ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA

NGÔ PÔ NA

XÂY DỰNG CÔNG CỤ SINH DỮ LIỆU THỬ TỰ ĐỘNG CHO CHƯƠNG TRÌNH JAVA

Chuyên ngành : Khoa học máy tính
Mã số : 60.48.01.01

TÓM TẮT LUẬN VĂN THẠC SĨ
KHOA HỌC MÁY TÍNH

Đà Nẵng – Năm 2017

Công trình được hoàn thành tại
TRƯỜNG ĐẠI HỌC BÁCH KHOA ĐÀ NẴNG

Người hướng dẫn khoa học : PGS.TS NGUYỄN THANH BÌNHI

Phản biện 1 : PGS.TS. Lê Mạnh Thạnh

Phản biện 2 : TS. Lê Xuân Việt

Luận văn được bảo vệ trước Hội đồng chấm Luận văn tốt nghiệp thạc sĩ ngành Khoa học máy tính họp tại Trường Đại học Bách khoa Đà Nẵng vào ngày 8 tháng 1 năm 2017

Có thể tìm hiểu luận văn tại :

- Trung tâm Thông tin - Học liệu, Đại học Đà Nẵng
- Trung tâm học liệu truyền thông, trường Đại học Bách Khoa, Đại học Đà Nẵng

MỞ ĐẦU

1. Lý do chọn đề tài

Phần mềm hiện nay được sử dụng rộng rãi trong đời sống, công việc, nhiều lĩnh vực khoa học, kinh tế và xã hội. Vì vậy, việc đảm bảo rằng phần mềm đáp ứng mong muốn của người sử dụng là rất quan trọng. Kiểm thử phần mềm là một trong những hoạt động cơ bản nhằm đảm bảo chất lượng phần mềm.

Việc phát triển phần mềm ngày càng được chuyên nghiệp hóa. Các phần mềm được phát triển ngày càng có quy mô lớn. Yêu cầu đảm bảo chất lượng phần mềm là một trong những mục tiêu quan trong nhất, đặc biệt trong một số lĩnh vực như y khoa, ngân hàng, hàng không... Việc kiểm thử, kiểm chứng phần mềm một cách thủ công chỉ đảm bảo được phần nào chất lượng của phần mềm. Vì vậy rất nhiều các tổ chức, công ty đã nghiên cứu và phát triển các lý thuyết cũng như công cụ để kiểm chứng, kiểm thử phần mềm một cách tự động.

Một số lợi ích có thể kể đến của kiểm thử tự động như: cải thiện hiệu quả công việc, cải thiện tính chính xác, cải thiện chất lượng kiểm thử và chất lượng của phần mềm. Tại các doanh nghiệp tư nhân hiện tại, công việc kiểm thử phần mềm đơn vị (unit test) thường được các lập trình viên thực hiện ngay trong quá trình viết mã nguồn của chương trình. Vì vậy dẫn đến một số vấn đề như sau:

- Không đảm bảo được tính khách quan.
- Các lập trình viên thường khó sử dụng các kỹ thuật kiểm thử hộp trắng vì không đủ chi phí thời gian.

Hiện tại trong các dự án mà tôi đang tham gia, ngôn ngữ lập trình chủ yếu được sử dụng là Java. Ngoài ra,trong các ngôn ngữ lập trình

hiện đại ngày nay, Java là ngôn ngữ lập trình phổ biến trong suốt 13 năm qua [1].

Vì những lý do trên, tôi đề xuất chọn đề tài luận văn cao học: "Xây dựng công cụ sinh dữ liệu thử tự động cho chương trình Java".

2. Mục đích và ý nghĩa đề tài

a. Mục đích

- Xây dựng công cụ sinh dữ liệu kiểm thử tự động cho chương trình nguồn Java. Nhằm mục đích thực hiện việc kiểm thử hộp trắng cho kiểm thử đơn vị một cách tự động và khoa học. Hướng đến mục tiêu giảm chi phí về thời gian và tài chính khi thực hiện công việc kiểm thử cho các lập trình viên/kiểm thử viên.
- Tìm hiểu về các thách thức gặp phải trong quá trình sinh dữ liệu thử tự động, từ đó đề xuất và cài đặt giải pháp để giải quyết các thách thức này.

b. Ý nghĩa khoa học

- Nghiên cứu về các kỹ thuật kiểm thử hộp trắng.
- Nghiên cứu về lý thuyết về tính thoả được.
- Xây dựng công cụ tự động sinh dữ liệu cho chương trình Java đảm bảo tiêu chí bao phủ lộ trình và tối ưu thời gian thực thi.

c. Ý nghĩa thực tiễn

- Giảm thời gian và chi phí cho việc kiểm thử hộp trắng của các lập trình viên khi thực hiện kiểm thử đơn vị.
- Dữ liệu thử tự động mang tính khách quan hơn, không bị phụ thuộc vào góc nhìn và kinh nghiệm của người lập trình.
- Kiểm tra được các lỗi tiềm ẩn trong mã nguồn.

