

Độ tương tự hành vi của chương trình
và thực nghiệm

Đỗ Đăng Khoa

Ngày 20 tháng 6 năm 2018

Todo list

Trích dẫn bài báo	32
scan quyet dinh sang pdf và include	33
Đưa một số đoạn code quan trọng	35

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 những kiến thức và kinh nghiệm quý báu để giúp tôi có đầy đủ kiến thức và nghị lực hoàn thành luận văn này.

Và tôi xin cảm ơn bạn bè, đồng nghiệp và những người thân trong gia đình đã tin yêu, động viên giúp tôi thêm nghị lực trong quá trình học tập và nghiên cứu.

Mặc dù đã cố gắng rất nhiều trong việc thực hiện luận văn, song với thời gian có hạn, nên luận văn 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.

Một lần nữa, tôi xin chân thành cảm ơn!

HỌC VIÊN

**Đỗ Đăng
Khoa**

Mục lục

Mục lục	6
1 Giới thiệu	8
1.1 Lý do chọn đề tài	8
1.2 Đối tượng, phạm vi, phương pháp nghiên cứu	9
1.3 Những nghiên cứu có liên quan	10
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	15
3 Đo độ tương tự về hành vi giữa các chương trình	18
3.1 Hành vi của chương trình	18
3.2 Một số phép đo độ tương tự hành vi	19
4 Thực nghiệm, đánh giá, kết luận	23
4.1 Dữ liệu thực nghiệm	23
4.2 Công cụ dùng trong thực nghiệm	23
4.3 Đánh giá kết quả thực nghiệm	23
4.4 Khả năng ứng dụng, hướng phát triển	23
4.5 Kết luận	23
Tài liệu tham khảo	24
Tài liệu tham khảo	29
A Quyết định XXX	33
B Phụ lục XXX	34
C Một số mã lệnh quan trọng	35

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 với các chương trình đào tạo lập trình Online, kỹ sư phần mềm... đang trở nên rất phổ biến. Trên thế giới cũng có nhiều chương trình đào tạo nổi tiếng như Massive Open Online Courses (MOOC) [37], edX [15], Coursera [10], Udacity [56] thu hút hàng ngàn sinh viên theo học. Một số chương trình học lập trình online như Pex4Fun [44] hay Code Hunt [22] là một nền tảng học lập trình online thông qua trò chơi.

Những lớp học như vậy thường có hàng trăm sinh viên tham gia, nhưng số lượng giảng viên tham giảng dạy chỉ vài người trong một lớp học. 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 từng sinh viên khi có yêu cầu. Chỉ riêng việc đọc, hiểu đánh giá kết quả mã lệnh do sinh viên viết đã tốn nhiều thời gian của giảng viên, 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 hơn giúp đỡ. Nhưng không phải lúc nào cũng có người bên cạnh giúp sinh viên viết mã chương trình, hoặc kinh nghiệm và kiến thức của những người này chưa chắc có thể đáp ứng được mong muốn của sinh viên.

Để giảm bớt những vất vả, khó khăn của 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 chất lượng học tập của sinh viên được tốt hơn. Sinh viên có thể biết được chương trình của mình viết đúng hay sai ngay lập tức.

Cách thức hoạt động của công cụ đánh giá kết quả này là đánh giá độ tương tự về hành vi của hai chương trình. Trong đó, công cụ sẽ tính toán tìm ra các mẫu dữ liệu đầu vào thử nghiệm chung cho cả hai chương trình, đưa từng mẫu

dữ liệu này 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 cho chương trình của sinh viên càng cao. Ngược lại, nếu tỷ lệ giống nhau 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
- Tìm hiểu và phân tích các kỹ thuật đo và áp dụng kỹ thuật sinh Test Case tự động trên các kỹ thuật đo
- 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 Sharp

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
- 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 tiếp cận đó là tự động so sánh những chỗ sai trong mã nguồn của sinh viên một cách hữu hạn với mã nguồn tham chiếu. Cách tiếp cận của phương pháp này đó 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. Về cơ bản, phương pháp này sử dụng khái niệm về độ tương tự ngữ nghĩa, cách tiếp cận của họ dựa trên cách hoạt động DFAs, trong khi đề tài này tôi tiếp cận dựa trên sự hoạt động của các chương trình.

