

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

Nguyễn Đức Thuận

**KỸ THUẬT THỰC THI TƯỢNG TRƯNG
VÀ ỨNG DỤNG TRONG KIỂM CHỨNG MỘT SỐ
TÍNH CHẤT CỦA CHƯƠNG TRÌNH**

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

HÀ NỘI - 2018

**ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

Nguyễn Đức Thuận

**KỸ THUẬT THỰC THI TƯỢNG TRƯNG
VÀ ỨNG DỤNG TRONG KIỂM CHỨNG MỘT SỐ
TÍNH CHẤT CỦA CHƯƠNG TRÌNH**

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành: Công nghệ thông tin

Cán bộ hướng dẫn: TS. Tô Văn Khánh

HÀ NỘI - 2018

**VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY**

Nguyen Duc Thuan

**SYMBOLIC EXECUTION AND APPLICATIONS IN
SOFTWARE VERIFICATION**

Major: Information Technology

Supervisor: Dr. To Van Khanh

HA NOI - 2018

Tóm tắt

Trong sản xuất phần mềm, việc đảm bảo chất lượng phần mềm đóng vai trò rất quan trọng, chính vì vậy các công cụ kiểm chứng chương trình ngày càng được phát triển rộng rãi. VTSE (*Verification Tool based on Symbolic Execution*) là một công cụ cho phép người sử dụng kiểm chứng một số tính chất của mã nguồn chương trình C/C++ dựa trên kỹ thuật thực thi tượng trưng, một kỹ thuật có nhiều ưu điểm vượt trội so với các kỹ thuật truyền thống.

VTSE có đầu vào gồm hai thành phần, thứ nhất là mã nguồn của chương trình, thứ hai là điều kiện do người dùng nhập vào. Đầu ra của VTSE là một báo cáo cho biết các điều kiện của người dùng có luôn thoả mãn trên một điều kiện cho trước hay không hoặc chương trình có chạy vào một số câu lệnh cụ thể trong chương trình hay không.

Dữ liệu thực nghiệm được sử dụng cho VTSE là hai bộ thực nghiệm Floats-cdfpl [1] (bộ dữ liệu thực nghiệm được sử dụng trong cuộc thi SV-COMP 2017 [1]) và Kratos [2] (các bài toán có số lượng dòng lệnh rất lớn). Kết quả thực nghiệm đối với từng bộ dữ liệu thực nghiệm như sau:

- Kết quả dữ liệu thực nghiệm Floats-cdfpl cho thấy VTSE thực hiện được 31 trên tổng số 40 bài toán, CBMC và 2LS đều giải được 40 trên tổng số 40 bài toán. Tuy số bài toán VTSE giải được ít hơn nhưng thời gian trung bình để thực hiện lại tốt hơn hai công cụ còn lại từ 450 đến 1000 lần.

- Kết quả dữ liệu thực nghiệm Kratos cho thấy VTSE hoàn toàn có khả năng có giải được các bài toán có số lượng dòng lệnh rất lớn (lên đến 2000 dòng lệnh). Kết quả thực nghiệm của VTSE là tương đối khả quan, ưu thế về tốc độ mở ra một hướng nghiên cứu mới trong công cuộc kiểm chứng chương trình. So với các công cụ khác tốc độ xử lý của VTSE là tốt hơn. Tuy rằng vẫn còn hạn chế là chưa giải được số một số câu lệnh phức tạp, nhưng trong tương lai hạn chế này có thể được khắc phục trong những phiên bản tiếp theo.

Abstract

In software process, quality assurance plays an important role, so software verification tools are increasingly being developed. VTSE (*Verification Tool based on Symbolic Execution*) is a software verification tool which allows users to verify some properties of C/C++ program based on symbolic execution technique, this is a technique that has many advantages over traditional techniques.

VTSE has two inputs, a source code of program and user's assertion. Output of VTSE is a report which demonstrates whether user's assertion are always satisfied with a pre-condition or not or whether the program execute into some specific commands in the program or not.

The experimental data for VTSE are the two sets of Floats-cdfpl (experimental data set is used in the competition of SV-COMP 2017) and Kratos (many lines of code problems). The result for each set are as follows:

- Result of Floats-cdfpl shows that VTSE can solve out of 40 problems, CBMC and 2LS solved 40 out of 40 problems. Although the number of problems VTSE can solve is less than CBMC and 2LS, the average time to solve better than others is about 450 to 1000 times.

- Result of Kratos shows that VTSE is able to solve problems which have many lines of code (up to 2000 lines of code). Result of VTSE is relatively positive, it's speed began a new research about software verification. With others, the speed of VTSE is better. Although there are still limitations of many of complex statements, this limitations can be resolved in next versions of VTSE.

LỜI CẢM ƠN

Trước tiên, em xin bày tỏ lòng biết ơn chân thành và sâu sắc nhất tới Thầy giáo, Tiến sĩ Tô Văn Khánh, người đã tận tình chỉ bảo, hướng dẫn, động viên và giúp đỡ em trong suốt quá trình thực hiện đề tài.

Với lòng biết ơn sâu sắc nhất, em xin gửi đến quý thầy cô giáo trong Khoa Công nghệ thông tin nói riêng và trong trường Đại học Công nghệ - Đại học Quốc Gia Hà Nội nói chung, đã cùng với tri thức và tâm huyết của mình để truyền đạt vốn kiến thức quý báu cho chúng em trong suốt thời gian học tập tại trường. Và đặc biệt, trong học kỳ này, Khoa đã tổ chức cho chúng em được tiếp cận với môn học mà theo em là rất hữu ích đối với sinh viên ngành Công nghệ thông tin cũng như tất cả các sinh viên thuộc các ngành khác của trường.

Con xin tỏ lòng biết ơn vô hạn đến bố mẹ vì công ơn sinh thành dưỡng dục, luôn động viên, chăm lo và dành trọn tình yêu thương cho con, động viên, ủng hộ con trên con đường học tập.

Cuối cùng, em xin gửi lời cảm ơn tới các anh chị và các bạn, đặc biệt là các thành viên lớp K59CLC đã ủng hộ, giúp đỡ trong suốt quá trình em học tập trên giảng đường đại học và thực hiện đề tài.

Em xin chân thành cảm ơn!

Hà Nội, ngày 26 tháng 04 năm 2018

Sinh viên

Nguyễn Đức Thuần

LỜI CAM ĐOAN

Em xin cam đoan đề tài “Kỹ thuật thực thi tượng trưng và ứng dụng trong kiểm chứng một số tính chất của chương trình” này là em và chưa từng được nộp như một báo cáo khoá luận tốt nghiệp nào tại trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội hoặc bất kỳ trường đại học nào khác. Những gì em viết ra không sao chép từ các tài liệu, không sử dụng các kết quả của người khác mà không trích dẫn cụ thể.

Em xin cam đoan công cụ VTSE - Công cụ kiểm chứng một số tính chất của mã nguồn chương trình C là do em phát triển, không sao chép mã nguồn của người khác. Nếu sai em hoàn toàn chịu trách nhiệm theo quy định của trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội.

Hà Nội, ngày 26 tháng 04 năm 2018

Tác giả

Nguyễn Đức Thuần

MỤC LỤC

<i>Mở đầu</i>	1
<i>CHƯƠNG 1. KIỂM CHỨNG CHƯƠNG TRÌNH VÀ THỰC THI TƯỢNG TRUNG</i>	2
1.1. Kiểm chứng chương trình	2
1.1.1. Tổng quan về kiểm chứng chương trình.....	2
1.1.2. Kiểm chứng mô hình	2
1.2. Kỹ thuật thực thi tượng trung	4
1.2.1. Tổng quan về kỹ thuật thực thi tượng trung	4
1.2.2. Ứng dụng của kỹ thuật thực thi tượng trung	5
1.3. Satisfiability Modulo Theories	7
1.3.1. Logic mệnh đề	7
1.3.2. Satisfiability Modulo Theories	7
1.3.3. Ứng dụng bộ giải SMT	7
<i>CHƯƠNG 2. CÔNG CỤ KIỂM CHỨNG CHƯƠNG TRÌNH VTSE</i>	9
2.1. Kiến trúc của VTSE	9
2.2. Trừu tượng hoá chương trình dựa trên thực thi tượng trung	10
2.2.1. Cây cú pháp trừu tượng (Abstract Syntax Tree - AST).....	11
2.2.2. Đồ thị luồng điều khiển (Control Flow Graph – CFG)	11
2.2.3. Xử lý vòng lặp trong CFG	14
2.2.4. Biểu diễn chương trình dưới dạng metaSMT	18
2.2.5. Biểu diễn chương trình dưới dạng logic vị từ.....	18
2.2.6. Kiểm tra các tính chất của chương trình dựa trên bộ giải SMT	19
2.3. Minh hoạ hoạt động của VTSE	19
2.4. Một số ứng dụng	23
2.4.1. Kiểm tra giá trị trả về của một hàm	23
2.4.2. Kiểm tra hậu điều kiện từ tiền điều kiện.....	24
2.4.3. Kiểm tra tính chất do người dùng đưa vào	25
2.4.4. Kiểm tra chương trình có chạy vào các câu lệnh xác định.....	26
2.4.5. Kiểm tra khối lệnh không bao giờ được thực thi.....	27
<i>CHƯƠNG 3: CÁC NGHIÊN CỨU VÀ ỨNG DỤNG LIÊN QUAN</i>	29
3.1. KLEE	29
3.1.1. Tổng quan	29
3.1.2. Mô hình hoạt động	29
3.1.3. Chức năng	30
3.1.4. Hạn chế	30
3.2. Cascade	31
3.2.1. Tổng quan	31
3.2.2. Thiết kế hệ thống	31
3.2.3. Điểm mạnh và điểm yếu	32
3.3. CBMC	32

3.3.1. Tổng quan	32
3.3.2. Kiến trúc	33
3.3.3. Điểm mạnh và điểm yếu	34
3.3.4. So sánh với VTSE.....	35
3.4. 2LS.....	35
3.4.1. Tổng quan	35
3.4.2. Kiến trúc	35
3.4.3. Điểm mạnh và điểm yếu	36
3.4.4. So sánh với VTSE.....	37
3.5. Ceagle.....	37
3.5.1. Tổng quan	37
3.5.2. Kiến trúc	37
3.5.3. Thiết kế của Ceagle	38
3.5.4. Điểm mạnh và điểm yếu	39
CHƯƠNG 4. THỰC NGHIỆM.....	40
4.1. Xây dựng công cụ VTSE	40
4.2. Kết quả thực nghiệm	40
4.2.1. Điều kiện thực nghiệm.....	40
4.2.2. Kết quả thực nghiệm.....	41
4.2.3. Hướng phát triển	45
KẾT LUẬN.....	46
Tài liệu tham khảo.....	47

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

MỤC LỤC HÌNH ẢNH

Hình 1: Cây thực thi tượng trưng của hàm <code>test()</code> trong ví dụ 1	6
Hình 2: Kiến trúc của VTSE	9
Hình 3: Khối điều khiển của CFG.....	13
Hình 4: Khối lệnh <code>goto</code> và lời gọi hàm	14
Hình 5: Xử lý vòng lặp trong CFG.....	14
Hình 6: Xử lý lệnh <code>goto</code>	15
Hình 7: Cây cú pháp trừu tượng AST cho mã nguồn ví dụ 2	20
Hình 8: Đồ thị luồng điều khiển cho ví dụ 2.....	21
Hình 9: Đồ thị luồng điều khiển sau khi xử lý vòng lặp và đồng bộ chỉ số.....	22
Hình 10: Kết quả chạy VTSE cho chương trình trong ví dụ 3	24
Hình 11: Kết quả chạy VTSE cho chương trình trong ví dụ 4	25
Hình 12: Kết quả chạy VTSE cho chương trình trong ví dụ 5	26
Hình 13: Kết quả chạy VTSE cho chương trình trong ví dụ 6	27
Hình 14: Kết quả chạy VTSE cho chương trình trong ví dụ 7	28
Hình 15: Sơ đồ hoạt động của KLEE	29
Hình 16: Kiến trúc hệ thống của Cascade.....	31
Hình 17: Ví dụ tập tin điều khiển của công cụ Cascade	32
Hình 18: Kiến trúc của CBMC	33
Hình 19: Kiến trúc của 2LS.....	35
Hình 20: Kiến trúc của Ceagle.....	38

MỤC LỤC BẢNG

Bảng 1. So sánh VTSE với các công cụ khác dựa trên bộ thực nghiệm <code>Floats-cdfpl</code> ...	43
Bảng 2. So sánh VTSE với các công cụ khác dựa trên bộ thực nghiệm <code>Kratos</code>	44

Mở đầu

Ngày nay, với việc thế giới đang bước vào cách mạng công nghiệp 4.0, máy tính và phần mềm máy tính ngày càng đóng vai trò quan trọng hơn trong mọi lĩnh vực của đời sống xã hội từ kinh tế, giao thông, vũ trụ cho đến nông nghiệp, y tế và giáo dục. Bên cạnh sự phát triển nhanh chóng của công nghiệp phần mềm, cùng với đó là yêu cầu của khách hàng đối với các sản phẩm phần mềm ngày càng tăng cao. Do đó, 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 đòi hỏi sự chính xác tuyệt đối như quân sự, y tế và hàng không vũ trụ.

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. Kiểm thử phần mềm là quá trình tạo ra và thực thi các ca kiểm thử để tìm ra lỗi trong chương trình nhằm đảm bảo độ tin cậy cao của một phần mềm. 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 người ta thường áp dụng các phương pháp kiểm chứng chương trình. Tuy nhiên 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.

Báo cáo khoá luận này sẽ đề xuất ra phương pháp kiểm chứng một số tính chất của chương trình dựa trên kỹ thuật thực thi tượng trưng. Phương pháp này đề xuất việc trừu tượng hóa một đoạn mã nguồn thành các biểu thức logic vị từ cấp một (First-order Logic) và kết hợp các điều kiện người dùng để đưa vào các bộ giải SMT.

Bài khoá luận cũng đề xuất kiến trúc và xây dựng công cụ mới là VTSE (Verification Tool based on Symbolic Execution), công cụ kiểm chứng một số tính chất của chương trình dựa trên kỹ thuật thực thi tượng trưng và bộ giải SMT. VTSE có các ứng dụng trong việc kiểm tra một số tính chất liên quan đến giá trị trả về của phương thức, các điều kiện tại vị trí nhất định trong chương trình hay kiểm tra đoạn lệnh không được thi hành.

Cuối cùng là kết quả thực nghiệm của VTSE trên hai bộ bài toán tiêu biểu là Floats-cdfpl và Kratos được sử dụng trong cuộc thi SV-COMP 2017 và nhiều bài báo nghiên cứu khoa học khác. Đồng thời thực hiện so sánh với một số công cụ kiểm chứng khác.

CHƯƠNG 1. KIỂM CHỨNG CHƯƠNG TRÌNH VÀ THỰC THI TƯỢNG TRUNG

Chương này sẽ tập trung trình bày tổng quan về kiểm chứng chương trình và kỹ thuật thực thi tượng trung trong kiểm chứng chương trình. Tiếp theo, kỹ thuật kiểm chứng mô hình và những ưu điểm cũng như hạn chế của nó cũng sẽ được trình bày trong chương này. Vào phần cuối của chương, báo cáo khóa luận sẽ trình bày tổng quan về Satisfiability Modulo Theories (SMT) và ứng dụng trong bộ giải SMT.

1.1. Kiểm chứng chương trình

1.1.1. Tổng quan về kiểm chứng chương trình

Trong quy trình sản xuất phần mềm, kiểm chứng [3] là một trong những giai đoạn quan trọng nhất. Trong khi các kỹ thuật kiểm thử chỉ có thể phát hiện khiếm khuyết của hệ thống mà không khẳng định được chương trình đó không có lỗi, thì các kỹ thuật kiểm chứng có thể giải quyết được vấn đề này.

