ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Thị Vân Anh

KẾT HỢP SINH BẤT BIẾN VÒNG LẶP VÀ KỸ THUẬT THỰC THI TƯỢNG TRƯNG TRONG KIỂM CHỨNG MỘT SỐ TÍNH CHẤT CỦA CHƯƠNG TRÌNH C/C++

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ thông tin

HÀ NỘI - 2019

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Nguyễn Thị Vân Anh

KẾT HỢP SINH BẤT BIẾN VÒNG LẶP VÀ KỸ THUẬT THỰC THI TƯỢNG TRƯNG TRONG KIỂM CHỨNG MỘT SỐ TÍNH CHẤT CỦA CHƯƠNG TRÌNH C/C++

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY
Ngành: Công nghệ thông tin

Cán bộ hướng dẫn: TS. Tô Văn Khánh

HÀ NỘI - 2019

VIETNAM NATIONAL UNIVERSITY, HA NOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Nguyen Thi Van Anh

APPLYING LOOP INVARIANT GENERATION AND SYMBOLIC EXECUTION IN C/C++ SOFTWARE VERIFICATION

BACHELOR'S THESIS
Major: Information Technology

Supervisor: Dr. To Van Khanh

HANOI - 2019

LỜI CẢM ƠN

Lời đầu tiên, em xin tỏ lòng biết ơn chân thành và sâu sắc tới thầy giáo TS.Tô Văn Khánh - người đã hướng dẫn tận tình, động viên và giúp đỡ em trong suốt quá trình thực hiện khóa luận tốt nghiệp này.

Em cũng xin gửi tới quý thầy cô Khoa Công nghệ thông tin nói riêng và trường Đại học Công nghệ - Đại học Quốc gia Hà Nội nói chung, đã tận tình truyền đạt những kiến thức quý báu làm nền tảng vững chắc hỗ trợ em trên con đường học tập và làm việc trong tương lai.

Con xin tỏ lòng biết ơn vô hạn tới bố mẹ và những người thân trong gia đình. Những người đã nuôi con khôn lớn, trưởng thành và là nguồn động viên, điểm tựa vững chắc của con trên con đường học tập.

Cuối cùng, em xin cảm ơn những tới các anh chị và các bạn, đặc biết là tập thể lớp K60CLC đã đồng hành cùng em trong suốt 4 năm học qua. Mọi người đã chia sẻ, giúp đỡ và đưa ra những góp ý chân thành cho em trong học tập và cuộc sống.

TÓM TẮT

Sự phát triển bùng nổ của ngành công nghiệp phần mềm trong những năm gần đây khiến cho vấn đề đảm bảo chất lượng phần mềm ngày càng được chú ý. Thực thi tượng trưng là một kĩ thuật phân tích chương trình cho phép sử dụng đầu vào là các giá trị tượng trưng, thay cho các giá trị cụ thể. Bất biến là công thức đúng trước và trong mỗi lần thực thi của vòng lặp, vì vậy công thức này có thể được áp dụng để chứng minh tính đúng đắn một phần của vòng lặp.

Trong nghiên cứu trước đó, công cụ VTSE sử dụng thực thi tượng trưng để kiểm chứng một số tính chất của chương trình C/C++ đã được cài đặt. Tuy nhiên, với phương pháp sử dụng là gỡ vòng lặp với số lần nhất định, VTSE chỉ có thể kiểm tra chương trình là đúng với số vòng lặp đã gỡ. Điều này cũng khiến VTSE có kết quả chưa tốt trên các bài toán có số gỡ lần lặp lớn hoặc không có giới hạn vòng lặp. Khóa luận này xin trình bày hướng tiếp cận xử lý vòng lặp trong kiểm chứng bằng cách kết hợp bất biến vòng lặp vào kĩ thuật thực thi tượng trưng.

Bất biến vòng lặp được sinh ra sử dụng bổ đề Farkas và một công cụ con - InvaGen áp dụng phương pháp này cũng được cài đặt. Sau đó, InvaGen được tích hợp vào công cụ kiểm chứng cũ để xây dựng phiên bản VTSEinv. Kết quả thực nghiệm của VTSEinv so sánh với phiên bản VTSE trước đây cùng hai công cụ CBMC và Veriabs là khả quan khi có số lượng bài giải tốt và trong thời gian nhanh hơn rất nhiều, đặc biệt là các bài toán có giới hạn vòng lặp lớn.

Từ khóa: *kiểm chứng phần mềm, thực thi tượng trưng, bất biến vòng lặp, phân tích mã nguồn*

ABSTRACT

Recent years, software quality assurance has evolved and become increasingly important, especially with classified safety- or business-critical software. Symbolic Execution is a program analysis technique that allows the execution of a program using symbolic values instead of concrete ones. A loop invariant is a formula that is true immediately before and after each iteration of a loop, therefore can be used to prove the partial correctness of the loop.

In earlier work, a tool called VTSE using Symbolic Execution technique to verify some properties of C/C++ program was implemented. However, because of using the unfolding method when handling loops, VTSE can only ensure the program is correct with a fixed number of loops and also have disadvantages if a high number of loop unfolding is required. This thesis introduces an approach to handle loops in verification by combining Loop Invariant and Symbolic Execution technique.

Loop invariants are generated using the method which is Farkas' Lemma and a tool InvaGen is implemented. The verification tool VTSEinv is improved by incorporating with the use of the loop invariant by integrating with InvaGen. Experiments are performed on a set of benchmarks by using loop invariant. Results are compared with VTSE using the unfolding method and two other tools from SV-COMP are CBMC and VeriAbs. VTSEinv when using invariant can solve more tasks and produces results in a much shorter time than other tools.

Keywords: *software verification, symbolic execution, loop invariant, source code analyzing*

LỜI CAM ĐOAN

Tôi xin cam đoan rằng những nghiên cứu về phương pháp kết hợp bất biến vòng lặp và kỹ thuật thực thi tượng trưng để kiểm chứng chương trình phần mềm được trình bày trong khóa luận này là của tôi và chưa từng được nộp như một báo cáo khóa luận tại trường Đại học Công Nghệ - Đại học quốc gia Hà Nội hoặc bất kỳ trường đại học khác. Những gì 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. Tôi xin cam đoan công cụ tôi trình bày trong khoá luận là do tôi tự phát triển, không sao chép mã nguồn của người khác. Nếu sai tôi hoàn toàn chịu trách nhiệm theo quy định của trường Đại Học Công Nghệ - Đại Học Quốc Gia Hà Nội.

Hà Nội, ngày 22 tháng 05 năm 2019

Sinh viên

Nguyễn Thị Vân Anh

Mục lục

Danh sách chữ viết tắt

AST	Abstract Syntax Tree
CFG	Control Flow Graph
SMT	Satisfiability Modulo Theories
SV-COMP	Software Verification Competition
VTSE	Verification Tool based on Symbolic Execution

Danh sách bảng

Danh sách hình vẽ

Chương 1

Mở đầu

1.1 Đặt vấn đề

Sự phát triển bùng nổ của ngành công nghiệp phần mềm trong những năm gần đây khiến cho vấn đề đảm bảo chất lượng phần mềm ngày càng được chú ý. Trong đó, kiểm thử là một phương pháp hữu hiệu để kiểm tra sự đúng đắn của chương trình, tuy nhiên, kiểm thử chỉ thi hành một tập nhỏ đường thực thi với các đầu vào là giá trị cụ thể. Điều đó có thể dẫn đến việc bỏ sót nhiều lỗi của chương trình khi thực hiện kiểm thử. Vì vậy, để chứng minh chương trình là không có lỗi, các phương pháp kiểm chứng phần mềm được ứng dụng.

Trong kiểm chứng chương trình, vòng lặp là một trong những vấn đề thách thức nhất. Việc xác định số lần lặp là không đơn giản và con số này còn phụ thuộc vào dữ liệu đầu vào. Một giải pháp cho vấn đề này là sử dụng bất biến vòng lặp. Bất biến là một công thức logic thể hiện các tính chất đúng với mọi lần thực thi của vòng lặp. Công thức này cung cấp thông tin cơ bản, thể hiện được mục đích và nội dung chính của vòng lặp là gì. Một bất biến đủ mạnh có thể giúp tránh được việc gỡ vòng lặp trong phương pháp kiểm chứng mô hình *(Model Checking)* [4], mà cụ thể là trong thực thi tượng trưng *(Symbolic Execution)* [10].

Chính vì sự quan trọng và khó khăn của bài toán, đã có rất nhiều công trình về tự động sinh bất biến vòng lặp từ mã nguồn chương trình được thực hiện. Những công

trình đầu tiên bắt đầu từ những năm 70. Các phương pháp nổi bật hiện nay trong sinh bất biến vòng lặp bao gồm diễn giải trừu tượng *(Abstract Refinement)* [3], các kĩ thuật giải ràng buộc [8][14], phương pháp CEGAR [9] và phương pháp áp dụng công thức hồi quy [13]. Trong đó, giải ràng buộc là hướng tiếp cận giúp sinh ra công thức bất biến một cách nhanh chóng và hiệu quả. Một trong những đề xuất của phương pháp này là áp dụng bổ đề Farkas để sinh bất biến vòng lặp.

Trong nghiên cứu trước đây, công cụ VTSE sử dụng thực thi tượng trưng để kiểm chứng một số tính chất của chương trình C/C++ đã được giới thiệu. Công cụ cho kết quả tốt trên bộ thực nghiệm của cuộc thi SV-COMP[1] và đồng thời có khả năng thực hiện các bài toán có số dòng lệnh lớn đến cực lớn. Tuy nhiên, với các bài toán có số lần lặp lớn hoặc không giới hạn lần lặp, phương pháp gỡ với số lần nhất định của VTSE cho hiệu quả không cao. Vì vậy, để giải quyết vấn đề này, khóa luận xin đề xuất phương pháp kết hợp bất biến vòng lặp và kĩ thuật thực thi tượng trưng để kiểm chứng chương trình. Bất biến vòng lặp được sinh tự động từ mã nguồn áp dụng bổ đề Farkas, sau đó kết nối bất biến với công cụ VTSE trước đây để tạo nên phiên bản cải tiến VTSEinv[2].

Các phần của khóa luận được cấu trúc như sau. **Chương 1** trình bày mục tiêu của khóa luận và các kiến thức lý thuyết nền tảng. **Chương 2** trình bày phương pháp sinh bất biến và kiến trúc công cụ được cài đặt áp dụng phương pháp này là InvaGen. Tiếp theo, cách thức kết nối công cụ sinh bất biến vào kiểm chứng chương trình được thể hiện ở **Chương 3**. **Chương 4** là các nghiên cứu liên quan đến một số phương pháp xử lý vòng lặp và sinh bất biến khác. **Chương 5** đưa ra kết quả thực nghiệm của VTSEinv so với phiên bản cũ và một số công cụ khác. Cuối cùng là **Kết luận** của toàn bộ khóa luận và công cụ, cũng như trình bày các hướng phát triển tiếp theo.

[1]https://sv-comp.sosy-lab.org/

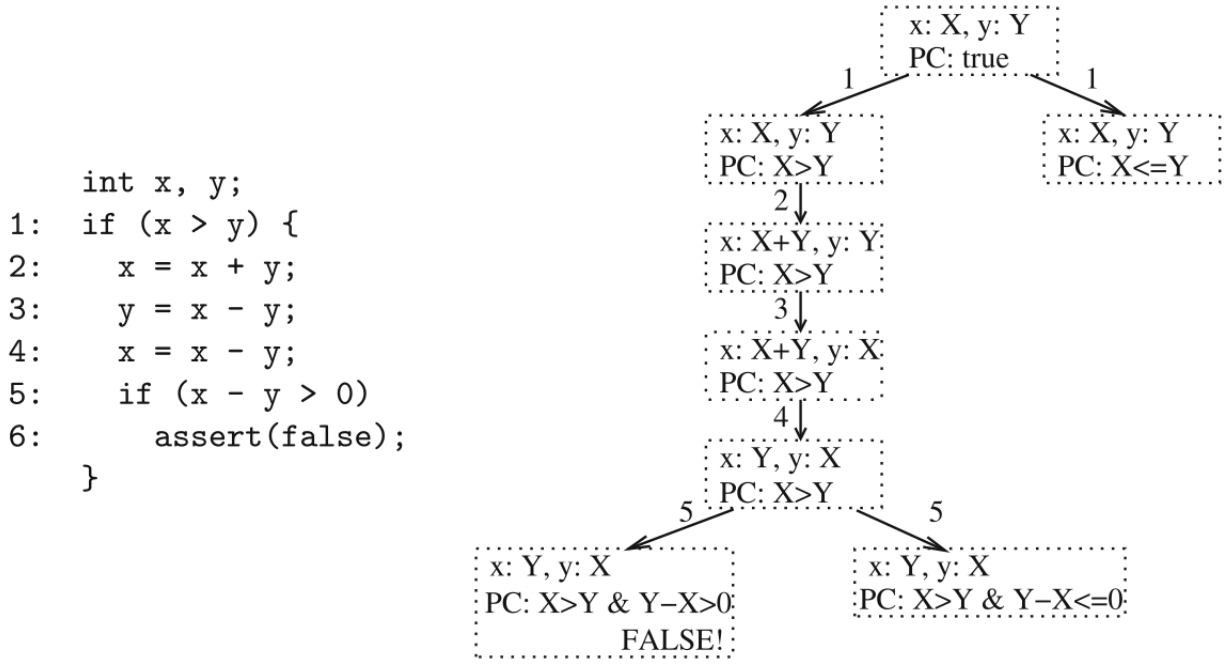
[2]https://github.com/vananhnt/Vtse-VerificationTool/tree/invariant-new

1.2 Kỹ thuật thực thi tượng trưng

Thực thi tượng trưng (*Symbolic Execution*) [10] là một kĩ thuật phân tích chương trình được đề xuất từ những năm 70, tuy nhiên đang ngày càng được quan tâm do sự phổ biến của máy tính và sự phát triển của các thuật toán mới. Trong kĩ thuật này, thay vì đầu vào là các giá trị cụ thể, chương trình được thực thi với các giá trị là tượng trưng. Vì vậy, đầu ra sinh bởi chương trình sẽ được thể hiện dưới dạng một hàm số của các đầu vào tượng trưng. Ứng dụng của thực thi tượng trưng có thể thấy từ sinh kịch bản kiểm thử tự động đến chứng minh tính đúng đắn của chương trình.

