ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Đoàn Duy Tùng

NGHIÊN CỨU GIẢI PHÁP KIỂM THỬ ĐƠN VỊ TỰ ĐỘNG CHO MÃ NGUỒN C/C++ CHỨA HÀM THIẾU ĐỊNH NGHĨA

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

HÀ NỘI - 2023

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Đoàn Duy Tùng

NGHIÊN CỨU GIẢI PHÁP KIỂM THỬ ĐƠN VỊ TỰ ĐỘNG CHO MÃ NGUỒN C/C++ CHỨA HÀM THIẾU ĐỊNH NGHĨA

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

Cán bộ hướng dẫn: PGS. TS. Phạm Ngọc Hùng

HÀ NỘI - 2023

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Doan Duy Tung

RESEARCH ON AN AUTOMATED UNIT TESTING SOLUTION FOR C/C++ SOURCE CODE CONTAINING MISSING FUNCTION DEFINITIONS

BACHELOR'S THESIS
Major: Information Technology

Supervisor: Assoc. Prof. Dr. Pham Ngoc Hung

Hanoi - 2023

LỜI CAM ĐOAN

Em xin cam đoan: Khóa luận tốt nghiệp với đề tài "Nghiên cứu giải pháp kiểm thử đơn vị tự động cho mã nguồn C/C++ chứa hàm thiếu định nghĩa" trong báo cáo này là của em. Những gì em viết ra không có sự sao chép từ các tài liệu, không sử dụng kết quả của người khác mà không trích dẫn cụ thể. Đây là công trình nghiên cứu cá nhân em tự phát triển, không sao chép mã nguồn của người khác. Nếu vi phạm những điều trên, em xin chấp nhận tất cả những truy cứu về trách nhiệm theo quy định của Trường Đại học Công nghệ - ĐHQGHN.

Hà Nội, ngày 27 tháng 12 năm 2023

Sinh viên

Đoàn Duy Tùng

LỜI CẢM ƠN

Lời đầu tiên cho phép em bày tỏ lòng biết ơn chân thành đến Khoa Công nghệ Thông tin - Trường Đại học Công nghệ, ĐHQGHN, nơi em đã có cơ hội tiếp xúc với những kiến thức mới mẻ và được học hỏi từ những người giáo viên xuất sắc khác.

Em xin gửi lời cảm ơn chân thành nhất đến thầy Phạm Ngọc Hùng, người đã là nguồn động viên và hướng dẫn quý báu trong suốt thời gian em học tập, làm việc tại phòng nghiên cứu. Sự hướng dẫn tận tâm và sự hỗ trợ của thầy là động lực lớn giúp em vượt qua những thách thức trong hành trình tìm hiểu, nghiên cứu và hoàn thiện khóa luận tốt nghiệp.

Cuối cùng, em xin được gửi lời cảm ơn chân thành tới anh Nguyễn Tùng Lâm, anh Nguyễn Vũ Bình Dương cũng như toàn thể các anh chị và các bạn tại Phòng thí nghiệm đảm bảo chất lượng phần mềm (Khoa Công nghệ thông tin, Trường Đại học Công nghệ, ĐHQGHN) đã luôn ủng hộ, động viên em trong quá trình hoàn thành khóa luận. Đặc biệt, em xin chân thành cảm ơn anh Trần Trọng Năm và các anh chị tại đơn vị FPT-GAM đã giúp đỡ, cho phép em sử dụng tài nguyên của đơn vị trong quá trình làm khóa luận.

Chúc mọi người luôn luôn vui vẻ và gặt hái được nhiều thành công trong cuộc sống.

TÓM TẮT

Tóm tắt: Kiểm thử các dự án nhúng C/C++ là bài toán nhận được sự quan tâm của các công ty phát triển phần mềm cũng như cộng đồng nghiên cứu trên thế giới. Nhằm phát hiện sớm các lỗi tiềm ẩn và nâng cao độ tin cậy của mã nguồn, quá trình kiểm thử thường được triển khai từ sớm, khi mã nguồn chưa đầy đủ. Để giảm chi phí, thời gian và công sức trong việc kiểm thử đơn vị, các phương pháp kiểm thử tự động đã và đang được áp dụng rộng rãi. Kiểm thử phần mềm tự động bao gồm hai hướng tiếp cận chính là kiểm thử tĩnh và kiểm thử động. Kế thừa hai hướng tiếp cận trên, kiểm thử tượng trưng động được đề xuất với nhiều cải tiến và thu được những thành tựu nhất định. Tuy nhiên, khi áp dụng các hướng tiếp cận này, một số hạn chế phát sinh do mã nguồn có thể chứa hàm thiếu định nghĩa. Khóa luận đề xuất một phương pháp nhằm tự động phát hiện và xử lý các hàm thiếu định nghĩa. Ý tưởng chính để xử lý hàm thiếu định nghĩa là sử dụng trình biên dịch và đồ thị cấu trúc mã nguồn để tìm kiếm các hàm thiếu định nghĩa rồi sinh thân hàm giả cho chúng. Khóa luận cũng đề xuất một phương pháp để xử lý lời gọi phương thức, giúp tăng độ phủ kiểm thử các đơn vị chứa nhiều tương tác giữa các đối tượng. Phương pháp đề xuất đã được cài đặt và tích hợp vào công cụ AKAUTAUTO - sản phẩm hợp tác nghiên cứu giữa Phòng thí nghiệm đảm bảo chất lượng phần mềm (khoa Công nghệ thông tin, Trường Đại học Công nghệ, ĐHQGHN) và đơn vị FPT-GAM (FPT Global Automative & Manufactoring) để tiến hành thực nghiệm đánh giá tính hiệu quả của phương pháp. Kết quả thực nghiệm cho thấy rằng phương pháp có khả năng xử lý các hàm thiếu định nghĩa trong thời gian ngắn và có khả năng sinh dữ liệu kiểm thử tự động với độ phủ cao trên một số mã nguồn mở và mã nguồn thực tế thuộc đơn vị FPT-GAM. Những kết quả tích cực của phương pháp đề xuất cho thấy tiềm năng ứng dụng thực tế để kiểm thử đơn vị các dự án C/C++ song song với quá trình phát triển.

Từ khóa: *Kiểm thử tự động, kiểm thử tượng trưng động, kiểm thử dòng điều khiển, hàm thiếu định nghĩa, sinh giả lập mã nguồn tự động, lời gọi phương thức*

ABSTRACT

Abstract: Testing embedded C/C++ projects is a challenge that has drawn the attention of software development companies as well as the research community worldwide. In order to detect hidden errors early and enhance the reliability of the source code, the testing process is often deployed early, when the source code is not yet complete. To reduce costs, time, and effort in unit testing, automated testing methods have been widely applied. Automated software testing comprises two main approaches: static testing and dynamic testing. Inheriting from these two approaches, concolic testing is proposed with various improvements and has achieved certain achievements. However, when applying these approaches, some limitations arise due to the possibility that the source code may contain undefined functions. This thesis proposes a method to automatically detect and handle undefined functions. The main idea for handling undefined functions is to use the compiler and the source code structure graph to search for undefined functions and then generate dummy functions for them. The thesis also proposes a method to handle method calls, helping to increase test coverage for units with multiple interactions between objects. The proposed method has been implemented and integrated into the AKAUTAUTO tool, a collaborative research product between the Software Quality Assurance Laboratory (Faculty of Information Technology, University of Engineering and Technology, VNU) and FPT-GAM (FPT Global Automotive & Manufacturing) to conduct experiments evaluating the effectiveness of the method. Experimental results show that the method can handle undefined functions in a short time and can generate automatic test data with high coverage on some open-source code and actual source code at FPT-GAM. The positive results of the proposed method demonstrate its potential for practical application in unit testing for C/C++ projects alongside the development process.

Keywords: *Automated test data generation, concolic testing, control flow testing, function prototype, unimplemented function, automated stub generation, method call*

Mục lục

Danh sách hình vẽ

Danh sách bảng

Danh sách thuật toán

Danh sách đoạn mã

Thuật ngữ

Thuật ngữ		Ý nghĩa
Từ viết tắt	**Từ đẩy đủ**	
AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
CFG	Control Flow Graph	Đồ thị dòng điều khiển
LOC	Lines of Code	Dòng mã
HTML	Hypertext Markup Language	Ngôn ngữ đánh dấu siêu văn bản
SMT	Satisfiability Modulo Theories	Lý thuyết mô-đun thỏa mãn
	Stub	Giả lập mã nguồn

Đặt vấn đề

Kiểm thử và đảm bảo chất lượng phần mềm là một phần quan trọng trong quá trình phát triển phần mềm [1]. Ngày nay, với sự phát triển mạnh mẽ của công nghệ, vai trò của kiểm thử ngày một thiết yếu hơn khi cuộc sống con người gắn liền với các phần mềm, các thiết bị điện tử thông minh bởi chúng hỗ trợ, đáp ứng các nhu cầu thiết yếu trong cuộc sống. Trong đó, các dự án nhúng viết bằng C/C++ đã và đang tạo ra các sản phẩm phần mềm thông minh cần thiết cho người dùng [16]. Với sự phát triển của ngành công nghiệp ô tô, các dự án nhúng C/C++ đang dần xuất hiện trong các phần mềm điều khiển trên xe hơi [8]. Để đảm bảo an toàn cho người dùng khi sử dụng các sản phẩm phần mềm, chất lượng mã nguồn cần được đảm bảo một cách chính xác theo các tiêu chuẩn nghiêm ngặt như ISO (ISO-9126 [2], ISO-26262 [11]) hay MISRA-C[1]. Thực tế cho thấy rằng phần lớn các dự án nhúng C/C++ có cấu trúc mã nguồn phức tạp, kích thước lớn và có nhiều sự tương tác giữa các mô-đun. Bởi vậy, để phát hiện sớm các lỗi phát sinh, quá trình kiểm thử thường được triển khai từ sớm, ngay cả khi mã nguồn chưa đầy đủ, mới chỉ có các lớp giao diện được đề ra bởi lập trình viên. Phương pháp kiểm thử thủ công bộc lộ nhiều khó khăn, tiêu tốn nhiều thời gian và chi phí khi triển khai kiểm thử từ sớm như vậy bởi quá trình kiểm thử đi kèm việc tạo giả lập mã nguồn cho các đơn vị chưa được cài đặt. Do đó, việc kiểm thử tự động cho các dự án nhúng C/C++, ngay cả khi mã nguồn chưa đầy đủ, đã và đang trở thành thách thức lớn được không chỉ cộng đồng nghiên cứu mà còn được các công ty phát triển phần mềm quan tâm.

Kiểm thử phần mềm tự động đã và đang được áp dụng rộng rãi nhằm giảm thiểu chi phí của quá trình kiểm thử. Các kỹ thuật kiểm thử tự động dựa trên đồ thị dòng điều khiển (Control Flow Graph - CFG) và đồ thị dòng dữ liệu là hai hướng nghiên cứu chính được nhiều nhà nghiên cứu quan tâm. Trong đó, kỹ thuật kiểm thử tự động dựa trên CFG

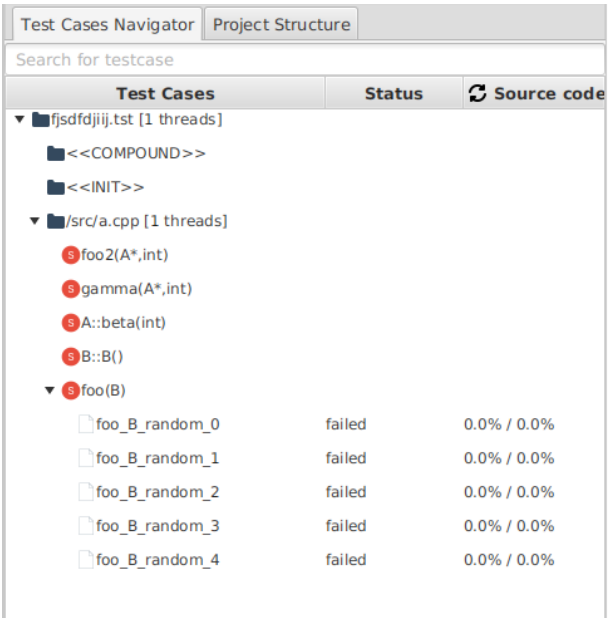
[1]https://misra.org.uk/misra-c/

được sử dụng phổ biến với hai hướng tiếp cận chính là kiểm thử tĩnh [3, 7] và kiểm thử động [10, 21]. Mỗi hướng tiếp cận đều có ưu, nhược điểm riêng nhưng nhìn chung đều mang lại hiệu quả cao trong việc đảm bảo chất lượng phần mềm. Cụ thể, kiểm thử tĩnh tập trung vào việc sinh dữ liệu kiểm thử dựa trên quá trình phân tích mã nguồn mà không cần chạy chương trình. Kĩ thuật kiểm thử tĩnh cung cấp nhiều lợi ích khi kĩ thuật này có thể áp dụng sớm trong vòng đời phát triển phần mềm, từ đó phát hiện sớm các lỗi phát sinh, các sai sót trong đặc tả, giúp giảm thời gian và công sức kiểm thử. Tuy nhiên, kiểm thử tĩnh có một số nhược điểm như yêu cầu nhiều tài liệu liên quan, không phát hiện được lỗi tiềm ẩn trong quá trình chạy. Do vậy, hướng tiếp cận kiểm này khó có thể tiến hành tự động hoàn toàn. Kiểm thử động là hướng tiếp cận kiểm thử dựa trên việc thực thi chương trình để phát hiện lỗi tiềm ẩn, sai sót trong cài đặt so với đặc tả. Hai hướng tiếp cận có những ưu, nhược điểm bổ trợ cho nhau nên chúng thường được áp dụng đồng thời trong quá trình kiểm thử.

Kế thừa hai hướng tiếp cận trên, Godefroid và cộng sự đã đề xuất phương pháp kiểm thử tượng trưng động [17], cài đặt trong công cụ DART [9], nhằm khắc phục nhược điểm và tận dụng những ưu điểm của kiểm thử tĩnh và kiểm thử động. Quá trình sinh dữ liệu kiểm thử trong phương pháp này bao gồm ba pha chính: (i) sinh dữ liệu kiểm thử khởi tạo, (ii) thực thi dữ liệu kiểm thử và phân tích đường thi hành, và (iii) sinh dữ liệu kiểm thử có hướng. Phương pháp bắt đầu bằng việc sinh ngẫu nhiên dữ liệu kiểm thử khởi tạo. Sau khi thực thi dữ liệu kiểm thử ngẫu nhiên, đơn vị kiểm thử có thể tồn tại một số câu lệnh và nhánh chưa được viếng thăm. Để giải quyết vấn đề này, kiểm thử tượng trưng động chọn một đường thi hành chưa được viếng thăm trên CFG của đơn vị kiểm thử. Điều này cho phép phương pháp sinh dữ liệu kiểm thử có hướng, nhằm phủ đường thi hành đó. Phương pháp này có khả năng đạt được độ phủ mã nguồn cao hơn so với các hướng tiếp cận truyền thống. Trong thực tế, kiểm thử tượng trưng động đã và đang được ứng dụng trong nhiều dự án với mục tiêu tăng cường độ phủ mã nguồn và phát hiện lỗi chương trình với chi phí thấp. Kiểm thử tượng trưng động sau đó thu hút sự quan tâm lớn từ cộng đồng nghiên cứu và các công ty phát triển phần mềm. Minh chứng là sự ra đời của một số công cụ kiểm thử dựa trên kĩ thuật tượng trưng động như CUTE [18], PathCrawler [22], CAUT [19], SDART [15], VFP [21], v.v. Hơn nữa, một số phương pháp khác như EXE [6], KLEE [4], hay KLOVER [14], thay vì thực thi chương trình với dữ liệu đầu vào được sinh thủ công hoặc ngẫu nhiên, đã áp dụng kiểm thử tượng trưng động để sinh dữ liệu đầu vào có giá trị tượng trưng có thể gây lỗi chương trình.

Hiện nay, dù đã có nhiều nghiên cứu xoay quanh kiểm thử tượng trưng động cho mã nguồn C/C++ nhưng vẫn tồn tại một số hạn chế khi áp dụng phương pháp trong quá trình kiểm thử đơn vị. Trong đó, một số hạn chế bắt nguồn từ việc mã nguồn kiểm thử chưa đầy đủ. Trong vòng đời phát triển phần mềm, mã nguồn trong pha cài đặt thường bắt đầu với việc đội ngũ lập trình viên thiết kế, đặc tả và cài đặt các lớp giao diện, kiểu dữ liệu trừu tượng, v.v. Mục đích của việc này là để các thành viên trong đội nhóm phát triển biết được dịch vụ sẽ được cung cấp bởi các thành phần trong mã nguồn, qua đó đẩy nhanh tốc độ cài đặt thành phần được giao. Điều này dẫn tới hạn chế đầu tiên đó là mã nguồn có thể chứa hàm thiếu định nghĩa. Hàm thiếu định nghĩa là những hàm mà có nguyên mẫu được khai báo, nhưng không có định nghĩa hàm. Điều này có thể xuất hiện khi nhóm phân chia công việc và mỗi thành viên chịu trách nhiệm về việc triển khai một số hàm cụ thể. Tuy nhiên, việc quản lý và theo dõi những hàm này trong quá trình phát triển và kiểm thử có thể gây khó khăn, đặc biệt khi số lượng hàm thiếu định nghĩa lớn. Bài toán trở nên phức tạp hơn khi áp dụng kiểm thử tự động bởi các công cụ kiểm thử tự động thường đòi hỏi mã nguồn đầy đủ để thực thi các ca kiểm thử. Chương trình chứa các hàm thiếu định nghĩa có thể biên dịch được nhưng không thể liên kết các tệp đối tượng để tạo thành một tệp thực thi. Theo nghiên cứu của khóa luận, hiện nay chưa có phương pháp tự động nào giúp xử lý các hàm thiếu định nghĩa. Điều này khiến quá trình kiểm thử mất nhiều thời gian và công sức hơn.

```
1  class B {
2    int b;
3    int stub(int x); // ERROR
4  }
5  int foo(B param) {
6    if (param.b == 1) {
7      int ret = param.stub(b);
8      if (param.b == 2)
9      return ret;
10   }
11   return 0;
12 }
```



Hình 0.1: Ví dụ sự ảnh hưởng của hàm thiếu định nghĩa khi kiểm thử đơn vị tự động.

Hình 0.1 mô tả ví dụ về sự ảnh hưởng của hàm thiếu định nghĩa khi kiểm thử bởi các công cụ kiểm thử tự động. Trong đó, khi kiểm thử đơn vị cho hàm `foo` trong đoạn mã bên trái với đầu vào là đối tượng `param` - đối tượng của lớp `B`, kết quả thực thi các ca kiểm thử đều là *failed*. Điều này dẫn đến công cụ không thể đánh giá được độ phủ mã nguồn kiểm thử, gây khó khăn trong việc đảm bảo chất lượng từ sớm. Trong đơn vị kiểm thử, đối tượng `param` gọi phương thức `stub` - phương thức chưa được cài đặt thân hàm. Điều này biến phương thức trở thành hàm thiếu định nghĩa. Do đó, khi sử dụng các công cụ kiểm thử tự động để kiểm thử đơn vị này, các ca kiểm thử đều không thể liên kết được, dẫn đến lỗi *failed* như Hình 0.1. Thực trạng trên đã diễn ra trong quá trình phát triển phần mềm thực tế và được các công ty phát triển phần mềm quan tâm bởi sự ảnh hưởng lớn của nó trong quá trình kiểm thử và đảm bảo chất lượng phần mềm.

Hạn chế thứ hai khi áp dụng kĩ thuật kiểm thử tượng trưng động đó là sự phụ thuộc vào các hàm, thành phần khác trong mã nguồn khiến quá trình thực hiện kiểm thử tốn nhiều thời gian song kết quả độ phủ chưa cao. Ở mức độ kiểm thử đơn vị, quá trình kiểm thử nên tập trung vào đơn vị chính mà không cần quan tâm đến các phụ thuộc. Năm 2022, Trần Nguyên Hương và các cộng sự đã giới thiệu phương pháp sinh giả lập mã nguồn (stub) tự động AS4UT [21] nhằm giải quyết vấn đề trên. Tuy nhiên, phương pháp vẫn còn nhiều hạn chế khi chưa giải quyết được lời gọi phương thức. C++ là ngôn ngữ hướng đối tượng nên lời gọi phương thức đóng vai trò cầu nối cung cấp hành vi của một đối tượng. Các lời gọi như vậy có thể thay đổi giá trị thuộc tính của đối tượng gọi. Sự thay đổi này có thể dẫn đến những điều kiện trong đơn vị kiểm thử không thể thỏa mãn được nếu chỉ đơn thuần áp dụng phương pháp AS4UT. Bài toán trở nên phức tạp hơn khi phương thức được gọi có thể là hàm thiếu định nghĩa. Do đó, cần thiết phải nghiên cứu và phát triển giải pháp kiểm thử đơn vị tự động hiệu quả cho mã nguồn C/C++ chứa hàm thiếu định nghĩa.

Để xử lý các hạn chế trên, khóa luận đề xuất một phương pháp nhằm tự động phát hiện và xử lý các hàm thiếu định nghĩa, đồng thời tự động sinh stub cho các lời gọi hàm trong quá trình kiểm thử đơn vị. Ý tưởng chính của phương pháp là cải tiến phương pháp kiểm thử đơn vị tự động truyền thống với hai bổ sung chính gồm (i) quá trình xử lý hàm thiếu định nghĩa sau quá trình phân tích mã nguồn, và (ii) bổ sung phương pháp xử lý lời gọi phương thức trong quá trình xử lý lời gọi hàm khi sinh stub tự động. Trong đó, quá trình xử lý hàm thiếu định nghĩa sẽ xác định danh sách các nguyên mẫu hàm thiếu định nghĩa cần quan tâm rồi sinh thân hàm giả cho chúng. Quy trình xử lý nguyên mẫu hàm

được tách thành hai quy trình con gồm xử lý nguyên mẫu hàm cơ bản và nguyên mẫu hàm ảo. Sau đó, phương pháp đồng thời áp dụng kỹ thuật kiểm thử tượng trưng động và phương pháp sinh stub tự động để sinh dữ liệu kiểm thử tự động cho các đơn vị được kiểm thử. Điều này cho phép phương pháp đề xuất rút ngắn được thời gian chuẩn bị môi trường kiểm thử và đồng thời đạt được độ phủ mã nguồn cao hơn khi kiểm thử các dự án chứa mã nguồn thiếu định nghĩa. Qua đó cho thấy khả năng ứng dụng thực tiễn của phương pháp đề xuất trong các dự án thực tế để giảm thiểu chi phí kiểm thử và nâng cao chất lượng chương trình.

Phần còn lại của khóa luận được trình bày như sau. Chương 1 trình bày một số kiến thức cơ sở liên quan đến chủ để sinh dữ liệu kiểm thử tự động cho các dự án C/C++, sinh giả lập mã nguồn tự động và làm rõ khái niệm hàm thiếu định nghĩa. Tiếp theo, Chương 2 chia sẻ chi tiết về phương pháp kiểm thử đơn vị tự động cho mã nguồn C/C++ chứa hàm thiếu định nghĩa. Chương 3 mô tả công cụ thực nghiệm và một số kết quả thực nghiệm đánh giá tính hiệu quả của phương pháp đề xuất. Cuối cùng, Chương Kết luận chia sẻ một số đúc kết về phương pháp đề xuất và thảo luận hướng nghiên cứu tiếp theo.

Chương 1

Kiến thức cơ sở

Trước khi đi sâu hơn về giải pháp kiểm thử đơn vị tự động cho mã nguồn C/C++ chứa hàm thiếu định nghĩa, chương này sẽ trình bày một số kiến thức cơ bản, quan trọng liên quan đến đề tài, tạo nền tảng cho việc hiểu rõ hơn về các khái niệm và phương pháp được đề cập trong khóa luận. Trước hết, khóa luận trình bày một số khái niệm, thuật ngữ liên quan đến kĩ thuật kiểm thử dòng điều khiển trong kiểm thử phần mềm. Tiếp theo, khóa luận chia sẻ chi tiết hơn về kĩ thuật kiểm thử tượng trưng động và làm rõ cơ chế sinh dữ liệu kiểm thử từ tập ràng buộc. Cuối cùng, khóa luận trình bày một số khái niệm liên quan đến hàm thiếu định nghĩa và mô tả tổng quan phương pháp sinh stub tự động.