Kiểm chứng là hoạt động giúp đánh giá một hệ thống phần mềm bằng cách xác định sản phẩm ở các pha phát triển có thỏa mãn các đặc tả yêu cầu phần mềm hoặc các thuộc tính của chương trình có thỏa mãn hay không? Sản phẩm ở đây có thể hiểu là sản phẩm trung gian như: Đặc tả yêu cầu, thiết kế chi tiết, mã nguồn. Những hoạt động kiểm tra tính đúng đắn của các pha phát triển được gọi là hoạt động kiểm chứng. Kiểm chứng trả lời cho câu hỏi “Hệ thống được xây dựng có đúng với đặc tả”. Mục đích của hoạt động kiểm chứng là xác thực xem đã xây dựng sản phẩm đúng cách không.

Về cơ bản, kiểm chứng hệ thống là kỹ thuật được áp dụng để xác minh tính đúng đắn của một thiết kế, một sản phẩm. Đó là những đặc tả quy định các hành vi đã thực hiện của hệ thống và những gì hệ thống không thực hiện từ đó tạo cơ sở cho hoạt động kiểm chứng. Hệ thống được coi là đúng nếu nó thỏa mãn tất cả các thuộc tính thu được từ các đặc tả cụ thể của nó.

1.1.2. Kiểm chứng mô hình

Trong thiết kế phần mềm và phần cứng của các hệ thống phức tạp, nhiều thời gian và công sức dành cho việc kiểm định hơn là việc xây dựng hệ thống. Các kỹ thuật được tìm kiếm để giảm thiểu những nỗ lực kiểm định trong khi gia tăng độ bao phủ. Các phương pháp hình thức đưa ra một triển vọng lớn để có thể tích hợp kiểm chứng trong quá trình thiết kế, cung cấp các kỹ thuật kiểm chứng hiệu quả hơn và giảm bớt thời gian kiểm chứng.

Kỹ thuật kiểm chứng mô hình [4] được dựa trên sự mô tả mô hình của các hành vi có thể của hệ thống qua những công thức toán học rõ ràng và chính xác. Kiểm chứng mô hình là kỹ thuật phát hiện lỗi chương trình theo cách thức vét cạn tất cả các trạng thái có thể có của hệ thống. Các công cụ kiểm chứng phần mềm mà sử dụng kỹ thuật kiểm chứng mô hình sẽ tiến hành kiểm tra tất cả những kịch bản có thể có của hệ thống. Bằng cách này, nó có thể chỉ ra rằng mô hình hệ thống có thật sự thỏa mãn các thuộc tính nhất định.

Công cụ kiểm chứng áp dụng kỹ thuật kiểm chứng mô hình kiểm tra tất cả các trạng thái có liên quan của hệ thống để kiểm tra xem chúng có thỏa mãn với đặc tính mà đặc tả mong muốn hay không. Nếu trạng thái gặp phải là vi phạm đặc tính đang xem xét, công cụ kiểm chứng sẽ cung cấp một phản ví dụ (*counter example*) cho thấy việc làm thế nào mà mô hình có thể tới được trạng thái không mong muốn. Phản ví dụ cũng mô tả một đường dẫn thực thi có thể dẫn tới từ những trạng thái khởi tạo của hệ thống tới những trạng thái vi phạm thuộc tính đang được kiểm chứng. Với sự giúp đỡ của mô phỏng, người sử dụng có thể xem lại các kịch bản vi phạm để có được những thông tin gỡ lỗi có ích và tích hợp các mô hình phù hợp.

Kiểm chứng mô hình đã có rất nhiều ưu điểm tuy nhiên vẫn không thể tránh khỏi được những khuyết điểm:

- Về ưu điểm kiểm chứng mô hình hướng tiếp cận chung có thể áp dụng với một phạm vi rộng lớn các ứng dụng như các hệ thống nhúng, công nghệ phần mềm và thiết kế phần cứng. Kiểm chứng mô hình hỗ trợ kiểm chứng từng phần, các thuộc tính có thể được kiểm tra riêng lẻ, do đó cho phép tập trung vào những đặc tính quan trọng trước. Kiểm chứng mô hình cung cấp các chuẩn đoán về thông tin trong trường hợp thuộc tính không có giá trị, điều này có ích trong việc hỗ trợ sửa lỗi. Việc sử dụng kiểm chứng mô hình không yêu cầu mức độ cao về sự tương tác của người dùng cũng không cần trình độ cao về chuyên môn. Kiểm chứng mô hình có được sự quan tâm, thu hút tăng lên nhanh chóng bởi ngành công nghiệp, có thể dễ dàng tích hợp trong vòng đời phát triển hiện có và dựa trên một nền tảng toán học vững chắc.

- Các khuyết điểm của kiểm chứng mô hình được thể hiện qua các điểm như sau. Kiểm chứng mô hình chủ yếu thích hợp với các ứng dụng chuyên về điều khiển và chưa phù hợp với các ứng dụng chuyên về cơ sở dữ liệu. Đối với hệ thống vô hạn trạng thái hoặc luận lý về kiểu dữ liệu trừu tượng (các logic không giải được và giải được một nửa), kiểm chứng mô hình tính toán không hiệu quả. Kiểm chứng mô hình chủ yếu kiểm chứng mô hình của hệ thống, và không kiểm chứng hệ thống thực sự (sản phẩm hoặc bản nguyên mẫu), và kết quả thu được tốt là do mô hình hệ thống. Kiểm chứng mô hình

vướng phải vấn đề bùng nổ không gian trạng thái, số lượng các trạng thái cần của hệ thống thực có thể vượt quá bộ nhớ của máy tính. Cách sử dụng của kiểm chứng mô hình đòi hỏi phải có một số chuyên gia trong việc trừu tượng hóa hệ thống để có được mô hình hệ thống nhỏ hơn với ít thuộc tính trạng thái hơn. Kiểm chứng mô hình không đảm bảo mang lại kết quả chính xác, với bất kỳ công cụ nào, công cụ kiểm chứng mô hình có thể chứa các khiếm khuyết về phần mềm.

1.2. Kỹ thuật thực thi tượng trưng

1.2.1. Tổng quan về kỹ thuật thực thi tượng trưng

Thực thi tượng trưng (Symbolic Execution) là một kỹ thuật phân tích chương trình tự động được giới thiệu vào những năm 70, sử dụng trong kiểm thử và kiểm chứng phần mềm để kiểm tra một tính chất nhất định có bị vi phạm bởi chương trình hay không. Ý tưởng chính của thực thi tượng trưng là thực thi chương trình với các giá trị tượng trưng (symbolic value) thay vì các giá trị cụ thể (concrete value) của các tham số đầu vào, kết quả là giá trị đầu ra được tính toán bởi chương trình và được biểu diễn bởi một biểu thức tượng trưng.

Trong thực thi cụ thể (Concrete Execution), một chương trình được chạy trên một đầu vào nhất định và một luồng điều khiển duy nhất được thực thi. Vì vậy, trong hầu hết các trường hợp thực thi cụ thể chỉ có thể đánh giá xấp xỉ tính chất của chương trình.

Ngược lại, trong quá trình thực thi tượng trưng, việc đi theo một nhánh cụ thể nào đó không phụ thuộc vào các giá trị của các tham số đầu vào. Tại tất cả các điểm rẽ nhánh, tất cả các nhánh sẽ được xem xét và kiểm tra nhằm định hướng cho quá trình thực thi tiếp theo của chương trình. Thực thi tượng trưng có thể đồng thời khám phá tất cả các luồng của một chương trình dưới nhiều đầu vào khác nhau. Điều này cho phép đảm bảo chương trình được kiểm tra một cách chính xác. Ý tưởng chủ chốt ở phương pháp này là thay vì giá trị đầu vào là rời rạc, phương pháp cho phép đầu vào là các giá trị tượng trưng.

Cuối cùng, một bộ giải SMT (Satisfiability Modulo Theories solver) được sử dụng để kiểm tra liệu có vi phạm nào trong quá trình thực thi hay không. Ngoài ra, thực thi tượng trưng còn có thể được áp dụng vào tìm kiếm lỗi bằng kiểm tra thời gian chạy hoặc kiểm tra các điều kiện thêm vào.

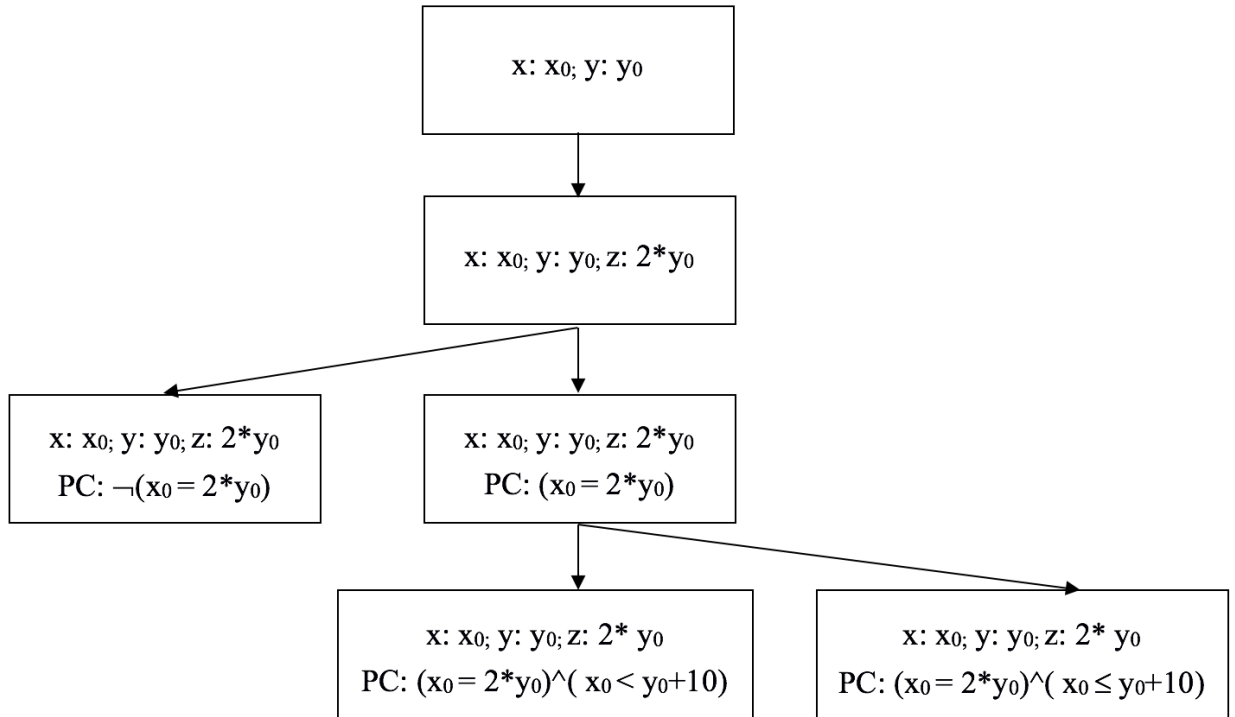
Thực thi tượng trưng có thể được chia làm 2 hướng tiếp cận: động (dynamic-DSE) và tĩnh (static-SSE). Thực thi tượng trưng động được thực hiện bằng cách tạo ra công thức của từng đường đi và kiểm tra từng đường thực thi cụ thể. Trong khi đó, thực

thi tượng trưng tĩnh sinh ra công thức cho cả chương trình nhằm mục đích kiểm tra tính đúng đắn của toàn bộ chương trình. So với DSE, SSE không bị ảnh hưởng bởi bùng nổ đường thực thi. Mọi đường thực thi được chuyển thành công thức cuối cùng và chuyển vào bộ giải. Vì vậy SSE có nhiều lợi thế khi tham gia kiểm chứng các chương trình có số dòng lệnh lớn. Tuy nhiên, SSE gặp nhiều khó khăn hơn DSE trong một số trường hợp, điển hình là xử lý vòng lặp và đệ quy do khó có thể xác định được số vòng lặp hay số lần đệ quy cần thực hiện.

1.2.2. Ứng dụng của kỹ thuật thực thi tượng trưng

Trong kiểm chứng, thực thi tượng trưng thường được sử dụng để sinh ra các giá trị kiểm thử (test input) cho một đường thi hành cụ thể (execution path) của chương trình. Mỗi đường thực thi là một chuỗi các giá trị đúng hoặc sai tương ứng. Tất cả các đường thực thi của chương trình có thể biểu diễn bởi một cấu trúc cây gọi là cây thực thi (execution tree). Mục đích của thực thi tượng trưng là sinh ra tất cả đầu vào có thể cho toàn bộ các đường thực thi. Chúng ta xét ví dụ 1 là chương trình C sau:

```
1    int twice (int v) {
2        return 2*v;
3    }
4    void test (int x, int y) {
5        int z = twice(y)
6        if (z == x) {
7            if (x > y+10) {
8                ERROR;
9            }
10       }
11   }
12
13   /* execute test() with symbolic inputs */
14   int main () {
15       x = sym_input();
16       y = sym_input();
17       test(x, y);
18       return 0;
19   }
```

Hình 1: Cây thực thi tượng trưng của hàm test() trong ví dụ 1

Bắt đầu thực thi tượng trưng hàm test() bằng việc gán giá trị cho các tham số đầu vào x và y lần lượt là x_0 và y_0 . Khởi tạo PC (Path Condition) nhận giá trị là True, tới câu lệnh rẽ nhánh if ($2*y_0 = x_0$) hai nhánh của chương trình tương ứng đều được thực thi với các giá trị tượng trưng là x_0 và y_0 . Tại đây biểu thức điều kiện rẽ nhánh là $2*y_0 = x_0$ và $\neg(2*y_0 = x_0)$ được bổ sung và PC theo hai nhánh khác nhau. Sau khi thực thi câu lệnh if($2*y_0 = x_0$) hàm test() được thực thi theo nhánh mà tồn tại giá bộ giá trị x, y thỏa mãn. Tương tự như trên khi gặp câu lệnh if ($x_0 > y_0 + 10$), PC theo hai nhánh tương ứng sẽ được cập nhật bổ sung là $(2*y_0 = x_0) \wedge (x_0 > y_0 + 10)$ và $(2*y_0 = x_0) \wedge (x_0 \leq y_0 + 10)$. Tại mỗi điểm rẽ nhánh PC sẽ được cập nhật và một thủ tục quyết định bằng bộ xử lý ràng buộc sẽ được sử dụng để xác định xem nhánh tương ứng với PC đó có khả thi hay không để điều hướng thực thi hiện thời đi theo nhánh đó. Nếu PC được đánh giá là không khả thi, thực thi tượng trưng sẽ dừng hoặc quay lui, và thực thi tượng trưng chỉ thực thi chương trình tương ứng theo nhánh mà PC được đánh giá là khả thi.

1.3. Satisfiability Modulo Theories

1.3.1. Logic mệnh đề

Logic mệnh đề (Propositional Logic) 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 và vì thế, logic mệnh đề không nghiên cứu những tính chất sâu hơn và các quan hệ phụ thuộc vào 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 hội, có nghĩa là một mệnh đề là một phép hội của các tuyển sơ cấp. Ví dụ:

$$(p \vee \neg q) \wedge (\neg r \vee q) \wedge (\neg z \vee q)$$

1.3.2. Satisfiability Modulo Theories

Satisfiability Modulo Theories 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.

Bộ giải SMT 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 bài báo cáo khóa luận này có sử dụng một bộ kết hợp các bộ giải SMT khác nhau để đem lại hiệu quả cao nhất. Cụ thể các bộ giải SMT được sử dụng đó là: Z3 [5] và raSAT [6].