Trạng thái của chương trình khi được thực thi tượng trưng bao gồm các giá trị tượng trưng của biến, một đường điều kiện *(path condition - PC)*, và một bộ đếm. Đường điều kiện là một biểu thức đại số không chứa lượng từ dựng trên các giá trị đầu vào tượng trưng. Từ mỗi đường điều kiện ta thu được các ràng buộc mà đầu vào cần thỏa mãn để chương trình có thể thực thi theo đường đó. Một cây thực thi tượng trưng *(symbolic execution tree)* thể hiện các tính chất của đường thực thi. Các nút của cây đại diện các trạng thái của chương trình và được kết nối bởi các chuyển trạng thái *(transition)* của hệ thống.

Xem xét đoạn mã nguồn trong Hình 1.1 (trái), chương trình đổi chỗ giá trị của hai biến số nguyên x và y, khi x lớn hơn y [16]. Hình 1.1 (phải) biểu diễn cây thực thi tương ứng. Khởi tạo, PC là *true* và x, y có giá trị tượng trưng lần lượt là X, Y. Tại mỗi điểm chia nhánh, PC được cập nhật theo các nhánh. Sau khi thực thi *if* tại 1, cây thực thi chia thành hai nhánh *if* và *else*, và PC được cập nhật tương ứng. Với mỗi câu lệnh, trạng thái mới của biến x, y cũng được cập nhật. Nếu một *path condition* là *false*, nghĩa là không có tập đầu vào nào thỏa mãn, trạng thái đấy không bao giờ xảy ra *(unreachable)* thì thực thi trên *path condition* đấy sẽ không được tiếp tục. Theo như ví dụ, tại câu lệnh *if* số 5 thì PC của nhánh *then* là không thỏa mãn, vì vậy câu lệnh 6 sẽ không bao giờ được thực thi vào.



Hình 1.1: Chương trình đổi chỗ hai số và cây thực thi tượng trưng tương ứng

1.3 Bất biến vòng lặp

1.3.1 Logic Hoare

Logic Hoare (Logic Floyd-Hoare) là một hệ chính quy cung cấp một tập các quy tắc suy luận tính đúng đắn của chương trình. Cài đặt của một chương trình là đúng một phần *(partially correct)* so với đặc tả nếu như giả định tiền điều kiện là đúng, thì nếu chương trình kết thúc, hậu điều kiện được thỏa mãn. Một chương trình là đúng hoàn toàn *(totally correct)* khi giả định tiền điều kiện là đúng, chương trình chắc chắn sẽ chấm dứt *(terminate)* và khi nó chấm dứt, thì hậu điều kiện thỏa mãn. Điều đó tương đương, chương trình đúng hoàn toàn khi thỏa mãn luật đúng một phần và chắc chắn chấm dứt.

Một bộ ba Hoare được biểu diễn ở dạng $\{P\}S\{Q\}$, là đúng nếu ta bắt đầu tại trạng thái P là đúng và thực thi S, thì S sẽ kết thúc tại trạng thái mà Q đúng. Theo

luận lý Hoare, tính đúng đắn một phần của luật **While** được định nghĩa như dưới đây:

$$\frac{\{Inv \wedge C\}\ \text{Body}\ \{Inv\}}{\{Inv\}\ \textbf{while}\ \text{C}\ \textbf{do}\ \text{Body}\ \textbf{done}\ \{\neg C \wedge Inv\}}$$

trong đó *Inv* là bất biến vòng lặp, *C* là biểu thức điều kiện và *Body* là các câu lệnh trong thân vòng lặp.

Vì vậy, cần tìm ra bất biến để có thể chứng minh tính đúng một phần của vòng lặp. Tuy nhiên, việc tìm ra bất biến đủ mạnh và phù hợp là một thách thức không dễ dàng. Đồng thời, khóa luận này sẽ chỉ đề cập đến chứng minh vòng lặp là đúng một phần và mặc định chương trình sẽ chấm dứt tại một thời điểm nhất định.

1.3.2 Bất biến vòng lặp

Vòng lặp trong một chương trình có thể được thể hiện với nhiều cấu trúc khác nhau. Dạng *while* sử dụng một điều kiện liên tục và kết thúc khi điều kiện này sai, dạng *do-until* luôn luôn thực thi vòng lặp ít nhất một lần, kiểm tra điều kiện ở cuối thay vì ban đầu, *for* duyệt qua một dãy số hoặc một cấu trúc dữ liệu. Tất cả các dạng đều là biến thể của vòng lặp và có thể dễ dàng chuyển đổi mà không thay đổi ngữ nghĩa. Mỗi vòng lặp trong chương trình đều có chứa các thành phần như dưới đây.

```
1  P; //pre-condition
2  Init;
3  While (Cond) {
4     //invariant Inv
5     Body;
6  }
```

Inv là bất biến của vòng lặp nếu thỏa mãn các điều kiện [7]:

(a) Với mỗi lần *Init* thực hiện, nghĩa là tại trạng thái trước vòng lặp, sẽ dẫn đến trạng thái mà *Inv* thỏa mãn.

(b) Với mỗi lần *Body* được thực thi, bắt đầu tại trạng thái *Inv* và *Cond* thỏa

mãn, sẽ dẫn đến một trạng thái *Inv* tiếp tục thỏa mãn.

Nếu có các tính chất trên, mỗi khi vòng lặp chấm dứt sẽ dẫn đến trạng thái *Inv* và $\neg Cond$. Mỗi vòng lặp được định nghĩa là một thực thi của *Init* và sau đó là không hoặc nhiều hơn thực thi của *Body*, được lặp khi *Cond* vẫn thỏa mãn. Vì vậy, nếu *Init* thỏa mãn bất biến và một lần nào đó *Body* thỏa mãn, thì *Init* theo sau bởi bao nhiêu lần thực hiện của *Body* cũng vậy. Biểu diễn bằng Logic Hoare, ta có bất biến *Inv* là công thức thỏa mãn:

- $\{P\}Init\{Inv\}$, được gọi là bước khởi đầu *(initiation)*
- $\{Iniv \wedge Cond\}Body\{Inv\}$, được gọi là bước quy hồi *(consecution; inductiveness)*.

Bất biến *Inv* không có vai trò ngữ nghĩa trực tiếp nhưng có thể giúp ta hiểu rõ hơn về vòng lặp và tính đúng đắn của vòng lặp.

1.3.3 Bổ đề Farkas

Sau nghiên cứu của Colón và các cộng sự [14] hướng tiếp cận sinh bất biến vòng lặp trên giải ràng buộc đã được phát triển mạnh mẽ. Nổi bật trong số đó là phương pháp áp dụng bổ đề Farkas. Với đầu vào là một hệ thống chuyển đổi trạng thái, bổ đề Farkas sử dụng kĩ thuật tính toán trên ràng buộc để sinh ra các bất biến vòng lặp ở dạng biểu thức tuyến tính.

Hệ thống chuyển trạng thái

Một hệ thống chuyển trạng thái P <V, L, l_0, θ, T> bao gồm một tập hợp các biến V, vị trí đầu l_0, tập hợp vị trí L, điều kiện kiểm tra θ và một tập các chuyển đổi T.

Bổ đề

Xem xét hệ thống S gồm các bất đẳng thức tuyến tính trên tập biến x_1, x_2, ..., x_n:

integer i, j **where** $i = 2 \wedge j = 0$
l_0 : **while true do**
$\begin{bmatrix} \quad i := i + 4 \\ l_1 : \quad \textbf{or} \\ \quad (i, j) := (i + 2, j + 1) \end{bmatrix}$

$L = \{l_0, l_1\},\ V = \{i, j\},$
$\Theta :\ (i = 2 \wedge j = 0),\ \mathcal{T} = \{\tau_0, \tau_1, \tau_2\},$
$\tau_0 :\ \langle l_0, l_1, true \rangle$
$\tau_1 :\ \langle l_1,\ l_0,\ (i' = i + 4 \wedge j' = j) \rangle$
$\tau_2 :\ \langle l_1,\ l_0,\ (i' = i + 2 \wedge j' = j + 1) \rangle$

Hình 1.2: Chương trình và hệ thống chuyển trạng thái tương ứng [14]

$$\text{S:} \begin{bmatrix} a_{11}x_1 + ... + & a_{1n}x_n + & b_1 & \leq 0 \\ \vdots & \vdots & \vdots & \\ a_{m1}x_1 + ... + & a_{mn}x_n + & b_m & \leq 0 \end{bmatrix}$$

Khi S là thỏa mãn, từ S có thể suy ra được bất đẳng thức

$$\psi : c_1x_1 + ... + a_{mn}x_n + b_m \leq 0 \tag{1.1}$$

khi và chỉ khi tồn tại các số thực không âm λ_0,λ_1,...λ_m thỏa mãn

$$c_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \quad ... \quad c_n = \sum_{i=1}^{m} \lambda_i a_{in}, \quad d = (\sum_{i=1}^{m} b_i) - \lambda_0 \tag{1.2}$$

S là không thỏa mãn khi và chỉ khi từ S suy ra bất đẳng thức $1 \leq 0$. Để ứng dụng, bổ đề Farkas có thể được thể hiện dưới dạng bảng như sau:

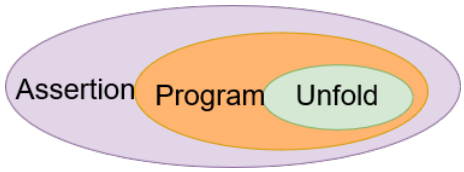
λ_0	$-1 \leq 0$	
λ_1	$a_{11}x_1 + ... + a_{1n}x_n + b_1 \leq 0$	} S
$\vdots$	$\vdots$	
λ_m	$a_{m1}x_1 + ... + a_{mn}x_n + b_m \leq 0$	
	$c_1x_1 + ... + c_nx_b + d \leq 0$	$\leftarrow \psi$
	$1 \leq 0$	$\leftarrow$ disable

Từ đó, ta có thể tìm ra bất biến vòng lặp dưới dạng một biểu thức có dạng:

$$c_1x_1 + c_2x_2 + ... + d \leq 0 \tag{1.3}$$

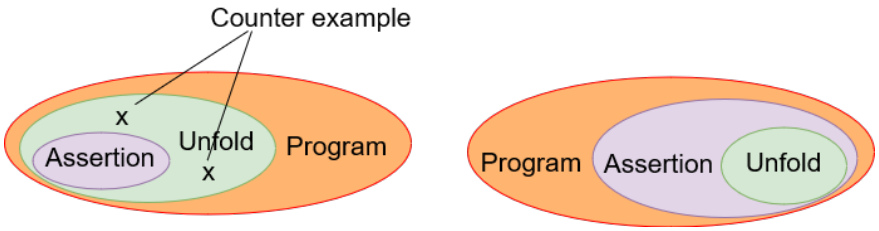
1.4 Đánh giá phương pháp gỡ vòng lặp

Giả sử miền giá trị thỏa mãn chương trình, thỏa mãn biểu thức điều kiện người dùng và biểu thức khi gỡ vòng lặp với số lần nhất định được thể hiện qua biểu đồ Venn dưới đây.



Hình 1.3: Biểu đồ Venn cho bài toán Unfold với kết quả **true**
Program - chương trình, Assertion - điều kiện người dùng, Unfold - gỡ vòng lặp.

Trong trường hợp bài toán cho kết quả đúng **true** (Hình 1.6), có nghĩa là điều kiện người dùng luôn được chương trình thỏa mãn, thì miền giá trị của Program sẽ luôn nằm trong miền Assertion. Ta có Unfold là một xấp xỉ dưới của Program, vậy nên trong trường hợp này ta luôn có thể chứng minh chương trình là **true** khi dùng phương pháp Unfold.



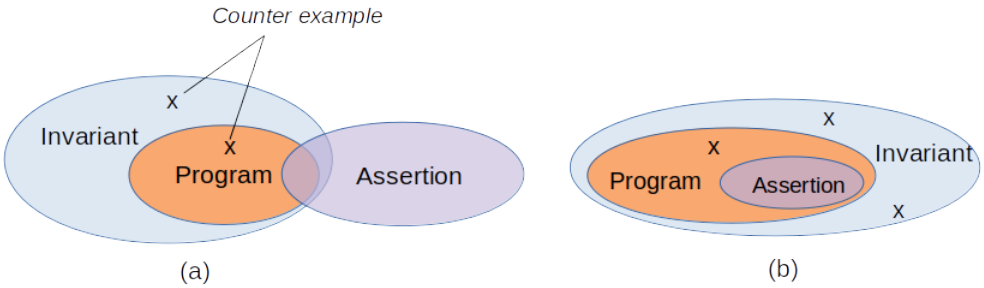
Hình 1.4: Biểu đồ Venn cho bài toán Unfold với kết quả **false**
Program - chương trình, Assertion - điều kiện người dùng, Unfold - gỡ vòng lặp, Counter Example - phản ví dụ.

Trong trường hợp bài toán cho kết quả đúng **false** (Hình 1.4), nghĩa là chương trình có chứa giá trị không thỏa mãn điều kiện người dùng, hay còn được gọi là phản ví dụ. Công thức Unfold là xấp xỉ dưới của Program, vì vậy, với trường hợp (trái) kết quả **false** và phản ví dụ sẽ được xác định đúng. Tuy nhiên với trường hợp (phải) thì

sẽ bị báo kết quả sai *(unsound)*. Chương trình là **false** nhưng khi sử dụng công thức Unfold sẽ đưa ra kết quả là **true**. Đây là lỗi bỏ sót *bug* và rất nghiêm trọng trong kiểm chứng phần mềm.

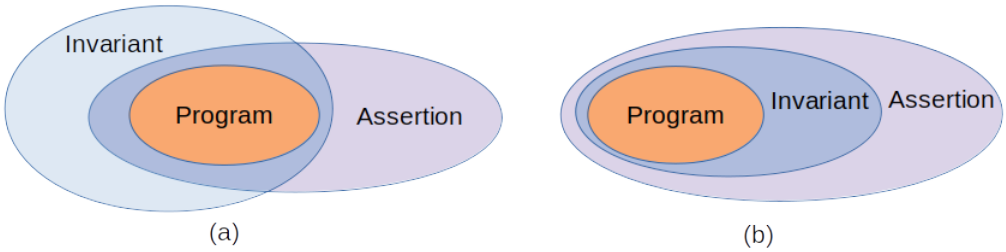
1.5 Đánh giá phương pháp xử lý vòng lặp bằng bất biến

Giả sử miền giá trị thỏa mãn chương trình, thỏa mãn biểu thức điều kiện người dùng và thỏa mãn bất biến vòng lặp lần lượt được thể hiện qua biểu đồ Venn.