1.1. Một số khái niệm, thuật ngữ trong kiểm thử dòng điều khiển

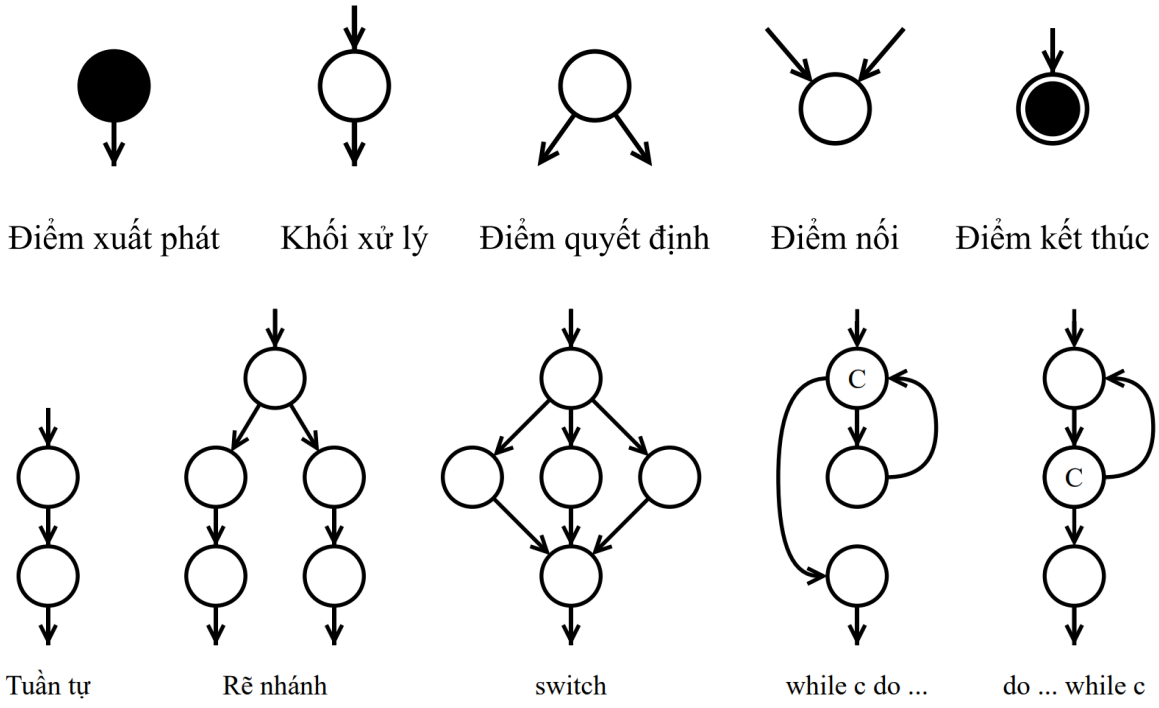
1.1.1. Cây cú pháp trừu tượng và đồ thị dòng điều khiển

Cây cú pháp trừu tượng (Abstract Syntax Tree - AST) là một biểu diễn cấu trúc cú pháp của một đoạn mã nguồn. AST được tạo ra từ quá trình phân tích cú pháp của ngôn ngữ lập trình và thường được sử dụng trong các bước biên dịch và xử lý ngôn ngữ. Về cơ bản, AST là một cách biểu diễn khác của mã nguồn mà trong đó, thay vì dạng văn bản thuần túy, các thành phần mã nguồn sẽ được thể hiện bằng từng thành phần trong cây. Việc sử dụng cây cú pháp trừu tượng trong kiểm thử tự động cho phép ta dễ dàng lấy ra các thông tin cần thiết thay vì phân tích mã nguồn từ dạng văn bản.

Đồ thị dòng điều khiển (Control Flow Graph - CFG) đóng vai trò quan trọng trong phương pháp kiểm thử dòng điều khiển bởi phương pháp này đều có mục tiêu viếng thăm tối đa các đỉnh trên đồ thị. Đồ thị dòng điều khiển có thể được xây dựng dễ dàng từ AST của mã nguồn. Đồ thị dòng điều khiển được định nghĩa như Định nghĩa 2.1.

Định nghĩa 2.1: Đồ thị dòng điều khiển là một đồ thị có hướng gồm các điểm tương ứng với các câu lệnh/nhóm câu lệnh và các cạnh là các dòng điều khiển giữa các câu lệnh/nhóm câu lệnh. Nếu *i* và *j* là các điểm của đồ thị dòng điều khiển thì tồn tại một cạnh từ *i* đến *j* nếu lệnh tương ứng với *j* có thể được thực hiện ngay sau lệnh tương ứng với *i* [1].

Đồ thị dòng điều khiển bao gồm các thành phần chính và các cấu trúc điều khiển phổ biến như trong Hình 1.1. Các thành phần cơ bản gồm điểm xuất phát, khối xử lý, điểm quyết định, điểm nối và điểm kết thúc. Các cấu trúc điều khiển cơ bản như tuần tự, rẽ nhánh, switch, vòng lặp while-do và do-while trong ngôn ngữ C/C++ được mô phỏng dưới dạng các thành phần cơ bản của CFG như Hình 1.1.



Hình 1.1: Các thành phần cơ bản và các cấu trúc điều khiển phổ biến trong CFG [1].

1.1.2. Đường thi hành

Đường thi hành là một tập hợp có thứ tự các đỉnh trên CFG thể hiện thứ tự viếng thăm khi thực thi đơn vị kiểm thử tương ứng. Nói cách khác, đường thi hành biểu diễn các câu lệnh được chạy khi thực thi đơn vị kiểm thử với một bộ tham số đầu vào. Định nghĩa chính xác của đường thi hành như Định nghĩa 2.2.

Định nghĩa 2.2: Đường thi hành là một đường đi từ điểm xuất phát đến điểm kết thúc của CFG, được biểu diễn bằng tập hợp các điểm từ v_1 đến v_n sao cho cứ hai điểm cạnh nhau thì có cạnh nối theo hướng từ trái qua phải. Nếu cạnh (v_i, v_j) là nhánh sai thì biểu thức điều kiện tại điểm v_i được viết dưới dạng phủ định $!v_i$ [1].

Đường thi hành có thể chứa rất nhiều câu lệnh và rất nhiều nhánh điều kiện. Điều này dẫn tới việc một đơn vị kiểm thử có thể chứa đường thi hành không khả thi và có thể bùng nổ số lượng đường thi hành. Một đường thi hành không khả thi khi tồn tại những điều kiện không thể thỏa mãn được. Một ví dụ có thể kể đến đó là câu lệnh so sánh `if(0 == 1)`. Phép so sánh này không bao giờ đúng nên các đỉnh CFG trong nhánh đúng của đỉnh so sánh trên sẽ không thể viếng thăm. Về khả năng bùng nổ số lượng đường thi hành, điều này có thể thấy rõ thông qua các vòng lặp. Các vòng lặp với số lần lặp chưa biết trước có thể tạo ra vô số đường thi hành. Vì vậy việc kiểm thử toàn bộ đường thi hành là một thách thức lớn.

Như vậy, việc sinh dữ liệu kiểm thử cho một đường thi hành tương ứng với quá trình tìm giá trị bộ tham số đầu vào sao cho viếng thăm được toàn bộ đỉnh trên đường thi hành. Trong thực tế, số lượng đường thi hành của một đơn vị kiểm thử có thể rất lớn. Do vậy, việc lựa chọn tập đường thi hành sao cho đạt độ phủ mã nguồn yêu cầu là một thách thức lớn.

1.1.3. Độ phủ mã nguồn

Độ phủ mã nguồn là một đơn vị đo số lượng thành phần trong đơn vị kiểm thử được viếng thăm bởi bộ dữ liệu tham số đầu vào. Các thành phần liên quan có thể là câu lệnh, điểm quyết định, điều kiện con, đường thi hành hay là sự kết hợp của chúng [1]. Bộ kiểm thử có độ tin cậy càng cao khi độ phủ đơn vị kiểm thử càng lớn. Ba loại độ phủ mã nguồn được sử dụng phổ biến trong thực tế gồm C_1, C_2 và C_3 [1]. Một số tác giả sử dụng các khái niệm tương ứng gồm Statement, Branch và MCDC (Modified Condition/Decision

Coverage). Khóa luận sử dụng khái niệm C_1, C_2 và C_3 với định nghĩa như sau:

- Độ phủ C_1: Được hiểu là độ phủ ở mức câu lệnh, tức mỗi câu lệnh trong đơn vị kiểm thử được thực thi ít nhất một lần sau khi chạy tập ca kiểm thử.
- Độ phủ C_2: Được hiểu là độ phủ ở mức điểm quyết định, tức mỗi điểm quyết định trong CFG của đơn vị kiểm thử đều được thực hiện ít nhất một lần cả nhánh đúng và sai sau khi chạy tập ca kiểm thử.
- Độ phủ C_3: Là độ phủ có điều kiện chặt nhất trong ba loại độ phủ. Điều kiện để đảm bảo độ đo này là các điều kiện con thuộc các điều kiện phức tạp tương ứng với các điểm quyết định trong đồ thị dòng điều khiển của đơn vị cần kiểm thử đều được thực hiện ít nhất một lần cả hai nhánh đúng và sai. Với các phần mềm yêu cầu tính đúng đắn cao, việc sử dụng độ đo C_3 để đánh giá mã nguồn là cần thiết.

1.2. Kiểm thử tượng trưng động

Như đã đề cập ở Chương Đặt vấn đề, kiểm thử tượng trưng động là kĩ thuật kiểm thử áp dụng đồng thời kiểm thử tĩnh và kiểm thử động với ba pha chính là sinh dữ liệu kiểm thử ngẫu nhiên, thực thi ca kiểm thử & phân tích đường thi hành và sinh dữ liệu kiểm thử có định hướng. Trong pha đầu tiên, một số dữ liệu kiểm thử được sinh ngẫu nhiên. Các dữ liệu kiểm thử sau đó được thực thi trong pha thứ hai. Nhờ việc thực thi các dữ liệu kiểm thử khởi tạo, kiểm thử tượng trưng động xác định được tập các điểm chưa được viếng thăm và tìm kiếm đường thi hành từ điểm xuất phát đến điểm đó. Một số nhà nghiên cứu lựa chọn đường thi hành ngắn nhất tới một điểm chưa viếng thăm nhằm tối ưu hóa số lượng ca kiểm thử như SDART [15]. Nếu tìm được đường thi hành chưa được viếng thăm, pha thứ ba chuyển đổi đường thi hành thành các ràng buộc sử dụng kỹ thuật thực thi giá trị tượng trưng [5]. Các ràng buộc được biểu diễn dưới dạng biểu thức logic và được chuyển đổi thành đầu vào của bộ giải hệ ràng buộc. Cuối cùng, bộ giải hệ ràng buộc chịu trách nhiệm giải các ràng buộc và cho ra được một nghiệm - dữ liệu kiểm thử mới. Quá trình sinh dữ liệu kiểm thử sau đó lặp lại từ pha thứ hai cho đến khi tất cả đỉnh trong CFG đã được thăm hoặc chương trình chạy quá thời gian quy định.

Kỹ thuật kiểm thử tượng trưng động có khả năng đạt độ phủ cao hơn bởi khả năng sinh dữ liệu có hướng dựa trên ràng buộc của đường thi hành. Tập ràng buộc là tập các

điều kiện tạo ra bởi tập các kiểm quyết định khi áp dụng kỹ thuật thực thi tượng trưng trên một đường thi hành. Nói cách khác, tập ràng buộc chứa các điều kiện mà dữ liệu đầu vào cần thỏa mãn để ca kiểm thử phủ đường thi hành mong muốn. Dữ liệu đầu vào như vậy được gọi là dữ liệu kiểm thử có hướng. Dữ liệu kiểm thử có hướng cho mỗi đường thi hành có thể thu được bằng cách giải hệ ràng buộc. Trong đó, giải hệ ràng buộc là việc tìm nghiệm cho một tập các ràng buộc, biểu diễn bởi các phép toán điều kiện, sao cho với các giá trị biến đầu vào trong nghiệm thì tất cả các ràng buộc được thỏa mãn [20]. Hiện nay, có nhiều thư viện và công cụ hỗ trợ giải hệ ràng buộc trong đó nổi bật là bộ giải Z3[1].

Bộ giải Z3 là một công cụ được phát triển bởi Nhóm Nghiên cứu về Kỹ thuật Phần mềm (RiSE) tại Microsoft Research. Được xây dựng chủ yếu bằng ngôn ngữ C++, Z3 hỗ trợ giải hệ ràng buộc của nhiều loại dữ liệu như số nguyên, số thực, mảng, hàm tượng trưng, vectơ bit, và các biểu thức số học khác nhau. Z3 được thiết kế để hỗ trợ định dạng SMT-LIB[2], một định dạng tiêu chuẩn sử dụng để giải quyết các vấn đề liên quan đến giải ràng buộc. Để sử dụng bộ giải, hệ ràng buộc cần được biểu diễn dưới định dạng SMT-LIB và lưu trữ trong tệp *.smt2. Cấu trúc của tệp này bao gồm ba phần chính: phần khai báo, phần định nghĩa ràng buộc và phần tiện ích. Phần khai báo đầu tiên trong tệp là nơi các biến trong hệ ràng buộc được khai báo theo cú pháp SMT-LIB. Sau đó, các ràng buộc được thêm vào bằng cách sử dụng lệnh assert. Cuối cùng, phần tiện ích cung cấp các cú pháp sẵn có để người dùng thực hiện các thao tác như đặt thời gian giải quyết tối đa, xác định mô hình giải ràng buộc, và nhiều tính năng khác.

1.3. Hàm thiếu định nghĩa

1.3.1. Tổng quan về hàm thiếu định nghĩa

Như đã đề cập ở Chương Đặt vấn đề, hàm thiếu định nghĩa là những hàm mà có nguyên mẫu được khai báo, nhưng không có thân hàm (định nghĩa hàm). Để làm rõ hơn khái niệm này, ta cần hiểu thêm về cách khai báo hàm trong ngôn ngữ C/C++. Về cơ bản, một hàm gồm hai phần chính đó là nguyên mẫu hàm và thân hàm. Trong đó, nguyên mẫu hàm thông báo cho trình biên dịch biết tên hàm, kiểu trả về của hàm và danh sách tham

[1]https://github.com/Z3Prover/z3

[2]http://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf

số mà hàm nhận. Thân hàm là phần nội dung của hàm nằm trong cặp ngoặc {}. Ngôn ngữ C/C++ cung cấp hai cách để người dùng khai báo một hàm (ngoại trừ hàm lambda) gồm khai báo đầy đủ và khai báo tách rời.

```
1 // Full function declaration
2 int foo(int a, int b) { return a + b; }
3 // Function prototype + Function defintion
4 int bar(int a, int b);
5 int echo(double);
6 int main() { bar(1,2); }
7 int bar(int a, int b) { return a - b; }
```

Đoạn mã 1.1: Ví dụ về khai báo hàm đầy đủ và khai báo hàm tách rời.

Đoạn mã 1.1 minh họa ví dụ về khai báo đầy đủ và khai báo tách rời. Dòng 1-3 thể hiện khai báo tách rời của hàm `foo` với nguyên mẫu hàm và thân hàm đều nằm trong cùng một câu lệnh khai báo. Dòng 6 thể hiện khai báo nguyên mẫu hàm của hàm `bar` và cho biết hàm này trả về giá trị kiểu `int` và nhận hai tham số đầu vào kiểu `int`. Dòng 7 cũng thể hiện nguyên mẫu hàm `echo`. Dòng 11-13 thể hiện khai báo định nghĩa hàm (thân hàm) của hàm `bar`.

Nguyên mẫu hàm là một trong những tính năng quan trọng của ngôn ngữ C/C++. Nguyên mẫu hàm cung cấp cho trình biên dịch biết những thông tin cơ bản của hàm và trình biên dịch ngầm hiểu rằng định nghĩa của hàm này có tồn tại. Cơ chế này giúp tăng tính tái sử dụng mã nguồn trong ngôn ngữ C/C++ bởi người dùng có thể tái sử dụng các nguyên mẫu hàm trong tệp header mà không cần biết nội dung chi tiết trong hàm đó.

Nguyên mẫu hàm là một tính năng hữu ích song nó cũng có nhược điểm. Nhược điểm của tính năng này đó là sự phụ thuộc vào người dùng. Một nguyên mẫu hàm không cần thiết phải có định nghĩa hàm bởi có thể nó không được sử dụng trong mã nguồn. Với ví dụ trong Đoạn mã 1.1, nguyên mẫu hàm `echo` không cần có định nghĩa bởi nó không được sử dụng trong bất kì thành phần nào khác. Trình biên dịch có khả năng tự xác định các nguyên mẫu hàm được sử dụng khi phân tích cú pháp mã nguồn ở bước biên dịch. Quá trình liên kết các tệp đối tượng có mục đích xác định định nghĩa hàm tương ứng với từng nguyên mẫu hàm. Ở bước này, nếu trình biên dịch không thể tìm thấy định nghĩa hàm tương ứng, nguyên mẫu hàm được coi là hàm thiếu định nghĩa.

1.3.2. Nguyên mẫu hàm ảo và lỗi thiếu bảng ký hiệu ảo

Hàm ảo là một cơ chế mới được giới thiệu trong ngôn ngữ C++ nhằm thể hiện các đặc trưng hướng đối tượng so với ngôn ngữ C. Hàm ảo là một hàm đặc biệt, chỉ có thể khai báo trong lớp và có khả năng ghi đè bởi hàm ở lớp dẫn xuất. Khi hàm ảo ở lớp cơ sở được gọi, chương trình sẽ tự động hiểu và chọn đúng đối tượng ở lớp dẫn xuất để gọi đúng hàm ghi đè. Khả năng trên được gọi là đa hình. Nguyên mẫu hàm ảo cũng có các đặc điểm tương tự như nguyên mẫu hàm cơ bản.

Để có được khả năng đa hình, ngôn ngữ C++ giới thiệu một khái niệm mới có tên là bảng ký hiệu ảo (Vtable). Vtable là bảng lưu trữ các khai báo biến, hàm có trong một kiểu dữ liệu tự định nghĩa do trình biên dịch tạo ra khi biên dịch một tệp mã nguồn. Vtable là công cụ giúp trình biên dịch biết một biến đã được định nghĩa hay chưa, biến đó thuộc kiểu gì, từ đó thông báo đến người dùng nếu có lỗi ngữ pháp. Vtable còn là công cụ giúp chương trình xác định được hàm nào sẽ được thực thi bằng cách sử dụng cơ chế liên kết động. Nếu hàm ảo, thuần ảo thiếu định nghĩa hàm thì khi liên kết, trình biên dịch sẽ báo lỗi do không biết chương trình sẽ cần hàm nào để chạy. Khác với các nguyên mẫu hàm cơ bản, trình biên dịch không hiển thị lỗi hàm thiếu định nghĩa nếu không tìm thấy định nghĩa của hàm ảo mà sẽ hiển thị lỗi thiếu bảng ký hiệu ảo. Nguyên nhân gây ra lỗi này là bởi trình biên dịch không cho phép gọi hàm ảo nếu nó không có ít nhất một hàm ghi đè.

```
1 class A {
2   virtual int echo() {}
3   virtual int foo(); // Trigger the undefined reference to vtable error
4   virtual int bar();
5 };
6 class B : public A {
7   int bar() {}
8 }
```

Đoạn mã 1.2: Ví dụ về lỗi thiếu bảng ký hiệu ảo.

Để làm rõ hơn ví dụ về lỗi thiếu bảng ký hiệu ảo, ta xét ví dụ trong Đoạn mã 1.2. Đoạn mã này chứa hai lớp `A` và `B` trong đó `B` là lớp dẫn xuất của `A`. Lớp `A` có chứa ba nguyên mẫu hàm ảo, trong đó hàm `bar` được ghi đè ở lớp dẫn xuất và hàm `echo` là khai

báo hàm ảo đầy đủ. Trong quá trình liên kết để tạo thành tệp thực thi, trình biên dịch báo lỗi thiếu Vtable cho kiểu `A` bởi trình biên dịch không thể tìm thấy hàm ghi đè nguyên mẫu hàm ảo `foo` ở lớp dẫn xuất.

1.3.3. Vai trò trong các dự án C/C++

Kể từ khi được giới thiệu trong ngôn ngữ C, nguyên mẫu hàm đóng vai trò quan trọng các dự án C/C++ và được sử dụng phổ biến trong quá trình phát triển phần mềm. Về mặt kỹ thuật, nguyên mẫu hàm giúp tạo ra một bản thiết kế cho các hàm trước khi chúng được triển khai. Điều này giúp kiểm soát việc sử dụng hàm, đồng thời cung cấp thông tin đầy đủ cho trình biên dịch để kiểm tra tính đúng đắn của mã nguồn. Về mặt phát triển, nguyên mẫu hàm cho phép các thành viên trong nhóm phát triển đặt ra những lớp giao diện, các hành vi sẽ được cung cấp bởi mô-đun họ phụ trách. Điều này thúc đẩy quá trình phát triển phần mềm nhanh hơn bởi các thành viên không cần phải đợi sự hoàn thiện từ các thành phần phụ thuộc.

Trong ngôn ngữ C++, vai trò của nguyên mẫu hàm được bổ sung trong đặc trưng kế thừa và đa hình với nguyên mẫu hàm ảo. Khi một lớp cơ sở chứa một hàm ảo, các lớp kế thừa có thể cung cấp triển khai riêng của hàm đó mà không cần thay đổi nguyên mẫu. Điều này tạo điều kiện cho việc chia sẻ chức năng giữa các lớp và giúp tăng tính linh hoạt của hệ thống. Nguyên mẫu hàm ảo tạo ra khả năng đa hình, cho phép gọi hàm dựa trên kiểu đối tượng thực tế thay vì kiểu của biến con trỏ hay tham chiếu. Điều này làm cho mã nguồn trở nên linh hoạt và dễ bảo trì, đặc biệt là trong các hệ thống lớn với nhiều lớp và đối tượng.

Do quá trình kiểm thử đơn vị diễn ra song song với quá trình phát triển nên ta không thể tránh khỏi tình trạng mã nguồn tồn tại một số nguyên mẫu hàm thiếu định nghĩa. Vì vậy, nhu cầu phát triển một giải pháp kiểm thử tự động đặc biệt cho mã nguồn C/C++ chứa hàm thiếu định nghĩa là vô cùng thiết yếu. Theo khảo sát của khóa luận về phương pháp xử lý hàm thiếu định nghĩa, hiện chưa có phương pháp nào được đề xuất vậy nên khóa luận sẽ thảo luận chi tiết về phương pháp giải quyết ở Chương 2.

1.4. Sinh giả lập mã nguồn tự động

Trong quá trình phát triển phần mềm, hai giai đoạn quan trọng là cài đặt và kiểm thử thường diễn ra đồng thời để giảm thiểu thời gian phát triển và đồng thời đảm bảo chất lượng sản phẩm. Tuy nhiên, điều này có thể dẫn đến việc mã nguồn kiểm thử chứa các thành phần chưa hoàn thiện. Điều này đặt ra thách thức khi đơn vị kiểm thử có thể chứa các lời gọi hàm chưa được đầy đủ, ảnh hưởng đến quá trình kiểm thử chính xác và hiệu quả.

Một số phương pháp sinh stub tự động đã được đề xuất để giải quyết bài toán trên. Trong đó, năm 2022, ông Trần Nguyên Hương và các cộng sự đã đề xuất phương pháp AS4UT [21] giúp sinh stub tự động, sử dụng trong kiểm thử đơn vị các dự án C/C++. Ý tưởng chính của phương pháp là xem mỗi lời gọi hàm như một biến giả. Giai đoạn tiền xử lý CFG được thêm vào hướng tiếp cận kiểm thử tượng trưng động. Trong giai đoạn này, mọi lời gọi hàm trong CFG của đơn vị kiểm thử được thay thế bằng các biến giả tương ứng. CFG sau khi biến đổi được sử dụng làm đầu vào để thực hiện quá trình thực thi tượng trưng giúp tìm ra các bộ dữ liệu kiểm thử có hướng. Phương pháp AS4UT đem lại một số kết quả tích cực khi áp dụng vào một số dự án mã nguồn mở ngôn ngữ C với 87.88% độ phủ C_1 trên dự án C-Algorithms[3]. Phương pháp này cũng cần ít hơn số ca kiểm thử để đạt được độ phủ cao hơn so với phương pháp kiểm thử tượng trưng động.

Phương pháp AS4UT khi áp dụng trên các dự án C++ bộc lộ một số hạn chế. Một trong số các hạn chế đó là việc xử lý lời gọi phương thức bởi phương pháp đang chú trọng vào kết quả trả về của lời gọi hàm mà chưa quan tâm tới các ảnh hưởng gây bởi lời gọi phương thức. Điều này có thể gây khó khăn khi kiểm thử các đơn vị có nhiều sự tương tác của các đội tượng bởi lời gọi phương thức có thể thay đổi gía trị thuộc tính của đối tượng.

[3]https://github.com/fragglet/c-algorithms

Chương 2

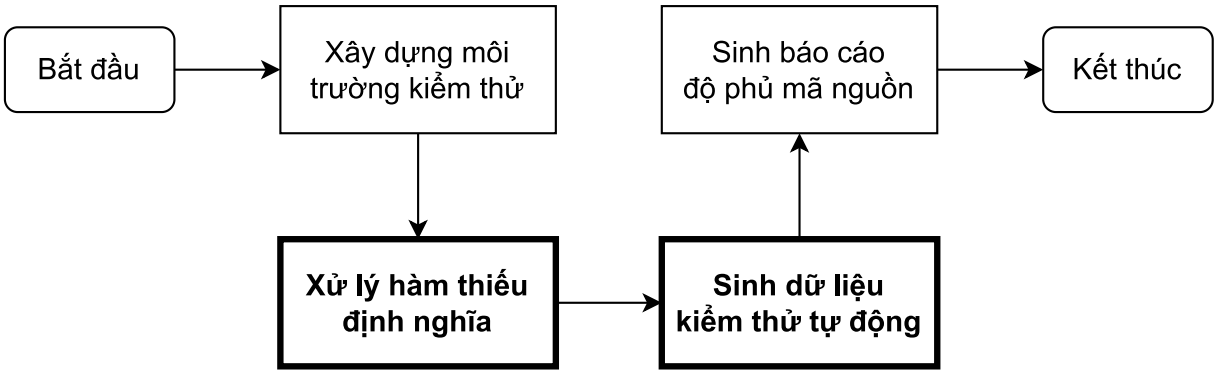
Phương pháp kiểm thử đơn vị tự động cho mã nguồn chứa hàm thiếu định nghĩa

2.1. Tổng quan phương pháp đề xuất

Về tổng quan, phương pháp đề xuất là sự kết hợp giữa phương pháp kiểm thử tượng trưng động [17] và phương pháp sinh giả lập mã nguồn tự động [21] với các cải tiến trong quá trình xử lý môi trường kiểm thử chứa hàm thiếu định nghĩa và quá trình thực thi tượng trưng lời gọi phương thức. Quá trình kiểm thử tự động bao gồm bốn pha được mô tả trong Hình 2.1. Các pha được in đậm trong hình vẽ thể hiện chúng được bổ sung hoặc cải tiến bởi phương pháp đề xuất.