3. Mục tiêu và nhiệm vụ

a. Mục tiêu

Xây dựng công cụ sinh dữ liệu tự động cho chương trình Java đảm bảo tiêu chí bao phủ lộ trình.. Các mục tiêu cụ thể như sau:

- Nghiên cứu về các phương pháp kiểm thử hộp trắng.
- Nghiên cứu về phương pháp giải các ràng buộc (SMT [2]).
- Nghiên cứu về các giải pháp sinh dữ liệu thử tự động cho chương trình Java và những thách thức hiện nay.
- Xây dựng công cụ sinh dữ liệu thử cho chương trình Java.

b. Nhiệm vụ

Để đạt được những mục tiêu trên, nhiệm vụ đặt ra của đề tài là:

- Nghiên cứu lý thuyết kiểm thử hộp trắng, tập trung vào các lý thuyết xây dựng tập dữ liệu kiểm thử.
- Nghiên cứu về lý thuyết về tính thoả được.
- Nghiên cứu và áp dụng kỹ thuật tối ưu hóa để tối ưu thời gian thực thi.

4. Đối tượng và phạm vi nghiên cứu

Luận văn tập trung vào nghiên cứu các đối tượng và phạm vi sau:

- Cấu trúc chương trình Java.
- Lý thuyết về tính thoả được của các bộ SMT và cách áp dụng để giải các ràng buộc.
- Phân tích, thiết kế và cài đặt ứng dụng sinh dữ liệu thử cho chương trình Java.

5. Phương pháp nghiên cứu

a. Phương pháp lý thuyết

– Tiến hành thu thập và nghiên cứu các tài liệu có liên quan đến đề tài.

- Nghiên cứu lý thuyết về thực thi ký hiệu và lý thuyết về tính thỏa được.
- Nghiên cứu về kiểm thử đơn vị và cách thực hiện.
- Nghiên cứu các phương pháp tối ưu hóa các đường thực thi.

b. Phương pháp thực nghiệm

- Nghiên cứu về phương pháp thực thi ký hiệu và các công cụ hỗ trợ thực thi ký hiệu hiện nay.
- Nghiên cứu đề xuất giải pháp tối ưu các đường thực thi trong quá trình thực hiện thực thi ký hiệu.

5. Kết luận

a. Kết quả của đề tài

- Thứ nhất, tìm hiểu về cách xây dựng và hoạt động của và phương pháp sinh dữ liệu kiểm thử tự động cho ngôn ngữ Java cũng như nghiên cứu về các thách thức hiện nay.
- Thứ hai, đề xuất và cài đặt giải pháp tối ưu trong việc sinh dữ liệu thử tự động nhằm tối ưu hóa thời gian nhưng vẫn đảm bảo tính đúng đắn, toàn vẹn và khả năng bao phủ của bộ dữ liệu thử tự động.

b. Hướng phát triển của đề tài

Trong quá trình nghiên cứu, bộ SMT Choco không cho kết quả tốt như các bộ SMT khác như Z3 hay CVC3. Vì vậy, trong thời gian tới chúng tôi sẽ tiếp tục nghiên cứu và triển khai tích hợp các bộ SMT mới và có khả năng hỗ trợ nâng cấp tốt hơn vào trong chương trình sinh dữ liệu kiểm thử

CHƯƠNG 1
CƠ SỞ LÝ THUYẾT

1.1. Tổng quan về kiểm thữ phần mềm

1.1.1. Các bước kiểm thử

1.1.2. Các mức kiểm thử

1.1.3. Các kỹ thuật kiểm thử phần mềm

1.2. Lý thuyết về tính thỏa được

Tính thỏa mãn là một trong những vấn đề quan trọng nhất của ngành khoa học máy tính. Các vấn đề cần tính thỏa mãn được ứng dụng trong cả phát triển phần cứng cũng như phần mềm, đặc biệt là kiểm định phần cứng, kiểm thử, lập lịch, đồ thị.