1.3.3. Ứng dụng bộ giải SMT

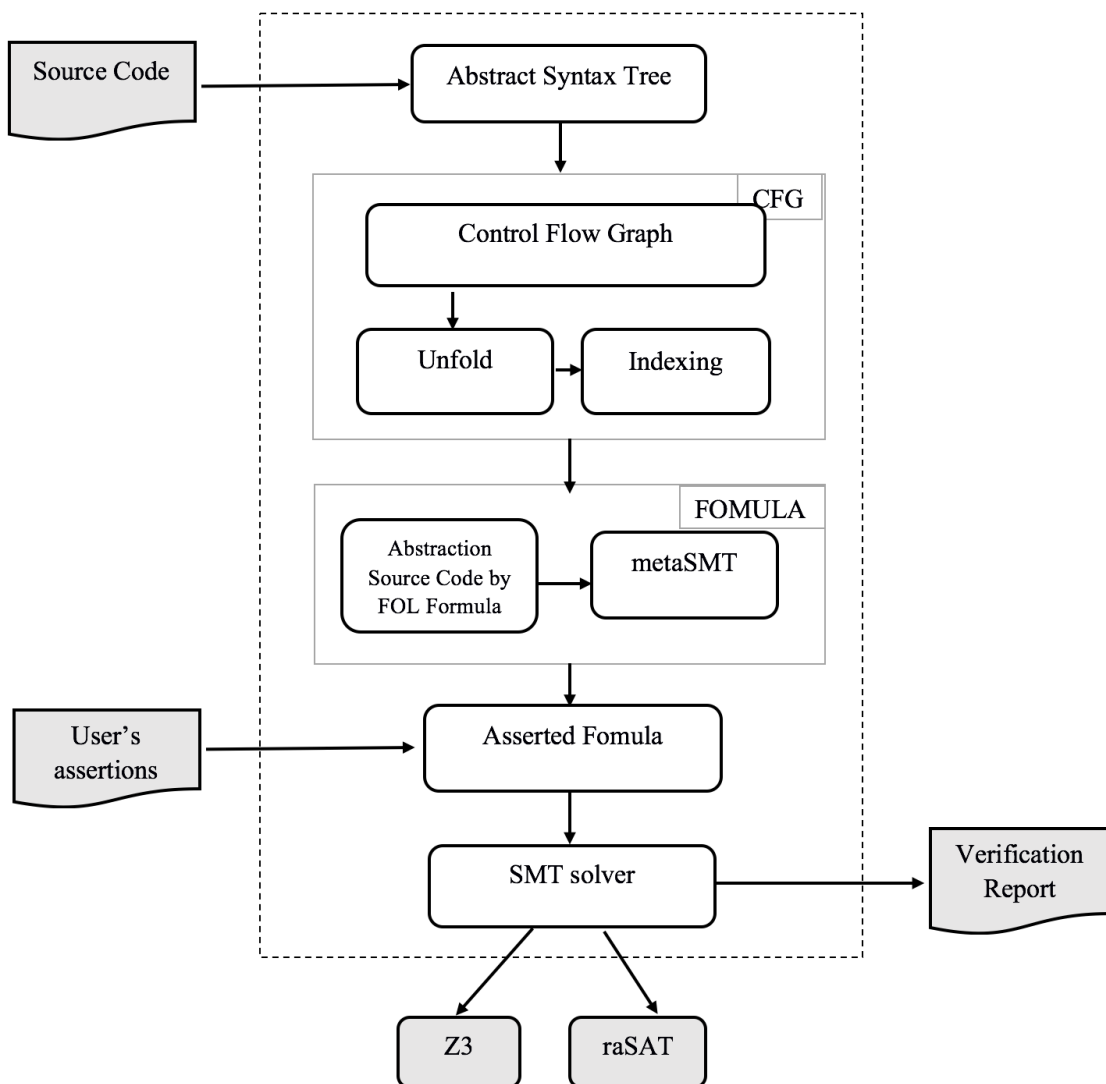
Bộ giải SMT (SMT solver) là một công cụ quyết định có khả năng xử lý các loại lý thuyết số học và các loại lý thuyết có thể quyết định khác. Lý thuyết quyết định được sử dụng trong bộ giải chẳng hạn như số học tuyến tính kết hợp với một bộ giải SAT hình thành nên bộ xử lý trung tâm của bộ giải SMT. Ở một cấp độ cao hơn, bộ giải có khả năng xử lý thay thế các thành phần lặp trong công thức như $(x + y < 12) \parallel ((x - y > 0) \vee (x > 6))$ với các biểu thức mệnh đề $A \parallel (B \vee C)$. Bộ giải SMT hiện tại cung cấp một giải pháp thỏa mãn đầu vào, việc không tìm thấy bất kỳ giải pháp nào được hiểu như là công thức không thỏa mãn. Điều đó giúp ta không cần phải lo lắng đến việc phân tích biểu thức trừu tượng. Nếu bộ giải SMT tìm được một giải pháp thỏa đáng, biểu thức ban đầu sẽ được khôi phục lại và được bộ xử lý trung tâm chấp nhận.

Sử dụng bộ giải SMT chúng ta dễ dàng có thể hình dung được xu hướng di chuyển vào các nhánh trong quá trình thực hiện chương trình, mô hình hóa các biểu thức điều kiện, số học và mảng. Bằng cách thêm vào những điều kiện khác nhau, chúng ta có thể sử dụng bộ giải SMT để đưa các điều kiện này vào những nhánh thi hành khác nhau của chương trình, đảm bảo vị trí lưu trữ luôn có một giá trị cụ thể và kiểm tra các điều kiện.

CHƯƠNG 2. CÔNG CỤ KIỂM CHỨNG CHƯƠNG TRÌNH VTSE

Trong chương này, báo cáo khoá luận xin đề xuất một phương pháp kiểm chứng một số tính chất của chương trình dựa trên kỹ thuật thực thi tượng trưng. Phương pháp này gồm sáu bước chính đi từ mã nguồn chương trình C, kết hợp điều kiện của người dùng cho tới khi có được kết quả thực thi. Cùng với đó, khoá luận cũng đề xuất ra kiến trúc của công cụ kiểm chứng một số tính chất của chương trình VTSE (Verification Tool based on Symbolic Execution) áp dụng phương pháp này. Và cuối cùng là phần minh hoạ hoạt động và ứng dụng của công cụ VTSE cũng sẽ được trình bày trong chương này.

2.1. Kiến trúc của VTSE



Hình 2: Kiến trúc của VTSE

VTSE có đầu vào gồm hai thành phần, đầu vào thứ nhất là mã nguồn của chương trình (Source Code), đầu vào thứ hai là điều kiện do người dùng nhập vào (User's

assertion). Đầu ra 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 vào thứ nhất là mã nguồn của chương trình, VTSE sẽ tiến hành trừu tượng hóa chương trình dựa trên kỹ thuật thực thi tượng trưng thông qua việc tiến hành tuần tự các bước: tạo cây cú pháp trừu tượng, tạo đồ thị luồng điều khiển, loại bỏ vòng lặp khỏi đồ thị luồng điều khiển và sau đó đánh chỉ số cho các biến để trừu tượng hóa chương trình dưới dạng 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. VTSE sử dụng 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 và raSAT.

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:

- SAFE tương ứng với điều kiện của người dùng đặt lên phương thức luôn đúng.
- UNSAFE 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.

2.2. Trừu tượng hoá 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 kỹ thuật kiểm chứng mô hình (*model checking*). Việc áp dụng kỹ 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, khoá luận đề xuất việc tổng quát hóa cho một phần chương trình bằng một biểu thức logic vị từ cấp một dựa trên kỹ thuật 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. Báo cáo khoá luận hướng đến 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.

2.2.1. Cây cú pháp trừu tượng (Abstract Syntax Tree - AST)

Cây cú pháp trừu tượng [7] trong khoa học máy tính là một cây biểu thị cấu trúc cú pháp của mã nguồn một chương trình. Mỗi nút của cây đại diện cho một phần tử cấu trúc trong mã nguồn thay vì đại diện cho mọi dòng lệnh cụ thể.

Cây cú pháp trừu tượng thường là kết quả được sinh ra từ bộ giải sau quá trình dịch mã nguồn và biên dịch. Cụ thể ở công cụ VTSE, CDT (C/C++ Development Tool) [8] được sử dụng để chuyển mã nguồn của ngôn ngữ C/C++ ra cây cú pháp trừu tượng. Để tìm ra được một phần tử cụ thể của cây cú pháp trừu tượng thì chỉ cần sử dụng thuật toán duyệt cây.

2.2.2. Đồ thị luồng điều khiển (Control Flow Graph – CFG)

2.2.2.1. Tính chất đồ thị luồng điều khiển

Đồ thị luồng điều khiển là một cách biểu diễn dòng điều khiển xảy ra khi chương trình được thực thi. Đó là một đồ thị có hướng mà mỗi cạnh đại diện cho một đường thực thi và mỗi nút là đại diện cho một khối lệnh cơ bản - khối lệnh không chứa câu lệnh nhảy hay rẽ nhánh.

2.2.2.2. Đồ thị luồng điều khiển trong VTSE

Đồ thị luồng điều khiển (CFG) trong VTSE là một đồ thị có hướng một chiều chứa tập hợp các nút, mỗi nút đại diện cho một cấu trúc xuất hiện trong mã nguồn.

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 trả về, câu lệnh gọi hàm.
- Nút điều kiện: chứa biểu thức điều kiện tương đương cho câu lệnh rẽ nhánh với 2 con trỏ tới 2 nhánh then và else trong đồ thị.
- Nút lặp: chứa biểu thức lặp và đánh dấu sự xuất hiện của các vòng lặp For, While-do, Do-While.
- 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, nút lời gọi hàm, nút nhảy và nhãn.

2.2.2.3. Xây dựng đồ thị luồng điều khiển

Thuật toán buildCFG(ASTNode)

Đầu vào: nút ASTNode trong cây AST

Đầu ra: hàm trả CFG chứa cặp (BeginNode, EndNode) tham chiếu tới nút đầu và nút cuối của CFG được xây dựng từ ASTNode

```
1  if (ASTNode là câu lệnh đơn giản)
2    node := nút CFG đơn chứa ASTNode
3    CFG = newCFG(node, 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    CFG := null
12  CFG := newCFG(begin.next, last);
13  else if (ASTNode là câu lệnh if - else)
14    thenBranch := newCFG tương ứng với nút then
15    elseBranch := newCFG 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    CFG := newCFG(conditionNode, endCondition)
23  else if (ASTNode là câu lệnh for)
24    Tạo BgForNode và EndForNode
25    InitNode := Nút chứa điều kiện khởi tạo vòng lặp
26    DecisionNode := nút chứa điều kiện lặp
27    thenBranch := newCFG tương ứng với nhánh thực thi thân chương trình for
28    elseBranch := CFGNode kết thúc vòng lặp for
29    IterationNode := tương ứng với node đánh dấu vòng lặp
30    EndOfThen := tương ứng với IterationNode
31    EndOfElse : Tương ứng với EndForNode
```

```

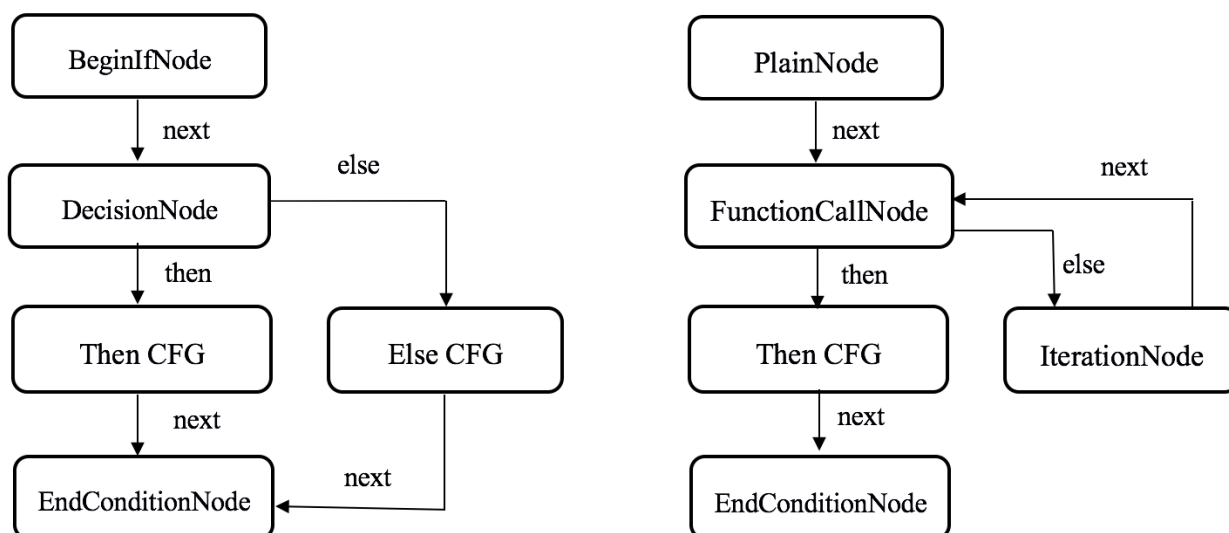
32   CFG := newCFG(BgForNode, EndForNode)
33   else if (ASTNode là câu lệnh do)
34       Tạo BeginNode và EndNode
34       DecisionNode := nút chứa điều kiện lặp
35       thenBranch := newCFG tương ứng với nhánh thực thi thân chương trình for
36       elseBanch := CFGNode kết thúc vòng lặp for
37       IterationNode := tương ứng với node đánh dấu vòng lặp
38       EndOfThen := tương ứng với IterationNode
39       EndOfElse : Tương ứng với EndForNode
40       CFG := newCFG(BeginNode, EndNode)
41   return CFG

```

Đối với một khối lệnh (Compound Statement), thuật toán tạo ra một dãy các nút tương ứng với các câu lệnh trong khối.

Đối với các câu lệnh rẽ nhánh, tạo ra nút bắt đầu rẽ nhánh, sau đó tạo nút điều kiện có 2 nhánh: một nhánh *then* trở đến CFG của khối lệnh *then*, nhánh *else* trở đến CFG của khối lệnh *else*.

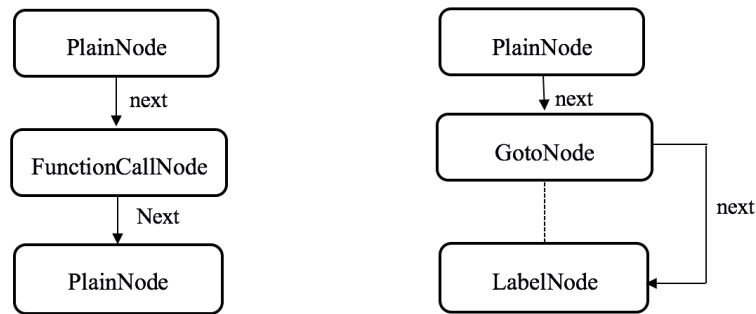
Đối với các câu lệnh lặp, tạo nút bắt đầu lặp, sau đó tạo nút điều kiện có hai nhánh: một nhánh *then* trở đến CFG của khối lệnh *then*, nhánh *else* trở đến nút Iteration chứa biểu thức lặp.



Hình 3: Khối điều khiển của CFG

Đối với câu lệnh có chứa lời gọi hàm, tạo các nút đơn chứa tham số của hàm được gọi, sau đó nối với nút lời gọi hàm.

Đối với lệnh *goto*, nối nút *goto* tới nút label tương ứng.

