                    (projectDir + @"\PexTests.xml");
                MethodInfo method = Utility.GetMethodDefinition
                    (Utility.GetAssemblyForProject(projectDir), "MetaProgram", "Check");
                foreach (var test in tests)
                {
                    try
                    {
                        object result = method.Invoke(null, test.TestInputs.ToArray());
                        pass++;
                    }
                    catch (Exception e)
                    {
                        if (e.InnerException.Message.Contains("Submission failed"))
                        {
                            notPass++;
                        }
                    }
                }
                metric = pass / (notPass + pass);
                sb.AppendLine(projectNo + "\t" +pass + "\t"+notPass+"\t"
                    +(pass+notPass)+"\t"+metric);
            }
        }
        File.WriteAllText(studentDir + @"\Metric1.txt", sb.ToString());
    }
}
```