Trong các lĩnh vực nói trên, nhiều các ứng dụng được xây dựng dựa trên việc tạo ra các công thức tiền tố và việc chứng minh tính hợp lệ của chúng. Cho dù hai thập niên gần đây, việc chứng minh tính hợp lệ của các định lý, biểu thức tiền tố có những tiến bộ đáng kể, tuy nhiên, không phải công thức nào cũng có thể chứng minh một cách tự động được. Lý do của vấn đề này là bởi lẽ nhiều công thức không quan tâm đến tính khả thi trong trường hợp tổng quát mà chỉ được quan tâm trong một lý thuyết nền tảng. Việc nghiên cứu tính khả thi của các công thức trong một lý thuyết nền tảng được gọi là các lý thuyết module về tính thỏa được (Satisfiability Modulo Theories hay SMT) và các công cụ để chứng minh một cách tự động các tính khả thi của những vẫn đề SMT được gọi là bộ giải SMT (SMT solver).

Trên thực tế, việc xây dựng và sử dụng các bộ giải SMT được phát triển khá sớm, từ đầu những năm 1980 [9]. Tại thời điểm đó, một số bộ giải được xây dựng bởi Greg Nelson và Derek Oppent tại trường đại học Stanford , Robert Shostak tại SRI, và Robert

Boyer và J Moore tại trường đại học ở Texas. Đến cuối những năm 1990, việc nghiên cứu SMT hiện đại dựa trên lợi thế của công nghệ SAT đã xây dựng nhiều bộ giải SMT tiến bộ hơn .

Như đã đề cập ở trên, trong khuôn khổ đồ án, việc đánh giá về tính đúng đắn, các nghiên cứu về thuật giải của từng bộ giải sẽ không được đề cập đến. Vấn đề được đặt ra ở đây là kết quả của bộ giải nào sẽ được đưa ra sớm nhất. Hiện nay, có rất nhiều các bộ giải như Absolver, Boolector, CVC3, OpenSMT, Yices, Z3… Do yêu cầu của hệ thống là phải đưa ra được giá trị thỏa mãn (nếu bài toán SMT đó có thỏa mãn) nên bộ giải hệ thống sử dụng phải hỗ trợ chức năng này. Ngoài ra hệ thống sử dụng đầu vào theo chuẩn của SMT-Lib và ngắt thời gian giải một bài toán (trong trường hợp bài toán cần thời gian giải quá lớn), do đó, bộ giải cần phải có hỗ trợ những chức năng này khi hoạt động. Từ những yêu cầu đó, hai bộ giải là Yices của SRI International và Choco thuần Java đã được lựa chọn trong JPF. Hai bộ giải này tuy có cấu trúc khác nhau nhưng cùng được dựa trên thuật giải DPLL (Davis-Putnam-Logemann-Loveland).

1.3. Thực thi ký hiệu

Trong hoạt động kiểm thử phần mềm, các ca kiểm thử thường được tạo ra một cách thủ công, gây tốn kém về chi phí cũng như thời gian để hoàn thành công đoạn này. Thực thi ký hiệu (Symbolic execution) được biết đến là một kỹ thuật nổi tiếng với khả năng tự động sinh những bộ ca kiểm thử có độ bao phủ cao với các tiêu chí kiểm thử nhằm phát hiện những lỗi sâu trong các hệ thống phần mềm phức tạp. Trong mục này sẽ trình bày các vấn đề tổng quan và một số kết quả của các nghiên cứu gần đây về kỹ thuật thực thi ký hiệu, đưa ra những thách thức cần giải quyết trong lĩnh vực

này như: sự bùng nổ đường thực thi của chương trình, khả năng giải các ràng buộc, mô hình hóa bộ nhớ, các vấn đề về tương tranh, đồng thời cũng tập trung vào phân tích và đề xuất giải pháp để sinh ra dữ liệu kiểm thử cho chương trình Java một cách tự động.

Hiện nay có rất nhiều công cụ nền tảng phục vụ cho hoạt động kiểm thử phần mềm như JUnit cho ngôn ngữ Java, NUnit, VSUnit cho .NET để thực thi các ca kiểm thử mức đơn vị. Tuy nhiên, các công cụ kiểm thử này không hỗ trợ việc sinh tự động các ca kiểm thử đơn vị. Viết các ca kiểm thử là một công việc nặng nhọc và tốn nhiều công sức. Có nhiều phương pháp khác nhau hỗ trợ việc sinh tự động các ca kiểm thử giúp giảm chi phí và thời gian thực hiện đã được nghiên cứu và đưa ra như: Dựa trên kiểm chứng mô hình (Model Checking), kểm thử ngẫu nhiên (Random Testing). Nhưng hạn chế của nó là kiểm tra cùng một hành vi thực thi của chương trình nhiều lần với những đầu vào khác nhau và chỉ có thể kiểm tra được một số trường hợp thực thi của chương trình. Thêm vào đó, kiểm thử ngẫu nhiên khó xác định được khi nào việc kiểm thử nên được dừng lại và nó không biết tại điểm nào không gian trạng thái đã được thám hiểm hết. Để xác định khi nào việc kiểm thử dừng lại thì hệ thống kiểm thử ngẫu nhiên được kết hợp với các tiêu chuẩn an toàn. Để khắc phục những hạn chế của kiểm thử ngẫu nhiên, phương pháp thực thi tương trưng xây dựng các ràng buộc trên các giá trị ký hiệu và giải các ràng buộc đó để sinh ra các giá trị đầu vào cho chương trình mà có thể bao phủ tất các dòng lệnh cũng như các nhánh thực thi của chương trình