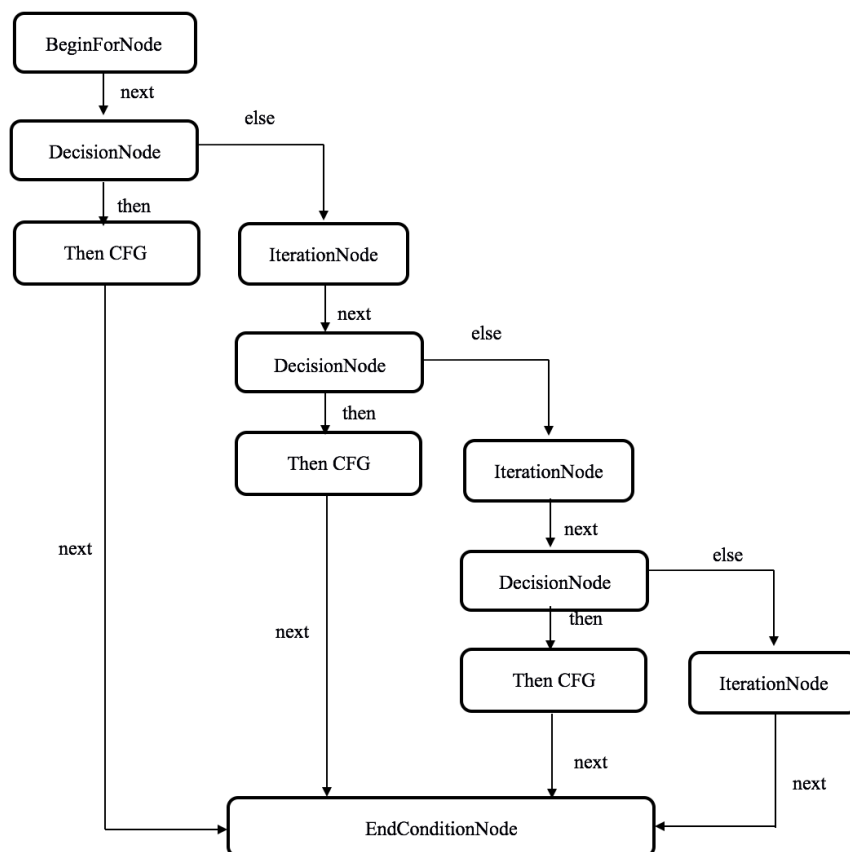


Hình 4: Khởi lệnh *goto* và lời gọi hàm

2.2.3. Xử lý vòng lặp trong CFG

Loại bỏ vòng lặp trong đồ thị luồng điều khiển

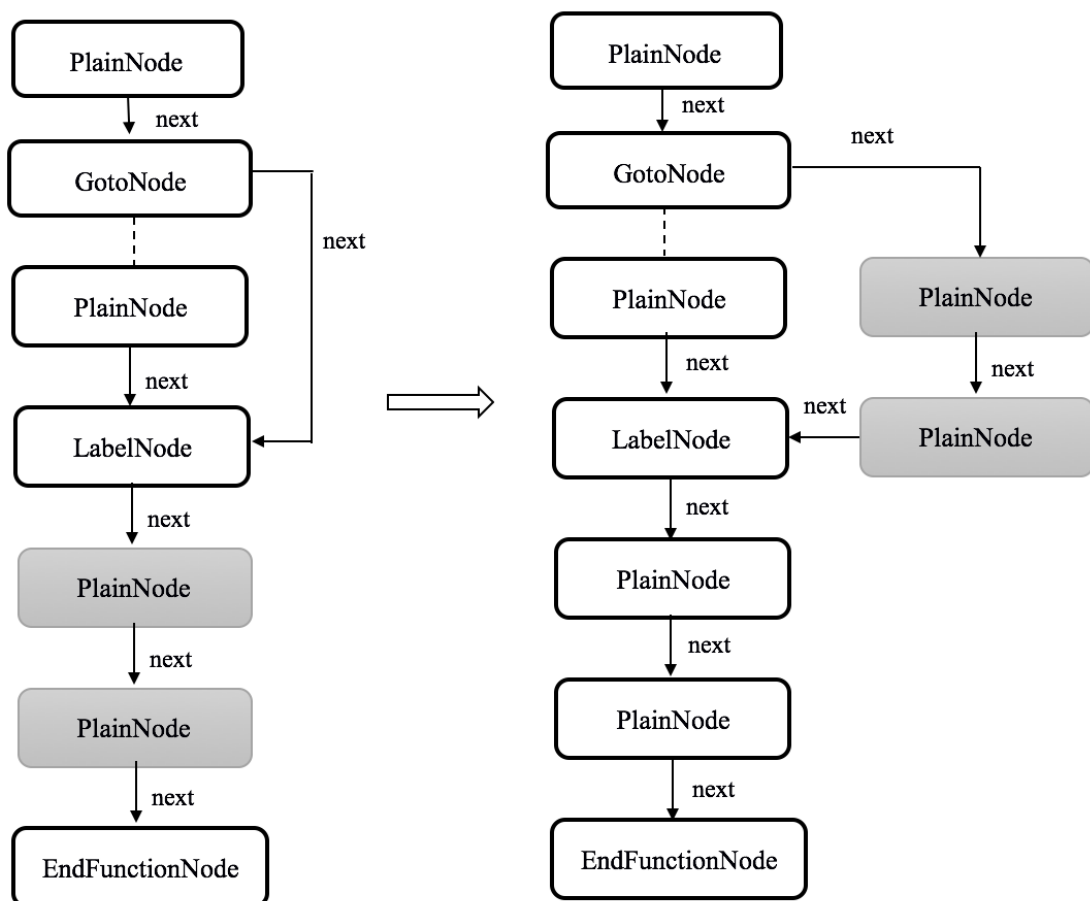
Để loại bỏ vòng lặp trong đồ thị, duyệt đồ thị từ đầu đến cuối, nếu bắt gặp BeginForNode hay BeginWhileNode, vòng lặp được gỡ dần dần bằng cách chuyển về một số hữu hạn câu lệnh If.



Hình 5: Xử lý vòng lặp trong CFG

Loại bỏ câu lệnh nhảy trong đồ thị luồng điều khiển

Khi duyệt đồ thị gặp lệnh nhảy, tạo một nhánh mới nối vào với GotoNode bằng cách sao chép đoạn đồ thị.



Hình 6: Xử lý lệnh goto

Trong bộ giải SMT 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 phương thức ta phải đánh chỉ số tương ứng với giá trị của biến trong từng thời điểm cụ thể của biến trong quá trình thực thi chương trình.

Biến cục bộ được khởi tạo chỉ số -1 khi khai báo mà chưa gán giá trị và 0 trong trường hợp ngược lại. Từ đây mỗi lần cập nhật giá trị, chỉ số biến tăng thêm một đại diện cho giá trị mới. Tham số được xử lý tương tự biến cục bộ với giá trị khởi tạo bằng 0. Biến toàn cục chỉ được khởi tạo 1 lần và tăng trong tất cả phương thức sử dụng nó.

Minh họa bằng một đoạn mã nguồn:

STT	Câu lệnh	n	a	b
1	public void foo(int n) {	0		
2	int a;	0	-1	
3	int b = 0;	0	-1	0
4	a = n * n;	0	0	0
5	b = a;	0	0	1
6	}			

Các biến được khởi tạo chỉ số theo từng loại (n, 0) (a, -1) (b, 0). Tại dòng lệnh $a = n * n$ và $b = a$ chỉ số 2 biến tăng tương ứng với sự thay đổi giá trị thu được (a, 0) và (b, 1).

Danh sách biến được lưu trong đối tượng VM thuộc lớp Variable Manager lưu nội dung tên, chỉ số của từng biến. VM được cập nhật song song với quá trình duyệt cây CFG.

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 chứa các 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 số 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 bên 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 chứa 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 số 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, công cụ phải tiến hành đồng bộ chỉ số của các biến bị thay đổi giá trị (chỉ số) trong khối lệnh rẽ nhánh. 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 đặt chỉ số của biến v sau khi kết thúc nút điều kiện là i_2
- 8 **else if** ($i_2 < i_1$)
- 9 tạo nút đồng bộ tương ứng với biểu thức $v_{i_1} = v_{i_2}$
- 10 thêm nút đồng bộ vào nhánh else
- 11 đặt chỉ số của biến v sau khi kết thúc nút điều kiện là i_1

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 số bên nhánh then nhỏ hơn thì thêm nút đồng bộ để tăng chỉ số lên bằng giá trị nhánh *else*. Thực hiện tương tự đối trong trường hợp chỉ số bên nhánh *else* nhỏ hơn. Như vậy sau khi kết thúc việc đánh 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 hai nhánh đặt được.

2.2.4. Biểu diễn chương trình dưới dạng metaSMT

Kết thúc việc đánh chỉ số cho toàn bộ các biến có mặt trong phương thức, ta sẽ đưa ra biểu diễn của tổng quát hóa chương trình dưới dạng metaSMT. Đưa ra biểu diễn dưới dạng metaSMT là bước đệm trước khi đưa ra công thức tổng quát hóa cuối cùng của phương thức. Biểu diễn 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 theo dõi cũng như dễ đọc so với công thức tổng quát hóa cuối cùng.

Nhiệm vụ chính của việc biểu diễn chương trình dưới dạng 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.

2.2.5. Biểu diễn chương trình dưới dạng logic vị từ

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). Công thức tuân theo khuôn dạng đầu vào của từng bộ giải SMT.

Mỗi câu lệnh trong chương trình sẽ trở thành 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 trong mã nguồn. Điều khác biệt ở đây là các biến trong công thức có kèm theo chỉ số tương ứng của chúng trong quá trình thực thi chương trình. Công thức này tuân thủ theo chuẩn đầu vào của bộ giải SMT.

2.2.6. Kiểm tra các tính chất của chương trình dựa trên bộ giải SMT

Sử dụng bộ giải SMT chúng ta dễ dàng có thể hình dung được xu hướng di chuyển vào các nhánh trong quá trình thực hiện chương trình. Chúng ta có thể mô hình hóa các biểu thức điều kiện, số học và mảng, đó chính là công thức tổng quát của chương trình. Bằng cách thêm gắn thêm các ràng buộc của người dùng ta thu được đầu vào bộ giải SMT hợp của các nhánh điều kiện khác nhau.

Gọi $F_{\text{abstraction}}$ là biểu diễn trừu tượng của phương thức, $F_{\text{assertion}}$ là điều kiện do người dùng nhập vào.

$$F_{\text{final}} = F_{\text{abstraction}} \wedge \neg F_{\text{assertion}}$$

F_{final} là biểu thức được đưa vào bộ giải.

Kết quả cuối cùng sau khi đưa vào bộ giải giúp xác định chương trình đã thỏa mãn điều kiện đặt ra hay chưa. Kết quả này chỉ có 2 dạng SAT và UNSAT.

Kết quả do bộ giải SMT trả về là SAT (satisfaction): tồn tại bộ giá trị của các biến làm thỏa mãn biểu thức F_{final} . Điều này tương đương với việc tồn tại bộ giá trị của các biến làm cho biểu thức $\neg F_{\text{assertion}}$ thỏa mãn, hay tồn tại bộ giá trị của các biến làm cho biểu thức $F_{\text{assertion}}$ 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 bộ giải SMT trả về là UNSAT (unsatisfaction): không tồn tại bộ giá trị của các biến làm thỏa mãn biểu thức F_{final} . Điều này tương đương với việc không có bộ giá trị nào của các biến làm cho biểu thức $\neg F_{\text{assertion}}$ thỏa mãn, hay không tồn tại bộ giá trị của các biến làm cho biểu thức $F_{\text{assertion}}$ không thỏa mãn. Trong trường hợp này, công cụ sẽ thông báo với người dùng là điều kiện người dùng đưa ra luôn đúng.

2.3. Minh họa hoạt động của VTSE

Để minh họa cho hoạt động của VTSE, giả sử ta có một chương trình tính tổng từ 1 đến trị tuyệt đối của n được viết bởi ngôn ngữ C như ví dụ 2:

```
1 int abs (int n) {  
2     if (n ≥ 0) {  
3         return n;  
4     } else {
```

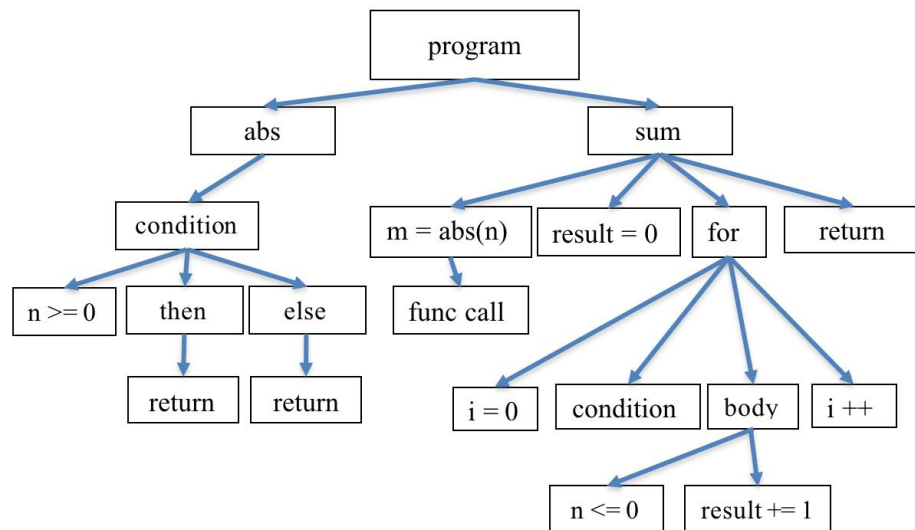
```

5      return -n;
6  }
7  }
8
9  int sum (int n) {
10     int result = 0;
11     int m = abs(n);
12     int i;
13     for (i = 0; i ≤ m; i++) {
14         result += i;
15     }
16     return result;
17 }

```

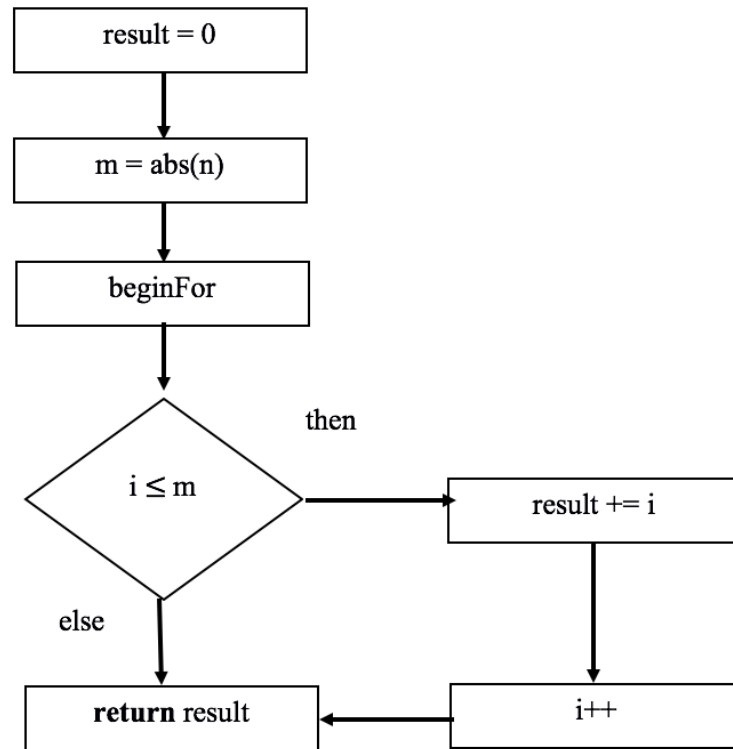
Sau khi có đầu vào là mã nguồn chương trình, VTSE tiến hành trừu tượng hóa đầu vào để sinh ra biểu thức vị từ cấp 1 theo các bước sau:

- Bước 1: Xây dựng cây cú pháp trừu tượng từ mã nguồn sử dụng thư viện CDT



Hình 7: Cây cú pháp trừu tượng AST cho mã nguồn ví dụ 2

- Bước 2: Xây dựng đồ thị luồng điều khiển từ cây cú pháp trừu tượng



Hình 8: Đồ thị luồng điều khiển cho ví dụ 2

- Bước 3: Xử lý vòng lặp, câu lệnh nhảy và lời gọi hàm trong đồ thị luồng điều khiển.