Automated Feedback Generation for Introductory Programming Assignments [34], 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. Luận văn này tập trung vào việc làm thế 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 [58], 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 [3] [4], phụ thuộc giá trị đầu vào, đầu ra [18], tóm tắt biểu tượng [31]. 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 [19]. 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 các chương trình bài tập đơn giản [15], 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 [26], 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 [29], cú pháp cây trừu tượng [4], biểu đồ phụ thuộc chương trình [32], số liệu dựa trên số lượng cú pháp [11] [35]. 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, gồm:

- Kiểm thử phần mềm
- Sinh ngẫu nhiên dữ liệu thử
- Kỹ thuật Dynamic Symbolic Execution.

2.1 Kiểm thử phần mềm

Các khái niệm cơ bản trong kiểm thử phần mềm

Khái niệm về kiểm thử phần mềm

- Kiểm thử phần mềm là quá trình khảo sát một hệ thống hay thành phần dưới những điều kiện xác định, quan sát và ghi lại các kết quả, và đánh giá một khía cạnh nào đó của hệ thống hay thành phần đó [46].
- Kiểm thử phần mềm là hoạt động khảo sát thực tiễn sản phẩm hay dịch vụ phần mềm trong đúng môi trường chúng dự định sẽ được triển khai nhằm cung cấp cho người có lợi ích liên quan những thông tin về chất lượng của sản phẩm hay dịch vụ phần mềm ấy. Mục đích của kiểm thử phần mềm là tìm ra các lỗi hay khiếm khuyết phần mềm nhằm đảm bảo hiệu quả hoạt động tối ưu của phần mềm trong nhiều ngành khác nhau [59].
- Chúng ta có thể định nghĩa như sau: Kiểm thử phần mềm là một tiến trình hay một tập hợp các tiến trình được thiết kế để đảm bảo phần mềm hoạt động đúng theo cái mà chúng đã được thiết kế, và không thực hiện bất cứ thứ gì không mong muốn. Đây là một bước quan trọng trong quá trình phát triển hệ thống, giúp cho người xây dựng hệ thống và khách hàng thấy được hệ thống mới đã đáp ứng yêu cầu đặt ra hay chưa.

Mục đích của kiểm thử phần mềm

- Tìm ra nhiều lỗi bằng việc đưa ra các dòng thời gian.
- Chứng minh được sản phẩm hoàn thành có những chức năng hay ứng dụng giống với bản đặc tả yêu cầu.

- Tạo ra các test case có chất lượng cao, thực thi hiệu quả...
- Một số lỗi cơ bản trong kiểm thử phần mềm như: lỗi ngay từ khi phân tích yêu cầu, lỗi từ bản đặc tả hệ thống, lỗi trong code, lỗi hệ thống và nguồn tài nguyên hệ thống, lỗi trong vấn đề phần mềm, phần cứng...

Các phương pháp kiểm thử

- Kiểm thử tĩnh (Static testing): Là phương pháp thử phần mềm đòi hỏi phải duyệt lại các yêu cầu và các đặc tả bằng tay, thông qua việc sử dụng giấy, bút để kiểm tra logic, lần từng chi tiết mà không cần chạy chương trình. Kiểu kiểm thử này thường được sử dụng bởi chuyên viên thiết kế người mà viết mã lệnh một mình.

Kiểm thử tĩnh cũng có thể được tự động hóa. Nó sẽ thực hiện kiểm tra toàn bộ bao gồm các chương trình được phân tích bởi một trình thông dịch hoặc biên dịch mà xác nhận tính hợp lệ về cú pháp của chương trình.

- Kiểm thử động (Dynamic testing): Là phương pháp kiểm thử thông qua việc dùng máy chạy chương trình để điều tra trạng thái tác động của chương trình. Đó là kiểm thử dựa trên các ca kiểm thử xác định bằng sự thực hiện của đối tượng kiểm thử hay chạy các chương trình. Kiểm thử động là kiểm tra cách thức hoạt động của mã lệnh, tức là kiểm tra sự phản ứng vật lý từ hệ thống tới các biến luôn thay đổi theo thời gian. Trong kiểm thử động, phần mềm phải thực sự được biên dịch và chạy. Kiểm thử động thực sự bao gồm làm việc với phần mềm, nhập các giá trị đầu vào và kiểm tra xem liệu đầu ra có như mong muốn hay không.

Các phương pháp kiểm thử động gồm có kiểm thử mức đơn vị – Unit Tests, kiểm thử tích hợp – Integration Tests, kiểm thử hệ thống – System Tests, và kiểm thử chấp nhận sản phẩm – Acceptance Tests.

Các chiến lược kiểm thử

Trong chiến lược kiểm thử, chúng ta có ba chiến lược kiểm thử hay dùng nhất là: kiểm thử hộp đen, kiểm thử hộp trắng, và kiểm thử hộp xám.