Hình 1.5: Biểu đồ Venn cho bài toán Invariant với kết quả **false**
Program - chương trình, Assertion - điều kiện người dùng, Invariant - bất biến, Counter Example - phản ví dụ.

Trong trường hợp bài toán có kết quả đúng là **false** (Hình 1.5). Khi này sẽ có những phản ví dụ là giá trị của Program nằm ngoài Assertion. Do bất biến vòng lặp là một xấp xỉ trên của công thức chương trình, vì vậy ta luôn có thể sử dụng bất biến vòng lặp để chứng minh chương trình là **false**. Tuy nhiên, khi này phản ví dụ được đưa ra sẽ không thể chắc chắn có nằm trong miền giá trị của Program hay không. Vì vậy, khi miền giá trị của Invariant càng sát với Program thì khả năng phản ví dụ đưa ra là giá trị của chương trình càng tăng.



Hình 1.6: Biểu đồ Venn cho bài toán Invariant với kết quả **true**
Program - chương trình, Assertion - điều kiện người dùng, Invariant - bất biến.

Trong trường hợp bài toán cho kết quả đúng **true** (Hình 1.6), ta có Invariant là một xấp xỉ trên của Program, vậy nên với chương trình **true** thì chưa chắc sử dụng bất biến sẽ cho kết quả đúng. Khi miền giá trị của Invariant là quá rộng (Hình 4.2 (a)) thì kết quả trả về sẽ là sai. Vì vậy, việc thu hẹp bất biến sát với chương trình sẽ có thể cho kết quả chính xác hơn. Xét mã nguồn C dưới đây, với một công thức bất biến yếu thì biểu thức điều kiện người dùng sẽ bị vi phạm, chương trình chỉ có thể được kiểm chứng **true** nếu một bất biến đủ mạnh được sinh ra.

```
1  int sumN(int n) {//pre-condition n >= 0;
2     int i = 0; s = 0;
3     while (i <= n) {
4     //strong invariant ((s == i*(i-1)/2) && (i <= n+1))
5     //weak invariant ((s >= 0) && (i <= n + 1))
6        s = s + i;
7        i = i + 1;}
8     return s;
9     VERIFIER_assert(return == n*(n+1)/2);
10    //assertion user want to verify}
```

Có thể nói, việc tìm được bất biến phù hợp và đủ mạnh để kiểm chứng chương trình là một bài toán không xác định *(undecidable)*. Cho đến nay, các phương pháp đã và đang được nghiên cứu đều nhắm tới việc sinh ra được bất biến đủ mạnh cho nhiều bài toán nhất có thể.

Chương 2

Phương pháp sinh bất biến vòng lặp bằng bổ đề Farkas

Trong chương này, khóa luận xin trình bày các bước sinh bất biến vòng lặp bằng bổ đề Farkas ở 2.1 . Đồng thời, công cụ InvaGen áp dụng phương pháp này đã được xây dựng. Phần 2.2 sẽ thể hiện kiến trúc và cách thức cài đặt công cụ này.

2.1 Phương pháp sinh bất biến vòng lặp

Phương pháp sinh bất biến vòng lặp tuyến tính đã được giới thiệu tại [14]. Đầu vào của phương pháp là một hệ thống chuyển đổi trạng thái như đã được trình bày ở Hình 1.2. Ban đầu, coi các bất biến cần tìm là công thức có dạng

$$c_1x_1 + c_2x_2 + .. + d \leq 0$$

mà trong đó, $x_1, x_2, ...$ là các biến xuất hiện trong vòng lặp, $c_1, c_2, ..., d$ là các hệ số cần tìm. Hai điều kiện để tìm ra bất biến quy nạp được thể hiện dưới dạng các bảng tính **Initiation** và **Consecution**. Cụ thể các bảng tính được xây dựng như sau.

Initiation: Tập câu lệnh khởi đầu θ, nghĩa là các câu lệnh đứng trước vòng lặp sẽ được đưa vào bảng tính.

λ_0	$-1 \leq 0$		
λ_1	$a_{11}x_1 + ... + a_{1n}x_n + b_1 \leq 0$	$\Big\}$	θ
$\vdots$	$\vdots$		
λ_m	$a_{m1}x_1 + ... + a_{mn}x_n + b_m \leq 0$		
	$c_1x_1 + ... + c_nx_b + d \leq 0$	$\leftarrow \varphi$	
	$1 \leq 0$	$\leftarrow$ disable	

Trong đó $\lambda_0, \lambda_1, ...\lambda_m$ là các số thực không âm, φ là công thức bất biến thỏa mãn trước khi bước vào vòng lặp.

Consecution: Với mỗi bảng Consecution thì tập câu lệnh được thực hiện trong từng biến đổi trạng thái sẽ được đưa vào bảng tính[1].

μ	$c_1x_1 + ... + c_nx_n$		$+ d_i \leq 0$		
λ_0			$-1 \leq 0$		
λ_1	$a_{11}x_1 + ... + a_{1n}x_n$	$+ a'_{11}x'_1 + ... + a'_{1n}x'_n$	$+b_1 \leq 0$	$\Big\}$	T
$\vdots$	$\vdots$	$\vdots$	$\vdots$		
λ_m	$a_{m1}x_1 + ... + a_{mn}x_n$	$+ a'_{m1}x'_1 + ... + a'_{mn}x'_n$	$+b_m \leq 0$		
		$c_1x'_1 + ... + c_nx'_b$	$+d \leq 0$	$\leftarrow \varphi'$	
			$1 \leq 0$	$\leftarrow$ disable	

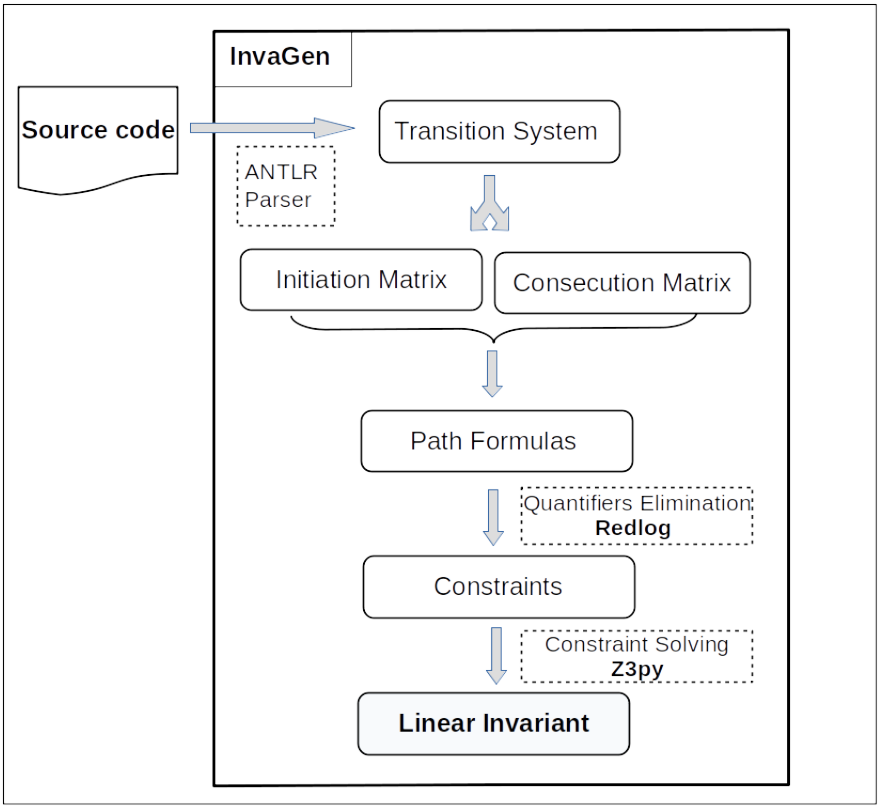
Trong đó $\mu, \lambda_0, \lambda_1, ...\lambda_m$ là các số thực không âm. φ' là công thức bất biến thỏa mãn điều kiện trong vòng lặp.

Từ hai bảng tính trên, ta thu được công thức ràng buộc giữa các hệ số $c_1, c_2, ..., d$. Với mỗi bộ nghiệm hệ số thỏa mãn công thức ràng buộc tìm được, từng công thức bất biến tương ứng được tạo nên. Khi này, bài toán chuyển về thuần tính toán số học. Việc quản lý các bảng tính và hệ số tương ứng có thể chuyển về bài toán xử lý ma trận và giải bất đẳng thức. Chi tiết phương pháp và thuật toán để giải ràng buộc được trình bày ở mục sau.

[1]a, b ký hiệu cho hệ số đã biết, c, d là các giá trị chưa biết

2.2 Xây dựng công cụ sinh bất biến InvaGen

2.2.1 Kiến trúc InvaGen



Hình 2.1: Kiến trúc của InvaGen

Phần này xin trình bày về kiến trúc của InvaGen - công cụ sinh bất biến vòng lặp sử dụng bổ đề Farkas. Kiến trúc này dựa trên phương pháp sinh bất biến tuyến tính bằng việc giải công thức ràng buộc tuyến tính đã được phát triển bởi Colon và các cộng sự [14]. Với đầu vào là mã nguồn C/C++ chứa vòng lặp tuân thủ các *template* đã được định nghĩa, thì đầu ra của InvaGen là các biểu thức bất biến ở dạng tuyến tính $c_1x_1 + c_2x_2 + ... + d \leq 0$. Ban đầu, mã nguồn được chuyển về dạng Hệ thống chuyển trạng thái, tiếp theo ma trận Initiation và Consecution sẽ được dựng lên từ các chuyển đổi trạng thái trên. Bằng việc tính toán trên các ma trận, công thức đại diện cho từng đường thực thi của chương trình được hình thành. Sau khi loại bỏ các

lượng từ, công thức ràng buộc về $c_1, c_2, ..., d$ được tìm ra và đưa vào bộ giải Z3py để sinh ra bộ nghiệm thỏa mãn, từ đó hình thành các bất biến của vòng lặp.

2.2.2 Hệ thống chuyển đổi trạng thái

Mã nguồn của chương trình sẽ được thể hiện dưới dạng Hệ thống chuyển đổi trạng thái (*Transition System*) đã được trình bày tại 1.3.3. Hệ thống chuyển đổi trạng thái là độc lập với ngôn ngữ lập trình và được phân tích sử dụng bộ phân tích cú pháp ANTLR4 [2]. Sau khi phân tích, ta thu được tập các biểu thức thể hiện trạng thái ban đầu θ và tập các chuyển trạng thái tiếp theo T.

Mã nguồn 2.1: Ví dụ file xml biểu diễn Transition System

```
<TransitionSystem>
   <Variables>x, y</Variables>
   <Initiation> x = 2 and y = 0 </Initiation>
   <Consecutions>
      <Transition> x' = x + 4; y' = y </Transition>
      <Transition> x' = x + 2; y' = y + 1 </Transition>
   </Consecutions>
</TransitionSystem>
```

2.2.3 Ma trận trạng thái Initiation và Consecution

Ta dựng lên ma trận trạng thái Initiation cho các trạng thái ban đầu θ và ma trận Consecution cho các chuyển trạng thái tiếp theo thuộc tập T. Trong bài toán này, một bảng tính Initiation và hai bảng tính Consecution được xây dựng. Vì vậy, số ma trận tương ứng sẽ được dùng để quản lý các hệ số trong bảng tính trên.

Ví dụ, với đầu vào là hệ thống được thể hiện ở Mã nguồn 3.1, bảng tính sẽ có giá trị như sau, trong đó $\lambda_0, \lambda_1, \lambda_2 \geq 0$.

[2]https://www.antlr.org/

λ_0					$-1 \leq 0$	
λ_1	x				$-2 = 0$	
λ_2			y		$= 0$	
	$c_1x + c_2y + d \leq 0$					$\leftarrow \varphi$

Khi đó, ma trận hệ số dưới đây của bảng tính được lưu vào hệ thống.

$$\begin{bmatrix} 0 & 0 & -\lambda_0 \\ \lambda_1 & 0 & -2\lambda_1 \\ 0 & \lambda_2 & 0 \end{bmatrix}$$

Do hệ thống có hai chuyển trạng thái, vì vậy hai bảng tính Consecution lần lượt được dựng lên, với $\mu, \lambda_1, \lambda_2 \geq 0$. Bảng tính và ma trận cho $\tau_1 : (x' = x + 4; y' = y)$ là

μ	$c_1x +$	c_2y			$+ d \leq 0$	
λ_0					$-1 \leq 0$	
λ_1	x		$-x'$		$+4 = 0$	
λ_2		y		$-y'$	$= 0$	
			$c_1x' + c_2y'$		$+d \leq 0$	$\leftarrow \varphi'$
					$1 \leq 0$	$\leftarrow$ disable

$$\begin{bmatrix} \mu c_1 & \mu c_2 & 0 & 0 & \mu d \\ 0 & 0 & 0 & 0 & -\lambda_0 \\ \lambda_1 & 0 & -\lambda_1 & 0 & 4\lambda_1 \\ 0 & \lambda_2 & 0 & -\lambda_2 & 0 \end{bmatrix}$$

Bảng tính và ma trận cho $\tau_2 : (x' = x + 2; y' = y + 1)$ được biểu diễn tương tự.