Ý tưởng của thực thi ký hiệu đã được đề xuất bởi, nhưng việc hiện thực ý tưởng mới chỉ được thực hiện trong những năm gần đây qua tiến bộ đáng kể trong lý thuyết giải các ràng buộc và các tiếp

cận mở rộng thực thi ký hiệu động , một kỹ thuật kết hợp giữa các giá trị cụ thể và giá trị ký hiệu cho các giá trị đầu vào.

1.4. Các kỹ thuật thực thi ký hiệu

a. Concolic testing

Thực thi ký hiệu động chính là sự kết hợp giữa thực thi cụ thể và thực thi ký hiệu. Trong thực thi ký hiệu động, chương trình được thực thi nhiều lần với những giá trị khác nhau của tham số đầu vào.

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

Tại các câu lệnh rẽ nhánh, biểu thức điều kiện rẽ nhánh sẽ được đánh giá theo các giá trị cụ thể của các tham số đầu vào. Nếu biểu thức điều kiện rẽ nhánh nhận giá trị là True thì biểu thức của điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của ràng buộc lộ trình và được ghi nhớ, đồng thời phủ định của điều kiện rẽ nhánh sẽ được sinh ra và được thêm vào một ràng buộc lộ trình tương ứng với nhánh còn lại mà sự thực thi cụ thể đó không đi theo. Một bộ xử lý ràng buộc sẽ được sử dụng để giải quyết các ràng buộc mới sinh ra này để sinh ra các giá trị cụ thể của tham số đầu vào. Trong trường hợp ngược lại biểu thức phủ định của điều kiện rẽ nhánh sẽ được thu gom vào ràng buộc của ràng buộc lộ trình tương ứng với nhánh mà sự thực thi hiện thời đang đi theo và được ghi nhớ. Đồng thời điều kiện rẽ nhánh sẽ được sinh ra và thêm vào PC tương ứng với nhánh còn lại mà sự thực thi hiện thời không đi theo. Các giá trị mới được

sinh ra của các tham số đầu vào sẽ tiếp tục được thực thi và quá trình này sẽ được lặp lại cho tới khi chương trình được thực thi theo tất cả các đường đi. Do các chương trình được thực thi với những giá trị cụ thể nên có thể thấy rằng tất cả các đường đi phân tích được trong quá trình thực thi ký hiệu động đều là các đường đi khả thi

b. Excution generated testing

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

1.5. Các thách thức và giải pháp

a. Bùng nổ đường đi

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

Trước hết, lưu ý rằng thực thi ký hiệu đã ngầm lọc ra tất cả các đường thực thi không phụ thuộc vào các biến ký hiệu đầu vào và những đường dẫn không khả thi (Infeasiblle path) trong quá trình giải các ràng buộc. Mặc dù như vậy, sự bùng nổ đường dẫn vẫn là một trong những thách thức lớn nhất đối với thực thi ký hiệu. Hiện nay có 2 cách tiếp cận chính để giải quyết vấn đề này là ưu tiên tìm kiếm kinh nghiệm (Heuristic search) và sử dụng kỹ thuật phân tích tính đúng đắn của chương trình.

