TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

**KHOA CÔNG NGHỆ THÔNG TIN**

CÔNG TRÌNH THAM DỰ

HỘI NGHỊ SINH VIÊN NGHIÊN CỨU KHOA HỌC CẤP KHOA

NĂM HỌC 2016 – 2017

Tên công trình: **Xây dựng công cụ kiểm chứng một số tính chất của chương trình dựa trên kĩ thuật thực thi tượng trưng**

Họ và tên sinh viên: **Nguyễn Đức Thuần** Nam

Lớp: K59CLC Khoa: Công nghệ thông tin

Họ và tên sinh viên: **Nguyễn Bá Tú** Nam

Lớp: K59CLC Khoa: Công nghệ thông tin

Người hướng dẫn: **TS. Tô Văn Khánh**

Hà Nội, 2017

# LỜI CAM ĐOAN

Chúng tôi xin cam đoan rằng những nghiên cứu về kiểm chứng một số tính chất của chương trình dựa trên kĩ thuật thực thi tượng trưng trong bài báo cáo này là của chúng tôi và chưa từng được nộp như một báo cáo nghiên cứu khoa học nào tại trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội hoặc bất kỳ trường đại học nào khác. Những gì chúng tôi viết ra không sao chép từ các tài liệu, không sử dụng các kết quả của người khác mà không trích dẫn cụ thể.

Chúng tôi xin cam đoan công cụ VTSE – Kiểm chứng một số tính chất của chương trình dựa trên kĩ thuật thực thi tượng trưng là do chúng tôi phát triển, không sao chép mã nguồn của người khác. Nếu sai chúng tôi hoàn toàn chịu trách nhiệm theo quy định của trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội.

Hà Nội, tháng 03 năm 2017

Xác nhận của GV hướng dẫn Sinh viên

**Tô Văn Khánh** **Nguyễn Đức Thuần Nguyễn Bá Tú**

Mục lục

[**I. Giới thiệu** 1](#_Toc478426704)

[**II. Thực thi tượng trưng** 3](#_Toc478426705)

[1. Tổng quan về kĩ thuật thực thi tượng trưng 3](#_Toc478426706)

[1.1. Kĩ thuật thực thi tượng trưng 3](#_Toc478426707)

[1.2. Satisfiability Modulo theories (SMT) 4](#_Toc478426708)

[1.2.1. Logic mệnh đề (Propositional logic) 4](#_Toc478426709)

[1.2.2. SMT 5](#_Toc478426710)

[2. Các công cụ liên quan 5](#_Toc478426711)

[2.1. KLEE 5](#_Toc478426712)

[2.2. Cascade 7](#_Toc478426713)

[**III. Công cụ kiếm chứng chương trình VTSE** 9](#_Toc478426714)

[1. Kiến trúc của VTSE 9](#_Toc478426715)

[2. Trừu tượng hóa chương trình dựa trên thực thi tượng trưng 10](#_Toc478426716)

[2.1. Cây cú pháp trừu tượng (AST) 11](#_Toc478426717)

[2.2. Xây dựng đồ thị luồng điều khiển 11](#_Toc478426718)

[2.3. Loại bỏ chu trình trong CFG 15](#_Toc478426719)

[2.4. Đánh chỉ số (indexing) cho các tham biến và xây dựng biểu thức metaSMT 17](#_Toc478426720)

[2.5. Công thức tổng quát cho phương thức 20](#_Toc478426721)

[2.6. Kết hợp điều kiện của người dùng và đưa vào bộ giải 21](#_Toc478426722)

[3. Minh họa hoạt động của VTSE 22](#_Toc478426723)

[**IV. Thực nghiệm** 26](#_Toc478426724)

[1. Xây dựng công cụ VTSE 26](#_Toc478426725)

[2. Một số ứng dụng thực nghiệm 26](#_Toc478426726)

[2.1. Kiểm tra giá trị trả về của một hàm 27](#_Toc478426727)

[2.2. Kiểm tra giá trị trả về của phương thức có luôn thuộc một khoảng cho trước hay không 28](#_Toc478426728)

[2.3. Kiểm tra các hậu điều kiện (Post-condition) thỏa mãn hay không thỏa mãn với các tiền điều kiện(Pre-condition) do người dùng nhập vào. 29](#_Toc478426729)

[2.4. Kiểm tra tính chất của các điều kiện phức hợp do người dùng nhập vào 30](#_Toc478426730)

[2.5. Các ứng dụng hướng tới sau này 31](#_Toc478426731)

[**Tổng kết** 32](#_Toc478426732)

[**Tài liệu tham khảo** 33](#_Toc478426733)

**DANH MỤC CÁC THUẬT NGỮ VIẾT TẮT**

|  |  |
| --- | --- |
| Thuật ngữ viết tắt | Viết đầy đủ |
| SMT | Satisfiability modulo theories |
| SE | Symbolic execution |
| SAT | Satisfiability |
| UNSAT | Unsatisfiability |
| AST | Abstract Syntax Tree |
| CFG | Control Flow Graph |

**DANH SÁCH CÁC HÌNH VẼ**

[Hình 1: Xây dựng đường thực thi cho chương trình tráo đổi giá trị của hai số nguyên x và y [1] 3](#_Toc478426684)

[Hình 2: Đường thi hành thực thi tượng trưng của chương trình đổi giá trị của 2 số [1] 4](#_Toc478426685)

[Hình 3: Sơ đồ hoạt động của KLEE [6] 5](#_Toc478426686)

[Hình 4: Kiến trúc hệ thống của Cascade [7] 7](#_Toc478426687)

[Hình 5: Tập tin điều khiển của Cascade [7] 8](#_Toc478426688)

[Hình 6: Kiến trúc của VTSE 9](#_Toc478426689)

[Hình 7:Ví dụ về AST [8] 11](#_Toc478426690)

[Hình 8: Các ví dụ về CFG 12](#_Toc478426691)

[Hình 9. Minh họa ý tưởng của phương pháp unfold CFG 16](#_Toc478426692)

[Hình 10: Cây cú pháp trừu tượng minh họa 23](#_Toc478426693)

[Hình 11: Đồ thị luồng điều khiển minh họa 24](#_Toc478426694)

[Hình 12: Đồ thị luồng điều khiển minh họa sau khi đánh chỉ số 25](#_Toc478426695)

[Hình 13: Mã nguồn phương thức tính giá trị tuyệt đối 27](#_Toc478426696)

[Hình 14: Kết quả kiểm tra giá trị trả về của phương thức 27](#_Toc478426697)

[Hình 15: Mã nguồn phương thức dùng kiểm tra miền giá trị 28](#_Toc478426698)

[Hình 16: Kết quả kiểm tra giá trị trả về có thuộc khoảng giá trị không 29](#_Toc478426699)

[Hình 17: Mã nguồn phương thức tính tổng từ 1 đến n 29](#_Toc478426700)

[Hình 18: Kết quả kiểm tra hậu điều kiện có thỏa mãn tiền điều kiện hay không. 30](#_Toc478426701)

[Hình 19: Kết quả kiểm tra tính chất của các điều kiện phức hợp 31](#_Toc478426702)

# I. Giới thiệu

Máy tính và phần mềm máy tính ngày càng đóng vai trò quan trọng trong nhiều lĩnh vực của đời sống xã hội như: kinh tế, giao thông, vũ trụ hay là trong lĩnh vực y tế, chăm sóc sức khỏe con người. Cho nên, việc đảm bảo chất lượng phần mềm là hết sức cần thiết và là công việc cực kì quan trọng, đặc biệt là trong các lĩnh vực y tế và hàng không vũ trụ. Bởi chỉ cần một sai sót nhỏ của hệ thống cũng có thể gây ra những thiệt hại rất lớn cả về tính mạng con người cũng như về kinh tế.

Hai kỹ thuật truyền thống đã và đang được sử dụng để đảm bảo chất lượng phần mềm là kiểm thử phần mềm và kiểm chứng phần mềm (Software verification). Tuy nhiên việc sử dụng các phương pháp kiểm thử chỉ đảm bảo hệ thống không có lỗi với một số bộ kiểm thử nhất định mà không thể chứng minh được hệ thống không có lỗi. Để chứng minh tính đúng đắn của hệ thống ta cần áp dụng các phương kiểm chứng chương trình. Nhưng các phương pháp kiểm chứng như phương pháp kiểm chứng mô hình (Model checking) lại gặp phải vấn đề về việc bùng nổ trạng thái với các hệ thống phức tạp. Chính vì vậy mà phương pháp kiểm chứng chương trình dựa trên kỹ thuật thực thi tượng trưng (Symbolic execution) là kỹ thuật được sử dụng thay thế cho phương pháp kiểm chứng mô hình thực hiện trên các hệ thống lớn và phức tạp.

Trong bài báo cáo này chúng tôi xin đề xuất ra phương pháp kiểm chứng một số tính chất của chương trình dựa trên kĩ thuật thực thi tượng trưng. Phương pháp này đề xuất việc trừu tượng hóa một đoạn mã nguồn bởi các biểu diễn là các biểu thức logic vị từ cấp một (First-order Logic), và kết hợp các điều kiện do người dùng đưa vào và đưa vào các bộ giải SMT Solver.

