

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: 60 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 TS.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 với thời gian có hạn nên 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	3
1 GIỚI THIỆU	6
1.1 Lý do chọn đề tài	6
1.2 Đối tượng, phạm vi, phương pháp nghiên cứu	7
1.3 Những nghiên cứu có liên quan	9
2 KIẾN THỨC CƠ SỞ	12
2.1 Kiểm thử phần mềm	12
2.2 Sinh ngẫu nhiên dữ liệu thử	15
2.3 Kỹ thuật Dynamic symbolic execution	17
3 ĐO ĐỘ TƯƠNG TỰ VỀ HÀNH VI GIỮA CÁC CHƯƠNG TRÌNH	26
3.1 Hành vi của chương trình	26
3.2 Một số phép đo độ tương tự hành vi	29
3.3 Tiêu chí đánh giá hiệu quả	34
4 THỰC NGHIỆM, ĐÁNH GIÁ, KẾT LUẬN	36
4.1 Dữ liệu thực nghiệm	36
4.2 Công cụ dùng trong thực nghiệm	37
4.3 Đánh giá kết quả thực nghiệm	41
4.4 Khả năng ứng dụng	45
4.5 Kết luận	47
Tài liệu tham khảo	49
A QUYẾT ĐỊNH GIAO LUẬN VĂN	57

<i>MỤC LỤC</i>	4
B Phụ lục XXX	59
C MỘT SỐ MÃ LỆNH QUAN TRỌNG	60

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
DFAs	Automated Grading of DFA Constructions
AAA	Arrange, Act, Assert

Chương 1

GIỚI THIỆU

Chương này trình bày lý do chọn đề tài, mục tiêu nghiên cứu, đối tượng, phạm vi nghiên cứu và những nội dung chính cần nghiên cứu. Qua đó, trình bày nhu cầu thực tiễn về một công cụ hỗ trợ cho việc dạy và học lập trình trong các trường đại học, cao đẳng.

1.1 Lý do chọn đề tài

Giới thiệu chung

Hiện nay, ngành Công nghệ thông tin đang rất phát triển với nhiều chương trình đào tạo lập trình Online, kỹ sư phần mềm... đang trở nên phổ biến. Một số chương trình đào tạo nổi tiếng như Massive Open Online Courses (MOOC) [42], edX [17], Coursera [11], Udacity [61] thu hút nhiều sinh viên theo học. Một số chương trình học lập trình online như Pex4Fun [49] hay Code Hunt [24] đây là những nền tảng học lập trình online thông qua trò chơi.

Những lớp học như vậy thường có nhiều sinh viên tham gia, nhưng chỉ có một vài giảng viên tham gia giảng dạy. Hằng ngày, công việc của giảng viên rất nhiều, họ còn phải thường xuyên kiểm tra, nhắc nhở và phải nghiên cứu giúp đỡ cho sinh viên. Chỉ riêng việc đánh giá kết quả mã lệnh do sinh viên viết đã tốn nhiều thời gian, nếu như bỏ qua thì giảng viên sẽ không theo dõi được quá trình học tập của sinh viên. Sinh viên khi gặp khó khăn trong việc viết mã chương trình, họ có thể nhờ bạn bè hoặc nhờ những người có kinh nghiệm giúp đỡ. Nhưng không phải lúc nào cũng có người bên cạnh để giúp đỡ cho họ, và kinh nghiệm cũng như kiến thức của những người này chưa chắc có thể đáp ứng được mong muốn của sinh viên.

Để giảm bớt khó khăn cho giảng viên và sinh viên, một công cụ hỗ trợ tự động đánh

giá kết quả chương trình của sinh viên với chương trình của giảng viên sẽ giúp cho giảng viên và sinh viên tiết kiệm được thời gian, giúp cho giảng viên quản lý việc học tập của sinh viên được tốt hơn. Sinh viên có thể nhanh chóng biết được chương trình của mình viết đúng hay sai.

Cách thức hoạt động của công cụ này là đánh giá độ tương tự về hành vi của hai chương trình. Công cụ sẽ tính toán để tìm ra các mẫu dữ liệu đầu vào chung cho cả hai chương trình, tiến hành lấy từng mẫu dữ liệu đầu vào chạy đồng thời trên cả hai chương trình và so sánh kết quả đầu ra của hai chương trình. Nếu kết quả đầu ra của hai chương trình có tỷ lệ giống nhau càng cao thì điểm số của sinh viên càng cao. Ngược lại, nếu tỷ lệ càng thấp thì tương ứng với điểm số của sinh viên càng thấp. Dựa trên kết quả này, giảng viên có thể nắm bắt được tình hình học tập của sinh viên và có hướng khắc phục những hạn chế mà sinh viên đang gặp phải. Đây cũng là một cách giúp cho sinh viên không đi lệch khỏi định hướng kiến thức, các kỹ thuật, kỹ năng lập trình và hạn chế được những nguy cơ tiềm ẩn trong cách viết mã lệnh chương trình. Đồng thời giúp tiết kiệm được thời gian cho cả giảng viên và sinh viên.

1.2 Đối tượng, phạm vi, phương pháp nghiên cứu

Mục tiêu nghiên cứu

Mục tiêu nghiên cứu chính

- Đánh giá độ tương tự về hành vi của các chương trình

Mục tiêu nghiên cứu cụ thể

- Tìm hiểu sự tương tự hành vi của chương trình
- Tìm hiểu kỹ thuật, công cụ sinh Test Case tự động và áp dụng kỹ thuật sinh Test Case tự động trên các kỹ thuật đo độ tương tự
- 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

- 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
- Phân tích, đánh giá dựa trên kết quả thực nghiệm

1.3 Những nghiên cứu có liên quan

Phân loại tự động

Automated Grading of DFA Constructions (DFAs) [2], đề xuất một phương pháp tự động so sánh những chỗ sai trong mã nguồn của sinh viên với mã nguồn tham chiếu. Cách tiếp cận của phương pháp này là chọn từng phần, bắt những đoạn cú pháp khác nhau dựa trên khoảng cách của cú pháp và sự khác biệt cơ bản về ngữ nghĩa trên một chuỗi giá trị đầu vào được chấp nhận. Trong luận văn này, cách tiếp cận là dựa trên sự hoạt động của các chương trình.

Automated Feedback Generation for Introductory Programming Assignments [54], đề xuất phương pháp tự động xác định những lỗi nhỏ nhất trong lời giải của sinh viên, và lời giải của sinh viên không chính xác về hành vi so với lời giải tham chiếu. Cách tiếp cận của bài báo đó là tập trung vào việc cung cấp các phản hồi, làm thế nào để sinh viên biết và khắc phục những lỗi cú pháp trong Chương trình của mình so với Chương trình tham chiếu. Trong nội dung nghiên cứu, Luận văn này tập trung vào việc làm thế nào định lượng độ tương tự hành vi của hai chương trình dựa vào các giá trị đầu vào và đầu ra của hai chương trình, không phân biệt hai chương trình có cấu trúc hay cú pháp khác nhau.