Hình 9: Đồ thị luồng điều khiển sau khi xử lý vòng lặp và đồng bộ chỉ số

- Bước 4: Đánh và đồng bộ hóa chỉ số của biến trong đồ thị luồng điều khiển.
- Bước 5: Sinh biểu thức vị từ cấp một
- $((\text{return_sum_0} = \text{result_sum_1}) \wedge$

$$\begin{aligned}
 &(((i_sum_1 \leq m_sum_0) \Rightarrow (\text{result_sum_1} = (\text{result_sum_0} + i_sum_1))) \wedge \\
 &((i_sum_1 = 0) \wedge ((m_sum_0 = \text{abs_n_1}) \wedge ((\text{abs_n_1} = \text{return_abs_0}) \wedge \\
 &(((n_abs_1 \geq 0) \Rightarrow (\text{return_abs_0} = n_abs_1)) \wedge ((\text{not } (n_abs_1 \geq 0)) \Rightarrow \\
 &(\text{return_abs_0} = n_abs_1))) \wedge ((n_abs_1 = n_sum_0) \wedge (\text{result_sum_0} = 0))))))
 \end{aligned}$$

- Bước 6: Kết hợp điều kiện người dùng đưa biểu thức vào bộ giải Z3, đưa ra kết quả.

2.4. Một số ứng dụng

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 chương trình để tạo thành một biểu thức trừu tượng hóa cuối cùng và đưa vào các bộ giải SMT để đưa ra kết luận.

Phần này sẽ đề cập đến một số ứng dụng 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 hậu điều kiện từ tiền điều kiện, kiểm tra tính chất của các điều kiện phức hợp do người dùng nhập vào, kiểm tra chương trình có chạy vào các câu lệnh đặc biệt được xác định trong chương trình, xác định vị trí đoạn mã nguồn có thể không bao giờ được thực thi (Unreachable code).

2.4.1. Kiểm tra giá trị trả về của một hàm

Bằng cách trừu tượng hóa mã nguồn của cả chương trình về một biểu thức biểu diễn bởi logic vị từ cấp 1, có các biến là các giá trị cụ thể, VTSE có thể kiểm tra được giá trị trả về của một hàm có thỏa mãn một điều kiện nào đó hay không.

Xét đoạn mã nguồn như ví dụ 3 sau:

```
1 float foo (float a, float b) {  
2     float result;  
3     if (a > b) {  
4         result = a / (b - 2);  
5     } else {  
6         result = a + b;  
7     }  
8     return result;  
9 }  
10 ASSERT((return > 0) ^ (return < 1));
```

Giả sử người dùng muốn kiểm tra giá trị trả về luôn thuộc đoạn trong đoạn $[0,1]$, đầu vào của VTSE gồm 2 thành phần:

- Đầu vào thứ nhất là mã nguồn của chương trình
- Điều kiện khoảng nghiệm muốn kiểm tra (hậu điều kiện):

$$(\text{return} > 0) \wedge (\text{return} < 1)$$

Sau khi chạy chương trình, kết quả VTSE đưa ra

```
<terminated> TestBenchmark (2) [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (11:07:14)
-Method name: foo
Status: unsafe
Counter example:
+ parameter: a float (1.0 / 2.0)
+ parameter: b float (1.0 / 8.0)
+ return = - 4.0 / 15.0
Total time: 30.0 (ms)
```

Hình 10: Kết quả chạy VTSE cho chương trình trong ví dụ 3

2.4.2. Kiểm tra hậu điều kiện từ tiền điều kiện

Công cụ VTSE cho phép kiểm tra tính thỏa mãn của hậu điều kiện so với các tiền điều kiện do người dùng nhập vào.

Để kiểm tra chương trình xây dựng công thức trên có chính xác hay không, ta có thể sử dụng VTSE với mã nguồn của chương trình C++ trong ví dụ 4 sau:

```
1  ASSUME((n ≥ 0) ∧ (n < 6));
2  int sum (int n) {
3      int result = 0;
4      for (int i = 0; i ≤ n; i++) {
5          result = result + i;
6      }
7      return 0;
8  }
9  ASSERT(return = (n * (n + 1))/2);
```

Đoạn mã này có chứa vòng lặp, người dùng có thể chọn số lần thực hiện vòng lặp, ở đây, chúng tôi chọn là 6.

Người dùng muốn kiểm tra biểu thức hậu điều kiện có thỏa mãn với tiền điều kiện không, trong trường hợp này cần thiết lập đầu vào VTSE như sau:

Biểu thức tiền điều kiện:

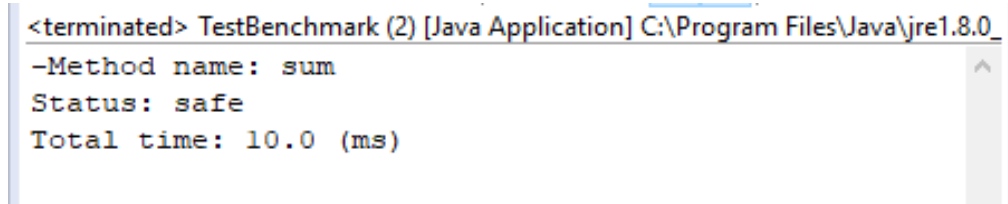
$$(n \geq 0) \wedge (n < 6)$$

Biểu thức trên để đảm bảo: biến đầu vào là số nguyên không âm và đảm bảo số vòng lặp được thực thi là 6.

Biểu thức hậu điều kiện chính là công thức cần kiểm tra:

$$\text{return} = n * (n + 1) / 2$$

Sau khi chạy VTSE, ta có kết quả:



```
<terminated> TestBenchmark (2) [Java Application] C:\Program Files\Java\jre1.8.0_...
-Method name: sum
Status: safe
Total time: 10.0 (ms)
```

Hình 11: Kết quả chạy VTSE cho chương trình trong ví dụ 4

2.4.3. Kiểm tra tính chất do người dùng đưa vào

Chương trình có thể kiểm tra sự thỏa mãn của bất kì biểu thức đầu ra nào mà người dùng muốn kiểm tra trên chương trình của mình.

Xét đoạn mã nguồn C++ trong ví dụ 5 về đoạn mã chương trình foo kiểm tra khoảng trả về của phương thức:

```
1  ASSUME((a < 0) ^ (b < 1));
2  float foo (float a, float b) {
3      float result = 0;
4      if (a > b) {
5          result = a / (b - 2);
6      } else {
7          result = a + b;
8      }
9      return result;
10 }
11 ASSERT(return * return < (a + b)/2);
```

Để kiểm tra giá trị trả về của phương thức có thỏa mãn biểu thức phức hợp:

$$\text{return} * \text{return} < (a + b) / 2$$

Khi đó, đầu vào của công cụ là:

- Mã nguồn chương trình C/C++
- Biểu thức phức hợp người dùng muốn kiểm tra (User's assertion):

$$\text{return} * \text{return} < (a + b) / 2$$

Kết quả của VTSE với chương trình trên là:

```
<terminated> TestBenchmark (2) [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (10:5)
-Method name: foo
Status: unsafe
Counter example:
+ parameter: a float (- 3.0 / 8.0)
+ parameter: b float (1.0 / 2.0)
+ return = 1.0 / 8.0
Total time: 20.0 (ms)
```

Hình 12: Kết quả chạy VTSE cho chương trình trong ví dụ 5

2.4.4. Kiểm tra chương trình có chạy vào các câu lệnh xác định

Vấn đề của kiểm chứng tính an toàn của chương trình là xác định một chương trình cho trước có vi phạm biểu thức đặt sẵn (hay có dẫn tới trạng thái không an toàn) hay không.

Ta xét ví dụ 6 dưới đây là một chương trình được định dạng theo chuẩn của cuộc thi SV-COMP:

```
1  int newton_1_1_true_unreach_call () {
2    float IN;
3    ASSUME(IN > -0.2 ^ IN < 0.2);
4    float x = IN – f(IN)/fp(IN);
5    x = x – f(x)/fp(x);
6    ASSERT(x < 0.1);
7    return 0;
8 }
```

Chương trình trên được xác nhận là an toàn (*safe*) khi không có đường thực thi nào vi phạm điều kiện được quy định trong câu lệnh **ASSERT** và ngược lại, được đánh giá là không an toàn khi vi phạm điều kiện được quy định trong câu lệnh **ASSERT**.

VTSE có thể kiểm tra tính an toàn của chương trình trên bằng cách quy định:

Biểu thức đầu vào (tiền điều kiện) được lấy trong câu lệnh **ASSUME**:

$$(IN + 0.2 > 0) \wedge (IN < 0.2)$$

Biểu thức đầu ra (hậu điều kiện) là điều kiện trong câu lệnh **ASSERT**:

$$(x < 0.1)$$

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

```
<terminated> TestBenchmark (2) [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (-Xmx1024m)
-Method name: newton_1_1_true_unreach_call
Status: safe
Total time: 10.0 (ms)
```

Hình 13: Kết quả chạy VTSE cho chương trình trong ví dụ 6

Công cụ đưa ra thông báo chương trình là luôn đúng, có nghĩa không bao giờ vi phạm điều kiện đặt trong câu lệnh ASSERT.

2.4.5. Kiểm tra khối lệnh không bao giờ được thực thi

Khối lệnh không được thực thi (*unreachable code*) là đoạn lệnh mà chương trình sẽ không bao giờ chạy vào khi chương trình thực thi. Trong lập trình, câu lệnh như `assert(0)` hay `assert(false)` thường được dùng để đánh dấu một đoạn lệnh không với tới được, như vậy khi việc thực thi của chương trình bị hủy bỏ do chạy vào lệnh `assert(0)`, ta có thể xác định được là chương trình không thực hiện đúng như mong đợi.

VTSE có thể ứng dụng để kiểm tra khi chương trình thực thi có chạy vào khối lệnh không bao giờ được thực thi hay không.

Xét ví dụ 7 sau đây:

```
1  int flag = 0;
2  int add (int x, int y) {
3      int n = 10, sum;
4      if (n > 10) {
5          sum = x + y;
6      } else {
7          int z = x * y; // Unreachable code
8          flag = 1;
9      }
10     return sum;
11 }
12 ASSERT(flag == 0)
```

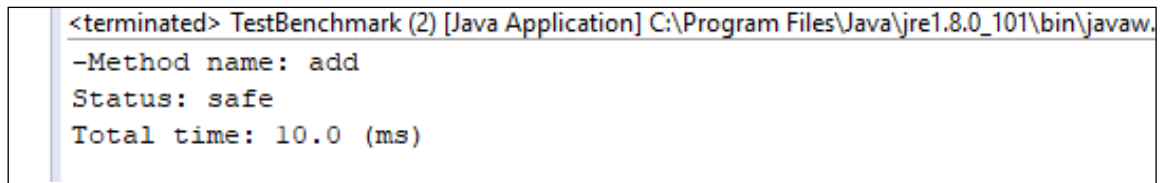
Để kiểm tra chương trình có chạy vào khối lệnh unreachable, ta đặt hậu điều kiện:

ASSERT(flag == 0)

Nếu VTSE trả về là hậu điều kiện luôn đúng với hàm đã cho (*safe*), có nghĩa là đoạn mã thay đổi giá trị biến `assert` không được chạy vào, hay là luồng điều khiển của chương trình không đi vào đoạn lệnh không bao giờ được thực thi.

Ngược lại, nếu VTSE trả về hậu điều kiện là không đúng (not safe), có nghĩa là chương trình thực hiện không đúng mong đợi và chạy vào đoạn lệnh mà đã được đánh dấu trước đó.

Với ví dụ trên, VTSE trả về kết quả:



```
<terminated> TestBenchmark (2) [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.  
-Method name: add  
Status: safe  
Total time: 10.0 (ms)
```

Hình 14: Kết quả chạy VTSE cho chương trình trong ví dụ 7

CHƯƠNG 3: CÁC NGHIÊN CỨU VÀ ỨNG DỤNG LIÊN QUAN

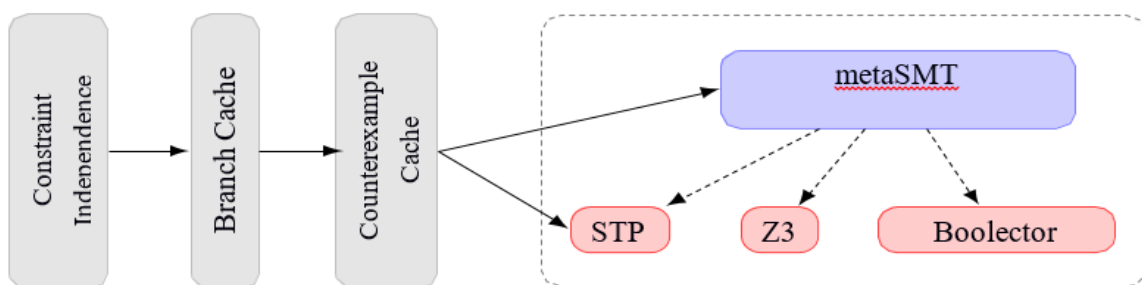
Chương này sẽ trình bày về các nghiên cứu các ứng dụng liên quan đến kiểm chứng và kiểm thử chương trình. Trong chương này, bài khoá luận đề cập tới 5 công cụ đó là KLEE [9], Cascade [10], CBMC [11], 2LS [12] và Ceagle [13]. KLEE là một công cụ có khả năng sinh các ca kiểm thử một cách tự động áp dụng kỹ thuật thực thi tượng trưng, được ra đời rất sớm là nền tảng để phát triển nhiều công cụ kiểm chứng, kiểm thử sau này. Cascade là một công cụ kiểm chứng áp dụng kỹ thuật thực thi tượng trưng, có kiến trúc và ứng dụng gần giống với VTSE, tuy nhiên việc điều khiển công cụ thực hiện phải được thực hiện thủ công mà chưa tự động hoá được, VTSE đã khắc phục được nhược điểm này của Cascade. CBMC là hai công cụ kiểm chứng áp dụng kỹ thuật kiểm chứng mô hình và đạt giải cao trong cuộc thi dành cho các công cụ kiểm chứng SV-COMP. Và cuối cùng là Ceagle, cũng là một công cụ kiểm chứng đạt thành tích cao tại cuộc thi SV-COMP và áp dụng kỹ thuật thực thi tượng trưng.

3.1. KLEE

3.1.1. Tổng quan

KLEE 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. KLEE thuộc nhóm công cụ sử dụng kỹ 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.

3.1.2. Mô hình hoạt động



Hình 15: Sơ đồ hoạt động của KLEE [9]

Constraint 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 ảnh 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 đưa ra một phản ví dụ (*Counterexample Cache*).

KLEE sử dụng phản ví dụ 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.

3.1.3. 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 $C++$

3.1.4. 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 lập với điều kiện tượng trưng và các đối tượng có kích cỡ tượng trưng (Symbolic size).