Bài báo cáo cũng đề xuất ra một công cụ mới là VTSE (Verification Tool based on Symbolic Execution), công cụ kiểm chứng một số tính chất của chương trình dựa trên kĩ thuật thực thi tượng trưng. VTSE có các ứng dụng trong việc kiểm tra giá trị trả về của một phương thức, kiểm tra giá trị trả về đó có luôn nằm trong một khoảng giá trị cho trước hay không hoặc VTSE có thể kiểm tra các điều kiện của người dùng muốn kiểm tra trên một phương thức có luôn đúng với bộ giá trị cho trước của các tham biến đầu hay không.

Bài báo cáo sẽ tập trung trình bày về việc trừu tượng hóa chương trình dựa trên kĩ thuật thực thi tượng trưng, cách biểu diễn biểu thức trừu tượng hóa của chương trình bằng các biểu diễn logic vị từ cấp một (First-order logic). Tiếp theo bài cáo sẽ giới thiệu về VTSE và các ứng dụng của công cụ.

# II. Thực thi tượng trưng

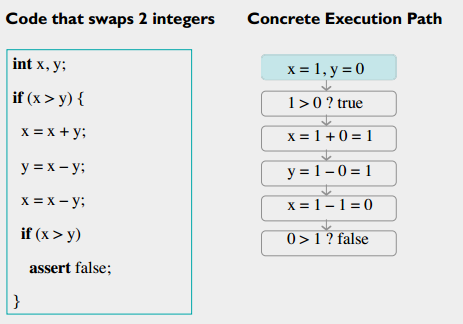
## **1. Tổng quan về kĩ thuật thực thi tượng trưng**

## **Kĩ thuật thực thi tượng trưng**

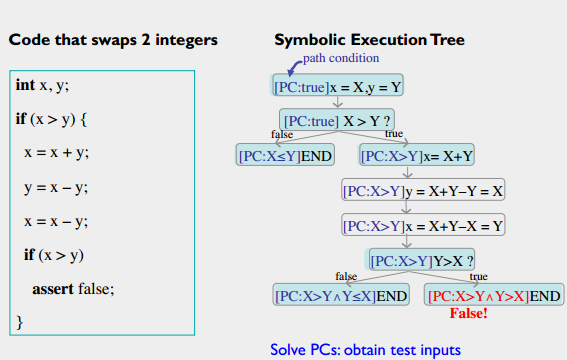
Symbolic execution là một phương pháp để phân tích các chương trình với các đầu vào không xác định. Thay vì sử dụng các giá trị cụ thể của đầu vào, nó sử dụng các biểu tượng như là đầu vào và thực hiện chương trình. Đối với mỗi nhánh thực hiện, điều kiện nhánh được ghi nhận. Điều này sẽ hữu ích khi tính toán các điều kiện về đầu vào để thực hiện đi theo nhánh đó cũng như kiểm tra tính satisfiability của nhánh nhằm thể bỏ qua những con đường không thể chạy tới (do không thỏa mãn điều kiện nhánh).

Symbolic state chứa thông tin về giá trị biểu tượng / biểu thức cho các biến, điều kiện đường và đếm chương trình.

Symbolic execution đã nhận được sự quan tâm trong những năm gần đây do sự phát triển của các thuật toán, sự tiến bộ của các khả năng tính toán và các ứng dụng decision procedures cũng như các yêu cầu của bài toán sinh test-case, phát hiện lỗi,…



Hình 1: Xây dựng đường thực thi cho chương trình tráo đổi giá trị của hai số nguyên x và y [1]



Hình 2: Đường thi hành thực thi tượng trưng của chương trình đổi giá trị của 2 số [1]

Bằng cách có được path conditions của tất cả các nhánh trong chương trình, test-cases có thể được tạo ra để tất cả các nhánh được phát hiện (coverage testing) cũng như các trạng thái và quá trình chuyển đổi có thể được diễn ra.

## **Satisfiability Modulo theories (SMT)**

## 1.2.1. Logic mệnh đề (Propositional logic)

Logic mệnh đề là nhánh đơn giản nhất của toán logic, có thể được hình dung bởi: Một mệnh đề là một câu có thể là đúng hay sai; nó phải là đúng hoặc sai nhưng không được là cả hai.

Trong logic mệnh đề, các câu được coi là đơn vị cơ bản (không sét sâu hơn được), và vì thế, logic mệnh đề không nghiên cứu những tính chất sâu hơn và các quan hệ mà phụ thuộc vào các bộ phận của các câu khác, chẳng hạn như các chủ ngữ và vị ngữ của câu logic.

Một định dạng rất quan trọng của logic mệnh đề là dạng chuẩn nối tiếp (conjunctive normal form) nơi một mệnh đề là một phép giao các chữ; nếu không thì là AND của ORS. Ví dụ:

### 1.2.2. SMT

Satisfiability Modulo theories (SMT) [2] là một lĩnh vực nghiên cứu các phương pháp kiểm tra tính thỏa mãn của các logic vị từ cấp một đối với một số lý thuyết *T* nào đó mà ta quan tâm.

SMT Solver là các bộ giải được cài đặt để kiểm tra tính thỏa mãn của các công thức logic vị từ cấp một. Trong công cụ được đề cập trong bản báo cáo này, chúng tôi sử dụng một bộ kết hợp các SMT Solver khác nhau để đem lại hiệu quả cao nhất. Cụ thể các SMT Solver được sử dụng đó là: Z3 [3], raSAT [4] và CVC4 [5].

## **Các công cụ liên quan**

### 2.1. KLEE

#### Tổng quan

KLEE [6] là một công cụ thực thi tượng trưng, có khả năng sinh các ca kiểm thử một cách tự động mà đạt được độ bao phủ cao trên một tập đa dạng của các chương trình phức tạp và môi trường chuyên sâu (được dịch từ “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs” - <http://hci.stanford.edu/cstr/reports/2008-03.pdf> ). Klee thuộc nhóm công cụ sử dụng kĩ thuật thực thi tượng trưng động ( Dynamic Symbolic Execution) với việc sử dụng kết hợp các thực thi tượng trưng và thực thi cụ thể

#### Mô hình hoạt động của KLEE

metaSMT

Boolector

Z3

STP

Constraint  
Independence

Branch Cache

Counterexample  
Cache

Hình 3: Sơ đồ hoạt động của KLEE [6]

* Constaint Independence

Tại bước này, Klee loại bỏ các ràng buộc không cần thiết. Ví dụ ta có ràng buộc C + {x + 2y = 5; y > 3; z > 20 ; w > 5} khi đi xuống một nhánh E = x > y thì ta có thể loại bỏ z > 20 và w > 5 mà không anh hưởng đến E.

* Branch Cache

Klee lưu lại kết quả truy vấn đến các nhánh vào bộ đệm branch cache

* Counterexample Cache

Klee thiết lập một tập hợp các điều kiện hoặc thiết lập một Counterexample Cache (phản ví dụ).

Klee sử dụng counterexample Cache để có thể đưa ra được kết quả một cách nhanh hơn để tiết kiệm thời gian cũng như chi phí.

* Tiếp theo, các ràng buộc được đưa đến các bộ giải ràng buộc (Constrain solver) thông qua cơ chế mạng metaSMT. Klee sẽ lựa chọn bộ giải phù hợp nhất cho từng trường hợp.

#### Chức năng

Chức năng chính của KLEE là sinh ra các ca kiểm thử một cách tự động với độ phủ cao, các ca kiểm thử có khả năng phát hiện được lỗi của chương trình thông qua các tập tin đầu ra kết quả.