- Kiểm thử hộp đen – Black box Một trong những chiến lược kiểm thử quan trọng là kiểm thử hộp đen, hướng dữ liệu, hay hướng vào ra. Kiểm thử hộp đen xem chương trình như là một “hộp đen” và hoàn toàn không quan tâm về cách cư xử và cấu trúc bên trong của chương trình. Thay vào đó là tập trung vào tìm các trường hợp mà chương trình không thực hiện theo các đặc tả của nó.

Kiểm thử hộp đen không có mối liên quan nào tới mã lệnh và kiểm thử viên chỉ rất đơn giản tam niệm là: một mã lệnh phải có lỗi. Sử dụng nguyên tắc “Hãy đòi hỏi và bạn sẽ được nhận”, những kiểm thử viên hộp đen tìm ra lỗi mà những lập trình viên không tìm ra. Nhưng, người ta nói kiểm thử hộp đen “giống như là đi trong bóng tối mà không có đèn vậy”, bởi vì kiểm thử viên không biết các phần mềm được kiểm tra thực sự được xây dựng như thế nào. Đó là lý do mà có nhiều trường hợp mà một kiểm thử viên hộp đen viết rất nhiều ca kiểm thử để kiểm tra một thứ gì đó mà đáng lẽ có thể chỉ cần kiểm tra bằng 1 ca kiểm thử duy

nhất, và hoặc một số phần của chương trình không được kiểm tra chút nào.

Do vậy, kiểm thử hộp đen có ưu điểm của “một sự đánh giá khách quan”, mặt khác nó lại có nhược điểm của “thăm dò mù”.

- Kiểm thử hộp trắng – White box. Là một chiến lược kiểm thử khác, trái ngược hoàn toàn với kiểm thử hộp đen, kiểm thử hộp trắng hay kiểm thử hướng logic cho phép bạn khảo sát cấu trúc bên trong của chương trình. Chiến lược này xuất phát từ dữ liệu kiểm thử bằng sự kiểm thử tính logic của chương trình. Kiểm thử viên sẽ truy cập vào cấu trúc dữ liệu và giải thuật bên trong chương trình (và cả mã lệnh thực hiện chúng).

Phương pháp kiểm thử hộp trắng cũng có thể được sử dụng để đánh giá sự hoàn thành của một bộ kiểm thử mà được tạo cùng với các phương pháp kiểm thử hộp đen. Điều này cho phép các nhóm phần mềm khảo sát các phần của 1 hệ thống ít khi được kiểm tra và đảm bảo rằng những điểm chức năng quan trọng nhất đã được kiểm tra.

- Kiểm thử hộp xám – Gray box testing Kiểm thử hộp xám đòi hỏi phải có sự truy cập tới cấu trúc dữ liệu và giải thuật bên trong cho những mục đích thiết kế các ca kiểm thử, nhưng là kiểm thử ở mức người sử dụng hay mức hộp đen. Việc thao tác tới dữ liệu đầu vào và định dạng dữ liệu đầu ra là không rõ ràng, giống như một chiếc “hộp xám”, bởi vì đầu vào và đầu ra rõ ràng là ở bên ngoài “hộp đen” mà chúng ta vẫn gọi về hệ thống được kiểm tra. Sự khác biệt này đặc biệt quan trọng khi quản lý kiểm thử tích hợp – Integration testing giữa 2 modul mã lệnh được viết bởi hai chuyên viên thiết kế khác nhau, trong đó chỉ giao diện là được đưa ra để kiểm thử. Kiểm thử hộp xám có thể cũng bao gồm cả thiết kế đối chiếu để quyết định, ví dụ, giá trị biên hay thông báo lỗi.

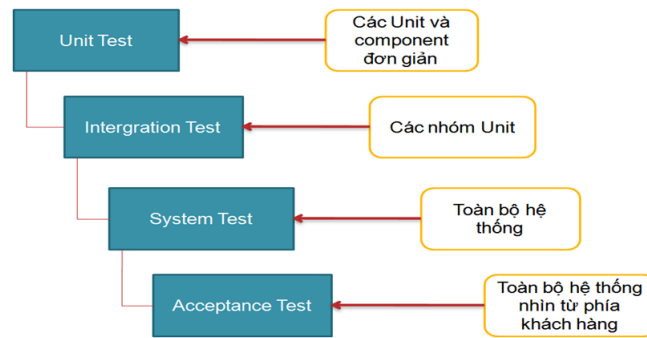
Các cấp độ kiểm thử trong kiểm thử phần mềm

Kiểm thử phần mềm gồm có các cấp độ:

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

Đảm bảo chất lượng phần mềm

Định nghĩa theo Daniel Galin [16]: 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 niềm tin rằng quá trình phát triển phần mềm phù hợp để thành lập các yêu cầu chức năng kỹ thuật cũng như các yêu cầu quản lý theo lịch trình và hoạt động trong giới hạn.