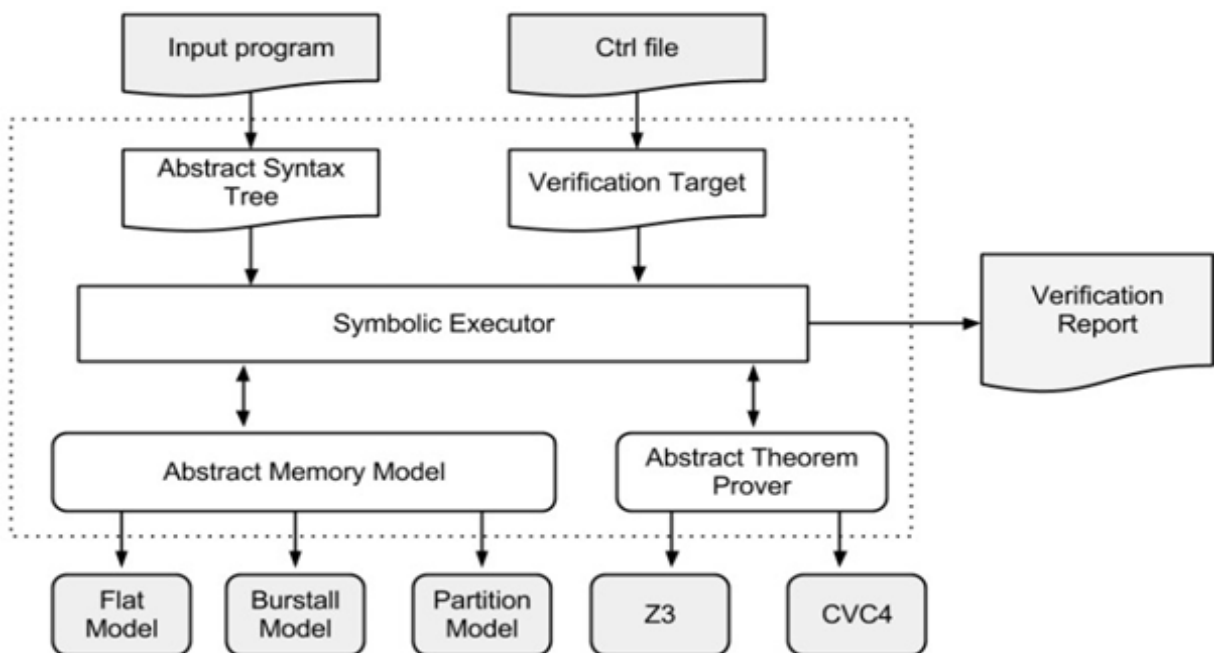
Ngoài ra, KLEE cũng không thể khám phá các vùng mã chết và vùng mã không bao giờ được thực thi (Unreachable code and dead code)

3.2. Cascade

3.2.1. Tổng quan

Cascade 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 nhiều biểu thức (assertion) 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 bộ giải SMT và đưa ra một minh chứng hoặc ví dụ cụ thể để chỉ ra rằng assertion đó có thể sai.

3.2.2. Thiết kế hệ thống



Hình 16: Kiến trúc hệ thống của Cascade [10]

Tập tin điều khiển (*Control file*). Tập tin điều khiển dùng để hướng dẫn thực thi tượng trưng, sử dụng ngôn ngữ XML.

Tất cả các tập tin điều khiển đều bắt đầu với một tập tin nguồn (*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 được thể hiện như sau:

<pre> int abs(int x) { int result; if(x>=0) result = x; else result = -x; return result; } </pre>	<pre> <controlFile> <sourceFile name="abs.c" id="1" /> <run> <startPosition fileId="1" line="1" /> <wayPoint fileId="1" line="4" /> <endPosition fileId="1" line="8" /> </run> <run> <startPosition fileId="1" line="1" /> <wayPoint fileId="1" line="6" /> <endPosition fileId="1" line="8" /> </run> </controlFile> </pre>
--	--

Hình 17: Ví dụ tập tin điều khiển của công cụ Cascade [10]

Lời gọi hàm (*function call*). 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 (*loop*). 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.

3.2.3. Điểm mạnh và điểm yếu

Cascade có khả năng kiểm chứng một số tính chất của chương trình theo điều kiện người của người dùng đưa vào thông qua một tập tin điều khiển. Cascade có khả năng xử lý các chương trình phức tạp với số lần lặp của vòng lặp được tùy chỉnh. Tuy nhiên việc xử lý chưa được tự động và Cascade vẫn chưa xử lý được vấn đề bùng nổ đường thi hành.

3.3. CBMC

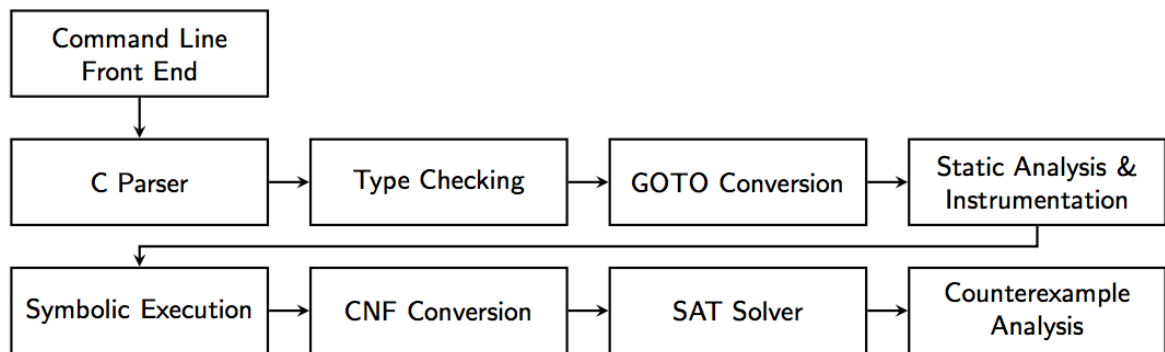
3.3.1. Tổng quan

CBMC (C Bounded Model Checker) là một công cụ kiểm chứng mô hình giới hạn (Bounded Model Checking) cho chương trình C, đã được phát triển và duy trì hơn mười năm. CBMC có thể chỉ ra những vi phạm vào một điều kiện đặt ra hoặc chứng minh chương trình an toàn với một giới hạn nào đó. CBMC tiến hành chuyển đổi một đầu vào chương trình C, với điều kiện cho trước được xác định trong chương trình và số lần lặp cho vòng lặp trong chương trình, tạo thành một công thức. Nếu công thức này là

thoả mãn, đồng nghĩa tồn tại đường thi hành của chương trình dẫn tới điều kiện cho trước. Trong cuộc thi SV-COMP, sự thoả mãn của công thức được quyết định bằng việc sử dụng bộ giải MiniSat.

3.3.2. Kiến trúc

Một công cụ kiểm chứng mô hình giới hạn như CBMC biến đổi những vấn đề về các đường thi hành thành những ràng buộc có thể được giải bằng các bộ giải SAT hay SMT. VVoiws một chương trình đầu vào được kèm theo là các điều kiện cho trước, CBMC sẽ đưa ra một công thức dưới dạng chuẩn tắc hội (Conjunctive Normal Form - CNF) của kết quả mà làm cho điều kiện được đưa ra trước xảy ra.



Hình 18: Kiến trúc của CBMC [11]

Giai đoạn tiếp nhận và xử lý đầu vào (*Front End*). Mô-đun đầu tiên của CBMC đó là giao diện người dùng thông qua dòng lệnh (*Command Line Front End*), tại đây CBMC sẽ được cấu hình theo những tham số mà người dùng đưa ra. Trình phân tích mã nguồn C (*C Parser*) sử dụng một bộ xử lý C có sẵn (ví dụ như *gcc -E*) để xây dựng một cây phân tích từ dữ liệu được tiền xử lý. Bộ kiểm tra dữ liệu (*Type Checking*) sẽ tạo ra một bảng ký hiệu với tên của từng loại và định danh ký hiệu bằng cách duyệt cây phân tích. Mỗi một ký hiệu được gán một thông tin kiểu bit. CBMC sẽ hủy bỏ nếu có bất kỳ một bất thường xảy ra ở giai đoạn này.

Giai đoạn xử lý trung gian (*Intermediate Representation*). CBMC sử dụng bộ chương trình là chương trình GOTO (*GOTO Programs*) làm đại diện trung gian. Trong ngôn ngữ này, tất cả các luồng điều khiển phi tuyến tính, vòng lặp và câu lệnh nhảy được chuyển đổi thành các câu lệnh goto có chức năng tương đương. Những câu lệnh là các nhánh chỉ dẫn bao gồm các điều kiện (tùy chọn). CBMC tạo ra một chương trình sử dụng các câu lệnh GOTO với mỗi chương trình C được xác định trên cây phân tích. Hơn

nữa, nó sẽ tạo ra một hàm main mới, hàm này đầu tiên sẽ gọi hàm khởi tạo cho các biến toàn cục rồi sau đó gọi các hàm gốc của chương trình.

Ở giai đoạn này, CBMC thực hiện một phân tích tĩnh để xử lý các con trỏ hàm đến một trường hợp phân tách trên toàn các hàm, kết quả là thu được một đồ thị gọi hàm tĩnh. Cùng với đó, các điều kiện được đặt vào để đảm bảo cho việc các con trỏ không hợp lệ hay rò rỉ bộ nhớ xảy ra.

Giai đoạn xử lý trung tâm (*Middle End*). CBMC thực hiện thực thi tượng trưng bằng cách duỗi vòng lặp với một số lần lặp là cố định, số lần lặp này có thể được cấu hình bởi người sử dụng cho mỗi vòng lặp cụ thể hoặc là tùy chỉnh cho toàn cục cho tất cả các vòng lặp. Trong quá trình này CBMC cũng chuyển các câu lệnh GOTO thành dạng tĩnh của những câu lệnh gán đơn giản (*Static Simple Assignment*). Truyền hằng số và đơn giản biểu thức là chìa khóa để đạt được sự hiệu quả và ngăn chặn việc thực thi những nhánh không khả thi. Vào cuối quá trình này, chương trình được biểu diễn như một hệ phương trình của các biến đã được đổi tên trong chương trình và với các câu lệnh điều kiện. Các điều kiện được ra nhằm mục đích xác định xem một câu lệnh có thực sự được thực thi trong một nhánh cụ thể nhất định hay không.

Giai đoạn cuối cùng (*Back End*). CBMC hỗ trợ rất nhiều bộ giải SMT khác nhau. Ở trong cuộc thi SV-COMP, CBMC sử dụng bộ giải MiniSat làm bộ giải của mình chính vì vậy, phương trình cuối cùng được biểu diễn thành công thức dạng chuẩn tắc hội bằng cách mô hình hóa đến mức chính xác bit. Một mô hình được tính toán bởi bộ giải SAT tương ứng với một đường thi hành vi phạm ít nhất một trong các điều kiện trong chương trình được xem xét một cách tỉ mỉ, và cùng với đó một phản ví dụ cụ thể được đưa ra được xem như là bằng chứng cho sự vi phạm điều kiện này. Ngược lại, nếu công thức cuối cùng không thỏa mãn thì không có điều kiện nào bị vi phạm trong giới hạn số lần lặp được xác định từ trước đó.

3.3.3. Điểm mạnh và điểm yếu

Là một công cụ kiểm chứng giới hạn và không có các phép biến đổi số vòng lặp một cách tối ưu, CBMC không thể chứng minh sự đúng đắn của một chương trình với số vòng lặp không bị giới hạn một cách tổng quát.

Điểm mạnh của kiểm chứng trong giới hạn đó là hiệu năng có thể dự đoán và khả năng tương thích với nhiều bộ dữ liệu khác nhau

CBMC có khả năng thực hiện hoàn toàn tự động và có khả năng phát hiện rất nhiều lỗi có thể có trong chương trình.

3.3.4. So sánh với VTSE

So với VTSE, CBMC sử dụng kỹ thuật kiểm chứng mô hình giới hạn. Mục đích của CBMC là kiểm tra toàn bộ các đường thi hành có thể có trong chương trình, còn VTSE hướng tới việc trừu tượng hoá chương trình thành một biểu thức logic vị từ cấp I duy nhất. Chính vì áp dụng kỹ thuật kiểm chứng mô hình nên CBMC sẽ phải đối mặt với việc bùng nổ đường thi hành còn VTSE giải quyết được vấn đề này là nhờ áp dụng kỹ thuật thực thi tượng trưng.

Ngoài ra, CBMC còn có khả năng biểu diễn biểu thức của mỗi đường thi hành dưới dạng bit-vector để đưa vào bộ giải thay vì biểu diễn dưới dạng biểu thức dạng chuẩn tắc hội.

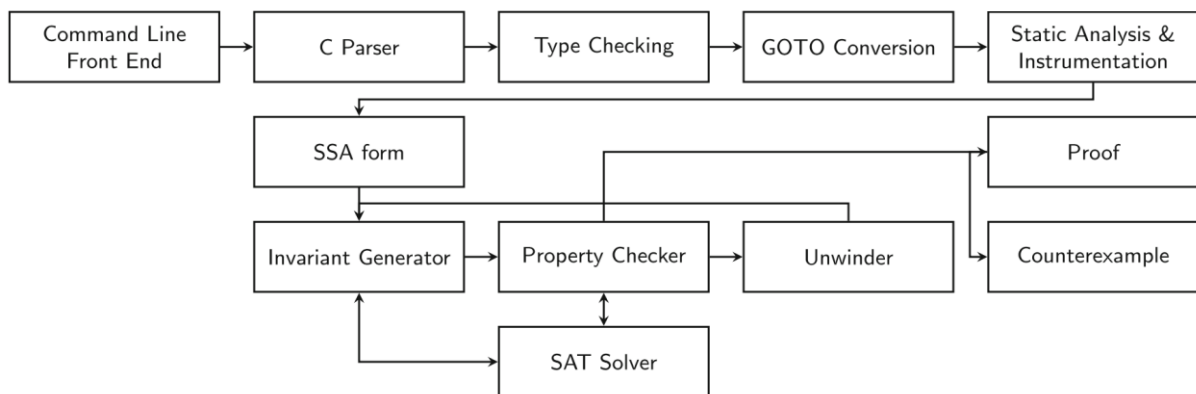
Ở cuộc thi SV-COMP, CBMC đứng thứ nhất năm 2015, đứng thứ 3 năm 2016 và năm 2017 tại hạng mục float-cdfpl.

3.4. 2LS

3.4.1. Tổng quan

2LS là một công cụ phân tích tĩnh và kiểm chứng mã nguồn C, được xây dựng trên nền tảng CPROVER. 2LS đạt độ chính xác đến mức bit, nó có thể chứng minh hay bác bỏ các điều kiện được đặt ra cho chương trình. 2LS áp dụng các kỹ thuật sinh bất biến cho vòng lặp, mô hình kiểm chứng giới hạn tăng dần và đầu vào tăng dần. 2LS sử dụng một thuật toán kết hợp ba kỹ thuật được gọi là kIki (*k-invariants* và *k-induction*)

3.4.2. Kiến trúc



Hình 19: Kiến trúc của 2LS [12]

Giai đoạn tiếp nhận và xử lý đầu vào (*Front End*). Mô-đun đầu tiên của CBMC đó là giao diện người dùng thông qua dòng lệnh (*Command Line Front End*), tại đây 2LS sẽ được cấu hình theo những tham số mà người dùng đưa ra. Trình phân tích mã nguồn C (*C Parser*) sử dụng một bộ xử lý C có sẵn (ví dụ như *gcc -E*) để xây dựng một cây phân tích từ dữ liệu được tiền xử lý. Được xây dựng trên nền tảng CPROVER, 2LS sử dụng các chương trình GOTO (*GOTO Programs*) làm đại diện trung gian. Trong ngôn ngữ này, tất cả các luồng điều khiển phi tuyến tính, vòng lặp và câu lệnh nhảy được chuyển đổi thành các câu lệnh goto có chức năng tương đương. Cũng giống như CBMC, 2LS thực hiện một phân tích tĩnh để xử lý các con trỏ hàm đến một trường hợp phân tách trên toàn các hàm, kết quả là thu được một đồ thị gọi hàm tĩnh. Cùng với đó, các điều kiện được đặt vào để đảm bảo cho việc các con trỏ không hợp lệ hay rò rỉ bộ nhớ xảy ra.