Semantic similarity-based grading of student programs [63], đề xuất một giải pháp đó là chuyển đổi Chương trình của học sinh và Chương trình tham chiếu về một dạng chung nhưng không thay đổi ngữ nghĩa, tiến hành so sánh đồ thị sự phụ thuộc vào hệ thống của hai chương 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 tương đương

Một số phương pháp kiểm tra sự tương đương về ngữ nghĩa, hành vi của các chương trình bằng cách sử dụng đồ thị biểu hiện sự phụ thuộc chương trình vào hệ thống

[4] [6], phụ thuộc giá trị đầu vào, đầu ra [27], tóm tắt biểu tượng [48]. Tất cả các phương pháp kiểm tra độ tương đương này đều trả về giá trị Boolean, và một số phương pháp cho kết quả về hành vi là như nhau.

Phương pháp tự động xác định những đoạn mã tương đương nhau về chức năng thông qua các thử nghiệm ngẫu nhiên [29]. Cách tiếp cận này xem xét 2 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 2 đoạn mã.

Phản hồi dựa trên trường hợp các thử nghiệm

Phương pháp tự động phân loại chương trình, bài tập đơn giản [23], cách tiếp cận của phương pháp này là so sánh dữ liệu được tạo ra trong quá trình thực thi chương trình với dữ liệu đã lưu trữ trước đó.

Phương pháp phân loại chương trình của sinh viên sử dụng ASSYST [28], tác giả đề xuất cách tiếp cận đó là tự động kiểm tra tính chính xác của chương trình và kiểu lập trình như mô đun, độ phức tạp và hiệu quả.

Pex4Fun sử dụng DSE để tạo ra các giá trị đầu vào thử nghiệm cho các chương trình và các chương trình khi thực thi các giá trị này sẽ cho ra các giá trị đầu ra khác nhau.

Phát hiện các đoạn mã giống nhau

Những nhà nghiên cứu đã đề xuất các phương pháp tiếp cận tính toán các điểm tương đồng của các đoạn mã và tự động nhận diện những đoạn mã giống nhau, như mã hóa báo cáo [31], cú pháp cây trừu tượng [5], biểu đồ phụ thuộc chương trình [34], số liệu dựa trên số lượng cú pháp [12] [39]. Các cách tiếp cận này tập trung vào các đoạn của mã nguồn và tính toán các điểm tương đồng dựa trên việc biểu diễn cú pháp hoặc ngữ nghĩa của các đoạn mã.

Tổng kết chương

Chương 1 giới thiệu tổng quan về lý do và mục đích chọn đề tài, đối tượng, phạm vi và phương pháp nghiên cứu thực hiện, những nội dung chính cần nghiên cứu và một số nghiên cứu khác có liên quan đến đề tài.

Chương 2

KIẾN THỨC CƠ SỞ

Chương này trình bày những kiến thức cơ sở để triển khai luận văn này, bao gồm kiến thức về kiểm thử phần mềm, kỹ thuật sinh ngẫu nhiên dữ liệu thử, và kỹ thuật Dynamic Symbolic Execution.

2.1 Kiểm thử phần mềm

Giới thiệu

Trong nền kinh tế 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 cho phần mềm là một công việc đòi hỏi nhiều nỗ lực, công sức, cũng như tiêu tốn nhiều hơn chi phí trong việc phát triển phần mềm. Hiện nay, một sản phẩm phần mềm chất lượng có thể được nhiều người sử dụng biết đến, nó mang lại hiệu quả tích cực trong công việc của người sử dụng. Tuy nhiên, một phần mềm kém chất lượng sẽ gây thiệt hại về kinh tế cũng như tiến độ công việc của người sử dụng. Phần mềm phải luôn đảm bảo được sự ổn định, không phát sinh lỗi trong quá trình sử dụng.

Việc 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 người phát triển phần mềm và người sử dụng thấy được hệ thống đã đáp ứng yêu cầu đặt ra.

Các phương pháp kiểm thử

Có nhiều phương pháp để kiểm thử phần mềm, nhưng trọng tâm là hai phương pháp kiểm 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 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 chạy chương trình để điều 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 luôn 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.

Các chiến lược kiểm thử

Kiểm thử phần mềm có nhiều chiến lược để kiểm thử, trong đó có hai chiến lược 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. Kiểm thử hộp đen tập trung vào tìm các trường hợp mà chương trình không thực hiện theo các đặc tả kỹ thuật. Kiểm thử viên hộp đen cố gắng tìm ra những lỗi mà lập trình viên không tìm ra. Tuy nhiên, phương pháp kiểm thử này cũng có mặt hạn chế của nó, kiểm thử viên không biết

các phần mềm được 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 ca kiểm thử duy nhất, 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.

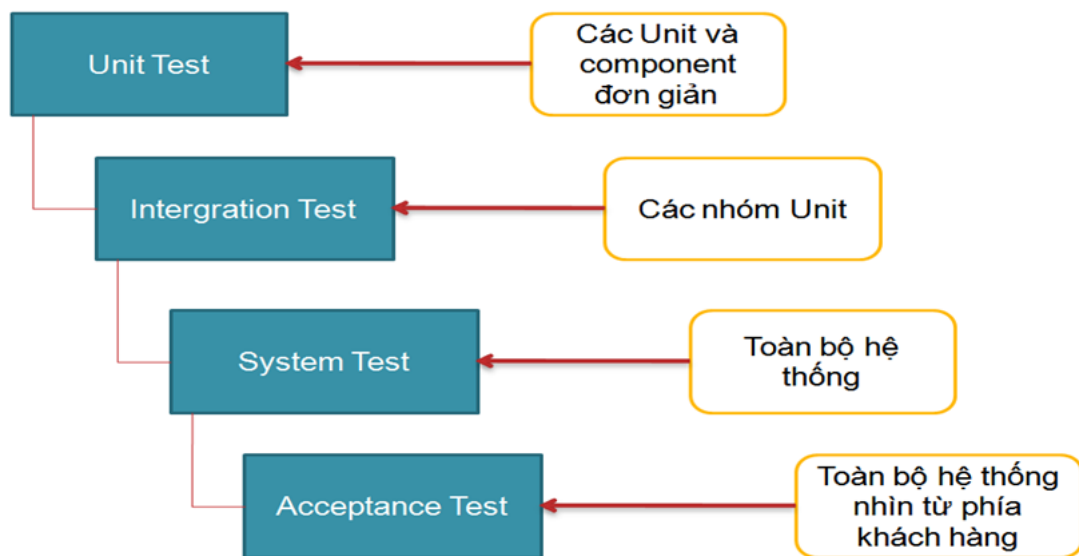
Kiểm thử hộp trắng – White box. Đây là một chiến lược kiểm thử khác, trái ngược hoàn toàn với kiểm thử hộp đen, kiểm thử hộp trắng hay còn gọi là kiểm thử hướng logic của phần mềm. Cách kiểm thử này cho phép tạo ra dữ liệu thử nghiệm từ việc kiểm tra, khảo sát cấu trúc bên trong và kiểm thử tính logic của chương trình. Dữ liệu thử nghiệm có độ phủ lớn, đảm bảo được 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.

Phương pháp kiểm thử hộp trắng cũng có thể được sử dụng để đánh giá một bộ kiểm thử được tạo với các phương pháp kiểm thử hộp đen. Trong các phép đo độ tương tự của đề tài, phương pháp kiểm thử hộp trắng là một phương pháp quan trọng, được áp dụng để đo hành vi của các chương trình.