Hình 2.1: Sơ đồ các cấp độ kiểm thử

2.2 Sinh ngẫu nhiên dữ liệu thử

Trong kiểm thử phần mềm, các ca kiểm thử thường được tạo ra thủ công do đó sẽ gây ra tốn kém chi phí và mất nhiều thời gian để hoàn thành phần mềm. Hiện nay, đã có nhiều phương pháp hỗ trợ việc sinh tự động các ca kiểm thử một cách có hệ thống, một phương pháp đơn giản nhất đó là kiểm thử ngẫu nhiên (random testing) [62]. Giả sử, với một hàm nhận giá trị đầu vào có kiểu string thì chỉ cần sinh ngẫu nhiên một luồng các bits để thể hiện cho một chuỗi. Kiểm thử ngẫu nhiên có ưu điểm là dễ dàng sinh các đầu vào ngẫu nhiên và hệ thống kiểm thử ngẫu nhiên không yêu cầu bộ nhớ lúc thực thi. Nhưng hạn chế của nó là kiểm tra cùng 1 hành vi thực thi của chương trình nhiều lần với những đầu vào khác nhau và chỉ có thể kiểm tra được một số trường hợp thực thi của chương trình. Ngoài ra, kiểm thử ngẫu nhiên khó xác định khi nào việc kiểm thử nên dừng lại và không biết không gian trạng thái đã được thám hiểm hết hay chưa.

Một giải pháp giúp khắc phục những hạn chế của kiểm thử ngẫu nhiên đó là việc thực thi tượng trưng (Symbolic execution) [22]. Việc thực thi tượng trưng là việc xây dựng các ràng buộc dựa vào các giá trị tượng trưng và giải quyết các giá trị tượng trưng đó để sinh ra giá trị đầu vào của chương trình mà có thể thực thi chương trình theo các đường đi thực thi khác nhau. Ý tưởng chính của thực thi tượng trưng đó là thực thi một chương trình với những giá trị tượng trưng. Có hai cách tiếp cận với thực thi tượng trưng đó là cách tiếp cận dựa trên phân tích tĩnh và phân tích động chương trình

2.3 Kỹ thuật Dynamic symbolic execution

Kỹ thuật Dynamic Symbolic Execution (DSE) [61] là một kỹ thuật thực thi tượng trưng dựa trên phân tích chương trình động. DSE chính là sự kết hợp giữa thực thi cụ thể và thực thi tượng trưng. Trong thực thi tượng trưng động, chương trình được thực thi nhiều lần với những giá trị khác nhau của tham số đầu vào.

Bắt đầu bằng việc chọn những tham số tùy ý cho các tham số đầu vào và thực thi chương trình với những giá trị cụ thể đó. Với những giá trị cụ thể này thì chương trình sẽ được thực thi theo một đường đi xác định. Thực thi chương trình với các giá trị cụ thể của tham số đầu vào và thu gom các ràng

buộc trong quá trình thực thi theo đường đi mà sự thực thi cụ thể này đi theo, đồng thời suy ra những ràng buộc mới từ những ràng buộc đã thu gom được.

Tại các câu lệnh rẽ nhánh, biểu thức điều kiện rẽ nhánh sẽ được đánh giá phụ thuộc vào các giá trị cụ thể của tham số đầu vào. Nếu biểu thức rẽ nhánh nhận giá trị là True thì biểu thức của điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của PC và được ghi nhớ, đồng thời phủ định của điều kiện rẽ nhánh sẽ được sinh ra và sẽ được thêm vào một PC tương ứng với nhánh còn lại mà thực thi cụ thể đó không đi theo. Một bộ xử lý ràng buộc (Constraint Solver) sẽ được sử dụng để giải các ràng buộc mới sinh ra này để sinh ra các giá trị cụ thể của tham số đầu vào. Ngược lại nếu biểu thức rẽ nhánh nhận giá trị là False thì biểu thức điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của PC tương ứng với nhánh đi mà sự thực thi hiện thời đang đi theo và được ghi nhớ. Đồng thời điều kiện rẽ nhánh sẽ được sinh ra và thêm vào PC tương ứng với nhánh đi còn lại mà sự thực thi hiện thời không đi theo. Các giá trị mới được sinh ra của các tham số đầu vào sẽ tiếp tục được thực thi và quá trình này sẽ được lặp lại cho tới khi chương trình thực thi theo tất cả các đường đi.