Ngoài ra, KLEE còn được sử dụng trong các công cụ khác với những ứng dụng đa dạng khác nhau, các công cụ này đều thừa kế phát triển hay mở rộng KLEE dựa trên khả năng thăm dò các đường thi hành của chương trình và tự động sinh các ca kiểm thử. Ví dụ:

* EDS – Execution Synthesis: công cụ có khả năng sửa lỗi chương trình C một cách tự động
* KLEENET: có khả năng tìm ra những lỗ hổng bảo mật trong hệ thống phân phối trước khi triển khai và tạo điều kiện cho việc mở rộng hệ thống mạng
* KLOVER: Có khả năng sinh ca kiểm thử tự động cho chương trình C++

#### Hạn chế

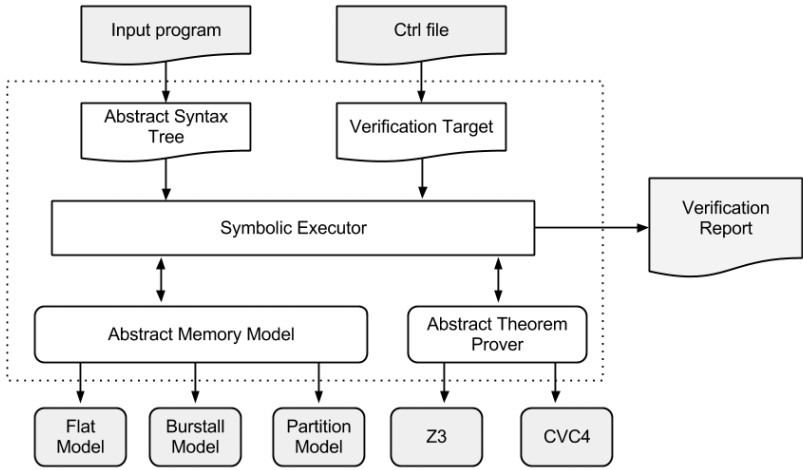
* KLEE chưa xử lý được vấn đề bùng nổ đường thi hành ( The path explosion problem )
* KLEE cũng không thể thực hiện với lặp với điều kiện tượng trưng và các đối tượng có kích cỡ tượng trưng (symbolic size)
* Ngoài ra, KLEE cũng không thể khám phá các vũng mã chết và vùng mã không thể với tới được. (unreachable code and dead code)

### 2.2. Cascade

#### Tổng quan

Cascade [7] là một công cụ phân tích chương trình tĩnh, nhận một chương trình và một tập tin điều khiển là đầu vào. Tập tin điều khiển quy định cụ thể một hay hay nhiều assertions để kiểm tra cùng với các giới hạn trong hoạt động của chương trình. Công cụ sinh ra các điều kiện kiểm chứng cho các assertion cụ thể và kiểm tra chúng bằng cách sử dụng một SMT solver và đồng thời sinh ra một minh chứng hoặc đưa ra một ví dụ cụ thể để chỉ ra rằng assertion đó có thể sai.

#### Thiết kế hệ thống của Cascade

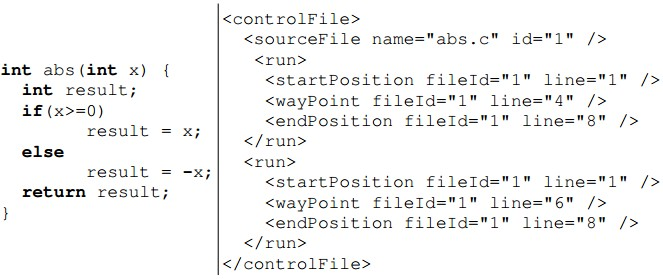


Hình 4: Kiến trúc hệ thống của Cascade [7]

* Tập tin điều khiển (Control file)

Tập tin điều khiển dùng để hướng dẫn thực thi tượng trưng, sử dụng ngôn ngữ XML và hỗ trợ các cấu trúc sau.

*Cấu trúc cơ bản.* Tất cả các tập tin điều khiển đều bắt đầu với một phần tên là *sourceFile*  mà đưa ra đường dẫn tới các file nguồn. Mỗi lần chạy sẽ bắt đầu với một dòng lệnh bắt đầu (startPosition) và kết với một dòng lệnh kết thúc (endPosition). Nếu mã nguồn chứa các nhánh, Cascade sẽ xem xét cả tất cả các nhánh. Nếu người dùng muốn thực thi một nhánh cụ thể, thì có thể yêu cầu một hay nhiều lệnh (wayPoint) trong khi thực thi. Một ví dụ về tập tin điều khiển



Hình 5: Tập tin điều khiển của Cascade [7]

*Lời gọi hàm.* Cascade hỗ trợ các lời gọi hàm qua nội tuyến. Nếu người dùng muốn thực thi một đường riêng biệt ở trong các hàm thì có thể sử dụng wayPoint tương tự như ở cấu trúc cơ bản.

*Vòng lặp.*  Theo mặc định, vòng lặp sẽ bị loại bỏ bằng cách tháo gỡ giới hạn vòng lặp. Một số mặc định của việc tháo gỡ vòng lặp (cho vòng lặp lặp lại một số lần nhất định) sẽ được quy định cụ thể trong dòng lệnh thực thi.

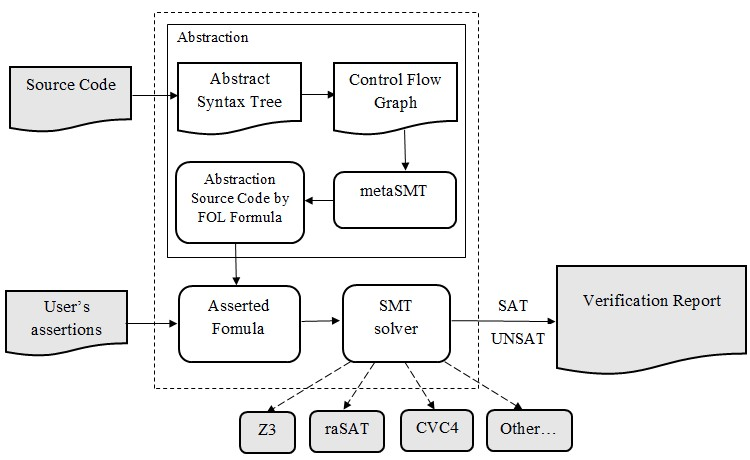
# III. Công cụ kiếm chứng chương trình VTSE

Chương này đề xuất phương pháp kiểm tra các tính chất chương trình được sử dụng trong công cụ VTSE – Verification Tool based on Symbolic Execution. Phương pháp đề xuất cách xây dựng biểu thức logic vị từ cấp một cho mã nguồn của chương trình. Sử dụng biểu thức xây dựng được để kiểm tra các tính chất của chương trình dựa vào SMT Solver.

Trong chương này, đầu tiên sẽ đưa ra kiến trúc của VTSE, sau đó nên lên phương pháp trừu tượng hóa chương trình để sinh ra biểu thức logic và cuối cùng đưa ra một ví dụ minh họa hoạt động của VTSE.

## **1. Kiến trúc của VTSE**

VTSE có kiến trúc như sau:



Hình 6: Kiến trúc của VTSE

VTSE có đầu vào gồm hai thành phần, thứ nhất là mã nguồn của chương trình (Source code), thứ hai là điều kiện do người dùng nhập vào (User’s assertion). Đầu ra của nó là một bản báo cáo hoạt động của công cụ (Verification Report), đưa ra kết quả điều kiện của người dùng có luôn thỏa mãn trên một điều kiện cho trước hay không.

Sau khi đã có đầu thứ nhất là mã nguồn của chương trình, VTSE sẽ tiến hành việc trừu tượng hóa chương trình dựa trên kĩ thuật thực thi tượng trưng thông qua việc tiến hành tuần tự các bước, tạo cây cú pháp trừu tượng (AST), tạo đồ thị luồng điều khiển (CFG), tạo công thức metaSMT rồi tiến hành biểu diễn trừu tượng hóa chương trình bằng các biểu diễn logic vị từ cấp một. Từ đây kết hợp với các điều kiện của người dùng (cũng được biểu diễn lại bằng các biểu diễn logic vị từ cấp một) để có công thức cuối cùng và đưa vào bộ giải SMT Solver. VTSE sử dụng rất nhiều các bộ giải khác nhau để đem lại hiệu năng lớn nhất trong việc giải các biểu diễn logic vị từ cấp một, trong đó sử dụng chủ yếu là Z3 [3], raSAT [4] và CVC4 [5].

Báo cáo hoạt động của công cụ sẽ đưa ra kết quả của việc chạy chương trình, có thể nhận được các kết quả sau:

- YES tương ứng với điều kiện của người dùng đặt lên phương thức luôn đúng

- NO tương ứng với điều kiện của người dùng sai trong một số trường hợp và VTSE sẽ đưa ra một phản ví dụ (counter example) để chứng minh cho điều này.

## **2. Trừu tượng hóa chương trình dựa trên thực thi tượng trưng**

Đa số các công cụ khác đều chỉ trừu tượng hóa cho một đường thi hành cụ thể nào đó trong chương trình, và do đó có thể gặp phải trường hợp bùng nổ trạng thái và chương trình sẽ có thể chạy vô hạn, đây là một vấn đề chung mà tất cả các chương trình kiểm chứng chương trình đều gặp phải, điển hình là các công cụ kiểm chứng dựa trên kĩ thuật kiểm chứng mô hình (model checking). Việc áp dụng kĩ thuật thực thi tượng trưng vào phương pháp kiểm chứng sẽ giúp loại bỏ được vấn đề bùng nổ trạng thái, từ đó có thể áp dụng vào các hệ thống vừa và lớn.

Thay vì biểu diễn từng đường thi hành cụ thể, báo cáo đề xuất việc tổng quát hóa cho một phần chương trình bằng một biểu thức logic vị từ cấp một dựa trên thực thi tượng trưng. Biểu thức logic được sinh ra sẽ có nhiều ứng dụng trong việc phân tích, kiểm tra mã nguồn của chương trình. Trong nghiên cứu hiện nay, chúng tôi hướng đế việc sử dụng biểu thức logic này nhằm kiểm tra sự thỏa mãn của mã nguồn với các điều kiện do người dùng yêu cầu.