Các cấp độ kiểm thử trong kiểm thử phần mềm

Kiểm thử phần mềm gồm có các cấp độ kiểm thử như sau:

- Kiểm thử đơn vị
- Kiểm thử tích hợp
- Kiểm thử hệ thống
- Kiểm thử chấp nhận sản phẩm



Hình 2.1: Sơ đồ các cấp độ kiểm thử

Đảm bảo chất lượng phần mềm

Định nghĩa theo Daniel Galin [18]: Đảm bảo chất lượng phần mềm là một tập hợp các hành động được lên kế hoạch một cách hệ thống để cung cấp đầy đủ thông tin quá trình phát triển phần mềm, các yêu cầu chức năng kỹ thuật cũng như các yêu cầu quản lý theo lịch trình và hoạt động trong giới hạn.

2.2 Sinh ngẫu nhiên dữ liệu 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).

Ví dụ: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);  
    }  
}
```

Đây 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 và hạn chế

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 khác 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.

Một kỹ thuật khác giúp khắc phục được hạn chế của kiểm thử ngẫu nhiên đó là kỹ thuật thực thi tượng trưng [33]. Thực thi tượng trưng là một kỹ thuật xây dựng các ràng buộc dựa vào các điều kiện tại các nút nhánh của chương trình, giải quyết các ràng buộc đó để sinh ra các giá trị đầu vào của chương trình. Thực thi các giá trị đầu vào này, chúng ta có thể duyệt được tất cả các nhánh của chương trình.

2.3 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	

Ví dụ:

 Mã lệnh 2.2: Minh họa kỹ thuật DSE

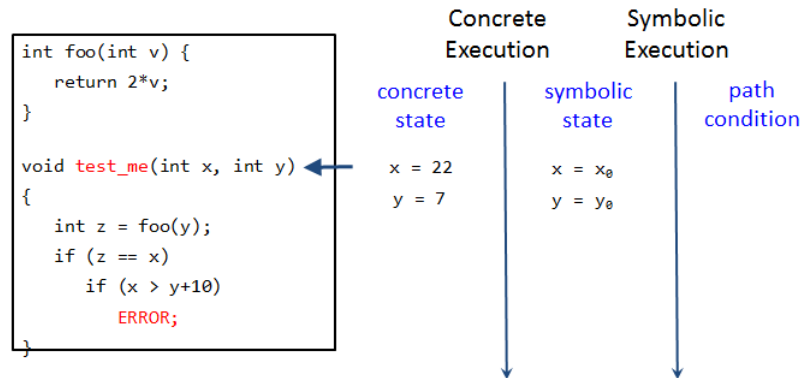
```