Pha đầu tiên trong phương pháp đề xuất là pha xây dựng môi trường kiểm thử. Hai nhiệm vụ chính của pha đó là thực hiện các tác vụ tiền xử lý trên mã nguồn kiểm thử và phân tích mã nguồn để xây dựng đồ thị cấu trúc mã nguồn kiểm thử. Nội dung chi tiết của pha xây dựng môi trường kiểm thử được trình bày trong Mục 2.2.

Pha tiếp theo trong phương pháp đề xuất là pha xử lý hàm thiếu nghĩa. Pha này được bổ sung so với phương pháp kiểm thử tự động truyền thống nhằm xử lý các vấn đề phát sinh bởi các hàm thiếu định nghĩa khi kiểm thử tự động cho các mã nguồn chưa hoàn chỉnh. Chi tiết về quá trình xử lý hàm thiếu định nghĩa được mô tả trong Mục 2.3.



Hình 2.1: Tổng quan phương pháp đề xuất.

Sau pha xử lý hàm thiếu định nghĩa, quá trình kiểm thử tự động chuyển sang pha sinh dữ liệu kiểm thử tự động cho các đơn vị kiểm thử. Khóa luận sử dụng phương pháp sinh kế thừa từ phương pháp sinh dữ liệu kiểm thử tượng trưng động và phương pháp sinh giả lập mã nguồn AS4UT (mô tả ở Mục 1.4). Trong pha sinh dữ liệu kiểm thử tự động, phương pháp đề xuất bổ sung bước sinh giả lập mã nguồn cho các hàm là phương thức của đối tượng nhằm tăng khả năng thực thi các câu lệnh, điều kiện có liên quan đến các đối tượng được sử dụng trong đơn vị kiểm thử. Nội dung chi tiết của pha sinh dữ liệu kiểm thử được trình bày ở Mục 2.4.

Cuối cùng, các dữ liệu độ phủ được thu thập ở pha sinh dữ liệu kiểm thử được phân tích và tạo thành báo cáo kiểm thử trong pha sinh báo cáo độ phủ mã nguồn. Khóa luận sử dụng công cụ tính độ phủ GNU Coverage - GCOV[1] và công cụ sinh báo cáo độ phủ LCOV để thực hiện nhiệm vụ của pha. Trong đó, GCOV là công cụ phân tích độ phủ tích hợp sẵn trong tập hợp các trình biên dịch GNU (GNU Compiler Collection - GCC), LCOV là một tiện ích mở rộng từ công cụ GCOV với nhiệm vụ sinh báo cáo độ phủ dạng ngôn ngữ siêu văn bản (Hypertext Markup Language - HTML) dựa trên các thông tin độ phủ của GCOV. GCOV và LCOV là hai công cụ có sẵn trong GCC và được sử dụng phổ biến trong các dự án C/C++ [12]. Quá trình thực thi ca kiểm thử ở pha trước sinh ra thông tin về số lần chạy qua một dòng, số lần chạy qua các nhánh điều kiện với từng trường hợp đúng, sai theo định dạng của GCOV. Thông tin này sau đó được LCOV tổng hợp, tính toán và đưa ra tệp báo cáo HTML.

[1]https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html

2.2. Xây dựng môi trường kiểm thử

Như đã đề cập ở Mục 2.1, pha xây dựng môi trường kiểm thử thực hiện hai nhiệm vụ chính. Nhiệm vụ thứ nhất đó là thực hiện hai tác vụ tiền xử lý mã nguồn. Tác vụ đầu tiên là tác vụ thiết lập môi trường nhằm biên dịch mã nguồn và các ca kiểm thử về sau. Tác vụ này cài đặt các lệnh biên dịch, xác định và liên kết các thư viện mà mã nguồn sử dụng, thiết lập kiểu độ phủ mong muốn cho các đơn vị kiểm thử. Tác vụ thứ hai là tác vụ tạo môi trường tính toán độ phủ. Tác vụ này nhân bản mã nguồn kiểm thử và bổ sung các câu lệnh đánh dấu nhằm phục vụ quá trình xác định các câu lệnh, nhánh được viếng thăm và tính toán độ phủ trong pha sinh dữ liệu kiểm thử tự động. Nhiệm vụ còn lại của pha đó là phân tích mã nguồn kiểm thử và xây dựng đồ thị cấu trúc mã nguồn. Bước phân tích mã nguồn sử dụng công cụ phân tích mã nguồn CDT Parser[2] để trích xuất AST của mã nguồn kiểm thử. Từ thông tin trên AST, phương pháp đề xuất xây dựng đồ thị cấu trúc mã nguồn với các thông tin được phân tích đến mức hàm.

Đồ thị cấu trúc mã nguồn

Đồ thị cấu trúc mã nguồn là một cấu trúc dữ liệu biểu diễn mối quan hệ và sự tương tác giữa các thành phần trong mã nguồn kiểm thử. Đồ thị này được biểu diễn bởi một đồ thị có hướng $G = (V, E)$, trong đó V là tập các đỉnh tượng trưng cho các thành phần trong các đơn vị kiểm thử và E là tập các cạnh có hướng nối giữa hai đỉnh tượng trưng cho mối quan hệ phát sinh giữa đỉnh đầu và đỉnh cuối. Mỗi đỉnh trong đồ thị biểu thị cho các đơn vị cơ bản như như tệp mã nguồn, lớp, biến, thư viện được sử dụng, các hàm, v.v. Để làm rõ thể hiện của các hàm thiếu định nghĩa trong đồ thị, khóa luận biểu thị hàm thành hai loại đỉnh hàm: đỉnh nguyên mẫu hàm và đỉnh định nghĩa hàm. Các hàm được khai báo đầy đủ biểu thị bởi đỉnh định nghĩa hàm. Mỗi cạnh trong mã nguồn thuộc một trong các cạnh được mô tả dưới đây.

- Cạnh cha con: Thể hiện mối quan hệ cha - con (hay mối quan hệ chứa) giữa hai đơn vị cơ bản trong đơn vị kiểm thử. Ví dụ cho mối quan hệ này có thể kể đến như lớp chứa một số thuộc tính và phương thức, tương đương quan hệ lớp là cha của các thuộc tính và phương thức.

[2]https://github.com/eclipse-cdt/cdt

- Cạnh Include: Thể hiện mối quan hệ Include giữa hai tệp mã nguồn. Ví dụ cho quan hệ này là dòng lệnh `#include` thường thấy trong các mã nguồn C/C++.
- Cạnh kế thừa: Thể hiện mối quan hệ kế thừa giữa lớp cha và lớp dẫn xuất. Hai đỉnh của cạnh là hai đỉnh lớp.
- Cạnh lời gọi hàm: Thể hiện tương tác giữa hai hàm trong mã nguồn kiểm thử. Hai đỉnh của cạnh là hai đỉnh hàm. Cạnh lời gọi hàm là thành phần quan trọng trong bước xác định các hàm cần tạo stub khi kiểm thử tự động cho một hầm bất kì.
- Cạnh định nghĩa: Thể hiện mối quan hệ định nghĩa giữa đỉnh nguyên mẫu hàm ảo và đỉnh định nghĩa hàm ảo. Quan hệ định nghĩa thể hiện nguyên mẫu hàm được trỏ tới tồn tại một khai báo định nghĩa.

Đồ thị cấu trúc mã nguồn được xây dựng bằng việc mở rộng từng đỉnh tệp với các cạnh cha con. Quá trình mở rộng đỉnh tệp cấu thành một cây cấu trúc tệp. Tiếp theo, dựa trên các cây cấu trúc tệp, đồ thị cấu trúc mã nguồn được hoàn thiện bằng cách phân tích AST kết hợp tìm kiếm để bổ sung các cạnh quan hệ còn lại. Cạnh định nghĩa là trường hợp đặc biệt được bổ sung sau vào đồ thị trong pha xử lý hàm thiếu định nghĩa.

Ví dụ về đồ thị cấu trúc mã nguồn

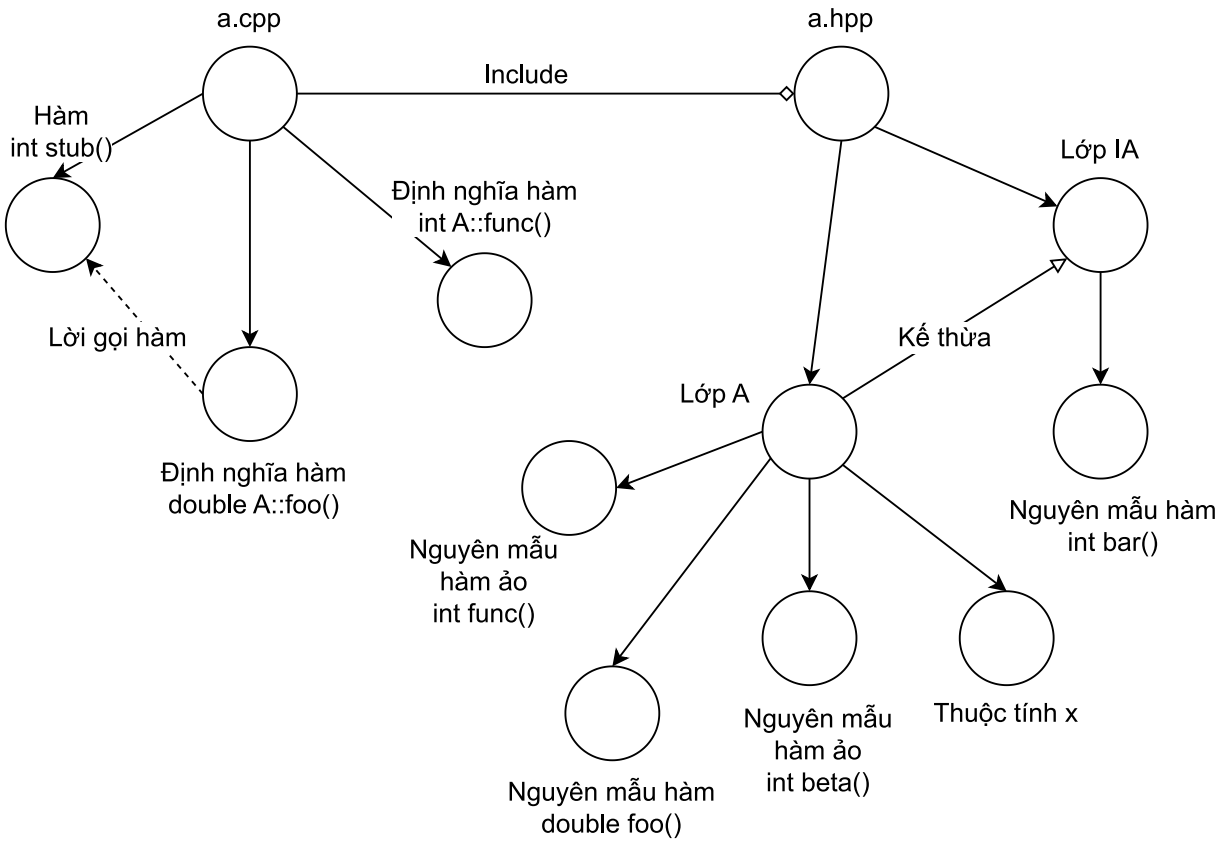
Hình 2.2 minh hoạ đồ thị cấu trúc mã nguồn của hai tệp mã nguồn a.hpp và a.cpp trong các Đoạn mã 2.1, 2.2. Trong đó, các cạnh nét liền mũi tên đen thể hiện cạnh cha con giữa hai đỉnh và các kiểu cạnh còn lại được ký hiệu trên hình vẽ với thể hiện tương ứng. Cây cấu trúc mã nguồn của tệp *a.cpp* gồm ba đỉnh hàm. Cây cấu trúc mã nguồn của tệp *a.hpp* gồm hai đỉnh lớp, mỗi đỉnh lớp có quan hệ cha con với một số đỉnh như đỉnh thuộc tính, đỉnh nguyễn mẫu hàm. Cạnh Include từ đỉnh tệp *a.cpp* tới đỉnh *a.hpp* tương đương dòng lệnh `#include "a.hpp"` trong Đoạn mã 2.1. Cạnh lời gọi hàm từ đỉnh định nghĩa hàm `double A::foo()` tới đỉnh định nghĩa hàm `int stub()` tương đương lời gọi hàm ở dòng 8 trong tệp *a.cpp*. Đoạn mã ví dụ cho thấy nguyên mẫu hàm ảo `int func()` trong lớp A có tồn tại định nghĩa bởi hàm `int A::func()` nhưng đồ thị thiếu cạnh định nghĩa về quan hệ này do cạnh định nghĩa giữa hai đỉnh sẽ được bổ sung sau pha xử lý hàm thiếu định nghĩa.

```cpp
// a.cpp
#include "a.hpp"
int A::func() {
  /* Function logic */
}
double foo() {
  /* Some logic */
  int ret = stub();
  /* Some logic */
}
int stub() {
  /* Function logic */
}
```

Đoạn mã 2.1: Mã nguồn tệp *a.cpp* minh họa đồ thị cấu trúc mã nguồn.

```cpp
// a.hpp
class IA {
  public:
  int bar();
};
class A : public IA {
  public:
  A() {}
  int x;
  virtual int beta();
  virtual int func();
  double foo();
};
```

Đoạn mã 2.2: Mã nguồn tệp *a.hpp* minh họa đồ thị cấu trúc mã nguồn.



Hình 2.2: Ví dụ minh hoạ về đồ thị cấu trúc mã nguồn xây dựng trên mã nguồn tệp *a.hpp* và *a.cpp*.

2.3. Xử lý hàm thiếu định nghĩa

Pha xử lý hàm thiếu định nghĩa được bổ sung với hai nhiệm vụ chính đó là xử lý nguyên mẫu hàm cơ bản thiếu định nghĩa tìm được bởi trình biên dịch và xử lý nguyên mẫu hàm ảo thiếu định nghĩa mà trình biên dịch không thể phát hiện được. Ý tưởng chung để giải quyết hai nhiệm vụ vừa đề cập dựa trên việc tìm kiếm các ứng viên nguyên mẫu hàm tương thích trong mã nguồn kiểm thử theo các tiêu chí thích hợp, sau đó sinh thân hàm giả cho các ứng viên này. Mục 2.3.1 mô tả cụ thể về phương pháp xử lý nhiệm vụ thứ nhất. Phương pháp xử lý nguyên mẫu hàm ảo được trình bày trong Mục 2.3.2.

2.3.1. Xử lý nguyên mẫu hàm cơ bản thiếu định nghĩa tìm bởi trình biên dịch

Khóa luận đề xuất phương pháp sử dụng trình biên dịch để tìm kiếm các nguyên mẫu hàm cần sinh thân hàm giả trong mã nguồn kiểm thử. Thuật toán 2.1 mô tả chi tiết quá trình tìm kiếm và xử lý các nguyên mẫu hàm cơ bản thiếu định nghĩa. Trước hết, thuật toán bắt đầu bằng việc thu thập danh sách các nguyên mẫu hàm cơ bản cần quan tâm trong tập các tệp mã nguồn S và khởi tạo tập T rỗng. Danh sách trả về bao gồm các xâu ký tự, mỗi xâu biểu thị một hàm bao gồm tên hàm và danh sách tham số của hàm đó (dòng 1-2). Tập T là tập hợp các nguyên mẫu hàm trong mã nguồn kiểm thử được sinh thân hàm giả. Sau khi có danh sách, thuật toán xử lý từng xâu trong danh sách đầu vào (dòng 3-14). Cụ thể, với mỗi xâu, một AST $undef\_ast$ được xây dựng từ chữ ký hàm mà xâu biểu thị (dòng 4). Tiếp đó, tên của nguyên mẫu hàm được trích xuất từ AST vừa xây dựng (dòng 5) và tập các nguyên mẫu hàm ứng viên có cùng tên trong mã nguồn kiểm thử được thu thập và lưu vào danh sách $candidates$ (dòng 6). Danh sách tham số của nguyên mẫu hàm $undef$ và phạm vi định danh (scope qualifier) được trích xuất từ AST phục vụ cho quá trình lọc (dòng 7-8). Danh sách ứng viên được lọc bởi Thuật toán 2.2 và thu được các ứng viên hợp lệ $qualified\_cans$ (dòng 9). Cuối cùng, từng ứng viên hợp lệ qc được sinh thân hàm giả và thêm vào tập T (dòng 11-12). Dữ liệu trong tập T được lưu lại để xây dựng môi trường kiểm thử cho mã nguồn đang xét trong các lần sau mà không cần xử lý lại từ đầu.

Thuật toán 2.1: Thuật toán xử lý hàm thiếu định nghĩa

Input: S - collection of source code files

Output: T - collection of handled undefined functions

```
1  undefined_functions ← Get necessary undefined functions in S;
2  T ← ∅;
3  for undef : undefined_functions do
4      undef_ast ← Construct AST from the string undef;
5      undef_name ← Get the name of the function from string undef_ast;
6      candidates ← Find all function prototypes having the same name with undef_name
         in source code;
7      undef_params ← Extract a list of parameters from undef_ast;
8      undef_scope ← Extract function scope qualifier from undef_ast;
9      qualified_cans ← call Algorithm 2.2 (undef_params, undef_scope, candidates);
10     for qc : qualified_cans do
11         t ← Generate simple function body for qc;
12         T ← T ∪ t
13     end
14 end
15 return T
```

Thu thập danh sách nguyên mẫu hàm cơ bản thiếu định nghĩa

Ý tưởng sử dụng các công cụ tiện ích của trình biên dịch để tìm nguyên mẫu hàm cần quan tâm xuất phát từ lợi thế trình biên dịch biết hàm nào sẽ được sử dụng trong các tệp đối tượng như đã trình bày ở Mục 1.3. Đoạn mã 2.3 mô tả cách dùng các công cụ tiện ích có trong trình biên dịch để tìm các nguyên mẫu hàm thiếu định nghĩa. Đầu tiên, phương pháp đề xuất sử dụng công cụ *g++* với cờ *-c* để biên dịch các tệp mã nguồn thành tệp đối tượng (object file) có đuôi *.o* (dòng 1). Sau đó, các tệp đối tượng này sẽ được liên kết bởi công cụ liên kết *ld* và cờ *-r* thành một tệp đối tượng tổng hợp *temp.o* (dòng 2). Lí do phương pháp đề xuất sử dụng công cụ liên kết *ld* thay vì *g++* là bởi *g++* yêu cầu người dùng phải hoàn thiện tất cả các nguyên mẫu hàm bị thiếu định nghĩa trước khi liên kết thành tệp tổng hợp. Cuối cùng, tệp đối tượng tổng hợp được phân tích bởi công cụ phân tích ký hiệu *nm*[3] với các cờ *-u* - trích xuất các hàm thiếu định nghĩa, *-C* -

[3]https://sourceware.org/binutils/docs/binutils/nm.html

```cpp
// a.cpp
#include "a.hpp"
#include "b.hpp"

int foo(A *a, int x) {
    int ret = a->bar(true);
    int max = maxT<int>(x, ret);
    max += a->echo();
    const int *p = &(a->x);
    ret -= func(p, max);
    return ret;
}

void gamma(A *a, int x) {
    double max = maxT<double>(x,
        a->x);
    a->x = max;
}
int unused(int x);
```

Đoạn mã 2.4: Mã nguồn tệp a.cpp.

```cpp
// a.hpp
#include "c.hpp"
class A {
public:
    int x;
    A(int _x) { x = _x; }
    int bar(bool b);
    double echo() {
        return x + delta(&x);
    }
};
int bar(bool b);
// b.hpp
template <typename T> T max(T a);
int func(int *a, int b);
int func(const int* a, int b);

// c.hpp
double delta(int* const x);
```

Đoạn mã 2.5: Mã nguồn tệp a.hpp, b.hpp và c.hpp.

chuyển ký hiệu máy thành ngôn ngữ lập trình C++ (dòng 3). Đầu ra của công cụ *nm* sẽ được ghi vào tệp *list.txt*.

```
g++ -c *.cpp
ld -r *.o -o temp.o
nm temp.o -u -C > list.txt
```

Đoạn mã 2.3: Các câu lệnh tìm nguyên mẫu hàm thiếu định nghĩa cần quan tâm sử dụng trình biên dịch.

Các Đoạn mã 2.4, 2.5 minh họa cách áp dụng phương pháp đề xuất trên tập đơn vị kiểm thử cần xử lý các nguyên mẫu hàm cơ bản thiếu định nghĩa với bốn tệp mã nguồn. Kiểm thử viên muốn kiểm thử tự động cho hai hàm có logic đầy đủ `foo` và `gamma`.

Hai hàm này chứa mã nguồn đầy đủ. Hai đoạn mã chứa một số nguyên mẫu hàm đó là `unused`, `bar`, `delta` và ba hàm trong tệp `b.hpp`. Hàm `foo` trong Đoạn mã 2.4 nhận đầu vào là một biến con trỏ đến đối tượng của lớp `A` và một biến kiểu nguyên `x`. Logic mã nguồn của `foo` gọi tới hai phương thức của biến `a` đó là `bar` và `echo`. Ngoài ra, hàm `foo` còn gọi tới nguyên mẫu hàm template `maxT` với kiểu khởi tạo `int` và hàm `func(const int* a, int b)` thuộc tệp `b.hpp`. Hàm `gamma` cũng có các đầu vào tương tự nhưng gọi nguyên mẫu hàm `maxT` nhưng sử dụng tham số kiểu `double` thay vì `int` như hàm `foo`.

```
1 func(int const*, int)
2 double maxT<double>(double, double)
3 int maxT<int>(int, int)
4 delta(int*)
5 A::bar(bool)
```

Đoạn mã 2.6: Danh sách các nguyên mẫu hàm thiếu định nghĩa tìm bởi trình biên dịch.

Đoạn mã 2.6 minh họa danh sách các nguyên mẫu hàm cần quan tâm trong tệp các đơn vị kiểm thử *a.cpp*, *a.hpp*, *b.hpp* và *c.hpp* khi áp dụng phương pháp đề xuất. Dòng đầu tiên cho biết nguyên mẫu hàm `func(int const*, int)` cần được xử lý, tương đương với hàm `func(const int* a, int b)` trong mã nguồn *b.hpp* (dòng 16 trong Đoạn mã 2.5). Dòng 2 và 3 biểu thị cú pháp của hàm template. Đối với trình biên dịch, các lời gọi hàm template với các kiểu đối số khác nhau sẽ tạo ra các chữ ký hàm khác nhau. Do hàm `foo` và `gamma` gọi hàm `maxT` với hai đối số kiểu khác nhau nên danh sách các nguyên mẫu hàm thiếu định nghĩa chứa chữ ký hàm cho cả hai kiểu này. Tiếp theo, nguyên mẫu hàm ở dòng 4 tương ứng hàm `delta(int* const x)` trong tệp c.hpp. Tham số hiển thị ở tệp mã nguồn và trên danh sách khác nhau là bởi trình biên dịch chỉ quan tâm tới kiểu của tham số và loại bỏ lớp lưu trữ (storage class) ngoài cùng của tham số đó. Tham số `x` của hàm `delta` có kiểu là `int* const` tức con trỏ hằng trỏ tới một biến kiểu nguyên. Trình biên dịch chuyển kiểu của tham số trên thành `int*`. Trong trường hợp của hai hàm `func`, tham số đầu của hai hàm khác nhau là bởi tham số `const int* a` được trình biên dịch chuyển thành kiểu `const int*` do từ khoá `const` là lớp lưu trữ của biến được tham số trỏ tới. Cuối cùng nguyên mẫu hàm `A::bar(bool)` tương ứng với nguyên mẫu hàm `int bar(bool b)` trong lớp `A` (dòng 7 trong Đoạn mã 2.5).

Lọc tìm ứng viên hợp lệ

Như đã mô tả trong ví dụ minh hoạ ở Đoạn mã 2.6, mỗi nguyên mẫu hàm trong danh sách trả về bởi trình biên dịch sẽ tương ứng với một số nguyên mẫu hàm cơ bản trong mã nguồn gốc. Để ánh xạ từng nguyên mẫu hàm cần quan tâm tới đúng nguyên mẫu hàm cơ bản trong đơn vị kiểm thử, phương pháp đề xuất sử dụng ba tiêu chí để đánh giá độ phù hợp của từng ứng viên. Tiêu chí đầu tiên đó là hàm ứng viên chưa được ánh xạ bởi bất kỳ nguyên mẫu hàm nào trên danh sách. Tiêu chí thứ hai đó là số lượng tham số và phạm vi định danh của hai hàm phải giống nhau. Tiêu chí này giúp giảm số lượng các ứng viên có cùng tên cần xét. Cuối cùng, ứng viên hợp lệ nếu từng tham số đã chuẩn hoá của ứng viên có cùng kiểu với tham số tương ứng của nguyên mẫu hàm.