μ	$c_1x +$	c_2y			$+ d \leq 0$	
λ_0					$-1 \leq 0$	
λ_1	x		$-x'$		$+2 = 0$	
λ_2		y		$-y'$	$+1 = 0$	
			$c_1x' + c_2y'$		$+d \leq 0$	$\leftarrow \varphi'$
					$1 \leq 0$	$\leftarrow$ disable

Với bảng tính trên, ta có ma trận hệ số dưới đây.

$$\begin{bmatrix} \mu c_1 & \mu c_2 & 0 & 0 & \mu d \\ 0 & 0 & 0 & 0 & -\lambda_0 \\ \lambda_1 & 0 & -\lambda_1 & 0 & 2\lambda_1 \\ 0 & \lambda_2 & 0 & -\lambda_2 & \lambda_2 \end{bmatrix}$$

2.2.4 Công thức đường thực thi

Như đã thể hiện ở trên, mỗi bảng tính sẽ có hai công thức đường thực thi *(path formula)* được tạo ra, một đường đại diện trường hợp thỏa mãn - φ, φ', một đại diện trường hợp không thỏa mãn - *disable*. Áp dụng bổ đề Farkas, trong từng *path formula*, ta xác định được mối quan hệ giữa $c_1, c_2, ..., d$ dựa trên $\mu, \lambda_0, \lambda_1, ...\lambda_m$. Công thức cuối cùng là hợp của tất cả các *path formula* có thể của chương trình, nghĩa là được sinh ra từ cả ma trận Initiation và các ma trận Consecution.
Với bài toán ở trên, ta có công thức tại Initiation là

$$(\exists \lambda_1, \lambda_2)[c_1 = \lambda_1 \wedge c_2 = \lambda_2 \wedge d = -2\lambda_1] \tag{2.1}$$

Với Consecution cho τ_1, công thức là

$$\exists(\mu, \lambda_i) \begin{bmatrix} \mu.c_1 + \lambda_1 = 0 \wedge \\ \mu.c_2 + \lambda_2 = 0 \wedge \\ c_1 = -\lambda_1 \wedge c_2 = \lambda_2 \wedge \\ (\mu - 1)d = \lambda_0 - 4\lambda_1 \end{bmatrix} \vee \exists(\mu, \lambda_i) \begin{bmatrix} \mu.c_1 + \lambda_1 = 0 \wedge \\ \mu.c_2 + \lambda_2 = 0 \wedge \\ \lambda_1 = 0 \wedge \lambda_2 = 0 \wedge \\ \mu.d - \lambda_0 = 1 \end{bmatrix} \tag{2.2}$$

Tương tự, ma trận cho $\tau_2 : (x' = x + 2; y' = y + 1)$ được biểu diễn và từ đó thu được công thức

$$\exists(\mu, \lambda_i) \begin{bmatrix} \mu.c_1 + \lambda_1 = 0 \wedge \\ \mu.c_2 + \lambda_2 = 0 \wedge \\ c_1 = -\lambda_1 \wedge c_2 = -\lambda_2 \wedge \\ d = \mu.d + 2\lambda_1 + \lambda_2 - \lambda_0 \end{bmatrix} \vee \exists(\mu, \lambda_i) \begin{bmatrix} \mu.c_1 + \lambda_1 = 0 \wedge \\ \mu.c_2 + \lambda_2 = 0 \wedge \\ \lambda_1 = 0 \wedge \lambda_2 = 0 \wedge \\ \mu.d + 2\lambda_1 + \lambda_2 - \lambda_0 = 1 \end{bmatrix} \tag{2.3}$$

Vì vậy, với hệ thống chuyển trạng thái được ví dụ, ta thu được ba *path formula* là các công thức (2.1), (2.2), (2.3). Thuật toán 1 sinh ra đường thực thi cuối cùng được

trình bày dưới đây. Trong đó, công thức được dựng lên từ từng bảng tính *Initiation* và *Consecution*, sau đó được nối thành công thức cuối cùng bằng các phép toán logic hội và tuyển tương ứng.

Thuật toán 1: Thuật toán sinh công thức đường thực thi

```
   Input  : Transition System ts
   Output: String pathformula
 1 init ← ts.getInitiation();
 2 initformula ← createFormula(init);
 3 pathformula ← initformula;
 4 for  X ∈ ts.getConsecutions() do
 5 |   /* join path formula for consecutions                    */
 6 |   consformula ← Or(createFormula(X.enable), createFormula(X.disable);
 7 |   pathformula ← And(pathformula, consformula);
 8 end
   Result: pathformula
 9 /* create formula from initiation and consecution matrix     */
10 Function createFormula(String[][] matrix) : String
11 |   formula ← empty;
12 |   for j ∈ matrix.getColumn() do
13 |   |   for i ∈ matrix.getRow() do
14 |   |   |   formula += sumByColumn(matrix[i],[j]);
15 |   |   end
16 |   end
17 end
```

2.2.5 Giải công thức ràng buộc và sinh bất biến

Công thức ràng buộc giữa $c_1, c_2, ...d$ là một biểu thức logic thu được khi rút gọn và loại bỏ lượng từ từ các công thức đường thực thi. Bộ loại bỏ lượng từ được sử dụng là công cụ REDLOG[3]. Sau khi qua REDLOG, ta thu được lần lượt các công thức ràng

[3]http://redlog.com

buộc tương ứng với từng *path formula*. Với ví dụ trên, ta có công thức ràng buộc cuối cùng là hội của các công thức (1), (2) và (3) như dưới đây.

```
REDLOG OUTPUT:
(1) 2*c1 + d <= 0
(2) c2 != 0 and c1 <= 0
(3) c2 != 0 and 2*c1 + c2 <= 0
```

Hình 2.2: Ràng buộc thu được sau khi dùng REDLOG

Sau đó, công thức ràng buộc được đưa vào bộ giải Z3py để sinh ra các bộ $c_1, c_2, ..., d$ thỏa mãn, đồng thời ta có các bất biến vòng lặp tương ứng. Ý tưởng để giải công thức ràng buộc là tìm kiếm giá trị thỏa mãn trên từng không gian của bộ nghiệm. Với công thức ràng buộc gồm ba biến c_1, c_2, d, từng không gian nghiệm được kiểm tra. Các không gian có thể là $c_1 < 0, c_2 < 0, d < 0$ hoặc $c_1 \geq 0, c_2 < 0, d < 0$ hoặc $c_1 \geq 0, c_2 \geq 0, d < 0$ ... Vì vậy với ba biến, số không gian cần kiểm tra là $2^3 = 8$. Trong ví dụ trên, sau khi giải ràng buộc sử dụng ý tưởng trên bằng Z3py, kết quả đưa ra được ba bộ nghiệm thỏa mãn.

c_1	c_2	d	$c_1x + c_2y + d$	≤ 0
0	-1	0	$-y$	≤ 0
-1	-1	2	$-x - y + 2$	≤ 0
-1	2	0	$-x + 2y$	≤ 0

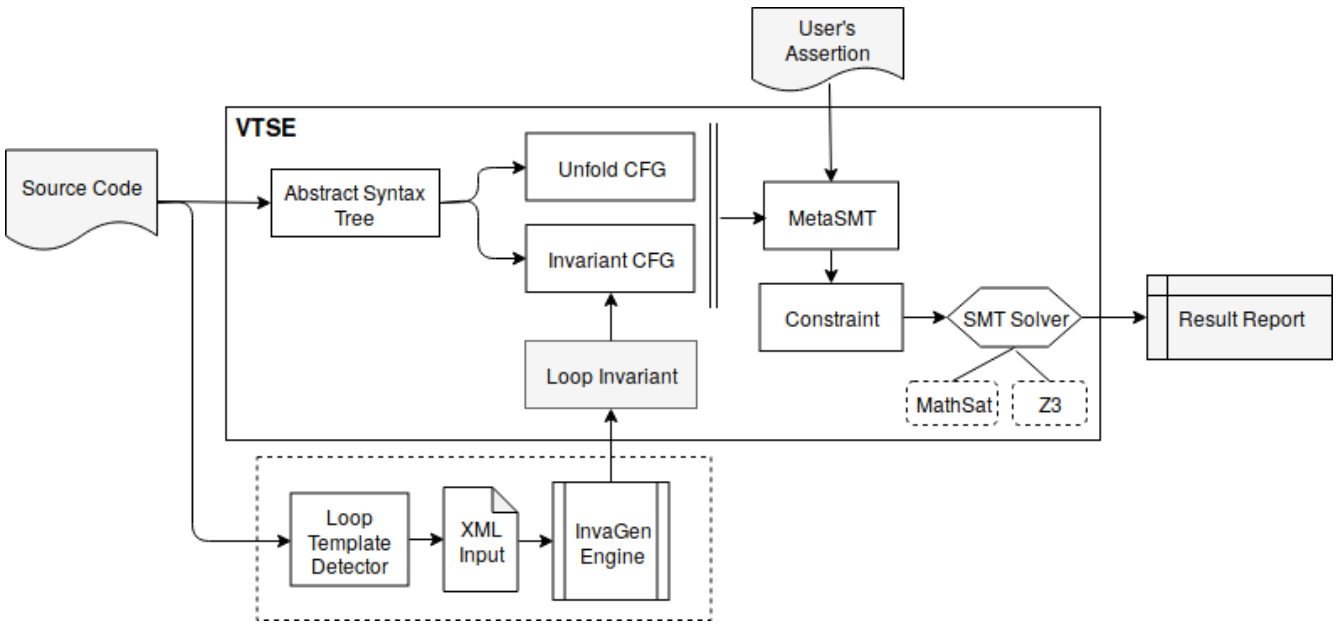
Cuối cùng, ta thu được công thức bất biến vòng lặp

$$\varphi : (y \geq 0) \wedge (x + y - 2 \geq 0) \wedge (x - 2y \geq 0)$$

Chương 3

Kết hợp bất biến trong kiểm chứng chương trình

3.1 Công cụ kiểm chứng chương trình VTSEinv

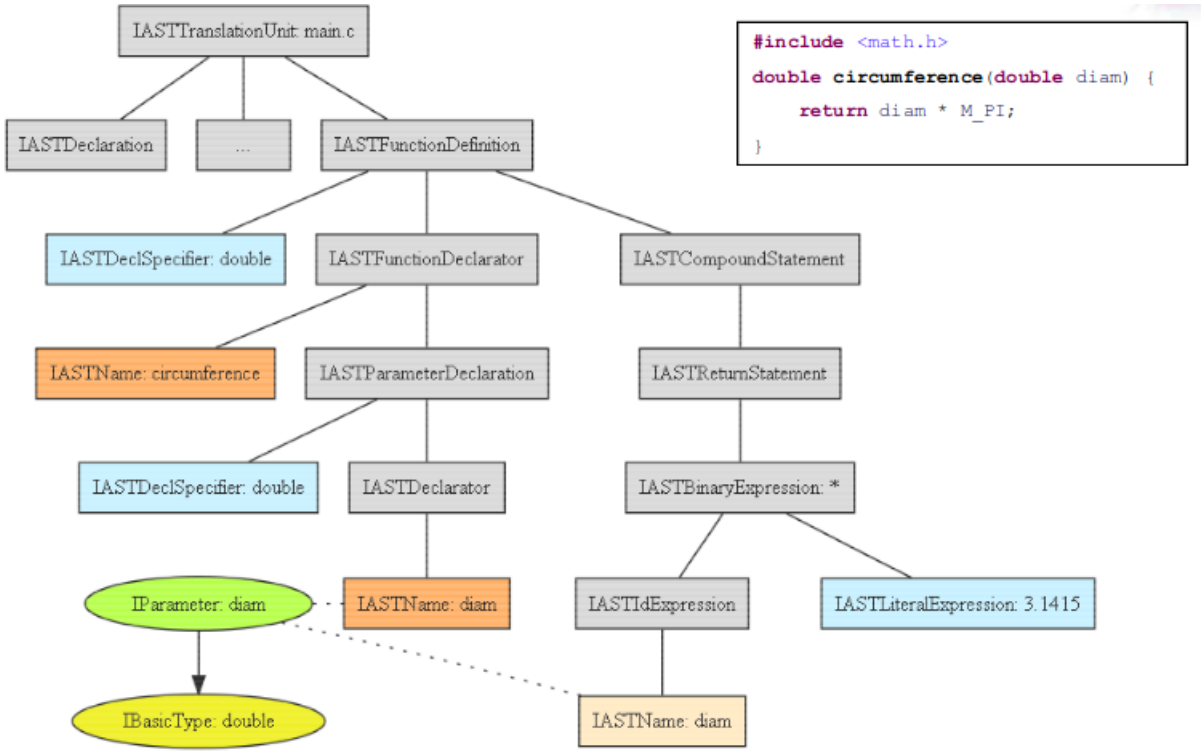


Hình 3.1: Kiến trúc của VTSEinv

Khóa luận xin được đề xuất một phương pháp kiểm chứng chương trình dựa trên sự kết hợp của kĩ thuật thực thi tượng trưng và sinh bất biến vòng lặp. Đồng thời, công cụ VTSEinv được xây dựng trên phương pháp này đã được cài đặt. VTSEinv là phiên bản mở rộng của công cụ VTSE đã được trình bày trong nghiên cứu trước đây[15].

3.1.1 Cây cú pháp trừu tượng

Cây cú pháp trừu tượng là một cách thức biểu diễn có cấu trúc của mã nguồn chương trình. Để chuyển từ mã nguồn C/C++ về cây cú pháp trừu tượng, thư viện CDT [1] của Eclipse được sử dụng. Nút nguồn của cây được CDT đưa ra là một IASTTranslationUnit. Mỗi thành phần con của cây đại diện cho một cấu trúc ngữ pháp của chương trình.



Hình 3.2: Cây cú pháp trừu tượng trong CDT

[1] https://www.eclipse.org/cdt/

3.1.2 Đồ thị luồng điều khiển

Đồ thị luồng điều khiển *(Control Flow Graph)* là một đồ thị có hướng thể hiện tất cả các đường có thể được thực thi của chương trình. Đồ thị được sinh ra bằng cách duyệt cây cú pháp trừu tượng từ trên xuống. Luồng điều khiển được xây dựng trong VTSEinv dựa trên 4 loại nút chính:
–*Nút đơn* tương ứng với các câu lệnh đơn bao gồm câu lệnh khởi tạo, câu lệnh gán và lệnh trả về.
–*Nút điều kiện* trong câu lệnh if-else, do-while, switch-case, và có chứa biểu thức điều kiện cùng hai nhánh trỏ đến hai phần then, else.
–*Nút đánh dấu* được dùng để đánh dấu một số vị trí như bắt đầu vòng lặp, bắt đầu rẽ nhánh, ...
–*Nút khác* bao gồm nút label, nút goto, nút không xác định ...

Xây dựng đồ thị luồng điều khiển

Phương pháp sử dụng là duyệt từng nút của cây cú pháp trừu tượng, khi đó với từng cấu trúc cú pháp, đồ thị tương ứng được xây dựng. Ban đầu, đồ thị của từng hàm con trong chương trình được tạo. Sau đó, VTSEinv sẽ nối tất cả các đồ thị con thành một đồ thị hoàn chỉnh của chương trình.

Cụ thể, sau khi có tập đồ thị của từng hàm, gọi là các *intraCFG*, đồ thị luồng điều khiển chung *interCFG* được xây dựng. Chương trình sẽ tìm đến hàm *main*, hoặc một hàm được định nghĩa sẵn, sau đó duyệt *intraCFG* của hàm này để tạo *interCFG*. Khi gặp nút trong đồ thị đánh dấu là lời gọi hàm, thì *intraCFG* tương ứng của hàm được gọi sẽ nối vào *interCFG*. Trước khi nối, các nút tương ứng cho tham số của hàm được gọi cũng được thêm vào *interCFG*. Nói chung, việc xây dựng đồ thị luồng điều khiển *intraCFG* là tương tự với công cụ VTSE phiên bản cũ, thuật toán cụ thể được trình bày tại Thuật toán 2.