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;
}
  
```

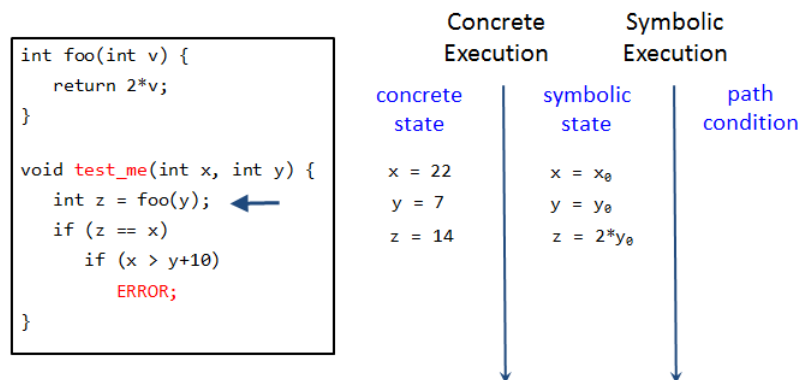
Trong ví dụ trên, chúng ta xem xét hàm *test_me* với hai tham số đầu vào là *intx* và *inty*, và hàm này không có giá trị trả về. Cách thức làm việc của DSE trên hàm *test_me* như sau:

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: x bằng một số x_0 và y bằng một số y_0 .



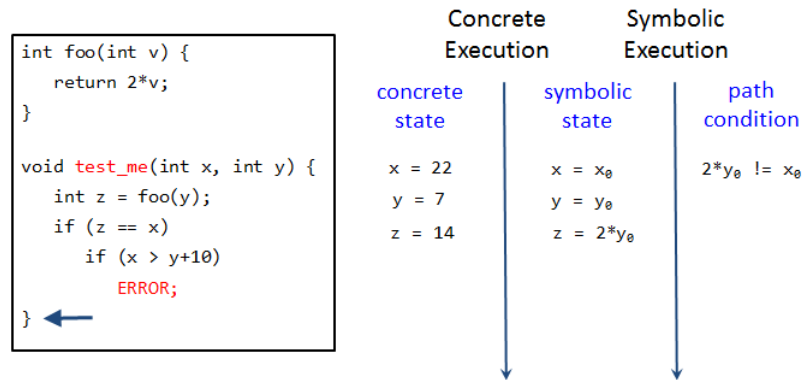
Hình 2.2: DSE khởi tạo các giá trị đầu vào

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



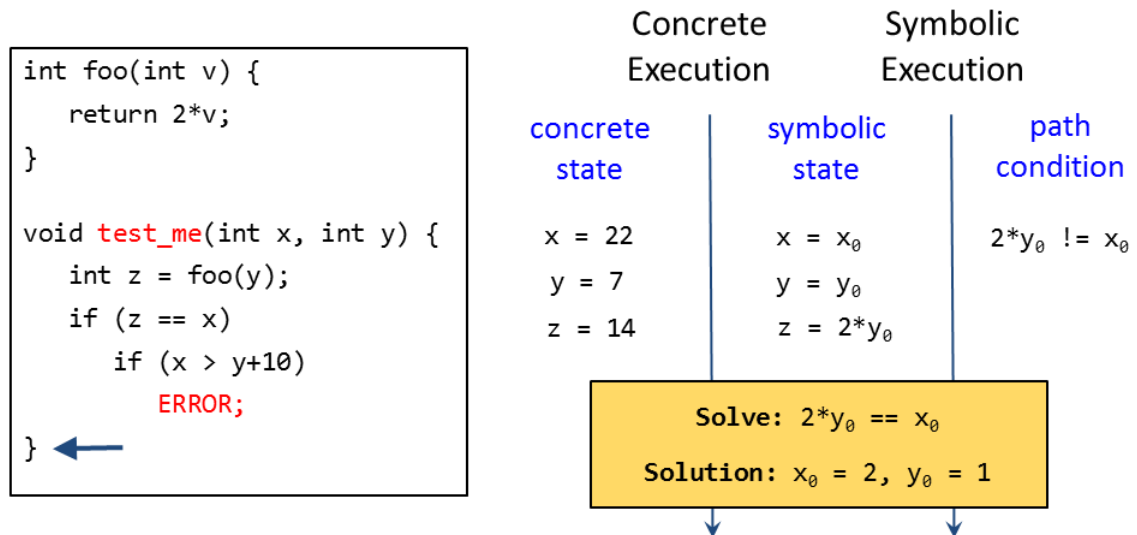
Hình 2.3: Số nguyên Z được gán bằng hàm foo(y)

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



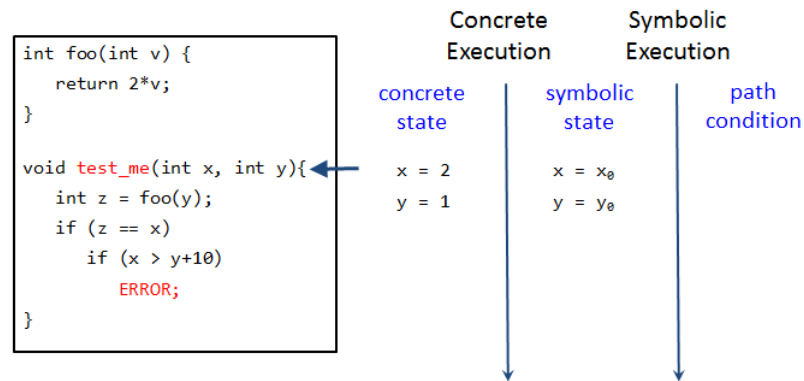
Hình 2.4: Với điều kiện $z == x$, giá trị của $z \neq x$ nên DSE chạy theo nhánh *false*

Sau khi kết thúc chương trình, DSE sẽ quay trở lại điểm nhánh gần nhất và cố gắng 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 $2 * y_0 = x_0$ thì hai số nguyên này sẽ là $x_0 = 2$ và $y_0 = 1$.



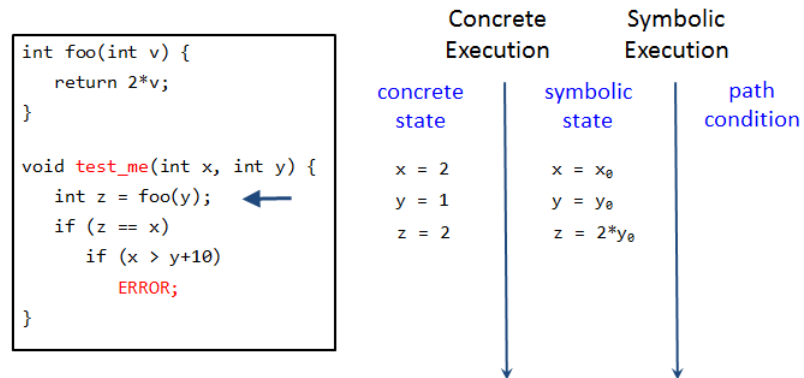
Hình 2.5: Các giá trị DSE sinh ra sau khi thực thi chương trình lần 1

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 $y = y_0$.

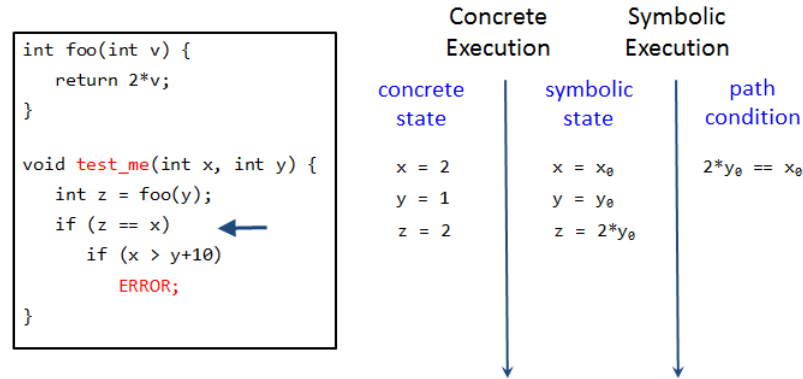


Hình 2.6: DSE khởi động lại hàm test_me

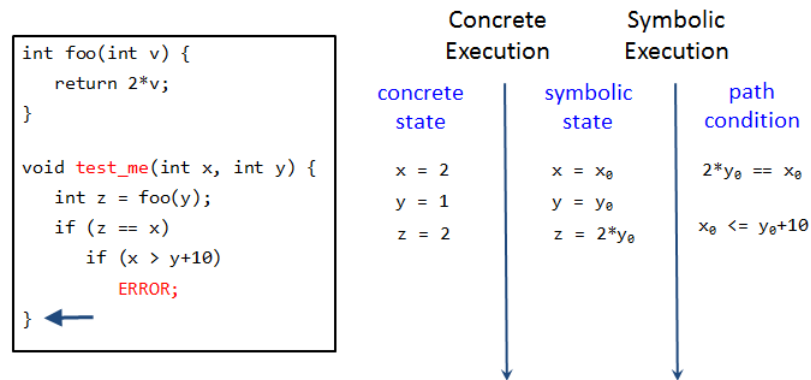
Sau khi thực hiện dòng đầu tiên, z có giá trị cụ thể 2 và giá trị biểu tượng $2 * y_0$.

Hình 2.7: Thực hiện dòng đầu tiên $int z = foo(y)$

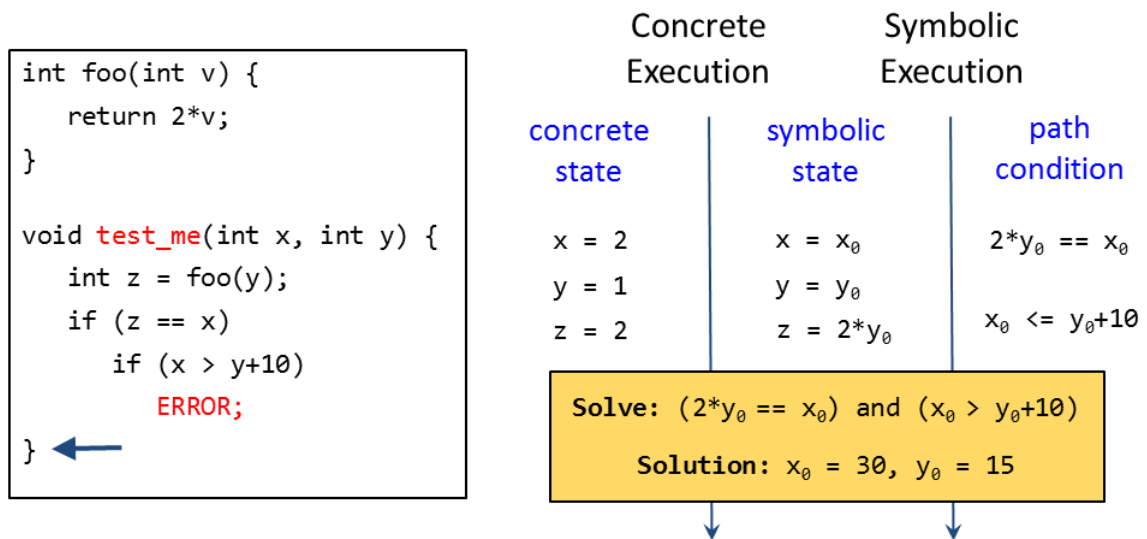
Ở dòng kế tiếp, chúng ta kiểm tra tình trạng nhánh $z == x$. Trong trường hợp này, điều kiện là đúng vì vậy điều kiện đường dẫn của chúng ta trở thành $2 * y_0 == x_0$. Sau đó DSE kiểm tra dòng tiếp theo của nhánh *true*.

Hình 2.8: 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 ứng $x_0 \leq y_0 + 10$ vào điều kiện đường dẫn, đây là sự phủ định của điều kiện nhánh mà DSE phát hiện là *false*.

Hình 2.9: DSE lấy nhánh *false* kết thúc chương trình

Vì DSE đã đến cuối chương trình, nó phủ nhận ràng 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 $2 * y_0 == x_0 \text{ AND } x_0 > y_0 + 10$. Để thỏa những ràng buộc này, DSE trả về $x_0 = 30$ và $y_0 = 15$.

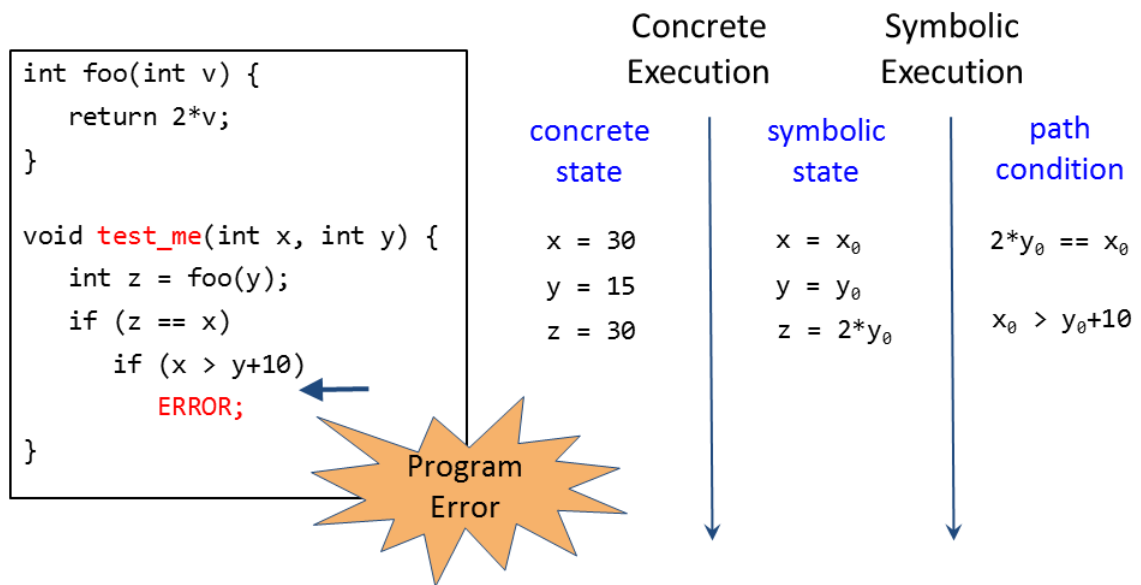


Hình 2.10: Các giá trị DSE sinh ra sau khi thực thi chương trình lần 2

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 2.11: 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 đã 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 [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	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/khooyip/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

Tổng kết chương

Nội dung trong Chương 2, tôi đã trình bày sơ lược những kiến thức cơ bản như: Kỹ thuật kiểm thử phần mềm; các kỹ thuật sinh dữ liệu thử; kỹ thuật Dynamic symbolic execution. Những kiến thức này giúp chúng ta có cái nhìn tổng quan về cách thức kiểm thử phần mềm, phương pháp giải quyết các ràng buộc trong chương trình để tạo ra các giá trị đầu vào thử nghiệm có độ phủ cao.

Chương 3

ĐO ĐỘ TƯƠNG TỰ VỀ HÀNH VI GIỮA CÁC CHƯƠNG TRÌNH

Chương này trình bày một số định nghĩa liên quan đến hành vi chương trình, các phép đo độ tương tự hành vi của chương trình và tiêu chí đánh giá các phép đo này.

3.1 Hành vi của chương trình

Để định lượng hai chương trình tương tự nhau, chúng ta nghiên cứu các định nghĩa liên quan 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 3.1: 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 3.2: 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 3.1 và 3.2 có tham số đầu vào cùng kiểu giá trị *int*. Mã lệnh 3.1 sử dụng cấu trúc *switch...case*, mã lệnh 3.2 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 $\text{exec}(P_1, I) = \text{exec}(P_2, I)$.

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 3.3: Chương trình P₁

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

Mã lệnh 3.4: Chương trình P₂

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

Mã lệnh 3.3 và Mã lệnh 3.4 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 3.5: 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 3.6: 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 3.5 và Mã lệnh 3.6 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 $\text{exec}(P_1, I_s) = \text{exec}(P_2, I_s)$ và khác nhau $\forall j \in I \setminus I_s$, ký hiệu $\text{exec}(P_1, j) \neq \text{exec}(P_2, j)$

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

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 3.7: Chương trình P_1

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

Mã lệnh 3.8: Chương trình P_2

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

Chúng ta thấy đoạn Mã lệnh 3.7 và 3.8 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*($x == \text{"XYZ"}\text{)return}0$; 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 $M_{SSE}(P_1, P_2) = |I_a| / |I_s|$.

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 3.9: 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 3.10: 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 3.9 và Mã lệnh 3.10 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.

3.3 Tiêu chí đánh giá hiệu quả