b. Giải các ràng buộc

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

c. Mô hình hóa bộ nhớ

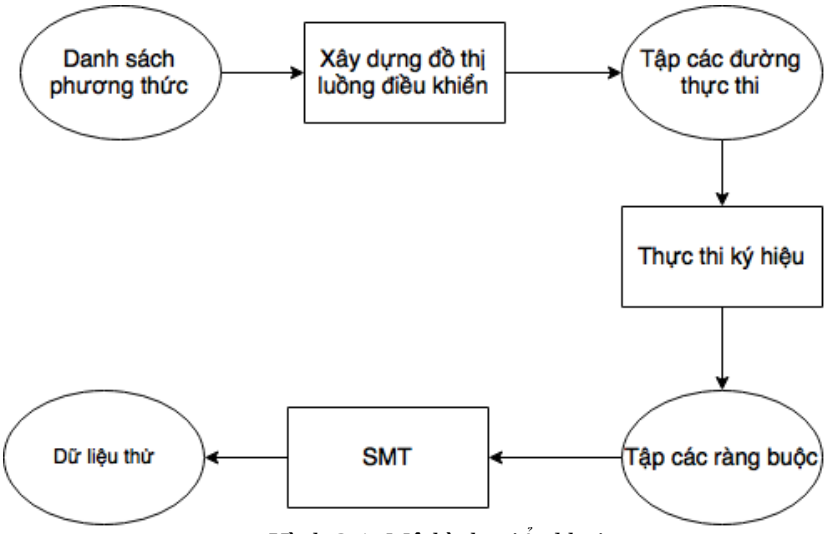
Độ chính xác của các câu lệnh của chương trình khi chuyển sang các ràng buộc ký hiệu có ảnh hưởng đáng kể đến độ bao phủ khi thực hiện thực thi ký hiệu. Ví dụ khi sử dụng mô hình hóa bộ nhớ mà xấp xỉ để thiết lập độ rộng cho tham biến kiểu Interger có thể có hiệu quả hơn trong thực tế nhưng mặt khác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

CHƯƠNG 2
GIẢI PHÁP SINH DỮ LIỆU THỬ CHO CHƯƠNG TRÌNH JAVA

2.1. Giải pháp sinh dữ liệu thử cho chương trình JAVA

a. Tổng quan về giải pháp

Mô hình tổng thể của giải pháp sinh dữ liệu thử cho chương trình Java được miêu tả như trong hình 2.1 dưới đây:



Hình 2.1. Mô hình triển khai

Tham số đầu vào của phương pháp là tệp nguồn Java cần phân tích, số vòng lặp tối đa (của các lệnh lặp for, while ...), và các tiêu chuẩn phủ. Mã nguồn Java được phân rã thành các phương thức và mức độ ưu tiên xử lý của chúng được tính toán, những phương thức ít phụ thuộc vào các phương thức khác hơn sẽ có độ ưu tiên xử lý cao hơn, như vậy những phương thức không có lời gọi hàm sẽ được xử lý đầu tiên. Cuối cùng chúng ta có một danh sách các

phương thức được xếp theo mức ưu tiên giảm dần, và chúng lần lượt được xử lý theo thứ tự đó.

Với mỗi phương thức, đầu tiên đồ thị luồng điều khiển được xây dựng, duyệt đồ thị luồng điều khiển sẽ cho ra các lộ trình thực thi của phương thức, tiếp đến có được tập ràng buộc tương ứng nhờ áp dụng phương pháp thực thi ký hiệu. Quá trình này được lặp lại cho đến khi tất cả các phương thức được phân tích hết.

Các lộ trình thực thi đã tìm được ở bước trên sẽ được đưa lần lượt vào SMT-Solver để kiểm tra tính thoả được và sinh ra dữ liệu kiểm thử tự động đảm bảo bao phủ được lộ trình đã đưa vào SMT-Solver

b. Phương pháp thực hiện

Bước 1: *Phân tích danh sách các phương thức*

Ta dựa vào 2 định nghĩa sau:

Định nghĩa 2.1: Đồ thị luồng với tập nút N, tập cạnh E, là một đồ thị có hướng liên thông, luôn luôn có duy nhất một điểm đầu vào, và ít nhất một điểm kết thúc [7].

Định nghĩa 2.2: Trong chương trình, đồ thị luồng biểu diễn cho những luồng điều khiển khả thi của nó, trong đó mỗi nút tương ứng cho các điểm chương trình (tập lệnh) và các cạnh thể hiện cho luồng điều khiển giữa các điểm đó [7].

Dựa vào cấu trúc một phương thức ta phân tách thành các khối độc lập. Sau đó lại gọi đệ quy để phân tích thành các khối cấu trúc con của nó. Trong bước này, có thể sử dụng công cụ JDT cho phép duyệt qua toàn bộ phương thức và nhận dạng các thành phần cấu tạo nên một câu lệnh như If, For, While...; từ đó dễ dàng xây dựng các đặc tả của một nút và giúp tập trung vào thuật toán xây dựng đồ thị.

Bước 2: *Sinh ra các lộ trình thực thi*

Tập hợp các lộ trình thực thi của một hàm có thể lấy được bằng cách duyệt đồ thị luồng điều khiển tương ứng với nó. Với những phương thức có lời gọi hàm con, mỗi nút của đồ thị hoặc là các câu lệnh (hoặc tập các câu lệnh) xác định, hoặc sẽ là lời gọi đến các hàm con đó.