Phần này sẽ trình bày về việc trừu tượng hóa chương trình để sinh ra biểu thức logic, các thành phần quan trọng nhất của VTSE.

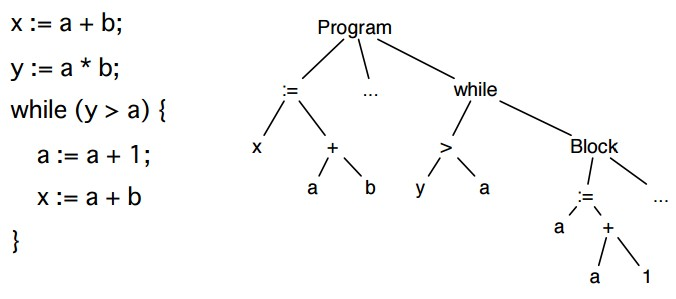
### 2.1. Cây cú pháp trừu tượng (AST)

Trong việc phân tích chương trình, cây cú pháp trừu tượng (AST, Abstract Syntax Tree [8]) đóng một vai trò rất quan trọng. AST là một cây có giới hạn, có nhãn và định hướng nhằm mục đích biểu diễn cấu trúc cú pháp trừu tượng mã nguồn của chương trình.

Ở đây, trừu tượng hóa cú pháp không phải là biễu diễn mọi chi tiết có mặt ở trong cú pháp thật của chương trình, mà nó sẽ định nghĩa các cấu trúc, các cú pháp khởi tạo.

Trong việc trừu tượng hóa chương trình, cây AST dùng để xây dựng nên đồ thị dòng điều khiển (Control Flow Graph), là bước đầu tiên để phân tích chương trình cũng như dựa vào đó để lấy được các thành phần cấu tạo nên chương trình.

Ví dụ về AST:



Hình 7:Ví dụ về AST [8]

### 2.2. Xây dựng đồ thị luồng điều khiển

#### 2.2.1. Giới thiệu về đồ thị luồng điều khiển

Đồ thị luồng điều khiển (Control Flow Graph, CFG) là một đồ thị có hướng mô tả cấu trúc logic của chương trình một cách trực quan và đơn giản hơn, gồm có các đỉnh tương ứng với các câu lệnh/nhóm câu lệnh và các cạnh là các dòng điều khiển giữa các câu lệnh/nhóm câu lệnh. Đỉnh đầu tiên của CFG là trạng thái đầu tiên của hàm, đỉnh cuối cùng là trạng thái kết thúc của hàm. Đỉnh i nối đến đỉnh j thì câu lệnh tương ứng đỉnh j có thể được thực thi sau khi thực hiện câu lệnh tương ứng ở đỉnh j.

#### 2.2.2. Cấu trúc, các thành phần của CFG

Trong một CFG điển hình, mỗi đỉnh sẽ biểu thị một khối lệnh cơ bản, mỗi một đường nối các đỉnh trong đồ thị sẽ biểu diễn sự chuyển tiếp giữa các khối lệnh. Trong đa phần các đồ thị, có hai khối lệnh điển hình, đó là khối lệnh thực thể và khối kết thúc.

CFG là một đồ thị có hướng, các chu trình thể thiện vòng lặp và mỗi một đường đi trong từ đỉnh đầu đến đỉnh cuối của nó thể hiện sự mô phỏng hóa một đường thi hành cụ thể có trong chương trình.

#### 2.2.3. Ví dụ về CFG

|  |  |
| --- | --- |
|  |  |

Hình 8: Các ví dụ về CFG

#### 2.2.4. Chức năng của CFG

CFG là một thành phần rất quan trọng và cần thiết. Nó được sử dụng rộng rãi trong nhiều công cụ về kiểm chứng, kiểm thử, phân tích chương trình và các trình biên dịch.

Trong công cụ này, CFG được sử dụng để có thể nắm bắt được tất cả các đường thi hành trong chương trình, để sau đó tiến hành gỡ các chu trình lặp của đồ thị và đưa ra công thức tổng quát hóa của nó.

#### 2.2.5. Tạo đồ thị luồng điều khiển từ cây cú pháp trừu tượng

Đồ thị luồng điều khiển được sinh ra từ cây cú pháp trừu tượng của phương thức. Bằng cách duyệt theo thứ tự trước trên cây AST sẽ lấy ra lần lượt các cấu lệnh trong phương thức tương ứng với từng nút trong cấy AST. Các nút trong CFG bao gồm:

- Nút đơn: tương ứng với các câu lệnh đơn giản trong chương trình như câu lệnh gán, câu lệnh khởi tạo, câu lệnh return.

- Nút điều kiện: chứa biểu thức điều kiện tượng trưng cho câu lệnh rẽ nhánh với 2 con trỏ tới 2 nhánh then và else trong CFG.

- Nút đánh dấu: nút đánh dấu bắt đầu, kết thúc phương thức; nút bắt đầu, kết thúc đoạn lặp; nút kết thúc đoạn rẽ nhánh, nút đồng bộ, nút rỗng.

Các biểu phép toán, biểu thức điều kiện trong các nút sử dụng trực tiếp các nút tương ứng trong cây AST để thuận tiện cho việc xử lý ở các bước sau.

**Thuật toán** buildCFG(ASTNode)

**Đầu vào:** nút ASTNode trong cây AST

**Đầu ra:** hàm trả bề cặp PairNode chưa tham chiếu tới nút đầu và nút cuối của CFG được xây dựng từ ASTNode

1. **if**  (ASTNode là câu lệnh đơn giản)
2. **2** node := nút CFG đơn chưa ASTNode

**3** tạo PairNode với nút đầu và nút cuối đều là node

**4 else if** (ASTNode là nút khối)

**5** begin := nút rỗng

6 last := begin

**7 for** (với mỗi nút n trong khối)

**8** subCFG = buildCFG(n)

**9** // nối subCFG vào CFG của khối

**10** last.next := subCFG.begin

**11** last := subCFG.end

**12** **if** (last == begin) // khối lệnh rỗng

**13** pairNode := null

**12** pairNode := PairNode(begin.next, last);

**13**  **else if** (ASTNode là câu lệnh if - else)

**14** thenBranch := CFG tương ứng với nút then

**15** elseBranch : = CFG tương ứng với nút then

**16** conditionNode := Nút chứa biểu thức điều kiện

**17** conditionNode.then := thenBranch.begin

**18** conditionNode.else := elseBranch.begin

**19** tạo nút endCondition để kết thúc đoạn CFG của if – else

**20** thenBranch.end = endCondition

**21** elseBranch.end = endCondition

**22** pairNode := PairNode(conditionNode, endCondition)

**23** **else if** (ASTNode là câu lệnh for)

**24** init := tạo CFG của nút khởi tạo trong for

**25** loopCondition := nút chứa điều kiện lặp

**26** body := tạo CFG của thân vòng for

**27**  update := tạo CFG của nút cập nhật giá trị các biến lặp

**28** loopBody := nối body với update // coi toàn bộ là thân vòng lặp

**29** tạo nút beginLoop và endLoop

**30** loopCondition.then := loopBody.begin

**31** tạo nút endCondition

**32** loopCondition.else := endCondition

**33** beginLoop.next := conditionNode

**34** endCondition.next := endLoop

**35** init.next := beginLoop

**36** pairNode := PairNode(init, endLoop)

**37** **else if** (ASTNode là câu lệnh do)

**38** loopCondition := nút chứa điều kiện lặp

**39** loopBody := tạo CFG của thân vòng lặp

**40** tạo nút beginLoop và endLoop

**41**  loopCondition.then := loopBody.begin

**42** tạo nút endCondition

**43** loopCondition.else := endCondition

**44** beginLoop.next := conditionNode

**45** endCondition.next := endLoop

**46** pairNode := PairNode(beginLoop, endLoop)

**47** **return** pairNode

Thuật toán mô tả quá trình xây dựng CFG từ cây AST. Đối với các câu lệnh đơn, thuật toán tạo ra nút tương ứng chứa biểu thức của câu lệnh đơn đó và tạo nút pairNode có con trỏ tới đầu và kết thúc đều là câu lệnh vừa tạo.

Đối với nút đại diện cho câu lệnh if, thuật toán tạo nút conditionNode chứa biểu thức. Tạo CFG tương ứng với 2 nhánh then, else rồi nối vào conditionNode. Cuối cùng tạo nút endCondition và nối vào 2 nhánh then, else.

Đối với các lệnh lặp, thuật toán tạo ra nút chứa điều kiện lặp. Thuật toán coi CFG tương ứng với thân vòng lặp là nhánh then của nút điều kiện. Còn nhánh else sẽ không chưa gì cả. Rồi tạo nút beginLoop và endLoop để đánh dấu bắt đầu và kết thúc vòng lặp. Điều này nhằm mục đích tạo thuận lợi cho việc unfold CFG.