Để đánh giá độ hiệu quả của các phép đo, chúng ta có thể áp dụng những tiêu chí cơ bản sau:

- Tốc độ xử lý
- Sử dụng tài nguyên
- Độ phủ của dữ liệu thử
- Kết quả đánh giá độ tương tự hành vi

Phép đo **RS** sử dụng kỹ thuật lấy ngẫu nhiên giá trị thử nghiệm trong miền giá trị đầu vào của cả hai Chương trình nên tốc độ xử lý của phép đo **RS** nhanh, đơn giản và sử dụng ít tài nguyên. Nhưng độ phủ dữ liệu thử nghiệm do phép đo **RS** tạo ra không cao, không phủ hết các trường hợp có thể thực thi của chương trình nên kết quả đánh giá độ tương tự hành vi của Chương trình đạt mức tương đối so với hành vi thực tế.

Phép đo **SSE** là một phép đo cải tiến của phép đo **RS**, khi sử dụng kỹ thuật **DSE** dựa trên Chương trình tham chiếu để tạo các giá trị đầu vào thử nghiệm. Vì phải khám tất cả các nhánh đường đi của Chương trình tham chiếu nên tốc độ xử lý của phép đo **SSE** sẽ chậm và chiếm nhiều tài nguyên hơn phép đo **RS**. Tập dữ liệu đầu vào thử nghiệm được tạo bởi phép đo **SSE** có khả năng phủ tất cả các nhánh của Chương trình tham chiếu nên kết quả đánh giá độ tương tự hành vi của phép đo **SSE** sẽ chính xác hơn kết quả đánh giá độ tương tự hành vi của phép đo **RS**.

Phép đo **PSE** sử dụng kỹ thuật **DSE** khám phá tất cả các nhánh đường đi của Chương trình kết hợp để tạo ra tập dữ liệu đầu vào thử nghiệm chung cho cả hai chương trình. Vì vậy, phép đo **PSE** sẽ tốn nhiều thời gian thực thi chương trình và chiếm nhiều tài nguyên hơn phép đo **SSE**. Dữ liệu thử nghiệm của phép đo **PSE** sẽ có độ phủ cao hơn phép đo **SSE**, vì tất cả các giá trị đầu vào thử nghiệm có khả năng phủ tất cả các nhánh của Chương trình tham chiếu và Chương trình cần tìm. Kết quả đánh giá độ tương tự hành vi của phép đo **PSE** sẽ chính xác hơn kết quả đánh giá độ tương tự của phép đo **SSE**.

Tổng kết chương

Nội dung chính được trình bày trong chương này bao gồm những 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, 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. Qua đó, giới thiệu một số tiêu chí đánh giá hiệu quả các kỹ thuật đo.

Chương 4

THỰC NGHIỆM, ĐÁNH GIÁ, KẾT LUẬN

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.

4.1 Dữ liệu thực nghiệm

Để tiến hành làm thực nghiệm những nội dung nghiên cứu 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 thiết kế.

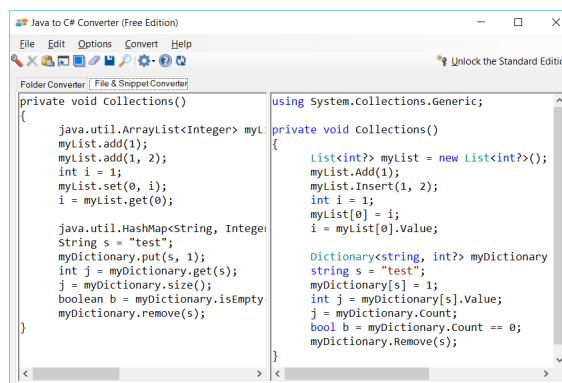
Code Hunt [24] là một game về lập trình, được sử dụng cho các cuộc thi viết mã và thực hành các kỹ năng lập trình do Microsoft phát triển. Code Hunt dựa trên công cụ Pex, ứng dụng kỹ thuật DSE khám phá các nhánh đường đi của chương trình để suy ra giá trị đầu vào có độ phủ cao. Mã Code Hunt là một được xử lý trực tuyến, trong đó mỗi câu đố 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 của câu hỏi. Hiện Code Hunt đã được hơn 350.000 người chơi sử dụng tính đến tháng 8 năm 2016. Dữ liệu từ các cuộc thi gần đây đã được công khai và cho phép những người quan tâm đến Code Hunt tải về để phân tích và nghiên cứu trong cộng đồng giáo dục.

Tập dữ liệu Code Hunt là một tập dữ liệu chứa các chương trình do sinh viên trên toàn thế giới viết, 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 cho đề tài của tôi, tôi đã thực hiện chuyển đổi những chương trình



Hình 4.1: Giao diện viết chương trình của Code Hunt

sử dụng ngôn ngữ lập trình Java thành ngôn ngữ C# bằng công cụ chuyển đổi của hãng Tangible Software Solutions, loại bỏ một số chương trình lỗi và không phù hợp với đề tài.



Hình 4.2: Chuyển đổi code Java sang C#

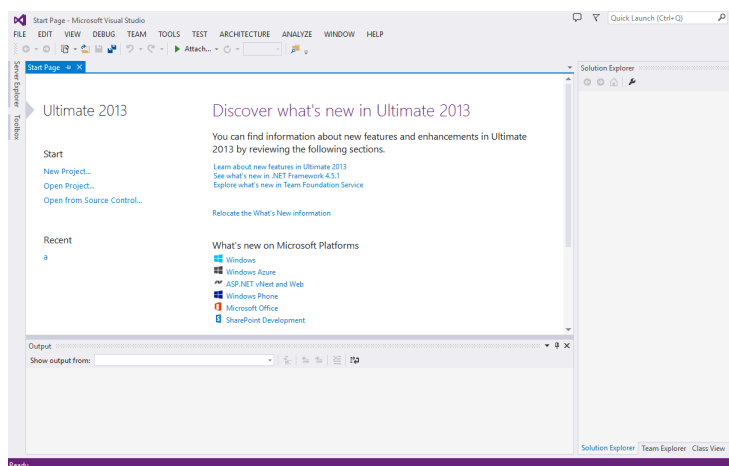
4.2 Công cụ dùng trong thực nghiệm

Trong quá trình làm thực nghiệm, tôi đã sử dụng một số công cụ để làm ví dụ minh họa cho các kỹ thuật đo độ tương tự hành vi của chương trình, cụ thể như sau:

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.



Hình 4.3: Giao diện phần mềm Visual studio 2013

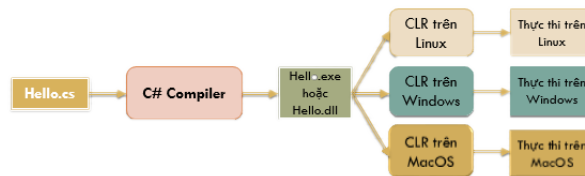
Visual Studio hỗ trợ nhiều ngôn ngữ lập trình khác nhau và cho phép trình biên tập mã và gỡ lỗi để hỗ trợ (mức độ khác nhau) hầu như mọi ngôn ngữ lập trình. Các ngôn ngữ tích hợp gồm có C, C++ và C++/CLI (thông qua Visual C++), VB.NET (thông qua Visual Basic.NET), C# (thông qua Visual C#) và F# (như 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.

Ngôn ngữ lập trình C#

Tổng quan

Ngôn ngữ lập trình C# 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 4.4: Quá trình dịch chương trình trong C#

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

Công cụ sinh dữ liệu thử Pex

Giới thiệu

Khái niệm về DSE và các ứng dụng sử dụng kỹ thuật DSE đã có từ lâu, nhưng Pex là một ứng dụng mở rộng hơn so với các phiên bản DSE trước. Trong Visual Studio, Pex đã được tích hợp như một Add-in, và có thể tạo ra các test case kết hợp với các bộ kiểm thử khác nhau như NUnit và MSTest.

Cũng như với Unit Test, ta có thể viết các lớp kiểm thử chứa các ca kiểm thử tham số hóa. Với sự hỗ trợ của Pex ta có thể thực thi các ca kiểm thử tham số hóa đó. Tuy nhiên không giống việc thực thi các lớp kiểm thử chứa các Unit Test, Pex chỉ thực thi được một ca kiểm thử tham số hóa trong mỗi lần chạy.

Mã lệnh 4.1: Ca kiểm thử tham số sử dụng 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);
}
```