Thuật toán 2.2 mô tả chi tiết cách áp dụng ba tiêu chí để lọc ra các ứng viên hợp lệ. Đầu vào của thuật toán gồm danh sách tham số trích xuất từ nguyên mẫu hàm cần tìm và danh sách nguyên mẫu hàm ứng viên trong mã nguồn kiểm thử. Mỗi ứng viên sẽ được xét duyệt lần lượt trên ba tiêu chí (dòng 3-28). Nếu ứng viên *candidate* đã được ánh xạ (hay có trạng thái *handled*), ứng viên sẽ được bỏ qua (dòng 3-4). Ngược lại, danh sách tham số của ứng viên *candidate* được trích xuất để sử dụng cho hai tiêu chí còn lại (dòng 6). Nếu số lượng tham số hoặc phạm vi định danh của ứng viên khác nguyên mẫu hàm đầu vào thì ứng viên này sẽ được bỏ qua (dòng 8-9). Để áp dụng tiêu chí thứ ba một cách hiệu quả, phương pháp đề xuất xét duyệt lần lượt từng tham số và dừng lại khi gặp tham số đầu tiên không hợp lệ. Trước hết, thuật toán khởi tạo biến *same_params* dùng để đánh dấu tất cả tham số đều hợp lệ và gán biến *iter* bằng vị trí đầu tiên trong danh sách tham số (dòng 11-12). Tiếp đó, quá trình lặp đánh giá từng tham số diễn ra cho đến hết danh sách tham số hoặc gặp tham số không hợp lệ (dòng 13-22). Tham số thứ *iter* của ứng viên và hàm đầu vào được lấy ra (dòng 13-14). Do tham số của hàm đầu vào đã được chuẩn hoá theo tiêu chuẩn của C++[4,5] nên tham số của ứng viên cũng sẽ được đưa về theo các tiêu chuẩn này. Nếu hai tham số đang xét có cùng kiểu thì thuật toán sẽ chuyển sang tham số tiếp theo (dòng 16-17). Ngược lại, quá trình lặp sẽ kết thúc và thuật toán đánh dấu tính hợp lệ là *False* (dòng 19-20). Sau quá trình đánh giá, nếu tất cả tham số của ứng viên đều hợp lệ, thuật toán chuyển trạng thái của ứng viên thành đã được ánh xạ và thêm vào danh sách ứng viên hợp lệ.

[4]https://www.open-std.org/jtc1/sc22/wg21/docs/standards

[5]https://en.cppreference.com/w/cpp/language/function

Thuật toán 2.2: Thuật toán lọc tìm ứng viên hợp lệ

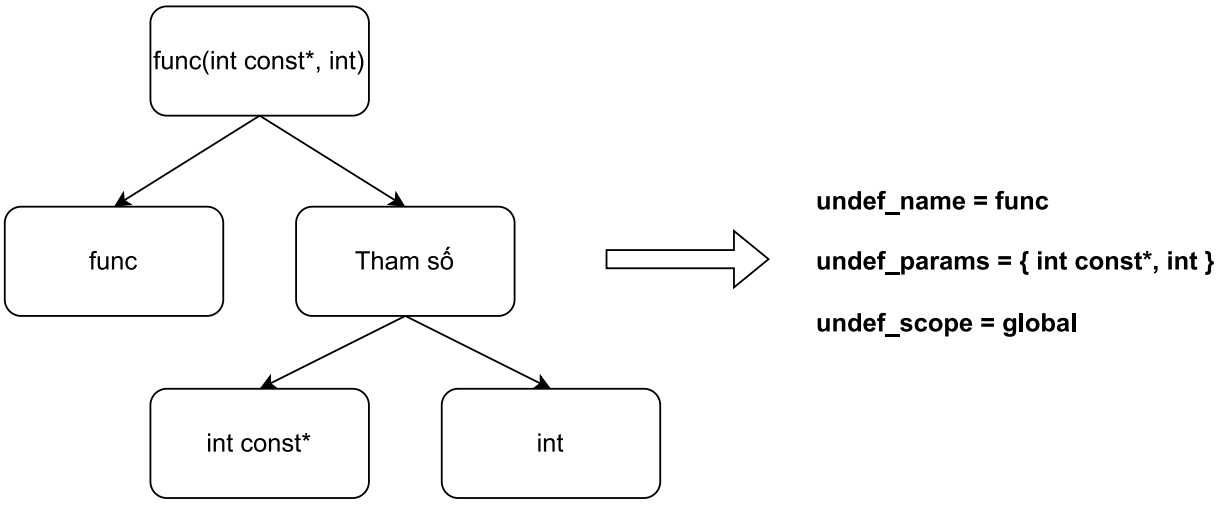
Input: *undef_params* - list of parameters extracted from the string *undef*
undef_scope - the scope qualifier of *undef*
candidates - list of function prototypes having the same name

Output: *qualified_cans* - list of qualified function prototypes

```
1  qualified_cans ← Initial empty list;
2  for candidate : candidates do
3      if candidate is handled then
4          continue
5      else
6          can_params ← Extract a list of parameters of function candidate;
7          can_scope ← Get the scope qualifier of candidate;
8          if can_params.length ≠ undef_params.length or can_scope ≠ undef_scope
            then
9              continue
10         else
11             same_params ← True;
12             iter ← 1 ;                /* Assume the list starts at index 1 */
13             while iter ≤ can_params.length do
14                 can_param_iter ← Normalize the iter^th param of can_params;
15                 undef_param ← Get the iter^th param of under_params ;
16                 if can_param_iter has the same type with under_params then
17                     iter++
18                 else
19                     same_params ← False;
20                     break
21                 end
22             end
23             if same_params ← True then
24                 Set candidate state to handled;
25                 Push candidate into qualified_cans;
26             end
27         end
28     end
29 end
30 return qualified_cans
```

Ví dụ về xử lý nguyên mẫu hàm thiếu định nghĩa

Để làm rõ cách hoạt động của Thuật toán 2.2, ta xét ví dụ áp dụng Thuật toán 2.1 trên danh sách các hàm trả về ở Đoạn mã 2.6. Xét ví dụ áp dụng phương pháp đề xuất trên xâu ký tự *func(int const*, int)* (dòng 1), trước hết AST của nguyên mẫu hàm được xây dựng như Hình 2.3. AST gồm hai thành phần chính: tên của hàm (nút trái) và danh sách tham số (nút phải). Dựa vào thông tin trên cây AST, thuật toán trích xuất được tên `func` và danh sách tham số `int const*, int` và phạm vi định danh là không gian tên toàn cục. Tiếp theo thuật toán thu thập danh sách các nguyên mẫu hàm ứng viên có cùng tên `func` trong mã nguồn kiểm thử. Từ Đoạn mã 2.5, thuật toán thu được hai hàm `func(int *a, int b)` và `func(const int* a, int b)` (dòng 15-16). Sau đó, phương pháp đề xuất áp dụng Thuật toán 2.2 với hai ứng viên cùng các thông tin trích xuất được. Bảng 2.1 minh hoạ cho quá trình lọc hai ứng viên. Các ứng viên được xét lần lượt theo thứ tự xuất hiện từ trên xuống của bảng. Với mỗi ứng viên, thuật toán lọc trên từng tiêu chí được liệt kê theo thứ tự từ trên xuống và thuật toán dừng khi tất cả tiêu chí đều hợp lệ hoặc gặp tiêu chí đầu tiên không hợp lệ. Bảng minh hoạ cho thấy ứng viên `func(int* a, int b)` không hợp lệ do kiểu của tham số đầu tiên sau khi chuẩn hoá là `int*`, không phải `int const*`. Ứng viên `func(const int* a, int b)` đều hợp lệ trên tất cả các tiêu chí nên nguyên mẫu hàm này sẽ được sinh thân hàm giả.

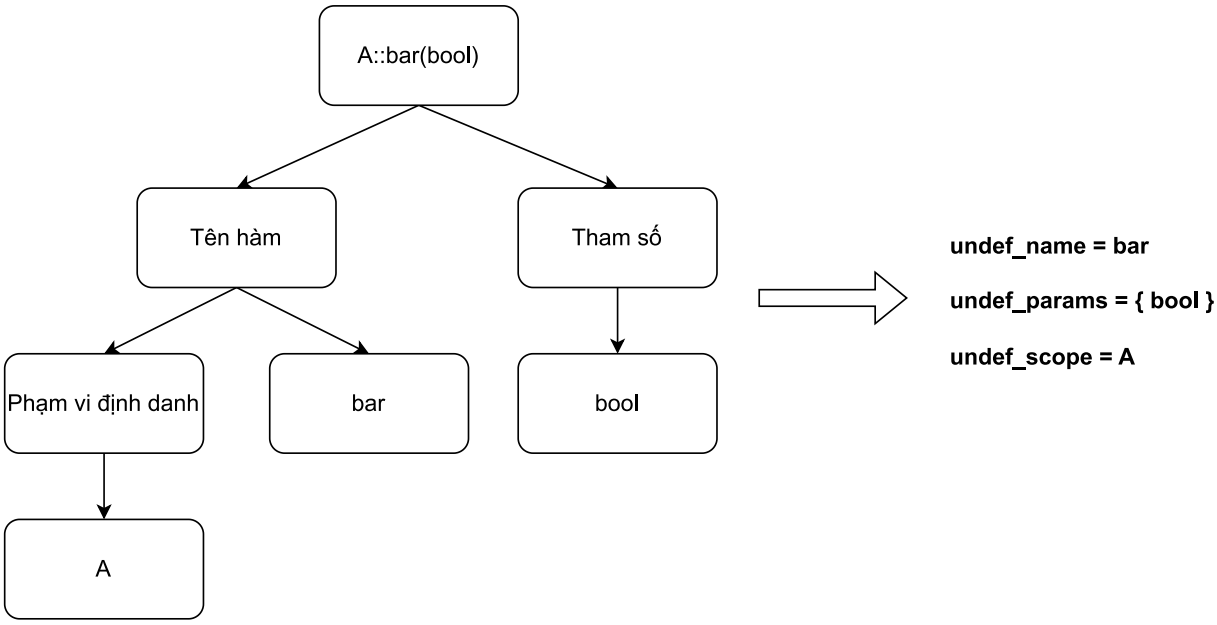


Hình 2.3: AST (bên trái) và các thông tin trích xuất được (bên phải) của nguyên mẫu hàm `func(int const*, int)` trong Đoạn mã 2.6.

Bảng 2.1: Bảng minh hoạ tính hợp lệ của hai ứng viên cùng tên `func` trên từng tiêu chí

Ứng viên	**Tiêu chí**		**Hợp lệ**
`func(int* a, int b)`	Chưa có ánh xạ		Có
	Số lượng tham số Phạm vi định danh		Có
	Danh sách kiểu tham số	int*	Không
`func(const int* a, int b)`	Chưa có ánh xạ		Có
	Số lượng tham số Phạm vi định danh		Có
	Danh sách kiểu tham số	int const*	Có
		int	Có

Khi xét áp dụng thuật toán với xâu `func(int const*, int)`, phương pháp đề xuất chưa bộc lộ khả năng giảm số lượng ứng viên cần xét của tiêu chí thứ hai. Nhằm làm rõ khả năng này, ta xét ví dụ áp dụng thuật toán với xâu `A::bar(bool)`. Tương tự như với xâu `func(int const*, int)`, trước hết, AST và các thông tin của nguyên mẫu hàm được trích xuất như Hình 2.4. Phạm vi định danh của nguyên mẫu hàm đang xét không phải không gian tên toàn cục mà thuộc về không gian tên của lớp `A`. Các ứng viên có cùng tên `bar` trong mã nguồn kiểm thử bao gồm hàm `A::bar(bool b)` và `bar(bool b)`. Dựa các thông tin trích xuất được, quá trình lọc ứng viên áp dụng Thuật toán 2.2 và được minh hoạ qua Bảng 2.2. Như vậy, khi xét ứng viên `bar(bool b)`, Thuật toán 2.2 không cần xét tính hợp lệ của tiêu chí thứ ba mà vẫn loại bỏ được ứng viên. Điều này có thể lý giải bởi phạm vi định danh của ứng viên này là không gian tên toàn cục, không phải trong không tên của lớp `A`. Áp dụng thuật toán đề xuất trên các xâu còn lại trong danh sách, phương pháp đề xuất thu được mã nguồn kiểm thử đã bổ sung các thân hàm giả cần thiết như trong Đoạn mã 2.7. Từ danh sách các nguyên mẫu hàm cần quan tâm trả về bởi trình biên dịch, phương pháp đề xuất đã thêm thân hàm giả `{/* Insert stub code later */}` vào các nguyên mẫu hàm tương ứng trong mã nguồn kiểm thử. Các nguyên mẫu hàm này sau đó sẽ được thiết lập và sinh stub tự động trong quá trình sinh dữ liệu kiểm thử tự động.



Hình 2.4: AST (bên trái) và các thông tin trích xuất được (bên phải) của nguyên mẫu hàm `A::bar(bool)` trong Đoạn mã 2.6.

Bảng 2.2: Bảng minh hoạ tính hợp lệ của hai ứng viên cùng tên `bar` trên từng tiêu chí

Ứng viên	**Tiêu chí**	**Hợp lệ**
`int A::bar(bool b)`	Chưa có ánh xạ	Có
	Số lượng tham số Phạm vi định danh	Có
	Danh sách kiểu tham số: bool	Có
int bar(bool b)	Chưa có ánh xạ	Có
	Số lượng tham số Phạm vi định danh	Không

2.3.2. Xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Như đã đề cập ở Mục 1.3, các hàm thiếu định nghĩa không chỉ gây lỗi thiếu tham chiếu khi liên kết các tệp đối tượng mà còn gây ra lỗi thiếu bảng hàm ảo. Điều này cũng đồng nghĩa với việc các nguyên mẫu hàm ảo thiếu định nghĩa, nguyên nhân gây ra lỗi, không thể tìm thấy bởi phương pháp đề xuất ở Mục 2.3.1 do quá trình liên kết các tệp đối tượng chỉ cho biết lớp đối tượng nào đang thiếu bảng hàm ảo mà không chỉ rõ rằng

```
// a.hpp
#include "c.hpp"
class A {
public:
    int x;
    A(int _x) { x = _x; }
    int bar(bool b) {/* Insert stub code later */ }
    double echo() {
        return x + delta(&x);
    }
};
int bar(bool b);
// b.hpp
template <typename T> T max(T a) {/* Insert stub code later */ }
int func(int *a, int b);
int func(const int* a, int b) {/* Insert stub code later */ }
// c.hpp
double delta(int* const x) {/* Insert stub code later */ }
```

Đoạn mã 2.7: Mã nguồn tệp a.hpp, b.hpp và c.hpp sau khi áp dụng phương pháp xử lý nguyên mẫu hàm thiếu định nghĩa.

do nguyên mẫu hàm ảo nào gây ra. Phương pháp đề xuất giải quyết vấn đề này dựa trên việc xét duyệt các đỉnh nguyên mẫu hàm ảo trên từng cây cấu trúc tệp và tìm kiếm sự tồn tại của cạnh quan hệ định nghĩa trỏ tới đỉnh này trong đồ thị cấu trúc mã nguồn. Nếu đỉnh nguyên mẫu hàm ảo không tồn tại cạnh định nghĩa thì phương pháp đề xuất sẽ sinh thân hàm giả cho hàm này.

Thuật toán xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Phương pháp xử lý nguyên mẫu hàm ảo thiếu định nghĩa được mô tả chi tiết trong Thuật toán 2.3. Đầu vào của thuật toán là đồ thị cấu trúc mã nguồn được xây dựng ở pha xây dựng môi trường kiểm thử. Thuật toán xét duyệt từng cây cấu trúc tệp $file\_tree$ trong đồ thị đầu vào. Với mỗi cây cấu trúc $file\_tree$, trước hết, thuật toán thu thập tập hợp T các cây cấu trúc có đỉnh tệp có cạnh Include tới đỉnh tệp của $file\_tree$ (dòng 2-4).

Thuật toán 2.3: Thuật toán xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Input: *project_graph* - the project structure graph

Output: *project_graph* - the modified graph with new definition edges

```
1  for file_tree : project_graph do
2      cur_file_node ← Get file node of file_tree;
3      T ← file_tree;
4      T ← T ∪ Get all file struct tree of file nodes that include cur_file_node;
5      for undef_virtual : file_tree do
6          undef_name ← Get the name of the function from undef_virtual;
7          candidates ← Find all definition functions having the same name with
             undef_name in T;
8          undef_params ← Extract a list of parameters from undef_virtual;
9          undef_scope ← Extract function scope qualifier from undef_virtual;
10         qualified_cans ← call Algorithm 2.2
             (undef_params, undef_scope, candidates);
11         if qualified_cans = ∅ then
12             Generate a simple function body for undef_virtual;
13         else
14             qc ← Get the first qualified candidate;
15             Generate definition edge for qc and undef_virtual on project_graph;
16         end
17     end
18 end
19 return project_graph
```

Bản thân *file_tree* cũng được thêm vào tập hợp này để xét trường hợp định nghĩa của nguyên mẫu hàm ảo được khai báo ngay trong chính cây cấu trúc chứa nguyên mẫu. Tiếp theo, thuật toán xét duyệt từng đỉnh nguyên mẫu hàm ảo có trong cây cấu trúc tệp *file_tree*. Với mỗi đỉnh nguyên mẫu hàm ảo, thuật toán trích xuất các thông tin như tên (dòng 6), danh sách tham số (dòng 8) và phạm vi định danh (dòng 9). Danh sách các đỉnh định nghĩa hàm ứng viên có cùng tên được thu thập (dòng 7). Sau đó, thuật toán áp dụng phương pháp lọc tìm kiếm ứng viên phù hợp trong Thuật toán 2.2 để lấy danh sách ứng viên hợp lệ (dòng 10). Nếu danh sách ứng viên hợp lệ rỗng tức nguyên mẫu hàm ảo không có định nghĩa hàm tương ứng thì thuật toán sẽ sinh thân hàm giả cho hàm này

(dòng 11-12). Ngược lại, do đỉnh định nghĩa hàm luôn là duy nhất nên thuật toán sinh cạnh định nghĩa giữa ứng viên hợp lệ duy nhất và đỉnh nguyên mẫu hàm ảo đang xét (dòng 14-15). Thuật toán diễn ra tương tự với các đỉnh nguyên mẫu hàm ảo còn lại và tương tự với các cây cấu trúc tệp khác. Đầu ra của thuật toán là đồ thị cấu trúc mã nguồn với các cạnh bổ sung. Đồ thị đầu ra sẽ được lưu lại để phục vụ cho các lần kiểm thử sau trên mã nguồn đang xét.

Ví dụ về xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Các Đoạn mã 2.1, 2.2 trong ví dụ minh họa độ thị cấu trúc mã nguồn chứa hai nguyên mẫu hàm ảo `beta()` và `func()`. Trong đó hàm `beta()` là nguyên mẫu hàm ảo thiếu định nghĩa. Để minh họa phương pháp đề xuất, ta xét ví dụ áp dụng Thuật toán 2.3 để xử lý hai nguyên mẫu hàm ảo trên. Trước hết, thuật toán xét cây cấu trúc tệp có đỉnh tệp *a.hpp* (gọi tắt là cây cấu trúc *a.hpp*) và thu thập tập hợp T chứa các cây cấu trúc tệp có đỉnh tệp có cạnh Include tới đỉnh tệp *a.hpp*. Dựa trên đồ thị cấu trúc mã nguồn như Hình 2.2, tập T thu được gồm hai cây cấu trúc: cây gốc đang xét *a.hpp* và cây cấu trúc *a.cpp*. Tiếp theo, thuật toán xét từng nguyên mẫu hàm ảo có trong cây cấu trúc *a.hpp*. Đối với hàm `func()`, thuật toán trích xuất các thông tin của hàm và thu được như sau: tên hàm là `func`, hàm không có tham số và phạm vi định danh là lớp `A`. Dựa trên thông tin này, thuật toán tìm các ứng viên có cùng tên `func` và thu được đỉnh định nghĩa hàm `A::func()`. Do ứng viên này hợp lệ trên cả ba tiêu chí của Thuật toán 2.2 nên ứng viên này là định nghĩa hàm của nguyên mẫu hàm `func()`. Do có tồn tại một ứng viên hợp lệ nên thuật toán sinh cạnh định nghĩa giữa đỉnh ứng viên và đỉnh nguyên mẫu hàm `func()`. Đối với hàm `beta()`, các bước tương tự được áp dụng. Tuy nhiên, do không tồn tại ứng viên có cùng tên `beta` nên nguyên mẫu hàm ảo này được xác định là thiếu định nghĩa. Do vậy, thuật toán sinh thân hàm giả cho hàm này theo quy tắc của phương pháp xử lý nguyên mẫu hàm thiếu định nghĩa ở Mục 2.3.1. Thuật toán tiếp tục xử lý tương tự với cây cấu trúc *a.cpp*.

2.4. Sinh dữ liệu kiểm thử tự động

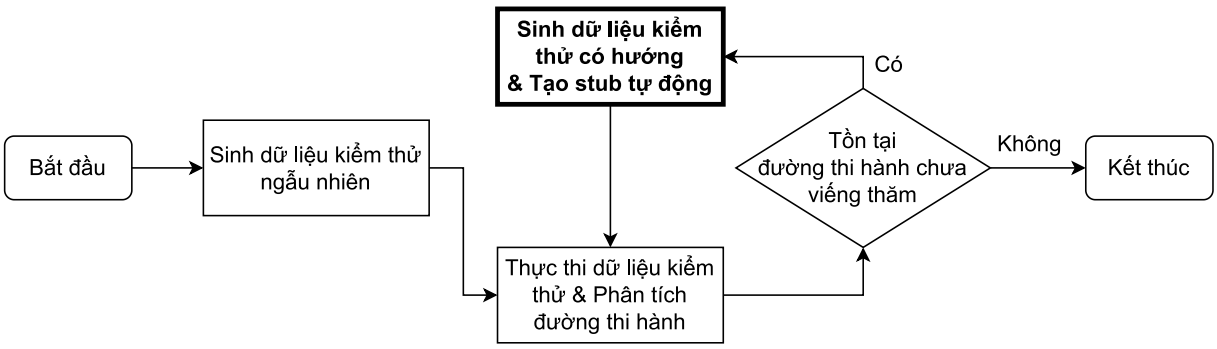
Trong các dự án thực tế, mỗi hàm cần kiểm thử có thể chứa rất nhiều lời gọi tới các hàm khác trong mã nguồn kiểm thử. Do các dự án hiện nay thường phát triển song

song với kiểm thử đảm bảo chất lượng nên mã nguồn có thể chứa nhiều hàm chưa được hoàn thiện logic hoặc mới chỉ có nguyên mẫu hàm. Khóa luận đã đề xuất phương pháp để giải quyết các nguyên mẫu hàm thiếu định nghĩa song vẫn cần giải quyết bài toán sinh dữ liệu kiểm thử tự động một cách hiệu quả cho các mã nguồn chưa hoàn thiện. Do vậy, khóa luận đề xuất phương pháp sinh dữ liệu kiểm thử tự động áp dụng phương pháp kiểm thử tượng trưng động [13, 15] và phương pháp sinh giả lập mã nguồn tự động AS4UT [21] để xử lý bài toán vừa đề cập. Với các dự án hướng đối tượng C++, các biến được sử dụng trong hàm có thể xoay quanh các đối tượng. Các đối tượng có thể chứa các phương thức chưa được hoàn thiện, chỉ có nguyên mẫu hàm để thông báo cho đội phát triển biết đối tượng này có thể làm gì. Như vậy, luồng logic trong hàm có thể ảnh hưởng lớn bởi các phương thức của đối tượng. Do phương pháp sinh stub tự động chưa xử lý được các trường hợp liên quan đến phương thức của đối tượng nên khóa luận bổ sung cải tiến để xử lý vấn đề trên.

2.4.1. Tổng quan pha sinh dữ liệu kiểm thử tự động

Về tổng quan, quá trình sinh dữ liệu kiểm thử tự động bao gồm ba bước như trong Hình 2.5. Các bước trong phương pháp đề xuất được kế thừa từ phương pháp sinh dữ liệu kiểm thử tự động cho con trỏ void và con trỏ hàm VFP [13]. Trong đó, bước sinh dữ liệu kiểm thử có hướng được bổ sung phương pháp sinh stub tự động. Bước đầu tiên trong quá trình là bước sinh dữ liệu kiểm thử ngẫu nhiên. Nhiệm vụ của bước này là phân tích kiểu dữ liệu của các tham số được sử dụng trong hàm đang kiểm thử. Các tham số này là tham số đầu vào của hàm, biến toàn cục và một đối tượng giả để gọi hàm nếu hàm đang kiểm thử là một phương thức trong lớp. Bước sinh dữ liệu ngẫu nhiên sẽ sinh giá trị ngẫu nhiên trong khoảng giá trị cho phép của kiểu dữ liệu cho từng tham số. Trong bước này, giá trị trả về của các lời gọi hàm cũng được sinh ngẫu nhiên. Tập hợp giá trị của tập tham số sẽ tạo thành các bộ dữ liệu kiểm thử. Bước tiếp theo là bước thực thi dữ liệu kiểm thử và phân tích đường thi hành. Bước này có hai nhiệm vụ chính. Nhiệm vụ thứ nhất đó là thực thi các bộ dữ liệu kiểm thử tạo ra bởi bước sinh dữ liệu kiểm thử ngẫu nhiên hoặc bước sinh dữ liệu kiểm thử có hướng. Đầu ra của việc thực thi dữ liệu kiểm thử là tập đường thi hành được viếng thăm bởi các bộ dữ liệu kiểm thử. Nhiệm vụ thứ hai của bước này đó là phân tích đường thi hành nhằm tìm ra đường thi hành ngắn nhất tới câu lệnh hoặc điều kiện chưa được viếng thăm. Nếu tồn tại đường thi hành chưa

được viếng thăm, bước bốn sinh dữ liệu kiểm thử có hướng và tạo stub tự động được thực hiện. Nhiệm vụ của bước bốn là sinh dữ liệu kiểm thử tự động sao cho thăm được các câu lệnh hoặc điều kiện chưa được thăm. Phương pháp AS4UT [21] được tích hợp trong quá trình thực thi tượng trưng nhằm xử lý các lời gọi hàm trong đơn vị kiểm thử. Cụ thể, khóa luận áp dụng bước tiền xử lý CFG trong phương pháp AS4UT trong quá trình thực thi tượng trưng trên một đường thi hành. Khóa luận cải tiến bước tiền xử lý CFG này và bổ sung bước xử lý lời gọi phương thức. Chi tiết về phương pháp xử lý lời gọi phương thức được trình bày ở Mục 2.4.2. Đầu ra của bước này là tập các ràng buộc trên đường thi hành đang xét. Sau đó, phương pháp đề xuất sử dụng bộ giải Z3 để tìm nghiệm cho các ràng buộc và chuyển nghiệm thành dữ liệu kiểm thử mới. Các dữ liệu kiểm thử mới này tiếp tục được thực thi và phân tích đường thi hành đầu ra. Quá trình sinh dữ liệu kiểm thử tự động lặp lại cho đến khi không thể tìm thấy đường thi hành chưa thử viếng thăm.