Thuật toán 2: Thuật toán xây dựng đồ thị luồng điều khiển

Input : AbstractSyntaxTree ast

Output: ControlFlowGraph intra_cfg

```
1  Function buildCFG(ASTNode astNode) : ControlFlowGraph
2      if astNode là câu lệnh đơn then
3          node ← node đơn chứa câu lệnh
4          cfg ← new CFG(node, node)
5      else if astNode là khối lệnh then
6          foreach thành phần n ∈ khối lệnh do
7              subCfg ← buildCfg(n)
8              nối subCfg vào cfg chính;
9          end
10     else if astNode là câu lệnh If-else then
11         thenBranch ← buidCFG(astNode.then)
12         elseBranch ← buildCFG(astNode.else)
13         decisionNode ← nút biểu thức điều kiện
14         gán decisionNode với thenBranch và elseBranch
15         tạo nút beginIf và endIf;
16         cfg ← new CFG(beginIf, endIf)
17     else if astNode là câu lệnh For then
18         tạo nút beginFor và endFor
19         initNode ← chứa câu lệnh bắt đầu vòng lặp
20         decisionNode ← chứa điều kiện lặp
21         thenBranch ← buildCFG(astNode.forBody)
22         elseBranch ← kết thúc for
23         iterationNode ← bước nhảy /* vd: i ++; i = i + 2; ...      */
24         cfg ← new CFG(beginFor, endFor)
25     else if astNode là lời gọi hàm then
26         functionCallNode ← câu lệnh chứa lời gọi hàm
27         /* cfg ban đầu chỉ chứa lời gọi, cfg của từng hàm được nối
              vào sau                                                  */
```

```
28
29      else if astNode là câu lệnh While-do/Do-while then
30          tương tự câu lệnh for
31          ...
32      else if astNode là câu lệnh Switch-case then
33          foreach n ∈ switch từ dưới lên do
34              if n là khối default then
35                  subCfg ← buildCFG(n)
36                  lastNode ← subCfg.start
37          else if n là khối case then
38              mỗi case tạo một khối CFG If-else
39              lastNode ← conditionNode trên cùng
40              tạo beginSwitch, endSwitch
41          cfg ← new CFG(beginSwitch, endSwitch)
42      return cfg
```

Gỡ vòng lặp trong đồ thị luồng điều khiển

Với vòng lặp trong đồ thị luồng điều khiển, có hai chiến lược được sử dụng là gỡ vòng lặp với số lần nhất định và sử dụng bất biến. Với phương pháp gỡ (còn gọi là *unfolding*), mỗi vòng lặp được chuyển về dạng một tập các câu lệnh *if-else* nối nhau. Số lần gỡ vòng lặp được định nghĩa bởi người dùng hoặc mặc định là 100 lần gỡ. Với từng lần gỡ, một bản sao của nút điều kiện và đoạn đồ thị thân vòng lặp được tạo ra và gắn vào đồ thị mới dưới dạng *if-else*. Để sao chép đồ thị, thư viện Kryo [2] phát triển bởi Google được sử dụng. Phương pháp gỡ vòng lặp được trình bày bằng Thuật toán 4 dưới đây.

[2]https://github.com/EsotericSoftware/kryo

Thuật toán 3: Thuật toán gỡ vòng lặp

Input : ControlFlowGraph loopGraph

Output: ControlFlowGraph unfoldGraph

```
1  /* create copy of then clause                                    */
2  decisionNode ← loopGraph.decision
3  cpyThenClause ← decisionNode.then
4  lastNode ← loopGraph.end
5  /* create multiple if-else from bottom to up                     */
6  for i ∈ number_of_loops do
7      condition ← copy(decisionNode)
8      condition.else ← endNode
9      condition.then ← cpyThenClause
10     cpyThenClause.exit.next ← lastNode
11     lastNode ← condition
12 end
13 unfoldGraph ← new ControlFlowGraph(lastNode, endNode)
```

3.1.3 Kết hợp bất biến vòng lặp vào VTSEinv

Để sinh biểu thức bất biến, mã nguồn chứa vòng lặp được kiểm tra xem có phù hợp với biểu mẫu *(template)* được định nghĩa trước. Do hiện nay phương pháp sinh bất biến là bổ đề Farkas, vì vậy nên *template* vòng lặp được xử lý cũng cần phù hợp với đầu vào của công cụ InvaGen đã được trình bày ở Chương 2. Các *template* được định nghĩa trước là các vòng lặp *for*, *while* đơn hoặc có chứa câu lệnh rẽ nhánh trong vòng lặp. Trường hợp vòng lặp lồng nhau *(nested loop)* chưa được xử lý. Đoạn Mã nguồn 3.1 thể hiện cho một chương trình chứa vòng lặp và đầu vào tương ứng của InvaGen để sinh bất biến. Để chuyển từ mã nguồn C về một hệ thống chuyển đổi trạng thái, ban đầu *template* của vòng lặp được kiểm tra và từ đấy những thành phần của câu lệnh lặp được lưu lại dưới dạng *Transition System*. Sau khi bất biến được sinh ra từ InvaGen, biểu thức bất biến được thêm vào đồ thị luồng điều khiển để dựng lên InvariantCFG. Thuật toán 4 trình bày các bước dựng đồ thị InvariantCFG.

Mã nguồn 3.1: Ví dụ chương trình C

```c
int main() {
   int x = 0;
   while (x < 1000) {
      if (x < 100) {
         x += 1;
      } else {
         x += 2;
      }}
   return 0;
}
```

Mã nguồn 3.2: TransitionSystem.xml

```
<TransitionSystem>
<Variables>x</Variables>
<Initiation> x = 0 </Initiation>
<Consecutions>
<Transition> x < 100; x' = x + 1
   </Transition>
<Transition> x >= 100; x' = x + 2
   </Transition>
</Consecutions>
</TransitionSystem>
```

Thuật toán 4: Thuật toán dựng đồ thị xử lý bất biến

Input : ControlFlowGraph inter_cfg

Output: ControlFlowGraph invariant_cfg

Function *iterateCFG(CFGNode inter_cfg.start): invariant_cfg*

- **if** *node là beginFor // beginWhile* **then**
  - /* thêm Invariant node */
  - decisionNode ← begin.next
  - notCondition ← getNotCondition(decisionNode)
  - invariantNode.next ← notCondition
  - invariantSubCFG ← new CFG(invariantNode,notCondition)
  - invariantSubCFG.exit ← iterateCFG(endNode)
- **else if** *node là beginIf* **then**
  - condition ← begin.next;
    condition.setThen(iterateCFG(condition.then))
  - condition.setElse(iterateCFG(condition.else))
- **else if** *node là plainNode // labelNode // endNode* **then**
  - node.next ← iterateCFG(node.next)

3.1.4 Đánh chỉ số và sinh biểu thức ràng buộc

Về cơ bản, phương pháp đánh chỉ số và duyệt ra biểu thức trừu tượng hóa chương trình trong VTSEinv tương tự phiên bản trước đó. Tuy nhiên, VTSEinv có sửa lại đầu ra cho phù hợp theo chuẩn của SV-COMP và kết nối thêm với một số bộ giải SMT khác ngoài Z3 như MathSat, raSat, ...

Để tạo ra công thức ràng buộc từ đồ thị luồng điều khiển, đồ thị được duyệt từ trên xuống dưới và đánh chỉ số cho từng lần xuất hiện của biến. Việc đánh chỉ số là cần thiết do trong bộ giải SMT không có khái niệm biến mà chỉ xử lý các hằng số. Vì vậy để thể hiện sự thay đổi của biến số, ta phải đánh chỉ số cho từng biến tương ứng trong quá trình thực thi chương trình. Biến được khởi tạo với chỉ số index -1, với mỗi lần cập nhật giá trị thì chỉ số của biến tăng thêm một đơn vị.

Công thức trừu tượng hóa của chương trình sau khi được sinh ra sẽ được kết hợp với điều kiện người dùng để tạo thành biểu thức ràng buộc cuối cùng. Gọi F_{final} là biểu thức cuối cùng, $F_{program}$ là biểu thức trừu tượng hóa của chương trình và $F_{assertion}$ là điều kiện người dùng. Ta có

$$F_{final} = F_{program} \wedge \neg F_{assertion}$$

Công thức này sẽ được chuyển về định dạng SMT-LIB2 và thêm vào bộ giải SMT. VTSEinv hỗ trợ hai bộ giải là Z3[3] và Mathsat[4]. Từ đầu ra của bộ giải, công cụ đưa kết quả kiểm chứng chương trình tương ứng.

- Kết quả do SMT trả về là **sat** (satisfaction): tồn tại giá trị để làm thỏa mãn F_{final}, nghĩa là điều kiện người dùng không phải luôn đúng với mã nguồn và có đường thực thi bị vi phạm. Khi này, VTSEinv trả về kết quả **false** và đưa ra phản ví dụ tương ứng.
- Kết quả do SMT trả về là **unsat** (unsatisfaction): không tồn tại giá trị để thỏa mãn F_{final}, nghĩa là điều kiện người dùng thỏa mãn so với mã nguồn. VTSEinv trong trường hợp này sẽ trả về **true**.
- Kết quả của SMT là **timeout**: bộ giải không đưa ra được kết quả trong thời gian quy định. VTSEinv trả về kết quả **timeout** tương tự.

[3]https://rise4fun.com/z3/

[4]http://mathsat.fbk.eu/

3.2 Minh hoạ hoạt động của VTSEinv

Trong phần này, khóa luận xin minh họa quy trình kiểm chứng của VTSEinv bằng hai phương pháp gỡ vòng lặp và sử dụng bất biến.

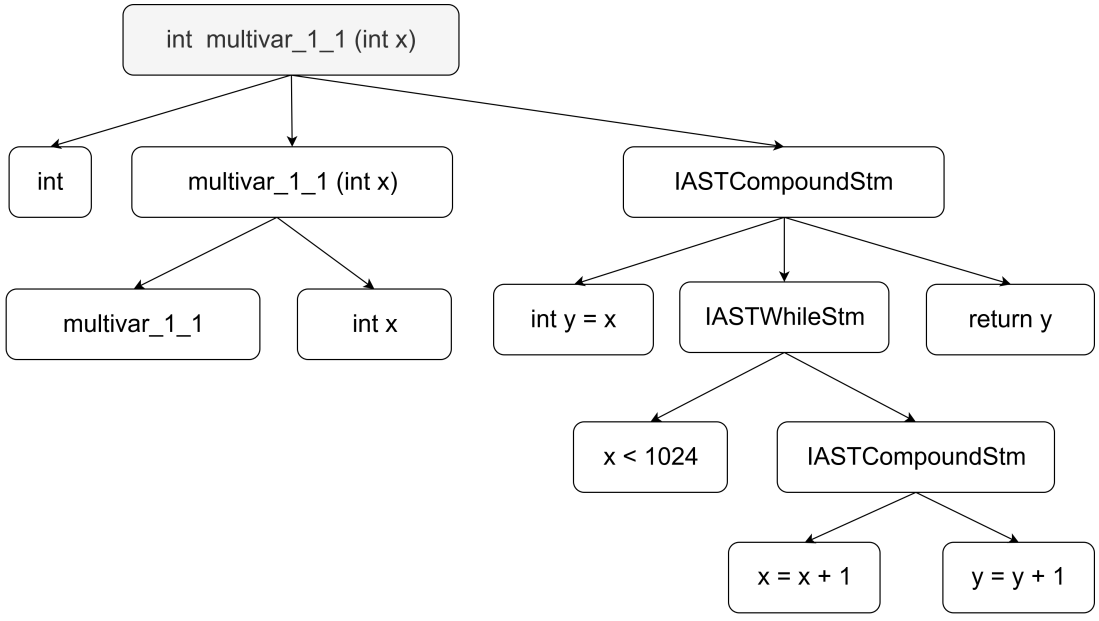
Mã nguồn chương trình

Xét bài toán kiểm chứng mã nguồn C dưới đây:

```
1 int multivar_1_1(int x) {//pre-condition x = 0;
2   int y = x;
3   while (x < 1024) {
4     x = x + 1;
5     y = y + 1;}
6   return y;
7   VERIFIER_assert(return < 1024); //assertion user want to verify}
```

Cây cú pháp trừu tượng

Thư viện CDT được sử dụng để sinh ra AST với cấu trúc như Hình **??**.



Hình 3.3: Cây cú pháp trừu tượng từ mã nguồn

Sinh bất biến vòng lặp

Mã nguồn C được xử lý chuyển về dạng Transition System tương ứng và đưa vào InvaGen.