Do các chương trình được thực thi với những giá trị cụ thể nên có thể thấy rằng, tất cả các đường đi phân tích được trong quá trình thực thi tượng trưng đồng đều là các đường đi khả thi.

Thuật toán Dynamic symbolic execution

- S: Tập hợp các lệnh của chương trình P
- s: Tập con của S ($s \subseteq S$)
- I: Tập hợp các đầu vào của P
- P(i): Thực thi chương trình đầu vào $i \in I$, sao cho s được thực thi
- J: Tập hợp các đầu vào của P được thực thi ($J = i|P(i)$)
- C(i): Ràng buộc thu gom từ việc thực thi P(i), hay còn gọi là điều kiện đường đi
- C'(i): Điều kiện đường đi suy ra từ C(i)

Bước 0: $J := \emptyset$ (tập rỗng)

Bước 1: Chọn đầu vào i không thuộc J (dừng lại nếu không có i nào được tìm ra)

Bước 2: Output i

Bước 3: Thực thi P(i); ghi nhớ đường đi C(i), suy ra C'(i)

Bước 4: Đặt $J := J \cup i$

Bước 5: Quay lại bước 1

Hiện nay, đã có những nghiên cứu về cách giải quyết các ràng buộc như Z3 [10], và nhiều công cụ sử dụng kỹ thuật DSE để giải quyết các ràng buộc và tạo ra các giá trị đầu vào có độ phủ cao như Pex [38] và SAGE [14]... hỗ trợ trên nhiều ngôn ngữ và nền tảng khác nhau.

Và một số công cụ khác:

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

Tổng kết chương

Chương này, trình bày khái quát và sơ lược những kiến thức 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, giải quyết các ràng buộc trong chương trình tìm ra các giá trị 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 và các phép đo đo được sử dụng trong luận văn bao gồm:

- Định nghĩa thực thi chương trình
- Định nghĩa tương đương hành vi
- Định nghĩa sự khác biệt về hành vi
- Định nghĩa độ tương tự hành vi
- Phép đo Random Sampling (RS)
- Phép đo Single Program Symbol Execution (SSE)
- Phép đo Paired Program Symbolic Execution
- Tiêu chí đánh giá các phép đo.

3.1 Hành vi của chương trình

Để định lượng hai chương trình tương tự nhau, chúng ta định nghĩa các khái niệm về hành vi chương trình và các định nghĩa liên quan đến sự tương tự của hai chương trình, và ví dụ minh họa cho các định nghĩa.

Thực thi chương trình

Định nghĩa 1 *Hành vi chương trình P là thực hiện hàm: $P \times I \rightarrow O$. Với giá trị đầu vào $i \in I$, giá trị đầu ra $o \in O$. Trong đó I là miền các giá trị đầu vào của chương trình P và O là tập hợp các giá trị đầu ra của chương trình P .*

Tương đương hành vi

Định nghĩa 2 *Hai chương trình P_1 và P_2 có cùng một miền các giá trị đầu vào I và tương đương về hành vi nếu $exec(P_1; I) = exec(P_2; I)$, với $\forall i \in I$ $exec(P_1; i) = exec(P_2; i)$.*

Ví dụ:

Mã lệnh 3.1: Tính y, sử dụng hàm 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: Tính y, sử dụng hàm 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;
}
```

Ví dụ trên cho chúng ta thấy 2 chương trình là tương đương nhau về hành vi. Hai chương trình có giá trị đầu vào là như nhau (cùng kiểu **Int**). Chương trình đầu tiên sử dụng hàm **switch...case**, chương trình tiếp theo sử dụng hàm **if...else** để kiểm tra giá trị đầu vào x, tuy cú pháp khác nhau nhưng cách xử lý và trả về kết quả **y** là như nhau.

Sự khác biệt về hành vi (Behavioral Difference)

Định nghĩa 3 Hai chương trình P_1 và P_2 có cùng một miền các giá trị đầu vào I và khác biệt về hành vi nếu $exec(P_1, I) \neq exec(P_2, I)$, $\exists i \in I$ $exec(P_1, i) \neq exec(P_2, i)$.

Tương tự hành vi (Behavioral Similarity)

Định nghĩa 4 Tương tự hành vi giữa hai chương trình P_1 và P_2 khi hai chương trình cùng miền giá trị đầu vào là tập hợp các giá trị $|I_s|/|I|$, trong đó $I_s \subseteq I$, nếu $exec(P_1, I_s) = exec(P_2, I_s)$, và $\forall j \in I \setminus I_s$, $exec(P_1; j) \neq exec(P_2; j)$.

3.2 Một số phép đo độ tương tự hành vi

Để đo sự tương đồng về hành vi giữa hai chương trình, chúng ta có thể đo bằng cách tính tỷ lệ đầu ra của hai chương trình trên cùng toàn bộ miền giá trị đầu

vào. Liệt kê tất cả các dữ liệu trong miền đầu vào và chạy từng đầu vào trên cả hai chương trình để so sánh các kết quả đầu ra. Nhưng việc này sẽ không khả thi với các chương trình có miền đầu vào lớn hoặc vô hạn.

Để đo độ tương tự hành vi được chính xác hơn, nếu chạy các dữ liệu đầu vào đại diện thay vì tất cả các dữ liệu đầu vào cho các chương trình. Bằng cách thống nhất lấy mẫu một phần dữ liệu đầu vào từ miền đầu vào, độ tương tự về hành vi sẽ được tính dựa trên tỷ lệ các mẫu đầu vào trên các mẫu đầu ra.

Dựa trên kỹ thuật Dynamic Symbolic Execution (**DSE**), chúng ta có thể tạo ra được dữ liệu đầu vào thử nghiệm có độ bao phủ cao, và sử dụng chúng trong các kỹ thuật đo độ tương tự.

Phép đo Random Sampling (RS)

Để tính toán sự giống nhau về hành vi giữa hai chương trình thông qua việc liệt kê tất cả các giá trị đầu vào có thể từ miền giá trị đầu vào là rất tốn thời gian và không khả thi. Thay vào đó chúng ta lấy mẫu ngẫu nhiên từ miền giá trị đầu vào để ước lượng tính toán, việc này sẽ không mất thời gian và khả thi hơn.

Phép đo **RS** là lấy mẫu thống nhất chung cho cả hai chương trình, dựa trên miền giá trị đầu vào của cả hai chương trình. Sau đó thực hiện chạy cả hai chương trình trên từng mẫu giá trị đầu vào thử nghiệm, tiến hành so sánh kết quả đầu ra của hai chương trình. Chúng ta có tỷ lệ số lượng các kết quả giống nhau của cả hai chương trình, trên tổng số lượng mẫu thống nhất chung của hai chương trình là kết quả của phép đo RS. Phép đo Random Sampling được định nghĩa như sau:

Định nghĩa 5 Hai chương trình P_1 và P_2 là hai chương trình có cùng miền giá trị đầu vào I và I_s là tập con các giá trị đầu vào của tập I , và I_a là tập con các giá trị đầu vào của tập I_s , trong đó $\forall i \in I_a$, sao cho $exec(P_1, i) = exec(P_2, i)$ và $\forall j \in I_s \setminus I_a$, $exec(P_1, j) \neq exec(P_2, j)$. Độ đo của kỹ thuật của RS sẽ là $M_{RS}(P_1, P_2) = |I_a|/|I_s|$.

Phép đo **RS** là một phương pháp đo tương đối đơn giản để tính độ tương tự hành vi. Nhưng phép đo **RS** cũng có hạn chế nhất định trong những trường hợp có những hành vi nhỏ giữa các chương trình và phép đo **RS** không thể phân biệt các hành vi hơi khác nhau này.

Mã lệnh 3.3: Chương trình P_1

```
public static int Puzzle(string x) {
    if (x == "Hello") return 0;
    return 1;
}
```

Mã lệnh 3.4: Chương trình P_2