Hình 2.5: Tổng quan quá trình sinh dữ liệu kiểm thử tự động.

2.4.2. Xử lý lời gọi phương thức của đối tượng

Như đã đề cập ở trên, phương thức của đối tượng có ảnh hưởng lớn tới hàm đang kiểm thử bởi thuộc tính của đối tượng có thể thay đổi qua các phương thức này. Khi đó, sau lời gọi phương thức, các biểu thức điều kiện sử dụng thuộc tính phụ thuộc nhiều vào sự thay đổi bởi lời gọi phương thức. Để thăm được các điều kiện chưa thăm được do ảnh hưởng bởi lời gọi phương thức, quá trình tiền xử lý CFG cần chú trọng xử lý các lời gọi này. Khóa luận đề xuất Thuật toán 2.4 để xử lý lời gọi phương thức. Ý tưởng chính của thuật toán đó là bổ sung một đối tượng giả kèm theo giá trị trả về khi xuất hiện lời gọi phương thức và cập nhật biến thực thi tượng trưng của đối tượng gọi phương thức bằng đối tượng giả.

Thuận toán 2.4 có bốn đầu vào lần lượt là bảng băm *MAP*, bảng tham chiếu *REF*, câu lệnh xuất lệnh lời gọi hàm *stm* và đỉnh CFG cfg_node tương ứng với câu lệnh đang xét. Bảng băm *MAP* là bảng băm ánh xạ các hàm sang số lần gọi của hàm đó trong đơn vị kiểm thử. Bảng tham chiếu *REF* là bảng lưu trữ ánh xạ giữa biến tượng trưng và giá trị tượng trưng tương ứng của nó. Bảng *MAP* và *REF* được phương pháp đề xuất kế thừa từ phương pháp AS4UT. Đỉnh CFG đầu vào được lấy từ CFG gốc của đơn vị kiểm thử.

Thuật toán 2.4 bắt đầu với việc trích xuất hàm được gọi $function$ từ đỉnh CFG đầu vào (dòng 1) và khởi tạo số lần gọi hiện tại của hàm đó là 0 (dòng 2). Nếu hàm $function$ đã tồn tại trong *MAP* thì thuật toán cập nhật số lần gọi hiện tại bằng số lần gọi cuối của $function$ trong *MAP* (dòng 3-5). Tiếp theo, thuật toán tăng số lần gọi hiện tại *cur_call* lên 1 (dòng 6) và thêm cặp ánh xạ $(function, cur\_call)$ vào *MAP* (dòng 7). Giá trị cũ nếu có của $function$ trong *MAP* sẽ được thay bởi cặp ánh xạ mới này. Thuật toán trích xuất tên $func\_name$ và kiểu trả về *return_type* của $function$ (dòng 8) và xác định đối tượng gọi hàm $function$ (dòng 9). Nếu đối tượng trả về khác *NULL* (hay hàm đang xét là phương thức) thuật toán sẽ tạo đối tượng giả trả về (dòng 11-16). Trước hết, tên $object\_name$ và kiểu $object\_type$ của đối tượng được trích xuất (dòng 11). Sau đó, thuật toán tạo tên đối tượng giả *stub_name* (dòng 12) và khởi tạo đối tượng giả với tên vừa tạo và kiểu đã xác định (dòng 13). Tiếp theo, thuật toán tìm kiếm biến tượng trưng ref_object có cùng tên và kiểu của đối tượng gốc (dòng 14). Đối tượng giả sẽ được đánh dấu trỏ tới biến tượng trưng ref_object (dòng 15) và được thêm vào *REF* (dòng 16). Việc đánh dấu đối tượng giả trỏ tới biến tượng trưng của đối tượng gốc có nghĩa là đối tượng giả này sẽ được sử dụng thay cho biến tượng trưng trong các câu lệnh sau của đơn vị kiểm thử. Bước sinh biến tượng trưng cho giá trị trả về (dòng 18-23) được kế thừa từ phương pháp AS4UT. Trước hết, thuật toán sinh biến tượng trưng cho giá trị trả về của hàm (dòng 19-20). Sau đó, biến tượng trưng này được thêm vào bảng *REF* (dòng 21) và thuật toán thay lời gọi hàm trong *stm* bằng biến tượng trưng vừa tạo. Cuối cùng, thuật toán trả về bảng *MAP*, *REF* và câu lệnh đã được cập nhật. Các thành phần sẽ tiếp tục được sử dụng bởi quá trình thực thi tượng trưng.

Thuật toán 2.4: Thuật toán xử lý lời gọi phương thức của đối tượng

Input: MAP - the map table transforming a function to its number of call
REF - the reference table containing symbolic variables
stm - the statement containing the function call expression
cfg_node - the corresponding CFG node of current stm

Output: MAP - the updated map table
REF - the updated reference table
stm - the refactored statement

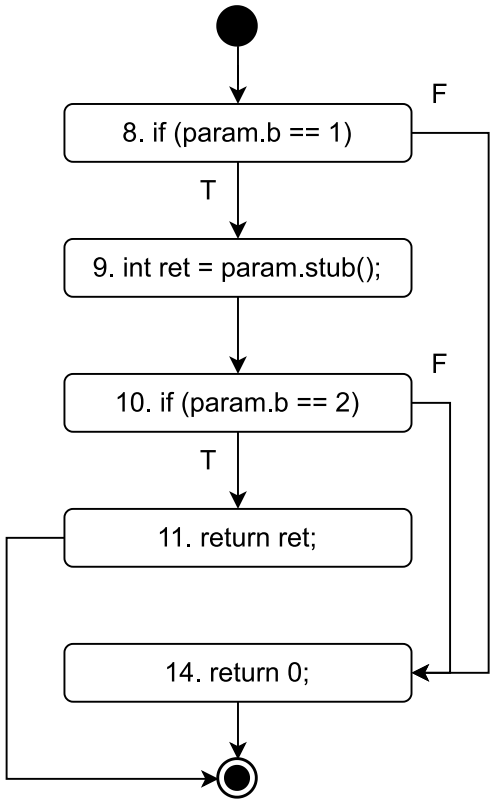
$function \leftarrow$ Get the called function from cfg_node;
$cur\_call \leftarrow 0$;
if $function \in MAP$ **then**
    $cur\_call \leftarrow$ Get the number of call of $function$ from MAP;
end
cur_call++;
Insert $(function, cur\_call)$ into MAP;
$(func\_name, return\_type) \leftarrow$ Get the name and return type of $function$;
$call\_object \leftarrow$ Get the object calling the $function$;
if $call\_object \neq NULL$ **then**
    $(object\_name, object\_type) \leftarrow$ Get the name and type of $call\_object$;
    $stub\_name \leftarrow func\_name + "\_call" + cur\_call + "\_" + object\_name$;
    $instance\_stub\_var \leftarrow$ Generate new instance symbolic variable $(stub\_name, object\_type)$;
    $ref\_object \leftarrow$ Get the class/struct symbolic variable having the same $(object\_name, object\_type)$ from REF;
    Mark $instance\_stub\_var$ pointing to ref_object;
    Add $instance\_stub\_var$ into REF;
end
if $return\_type$ ***is not*** $void$ **then**
    $stub\_name \leftarrow func\_name + "\_call" + cur\_call$;
    $return\_stub\_var \leftarrow$ Generate return symbolic variable $(stub\_name, return\_type)$;
    Add $return\_stub\_var$ into REF;
    $stm \leftarrow$ Replace function call expression by $stub\_name$;
end
return (MAP, REF, stm)

2.4.3. Ví dụ về thực thi tượng trưng kết hợp xử lý lời gọi phương thức của đối tượng

Đoạn mã 2.8 mô tả đoạn mã nguồn ví dụ minh họa phương pháp tạo stub tự động cho phương thức của đối tượng. Đơn vị được kiểm thử là hàm `foo(B param)` với tham số đầu vào là `param`, một đối tượng của lớp `B`. Đồ thị dòng điều khiển của đơn vị này được mô tả trong Hình 2.6. Logic của đơn vị kiểm thử gồm hai câu lệnh điều kiện, một câu lệnh gọi hàm và một hai câu lệnh trả giá trị. Dựa trên đồ thị dòng điều khiển khi xét đường thi hành 8-9-10-11, ta có thể nhận thấy điều kiện `if (param.b == 2)` tại dòng 10 bị ảnh hưởng với câu lệnh gọi phương thức ở dòng số 9.

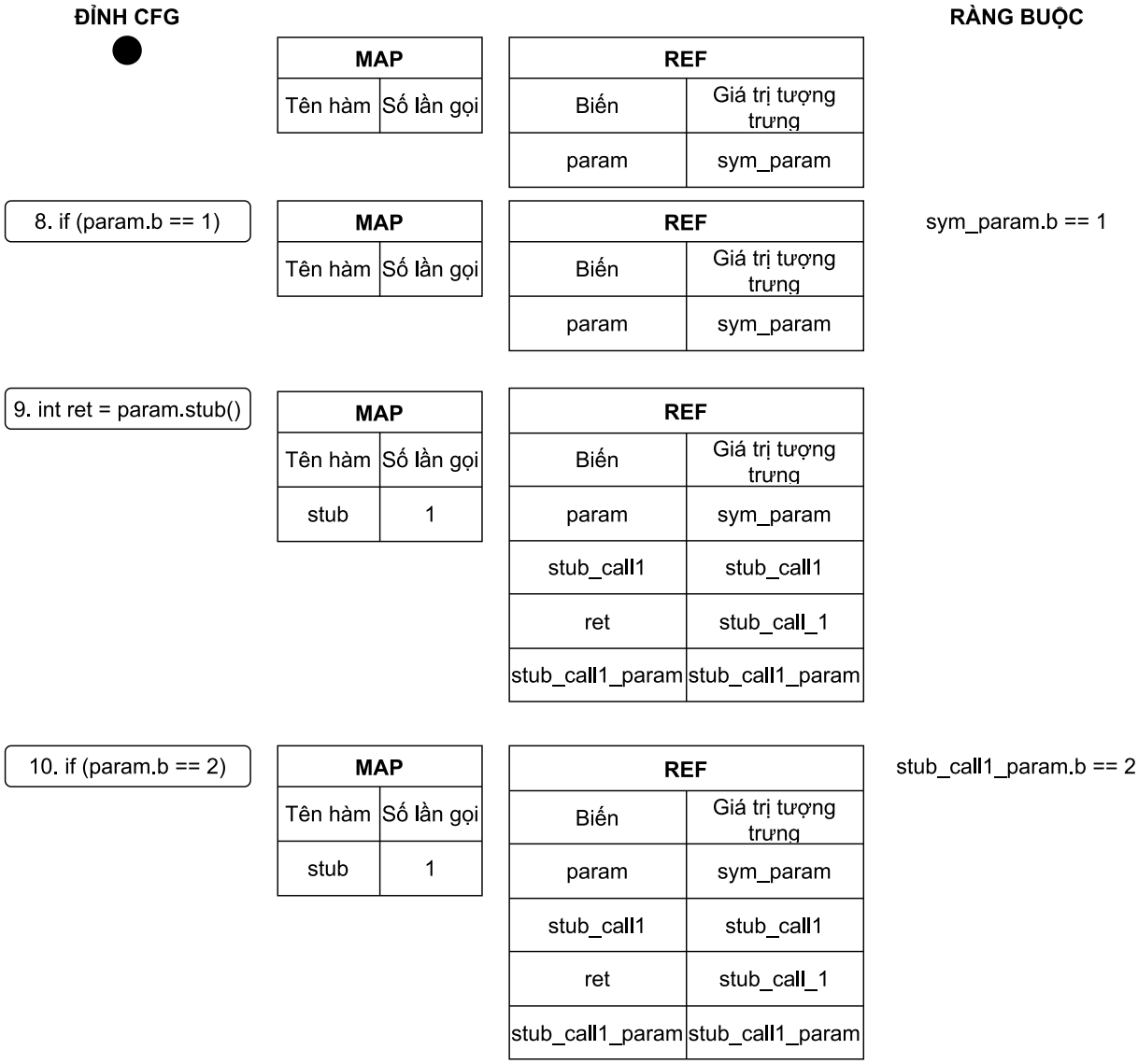
```
1  class B {
2    public:
3    int b;
4    B() {}
5    int stub() {}
6  };
7  int foo(B param) {
8    if (param.b == 1) {
9      int ret = param.stub();
10     if (param.b == 2) {
11       return ret;
12     }
13   }
14   return 0;
15 }
```

Đoạn mã 2.8: Mã nguồn minh họa phương pháp tạo stub tự động cho phương thức của đối tượng.



Hình 2.6: Đồ thị dòng điều khiển của hàm `foo(B param)` trong Đoạn mã 2.8.

Hình 2.7 mô tả quá trình thực thi tượng trưng áp dụng phương pháp đề xuất trên đường thi hành 8-9-10-11. Hình vẽ có bốn cột lần lượt biểu diễn đỉnh CFG đang xét, dữ liệu trong bảng *MAP* và bảng *REF* sau bước thực thi tượng trưng tại đỉnh CFG và cuối cùng là ràng buộc mới được sinh ra. Đầu tiên, ở trạng thái khởi tạo, quá trình thực



Hình 2.7: Quá trình thực thi tượng trưng kết hợp Thuật toán 2.4.

thi tượng trưng tạo các biến tượng trưng cho tham số của hàm. Hàm `foo(B param)` có một tham số đầu vào nên một biến tượng trưng `param` được tạo ra và có giá trị tượng trưng `sym_param`. Khi xét đỉnh CFG của câu điều kiện `if (param.b == 1)`, dựa trên bảng *REF* quá trình thực thi tượng trưng tạo ra ràng buộc mới `sym_param.b == 1`. Xét tiếp đỉnh CFG của câu lệnh `int ret = param.stub()`, lúc này, Thuật toán 2.4 được áp dụng để xử lý lời gọi phương thức của đối tượng `param`. Trước hết thuật toán cập nhật số lần gọi hàm `stub` lên 1 và lưu vào bảng *MAP*. Do hàm `stub` là phương thức nên thuật toán sẽ sinh đối tượng giả kèm theo giá trị trả về cho hàm này. Ba biến tượng trưng mới được cập nhật trong bảng *REF* gồm `stub_call1` - giá trị trả về tượng trưng, `ret` - biến được gán giá trị trả về của phương thức và `stub_call1_param` - đối tượng giả trỏ tới

biến tượng trưng `param`. Xét tiếp tới đỉnh CFG của câu điều kiện `if (param.b == 2)`, dựa trên thông tin của bảng *REF*, quá trình thực thi tượng trưng sẽ chuyển đổi biến `param` gốc thành `sym_param` và từ `sym_param` thành `stub_call1_param`. Như vậy một ràng buộc mới được sinh ra đó là `stub_call1_param.b == 2`. Ràng buộc này có thể hiểu là thông qua lời gọi phương thức `stub`, giá trị của thuộc tính `b` trong đối tượng `param` cần phải thay đổi thành 2 thay vì 1 như khởi tạo. Qua quá trình thực thi tượng trưng trên đường thi hành 8-9-10-11 kết hợp với bảng *MAP* và *REF*, ta thu được ba ràng buộc stub_call == 1, sym_param.b == 1, stub_call1_param.b == 2. Các ràng buộc này sau đó được chuẩn hóa theo cú pháp của SMT-LIB (ví dụ như hay `.b` thành `_attr_b`) và được đưa vào bộ giải Z3 để tìm nghiệm. Từ nghiệm tìm được, ta xác định được bộ dữ liệu kiểm thử để đi được đường thi hành 8-9-10-11 đó là biến `param` đầu cần khởi tạo thuộc tính `b` với giá trị 1, hàm `stub` được gọi 1 lần và trả về giá trị bất kỳ đồng thời thay đổi giá trị của thuộc tính `b` thành 2.

```
1   ....
2   int stub() {
3     static int cur_call = 0;
4     cur_call++;
5     if (TEST_CASE_NAME == "foo") {
6       if (cur_call == 1) {
7         int stub_call1_param_attr_b = 2;
8         this->b = stub_call1_param_attr_b;
9         int stub_call1 = 0;
10        return stub_call1;
11      }
12    }
13  }
14  ....
```

Đoạn mã 2.9: Sự thay đổi của hàm `stub` với bộ dữ liệu kiểm thử mới.

Đoạn mã 2.9 mô tả nội dung của hàm `stub` với bộ dữ liệu kiểm thử mới. Biến tĩnh `cur_call` lưu số lần gọi hàm trong một lần thực thi ca kiểm thử. Biến này được tăng lên một với mỗi lần gọi hàm. Câu lệnh điều kiện ở dòng 8 dùng để kiểm tra xem chương

trình đang thực hiện kiểm thử cho hàm nào. Biến `TEST_CASE_NAME` là một xâu kí tự toàn cục tạo bởi trình điều khiển kiểm thử nhằm lưu tên hàm đang kiểm thử. Do ví dụ đang xét kiểm thử trên hàm `foo` nên giả lập mã nguồn của hàm `stub` sẽ nằm trong các dòng 9-14. Câu lệnh điều kiện `cur_call == 1` thể hiện nếu đây là lần gọi thứ nhất của hàm `stub`, hàm này sẽ thực hiện các giả lập mã nguồn được viết trong nhánh đúng của điều kiện này. Dòng 10-13 của mã nguồn kiểm thử cho biết mã nguồn giả lập cho lần gọi thứ nhất thực hiện hai nhiệm vụ. Nhiệm vụ thứ nhất đó là thay đổi giá trị của thuộc tính `b` (dòng 11). Nhiệm vụ thứ hai đó là tạo giá trị trả về cho hàm. Do không có ràng buộc liên quan đến giá trị trả về của hàm `stub` nên phương pháp đề xuất chọn giá trị 0 để trả về.

Chương 3

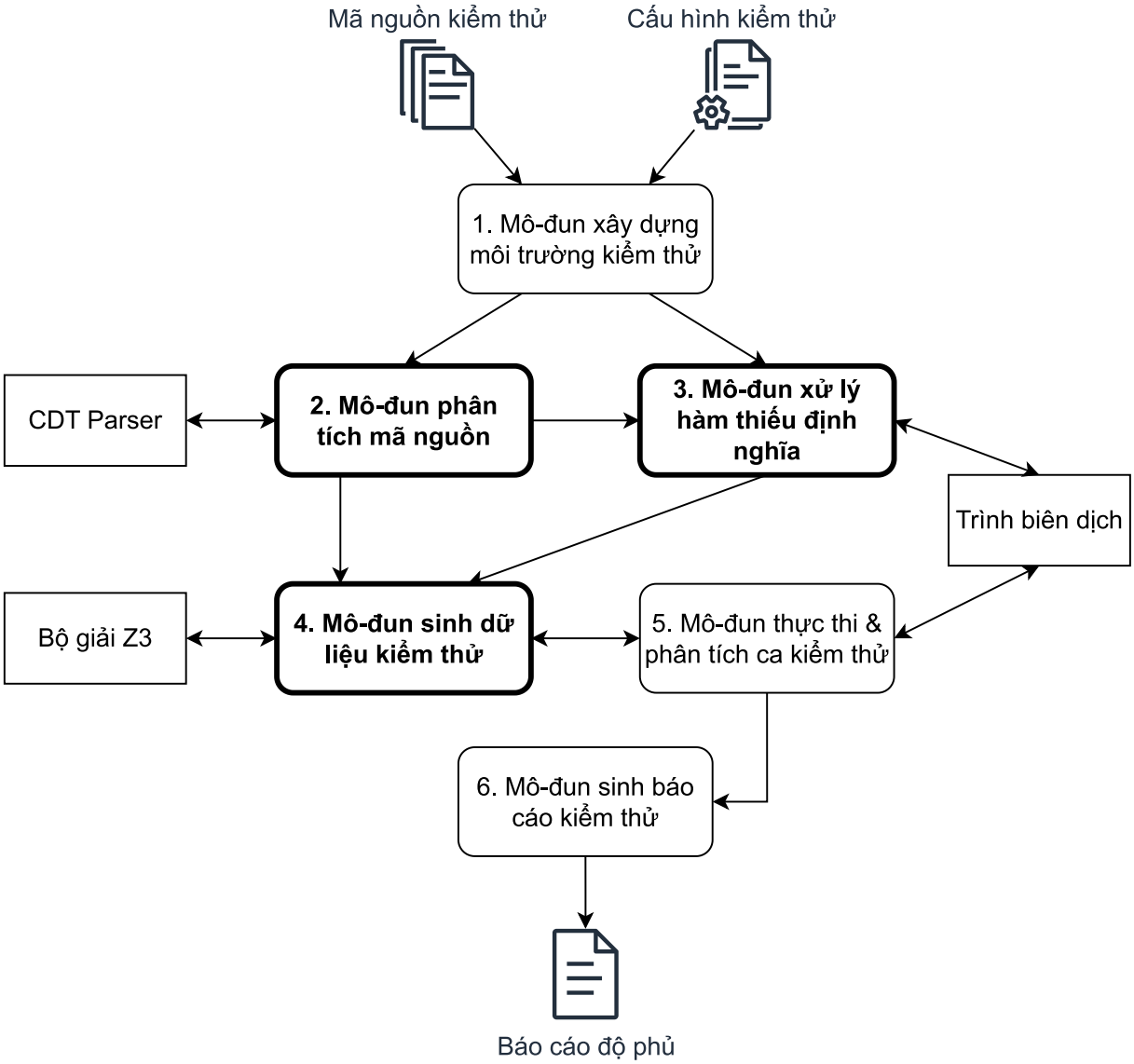
Cài đặt công cụ và thực nghiệm

3.1. Công cụ thực nghiệm

Nhằm đánh giá tính hiệu quả của phương pháp đề xuất, khóa luận đã cài đặt một số mô-đun và tích hợp phương pháp đề xuất trên công cụ AKAUTAUTO và thực hiện thực nghiệm trên công cụ này. AKAUTAUTO là một công cụ kiểm thử tự động cho mã nguồn C/C++, được nghiên cứu và phát triển bởi Phòng thí nghiệm Đảm bảo chất lượng phần mềm (Khoa Công nghệ thông tin, Trường Đại học Công nghệ, ĐHQGHN) và đơn vị FPT-GAM (FPT Global Automative & Manufacturing). Khóa luận kế thừa kiến trúc của công cụ AKAUTAUTO phiên bản 5.9.2, phiên bản tích hợp sẵn phương pháp kiểm thử tượng trưng động và phương pháp sinh giả lập mã nguồn tự động AS4UT, và bổ sung, cải tiến một số mô-đun trong kiến trúc gốc để tích hợp phương pháp đề xuất (phiên bản 5.9.2-thesis).

3.1.1. Tổng quan kiến trúc công cụ AKAUTAUTO

Công cụ AKUTAUTO được phát triển bằng ngôn ngữ Java với kiến trúc bao gồm sáu mô-đun chính lần lượt là mô-đun xây dựng môi trường kiểm thử, mô-đun phân tích mã nguồn, mô-đun xử lý hàm thiếu định nghĩa, mô-đun sinh dữ liệu kiểm thử, mô-đun thực thi & phân tích ca kiểm thử và mô-đun sinh báo cáo kiểm thử. Hình 3.1 mô tả kiến trúc của công cụ AKAUTAUTO gồm sáu mô-đun kể trên và các thành phần ngoài gồm trình biên dịch C/C++, bộ giải Z3 và thư viện phân tích mã nguồn CDT Parser. Trong



Hình 3.1: Kiến trúc công cụ AKAUTAUTO.

đó, khóa luận đã bổ sung mô-đun xử lý hàm thiếu định nghĩa so với kiến trúc gốc, và cải tiến hai mô-đun phân tích mã nguồn và mô-đun sinh dữ liệu kiểm thử (thể hiện bởi các khối in đậm).