### 2.3. **Loại bỏ chu trình trong CFG**

Vòng lặp là một trong những vấn đề lớn mà thực thi tượng trưng phải giải quyết. Các vòng lặp thường lặp với một số lần không cố định trong mỗi lần thực thi. Đó là nguyên nhân chính của việc bùng nổ số lượng các đường thi hành của chương trình, có thể dẫn tới số lần lặp vô hạn do không thể xác định chính xác số lần lặp. Xử lý và phân tích vòng lặp là một bài toán khó, phức tạp không chỉ đối với thực thi tượng trương mà đối với tất cả các phương pháp phân tích mã nguồn chương trình. Tính đến hiện nay chưa có một giải pháp nào triệt để cho vấn đề này.

Trong báo cáo này, chúng tôi đề xuất ra một kĩ thuật để có thể gỡ bỏ được các chu trình có trong CFG. Kĩ thuật này được gọi là unfold CFG, và được tiến hành ngay sau khi tạo được CFG từ AST. Ý tưởng chính của kĩ thuật này là coi mỗi lần lặp của chương trình nh ư một lệnh rẽ nhánh if-else. Mỗi câu lệnh này sẽ có biểu thức điều kiện là điều kiện lặp của vòng lặp. Nhánh then của câu lệnh là các câu lệnh trong thân vòng lặp còn nhánh else sẽ không chứa câu lệnh nào. Lần lặp tiếp theo của vòng lặp sẽ là một câu lệnh cuối cung trong trong các câu lệnh của nhánh then. Cứ thế các câu vòng lặp sau sẽ trở thành câu lệnh con trong vòng lặp trước. Số lần lặp của vòng lặp hay số câu lệnh rẽ nhánh đươc tạo ra sẽ do người dùng yêu cầu.

Xét đoạn mã nguồn chưa vòng lặp sau:

while (i < n) {

sum = sum + i;

i++;

}

Với đoạn mã nguồn trên, ý tưởng unfold CFG được thể hiện trong hình dưới.

T

F

Đúng

Sai

T

F

*Hình 9.a) CFG của vòng lặp* *Hình 9.b) CFG của của câu lệnh rẽ nhánh tương ứng vơi một lần lặp*

Đúng

Sai

T

F

T

F

*Hình 9.c) CFG unfold với 2 lần lặp*

Hình 9. Minh họa ý tưởng của phương pháp unfold CFG

Như vậy, sau khi kêt thúc quá trình unfold CFG thì CFG của phương thức sẽ chỉ còn các nút đơn và các nút rẽ nhánh. Điều này tương đương với việc phương thức chỉ chứa các câu lệnh đơn giản và câu lệnh rẽ nhánh. Các câu lệnh lặp trong phương thức đã được chuyển về dạng các câu lệnh rẽ nhánh if-else lồng nhau.

### 2.4. Đánh chỉ số (indexing) cho các tham biến và xây dựng biểu thức metaSMT

#### 2.4.1. Đánh chỉ số cho các biến

Trong SMT Solver không có khái niệm biến mà chỉ có các hằng số. Trong khi giá trị của các biến trong phương thức thay đổi liên tục. Vì vậy để thể hiện sự thay đổi của các biến số trong trong phương thức ta phải đánh chỉ số tương ứng với giá trị của biến trong từng thời điểm cụ thể của biến trong trong quá trình thực thi chương trình. Khi bắt đầu môt phương thức nào đó, các tham số của hàm sẽ có chỉ số ban đầu là 1. Đối với các biến cục bộ trong thân phương thức, khi được khai báo mà chưa được khởi tạo sẽ có chỉ số là -1. Sau khi được gán giá trị thì biến sẽ có chỉ số là 0. Mỗi lần giá trị của biến được thay đổi chỉ số của biến sẽ tăng lên biểu để biểu diễn giá trị tương ứng của biến đó.

Minh họa bằng một đoạn mã nguồn:

|  |  |
| --- | --- |
| Mã nguồn | Biến : Chỉ số |
| 1 public void foo(int n) {  2 int a;  3 int b = 0;  4 a = n \* n;  5 b = a;  … …  } | (n : 0)  (n : 0), (a : -1)  (n : 0), (a : -1), (b : 0)  (n : 0), (a : 0), (b : 0)  (n : 0), (a : 0), (b : 1) |

Trong đoạn mã nguồn trên, ở dòng 1, n là tham số của phương thức nên có chỉ số là 0. Ở dòng thứ 2, biến a được khai báo nhưng chưa được khởi tạo giá trị nên chỉ sô là -1. Dòng tiếp theo b được khởi tạo và gán giá trị là 0 nên có chỉ số là. Dòng thức 4 biến a được gán giá trị là n \* n lên chỉ số tăng lênh 1 thành 0. Và ở dòng thứ 5 giá trị của biến b được gán bằng giá trị của a nên chỉ sô sẽ trở thành 1.

Để thuận tiện cho bước đánh chỉ mục này, chúng tôi sử dụng các đối tượng thuộc lớp Variable chứa thông tin của các biến trong phương thức bao gồm tên biến, kiểu giá trị của biên và chỉ số của biến. Đối với các biến tham số thì chỉ số mặc định là 0, còn các biến cục bộ trong phương thức có giá trị là -1. Bằng cách duyệt đồ thị theo chiều sâu và coi cả đoạn đồ thì thì nút CoditionNode cho tới nút EndCondition như một ta sẽ duyệt lần lượt từng câu lệnh theo thứ tự thực hiện trong trình. Mỗi khi duyệt tới một nút nào đó, ta sẽ lây biểu thức toán học trong nút đo để đánh chỉ số. Thuật toán đánh chỉ số cho các biểu thức toán học được trình bày như sau:

**Thuật toán** index(exp, listVar)

**Đầu vào:** Biểu thức toán học exp được lưu dưới dạng cây và danh sách listVar chứacác biến kiểu Variable chứa thông tin về các trong phương thức

**Đầu ra:** Biểu thức toán học sau khi đánh chỉ số các biến

**1** **if**  (exp là phép câu lệnh gán)

**2** assignment := biểu thức gán

**3** index(assignment, listVar) // để đánh chỉ số cho assignment

**4**  assigned := tên biến được gán giá trị ở vế trái của câu lệnh gán

**5** var := biến kiểu Variable trong listVar có tên là assigned

**6** Tăng chỉ sô của var lên 1 đơn vị

**7** Gán giá trị mới của tên biến được gán trong câu lệnh gán là assigned\_i

// i là chỉ số của biến sau khi tăng lên 1

**8** **else if** (exp là phép toán hai ngôi)

**9** left := toán tử bên trái của phép toán

**10** right : = toán tử bên phải của phép toán

**11** index(left, listVar) // để đánh chỉ số cho toán tử bến trái

**12**  index(right, listVar) // để đánh chỉ số cho toán tử bên phải

**13** **else if** (exp là phép khai báo biến cục bộ)

**14**  **if** (exp có chưa biểu thức gán giá trị)

**15** assignment := biểu thức gán

**16** index(assignment, listVar) // để đánh chỉ số cho assignment

**17**  assigned := tên biến được gán giá trị ở vế trái của câu lệnh gán

**18** var := biến kiểu Variable trong listVar có tên là assigned

**19** Tăng chỉ sô của var lên 1 đơn vị

**20** Gán giá trị mới của tên biến được gán trong câu lệnh gán là assigned\_i

// i là chỉ số của biến sau khi tăng lên 1

**21** **else if** (exp chỉ chứa một biến đơn thuần)

**22** name := tên của biến trong exp

**23** var := biến kiểu Variable trong listVar với tên name

**24** Gán giá trị mới của tên biến cho exp là name\_i

// i là chỉ số hiện tại của biến

Một vấn đề đặt ra ở đây là đối với các câu lệnh rẽ nhánh, ở mỗi nhánh thì việc thay đổi giá trị của cùng một biến có thể khác nhau. Điều này làm cho chỉ số của biến sau khi kết thúc khối lệnh rẽ nhánh là khác nhau. Như vậy ở các câu lệnh tiếp theo sẽ rất khó khăn trong việc xác định chỉ số thật sự để tính toán. Để giải quyết vấn đề này, chúng tôi phải tiến hành đồng bộ chỉ số của các biến bị thay đổi giá trị (chỉ số) trong khối lệnh rẽ nhánh. Công việc này được thực hiện ngay sau khi đánh xong chỉ số cho khối lệnh. Chúng tôi sử sẽ thêm vào một loại nút đặc biệt vào trong CFG để biểu diễn cho các biểu thức đồng bộ chỉ số. Việc đồng bộ chỉ số ở được thực hiện ngay sau khi đánh chỉ số ở cả hai nhánh của nút điều kiện.