```
public static int Puzzle(string x) {
    return 1;
}
```

Trong ví dụ trên, chương trình P_1 và P_2 có hành vi khác biệt nhỏ nhưng độ đo **RS** không thể phân biệt được sự khác biệt này và trả về độ đo bằng 1.

Phép đo Single Program Symbol Execution (SSE)

Phép đo SSE là một phép đo cải tiến của phép đo RS, bằng cách sử dụng kỹ thuật DSE để khám phá các đường dẫn của chương trình thực thi. Để tính toán độ đo SSE, chúng ta chọn một chương trình làm chương trình tham chiếu, áp dụng kỹ thuật DSE để tạo ra các đầu vào thử nghiệm của chương trình tham chiếu. Sau đó chạy cả hai chương trình dựa trên các giá trị đầu vào thử nghiệm của chương trình tham chiếu. Tỷ lệ số lượng các kết quả giống nhau của cả hai chương trình trên tổng số các giá trị đầu vào của chương trình tham chiếu là kết quả của phép đo SSE. Chúng ta có thể định nghĩa phép đo SSE cụ thể như sau.

Định nghĩa 6 Hai chương trình P_1 và P_2 là hai chương trình có cùng miền giá trị đầu vào I , và chương trình P_1 là chương trình tham chiếu. Trong đó, I_s là tập các giá trị đầu vào được tạo bởi DSE dựa trên chương trình P_1 , và I_a là tập con các giá trị đầu vào của tập I_s , sao cho $\forall i \in I_a, \text{exec}(P_1, i) = \text{exec}(P_2, i)$ và $\forall j \in I_s \setminus I_a, \text{exec}(P_1, j) \neq \text{exec}(P_2, j)$. Độ đo của kỹ thuật của SSE sẽ là $M_{SSE}(P_1, P_2) = |I_a|/|I_s|$.

Ngược lại với RS, SSE khám phá những đường dẫn khả thi khác nhau để tạo dữ liệu đầu vào của chương trình. Những SSE vẫn còn những hạn chế, đó là SSE không xem xét chương trình cần phân tích mà chỉ dựa vào các đầu vào thử nghiệm của chương trình tham chiếu. Các đầu vào thử nghiệm này không nắm bắt được các hành vi của chương trình phân tích, những chương trình phân tích sẽ có những hành vi khác với chương trình tham chiếu, SSE sẽ đánh giá không chính xác hành vi tương đương của hai chương trình. Ngoài ra, một số chương trình có thể có những vòng lặp vô hạn phụ thuộc vào giá trị đầu vào nên chúng ta không thể liệt kê được tất cả các đường dẫn của chương trình.

Kỹ Thuật Paired Program Symbolic Execution (PSE)

Để giải quyết hạn chế của SSE đó là không kiểm tra hành vi trong chương trình cần phân tích, kỹ thuật PSE xây dựng một chương trình ghép nối giữa chương trình cần phân tích với chương trình tham chiếu và tạo ra các đầu vào thử nghiệm bằng cách khám phá các đường dẫn trong chương trình được ghép nối. Chương trình ghép nối này chia sẻ cùng một miền giá trị đầu vào với cả hai chương trình, chạy cùng một giá trị đầu vào này trên cả hai chương trình và xác nhận kết quả đầu ra của hai chương trình là giống nhau. DSE tạo ra các đầu vào thử nghiệm dựa trên chương trình ghép nối, các đầu vào thử nghiệm này bao gồm các đầu vào đúng và không đúng. Các đầu vào đầu vào thử nghiệm đúng là những giá trị khi chạy 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 chạy trên cả hai chương trình sẽ cho kết quả khác nhau. Do đó, độ đo của kỹ thuật PSE được tính bằng tỷ lệ các giá trị đầu vào thử nghiệm đúng trên tổng số giá trị đầu vào thử nghiệm.

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

Định nghĩa 7 Hai chương trình P_1 và P_2 là hai chương trình có cùng miền giá trị đầu vào I . Chúng ta có chương trình P_3 là chương trình kết hợp của P_1 và P_2 có dạng ($exec(P_1, I) = exec(P_2, I)$), thỏa điều kiện đầu vào và khẳng định điều kiện là đúng, $exec(P_3, i) = T$ với giá trị đầu vào i trên P_3 là thỏa điều kiện. Trong đó, I_s là tập các giá trị đầu vào được tạo bởi DSE trên P_3 , và I_a là tập con các giá trị đầu vào của tập I_s , sao cho $\forall i \in I_a, exec(P_3, i) = T$ và $\nexists j \in I_s \setminus I_a, exec(P_3, j) = T$. Độ đo của PSE sẽ là $M_{PSE}(P_1, P_2) = |I_a|/|I_s|$.

Kỹ thuật PSE xây dựng một chương trình ghép nối giữa hai chương trình đã cải thiện được hạn chế của kỹ thuật SSE đó là chỉ khám phá hành vi của chương trình tham chiếu. Tuy nhiên, kỹ thuật PSE cũng có hạn chế trong việc xử lý các vòng lặp lớn hoặc vô hạn trong số các chương trình được ghép nối. Để giảm bớt hạn chế này, chúng ta có thể giới hạn miền đầu vào hoặc đếm số vòng lặp. Ngoài ra, kỹ thuật PSE khám phá đường dẫn của chương trình ghép nối nên sẽ tốn thời gian và tài nguyên hơn so với kỹ thuật SSE.

Tổng kết chương

Nội dung chương 3 này trình bày sơ lược những định nghĩa về thực thi chương trình, hành vi của chương trình, độ tương tự về hành vi của chương trình. Mô tả, định nghĩa 3 kỹ thuật đo RS, SSE, PSE và nêu lên những ưu và nhược điểm cũng như hướng khắc phục của 3 kỹ thuật. Ba kỹ thuật đo này là nội dung chính trong luận văn của tôi.

Chương 4

Thực nghiệm, đánh giá, kết luận

Chương này trình bày ...

4.1 Dữ liệu thực nghiệm

Phần này trình bày về kho dữ liệu dùng để thực nghiệm, Code Hunt [REF].

4.2 Công cụ dùng trong thực nghiệm

Phần này trình bày những công cụ được sử dụng để triển khai thực nghiệm như: công cụ sinh dữ liệu kiểm thử, môi trường lập trình, ...

4.3 Đánh giá kết quả thực nghiệm

Phần này trình bày những kết quả đo được trên bộ dữ liệu thực nghiệm đã nêu trong Phần 4.1.

4.4 Khả năng ứng dụng, hướng phát triển

Phần này trình bày một số ứng dụng có thể của việc đo độ tương tự về hành vi của chương trình cùng hướng phát triển trong tương lai.

Đánh giá tiến bộ trong lập trình

Xếp hạng tự động

Gợi ý giải pháp lập trình

Hướng phát triển

4.5 Kết luận

Tổng kết chương

Tổng kết chương viết ở đây.

«««< HEAD

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] 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.
- [4] 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.
- [5] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *Software Maintenance, 1992. Proceedings., Conference on*, pages 41–50. IEEE, 1992.
- [6] 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.
- [7] 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.