Cần lưu ý rằng thay vì chèn ràng buộc lộ trình của hàm con, có thể thay thế nó bằng lộ trình thực thi tương ứng. Tuy nhiên kết quả của một hàm chỉ phụ thuộc vào đối số của nó, nói cách khác các biến khác được tạo ra và sử dụng bên trong hàm con được gọi không ảnh hưởng đến lộ trình thực thi hiện tại. Do vậy việc sử dụng lộ trình ràng buộc sẽ là tối ưu khi mà các biến số bên trong nó đã được xử lý bằng thực thi ký hiệu.

Bước 3: *Sinh ra tập các ràng buộc*

Định nghĩa 2.3: Ràng buộc lộ trình[7] là một tập hợp các biểu thức luận lý:

$$PC = c_1 \wedge c_2 \wedge \ldots \wedge c_n$$

trong đó n là số nút điều kiện của đường thực thi, c biểu diễn ràng buộc tương ứng với mỗi điều kiện đó .

Tập lộ trình thực thi của các phương thức được xử lý để cho ra tập lộ trình ràng buộc bằng cách sử dụng phương pháp Thực thi ký hiệu. Mỗi bước thực hiện trên lộ trình thực thi được mô tả bởi trạng thái bởi các biến đầu vào (input) và các biến bên trong phương thức

Bước 4 : *Sinh ra các dữ liệu kiểm thử từ bộ SMT solver*

Để làm việc với SMT ta thực hiện công việc chuẩn hóa các đầu vào theo định dạng SMT – LIB. Mỗi nút của đường ràng buộc sẽ được chuyển từ dạng trung tố sang dạng tiền tố, từ khoá assert được sử dụng để thêm các ràng buộc vào SMT

2.2. Vấn đề tối ưu các ràng buộc

Một trong những nhược điểm chính của sinh dữ liệu thử tự động là cần phải tối thiểu các ràng buộc nhằm đảm bảo tốc độ thực thi và hao tốn về tài nguyên như đã trình bày trong mục 1.5, vì vậy thuật toán tối ưu hóa các ràng buộc đã được cài đặt tích hợp thêm vào trong công cụ có sẵn và đảm bảo các điểm sau:

- Hạn chế số lượng các ràng buộc được đưa vào SMT để giảm tiêu tốn về tài nguyên.
- Đảm bảo tính đúng đắn, không làm sai lệch kết quả đầu ra của chương trình sau khi áp dụng thuật toán.

Vì vậy, bài toán đặt ra là phải cài đặt thuật toán tối ưu hóa vào trong chương trình mã nguồn mở có sẵn để tăng tốc độ thực thi nhưng vẫn đảm bảo tính đúng đắn của chương trình.

2.3. Giải quyết các ràng buộc

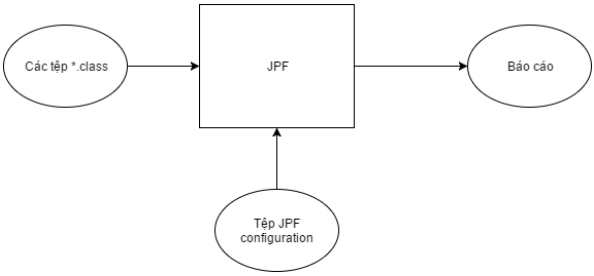
Trong quá trình thực hiện việc tối ưu hoá các ràng buộc, chúng tôi sử dụng công cụ mã nguồn mở JPF trong việc hỗ trợ thực thi ký hiệu và chỉ tập trung vào việc cài đặt và thử nghiệm thuật toán để tối ưu hoá các ràng buộc. Mô hình thực hiện được biểu diễn: Với sự hỗ trợ từ công cụ mã nguồn mở JPF, có thể triển khai quá trình thực hiện như trong thuật toán được trình bày cụ thể trong mục 2.5

2.4. Ứng dụng JAVA PATH FINDER

a. Giới thiệu về JPF

JPF là một máy ảo cho Java bytecode, có nghĩa là nó là một chương trình nhận đầu vào là một chương trình Java để thực thi. Nó được sử dụng để tìm lỗi trong các chương trình này, vì vậy ta cũng cần phải cung cấp cho nó các thuộc tính để kiểm tra như đầu vào [10].

JPF sẽ cho ra một báo cáo đã được tạo ra bởi JPF để phân tích.Cơ chế của JPF được thực hiện như hình minh họa sau:



Hình 2.3. Cấu trúc của JPF

b. Giới thiệu về bộ mở rộng JPF-SYMBC

JPF-SYMBC là một gói mở rộng của JPF nhằm mục tiêu hỗ trợ người lập trình trong việc thực hiện thực thi ký hiệu.