Giai đoạn xử lý trung tâm (*Middle End*). 2LS thực hiện phân tích tĩnh để lấy được luồng dữ liệu cho mỗi hàm của chương trình GOTO. Kết quả là dạng tĩnh của những câu lệnh gán đơn giản (*Static Simple Assignment - SSA*) trong đó các vòng lặp đã được cắt đi các cạnh nối lên đầu vòng lặp. Do đó SSA là xấp xỉ gần đúng cho chương trình GOTO, sau đó 2LS tinh chỉnh phép tính gần đúng này bằng các bước tính toán và biến đổi. Truyền hằng số và đơn giản biểu thức là chìa khóa để đạt được sự hiệu quả và ngăn chặn việc thực thi những nhánh không khả thi.

Giai đoạn cuối cùng (*Back End*). 2LS yêu cầu bộ giải có khả năng xử lý gia tăng. Vì khả năng xử lý gia tăng của SMT bị tụt lại phía sau so với các SAT Solver nên 2LS sử dụng bộ giải Glucose. Do đó, cũng giống như CBMC, biểu thức SSA sẽ được chuyển đổi thành một công thức dạng chuẩn tắc hội (CNF) bằng cách mô hình với độ chính xác mức bit. Công thức này được mở rộng từng bước để thực hiện sinh bất biến vòng lặp bằng cách sử dụng tổng hợp dựa trên mẫu. Một mô hình được tính toán bởi bộ giải SAT tương ứng với một đường thi hành vi phạm ít nhất một trong các điều kiện trong chương trình được xem xét một cách tỉ mỉ, và cùng với đó một phản ví dụ cụ thể được đưa ra được xem như là bằng chứng cho sự vi phạm điều kiện này. Ngược lại, nếu công thức cuối cùng không thỏa mãn thì không có điều kiện nào bị vi phạm trong giới hạn số lần lặp được xác định từ trước đó.

3.4.3. Điểm mạnh và điểm yếu

2LS sử dụng kIKI có khả năng đưa ra những bằng chứng cũng như việc bác bỏ dựa vào việc sử dụng thuật toán có độ chính xác bit. Việc bác bỏ thông qua đuổi vòng

lập, bằng chứng có được thông qua sinh bất biến cho vòng lặp và *k-induction*. Sự kết hợp này đem lại hiệu quả cao cho 2LS. Tuy nhiên, một số bài toán có chứa những kiểu dữ liệu phức tạp như mảng hoặc danh sách liên kết, đòi hỏi những yêu cầu xử lý cao hơn thì 2LS vẫn đang bị hạn chế trong việc xử lý.

Tại cuộc thi SV-COMP, 2LS có thành tích đứng đầu trong hạng mục floats-cdfpl năm 2016 và đứng thứ 4 năm 2017.

3.4.4. So sánh với VTSE

Có thể thấy rằng 2LS là công cụ được cải tiến từ công cụ CBMC, áp dụng thêm các thuật toán xử lý tối ưu ràng buộc. Cũng giống như CBMC, 2LS sử dụng kỹ thuật kiểm chứng mô hình nhằm thi hành tất cả các đường thi hành có thể có của chương trình. Vấn đề bùng nổ đường thi hành đã được cải tiến tuy nhiên chưa triệt để bằng các công cụ kiểm chứng áp dụng kỹ thuật thực thi tượng trưng như VTSE.

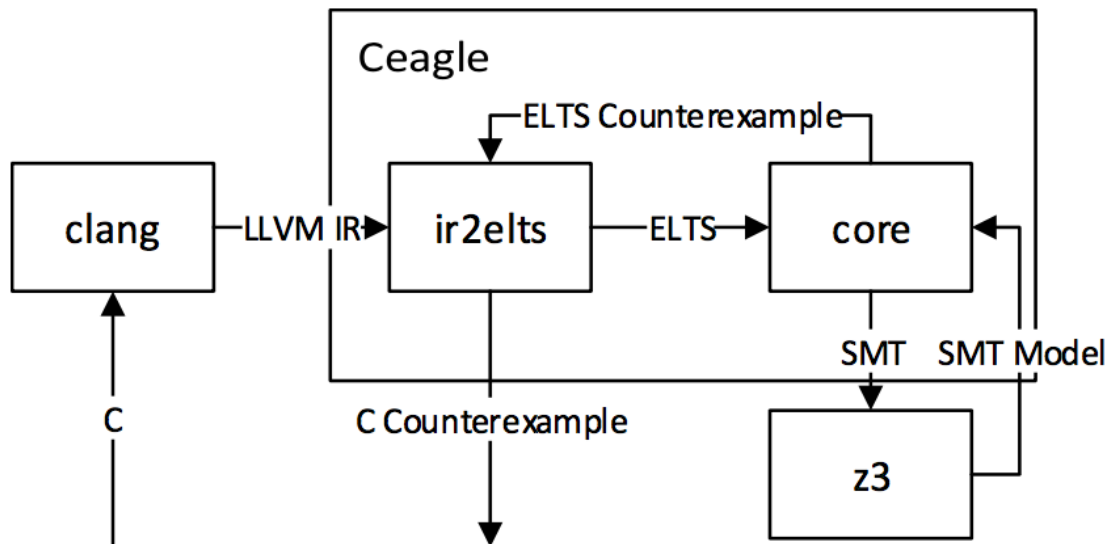
3.5. Ceagle

3.5.1. Tổng quan

Ceagle là một công cụ kiểm chứng cho chương trình mã nguồn C đạt được độ cân bằng cao giữa hiệu năng và độ chính xác mà không cần nhiều thao tác của người sử dụng. Nó đạt độ chính xác và hiệu quả cao là do sử dụng một mô hình ngôn ngữ tương đương về ngữ nghĩa và đặc biệt được thiết kế cho chương trình mã nguồn C, cùng với đó là sự tối ưu hoá đa dạng giúp việc kiểm tra tính thoả mãn của biểu thức một cách nhanh hơn.

3.5.2. Kiến trúc

Mô hình ngôn ngữ giúp cho việc mở rộng hệ thống chuyển tiếp được gán nhãn (*Labeled Transition System - LTS*) ban đầu với nhiều hơn các loại biến, câu lệnh C, và các hàm được dựng sẵn. Hệ thống chuyển tiếp có nhãn mở rộng (*Extended Labeled Transition System - ELTS*) kế thừa các thuộc tính bên trong của LTS truyền thống phù hợp cho việc thực hiện các thuật toán kiểm tra mô hình khác nhau. Dựa trên ELTS, một số hoạt động tối ưu hoá được thực hiện trong việc phân tích chương trình C, xây dựng và phân tích ELTS giúp cho Ceagle nhanh hơn và cải thiện về bộ nhớ hơn so với các công cụ kiểm chứng C khác.



Hình 20: Kiến trúc của Ceagle [13]

Kiến trúc của Ceagle được mô tả trong hình trên. Ceagle gồm bốn thành phần chính: *clang*, *ir2elts*, *core*, và *z3*. Chương trình C đầu tiên được phân tích bởi LLVM *clang*, tạo ra chương trình LLVM IR. Sau đó, thành phần *ir2elts* chuyển chương trình LLVM IR được sinh ra thành chương trình mô hình ELTS. Cuối cùng, thành phần *core* dựng nên các ràng buộc SMT từ ELTS, và đưa chúng làm đầu vào của của thành phần *z3* cho việc kiểm chứng. Nếu thuộc tính an toàn không thỏa mãn, một phản ví dụ sẽ được sinh ra và được thể hiện tự động bằng việc thực hiện công cụ truy vết. Trong toàn bộ quá trình thực hiện, đại diện đặc trưng LLVM IR và chương trình mô hình ELTS đóng vai trò quan trọng trong việc tối ưu cấp độ mã nguồn và cấp độ mô hình tương ứng.

Tất cả các mô hình chuyển đổi và xây dựng đều được thực hiện tự động, người dùng chỉ cần đưa vào mã nguồn chương trình C cùng với đó là điều kiện xác nhận an toàn tương ứng. Nó rất dễ sử dụng và không cần thao tác phức tạp hay nền tảng kiến thức về kiểm chứng.

3.5.3. Thiết kế của Ceagle

LLVM clang: Thành phần này được sử dụng chuyển đổi chương trình C thành LLVM IR.

Translator ir2elts: Thành phần này chuyển đổi chương trình LLVM IR thành mô hình ELTS.

Formalizer core: Thành phần này sinh ra các ràng buộc SMT dựa trên mô hình ELTS và các điều kiện.

Solver z3: Thành phần này kiểm tra sự thoả mãn của các biểu thức ràng buộc, các điều kiện bị vi phạm được xác định bởi: (i) nếu kết quả là thoả mãn với mô hình SMT, điều đó đồng nghĩa tồn tại giá trị của các biến mà làm cho chương trình thực thi điều kiện vi phạm, chính vì thế mà kiểm chứng được rằng điều kiện đã bị vi phạm; (ii) nếu kết quả là không thoả mãn, đồng nghĩa với đường thi hành hiện tại là an toàn; (iii) nếu tất cả các đường thi hành đều cho kết quả không thoả mãn, chương trình được kiểm chứng là an toàn.

3.5.4. Điểm mạnh và điểm yếu

Ceagle sử dụng tối ưu hoá công cụ theo từng cấp bậc (chương trình LLVM IR, mô hình ELTS và công thức ràng buộc) nên tốc độ đưa ra kết quả kiểm chứng một chương trình C là rất tốt. Việc tối ưu hoá ở nhiều cấp độ còn giảm bộ nhớ sử dụng, dẫn đến Ceagle có thể áp dụng cho nhiều chương trình mã nguồn C có số lượng dòng lệnh rất lớn (lên đến 17.000 dòng lệnh). Tuy nhiên, do kiểm chứng theo từng đường thi hành có thể của chương trình nên chính vì vậy, đối với những chương trình có nhiều vòng lặp hoặc nhiều nhánh phức tạp, Ceagle có thể gặp tình trạng bùng nổ trạng thái và không giải quyết được. Bên cạnh đó những câu lệnh phức tạp và các dữ liệu đặc biệt như mảng hay cấp phát bộ nhớ động cũng đang là vấn đề chưa giải quyết được đối với Ceagle.

CHƯƠNG 4. THỰC NGHIỆM

Chương này sẽ đề cập đến hai phần chính đó là xây dựng công cụ VTSE và đánh giá VTSE dựa trên các bộ thực nghiệm đã có sẵn so với các công cụ khác. Từ kết quả thực nghiệm, chúng ta sẽ có thể thấy được hiệu quả của công cụ VTSE so với những công cụ kiểm chứng khác, từ đó có thể thấy được phương pháp xây dựng nên công cụ VTSE là một phương pháp tốt.

4.1. Xây dựng công cụ VTSE

Công cụ kiểm chứng tính chất của mã nguồn C/C++ được xây dựng trên phương pháp thực thi tương trưng với quy trình đã trình bày ở mục 2. Công cụ được viết bằng ngôn ngữ Java. VTSE sử dụng bộ giải ràng buộc Z3 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 CDT để phân tích mã nguồn và sinh ra cây cú pháp trừu tượng. Trình tự hoạt động của VTSE được thực hiện theo các bước:

- **Bước 1:** Nhận đầu vào của công cụ là một tập tin mã nguồn C/C++ và điều kiện do người dùng đưa vào. Đây là đầu vào để xác định các điểm đặt các câu lệnh điều kiện trong chương trình
- **Bước 2:** Tạo cây cú pháp trừu tượng AST sử dụng thư viện CDT.
- **Bước 3:** Xây dựng CFG từ AST đã sinh ra.
- **Bước 4:** Xử lý các vòng lặp và câu lệnh nhảy có trong CFG
- 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

Với các chương trình chứa nhiều phương thức có lời gọi đến nhau, thực hiện nối các CFG riêng rẽ của từng phương thức vào thành một CFG toàn cục của cả chương trình.

- **Bước 5:** Biểu diễn chương trình dưới dạng metaSMT từ đồ thị luồng điều khiển đã thực hiện. Ở bước này ta tiến hành duyệt đồ thị luồng điều khiển và đánh chỉ số, đồng bộ chỉ số các biến.
- **Bước 6:** Biểu diễn chương trình dưới dạng logic vị từ và đưa vào bộ giải SMT sau đó đưa ra báo cáo về các tính chất của chương trình.

4.2. Kết quả thực nghiệm

4.2.1. Điều kiện thực nghiệm

Trong bài khoá luận này sẽ trình bày kết quả thực nghiệm của VTSE với các công cụ khác dựa trên hai bộ dữ liệu thực nghiệm, một được lấy từ cuộc thi SV-COPM là ReachSafety/Floats-cdfpl và bộ thực nghiệm còn lại là bộ dữ liệu thực nghiệm Kratos.

Floats-cdfpl là một bộ dữ liệu của thuộc thi SV-COMP và được viết bằng ngôn ngữ C. SV-COMP (Competition on Software Verification) là một cuộc thi uy tín được tổ chức hằng năm dành cho các công cụ kiểm chứng nhằm mục đích tìm ra công cụ có khả năng giải quyết được nhiều các bài toán với thời gian ngắn nhất. SV-COMP có rất nhiều các bài toán khác nhau, tuy nhiên, do khả năng của công cụ VTSE chưa được mở rộng nên báo cáo khoá luận chỉ lựa chọn bộ dữ liệu Floats-cdfpl làm bộ dữ liệu thực nghiệm cho VTSE.

Bộ dữ liệu thực nghiệm Floats-cdfpl bao gồm 40 bài toán khác nhau, dữ liệu trong các bài toán là các dữ liệu phi tuyến tính, được chia làm 3 chủ đề đó là:

- Các bài toán về khai triển hàm sine
- Các bài toán về khai triển hàm căn bậc hai
- Các bài toán về công thức tính xấp xỉ Newton của hàm sine

Bộ dữ liệu thực nghiệm Kratos được phát triển bởi FBK-IRST [2], nhằm mục đích làm dữ liệu thực nghiệm cho khả năng xử lý các chương trình lớn của các công cụ kiểm chứng. Các bài toán trong bộ dữ liệu thực nghiệm này đa phần đều là những chương trình có số dòng lệnh lớn và rất lớn từ 500-2000 dòng lệnh mỗi chương trình. Các công cụ kiểm chứng khi gặp những bài toán có số dòng lệnh lớn thường bị tràn bộ nhớ hoặc rơi vào trạng thái bế tắc do bùng nổ trạng thái, chính vì vậy bài khoá luận này quyết định chọn bộ dữ liệu thực nghiệm Kratos để kiểm tra khả năng của VTSE.

4.2.2. Kết quả thực nghiệm

Phần này sẽ đưa ra kết quả thực nghiệm công cụ VTSE, bao gồm hai phần thực nghiệm so sánh với các công cụ khác trên bộ dữ liệu thực nghiệm Floats-cdfpl và bộ dữ liệu thực nghiệm Kratos. Kết quả thực nghiệm được thực hiện trên CPU Intel i5-5200U 2.2 GHz với RAM 4GB, thời gian để thực hiện 1 bài toán tối đa là 1500 giây.

4.2.2.1. Kết quả bộ thực nghiệm Floats-cdfpl

Bảng 1 cho thấy kết quả thực nghiệm của VTSE đối với bộ thực nghiệm Floats-cdfpl và so với các công cụ khác. Ở đây bài khoá luận so sánh với hai công cụ khác đó là 2LS và CMBC.

Hai công cụ này đều có thành tích cao tại cuộc thi SV-COMP hạng mục Floats-cdfpl, chính vì vậy mà việc so sánh thực nghiệm của VTSE với hai công cụ này sẽ cho chúng ta một kết quả trực quan về khả năng kiểm chứng chương trình C của VTSE.

Công cụ sử dụng để so sánh là các phiên bản tham gia mới nhất tham gia dự thi SV-Comp 2017: CBMC-5.8 và 2LS-0.5.