Đầu vào của công cụ AKAUTAUTO gồm hai thành phần chính lần lượt là các tệp mã nguồn kiểm thử và cấu hình kiểm thử. Hai thành phần này được cung cấp bởi kiểm thử viên hoặc lập trình viên. Mô-đun xây dựng môi trường kiểm thử là mô-đun tiếp nhận các thành phần đầu vào và đảm nhiệm vai trò thực hiện hai tác vụ tiền xử lý mã nguồn trong pha xây dựng môi trường kiểm thử. Hai tác vụ tiền xử lý đã được đề cập ở Mục 2.2 gồm thiết lập môi trường và tạo môi trường tính toán độ phủ. Mô-đun xây dựng môi trường kiểm thử cấu thành bởi ba thành phần chính đó là thành phần thiết lập

môi trường, thành phần tải dự án và thành phần tạo môi trường tính toán độ phủ. Trong đó, hai thành phần thiết lập môi trường và tạo môi trường tính toán độ phủ thực hiện hai tác vụ tương ứng đã nêu. Thành phần tải dự án có nhiệm vụ xác định các tệp mã nguồn được người dùng thiết lập, sau đó kiến tạo đỉnh tệp của từng cây cấu trúc tệp tương ứng. Đầu ra của mô-đun gồm các tệp mã nguồn đã được thêm lệnh đánh dấu và tập các cây cấu trúc tệp cơ bản sinh bởi thành phần tải dự án. Các tệp mã nguồn chứa lệnh đánh dấu sẽ được sử dụng bởi mô-đun xử lý hàm thiếu định nghĩa nhằm sinh thân hàm giả trên tệp mã nguồn này. Tập cây cấu trúc tệp cơ bản và mã nguồn gốc sẽ được phân tích bởi mô-đun phân tích mã nguồn.

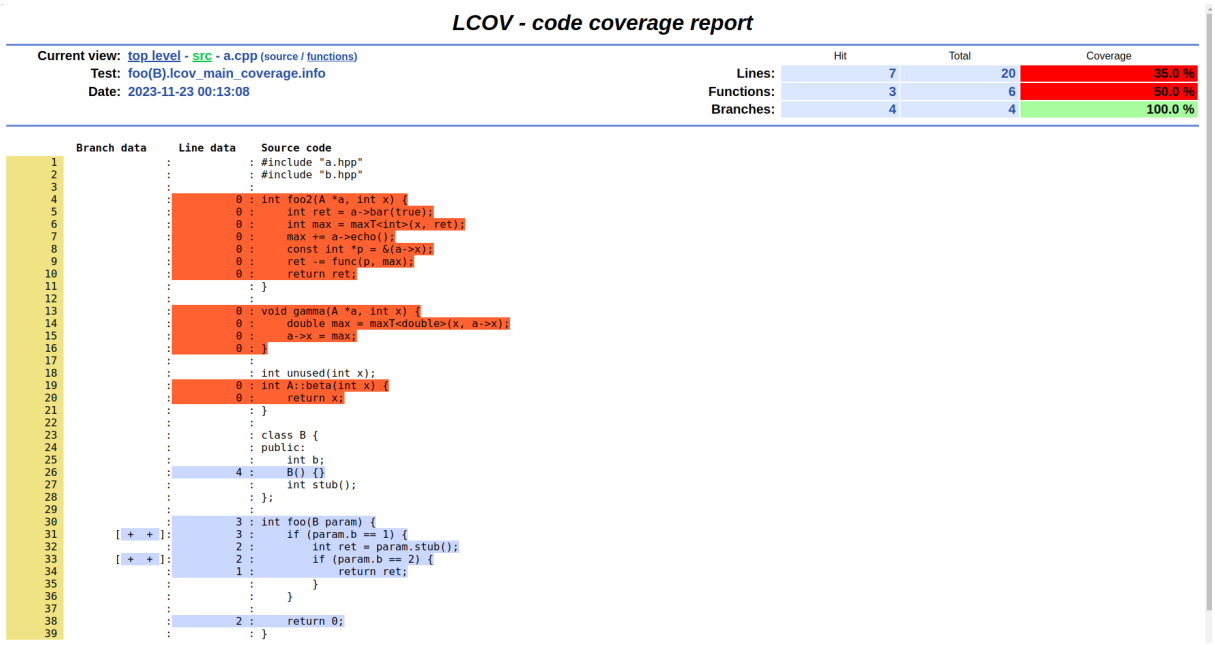
Các mô-đun tiếp theo trong kiến trúc của công cụ AKAUTAUTO được khóa luận bổ sung hoặc cải tiến gồm mô-đun phân tích mã nguồn, mô-đun xử lý hàm thiếu định nghĩa và mô-đun sinh dữ liệu kiểm thử. Chi tiết về sự cải tiến của mô-đun phân tích mã nguồn và mô-đun sinh dữ liệu kiểm thử được trình bày lần lượt ở Mục 3.1.2 và Mục 3.1.4. Mô-đun xử lý hàm thiếu định nghĩa được bổ sung so với kiến trúc gốc nhằm giải quyết các vấn đề phát sinh bởi nguyên mẫu hàm thiếu định nghĩa. Nội dung chi tiết của mô-đun này được trình bày ở Mục 3.1.3.

Khóa luận kế thừa mô-đun thực thi và phân tích ca kiểm thử từ kiến trúc gốc của công cụ. Mô-đun này đóng vai trò sinh trình điều khiển kiểm thử, thực thi trình điều khiển và phân tích đường thi hành thu được sau khi chạy ca kiểm thử. Cấu trúc của mô-đun gồm ba thành phần chính với nhiệm vụ chi tiết của từng thành phần như sau:

- Thành phần sinh trình điều khiển kiểm thử: Đảm nhiệm hai vai trò chuyển đổi bộ nghiệm giải từ ràng buộc thành dữ liệu kiểm thử C++ và sinh trình điều khiển kiểm thử dựa trên dữ liệu mới. Trình điều khiển kiểm thử là một tệp C++ được tự động sinh ra nhằm chuyển đổi dữ liệu kiểm thử C++ thành các cú pháp C++ giúp khởi tạo giá trị tham số đầu vào của đơn vị kiểm thử. Các cú pháp C++ trong trình điều khiển gồm hai phần chính đó là phần định nghĩa tham số đầu vào và phần gọi đơn vị kiểm thử. Trình điều khiển cũng đảm nhiệm chuyển hóa dữ liệu kiểm thử C++ thành các mã nguồn giả lập cho các hàm cần stub.
- Thành phần thực thi ca kiểm thử: Có nhiệm vụ biên dịch trình điều khiển và các tệp mã nguồn chứa lệnh đánh dấu thành tệp đối tượng rồi sau đó liên kết các tệp này để tạo thành tệp thực thi. Công cụ chạy tệp thực thi này và thu được danh sách các câu lệnh, các nhánh điều kiện được viếng thăm bởi ca kiểm thử.

- Thành phần phân tích đường thi hành: Có chức năng ánh xạ danh sách các câu lệnh, nhánh điều kiện được viếng thăm sang đỉnh tương ứng trong CFG của đơn vị kiểm thử. Tiếp đó, thành phần cập nhật dữ liệu viếng thăm của CFG dựa trên tập đỉnh được ánh xạ. Cuối cùng, thành phần phân tích đường thi hành sẽ tìm đường thi hành ngắn nhất chưa được viếng thăm và truyền đường thi hành này sang mô-đun sinh dữ liệu kiểm thử.

Mô-đun cuối cùng trong kiến trúc công cụ là mô-đun sinh báo cáo kiểm thử. Mô-đun này có vai trò tính toán độ phủ của đơn vị kiểm thử dựa trên dữ liệu viếng thăm của CFG và tạo báo cáo kiểm thử sau khi quá trình sinh dữ liệu kiểm thử tự động kết thúc. Hình 3.2 mô tả ví dụ báo cáo kiểm thử LCOV của hàm `foo`. Khóa luận đã sinh dữ liệu kiểm thử tự động cho hàm `foo` và đạt được độ phủ 100% câu lệnh cũng như 100% nhánh với ba ca kiểm thử. Các thông tin ở góc trên phải của báo cáo cho biết 7/20 dòng lệnh của tệp chứa hàm `foo` và 4/4 nhánh điều kiện có trong hàm được thăm bởi ba ca kiểm thử này.

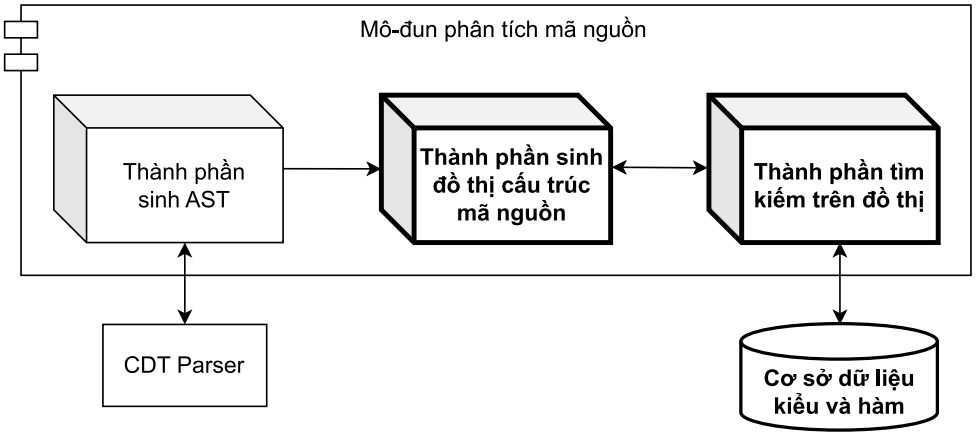


Hình 3.2: Báo cáo kiểm thử LCOV của hàm `foo`.

3.1.2. Mô-đun phân tích mã nguồn

Mô-đun phân tích mã nguồn đảm nhiệm vai trò trích xuất, phân tích thông tin từ AST của mã nguồn kiểm thử, xây dựng đồ thị cấu trúc mã nguồn và cung cấp dịch vụ

tìm kiếm thông tin đỉnh trên đồ thị. Hình 3.3 mô tả cấu trúc của mô-đun với ba thành phần lần lượt là thành phần sinh AST, thành phần sinh đồ thị cấu trúc mã nguồn và thành phần tìm kiếm trên đồ thị. Trong đó, khóa luận kế thừa thành phần sinh AST, cải tiến mô-đun sinh đồ thị cấu trúc mã nguồn và bổ sung thành phần tìm kiếm trên đồ thị.



Hình 3.3: Các thành phần trong mô-đun phân tích mã nguồn.

Mô-đun phân tích mã nguồn nhận đầu vào là mã nguồn kiểm thử gốc và các đỉnh tệp từ đầu ra của mô-đun xây dựng môi trường. Bắt đầu từ mỗi đỉnh tệp, dựa trên quan hệ cha con trên AST của từng tệp mã nguồn, quá trình sinh cây cấu trúc tệp được diễn ra song song. Sau đó, tập cây cấu trúc tệp được chuyển sang thành phần hoàn thiện đồ thị cấu trúc để bổ sung cạnh kế thừa và cạnh lời gọi hàm. Tại thành phần này, từng cây cấu trúc tệp được xét và tìm kiếm các đỉnh có tiêu chí phù hợp với kiểu cạnh đang xét trên các cây cấu trúc còn lại. Kết quả của quá trình xây dựng là đồ thị cấu trúc mã nguồn kiểm thử.

Khóa luận đã cải tiến thành phần sinh đồ thị cấu trúc mã nguồn như sau. Trước hết, khóa luận tích hợp thành phần ánh xạ cây cấu trúc và thành phần phân tích phụ thuộc [21] trong kiến trúc gốc thành mô-đun sinh đồ thị cấu trúc mã nguồn. Quá trình tích hợp bổ sung các luồng xử lý song song quá trình sinh cây cấu trúc tệp giúp giảm thời gian phân tích mã nguồn. Sau đó, khóa luận bổ sung các loại đỉnh mới biểu thị cho các kiểu khai báo mới trong ngôn ngữ C++ như không gian tên, câu lệnh sử dụng không gian tên, khai báo lớp không cần tên kiểu, v.v. Qua đó, thành phần này có thể phân tích và biểu diễn được các đặc trưng mới của C++ trên đồ thị cấu trúc mã nguồn.

Tìm kiếm thông tin đỉnh trên đồ thị là quá trình cốt lõi phục vụ cho quá trình xử lý hàm thiếu định nghĩa và quá trình sinh dữ liệu kiểm thử tự động. Trong kiến trúc gốc, quá trình tìm kiếm duyệt qua các đỉnh trên đồ thị, kiểm tra thông tin của đỉnh có hợp với tiêu chí tìm kiếm rồi thêm vào tập kết quả nếu hợp lệ. Trong thực tế, đồ thị cấu trúc thường rất lớn. Vậy nên quá trình tìm kiếm tốn nhiều bộ nhớ bởi quá trình đệ quy xuống đỉnh. Để khắc phục nhược điểm này, khóa luận bổ sung thành phần tìm kiếm trên đồ thị với ý tưởng chính dựa trên tìm kiếm kết hợp bảng băm và cơ sở dữ liệu. Việc tìm kiếm thông tin đỉnh thường sử dụng hai tiêu chí đó là tên hàm và tên kiểu. Tận dụng điều này, khóa luận tạo cơ sở dữ liệu lưu trữ thông tin về kiểu và hàm của mã nguồn kiểm thử. Cơ sở dữ liệu gồm hai trường chính đó là *id* - khóa chính biểu thị mã định danh của hàm hoặc kiểu và *name* - tên hàm hoặc tên kiểu. Sau quá trình xây dựng đồ thị cấu trúc mã nguồn, mỗi đỉnh trong đồ thị được cấp một mã định danh cố định *id*. Mã định danh này là giá trị băm xâu đường dẫn của đỉnh, trong đó xâu đường dẫn là chuỗi tên các đỉnh trên đường đi từ đỉnh tệp xuống đỉnh đang xét. Khóa luận sử dụng một bảng băm để ánh xạ *id* và đỉnh sở hữu *id*. Để nhanh chóng tìm kiếm tên trong cơ sở dữ liệu, khóa luận sử dụng hệ quản trị cơ sở dữ liệu SQLite[1] với cơ chế FULLTEXT INDEX trên cột *name*. Để tìm kiếm một đỉnh phù hợp, trước hết quá trình tìm kiếm sẽ thu thập các dòng trong cơ sở dữ liệu có chứa tên hàm hoặc tên kiểu, trích xuất *id* và dựa vào bảng băm để lấy ra đỉnh cần xét.

Quá trình nghiên cứu, cải tiến mô-đun phân tích mã nguồn tốn nhiều thời gian do số lượng tệp ảnh hưởng bởi mô-đun này là rất lớn (113 tệp ảnh hưởng và 5476 LOC thay đổi). Tuy nhiên, quá trình trên là cần thiết bởi công cụ AKAUTAUTO phiên bản 5.9.2 thường gặp lỗi hết bộ nhớ ảo trong quá trình phân tích mã nguồn khi chạy trên các dự án có kích thước lớn, cấu trúc phức tạp và có nhiều sự tương tác trong các thành phần với nhau. Sau quá trình cải tiến, công cụ AKAUTAUTO phiên bản 5.9.2-thesis đã có thể phân tích được những bộ mã nguồn có kích thước lớn, đồng thời không gặp tình trạng hết bộ nhớ ảo trong quá trình chạy.

3.1.3. Mô-đun xử lý hàm thiếu định nghĩa

Mô-đun xử lý hàm thiếu định nghĩa được khóa luận bổ sung so với kiến trúc gốc nhằm thực hiện nhiệm vụ của pha xử lý hàm thiếu định nghĩa. Hình 3.4 mô tả các thành

[1]https://www.sqlitetutorial.net/sqlite-full-text-search/

phần cấu thành lên mô-đun gồm thành phần thu thập, thành phần lọc tìm ứng viên, thành phần xử lý nguyên mẫu hàm ảo và thành phần sinh thân hàm giả. Mô-đun xử lý hàm thiếu định nghĩa được thiết kế để xử lý hai loại nguyên mẫu hàm thiếu định nghĩa như đã mô tả ở Mục 2.3. Chi tiết về các thành phần như sau.

- Thành phần thu thập: Đóng vai trò thu thập danh sách các nguyên mẫu hàm thiếu định nghĩa cần sinh thân hàm giả. Thành phần thu thập sử dụng các công cụ tiện ích của trình biên dịch để thu thập danh sách này. Phương pháp thu thập đã được trình bày ở Mục 2.3.1.

- Thành phần xử lý nguyên mẫu hàm ảo: Có chức năng tìm kiếm các nguyên mẫu hàm ảo thiếu định nghĩa sử dụng phương pháp đề xuất ở Mục 2.3.2. Danh sách các nguyên mẫu hàm ảo tìm được sẽ được truyền sang thành phần lọc tìm ứng viên để xử lý theo Thuật toán 2.3. Các nguyên mẫu hàm ảo còn lại sẽ được sinh cạnh định nghĩa tương ứng với đỉnh biểu thị hàm trong đồ thị cấu trúc.

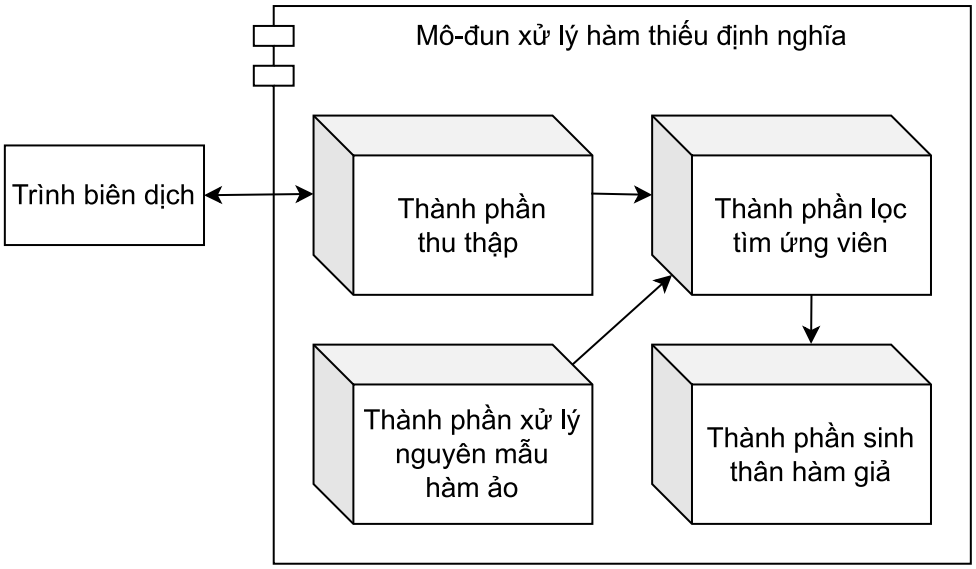
- Thành phần lọc tìm ứng viên: Thực hiện quá trình áp dụng Thuật toán 2.2 trên danh sách các hàm thiếu định nghĩa trả về bởi thành phần thu thập và các nguyên mẫu hàm ảo thiếu định nghĩa tìm được. Đầu ra của thành phần là danh sách các đỉnh trên đồ thị cấu trúc mã nguồn cần được sinh hàm giả.

- Thành phần sinh thân hàm giả: Đảm nhiệm vai trò sinh thân hàm giả cho danh sách các đỉnh được tổng hợp bởi thành phần lọc tìm ứng viên.

Đầu ra của mô-đun xử lý hàm thiếu định nghĩa là mã nguồn chứa các câu lệnh đánh dấu đã được bổ sung thân hàm giả cho các nguyên mẫu hàm thiếu định nghĩa. Mã nguồn đã chỉnh sửa này sau đó được sử dụng để tính toán độ phủ khi chạy các ca kiểm thử bởi mô-đun thực thi và phân tích ca kiểm thử.

3.1.4. Mô-đun sinh dữ liệu kiểm thử

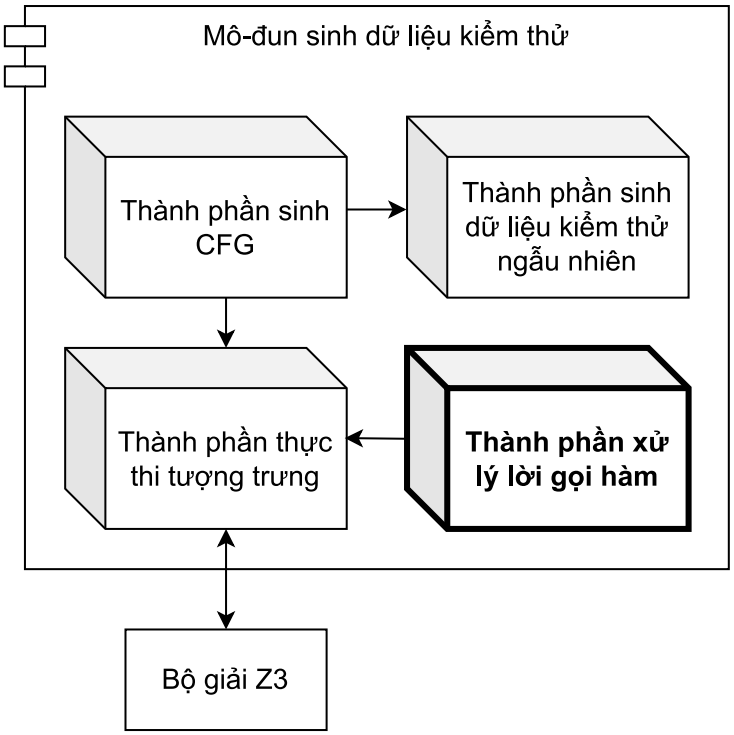
Mục tiêu của mô-đun sinh dữ liệu kiểm thử là tự động tạo ra các ca kiểm thử cho các đơn vị cần kiểm thử. Mô-đun được thiết kế như Hình 3.5 với bốn thành phần lần lượt là thành phần sinh CFG, thành phần sinh dữ liệu kiểm thử ngẫu nhiên, thành phần thực thi tượng trưng và thành phần xử lý lời gọi hàm. Trong đó, thành phần xử lý lời gọi hàm



Hình 3.4: Các thành phần trong mô-đun xử lý hàm thiếu định nghĩa.

được cải tiến bởi phương pháp đề xuất ở Mục 2.4.2, các thành phần còn lại được kế thừa từ kiến trúc gốc. Chi tiết về các thành phần như sau.

- Thành phần sinh CFG: Có chức năng xây dựng CFG cho các đơn vị trong mã nguồn.
- Thành phần sinh dữ liệu kiểm thử ngẫu nhiên: Đảm nhiệm vai trò sinh các ca kiểm thử với dữ liệu ngẫu nhiên trong pha sinh dữ liệu kiểm thử tự động (mô tả ở Mục 2.4).
- Thành phần thực thi tượng trưng: Đóng vai trò thực hiện quá trình thực thi tượng trưng trên đường thi hành chưa được viếng thăm và sinh ra các ràng buộc tạo các điểm quyết định trong đồ thị dòng điều khiển. Sau đó, thành phần thực thi tượng trưng sử dụng bộ giải Z3 để giải các ràng buộc và tạo ra các giá trị đầu vào mới.
- Thành phần xử lý lời gọi hàm: Có nhiệm vụ tạo giả lập mã nguồn tự động cho các lời gọi hàm xuất hiện trong đường thi hành. Khóa luận kế thừa thành phần xử lý lời gọi hàm dựa trên phương pháp AS4UT và áp dụng phương pháp xử lý lời gọi phương thức được đề xuất ở Mục 2.4.2. Đầu ra của thành phần là các thay đổi cần thiết trên CFG, bảng *MAP* và *REF* để giúp quá trình thực thi tượng trưng sinh ra bộ dữ liệu kiểm thử mới cho đường thi hành chưa viếng thăm.



Hình 3.5: Các thành phần trong mô-đun sinh dữ liệu kiểm thử.

3.2. Mục tiêu, độ đo đánh giá và dữ liệu thực nghiệm

Mục tiêu tổ chức thực nghiệm là để đánh giá tính hiệu quả của phương pháp đề xuất trên hai tiêu chí với các độ đo tương ứng như sau:

1. Đánh giá tính hiệu quả trong quá trình chuẩn bị môi trường kiểm thử cho mã nguồn thiếu định nghĩa giữa phương pháp xử lý hàm thiếu định nghĩa (trình bày ở Mục 2.3) và phương pháp truyền thống. Để so sánh giữa hai phương pháp, khóa luận sử dụng độ đo thời gian cần thiết để chuẩn bị môi trường kiểm thử. Ngoài ra, khóa luận cũng xem xét tính đúng đắn của phương pháp đề xuất.

2. Đánh giá tính hiệu quả khi sinh dữ liệu kiểm thử tự động cho một số mã nguồn C/C++ giữa phương pháp đề xuất ở Mục 2.4.2 và phương pháp hiện tại trong công cụ phiên bản 5.9.2 - phương pháp kiểm thử tượng trưng động kết hợp tự động giả lập mã nguồn AS4UT (gọi tắt là phương pháp AKAUTAUTO). Để so sánh giữa hai phương pháp, khóa luận sử dụng các độ đo về độ phủ mã nguồn C_1, C_3, số lượng ca kiểm thử sinh ra, thời gian sinh và bộ nhớ sử dụng trung bình cho mỗi ca kiểm thử.