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

c. Tối ưu hoá các ràng buộc lộ trình để phục vụ cho quá trình thực thi ký hiệu

Việc giải các ràng buộc(Constraint solving) là một phần không thể thiếu của việc thực thi ký hiệu, việc giải quyết các constraint là phần tốt nhiều tài nguyên nhất trong quá trình thực thi ký hiệu. Trong thực tế, hiệu suất của các chế giải quyết được sử dụng bởi một kỹ thuật thực thi ký hiệu có thể ảnh hưởng đáng kể đến hiệu

suất tổng thể của nó. Không may, giải quyết hạn chế này chủ yếu được sử dụng trong thực thi ký hiệu không tận dụng bất kỳ ngữ cảnh và thông tin tên miền có sẵn. Để giải quyết vấn đề này, đề tài đã sử dụng một chiến lược tối ưu hóa mới, sử dụng miền và các thông tin theo ngữ cảnh để tối ưu hóa hiệu suất của giải quyết các ràng buộc trong quá trình thực thi ký hiệu.

Quá trình thực thi ký hiệu thực thi một chương trình với đầu là quá trình kiểm tra tất cả các lộ trình có thể trong chương trình . Đối với mỗi lộ trình, quá trình thực thi ký hiệu cập nhật các trạng thái mang tính biểu tượng của chương trình theo ngữ nghĩa. Ở phần cuối của quá trình thực thi ký hiệu, rang buộc lộ trình được đưa váo các bộ solver và thông qua các SMT để tìm ra kết quả

Phương án tối ưu sau đây được gọi là phương pháp rút gọn miền, với 2 ràng buộc C_a và C_b trong 1 tập ràng buộc C được gọi là phụ thuộc nếu ít nhất thỏa mãn :

- Phụ thuộc trực tiếp: C_a và C_b cùng tồn tại trong một ràng buộc.
- Phụ thuộc không trực tiếp: C_a và C_b có phụ thuộc trực tiếp tới một biến C_c.

Giải pháp chứa ràng buộc không có sự phụ thuộc với các biến ràng buộc mục tiêu thì không ảnh hưởng đến kết quả.vì vậy ta có thể loại bỏ ràng buộc này khỏi ràng buộc lộ trình và đưa vào trở lại sau quá trình thực thi. Đối với các ràng buộc thực tế hơn, đơn giản hóa này có thể dẫn đến tiết kiệm đáng kể về số lượng hạn chế mà bộ giải SMT phải xử lý.

Cách tiếp cận tối ưu dựa trên việc giảm các ràng buộc trên miền dữ liệu được thực hiện như sau

- Bước 1: Chia các biến mục tiêu thành các nhóm theo độ phụ thuộc.
- Bước 2: Tìm 2 biến có độ phụ thuộc lớn nhất và bé nhất.
- Bước 3: Thay thế lần lượt các biến mục tiêu với giá trị đầu vào với từng nhóm và thực hiện bước lược bỏ các phụ thuộc như đã trình bày ở trên.

2.5. Cài đặt tối ưu các ràng buộc với JPF

a. Kiến trúc của JPF và mở rộng JPF-SYMBC

b. Các thiết lập của JPF

c. Cài đặt tối ưu ràng buộc

d. Thuật toán tối ưu

Thuật toán tối ưu các ràng buộc được trình bày trong chương 2 được triển khai như sau:

Input: Ràng buộc lộ trình, giá trị các biến mục tiêu và biến cố định, bộ giải SMT.

Output: kết quả sau khi giải bằng SMT (sat, usat hoặc unknown).

```
Constraint cur = getTargetConstraint(index, pc)
List targetsymlist = getSymVars(cur)
result = unknown
for i = 1 to length(targetsymlist) do
 groups[] = formGroupsofX(i, targetsymlist)
 List dependencies = []
 for a = 0 to length(groups) do
        dependencies[a] = getDependencies(groups[a], pc)
 end
 int selectDep[0] = findSmallestDepIndex(dependencies)
 int selectDep[1] = findLargestDepIndex(dependencies)
 for b = 0 to length(selectedDep) do
    List grpDep = dependencies[(selectDep[b])]
    List   symbolicVars   =   join(groups(selectDep[b]),
grpDep)
```

```
        newPC = modifyPC(symbolicVars,concreteValuesMapping,pc)
        output = callSolver(solver,newPC)
        if output = "sat" then
            return output
        end
        else if output = "unsat" then
            result = output
        end
    end
    end
```

Trong thuật toán tối ưu hoá ràng buộc, chúng tôi có một số phương thức quan trọng:

- findSmallestDepIndex(): nhận đầu vào là một tập các phụ thuộc, và trả về giá trị của phụ thuộc có ít sự phụ thuộc nhất.
- findLargestDepIndex(): nhận đầu vào là một tập các phụ thuộc, và trả về giá trị của phụ thuộc có ít sự phụ thuộc nhất.
- modifyPC(): Thay thế các giá trị ký hiệu của biến mục tiêu bằng các giá trị hiện thời của lộ trình.