Khi thực hiện thực nghiệm, trong bài khoá luận này đặt thời gian dừng của cả 3 chương trình là 1500s, cả 3 đều chạy các tùy chọn mặc định của công cụ. Kết quả được đưa ra có thể là các trường hợp: *safe*, *unsafe* và *timeout* tương đương với đánh giá chương trình là an toàn, vi phạm điều kiện cho trước hoặc không cho ra kết quả trong thời gian giới hạn.

Khi so sánh với CBMC và 2LS, VTSE là công cụ duy nhất không chạy được cả bộ 40 bài toán khi cho ra kết quả 31/40 chương trình. Tuy nhiên, khi xét đến yếu tố thời gian, VTSE lại cho ra kết quả nổi bật so với hai công cụ còn lại ở những bài toán giải được.

Ngoài các chương trình từ *newton_3_1_*_unreach_call* đến *newton_3_8_false_unreach_call* VTSE chưa cho ra kết quả trong 1500s, còn có các bài toán CBMC và 2LS mất thời gian lâu để giải mà VTSE có thể xử lý với tốc độ rất nhanh. Điển hình như bài toán *sine_4_true_unreach_call*, trong khi CBMC và 2LS mất lần lượt 1249.95s và 70.837s để giải, VTSE cho ra kết quả trong 0.011s. Có thể thấy rằng, việc áp dụng kỹ thuật thực thi tượng trưng so với kỹ thuật kiểm chứng mô hình cho thấy sự hiệu quả trong thời gian xử lý vượt trội của VTSE.

4.2.2.2. Kết quả bộ thực nghiệm Kratos

Bảng 2 cho thấy kết quả thực nghiệm của VTSE đối với bộ dữ liệu thực nghiệm Kratos. Các bài toán trong bộ thực nghiệm Kratos đều là những chương trình có số dòng lệnh lớn, luồng điều khiển phức tạp do chứa nhiều câu lệnh điều kiện, vòng lặp và câu lệnh nhảy goto.

Với bộ thực nghiệm Kratos, báo cáo khoá luận so sánh VTSE với công cụ CBMC (do 2LS không hỗ trợ bộ thực nghiệm này). Cả hai công cụ được đặt thời gian chạy tối đa là 1500(s), với số lần đuổi vòng lặp lần lượt là 3, 5 và 10 lần.

Kết quả thực nghiệm cho thấy, VTSE cho rất kết quả rất nhanh. Với số các lần lặp khác nhau, VTSE đều cho kết quả tốt hơn so với CBMC, cụ thể với 3 lần lặp VTSE giải được 16 bài toán, trong khi CBMC chỉ giải được 9 bài toán, với 5 lần lặp VTSE giải được 13 bài toán, CBMC chỉ giải được 7 bài toán và với 10 lần lặp VTSE giải được 8 bài toán còn CBMC là 4 bài toán. Tuy nhiên đối với các bài toán giải được, thời gian trung bình để giải một bài toán của VTSE lại cao hơn so với CBMC. Nhưng số lượng bài toán trong bộ thực nghiệm Kratos mà VTSE có thể giải được là một kết quả rất tốt, điều đó cho thấy rằng VTSE có khả năng giải được các chương trình có số dòng lệnh rất lớn (lên đến 2000 dòng lệnh).

Bảng 1. So sánh VTSE với các công cụ khác dựa trên bộ thực nghiệm Floats-cdfpl

STT	Chương trình	LOC	VTSE		CBMC		2LS	
			Kết quả	Thời gian	Kết quả	Thời gian	Kết quả	Thời gian
1	newton_1_1_true_unreach_call	48	S	0.032	S	330.324	S	26.052
2	newton_1_2_true_unreach_call	48	S	0.029	S	229.267	S	32.393
3	newton_1_3_true_unreach_call	48	S	0.019	S	493.696	S	28.681
4	newton_1_4_false_unreach_call	48	U	0.018	U	20.559	U	9.481
5	newton_1_5_false_unreach_call	48	U	0.024	U	17.465	U	16.517
6	newton_1_6_false_unreach_call	48	U	0.017	U	9.142	U	9.499
7	newton_1_7_false_unreach_call	48	U	0.027	U	6.162	U	3.602
8	newton_1_8_false_unreach_call	48	U	0.038	U	11.217	U	12.596
9	newton_2_1_true_unreach_call	48	S	0.56	S	273.607	S	123.449
10	newton_2_2_true_unreach_call	48	S	0.528	S	343.148	S	98.819
11	newton_2_3_true_unreach_call	48	S	0.541	S	412.942	S	83.836
12	newton_2_4_true_unreach_call	48	S	0.503	S	371.692	S	119.9
13	newton_2_5_true_unreach_call	48	S	0.515	S	747.501	S	193.09
14	newton_2_6_false_unreach_call	48	U	0.58	U	169.122	U	18.057
15	newton_2_7_false_unreach_call	48	U	0.518	U	171.063	U	10.827
16	newton_2_8_false_unreach_call	48	TO	0.785	U	110.768	U	34.741
17	newton_3_1_true_unreach_call	48	TO	1500	S	644.915	S	232.768
18	newton_3_2_true_unreach_call	48	TO	1500	S	689.052	S	244.979
19	newton_3_3_true_unreach_call	48	TO	1500	S	646.655	S	319.251
20	newton_3_4_true_unreach_call	48	TO	1500	S	751.86	S	291.928
21	newton_3_5_true_unreach_call	48	TO	1500	S	702.424	S	310.889
22	newton_3_6_false_unreach_call	48	TO	1500	U	50.856	U	288.599
23	newton_3_7_false_unreach_call	48	TO	1500	U	617.201	U	157.526
24	newton_3_8_false_unreach_call	48	TO	1500	U	309.053	U	97.281
25	sine_1_false_unreach_call	32	U	0.021	U	1.484	U	1.783
26	sine_2_false_unreach_call	32	U	0.024	U	8.683	U	2.377
27	sine_3_false_unreach_call	32	U	0.024	U	6.171	U	1.985
28	sine_4_true_unreach_call	32	S	0.011	S	1249.95	S	70.837
29	sine_5_true_unreach_call	32	S	0.02	S	74.886	S	7.463
30	sine_6_true_unreach_call	32	S	0.02	S	31.956	S	6.343
31	sine_7_true_unreach_call	32	S	0.01	S	3.807	S	6.343
32	sine_8_true_unreach_call	32	S	0.02	S	214.917	S	4.808
33	square_1_false_unreach_call	33	U	0.012	U	3.268	U	4.923
34	square_2_false_unreach_call	33	U	0.012	U	3.162	U	2.749
35	square_3_false_unreach_call	33	U	0.022	U	27.049	U	2.779
36	square_4_true_unreach_call	33	S	0.02	S	275.889	S	714.668
37	square_5_true_unreach_call	33	S	0.012	S	334.017	S	10.394
38	square_6_true_unreach_call	33	S	0.011	S	203.2	S	234.158
39	square_7_true_unreach_call	33	S	0.01	S	216.98	S	38.928
40	square_8_true_unreach_call	33	S	0.02	S	1.568	S	2.635
	Tổng thời gian			4.218		6263.894		1899.972
	Thời gian trung bình			0.136		202.061		61.289

Bảng 2. So sánh VTSE với các công cụ khác dựa trên bộ thực nghiệm Kratos

ID	Chương trình	V	LOC	3 vòng lặp				5 vòng lặp				10 vòng lặp			
				VTSE		CBMC		VTSE		CBMC		VTSE		CBMC	
1	pc_sfifo_1	S	360	S	0,15	S	0,095	S	3,5	S	2,571	S	8,7	S	44,271
2	pc_sfifo_2	S	465	S	0,04	S	0,096	S	1,9	S	5,298	S	3,5	S	84,184
3	token_ring_1	S	459	S	0,11	S	1,847	S	3,092	S	5,371	S	8,7	S	28,848
4	token_ring_2	S	582	S	0,11	S	1,155	S	5,321	S	4,934	S	80,5	S	212,87
5	token_ring_3	S	705	S	0,46	S	1,11	S	9,152	S	9,422	TO	1500	O	--
6	token_ring_4	S	828	S	1,09	S	1,504	S	20,454	S	14,5	TO	1500	O	--
7	token_ring_5	S	951	S	1,05	S	2,26	S	97,326	S	22,94	TO	1500	O	--
8	token_ring_10	S	1566	S	6,78	S	9,239	S	1078,66	TO	1500	O	--	O	--
9	token_ring_12	S	1812	S	43,79	S	14,933	TO	1500	O	--	O	--	O	--
10	transmitter_1	U	437	U	0,96	S*	0,199	U	1,14	S*	2,32	U	2,95	S*	19,425
11	transmitter_2	U	557	U	1,8	S*	0,419	U	4,844	S*	2,771	U	17,42	S*	27,807
12	transmitter_3	U	679	U	5,6	S*	0,772	U	5,716	S*	3,261	U	111,31	O	--
13	transmitter_4	U	801	U	7,4	S*	1,306	U	9,917	S*	5,348	U	147,89	O	--
14	transmitter_5	U	923	U	13,9	S*	2,045	U	54,237	S*	8,487	TO	1500	O	--
15	transmitter_11	U	1655	U	106,6	S*	12,483	TO	1500	O	--	TO	1500	O	--
16	transmitter_12	U	1777	U	1014,9	S*	16,79	TO	1500	O	--	TO	1500	O	--
Tổng thời gian (các bài toán giải đúng)					1204,7		32,239		1295,26		87,22		380,97		417,4
Số bài giải được					16		9		13		7		8		4

Chú thích:

LOC: line of codes - số dòng lệnh trong chương trình

S: safe; U: unsafe; T: Timeout; O: Out of memory; S: sai so với kết quả từ cuộc thi*

V: Kết quả chính thức từ cuộc thi

Từ bảng thực nghiệm trên Floats-cdfpl, chúng ta có thể thấy rằng VTSE thực hiện rất tốt đối với những bài toán có dữ liệu là phi tuyến tính, thời gian chạy của VTSE là tốt hơn rất nhiều so với các công cụ khác. Tuy nhiên, với các bài toán mà VTSE chưa thể giải quyết đó là do tràn bộ nhớ, các bộ giải mà VTSE hiện đang sử dụng chưa thể giải quyết với dữ liệu cần bộ nhớ lớn như vậy.

Kết quả thực nghiệm cũng cho thấy VTSE thực hiện rất tốt các bài toán trong bộ thực nghiệm Kratos, có thể giải được các bài toán này trong thời gian hữu hạn. Tuy thời gian xử lý không được nhanh nhưng có thể thấy rằng VTSE hoàn toàn có khả năng xử lý được các chương trình có số lượng dòng lệnh từ lớn đến rất lớn.

4.2.3. Hướng phát triển

Qua các kết quả thực nghiệm trên hai bộ dữ liệu đã nêu trên, chúng ta có thể thấy rằng VTSE đã đạt được một số kết quả rất tốt. Công cụ đã có thể giải quyết được gần hết các bài toán trong một bộ dữ liệu thực nghiệm của SV-COMP và cho kết quả tốt hơn hẳn các công cụ đã tham gia và đạt giải cao trong cuộc thi này. Ngoài ra, VTSE có khả năng thực hiện được các bài toán có số lượng dòng lệnh lớn đến cực lớn.

Trong tương lai, VTSE sẽ được phát triển để có thể giải quyết được nhiều bộ dữ liệu thực nghiệm trong cuộc thi SV-COMP và sau đó là đăng ký tham dự cuộc thi này. Các vấn đề mà VTSE sẽ giải quyết trong tương lai gần đó là: dữ liệu mảng, dữ liệu văn bản, dữ liệu với bộ nhớ động và xử lý các câu lệnh phức tạp. Ngoài ra, VTSE đang cố gắng sinh bất biến cho vòng lặp một cách chính xác nhất.

KẾT LUẬN

Qua bài báo cáo khoá luận này, ngoài việc tiếp thu được những tri thức quan trọng liên quan đến kiểm chứng chương trình cũng như kỹ thuật thực thi tượng trưng, một kỹ thuật quan trọng được áp dụng rất nhiều trong đảm bảo chất lượng phần mềm thì kết quả khoá luận đã đạt được những điều sau:

- Khoá luận đã đề xuất một phương pháp để kiểm chứng một số tính chất của chương trình dựa trên thực thi tượng trưng. Phương pháp kiểm chứng chương trình dựa trên kỹ thuật thực thi tượng trưng là một kỹ 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.

- Xây dựng được kiến trúc công cụ VTSE thực hiện kiểm chứng chương trình dựa trên phương pháp này. Trong quá trình thực nghiệm, công cụ đã thực hiện trên một số bộ dữ liệu thực nghiệm như Floats-cdfpl và Kratos, thu được kết quả tương đối khả quan, có ưu thế về tốc độ xử lý trong một số trường hợp hơn các công cụ khác như CBMC, 2LS...

Các ứng dụng của VTSE được trình bày trong khoá luận 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, kiểm tra chương trình có chạy vào các điểm đặt trước hay không, kiểm tra khối lệnh không bao giờ thi hành.

Ngoài các ứng dụng được nêu trong bài khoá luận 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*, xử lý dữ liệu mảng, dữ liệu văn bản, dữ liệu với bộ nhớ động.

Kết quả thực thi của VTSE tương đối khả quan, ưu thế về tốc độ mở ra một hướng nghiên cứu mới trong công cuộc kiểm chứng chương trình. So với các công cụ khác tốc độ xử lý của VTSE là tốt hơn. Tuy rằng vẫn còn một số hạn chế chưa giải quyết hết toàn bộ vấn đề, nhưng trong tương lai hạn chế này hoàn toàn có thể được khắc phục trong những phiên bản tiếp theo.

Tài liệu tham khảo

- [1] Dirk Beyer, *Software Verification with Validation of Results (Report on SV-COMP 2017)*, Munich Germany, 2017.
- [2] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, *KRATOS – A Software Model Checker for System C*, 2011.
- [3] Armin Biere, Marijn Heule, Hans van Maaren, Toby Walsch, *Handbook of Satisfiability*, 2008.
- [4] Christel Baier Joost, Pieter Katoen, *Principles of Model Checking*, 2008.
- [5] Leonardo De Moura, Nikolaj Bjørner, *Z3: An Efficient SMT Solver*, 2008.
- [6] Vu Xuan Tung, To Van Khanh, Mizuhito Ogawa, *raSAT: an SMT solver for polynomial constraints*, 2017.
- [7] J. C. King, *Symbolic Execution and Program Testing*, 1976.
- [8] Andrea Janes, Alberto Sillitti, Danila Piatov, Giancarlo Succi, *Using the Eclipse C/C++ Development Tooling as a Robust, Fully Functional, Actively Maintained, Open Source C++ Parser*, 2012.
- [9] C. Cadar, D. Dunbar, and D.R., *KLEE: Unassisted and Automatic Generation of High Coverage Tests for Complex Systems Programs*, Proc. 8th USENIX Symp. Operating Systems Design and Implementation (OSDI 08), 2008, pp. 209-224.
- [10] Clark Barrett, Wei Wang, Thomas Wies, *Cascade 2.0*, 2014.
- [11] Daniel Kroening and Michael Tautschnig, *CBMC – C Bounded Model Checker*, 2014.
- [12] Daniel Kroening and Peter Schrammel, *2LS for Program Analysis*, 2016.
- [13] Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jiaguang Sun, *C Code Verification based on the Extended Labeled*, 2016.
- [14] Anand Yeolekar and Divyesh Unadkat, *Assertion Checking Using Dynamic Inference*, 2013.
- [15] Cristian Cadar, Koushik Sen, *Symbolic Execution for Software Testing: Three Decades Later*, 2013.
- [16] Milena Vujosevic-Janicic, Viktor Kuncak, *Development and Evaluation of LAV: an SMT-Based Error Finding Platform*, 2012.
- [17] Corina Pasareanu, *Symbolic Execution and Software Testing*, 2013.