Về môi trường thực nghiệm, khóa luận sử dụng máy tính với các thông số cấu hình như sau: Ubuntu 22.04, AMD® Ryzen™ 7-5800H CPU @ 3.2GHz x 8, 16GBs RAM. Thực nghiệm được tiến hành trên một số mã nguồn mở trên Github và mã nguồn dự án thực tế thuộc đơn vị FPT-GAM như sau:

- C-plus-plus[2] (42179 LOC): Mã nguồn mở viết bằng ngôn ngữ C++ cài đặt các thuật toán thường sử dụng trong lập trình thi đấu và khoa học máy tính.
- Box2d[3] (78811 LOC): Một thư viện cung cấp các phương thức để tính toán sự tương tác vật lý giữa các vật thể trong môi trường 2 chiều.
- FPT1 (23024 LOC): Mô-đun vận hành và điều hướng các yêu cầu gửi đến các dịch vụ trong trình điều khiển đa phương tiện, thuộc một dự án phát triển phần mềm xe ô tô của đơn vị FPT-GAM.
- FPT2 (35891 LOC): Mô-đun cung cấp dịch vụ quản lý danh bạ, cuộc gọi trên trình điều khiển đa phương tiện, cũng thuộc dự án trên.

Khóa luận sử dụng hai dự án mã nguồn mở nhằm đảm bảo tính khách quan của thực nghiệm. Trong đó, dự án C-plus-plus được sử dụng để đánh giá khả năng sinh dữ liệu kiểm thử của phương pháp đề xuất trên các mã nguồn đơn giản, ít sự liên kết giữa các tệp. Dự án Box2d được chọn để đánh giá tính hiệu quả của phương pháp đề xuất trên các mã nguồn sử dụng nhiều đặc trưng mới của C++. Thêm vào đó, khóa luận cũng sử dụng hai mã nguồn thuộc đơn vị FPT-GAM nhằm đồng thời đánh giá tính hiệu quả và khả năng áp dụng thực tiễn của phương pháp đề xuất.

3.3. Đánh giá khả năng xử lý hàm thiếu định nghĩa

3.3.1. Cách thức tổ chức thực nghiệm

Nhằm đánh giá tính hiệu quả về mặt thời gian khi xử lý hàm thiếu định nghĩa, đồng thời đảm bảo tính khách quan của thực nghiệm, quá trình thực nghiệm được tiến hành bởi lập trình viên[4] trực thuộc đơn vị FPT-GAM. Để mô phỏng môi trường chứa hàm

[2]https://github.com/TheAlgorithms/C-Plus-Plus
[3]https://github.com/erincatto/box2d
[4]Trần Trọng Năm - namtt25@fpt.com

thiếu định nghĩa, khóa luận tiến hành thực nghiệm tính toán thời gian cần thiết để chuẩn bị môi trường kiểm thử cho một số mô-đun trong các mã nguồn mở kể trên. Ba mô-đun thực nghiệm gồm mô-đun collision (xử lý va chạm) trong mã nguồn Box2d, mô-đun FPT1 và mô-đun FPT2. Do mã nguồn C-plus-plus gồm các tệp mã nguồn không liên quan đến nhau nên khóa luận không đánh giá khả năng xử lý hàm thiếu định nghĩa trên mã nguồn này.

Chuẩn bị môi trường kiểm thử là bước đầu tiên trong quy trình kiểm thử tự động. Trong bước này, lập trình viên cần bóc tách mã nguồn mình cần kiểm thử trong dự án và chỉnh sửa mã nguồn sao cho ta có thể tạo được tệp thực thi từ mã nguồn. Để tạo được tệp thực thi, lập trình viên cần xác định các kiểu dữ liệu và các hàm được sử dụng trong mã nguồn mà nằm ở mô-đun khác rồi xử lý các thành phần này để biên dịch được mã nguồn. Sau đó, lập trình viên cần xử lý các hàm thiếu định nghĩa phát sinh do mô-đun chưa phát triển xong để liên kết được các tệp và tạo thành tệp thực thi. Khóa luận tiến hành thu thập thời gian lập trình viên chuẩn bị môi trường kiểm thử giữa phương pháp truyền thống (tức làm thủ công các bước) và phương pháp đề xuất (tức làm thủ công hai bước đầu và để công cụ tự động xử lý các hàm thiếu định nghĩa).

3.3.2. Kết quả thực nghiệm

Bảng 3.1 trình bày kết quả thực nghiệm trên mô-đun Box2d, FPT1 và FPT2. Các cột trong bảng có ý nghĩa như sau:

- "Module": Tên mô-đun được xét.
- "File": Số lượng tệp trong mô-đun.
- "Undef": Số lượng hàm thiếu định nghĩa sau khi bóc tách mô-đun.
- "Compilable": Thời gian cần thiết để bóc tách mô-đun và chỉnh sửa mã nguồn sao cho biên dịch được.
- "Linkable": Thời gian cần thiết để xử lý các hàm thiếu định nghĩa sao cho tạo tệp thực thi được. Trong đó, cột "Manual" thể hiện thời gian xử lý thủ công còn cột "Propose" thể hiện thời gian xử lý sử dụng phương pháp đề xuất (tính bằng giây).

Bảng 3.1: Kết quả thời gian chuẩn bị môi trường trên ba mô-đun thực nghiệm

Module	**File**	**Undef**	**Compilable**	**Linkable**	
				Manual	Propose
collision (Box2d)	53	5	0:02:39	0:08:23	0:00:04
FPT1	45	263	1:04:22	0:14:48	0:00:28
FPT2	126	333	2:05:56	1:31:01	0:02:24

Kết quả thực nghiệm cho thấy thời gian xử lý các hàm thiếu định nghĩa thủ công chiếm một phần đáng kể trong tổng thời gian chuẩn bị môi trường, trong đó thời gian xử lý trên mô-đun collision là 75,98%, 18,70% với mô-đun FPT1 và 41,96% với mô-đun còn lại. Khi áp dụng phương pháp đề xuất, thời gian xử lý các hàm thiếu định nghĩa giảm đáng kể so với phương pháp truyền thống, giảm 8 phút 19 giây trên mô-đun collision, 14 phút 20 giây trên mô-đun FPT1 và giảm 1 tiếng 29 phút trên mô-đun còn lại. Dựa vào kết quả, có thể nhận thấy rằng thời gian xử lý của hai phương pháp phụ thuộc vào số lượng hàm thiếu định nghĩa có trong mô-đun.

3.3.3. Đánh giá

Kết quả thực nghiệm cho thấy rằng phương pháp đề xuất có khả năng giảm đáng kể thời gian chuẩn bị môi trường kiểm thử cho mã nguồn chứa hàm thiếu dịnh nghĩa. Một số lí do chính dẫn đến sự cải thiện đáng kể này như sau:

- Trước hết, thời gian xử lý thủ công bị ảnh hưởng bởi kích thước và độ phức tạp của mã nguồn. Các mô-đun có thể phụ thuộc bởi rất nhiều mô-đun khác và đồng thời chúng cũng chứa rất nhiều tệp. Điều này gây khó khăn cho lập trình viên khi họ cần xác định vị trí các tệp chứa hàm thiếu định nghĩa. Phương pháp đề xuất chỉ quan tâm tới danh sách các hàm trả về bởi trình biên dịch và kết hợp duyệt trên đồ thị cấu trúc mã nguồn nên nhanh chóng xác định và xử lý đúng hàm thiếu định nghĩa.
- Quá trình xử lý hàm thiếu định nghĩa thủ công đòi hỏi lập trình viên có kiến thức về mã nguồn kiểm thử. Điều này khiến lập trình viên tốn nhiều thời gian để nghiên cứu mã nguồn và xử lý chính xác các hàm thiếu định nghĩa. Trong bối cảnh ngôn ngữ C++ đang được phát triển với nhiều đặc trưng mới ra mắt, việc xử lý thủ công có thể tiêu tốn nhiều chi phí về nhân lực và thời gian hơn.

- Cuối cùng, phương pháp đề xuất có thể xử lý song song các hàm trong khi lập trình viên cần xử lý tuần tự từng hàm. Điều này dẫn tới thời gian xử lý giảm đáng kể so với phương pháp truyền thống.

Để đánh giá tính đúng đắn của phương pháp đề xuất, khóa luận đã tiến hành thực nghiệm sinh dữ liệu kiểm thử tự động cho các đơn vị trong mã nguồn sau khi xử lý hàm thiếu định nghĩa (trình bày ở Mục 3.4). Kết quả cho thấy các ca kiểm thử đều thực thi được và không gặp lỗi trong quá trình liên kết. Như vậy, phương pháp đề xuất đã xử lý đúng và đủ các hàm thiếu định nghĩa cần thiết. Điều này được khẳng định bởi nếu phương pháp xử lý thiếu hoặc sai hàm, mã nguồn kiểm thử không thể liên kết và thực thi các ca kiểm thử được. Tính đến thời điểm nghiên cứu của khóa luận, phương pháp đề xuất đã giải quyết được các mã nguồn thực nghiệm chứa các đặc trưng của ngôn ngữ C++ ở nhiều phiên bản, mới nhất là bản C++20. Tuy nhiên trong các phiên bản tương lai, tính đúng đắn của phương pháp đề xuất có thể bị ảnh hưởng.

Như vậy, phương pháp đề xuất có thể áp dụng vào thực tiễn để giúp giảm thời gian kiểm thử đơn vị tự động. Tuy nhiên, có thể nhận thấy rằng thấy phương pháp đề xuất vẫn còn phụ thuộc vào bước bóc tách mã nguồn và xử lý sao cho biên dịch được. Việc phụ thuộc vào các bước thủ công khiến quá trình chuẩn bị môi trường kiểm thử tốn nhiều thời gian, đặc biệt là khi kiểm thử cho mã nguồn kích thước lớn. Một số lí do dẫn đến phương pháp chưa hoàn toàn tự động được như sau:

- Quá trình bóc tách mã nguồn yêu cầu công cụ trích xuất được đúng các tệp có liên quan đến mô-đun người dùng mong muốn kiểm thử. Điều này gây khó khăn bởi để trích xuất được chính xác mô-đun, công cụ cần phân tích mã nguồn của cả dự án để biết được các quan hệ giữa mô-đun và các thành phần khác. Việc phân tích mã nguồn dự án tốn nhiều thời gian và có thể cần phân tích nhiều lần do quá trình phát triển song song với quá trình kiểm thử.

- Quá trình tự động xử lý các kiểu dữ liệu và các hàm ở mô-đun khác sao cho biên dịch được mã nguồn có thể không chính xác. Khi kiểm thử, ta muốn hạn chế sự tác động đến mã nguồn gốc. Do vậy, các kiểu dữ liệu và hàm ở mô-đun khác thường được sao chép ở một tệp và mô-đun kiểm thử sẽ sử dụng tệp sao chép này. Khi tự động hóa quá trình này, công cụ cần phân tích mã nguồn để biết chính xác nên sao chép kiểu nào, hàm nào và vị trí ở đâu.

3.4. Đánh giá khả năng sinh dữ liệu kiểm thử tự động

3.4.1. Cách thức tổ chức thực nghiệm

Nhằm đánh giá tính hiệu quả khi sinh dữ liệu kiểm thử tự động, khóa luận tiến hành thực nghiệm trên công cụ AKAUTAUTO phiên bản 5.9.2, được cài đặt phương pháp kiểm thử tượng trưng động AKAUTAUTO và phương pháp AS4UT, và phiên bản 5.9.2-thesis, được cài đặt phương pháp đề xuất. Các mã nguồn được sử dụng trong thực nghiệm này gồm mã nguồn C-plus-plus, mã nguồn Box2d và mã nguồn FPT1. Khóa luận chưa thể đánh giá khả năng sinh dữ liệu kiểm thử tự động trên mã nguồn FPT2 do mã nguồn này được bảo mật bởi đơn vị FPT-GAM.

3.4.2. Kết quả thực nghiệm

Bảng 3.2 trình bày kết quả thực nghiệm sinh dữ liệu kiểm thử tự động cho một số tệp trong các mã nguồn C-plus-plus, Box2d và FPT1 giữa phương pháp AKAUTAUTO và phương pháp đề xuất. Trong đó, bốn tệp đầu được lấy từ dự án C-plus-plus, ba tệp tiếp theo được lấy từ dự án Box2d và năm tệp còn lại từ mô-đun FPT1. Các cột trong bảng có ý nghĩa như sau:

- "File": Tên tệp mã nguồn đuôi .cpp được kiểm thử.
- "Unit": Số lượng hàm trong tệp mã nguồn được kiểm thử.
- "C1": Độ phủ C_1 của tệp mã nguồn.
- "C3": Độ phủ C_3 của tệp mã nguồn.
- "Num": Số lượng ca kiểm thử cần thiết để đạt được độ phủ tương ứng.
- "Mem": Bộ nhớ sử dụng trung bình (KB) khi sinh dữ liệu kiểm thử tự động cho một hàm trong tệp.
- "Time": Thời gian sinh trung bình (giây) khi sinh dữ liệu kiểm thử tự động cho một hàm trong tệp.

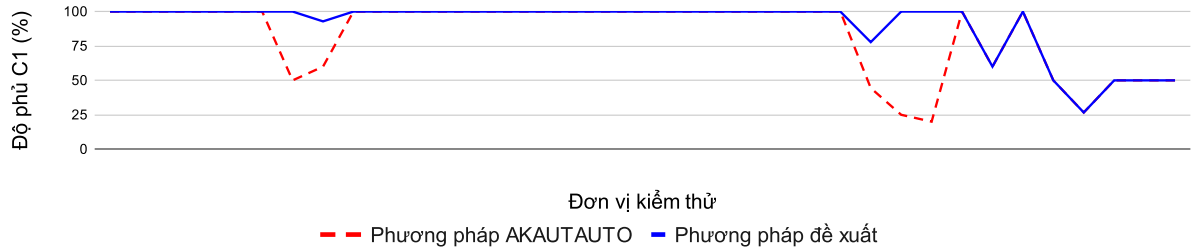
Bảng 3.2: Kết quả thực nghiệm sinh dữ liệu kiểm thử tự động cho một số mã nguồn

File	Unit	AKAUTAUTO Current Method					Proposed Method				
		C1	C3	Num	Mem	Time	C1	C3	Num	Mem	Time
avltree.cpp	10	91	80.55	38	1856.31	4.42	**99.29**	**100**	25	1299.69	4.73
binary_search_tree.cpp	9	100	100	24	1048.37	5.61	**100**	**100**	42	870.71	10.90
binaryheap.cpp	11	77.22	24.16	47	357.88	2.51	**94.34**	**73.75**	46	238.66	8.32
double_linked_list.cpp	6	54.44	31.66	15	3269.37	4.43	**54.44**	**31.66**	34	405.41	14.94
b2_dynamic_tree.cpp	12	59.06	45.12	56	1101.79	175.82	**72.13**	**64.32**	37	295.80	102.15
b2_board_phase.cpp	7	49.87	26.85	45	1611.60	206.98	**83.81**	**68.75**	31	5105.65	83.35
b2_collide_edge.cpp	3	51.85	44	15	2611.99	91.44	**76.44**	**67.13**	32	6020.06	263.73
FPT1_file1.cpp	185	89.26	60.67	668	442.53	268.36	**98.43**	**96.05**	703	188.49	46.28
FPT1_file2.cpp	17	46.40	29.29	40	1353.71	54.60	**95.41**	**86.55**	75	23.79	63.33
FPT1_file3.cpp	1	66.67	16.67	7	21.34	72.75	**100**	**100**	7	15.20	54.20
FPT1_file4.cpp	4	63.07	16.67	13	108.48	278.04	**63.07**	**16.67**	10	6.68	21.35
FPT1_file5.cpp	2	92.86	50	4	23.34	10.91	**92.86**	**50**	4	4.37	9.36

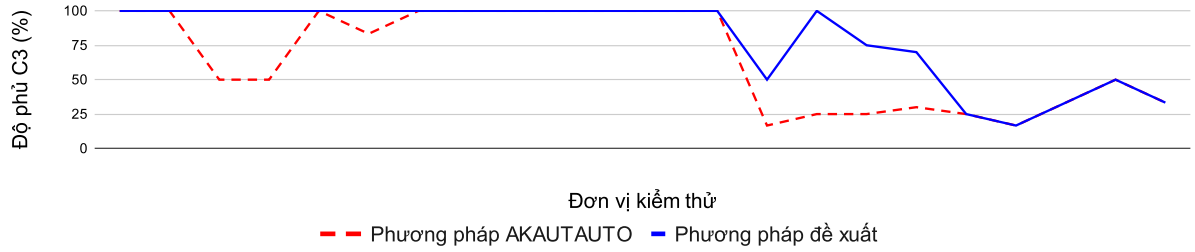
Kết quả thực nghiệm ở Bảng 3.2 cho thấy phương pháp đề xuất sinh dữ liệu kiểm thử có độ phủ C_1, C_3 luôn cao hơn hoặc bằng phương pháp AKAUTAUTO trên các tệp mã nguồn được kiểm thử. Trong đó, phương pháp đề xuất đạt độ phủ cao hơn khi kiểm thử cho 2/4 tệp của dự án C-plus-plus, 3/3 tệp của dự án Box2d và 3/5 tệp của dự án FPT1. So sánh độ phủ trung bình giữa hai phương pháp trên các tệp của từng dự án, phương pháp đề xuất có độ phủ trung bình cao hơn và tăng đáng kể khi xét độ phủ C_3. Cụ thể, với bốn tệp thuộc dự án C-plus-plus, độ phủ C_1, C_3 trung bình tăng lần lượt 6.35% và 17.26%. Ba tệp của dự án Box2d có độ phủ C_1, C_3 trung bình tăng lần lượt 23.87% và 28.08%. Năm tệp của dự án FPT1 tăng lần lượt 18.30% và 35.19% với các độ đo tương ứng.

Về mặt độ phủ C_1, C_3 đạt được trên từng hàm kiểm thử, kết quả trên ba dự án đều cho thấy phương pháp đề xuất đạt được độ phủ cao hơn hoặc bằng phương pháp AKAUTAUTO. Trước hết, với dự án C-plus-plus, các Hình 3.6, 3.7 cho thấy độ phủ C_1, C_3 của phương pháp đề xuất đạt 100% cho hầu hết các hàm trong hai tệp đầu, tăng mạnh độ phủ C_3 cho các hàm trong tệp `binary_search_tree.cpp` và giữ nguyên đối với các hàm trong tệp còn lại. Trên dự án Box2d, Hình 3.8 và 3.9 cho thấy phương pháp đề xuất có tăng đáng kể độ phủ C_1, C_3 trên 10/22 đơn vị. Tuy nhiên, phương pháp đề xuất không cải thiện độ phủ trên 12 đơn vị còn lại. Cuối cùng, với dự án FPT1, các Hình 3.10, 3.11 cho thấy phương pháp đề xuất tăng độ phủ C_1, C_3 với phần lớn các đơn vị, trong đó 95/209 đơn vị với độ phủ C_1 và 173/209 đơn vị với độ phủ C_3. Kết quả chi tiết độ phủ

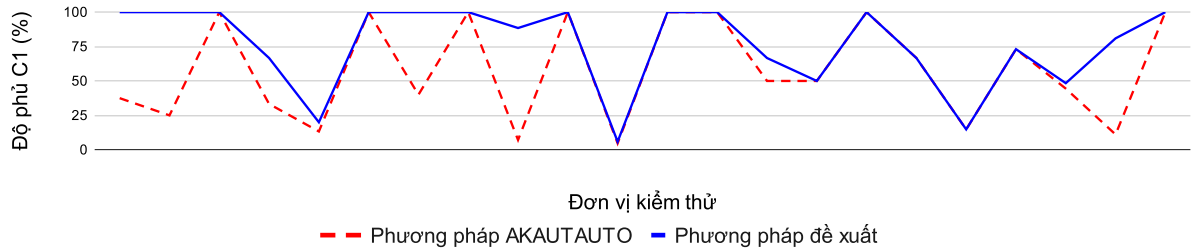
trên từng đơn vị kiểm thử cho thấy rằng đối với các đơn vị đã đạt 100% bởi phương pháp AKAUTAUTO, phương pháp đề xuất cũng đạt 100% với các đơn vị đó. Tuy nhiên, kết quả độ phủ của phương pháp AKAUTAUTO cho thấy tuy độ phủ C_1 của đơn vị có thể đạt 100% nhưng các nhánh điều kiện trong đơn vị có thể chưa được kiểm tra hết, dẫn tới độ phủ C_3 thấp.



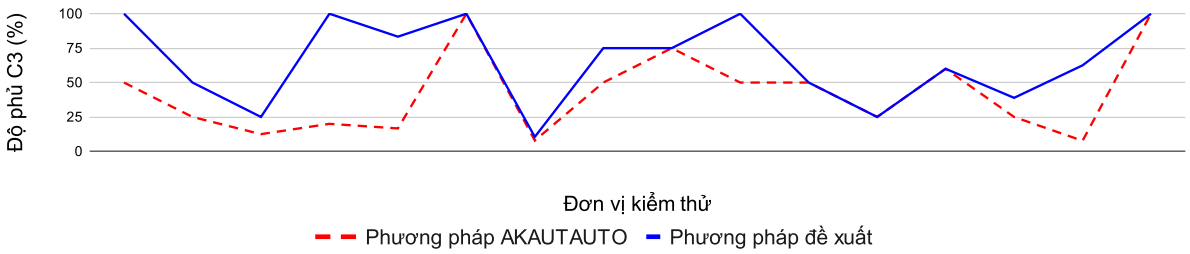
Hình 3.6: So sánh độ phủ C_1 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong C-plus-plus.



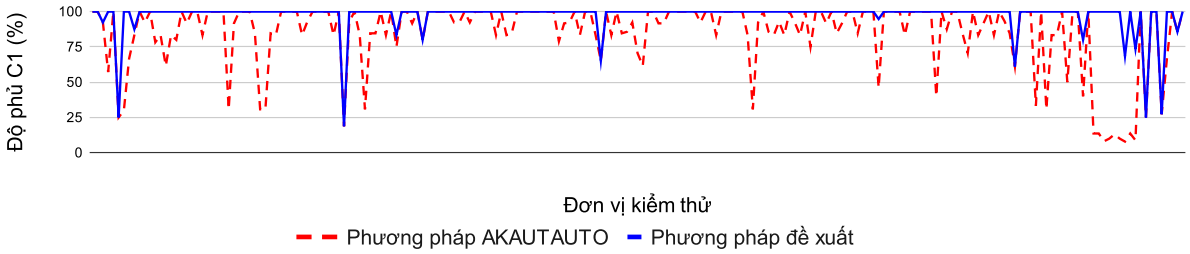
Hình 3.7: So sánh độ phủ C_3 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong C-plus-plus.



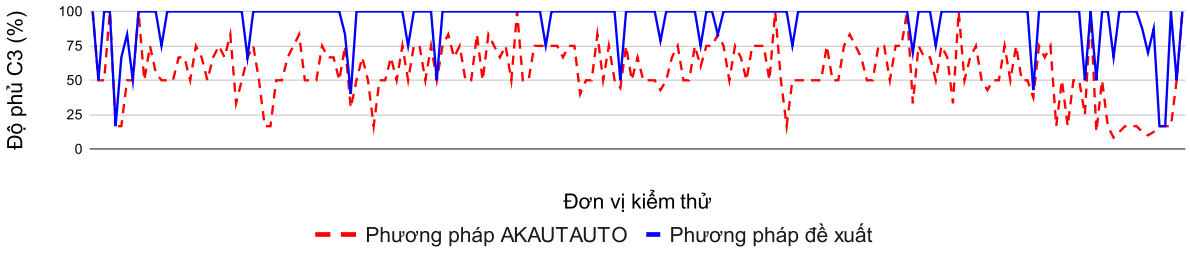
Hình 3.8: So sánh độ phủ C_1 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong Box2d.



Hình 3.9: So sánh độ phủ C_3 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong Box2d.



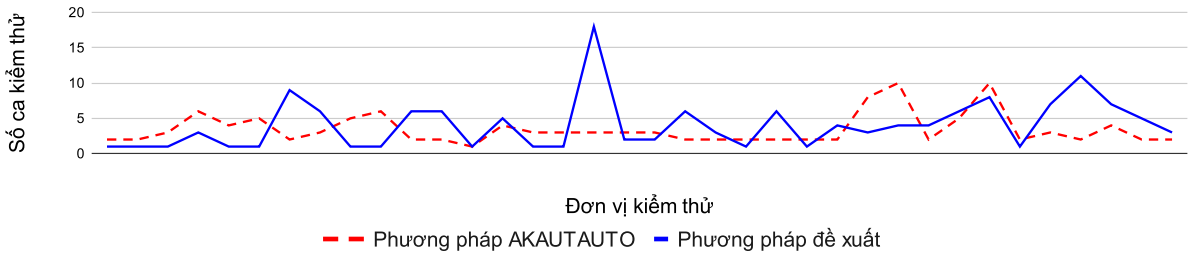
Hình 3.10: So sánh độ phủ C_1 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên các hàm trong mô-đun FPT1.



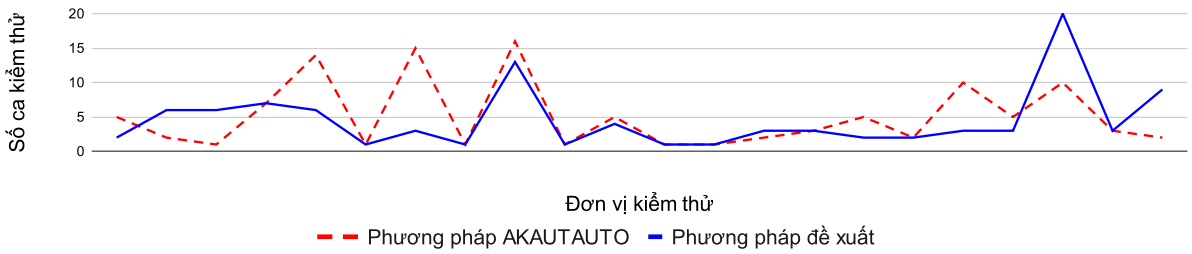
Hình 3.11: So sánh độ phủ C_3 giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên các hàm trong mô-đun FPT1.