- [8] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
- [9] 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.
- [10] Coursera. <https://www.coursera.org/>.

- [11] 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.
- [12] John A Darringer and James C King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.
- [13] 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.
- [14] 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.
- [15] EdX. <https://www.edx.org/>.
- [16] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.
- [17] 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.
- [18] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284–1288, 1991.
- [19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [21] Jan B Hext and JW Winings. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275, 1969.
- [22] Code Hunt. <https://www.microsoft.com/en-us/research/project/code-hunt/>.
- [23] 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.
- [24] 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.
- [25] 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.

- [26] David Jackson and Michelle Usher. Grading student programs using as-syst. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM, 1997.
- [27] 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.
- [28] Edward L Jones. Grading student programs-a software testing approach. *Journal of Computing Sciences in Colleges*, 16(2):185–192, 2001.
- [29] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [30] 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.
- [31] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [32] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.
- [33] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Re-bêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*, volume 12, pages 712–717. Springer, 2012.
- [34] 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.
- [35] 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.
- [36] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.
- [37] Coursera MOOC on Software Engineering for SaaS. <https://www.coursera.org/course/saas>.
- [38] 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.

- [39] Laura Pappano. The year of the mooc. *The New York Times*, 2(12):2012, 2012.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] 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.
- [44] Pex4Fun. <https://pexforfun.com/>.
- [45] Roger S Pressman. *Software engineering: a practitioner’s approach*. Palgrave Macmillan, 2005.
- [46] Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990(121990):3, 1990.
- [47] 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.
- [48] 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.
- [49] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.
- [50] 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.
- [51] 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.

- [52] 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.
- [53] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.
- [54] 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.
- [55] 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.
- [56] Udacity. <http://www.udacity.com/>.
- [57] 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.
- [58] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. volume 49, pages 99–107. Elsevier, 2007.
- [59] Wikipedia. <https://en.wikipedia.org>.
- [60] 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.
- [61] 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.
- [62] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

=====

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] 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.
- [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] 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.
- [6] 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.
- [7] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
- [8] 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.
- [9] John A Darringer and James C King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.
- [10] 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.

- [11] 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.
- [12] 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.
- [13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [14] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.
- [15] Jan B Hext and JW Winings. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275, 1969.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] Edward L Jones. Grading student programs-a software testing approach. *Journal of Computing Sciences in Colleges*, 16(2):185–192, 2001.
- [21] 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.
- [22] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [23] 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.
- [24] 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.

- [25] Christophe Meudec. Atgen: automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, 2001.
- [26] 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.
- [27] Laura Pappano. The year of the mooc. *The New York Times*, 2(12):2012, 2012.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.
- [35] 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.
- [36] 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.

- [37] 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.
- [38] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for. net. *Tests and Proofs*, pages 134–153, 2008.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.

»»»> 0e8bde2646f6e6475cd4bf4c8ddb35cf5d507cec

Trích dẫn
bài báo

Phụ lục A

Quyết định XXX

scan quyết
định sang
pdf và in-
clude


Phụ lục B

Phụ lục XXX

Nội dung phụ lục viết ở đây.

Phụ lục C

Một số mã lệnh quan trọng



Đưa một số
đoạn code
quan trọng