Mã nguồn 3.3: multival_1_1.xml

```
<TransitionSystem>
   <Variables> x; y </Variables>
   <Initiation> y = x </Initiation>
   <Consecutions>
      <Transition> x' = x + 1; y'= y + 1 </Transition>
   </Consecutions>
</TransitionSystem>
```

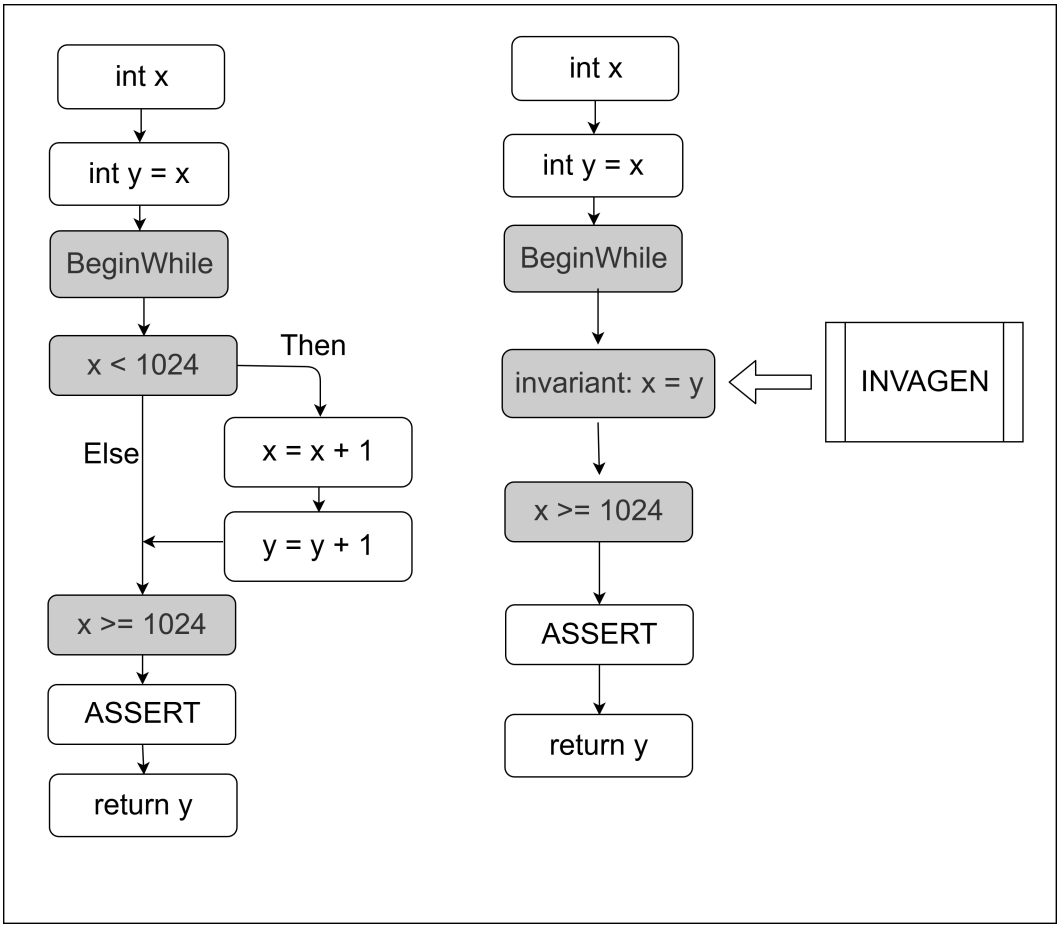
Với file đầu vào xml ở trên, InvaGen xử lý và đưa ra kết quả là công thức bất biến của vòng lặp như sau:

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
================================================
Generate invariants for multival_1_1_true.xml
Runtime : 1.115 (s)
Invariant:
-y+x <= 0 and y-x <= 0

Process finished with exit code 0
```

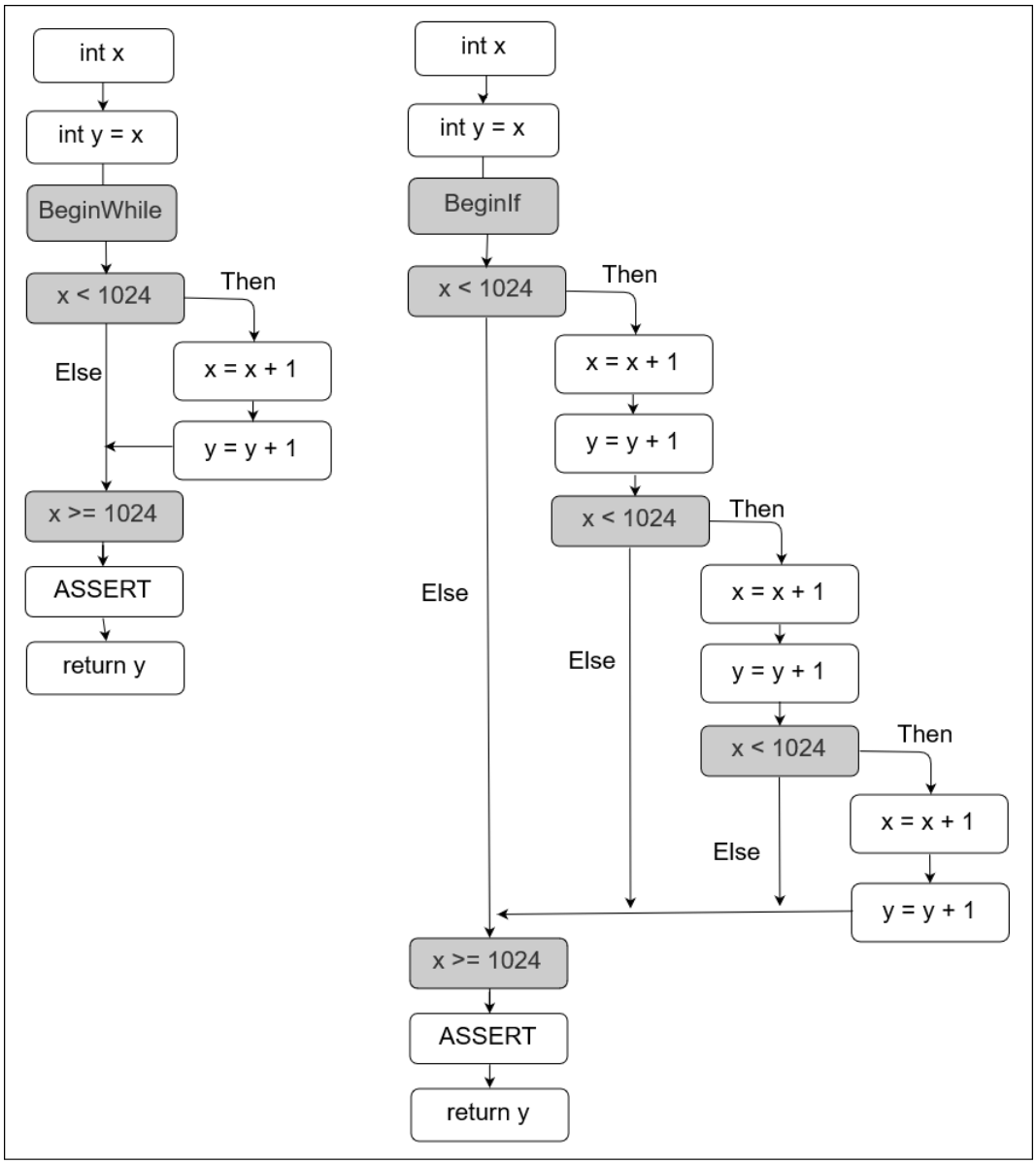
Đồ thị luồng điều khiển

Có hai đồ thị luồng điều khiển được xây dựng riêng biệt cho hai chiến lược xử lý vòng lặp khác nhau. Với phương pháp dùng bất biến vòng lặp, công thức bất biến sau khi thu từ InvaGen sẽ được gắn thẳng vào CFG. Trong đồ thị xử lý bất biến vòng lặp, công thức bất biến được gắn thay cho thân vòng lặp. Khi này, bất biến là một xấp xỉ trên, thể hiện các tính chất luôn được thỏa mãn xuyên suốt các lần thực thi của vòng lặp. Công thức bất biến càng sát với vòng lặp thực tế thì khi sử dụng để kiểm chứng chương trình sẽ càng cho kết quả chính xác.



Hình 3.4: CFG chèn bất biến vòng lặp

Phương pháp gỡ vòng lặp sẽ chuyển câu lệnh lặp thành một tập các lệnh *if-else* với số lần gỡ nhất định. Con số này được định nghĩa bởi người dùng trước khi bắt đầu kiểm chứng, hoặc được gỡ theo mặc định là 100 lần. Để dễ dàng biểu diễn, ví dụ minh họa trên sẽ được gỡ với số lần lặp là 3, tương đương thân của vòng *while* được thực thi 3 lần. Đồ thị luồng điều khiển khi gỡ vòng lặp được trình bày ở Hình 3.5. Với mỗi lần gỡ vòng lặp, số lượng biến được tăng lên đáng kể. Vì vậy, với những bài toán có giới hạn lặp lớn, kích thước đồ thị luồng điều khiển khi gỡ sẽ tăng nhanh, dẫn đến làm phức tạp hóa công thức ràng buộc.



Hình 3.5: CFG phương pháp gỡ vòng lặp

Đánh chỉ số và sinh công thức ràng buộc

Với đồ thị invariantCFG hay unfoldCFG, đồ thị đều được đánh chỉ số và duyệt từ trên xuống để thu được công thức trừu tượng của mã nguồn chương trình $F_{program}$. Kết hợp với công thức điều kiện người dùng $F_{assertion}$, VTSEinv tạo ra file định dạng SMT-LIBv2, là đầu vào của bộ giải SMT.

Mã nguồn 3.4: File *multival_1_1.smt* phương pháp bất biến vòng lặp

```
(declare-fun x_multivar_1_1_-1 () Int)
   ...
(declare-fun y_multivar_1_1_0 () Int)
(declare-fun return_multivar_1_1_0 () Int)
(assert (and (and (=> (< x_multivar_1_1_0 1024)
(and (= x_multivar_1_1_1 y_multivar_1_1_0)
   (= x_multivar_1_1_1 1024)))
   (=> (not (< x_multivar_1_1_0 1024))
   (= x_multivar_1_1_1 x_multivar_1_1_0)))
   (= return_multivar_1_1_0 y_multivar_1_1_0)))
   (assert (= x_multivar_1_1_0 0))
(assert (not (< return_multivar_1_1_0 1024)))
(check-sat)
(get-model)
```

Cuối cùng, VTSEinv dựa vào kết quả của bộ giải SMT để đưa ra kết luận. Đồng thời, một báo cáo Excel thể hiện kết quả kiểm chứng được tạo ra. Dưới đây thể hiện hai kết quả của VTSEinv khi lần lượt sử dụng bất biến vòng lặp và gỡ vòng lặp 100 lần.

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
- Method name: multivar_1_1
Status: false
Counter example:
    + parameter: x int (0)
    + return = 1024
Total time: 20.0 (ms)
sat
(model
  (define-fun x_multivar_1_1_1 () Int
    1024)
  (define-fun x_multivar_1_1_0 () Int
    0)
  (define-fun return_multivar_1_1_0 () Int
    1024)
  (define-fun y_multivar_1_1_0 () Int
    1024)
)
```

Ket qua thuc nghiem

ID	function	pre-condition	post-condition	status	constra	totalTime (s)	counter example
1	multivar_1_1	x = 0	return < 1024	false	0.02	0.234	x = 0; return = 1024

Hình 3.6: Kết quả từ Z3 (trên) và báo cáo Excel (dưới) của VTSEinv dùng Invariant