Lựa chọn đầu vào kiểm thử với Pex

Để có thể sinh các đầu vào cụ thể cho các tham số Unit test, Pex cần phải phân tích chương trình với các tham số kiểm thử này. Có 2 kỹ thuật phân tích chương trình đó là:

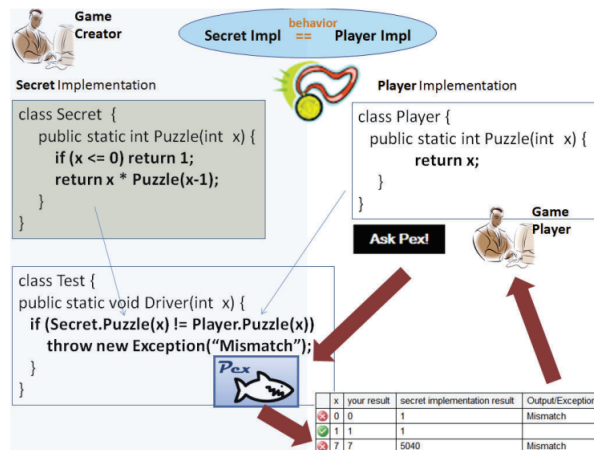
- Phân tích tĩnh (static analysis): Kiểm chứng một tính chất nào đó của chương trình bằng việc phân tích tất cả các đường đi thực thi. Kỹ thuật này coi các cảnh báo (violations) là các lỗi (error).
- Phân tích động (dynamic analysis): Kiểm chứng một tính chất bằng việc phân tích một số đường đi thực thi. Đây là một kỹ thuật phân tích động hỗ trợ việc phát hiện ra các lỗi (bugs) nhưng không khẳng định được rằng có còn những lỗi khác hay không. Các kỹ thuật này thường không tìm ra được tất cả các lỗi.