**Thuật toán:** syncIndex

**Đầu vào:** conditionNode chứa khối lệnh rẽ nhánh sau khi đã đánh chỉ số.

Hai danh sách các biến của chương trình sau khi đánh chỉ số hai nhánh của câu lệnh

**Đầu ra:** Khối lệnh rẽ nhánh sau khi đánh chỉ số cùng với các nút đồng bộ chỉ số tương ứng cho mỗi nhánh

**1** **for** (với mỗi biến v có trong chương trình)

**2** i1 := chỉ số của v sau khi thực hiện đánh chỉ số ở nhánh then

**3**  i2 := chỉ số của v sau khi thực hiện đánh chỉ số ở nhánh else

**4** **if** (i1 < i2)

**5** tạo nút đồng bộ tương ứng với biểu thức v\_i2 = v\_i1

**6** thêm nút đồng bộ vào nhánh then

**7** sét chỉ số của biến v sau khi kết thúc nút điều kiện là i2

**8** **else if** (i2 < i1)

**9** tạo nút đồng bộ tương ứng với biểu thức v\_i1 = v\_i2

**10** thêm nút đồng bộ vào nhánh else

**11** sét chỉ số của biến v sau khi kết thúc nút điều kiện là i1

Trong thuật toán trên ta sẽ lấy chỉ số tương ứng với từng biến trong hai nhánh của nút điều kiện. So sánh nếu chỉ sô bên nhánh then nhỏ hơn thì thêm nút đồng bộ với biểu thức v\_i2 = v\_i1 và đăt chỉ sô của biến v sau cả nút điều kiên là i2. Tức là ta tăng chỉ số của biến v bên nhánh then lên bằng giá trị bên nhánh else sau đo gán giá trị mới này bằng với giá trị cuối trong lần thay đổi cuối cuối cùng bên nhánh then. Thực hiện tương tự đối trong trường hợp chỉ số bên nhánh else nhỏ hơn bên nhánh then. Như vậy sau khi kết thúc việc đánh chỉ số và đồng bộ chỉ số cho câu lệnh rẽ nhánh thì chỉ số của các biến trong phương thức sẽ là giá trị lớn nhất mà mà hai nhánh đặt được.

#### 2.4.2. Công thức metaSMT

Kết thúc việc đánh chỉ số cho toàn bộ các biến có mặt trong phương thức, ta sẽ đưa ra dạng công thức sơ khai của công thức tổng quát hóa, đó là công thức metaSMT. Đưa ra công thức metaSMT là bước đệm trước khi đưa ra công thức tổng quát hóa cuối cùng của phương thức. Công thức metaSMT cũng gần giống như công thức tổng quát hóa cuối cùng là bao gồm công thức của tất cả các nhánh thi hành có thể có của chương trình, chỉ khác là ở dạng metaSMT biểu thức sẽ gần giống với biểu thức ở câu lệnh thật và dễ quan sát cũng như dễ hiểu hơn so với công thức tổng quát hóa cuối cùng.

Nhiệm vụ chính của metaSMT là giúp chúng ta kiểm soát lỗi và nắm bắt được dạng của công thức cuối cùng, giúp cho việc trừu tượng hóa được minh bạch và dễ kiểm soát hơn.

### **2.5. Công thức tổng quát cho phương thức**

Sau khi unfold CFG và thực hiện đánh chỉ số cho các biến của phương thức, tiếp theo chúng tôi sẽ đưa ra được công thức tổng quát của phương thức. Công thức tổng quát được đưa ra dựa trên metaSMT của phương thức và được biểu diễn dưới dạng ký pháp tiền tố của các biểu thức logic vị từ cấp một (First-order logic).

Mỗi câu lệnh trong chương trình sẽ trở thành một một biểu thức logic vị từ bậc nhất. Công thức logic của cả phương thức sẽ là phép hội giữa các biểu thức logic này. Đối với các câu lệnh gán, các phép tính đại số, các biểu thức so sánh thì biểu thức có dạng tương tự giống như trông mã nguồn. Điều khác biệt ở đây là ta viết các biến số kèm theo chỉ số tương ứng của chúng trong quá trình thực thi chương trình.

Đối với các câu lệnh rẽ nhánh

**if** (điều\_kiện)

khối lệnh then;

**else**

khối lệnh else;

Đoạn mã nguồn có biểu thức logic vị từ tương ứng là:

f = (A => B) ^ (¬ A => C)

trong đó:

* A: là biểu thức logic vị từ của điều\_kiện
* B: là biểu thức logic vị từ của khối lệnh then
* C: là biểu thức logic vị từ của khổi lệnh else

### 2.6. Kết hợp điều kiện của người dùng và đưa vào bộ giải

Mục đích của quá trình sinh ra công thức tổng quát của chương trinh là để kiểm tra một số tính chất của chương trình dựa vào SMT Solver. Việc kiểm tra được thực hiện bằng cách lấy công thức tổng quát sau khi được sinh ra bằng các bước nêu trên sẽ kết hợp với điều kiện do người dùng nhập vào. Sau đó đưa vào bộ giải SMT Solver kiểm tra và đưa ra kết quả cuối cùng cho người dùng. Gọi Fabstaction là biểu diễn trừu tượng của phương thức, Fassertion là điều kiện do người dùng nhập vào. Để kiểm kiểm tra xem điều kiện của người dùng có luôn thỏa mãn hay là không thì chúng tôi thực hiện biểu thức như sau để đưa vào các SMT Solver

F­final = Fabsrtaction ^ ¬­Fassertion

Đưa biểu thức Ffinal này vào trong SMT Solver, phân tích kết quả do SMT Solver trả về và thông báo với người dùng:

- Kết quả do SMT Solver trả về là SAT (satification): tồn tại bộ giá trị của các biến làm thỏa mãn biểu thức Ffinal. Điều này tương đương với việc tồn tại bộ giá trị của các biến làm cho biểu thức ¬­Fassertion thỏa mãn, hay tồn tại bộ giá trị của các biến làm cho biểu thức Fassertion không thỏa mãn. Trong trường hợp này, công cụ sẽ thông báo với người dùng là biểu thức người dùng không phải là luôn đúng với mã nguồn, kèm theo một phản ví dụ (counter example) với các giá trị tương ứng của các biến làm cho biểu thức không thỏa mãn. Dựa vào ví dụ này người dùng có thể xác định ra lỗi có trong mã nguồn.

- Kết quả do SMT Solver trả về là UNSAT (unsatification): không tồn tại bộ giá trị của các biến làm thỏa mãn biểu thức Ffinal. Điều này tương đương với việc không có bộ giá trị nào của các biến làm cho biểu thức ¬­Fassertion thỏa mãn, hay không tồn tại bộ giá trị của các biến làm cho biểu thức Fassertion không thỏa mãn. Trong trường hợp này, công cụ sẽ thông báo với người dùng là biểu thức người dùng luôn đúng.

**3. Minh họa hoạt động của VTSE**

Để minh họa cho việc thực hiện chương trình, chúng tôi xin đưa ra 1 ví dụ cụ thể. Giả sử ta có một phương thức tính giai thừa cho một số nguyên dương n (factorial) được viết bởi ngôn ngữ Java như sau:

public static int factorial(){

int fac = 1;

for(int i = 2; i <= n; i++)

{

fac = fac \* i;

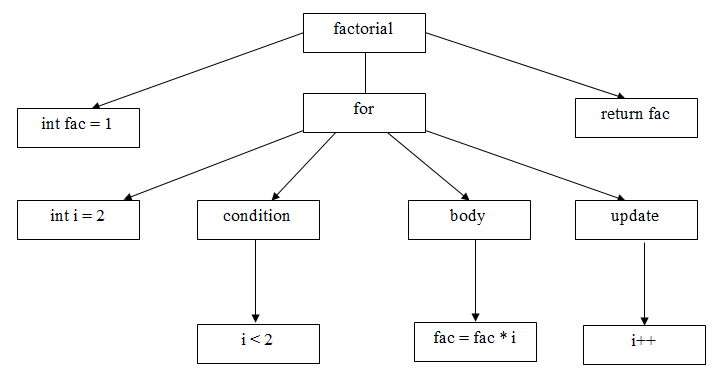
}

return fac;

}

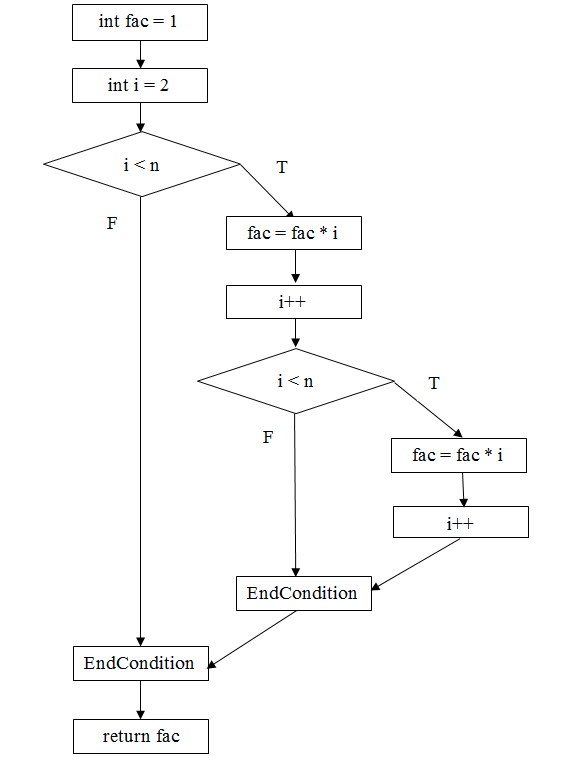
Sau khi nhận được đầu vào là mã nguồn của chương trình, VTSE sẽ tiến hành trừu tượng hóa chương trình bởi các biểu diễn logic vị từ cấp 1 và được thực hiện tuần tự qua các bước sau:

- Bước 1: Xây dựng cú pháp cây trừu tượng. Từ đoạn mã nguồn của phương thức trên ta xây dựng được cây cú pháp trừu tượng như hình 9:



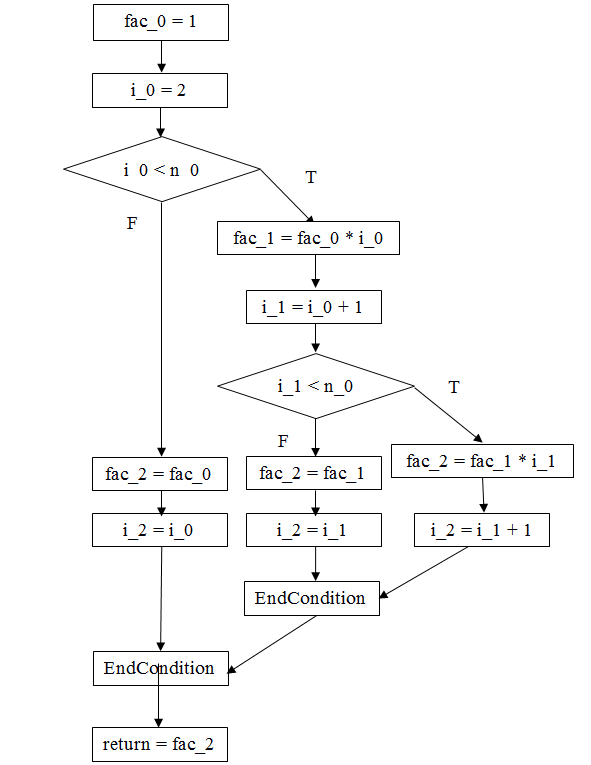
Hình 10: Cây cú pháp trừu tượng minh họa

- Bước 2: Xây dựng đồ thị luồng điều khiển từ cây cú pháp trừu tượng đã xây dựng ở bước 1 và thực hiện unfold CFG. Trong ví dụ này, unfold được thực hiện với 2 lần lặp (Hình 11).



Hình 11: Đồ thị luồng điều khiển minh họa

- Bước 3: Đánh chỉ số (indexing) cho các tham biến và xây dựng biểu thức metaSMT (Hình ).



Hình 12: Đồ thị luồng điều khiển minh họa sau khi đánh chỉ số

- Bước 4: Xây dựng công thức tổng quát cho phương thức:

Ta đưa ra công thức tổng quát cho phương thức:

Fabstraction = (fac\_0 = 2)^ (i\_0 = 2)

^ ((i\_0 < n\_0)=> ((fac\_1 = fac\_0 \* i\_0) ^ (i\_1 = i\_0+1)

^ ((i\_1 < n\_0)=> ((fac\_2 = fac\_1 \* i\_1) ^ (i\_2 = i\_1+1))

^ (¬(i\_1 < n\_0)=> (fac\_2 = fac\_1) ^ (i\_2 = i\_1)))

^ (¬(i\_0 < n\_0) => (fac\_2 = fac\_0) ^ (i\_2 = i\_0))

^ (return = fac\_2)

- Bước 5: Kết hợp điều kiện người dùng và đưa vào bộ giải

Sau khi có được công thức tổng quát cho phương thức, ta đưa công thức vào bộ giải SMT Solver cùng với các bộ điều kiện do người dùng nhập vào. Từ đó dựa vào kết quả trả về, ta đưa ra kết luận về bộ điều kiện đó.

# **IV. Thực nghiệm**

## **Xây dựng công cụ VTSE**

Công cụ kiểm chứng tính chất chương trình dựa trên thực thi tượng trưng được xây dựng theo quy trình và phương pháp nêu trong chương III. Công cụ sử được viết bằng ngôn ngữ Java. Chúng tôi sử dụng bộ giải rằng buộc Z3 [3] vì đây là một trong những solver mạnh nhất hiện nay để giải các biểu diễn logic vị từ cấp một. Bên cạnh đó, công cụ sử dụng thư viện mã nguồn mở Spoon [9] để phân tích mã nguồn và tạo ra cây cú pháp trừu tượng.

Như đã trình bày trong chương III, trình tự hoạt động của VTSE sẽ được thực hiện như sau:

* Nhận đầu vào của công cụ là một tập tin mã nguồn viết bằng ngôn ngữ Java.
* Tạo cây cú pháp trừu tượng AST sử dụng thư viện Spoon.
* Xây dựng CFG từ AST đã sinh ra.
* Với những đoạn mã nguồn chứa vòng lặp, VTSE cho phép người dùng tham số hóa vòng lặp để thực hiện vòng lặp với một số lần hữu hạn xác định (bounded loop).
* Tạo dạng tiền công thức metaSMT từ đồ thị luồng điều khiển đã thực hiện unfold CFG, ở bước này ta tiến hành duyệt đồ thị luồng điều khiển và tiến hành đánh chỉ số, đồng bộ chỉ số các biến.
* Đưa ra biểu diễn trừu tượng hóa cuối cùng của phương thức từ metaSMT
* Kết hợp với điều kiện do người dùng nhập vào để đưa vào các SMT Solver
* Kết quả do bộ giải đưa ra được phân tích và đưa ra cho người dùng thông tin về việc thỏa mãn hay không thỏa mãn của phương thức đối với điều kiện kiểm tra, cùng với đó là ví dụ minh họa.

## **Một số ứng dụng thực nghiệm**

Ứng dụng của chương trình có phạm vi rất lớn trong các công việc xử lý nặng về tính toán, đặc biệt là các chương trình có nhiều câu lệnh điều khiển rẽ nhánh và biến đổi phức tạp.

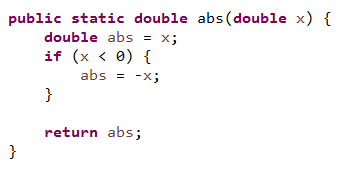
Ta sẽ kết hợp điều kiện của người dùng muốn kiểm tra với công thức tổng quát của chươn trình để tạo thành một biểu thức trừu tượng hóa cuối cùng và đưa vào các SMT Solver để đưa ra kết luận.

Phần này sẽ đề cập đến 4 ứng dụng chính của VTSE đó là kiểm tra giá trị trả về của một hàm, kiểm tra giá trị trả về một phương thức có luôn thuộc một khoảng cho trước, kiểm tra các hậu điều kiện (Post-condition) thỏa mãn hay không thỏa mãn với các tiền điều kiện (Pre-condition) do người dùng nhập vào, kiểm tra tính chất của các điều kiện phức hợp do người dùng nhập vào.

### Kiểm tra giá trị trả về của một hàm

VTSE sẽ trừu tượng hóa phương thức thành biểu diễn cuối cùng bởi các biểu diễn logic vị từ cấp một, cả một phương thức phức tạp được tựu trung lại bằng một biểu diễn cuối cùng, mang một giá trị cụ thể. Giá trị này được tạo nên bởi các biến có mặt trong phương thức, do đó ta có thể kiểm tra được giá trị này có thỏa mãn một điều kiện nào đó hay không.

* Xét đoạn mã nguồn phương thức tính giá trị tuyệt đối của một số



Hình 13: Mã nguồn phương thức tính giá trị tuyệt đối

* Điều kiện người dùng muốn kiểm tra (User’s Assertion) liên quan đến giá trị trả về của phương thức được biểu diễn bởi ràng buộc sau:

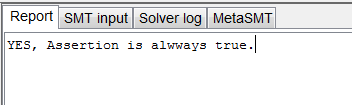
return >= 0

Tức là người dùng muốn kiểm tra xem giá trị trả về có luôn thỏa mãn điều kiện này hay không ?

Như vậy, đầu vào của VTSE bao gồm hai thành phần:

* Đầu vào thứ nhất là mã nguồn của phương thức (hình 13)
* Đầu vào thứ hai là điều kiện do người dùng nhập vào mong muốn để kiểm tra: return >= 0

Sau khi kết thúc việc chạy chương trình, kết quả VTSE đưa ra như sau:



Hình 14: Kết quả kiểm tra giá trị trả về của phương thức

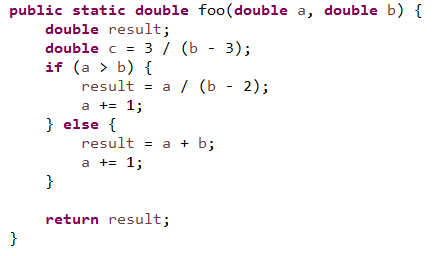
Như vậy công cụ đưa ra thông báo là điều kiện được dùng để kiểm tra không phải khi nào cũng chính xác. Điêu kiện này là hoàn toàn đúng với tính chất của giá trị tuyệt đối.