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
- Method name: multivar_1_1
Status: false
Counter example:
    + parameter: x int (0)
    + return = 1024
Total time: 140.0 (ms)
sat
(model
  (define-fun y_multivar_1_1_1024 () Int
    1024)
  (define-fun x_multivar_1_1_2048 () Int
    1024)
  (define-fun x_multivar_1_1_2047 () Int
    1023)
  (define-fun y_multivar_1_1_1023 () Int
    1023)
```

Ket qua thuc nghiem

ID	function	pre-condition	post-condition	status	constra	totalTime (s)	counter example
1	multivar_1_1	x = 0	return < 1024	false	0.14	12.465	x = 0; return = 1024

Hình 3.7: Kết quả từ Z3 (trên) và báo cáo Excel (dưới) của VTSEinv dùng Unfold

Dùng cả hai phương pháp, công cụ đưa ra kết quả **false** và với tham số đầu vào (parameter) $x = 0$ thì giá trị của kết quả trả về là $return = 1024$. Có thể thấy, với bài toán trên, cả hai phương pháp đều có thể đưa ra kết quả kiểm chứng chương trình là **false** và đưa ra phản ví dụ chính xác. Tuy thời gian sinh ra công thức ràng buộc là không chênh lệch nhiều, nhưng tổng thời gian giải khi sử dụng bất biến cho tốc độ tốt hơn so với gỡ vòng lặp.

Chương 4

Nghiên cứu liên quan

Trong chương này, khóa luận xin trình bày một số nghiên cứu liên quan đến các công cụ kiểm chứng phần mềm và phương pháp sinh bất biến vòng lặp khác. Hai công cụ được nêu ra là CBMC và VeriAbs, đây là hai công cụ được tiến hành thực nghiệm và so sánh với VTSEinv. Sau đó, hai kĩ thuật sinh bất biến đang phát triển trong thời gian gần đây, đã được thể hiện là cho kết quả tốt hơn so với việc dùng bổ đề Farkas sẽ được trình bày.

4.1 Một số công cụ kiểm chứng phần mềm

CBMC là một công cụ đi đầu trong kiểm chứng phần mềm với kĩ thuật Bounded Model Checking. Trong khi đó, Veriabs là công cụ mở rộng trên nền tảng CBMC khi áp dụng thêm phương pháp *k-induction* và bất biến vòng lặp.

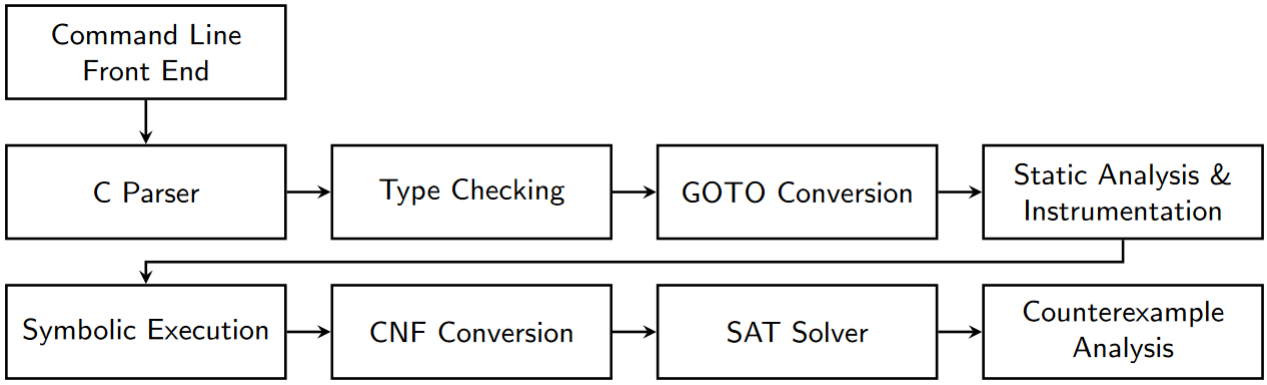
4.1.1 CBMC

Công cụ C Bounded Model Checker (CBMC) [11] có thể tìm ra sự vi phạm của một biểu thức điều kiện *assertion* trong chương trình C hoặc chứng minh biểu thức là đúng với một giới hạn cho trước. CBMC cài đặt một bộ dịch *bit-percise* của một đầu vào chương trình C, nghĩa là biến đổi chương trình với vòng lặp được gỡ một số lần

nhất định về công thức bit-vector với kích thước cố định. Nếu công thức là thỏa mãn thì có nghĩa là chương trình dẫn đến một *assertion* bị vi phạm. Bộ giải để xét tính thỏa được của công thức được sử dụng là MiniSat 2.2.0 [6].

Kiến trúc của CBMC

Bounded model checker như CBMC chuyển bài toán phân tích đường thực thi của chương trình về công thức ràng buộc có thể được giải bằng độ giải SAT hay SMT. Với việc sử dụng bộ giải SAT, CBMC có thể nhận đầu vào là một chương trình C với các điều kiện *assertion* cần kiểm chứng. CBMC cho ra một biểu thức CNF biểu diễn các đường thực thi của chương trình dẫn đến vi phạm *assertion*. Quy trình CBMC biến đổi đường thực thi được trình bày ở Hình 4.1. Kiến trúc của CBMC bao gồm bốn phần chính.



Hình 4.1: Kiến trúc của CBMC

Front end. Ban đầu, CBMC cho phép người dùng nhập một số tùy chọn ví dụ như độ rộng của bit, số vòng lặp, ... Một *C parser* xây dựng cây cú pháp từ mã nguồn rồi chuyển qua bộ *Type checking*. Mỗi kí tự trong cây được gán với một kiểu thông tin ở dạng bit.

Intermediate Representation. CBMC sử dụng *GOTO programs* như một biểu diễn trung gian. Với biểu diễn trung gian này, mọi luồng thực thi phi tuyến tính ví dụ như if, switch, vòng lặp, ... được chuyển về các câu lệnh goto tương ứng.

Middle end. CBMC áp dụng thực thi tượng trưng bằng cách gỡ vòng lặp một số lần hữu hạn, con số này được quyết định bởi người dùng. Sau khi thực hiện gỡ vòng lặp, CBMC chuyển chương trình dạng GOTO về dạng *static single asignment (SSA)*.

Back end. CBMC chuyển chương trình từ dạng SSA về biểu thức CNF bằng cách lập mô hình bit của tất cả các câu lệnh kèm thêm các ràng buộc Boolean. Sau đó, công thức CNF được đưa vào MiniSat 2.2.0 để kiểm chứng và đưa ra phản ví dụ nếu có.

Điểm mạnh và hạn chế

CBMC có điểm mạnh về tốc độ và số bài toán có thể giải được. Về mặt hạn chế, CBMC không thể xử lý các bài toán mà vòng lặp là không giới hạn. Tuy nhiên, với bài toán chứa vòng lặp, CBMC đặt một mức thời gian nhất định và bắt buộc chương trình chấm dứt với đầu ra TRUE/FALSE để đưa ra một kết quả khả thi. Vì vậy, trong bộ thực nghiệm xử lý vòng lặp, nhiều bài toán khi giải bằng CBMC cho ra kết quả là *unsound*.

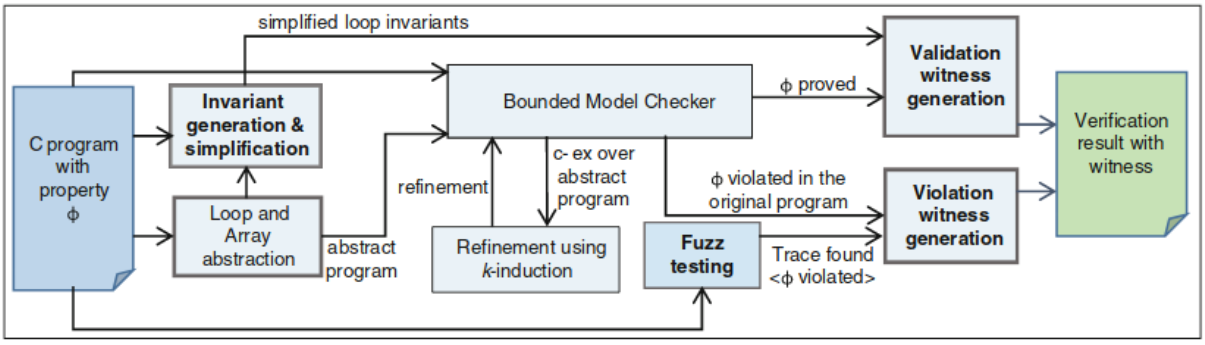
4.1.2 VeriAbs

VeriAbs [5] là một công cụ kiểm chứng chương trình C/C++ sử dụng kĩ thuật k-induction để mở rộng phương pháp Bounded Model Checking cho chương trình với vòng lặp lớn hoặc không xác định được giới hạn lặp. VeriAbs trừu tượng hóa những vòng lặp như vậy thành các vòng lặp xác định với số lần nhỏ hơn và có thể được chứng minh bởi BMC. Ý tưởng này dựa trên khái niệm thu gọn vòng lặp *(loop shrinkability)* [2].

Kiến trúc và quy trình kiểm chứng của VeriAbs

Quy trình kiểm chứng của VeriAbs được thể hiện ở Hình 4.2. VeriAbs chuyển file mã nguồn C qua một Tata Consultancy Services (TCS)[1] để sinh ra biểu diễn trung

[1]https://www.tcs.com/



Hình 4.2: Kiến trúc VeriAbs

gian *(intermediate representation)* của chương trình. Sau đó, bản IR này được phân tích sử dụng PRISM để thực hiện trừu tượng hóa và tính toán. VeriAbs sử dụng C Bounded Model Checker (CBMC) [11] phiên bản 5.8 với MiniSat [6] để kiểm chứng trừu tượng hóa hoặc chương trình ban đầu với số vòng lặp đã biết. VeriAbs sinh ra *correct witnesses* của chương trình bằng cách tạo ra các bất biến vòng lặp từ hậu điều kiện. Để tạo bất biến vòng lặp, công cụ sử dụng Z3.4.5.1[2] để loại bỏ lượng từ, sau đó bất biến được thêm vào *control flow automaton* sinh bởi CPAchecker bản 1.6.1 [1]. Để sinh ra *error witnesses*, VeriAbs sử dụng CBMC bản 5.8.

Điểm mạnh và hạn chế

Điểm mạnh chính của VeriAbs là tính an toàn *(sound)*. Tất cả các biến đổi thực hiện ở công cụ là xấp xỉ trên. Khi sử dụng CBMC, công cụ cho phép một tùy chọn **unwinding-assertions**, để đảm bảo nếu công cụ thông báo là tính chất này thỏa mãn thì điều đó là chắc chắn. Một lợi thế khác của VeriAbs là công cụ biến đổi tất cả các vòng lặp trong chương trình về vòng lặp trừu tượng với số lượng lần lặp là xác định, vì vậy, *bounded model checker* có thể được sử dụng để kiểm chứng.

Hạn chế của Veriabs là không có quy trình diễn giải trừu tượng để có thể tìm ra lỗi. Tuy nhiên, công cụ có thể tìm được lỗi thông qua *testing*. Veriabs phụ thuộc vào Z3 để loại bỏ lượng từ và sinh ra bất biến vòng lặp, phụ thuộc vào CPAChecker để sinh ra *automata* của chương trình.

[2]https://rise4fun.com/z3/

4.2 Phương pháp sinh bất biến vòng lặp

4.2.1 Sinh bất biến sử dụng Nội suy Craig

Nội suy Craig

Cho một cặp công thức (A,B) trong đó $A \wedge B$ là không thỏa mãn, một công thức nội suy cho (A, B) là một công thức I với các tính chất sau[13]:

- A $\rightarrow$ I
- I $\wedge$ B là *unsat*
- Tập ngôn ngữ của I thuộc giao của A và B.

Bổ đề nội suy Craig nói rằng một nội suy luôn tồn tại với A và B là *unsat*.

Sinh bất biến vòng lặp sử dụng nội suy

Hệ thống công thức được sử dụng để thể hiện hệ thống chuyển đổi trạng thái dưới dạng công thức vị từ cấp một. Ta gọi S là một trạng thái của hệ thống. Với mọi kí hiệu s $\in$ S, ta có kí hiệu s' đại diện cho các giá trị s tại một đơn vị trạng thái tiếp theo. Máy sinh bất biến vòng lặp G là một thủ tục nhận đầu vào M và đầu ra là một dãy các công thức bất biến.

Thuật toán sinh bất biến sau được dựa trên *unfolding* hệ thống chuyển đổi trạng thái trong Bounded Model Checking. Với k $\geq$ 0, khi gỡ M k-bước, ta có công thức:

$$U_k(M) = I, T, T^(1), ..., T^{(k-1)}, \neg P^k$$

trong đó I - *initiation* là trạng thái mở đầu, T - *transition* là các biến đổi của chương trình và P - *postcondition* là hậu điều kiện.Công thức này thể hiện một tập các thực thi của hệ thống chuyển đổi trạng thái khi chạy k-bước và kết thúc ở trạng thái *unsafe*. Hệ thống M là *safe* khi $U_k(M)$ là không thỏa mãn với $\forall k$.

Một trong những công cụ đầu tiên sử dụng nội suy Craig được cài đặt là SPASS[3].

[3]https://www.spass-prover.org

Thuật toán 5 trình bày phương pháp sinh bất biến vòng lặp trong SPASS.

Thuật toán 5: Thuật toán sinh bất biến sử dụng nội suy Craig

Input : Hệ thống M=(I,T,P) có một công thức bất biến

Output: Một chuỗi các công thức bất biến của M