Pex cài đặt một kỹ thuật phân tích chương trình bằng cách kết hợp cả hai kỹ thuật phân tích chương trình ở trên gọi là DSE [67, 21]. Về bản chất Pex là một công cụ hỗ trợ kỹ thuật kiểm thử hộp trắng (white-box testing). Tương tự như kỹ thuật phân tích chương trình tĩnh, Pex chứng minh được rằng một tính chất được kiểm chứng trong tất cả các đường đi khả thi. Pex chỉ báo cáo (reporting) về các lỗi thực sự như với kỹ thuật phân tích chương trình động.

Pex sử dụng bộ xử lý ràng buộc Z3 [15] kết hợp với các lý thuyết toán học khác như hàm chưa định nghĩa, lý thuyết mảng, bit-vector [35] để giải quyết ràng buộc sinh ra

trong quá trình thực thi tượng trưng động và sinh ra các đầu vào kiểm thử cụ thể cho tham số kiểm thử.

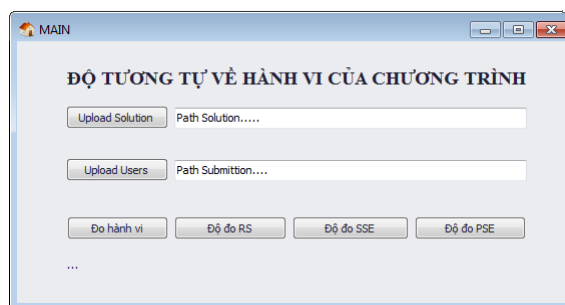
Mô hình ứng dụng Pex



Hình 4.5: Mô hình ứng dụng Pex

4.3 Đánh giá kết quả thực nghiệm

Nội dung Chương 2 và Chương 3 đã trình bày một số lý thuyết về kiểm thử phần mềm, các kỹ thuật sinh dữ liệu thử nghiệm, và các kỹ thuật đo độ tương tự hành vi chương trình **RS**, **SSE**, **PSE**. Áp dụng những lý thuyết trên, tôi thực nghiệm bằng cách xây dựng một ứng dụng, mô tả quá trình hoạt động của các kỹ thuật đo, và đánh giá kết quả các kỹ thuật đo, kết quả thực nghiệm đạt được như sau:



Hình 4.6: Giao diện màn hình chính

Hai nút đầu tiên, cho phép chúng ta chọn file chương trình tham chiếu và files cần tính độ tương tự. Bên dưới là các chút chức năng đo hành vi, và tính toán độ tương tự hành vi của chương trình theo các độ đo khác nhau RS, SSE và PSE.

Một số files mẫu chương trình tham chiếu và chương trình cần tính độ tương tự (chương trình của sinh viên) có nội dung như sau:

Mã lệnh 4.2: 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;
            default: y = x - 100; break;
        }
        return y;
    }
}
```

Mã lệnh 4.3: 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 4.4: 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;

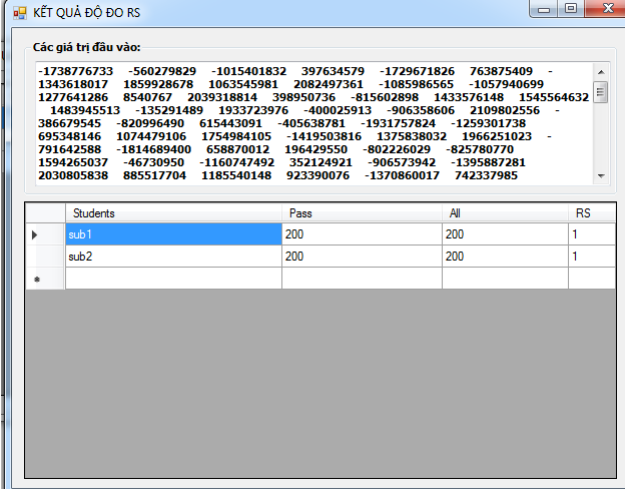
        else return y = x - 100;

    }

}

```

Kết quả độ đo RS



The screenshot shows a window titled "KẾT QUẢ ĐỘ ĐO RS". It contains a text area with a list of 40 random integers. Below this is a table with 4 columns: "Students", "Pass", "All", and "RS". The table has two rows of data, both showing a "Pass" of 200, "All" of 200, and "RS" of 1.

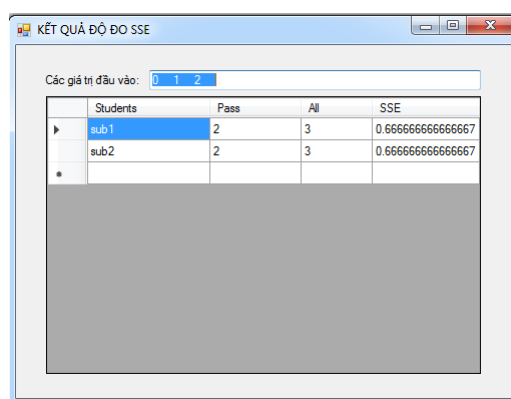
Students	Pass	All	RS
sub1	200	200	1
sub2	200	200	1

Hình 4.7: Kết quả độ đo RS

Dựa vào kết quả độ đo RS ở trên, chúng ta có danh sách các giá trị đầu vào được sinh ngẫu nhiên từ miền giá trị đầu vào của các chương trình. Hai chương trình của sinh viên được đánh giá là tương đương với chương trình tham chiếu và đều có kết quả là 1. Trong khi đó, điều kiện của Chương trình tham chiếu (*case2* : $y+ = 8; break;$), cấu trúc điều kiện của sinh viên thứ nhất (*case2* : $y+ = 8; break;$), cấu trúc điều kiện

của sinh viên thứ hai ($(if(x == 2) return y* = 4;)$ và $(if(x == 3) return y* = 6;)$, mã lệnh của hai sinh viên đều có hành vi khác biệt so với Chương trình tham chiếu. Kết quả độ đo **RS** cho kết quả ở mức tương đối, giá trị thử nghiệm không phủ hết các trường hợp chương trình có khả năng thực thi.