CHƯƠNG 3
CÀI ĐẶT VÀ THỬ NGHIỆM

3.1. Mọi trường cài đặt

3.2. Thử nghiệm

3.2.1. Đánh giá khả năng bao phủ

Để đánh giá độ bao phủ chúng tôi sẽ tiến hành thử nghiệm sinh dữ liệu thử của công cụ trên các ba phương thức testMe1, testMe2, testMe3 để kiểm tra sự thay đổi của công cụ sau khi được cài đặt thuật toán tối ưu.

Phương thức testMe1, testMe2, testMe3 được tiến hành thực hiện sinh dự liệu thử tự động với 2 trường hợp như sau: chương trình sinh dữ liệu kiểm thử được cài thuật toán tối ưu và chương trình sinh dữ liệu kiểm thử không được cài đặt thuật toán tối ưu. Trong trình bày của bảng kết quả, ký hiệu "-" đại diện cho việc dữ liệu đầu vào của biến có thể nhận bất cứ giá trị nào.

3.2.2. Đánh giá tính hiệu quả

Để đánh giá tính hiệu quả trong việc giảm thời gian thực thi của công cụ khi được cài đặt thuật toán tối ưu, một bộ dữ liệu (được ghi lại chi tiết trong phần Phụ lục) gồm 10 phương phức được chọn và tiến hành quá trình sinh dữ liệu thử tự động. Quá trình sinh dữ liệu thử tự động được thực hiện trong 2 trường hợp: công cụ được cài đặt thuật toán tối ưu và công cụ không được cài đặt thuật toán tối ưu. Bảng kết quả sau được thực thi trên hệ thống máy tính với cấu hình: CPU core i3, 3GB RAM.

Bảng 3.7. Kết quả thử nghiệm

Tên phương thức kiểm thử	**Thời gian thực thi (milisecond)**		
	Không cài đặt thuật toán tối ưu	**Cài đặt thuật toán tối ưu**	**Tỷ lệ giảm**
Test1	1332	1032	24%
Test2	1282	867	33%
Test3	1334	932	31%
Test4	1055	739	30%
Test5	1389	956	32%
Test6	918	643	30%
Test7	1277	893	31%
Test8	1364	971	29%
Test9	984	692	30%
Test10	1191	834	27%

3.3. Kết luận

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

1. Kết quả đạt được

Qua đề tài "Xây dựng công cụ sinh dữ liệu thử tự động cho chương trình Java", tôi đã tìm hiểu, nghiên cứu được một số kết quả sau:

Thứ nhất, tìm hiểu về cách xây dựng và hoạt động của và phương pháp sinh dữ liệu kiểm thử tự động cho ngôn ngữ Java cũng như nghiên cứu về các thách thức hiện nay.

Thứ hai, đề xuất và cài đặt giải pháp tối ưu trong việc sinh dữ liệu thử tự động nhằm tối ưu hóa thời gian nhưng vẫn đảm bảo tính đúng đắn, toàn vẹn và khả năng bao phủ của bộ dữ liệu thử tự động.

2. Hạn chế

Giải pháp đề ra dựa trên sự phụ thuộc của các ràng buộc với nhau, vì vậy độ hiệu quả của thuật toán sẽ bị ảnh hưởng rất lớn trong trường hợp dữ liệu đầu vào có sự phụ thuộc quá nhiều với nhau.

Giải pháp đề ra vẫn còn thực thi ký hiệu để bao phủ tất cả các nhánh của cây thực thi, do đó trong trường hợp chương trình quá phức tạp công cụ có thể không hoạt động được vì sự bùng nổ về đường đi.

Một điểm hạn chế khác của thuật toán này là trong trường hợp dữ liệu đầu vào quá đơn giản thì việc thực hiện các biện pháp tiền xử lý của thuật toán sẽ khiến cho chương trình mất thêm tài nguyên trong việc thực hiện.

3. Hướng phát triển

Trong quá trình nghiên cứu, bộ SMT Choco không cho kết quả tốt như các bộ SMT khác như Z3 hay CVC3. Vì vậy, trong thời gian tới chúng tôi sẽ tiếp tục nghiên cứu và triển khai tích hợp các bộ

SMT mới và có khả năng hỗ trợ nâng cấp tốt hơn vào trong chương trình sinh dữ liệu kiểm thử.

Đà Nẵng, 21/3/2017

Lê Hữu Nam