```
1  i ← 1; k ← 1
2  Lặp:
3  Sử dụng bộ giải cho U_k(M)
4  if bộ giải trả về một nội suy I then
5  |  for j = 1 → k+1 do
6  |  |  Xây dựng ⋁_{l=1...j}(I_l)
7  |  |  k++
8  |  end
9  else
10 |  //bộ giải không trả về được nội quy
11 |  i ++
```

Thuật toán trên sử dụng chiến lược gỡ tiến dần dần từng trạng thái của hệ thống. Vì vậy, bất biến sinh ra thường là xấp xỉ trên rất rộng của chương trình và không đủ mạnh để chứng minh tính chất của hệ thống. Một chiến lược gỡ trạng thái mới hơn được giới thiệu tại ASE 2017 [17]. Trong báo cáo này, thay vì chỉ dùng *forward unfolding*, các tác giả kết hợp cả *backward unfolding* để rút gọn được bất biến là sát với hệ thống nhất. Phương pháp này có ưu thế so với bổ đề Farkas do *backward unfolding* sinh ra được những bất biến khác nhau tùy vào điều kiện cần kiểm tra. Vì vậy, có thể đảm bảo bất biến là đủ mạnh để kiểm chứng chương trình với từng điều kiện người dùng cụ thể

4.2.2 Sinh và làm mịn bất biến bằng phương pháp lấy mẫu chọn lọc

Như đã nói ở trên, việc tìm được bất biến đủ mạnh để phân tích và kiểm chứng chương trình là bài toán không quyết định. Một trong những thuật toán làm mịn bất biến đã được Jiajying Li và cộng sự [12] đưa ra tại hội nghị ASE 2017. Tác giả đã đề

xuất phương pháp sinh bất biến vòng lặp bằng cách kết hợp Học máy và Kiểm chứng chương trình, đồng thời giới thiệu công cụ ZILU [4] sử dụng phương thức này. Với đầu vào là một chương trình chứa vòng lặp tuân thủ Logic Hoare, công cụ ZILU sẽ ngẫu nhiên kiểm tra chương trình *(testing)*, thu thập trạng thái chương trình tại thời gian chạy và phân loại trạng thái xem chúng có thỏa mãn bất biến được tìm thấy. Tiếp theo, kỹ thuật phân loại được sử dụng để tạo ra bất biến vòng lặp tự động. Cuối cùng, ZILU làm mịn bất biến sử dụng Lấy mẫu có chọn lọc. Đến khi bất biến không thể làm mịn hơn, công cụ kiểm tra liệu bất biến đó có thể được dùng để chứng minh bộ ba Hoare.

Lấy mẫu chọn lọc – Selective sampling

Mẫu có thể được thu thập bằng ba chiến lược khác nhau. Ban đầu, các mẫu được chọn ngẫu nhiên để làm tập mẫu khởi đầu. Chi phí để sinh dữ liệu ngẫu nhiên thường là thấp, tuy nhiên thường cần một lượng lớn dữ liệu ngẫu nhiên để có thể đưa ra kết quả chính xác. Chiến lược thứ hai, lấy mẫu chọn lọc tiêu tốn nhiều hơn, tuy nhiên, mẫu chọn lọc thu được nhiều lợi ích từ lấy mẫu ngẫu nhiên. Cách cuối cùng là chọn mẫu bằng kiểm chứng. Khi một bất biến tiềm năng không thỏa mãn một trong ba tính chất của bất biến, bộ kiểm chứng đưa ra phản ví dụ rồi thêm vào như một mẫu mới.

Phương pháp dự đoán và kiểm tra - Guess and check

Công cụ Zilu được xây dựng dựa trên phương pháp *"guess and check"*. Việc này bắt đầu với sinh ra một tập giá trị V (hay còn gọi là mẫu - *sample*) và phân loại chúng vào các nhóm khác nhau, có thể là một nhóm thỏa mãn bất biến vòng lặp và nhóm còn lại gồm các giá trị không thỏa mãn. Một thuật toán học tập sau đấy được sử dụng để khái quát hóa các giá trị thỏa mãn từ đó đoán ra công thức bất biến vòng lặp có thể. Các công thức đó sẽ được kiểm tra sử dụng kĩ thuật kiểm chứng chương trình (ví dụ như thực thi tượng trưng) để xem liệu chúng có thỏa mãn ba điều kiện của bất biến vòng lặp đã được nêu ở Chương 1. Nếu có một điều kiện bị vi phạm, ta thu được phản ví dụ thể hiện rằng bất biến thu được là không chính xác.

[4]ZILU repo. https://github.com/lijiaying/zilu, 2017

Chương 5

Kết quả thực nghiệm

5.1 Điều kiện thực nghiệm

Trong chương này, khóa luận sẽ trình bày kết quả thực nghiệm của VTSEinv với các phương pháp xử lý vòng lặp cũ và các công cụ khác trên bộ dữ liệu được lấy từ cuộc thi SV-COMP là **ReachSafety/Loop-acceleration**. SV-COMP (Competition on Software Verification) [1] là một cuộc thi uy tín được tổ chức thường niên dành cho các công cụ kiểm chứng nhằm tìm ra kĩ thuật và công cụ có khả năng giải quyết được nhiều bài toán trong thời gian ngắn nhất. Bộ dữ liệu được sử dụng để so sánh bao gồm các chương trình viết bằng ngôn ngữ C. Các bài toán hầu hết là các chương trình chứa một vòng lặp *while* đơn, tập trung kiểm tra các tính chất của vòng lặp. Đây là bộ thực nghiệm để SV-COMP kiểm tra khả năng xử lý vòng lặp của các công cụ tham gia.

Mỗi bài toán là một chương trình bao gồm mã nguồn và điều kiện người dùng cần kiểm tra, có định dạng tương tự như được trình bày ở phần 3.2. Kết quả đưa ra của mỗi bài có thể là một trong ba trường hợp:

- *True*: Điều kiện người dùng được thỏa mãn, không có đường thực thi bị vi phạm
- *False*: Điều kiện nhập vào bị vi phạm và có một phản ví dụ được đưa ra.
- *Timeout/Overflow*: Công cụ không thể đưa ra kết quả trong thời gian quy định

[1] https://sv-comp.sosy-lab.org/2019/

hoặc tràn bộ nhớ.

Các công cụ tham gia thực nghiệm bao gồm VTSEinv, VTSE [15], VeriAbs [5] và CBMC [11]. Trong đó, VeriAbs là công cụ kiểm chứng mã nguồn C/C++ giành giải Bạc SV-COMP 2018 và giải Vàng SV-COMP 2019 ở hạng mục ReachSafety. CBMC là một trong những công cụ đi đầu trong kiểm chứng phần mềm sử dụng kĩ thuật *Bounded Model Checking*. CBMC cũng tham gia SV-COMP trong nhiều năm và đạt giải Vàng năm 2015. Sau đó, rất nhiều các công cụ kiểm chứng đã được phát triển dựa trên nền tảng của CBMC như JBMC, ESBMC, 2LS, ...

5.2 Kết quả thực nghiệm

Kết quả thực nghiệm được trình bày ở Bảng 5.1 gồm các phần so sánh với phiên bản VTSE cũ và công cụ khác trên bộ bài toán **Loop-acceleration**. Thực nghiệm được thực hiện trên máy Intel i5-5200U 2.2GHz x 4, 8GB RAM, với thời gian thực hiện bài toán tối đa là 500 giây.

Trong bảng thực nghiệm, kết quả đúng của bài toán dựa theo SV-COMP và giới hạn số lần vòng lặp thực thi được thể hiện lần lượt ở cột *ER - Expected Result* và *Loop bound*. Mỗi công cụ khi giải sẽ cho đầu ra ở cột R và kèm theo là thời gian thực hiện tính theo giây ở T. VTSE (phiên bản trước đây) và CBMC được chạy thực nghiệm với hai phương án dưới đây:

- *Unfold I*: Sử dụng phương pháp gỡ vòng lặp với số lần gỡ lặp là 100 lần.
- *Unfold II*: Sử dụng phương pháp gỡ vòng lặp với số lần gỡ lặp được bài toán đưa ra *(loop bound)*. Trong trường hợp vòng lặp không có giới hạn *(unbounded loop)*, mặc định số lần gỡ là 100.

Trong khi đó, VTSEinv sử dụng tùy chọn *Invariant* - kiểm chứng dùng bất biến và VeriAbs khi xử lý vòng lặp dùng kỹ thuật *k-induction* trong BMC kết hợp với bất biến vòng lặp.

Về kết quả, so với VTSE trước đây và CBMC, VTSEinv có số lượng bài toán giải đúng và thời gian thực hiện tốt hơn. Khi thực hiện gỡ đúng số vòng lặp, VTSE luôn ra kết quả đúng (trừ với bài toán không có giới hạn lặp), tuy nhiên sẽ gặp phải

vấn đề *Timeout* khi số lần gỡ là quá lớn. Giảm bớt số lần gỡ giải quyết được vấn đề quá thời gian, tuy nhiên lại đưa ra kết quả không chính xác *(unsound)*, do khi này công thức ràng buộc được sinh ra là xấp xỉ dưới của chương trình. Trong khi đó, VTSEinv có số bài giải đúng là **14/17** bài, tốt hơn nhiều VTSE và CBMC khi hai công cụ này chỉ giải được 7 hoặc 8 bài toán. Vì công thức bất biến là xấp xỉ trên, chỉ có 3 bài toán đưa ra kết quả báo động giả *(false alarm)*, còn lại đều được giải với tốc độ nhanh, tổng thời gian để kiểm chứng cả 17 chương trình chỉ là **0.385(s)**. Với bài toán khi số lần gỡ lặp không cao, và tất cả đều có thể đưa ra kết quả đúng như *multival_1_1.c*, VTSEinv cũng chỉ mất **0.014(s)** khi VTSE và CBMC lần lượt cần **12.267(s)** và **56.248(s)**. Có thể nói, VTSEinv đã cải thiện được kết quả của VTSE khi kiểm chứng các bài toán có vòng lặp.

Khi so sánh với VeriAbs, có thể thấy VTSEinv kém về số lượng bài giải được, đồng thời VeriAbs luôn đưa ra kết quả đúng khi kiểm chứng. Tuy nhiên, thời gian để giải một bài toán của VTSEinv là nhanh hơn rất nhiều VeriAbs. Ví dụ, trong bài *mono4_1.c*, VTSEinv chỉ mất **0,013(s)** trong khi VeriAbs không thể đưa ra kết quả trong thời gian **500(s)**. Điều này dẫn đến tổng thời gian kiểm chứng của VTSEinv chỉ là **0.385(s)** so với **3154.8(s)** của VeriAbs.

Có thể thấy, sử dụng bất biến vòng lặp luôn cho kết quả chính xác khi chứng minh được chương trình là đúng đắn (bài *multivar_1_1.c* và *simple_2_2.c*). Tuy nhiên VTSEinv có hạn chế là đưa ra kết quả *false alarm*, nghĩa là báo có lỗi trong khi chương trình đúng. Tuy nhiên, trong kiểm chứng phần mềm, việc đưa ra kết quả là *unsound*, nghĩa là chứng minh chương trình đấy đúng trong khi có lỗi, có hậu quả nặng hơn rất nhiều so với *false alarm*. VTSEinv đã xử lý được vấn đề này của CBMC và VTSE khi hai công cụ này sử dụng phương pháp gỡ vòng lặp. VTSEinv cũng có sự khác biệt lớn về tốc độ so với VeriAbs. Để không đưa ra các kết quả *false alarm*, bất biến vòng lặp sinh ra cần được làm mịn để càng sát với công thức ràng buộc đúng của chương trình càng tốt. Với một phương pháp sinh bất biến vòng lặp đủ tốt, VTSEinv sẽ có thể vừa chiếm ưu thế về tốc độ, vừa cho kết quả chính xác hơn.

Bảng 5.1: Kết quả bộ thực nghiệm Loop-acceleration

ID	Chương trình	ER	Loop bound	VTSEinv		VTSE				CBMC				VeriAbs	
				Invariant		Unfold I		Unfold II		Unfold I		Unfold II			
				R	T (s)	R	T (s)	R	T (s)	R	T (s)	R	T (s)	R	T (s)
1	const_1_1	T	1024	**F**	**0.123**	T	3.042	T	6.072	T	0.143	T	0.24	T	15.301
2	functions_1_1	F	10000	F	0.023	**T**	**1.344**	O	*500*	**T**	**0.197**	F	1.729	F	66.014
3	functions_1_2	T	10000	**F**	**0.013**	T	0.958	O	*500*	T	0.154	T	1.861	O	*500*
4	mono1_1	F	1E+06	F	0.012	F	0.956	O	*500*	**T**	**1.91**	**T**	89.492	F	361.686
5	mono3_1	F	1E+06	F	0.024	**T**	**1.006**	O	*500*	**T**	**1.928**	**T**	131.062	F	362.33
6	mono4_1	F	1E+06	F	0.013	F	1.327	O	*500*	**T**	**1.853**	**T**	100.396	O	500
7	mono5_1	F	-	F	0.013	**T**	**2.63**	**T**	**2.704**	**T**	**1.293**	**T**	78.772	F	364.858
8	mono6_1	F	-	F	0.024	**T**	**0.981**	**T**	**1.003**	**T**	**1.387**	**T**	82.862	F	366.051
9	multivar_1_1	T	1024	**T**	**0.014**	T	1.039	T	12.267	T	0.743	T	56.248	T	8.509
10	multivar_1_2	F	100	F	0.03	**T**	**1.002**	F	3.594	F	0.482	F	0.368	F	7.945
11	phases_1_2	F	1000	F	0.012	**T**	**0.969**	F	11.919	**T**	**0.143**	**T**	**0.884**	F	360.001
12	simple_1_1	F	10000	F	0.012	**T**	**1.438**	O	*500*	**T**	**0.146**	**T**	**0.792**	F	205.325
13	simple_2_1	T	10000	**T**	**0.022**	T	1.03	O	*500*	T	0.354	O	*500*	T	7.067
14	simple_2_2	F	10000	F	0.012	**T**	**1.097**	O	*500*	F	0.323	O	*500*	F	6.105
15	underapp_1_1	F	6	F	0.012	F	0.831	F	0.807	F	0.143	F	0.1	F	9.173
16	underapp_2_1	F	6	F	0.013	F	1.372	F	0.754	**T**	**0.139**	F	0.107	F	6.091
17	underapp_2_2	T	6	**F**	**0.013**	T	1.265	T	0.672	T	0.138	T	0.1	T	8.352
	Số bài đúng			**14**		**9**		**7**		**8**		**7**		**15**	
	Tổng thời gian				**0.385**		**22.287**		**4039.79**		**11.476**		**1545.01**		**3154.81**

False alarm Unsound Timeout/Overflow

ER (Expected result): Kết quả đúng, Loop bound: Giới hạn vòng lặp, R: Kết quả của công cụ, T(s): Thời gian(s)

T: True, F: False, O: Time out/Overflow

Kết luận

Khóa luận đã tìm hiểu và trình bày phương pháp sinh bất biến vòng lặp cũng như cách thức kết hợp bất biến với thực thi tượng trưng để kiểm chứng chương trình C/C++. Đồng thời, công cụ VTSEinv, phiên bản cải tiến của VTSE đã được cài đặt để tăng khả năng kiểm tra các chương trình có chứa vòng lặp. VTSEinv bao gồm các thành phần trừu tượng hóa như trước đây tích hợp với công cụ con InvaGen để sinh bất biến vòng lặp. Chiến lược sinh bất biến được nghiên cứu và cài đặt là bổ đề Farkas kết hợp với giải ràng buộc.

Với bộ thực nghiệm *Loop-acceleration*, VTSEinv đã cho kết quả khả quan khi so sánh với hai công cụ tham gia thi SV-COMP có chiến lược xử lý vòng lặp khác là CBMC và VeriAbs. So với CBMC, VTSEinv giải được gần gấp đôi số bài toán trong thời gian nhanh hơn rất nhiều. Không chỉ vậy, phương pháp dùng bất biến đã khắc phục được nhược điểm sinh ra kết quả *unsound* do việc đánh giá xấp xỉ dưới của CBMC. So với VeriAbs, công cụ cũng áp dụng bất biến vào một phần của quá trình xử lý vòng lặp thì VTSEinv giải được ít hơn hai bài toán, tuy nhiên vẫn chiếm ưu thế về tốc độ thực thi.

Trong tương lai, để cải thiện kết quả của VTSEinv thì việc tìm kiếm hoặc thắt chặt bất biến vòng lặp là rất quan trọng. Hai phương pháp sinh bất biến đang nổi bật hiện nay đã được tìm hiểu và trình bày trong Chương 4. Đồng thời, VTSE cũng có thể mở rộng khả năng ứng dụng bằng cách tối thiểu hóa các ràng buộc của chương trình, tiếp tục phát triển để xử lý các tính chất khác của mã nguồn C/C++ như dữ liệu mảng, dữ liệu với bộ nhớ động, ...

Tài liệu tham khảo

[1] Beyer, D., Keremoglu, M.E. (2011) *CPAchecker: a tool for configurable software verifi-cation.* In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp.184–190. Springer, Heidelberg.

[2] Chimdyalwar, B., Kumar, S., Shrotri, U. (2013) Precise range analysis on large industrycode. In: Proceedings of the 2013 9th Joint Meeting on Foundations of SoftwareEngineering, pp. 675–678. ACM

[3] Cousot, P., Halbwachs, N. (1978) *Automatic Discovery of Linear Restraints among Variables of a Program.* In: POPL, ACM 84–96

[4] Clarke E.M., Henzinger T.A., Veith H. (2018) *Introduction to Model Checking.* In: Clarke E., Henzinger T., Veith H., Bloem R. (eds) Handbook of Model Checking. Springer, Cham

[5] Darke P. et al. (2018) *VeriAbs: Verification by Abstraction and Test Generation.* In: Beyer D., Huisman M. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2018. Lecture Notes in Computer Science, vol 10806. Springer, Cham

[6] E en, N., Soensson, N. (2004) *An extensible SAT-solver.* In: Giunchiglia, E., Tacchella,A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg

[7] Furia, C. A., Meyer, B., and Velder, S. (2012). *Loop invariants: analysis, classification, and examples.* ACM Comput. Surv. V, N, Article A (YYYY), 51 pages.

[8] Gupta, A., Rybalchenko, A. (2009) *Invgen: An efficient invariant generator.* In: Computer Aided Verification, Springer 634–640

[9] Henzinger, T., Jhala, R., Majumdar, R., Sutre, G. (2003) *Software verification with BLAST.* In: International conference on Model checking software. 235–239

[10] King, J.C. (1976) *Symbolic execution and program testing.* Commun. ACM 19(7), 385–394

[11] Kroening, Daniel & Tautschnig, Michael. (2014). CBMC – C Bounded Model Checker. 8413. 389-391. 10.1007/978-3-642-54862-8_26.

[12] Li, Jiaying & Sun, Jun & Li, Li & Le, Quang Loc & Lin, Shang-Wei. (2017). *Automatic loop-invariant generation and refinement through selective sampling.* 782-792. 10.1109/ASE.2017.8115689.

[13] McMillan, K. (2003) *Interpolation and sat-based model checking.* In: Computer Aided Verification, Springer 1–13

[14] Michael Colon, Sriram Sankaranarayanan, and Henny Sipma (2003) *Linear Invariant Generation using Non-linear Constraint Solving*, in Computer-aided Verification (CAV).

[15] N Thi Van Anh, T Van Khanh, N Thi Thuy (2018) *VTSE – Verification Tool based on Symbolic Execution.* In: VNU-UET Technical Report http://eprints.uet.vnu.edu.vn/eprints/3053/

[16] Păsăreanu, C.S. & Visser, W. Int J, (2009) *A survey of new trends in symbolic execution for software testing and analysis*, Software Tools Technology Transfer 11: 339. https://doi.org/10.1007/s10009-009-0118-1

[17] Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, Henri Hansen (2017) *FiB: squeezing loop invariants by interpolation between Forward/Backward predicate transformers.* ASE 2017: 793-803