Kết quả độ đo SSE



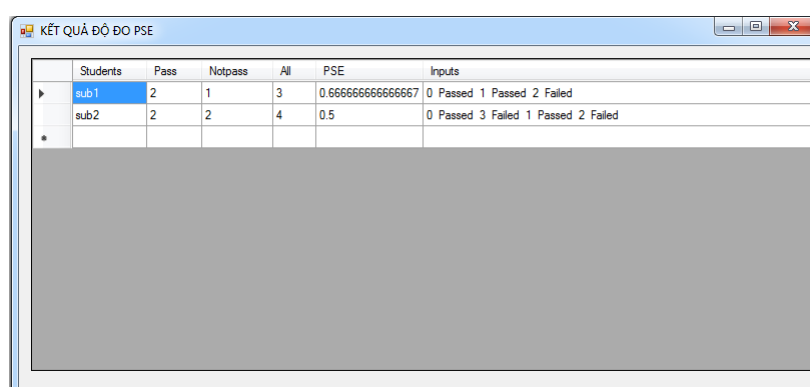
Students	Pass	All	SSE
sub1	2	3	0.6666666666666667
sub2	2	3	0.6666666666666667

Hình 4.8: Kết quả độ đo SSE

Phân tích Chương trình tham chiếu với kỹ thuật DSE chúng ta có các giá trị đầu vào thử nghiệm lần lượt là 0, 1, 2. Các giá trị đầu vào thử nghiệm này khi thực thi trên chương trình của hai sinh viên cho 2 kết quả đầu giống nhau trên tổng số 3 kết quả so với chương trình tham chiếu, đạt tỷ lệ 0,66. Kết quả độ đo SSE cho chúng ta một kết quả chính xác hơn kết quả độ đo RS. Những phép đo SSE lại khi không xem xét các hành vi trong chương trình của sinh viên, không tạo ra được các giá trị đầu vào thử nghiệm có khả năng phủ hết các nhánh trong Chương trình của sinh viên. Trong khi Chương trình của sinh viên hiện có hành vi khác biệt so với Chương trình tham chiếu.

Kết quả độ đo PSE

Kết quả độ đo PSE đã thay đổi so với kết quả độ đo SSE. PSE tạo ra 4 giá trị đầu vào thử nghiệm trên Chương trình kết hợp giữa Chương trình tham chiếu và Chương



	Students	Pass	Notpass	All	PSE	Inputs
▶	sub1	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
*						

Hình 4.9: Kết quả độ đo PSE

trình của sinh viên thứ 2, lần lượt là 0, 1, 2, 3. Kết quả phép đo PSE cho sinh viên thứ nhất đạt 0.66, và kết quả phép đo PSE của sinh viên thứ hai đạt 0,5, kết quả này chính xác nếu chúng ta so sánh với việc phân tích thủ công. Các giá trị thử nghiệm đáp ứng được mục tiêu phủ hết các nhánh của Chương trình cần tính và Chương trình tham chiếu.

4.4 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, các 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 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 thường 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.

Hướng phát triển

Qua quá trình nghiên cứu và triển khai thực nghiệm, trong tương lai đề tài hướng tới phát triển thành một ứng dụng hoàn chỉnh với việc bổ sung và hoàn thiện một số chức năng như sau:

- Phát triển ứng dụng có thể chạy trên Web
- Quản lý kết quả học tập sinh viên
- Thêm chức năng đánh giá, xếp hạng tự động
- Thêm chức năng gợi ý giải pháp lập trình
- Cải tiến các độ đo để cho kết quả tốt hơn và nhanh hơn
- Phát triển thêm các nền tảng lập trình khác như Java, C++..

4.5 Kết luận

Qua quá trình nghiên cứu đề tài, có thể thấy rằng việc phát triển và ứng dụng các kỹ thuật đo đang dần trở nên phổ biến trong các chương trình giáo dục và đào tạo lập trình viên online. Những lợi ích, hiệu quả của việc đánh giá độ tương tự hành vi mang lại là rất thiết thực. Các kỹ thuật này không chỉ giúp quá trình giảng dạy của giảng viên được thuận lợi hơn, tiết kiệm được thời gian cũng như công sức trong công tác quản lý. Ngoài ra, sinh viên có được một môi trường tốt để tự rèn luyện, nâng cao các kỹ năng lập trình của bản thân. Việc tạo động lực giúp sinh viên có sự hứng thú và đam mê lập trình là rất cần thiết. Một khi sinh viên có tư duy và kỹ năng lập trình tốt, sinh viên sẽ tự tin vào năng lực của bản thân để tiếp tục phát triển sự nghiệp sau khi ra trường.

Đề tài đã thực hiện nghiên cứu nhiều vấn đề, như nghiên cứu kiểm thử phần mềm, sinh ngẫu nhiên dữ liệu thử, hay kỹ thuật DSE một kỹ thuật được ứng dụng trong công cụ PEX của Microsoft để giải quyết các ràng buộc sinh ra các tham số đầu vào thử nghiệm có độ phủ cao, nghiên cứu các phép đo RS, SSE, PSE để đo độ tương tự hành vi của chương trình. Trên cơ sở lý thuyết các kỹ thuật đo, xây dựng một công cụ để minh họa cho các phép đo. Kết quả của các phép đo là tương đối tốt, tạo ra nhiều hướng phát triển trong tương lai.

Trong quá trình thực hiện đề tài, bản thân tôi cũng gặp phải rất nhiều khó khăn như: Lượng kiến thức cơ sở cần phải nghiên cứu để triển khai thực hiện đề tài rất nhiều;

cùng với đó là kinh nghiệm của bản thân tôi trong việc thực hiện các đề tài chưa có. Tuy nhiên, với sự động viên và tận tình giúp đỡ của giáo viên hướng dẫn, đề tài đã đạt được mục tiêu đề ra, có thể mở ra nhiều hướng nghiên cứu và phát triển khác của đề tài như cải tiến các kỹ thuật đo, kết hợp với kỹ thuật DSE để có kết quả các phép được chính xác hơn, nhạy hơn. Tiếp tục nghiên cứu, phát triển trên các ngôn ngữ khác như Java, C++ ... và chạy được trên nhiều nền tảng PC, Web.

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 white-box 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 multilingualistic 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 soft-

- ware 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.

Phụ lục A

QUYẾT ĐỊNH GIAO LUẬN VĂN

Quyết định

Phụ lục B

Phụ lục XXX

Nội dung phụ lục viết ở đây.

Phụ lục C

MỘT SỐ MÃ LỆNH QUAN TRỌNG

1. Mã nguồn 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);
    }
}
}
}
}

```

2. Mã nguồn 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);
}
}

```

3. Mã nguồn 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);
                }
            }
        }
    }
}

```

4. Mã nguồn 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);
    }
}
```

5. Mã nguồn thực thi DSE trên Chương trình tham chiếu

```
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);
                }
            }
        }
    }
}
```

```
}
```

6. Mã nguồn thực thi DSE trên Chương trình kết hợp

```
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));
                }
            }
        }
    }
}
```

```

        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());
}
}
}

```

8. Mã nguồn 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\tt#match\tt#all\ttmetric2");
    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.TestOuput is Exception){
            string type1 =
                ((Exception)result).InnerException.GetType().ToString();
            string type2 =
                ((Exception)test.TestOuput).InnerException.GetType().ToString();
            if (type1 == type2){
                match++;
            }
        }
    }
    else {
        if (test.TestOuput is Exception)
            continue;
        if (result == null){
            if (test.TestOuput == null)
                match++;
        }
        else if (result is Int32){
            if ((int)result == (int)test.TestOuput)
                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 + @"\Metric2.txt",
    sb.ToString());
}
}
}

```

9. Mã nguồn 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\tt#match\tt#all\ttmetric3");
    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.TestOuput is Exception) {
            string type1 =
                ((Exception)result).InnerException.GetType().ToString();
            string type2 =
                ((Exception)test.TestOuput).InnerException.GetType().ToString();
            if (type1 == type2) {
                match++;
            }
        }
    }
}
else {
    if (test.TestOuput is Exception)
        continue;
    if (result == null){
        if (test.TestOuput == null)
            match++;
    }
    else if (result is Int32){
        if ((int)result == (int)test.TestOuput)
            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());  
}  
}
```