### Kiểm tra giá trị trả về của phương thức có luôn thuộc một khoảng cho trước hay không

Việc biết được giá trị trả về của một phương thức có luôn nằm trong một khoảng cho trước nào đó hay không sẽ giúp ích rất nhiều trong kiểm định giá trị của phương thức đó. Nó đóng một vai trò quan trọng trong quá trình kiểm chứng và sinh ra các ca kiểm thử của chương trình.

Ở đây chúng tôi dùng đoạn mã nguồn có chứa câu lệnh if và nhiều lệnh tính toán để kiểm tra miền giá trị trả về của phương thức.

* Xét đoạn mã nguồn của phương thức sau đây:



Hình 15: Mã nguồn phương thức dùng để kiểm tra

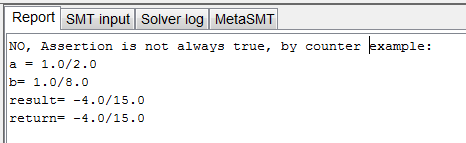
Giả sử người dùng muốn kiểm tra giá trị trả về luôn thuộc đoạn trong đoạn [0, 1], được biểu diễn bằng ràng buộc sau:

(return >= 0) ^ (return <= 1)

Đầu vào của VTSE gồm 2 thành phần:

* Đầu vào thứ nhất là mã nguồn của chương trình (hình 15)
* Điều kiện người dùng muốn kiểm tra (User’s assertion) được biểu diễn bởi ràng buộc: (return >= 0) ^ (return <= 1)

Sau khi chạy chương trình, kết quả do VTSE đưa ra là:



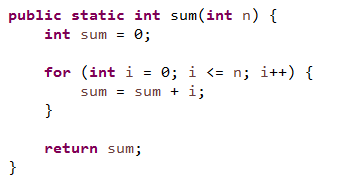
Hình 16: Kết quả kiểm tra giá trị trả về có thuộc khoảng giá trị không

Như vậy với giá trị của a = 1/2 và b = 1/8 thì giá trị trả về sẽ bằng -4/15 nằm ngoài đoạn [0; 1].

### Kiểm tra các hậu điều kiện (Post-condition) thỏa mãn hay không thỏa mãn với các tiền điều kiện (Pre-condition) do người dùng nhập vào.

VTSE sẽ cho người dùng nhập vào các hậu điều kiện và tiền điều kiện để kiểm tra tính thỏa mãn của các hậu điều kiện dựa trên các tiền điều kiện do người dùng nhập vào.

* Xét đoạn phương thức tính tổng của một từ 1 đến n



Hình 17: Mã nguồn phương thức tính tổng từ 1 đến n

Trong đoạn mã nguồn này có chứa vòng lặp. Chúng tôi chọn 6 lần lặp để cho công cụ thực hiện unfold CFG.

* Người dùng muốn kiểm tra hậu điều kiện (post-condition) có luôn thỏa mãn với tiền điều kiện nào đó hay không thì trong trường hợp này, điều kiện do người dùng nhập vào (User’s assertion) được biểu diễn bởi 2 điều kiện là

+ Biểu thức tiền điều kiện:

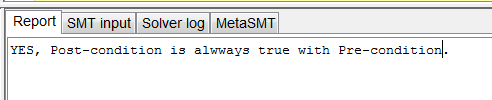
(n >= 0) ^ (n < 6)

Ở đây, điều kiện (n >= 0) để công cụ chỉ kiểm tra với đầu vào là số nguyên không âm. Còn điều kiện (n < 6) là điều kiện bổ sung quy định giá trị của biến n không vượt quá số lần unfold CFG

+ Biểu thức hậu điều kiện:

return = n \* (n + 1) / 2

Sau khi chạy chương trình, VTSE đưa ra kết quả là:



Hình 18: Kết quả kiểm tra hậu điều kiện có thỏa mãn tiền điều kiện hay không.

Công cụ đưa ra thông báo hậu điều kiện luôn đúng với tiền điều kiện đưa ra. Công cụ đã kiểm tra được tính chất của tổng dãy số từ 1 đến n. Hạn chế ở đây là phải đưa thêm điều kiện để giới hạn sô vòng lặp bằng biểu thức (n < 6).

### Kiểm tra tính chất của các điều kiện phức hợp do người dùng nhập vào

Đây cũng chính là ứng dụng tổng quát nhất mà công cụ VTSE muốn hướng đến. Chương trình có thể kiểm tra sự thỏa mãn của bất kì bộ giả định nào mà người dùng muốn kiểm tra trên chương trình của mình.

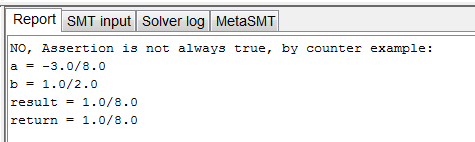
Xét đoạn mã nguồn của phương thức foo trong phần kiểm tra khoảng trả về của phương thưc.

Điều kiện do người dùng nhập vào (User’s Assertion):

return \* return > (a + b) / 2

tức là người dùng muốn kiểm tra xem giá trị trả về có luôn thỏa mãn điều kiện này hay không.

Sau khi chạy chương trình, kết quả VTSE đưa ra là:



Hình 19: Kết quả kiểm tra tính chất của các điều kiện phức hợp

Như vậy với giá trị của a là -3/8 và b là 1/2 thì giá trị trả về của phương thức là 1/8, điều này không thỏa mãn điều kiện kiểm tra.

# Tổng kết

Trong bài báo cáo này, chúng tôi giới thiệu một phương pháp mới để kiểm chứng một số tính chất của chương trình dựa trên thực thi tượng trưng đồng thời đưa ra công cụ VTSE thực hiện kiểm chứng chương trình dựa trên phương pháp này.

Phương pháp kiểm chứng chương trình dựa trên kĩ thuật thực thi tượng trưng là một kĩ thuật quan trọng bởi phương pháp này giúp chúng ta tránh khỏi việc bùng nổ trạng thái gặp phải khi áp dụng phương pháp kiểm chứng mô hình.

Các ứng dụng của VTSE được trình bày trong bản báo cáo đó là kiểm tra giá trị trả về của một phương thức, kiểm tra giá trị trả về của phương thức có luôn thuộc một khoảng xác định, kiểm tra các hậu điều kiện có luôn thỏa mãn với một tiền điều kiện cho trước và kiểm tra tính chất của các bộ phức hợp do người dùng nhập vào.

Ngoài các ứng dụng được nêu trong bài báo cáo này, VTSE có thể mở rộng khả năng ứng dụng của mình lên các chương trình có phạm vi rộng hơn bằng cách tối thiểu hóa các ràng buộc của chương trình. VTSE đang tiếp tục phát triển để có thể xử lý được các câu lệnh nhảy trong chương trình như là break hay continue. Và VTSE có thể phát hiện ra các mã lệnh chết trong chương trình (dead code) nhờ vào việc xác định các biểu diễn logic không liên quan trong chương trình thực thi.

Tuy nhiên, VTSE vẫn còn hạn chế với vấn đề vòng lặp, chưa thực hiện được một cách chính xác nhất khi trừu tượng hóa vòng lặp bởi các biểu diễn logic. Trong tương lai, VTSE sẽ tiếp tục cải thiện về vấn đề vòng lặp này.

# Tài liệu tham khảo

1. Corina Pasareanu IOS Press (2013) , *Symbolic Execution and Software Testing*
2. SMT-LIB The Satisfiability Modulo Theories Library

http://smtlib.cs.uiowa.edu/

1. Z3 – SMT Solver

https://z3.codeplex.com/

1. To Van Khanh and [Mizuhito Ogawa](http://www.jaist.ac.jp/~mizuhito), [raSAT: SMT for polynomial inequality](http://www.jaist.ac.jp/~mizuhito/papers/report/JAIST-IS-RR-2013-003.pdf)ra

http://www.jaist.ac.jp/~s1310007/raSAT/

1. CVC4 – The SMT Solver

http://cvc4.cs.stanford.edu/web/

1. Cristian Cadar, Daniel Dunbar, Dawson Engler (2008), “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”, Stanford University Cascade, *Wei Wang, Clark Barrett, and Thomas Wies*
2. Abstract Syntax Tree, page 4
3. Spoon - Source Code Analysis and Transformation for Java

http://spoon.gforge.inria.fr/

1. James C. King (1976) , *Symbolic Execution and Program Testing*
2. Development and Evaluation of LAV: an SMT-Based Error Finding Platform
3. Cristian Cadar, Koushik Sen (2013), *Symbolic Execution for Software Testing: Three Decades Later*
4. Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi, *A Survey of Symbolic Execution Techniques,* Sapienza University of Rome