Xét số lượng ca kiểm thử tối thiểu sinh để đạt độ phủ tương ứng của hai phương pháp, có thể nhận thấy rằng số ca kiểm thử tối thiểu phụ thuộc vào từng dự án. Đối với dự án C-plus-plus, Hình 3.12 cho thấy rằng với các đơn vị không có quá nhiều sự chênh lệnh về độ phủ, số ca kiểm thử tối thiểu chênh lệch ít. Tuy nhiên, với các đơn vị có sự tăng về độ phủ C_1 hoặc C_3, số ca kiểm thử tăng trong khoảng 2-6 lần. Đặc điểm tương tự diễn ra khi kiểm thử dự án Box2d (mô tả ở Hình 3.13). Một số trường hợp đột biến về sự chênh lệnh số ca kiểm thử chủ yếu bởi phương pháp xử lý lời gọi phương thức hoặc

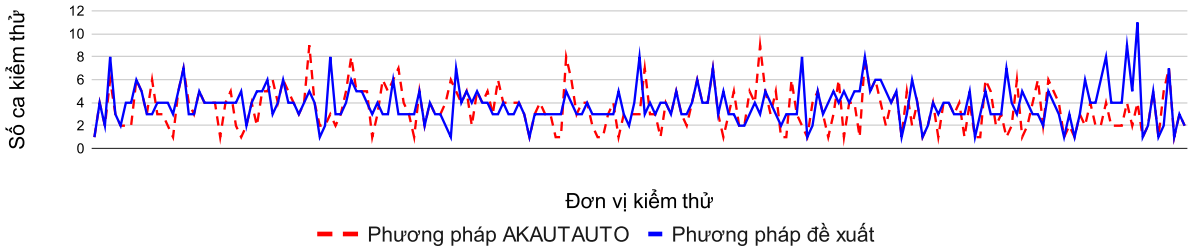
đơn vị kiểm thử chứa nhiều vòng lặp. Với dự án FPT1, do mã nguồn chứa nhiều sự tương tác giữa các đối tượng, Hình 3.14 cho thấy phần lớn các đơn vị (164/209) có số lượng ca kiểm thử tối thiểu của phương pháp đề xuất nhiều hơn phương pháp AKAUTAUTO.



Hình 3.12: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong C-plus-plus.



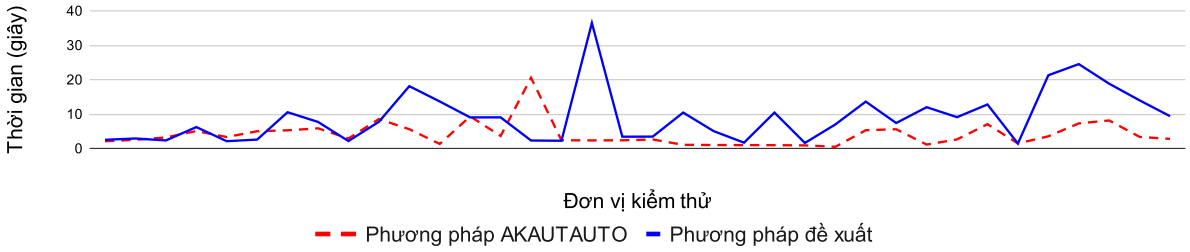
Hình 3.13: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong Box2d.



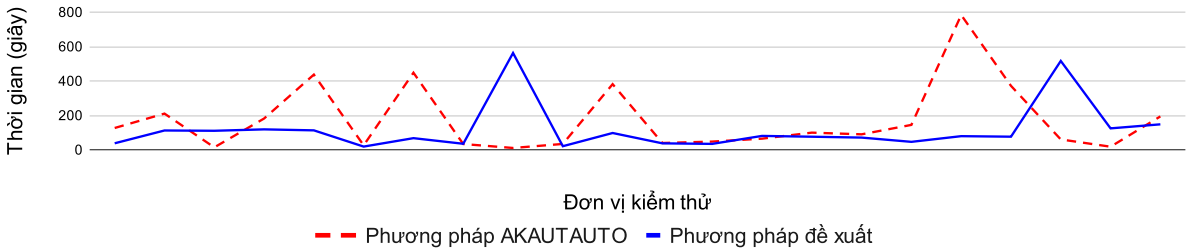
Hình 3.14: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên các hàm trong mô-đun FPT1.

Về thời gian sinh dữ liệu kiểm thử tự động, kết quả cho thấy có 4/4 tệp của dự án C-plus-plus, 1/3 tệp của dự án Box2d và 1/5 tệp của dự án FPT1 mà phương pháp đề xuất sử dụng nhiều thời gian hơn. Ngược lại, phương pháp đề xuất có thời gian sinh ít hơn phương pháp AKAUTAUTO trên các tệp còn lại. Hình 3.15, 3.16 và 3.17 phản ánh thời gian sinh dữ liệu kiểm thử cho các tệp trên từng dự án. Do dự án C-plus-plus không chứa nhiều lời gọi phương thức nên phương pháp đề xuất sử dụng nhiều thời gian hơn

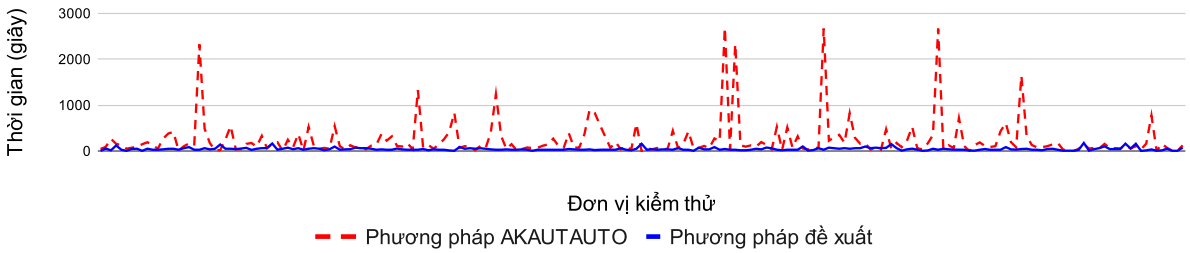
trên phần lớn các đơn vị kiểm thử. Với hai dự án còn lại, có thể nhận thấy rằng đa phần phương pháp đề xuất sử dụng ít thời gian hơn. Do dự án Box2d chứa nhiều vòng lặp nên thời gian sử dụng phụ thuộc vào số lượng ca kiểm thử và số lần lặp.



Hình 3.15: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong C-plus-plus.



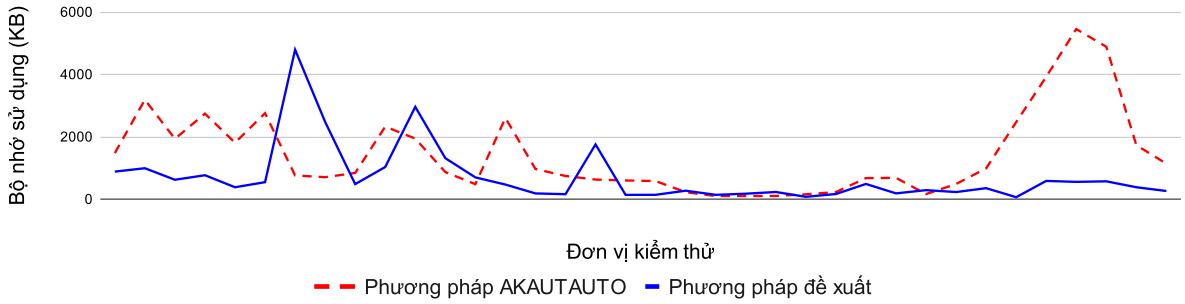
Hình 3.16: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong Box2d.



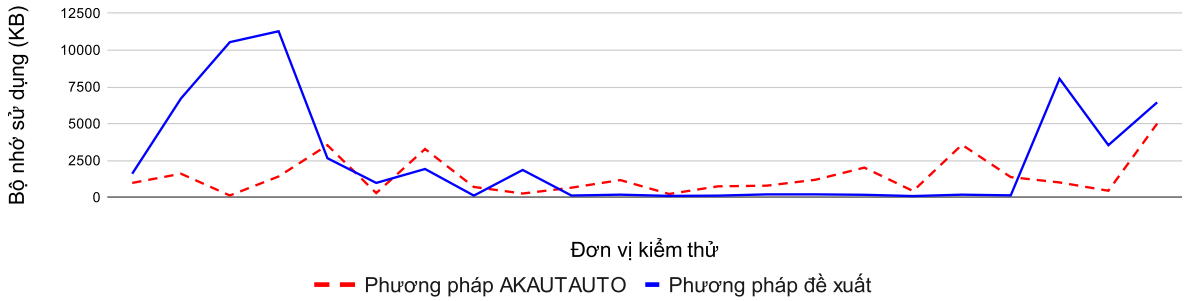
Hình 3.17: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên các hàm trong mô-đun FPT1.

Về bộ nhớ sử dụng để sinh dữ liệu kiểm thử, phương pháp đề xuất sử dụng ít bộ nhớ hơn trên phần lớn các tệp được kiểm thử (10/12). Các trường hợp sử dụng nhiều bộ nhớ hơn là bởi quá trình thực thi tượng trưng xử lý nhiều vòng lặp và các lời gọi phương

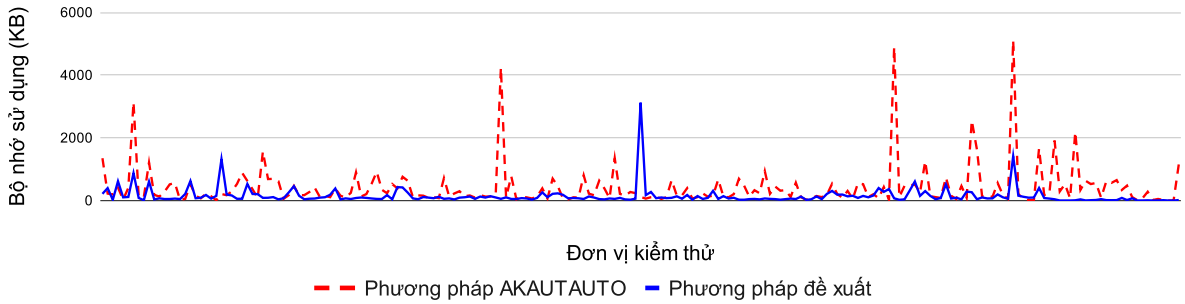
thức. Ngoài các trường hợp trên, không có lý do rõ ràng giải thích cho sự khác biệt trong lượng bộ nhớ sử dụng của hai phương pháp.



Hình 3.18: So sánh bộ nhớ sử dụng ra giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong C-plus-plus.



Hình 3.19: So sánh bộ nhớ sử dụng ra giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên một số hàm trong Box2d.



Hình 3.20: So sánh bộ nhớ sử dụng giữa phương pháp đề xuất và phương pháp AKAUTAUTO trên các hàm trong mô-đun FPT1.

3.4.3. Đánh giá

Kết quả thực nghiệm cho thấy rằng phương pháp đề xuất hiệu quả hơn về mặt độ phủ mã nguồn so với phương pháp AKAUTAUTO khi sinh dữ liệu kiểm thử tự động, đặc biệt ở độ phủ C_3. Một số lí do chính dẫn đến sự cải thiện như sau:

- Phương pháp đề xuất kế thừa phương pháp kiểm thử tượng trưng động nên có cùng khả năng sinh dữ liệu tự động cho các kiểu dữ liệu nguyên thủy và tự định nghĩa với phương pháp AKAUTAUTO. Bởi vậy, các dự án không chứa nhiều lời gọi phương thức như C-plus-plus vẫn có thể áp dụng phương pháp đề xuất để sinh dữ liệu kiểm thử.

- Phương pháp đề xuất kế thừa và cải tiến phương pháp AS4UT nên có khả năng giả lập mã nguồn trả về tự động tương tự như AS4UT. Điều này khiến quá trình thực thi tượng trưng xử lý được các ràng buộc liên quan đến giá trị trả về của lời gọi hàm, giúp tăng độ phủ.

- Độ phủ mã nguồn có sự tăng đột phá bởi cải tiến trong quá trình xử lý lời gọi hàm. Dự án Box2d và FPT1 được viết trên ngôn ngữ C++, sử dụng nhiều đặc trưng hướng đối tượng và các thành phần trong mã nguồn tương tác với nhau thông qua lời gọi phương thức của đối tượng. Do vậy, mã nguồn chứa nhiều biểu thức điều kiện liên quan đến thuộc tính của các đối tượng. Phương pháp đề xuất đã xử lý các lời gọi phương thức nên quá trình thực thi tượng trưng có thể giải được các điều kiện có liên quan đến đối tượng gọi hàm. Phương pháp AS4UT chỉ xử lý cho kết quả trả về của lời gọi hàm nên quá trình thực thi tượng trưng chưa giải được các điều kiện liên quan đến sự thay đổi giá trị của thuộc tính thông qua lời gọi hàm.

Tuy nhiên, có thể nhận thấy rằng một số tệp mã nguồn không tăng kết quả độ phủ so với phương pháp AKAUTAUTO. Điều này được lí giải bởi các nguyên nhân sau:

- Đơn vị kiểm thử chứa nhiều kiểu dữ liệu chưa được hỗ trợ bởi phương pháp đề xuất như con trỏ thông minh, kiểu dữ liệu template, v.v. Phương pháp đề xuất chưa phân tích được các câu lệnh, điều kiện chứa các kiểu dữ liệu này nên không thể sinh ra các ràng buộc thỏa mãn các câu lệnh, điều kiện tương ứng.

- Hạn chế trong việc chuyển đổi các câu lệnh, điều kiện trong mã nguồn trong quá trình thực thi tượng trưng khiến phương pháp không tìm được nghiệm khả thi.
- Đơn vị kiểm thử không chứa lời gọi phương thức nên phương pháp đề xuất không cải thiện được độ phủ. Đóng góp chính của phương pháp đề xuất trong việc sinh dữ liệu kiểm thử tự động nằm ở việc xử lý các lời gọi phương thức. Vì vậy, đối với các đơn vị không chứa lời gọi phương thức, phương pháp đề xuất sẽ cho ra kết quả giống với phương pháp AKAUTAUTO áp dụng AS4UT.
- Phương pháp đề xuất chưa xử lý tốt vòng lặp xuất hiện trong đơn vị kiểm thử. Các dự án C++ thường chứa nhiều câu lệnh duyệt phần tử trong các tập hợp như `vector`, `set`. Các ràng buộc liên quan đến phép duyệt như vậy chưa được hỗ trợ.
- Phương pháp đề xuất chưa xử lý được lời gọi hàm trong thư viện khiến quá trình thực thi tượng trưng không giải được ràng buộc liên quan đến kết quả của lời gọi thư viện.

Về khía cạnh số lương ca kiểm thử, phương pháp đề xuất sinh nhiều ca kiểm thử hơn nếu mã nguồn chứa lời gọi phương thức. Nguyên nhân là do phương pháp đề xuất cần sinh thêm ca kiểm thử để phủ được điều kiện nhánh liên quan đến sự thay đổi của đối tượng sau lời gọi phương thức.

Hai khía cạnh cuối cùng mà khóa luận đánh giá đó là thời gian và bộ nhớ sử dụng để sinh dữ liệu kiểm thử. Hai yếu tố này phụ thuộc nhiều vào cấu trúc và độ phức tạp của mã nguồn. Trong đó, thời gian xử lý bị ảnh hưởng bởi quá trình thực thi tượng trưng, tìm kiếm kiểu dữ liệu hợp lý và quá trình sinh bộ tham số đầu vào cho đơn vị kiểm thử. Với các dự án sử dụng nhiều lời gọi phương thức, phương pháp đề xuất sử dụng ít thời gian ít hơn. Nguyên nhân là bởi phương pháp không mất quá nhiều thời gian để giải một đường đi thi hành chứa lời gọi phương thức. Phương pháp AKAUTAUTO cố gắng tìm nghiệm cho các đường thi hành như vậy nên tiêu thụ nhiều thời gian hơn. Khía cạnh bộ nhớ sử dụng cũng ảnh hưởng bởi các yếu tố tương tự. Tuy nhiên, từ kết quả thực nghiệm, có thể nhận thấy rằng phương pháp đã giải quyết được vấn đề liên quan đến lời gọi phương thức và thể hiện tính khả thi khi áp dụng thực tiễn.

Kết luận

Khóa luận đã mô tả bối cảnh cụ thể về hai hạn chế và sự cần thiết trong việc phát triển một phương pháp kiểm thử hiệu quả khi kiểm thử đơn vị tự động cho mã nguồn C/C++ chứa hàm thiếu định nghĩa. Trong đó hạn chế đầu tiên liên quan đến quá trình chuẩn bị môi trường kiểm thử tự động bởi mã nguồn có thể chứa hàm thiếu định nghĩa gây một số lỗi như lỗi thiếu định nghĩa hàm hoặc lỗi thiếu bảng ký hiệu ảo. Hạn chế này khiến các công cụ kiểm thử tự động không thể chạy các ca kiểm thử trên mã nguồn và tiêu tốn nhiều thời gian để xử lý thủ công. Khóa luận cũng đề cập hạn chế thứ hai liên quan đến việc sinh dữ liệu kiểm thử tự động và sinh stub tự động cho mã nguồn chứa lời gọi phương thức.

Khóa luận đã giới thiệu một phương pháp kiểm thử đơn vị tự động cho mã nguồn C/C++ chứa hàm thiếu định nghĩa để giải quyết hai hạn chế trên. Thứ nhất, khóa luận bổ sung pha xử lý hàm thiếu định nghĩa so với phương pháp truyền thống để giải quyết hạn chế thứ nhất. Việc xử lý hàm thiếu định nghĩa được chia thành hai phần: xử lý nguyên mẫu hàm cơ bản và xử lý nguyên mẫu hàm ảo. Trong đó, khóa luận đã sử dụng các công cụ tiện ích của trình biên dịch để thu thập danh sách nguyên mẫu hàm cơ bản thiếu định nghĩa cần quan tâm, sau đó lọc tìm ứng viên trên đồ thị cấu trúc mã nguồn và sinh thân hàm giả cho chúng. Đối với nguyên mẫu hàm ảo thiếu định nghĩa, khóa luận kết hợp tìm kiếm trên đồ thị cấu trúc mã nguồn và thuật toán lọc tìm ứng viên để xử lý các hàm ảo gây lỗi thiếu bảng ký hiệu ảo. Thứ hai, khóa luận đã cải tiến phương pháp sinh stub tự động bằng phương pháp xử lý lời gọi phương thức. Ý tưởng của phương pháp dựa trên phương pháp AS4UT nhưng bổ sung thêm bước sinh đối tượng giả cho các lời gọi phương thức. Sau đó, phương pháp đề xuất sử dụng phương pháp kiểm thử tượng trưng động để tạo ra các bộ dữ liệu kiểm thử mới.

Khóa luận đã cài đặt và tích hợp phương pháp đề xuất vào phiên bản 5.9.2-thesis của công cụ AKAUTAUTO để tiến hành một số thực nghiệm đánh giá tính hiệu quả của phương pháp đề xuất. Kết quả thực nghiệm cho thấy phương pháp đề xuất có khả năng rút ngắn đáng kể thời gian chuẩn bị môi trường kiểm thử tự động cho mã nguồn thiếu định nghĩa. Kết quả cũng cho thấy rằng phương pháp đề xuất có khả năng sinh dữ liệu kiểm thử đạt độ phủ cao hơn so với phương pháp AKAUTAUTO, đặc biệt trên các dự án có nhiều lời gọi phương thức. Tuy rằng có sự chênh lệch giữa thời gian và bộ nhớ sử dụng để sinh dữ liệu kiểm thử giữa hai phương pháp nhưng thực nghiệm cho thấy sự chênh lệnh này chấp nhận được khi áp dụng thực tế.

Mặc dù phương pháp đề xuất đã đạt được một số kết quả đáng quan tâm như trên, phương pháp đề xuất vẫn còn một số nhược điểm cần hoàn thiện trong tương lai. Trước hết, phương pháp hiện tại chưa hỗ trợ hoàn toàn quy trình bóc tách mô-đun và xử lý hàm thiếu định nghĩa. Để áp dụng được phương pháp đề xuất, khóa luận cần kiểm thử viên bóc tách mô-đun sao cho biên dịch được. Xét về khả năng sinh dữ liệu kiểm thử tự động, phương pháp đề xuất chưa cải thiện được độ phủ khi kiểm thử tự động cho các đơn vị chứa ít lời gọi phương thức. Ngoài ra, phương pháp hiện tại chưa giải quyết một số nhược điểm của phương pháp kiểm thử tượng trưng động. Bên cạnh đó, phương pháp đề xuất cần được áp dụng trên các dự án C/C++ lớn hơn.

Nhìn chung, tuy phương pháp đề xuất còn một số hạn chế, không thể phủ nhận rằng phương pháp đã có kết quả tích cực và bộc lộ tiềm năng áp dụng thực tiễn trong việc kiểm thử đơn vị tự động các dự án C/C++. Phương pháp đề xuất giải quyết được các hạn chế phát sinh bởi mã nguồn chứa hàm thiếu định nghĩa. Qua đó, phương pháp giúp tự động hóa quá trình kiểm thử đơn vị các dự án C/C++ với chi phí tiết kiệm hơn.

Tài liệu tham khảo

Tiếng Việt

[1] Phạm Ngọc Hùng, Trương Anh Hoàng, and Đặng Văn Hưng (2014). *Giáo trình Kiểm thử phần mềm*. NXB Đại học Quốc gia Hà Nội.

Tiếng Anh

[2] Alabbas Alhaj Ali (2017). "ISO 26262 functional safety standard and the impact in software lifecycle". In: *J. Univ. Appl. Sci*.

[3] Graham Buckle (1998). "Static Analysis of Safety Critical Software (Techniques, Tools, and Experiences)". In: *Industrial Perspectives of Safety–critical Systems*. Ed. by Felix Redmill and Tom Anderson. London: Springer London, pp. 150–168. ISBN: 978–1–4471–1534–2.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler (2008). "KLEE: Unassisted and Automatic Generation of High–Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, pp. 209–224.

[5] Cristian Cadar and Koushik Sen (Feb. 2013). "Symbolic Execution for Software Testing: Three Decades Later". In: *Commun. ACM* 56.2, pp. 82–90. ISSN: 0001–0782. DOI: 10.1145/2408776.2408795.

[6] Cristian Cadar et al. (Dec. 2008). "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2. ISSN: 1094–9224. DOI: 10.1145/1455518.1455522.

[7] Brian Chess and Jacob West (2007). *Secure Programming with Static Analysis*. First. Addison–Wesley Professional. ISBN: 9780321424778.

[8] Bernd Gassmann et al. (2019). "Towards standardization of av safety: C++ library for responsibility sensitive safety". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, pp. 2265–2271.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen (2005). "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036.

[10] Elena Grigorenko and Robert Sternberg (July 1998). "Dynamic Testing". In: *Psychological Bulletin* 124, pp. 75–111. DOI: 10.1037/0033-2909.124.1.75.

[11] Martin Hillenbrand (2012). "Funktionale Sicherheit nach ISO 26262 in der Konzeptphase der Entwicklung von Elektrik/Elektronik Architekturen von Fahrzeugen". German. PhD thesis. 398 pp. ISBN: 978-3-86644-803-2. DOI: 10.5445/KSP/1000025616.

[12] Zhenmai Hu Jr (2021). "A Software Package for Generating Code Coverage Reports With Gcov". In.

[13] Tran Nguyen Huong et al. (2022). "An Automated Stub Method for Unit Testing C/C++ Projects". In: *2022 14th International Conference on Knowledge and Systems Engineering (KSE)*, pp. 1–6. DOI: 10.1109/KSE56063.2022.9953784.

[14] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan (2011). "KLOVER: A symbolic execution and automatic test generation tool for C++ programs". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, pp. 609–615.

[15] Duc–Anh Nguyen et al. (2019). "Improvements of Directed Automated Random Testing in Test Data Generation for C++ Projects". In: *International Journal of Software Engineering and Knowledge Engineering* 29.09, pp. 1279–1312. DOI: 10.1142/S0218194019500402.

[16] Philip J Plauger (1997). "Embedded C++: An Overview". In: *Embedded Systems Programming* 10, pp. 40–53.

[17] Koushik Sen (2007). "Concolic Testing". In: *Proceedings of the Twenty–Second IEEE/ACM International Conference on Automated Software Engineering*. ASE

'07. Atlanta, Georgia, USA: Association for Computing Machinery, pp. 571–572. ISBN: 9781595938824. DOI: 10.1145/1321631.1321746.

[18] Koushik Sen, Darko Marinov, and Gul Agha (2005). "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE–13. Lisbon, Portugal: Association for Computing Machinery, pp. 263–272. ISBN: 1595930140. DOI: 10.1145/1081706.1081750.

[19] T. Sun et al. (2009). "Towards Scalable Compositional Test Generation". In: *2009 Ninth International Conference on Quality Software*, pp. 353–358. DOI: 10.1109/QSIC.2009.53.

[20] Edward P. K. Tsang (1993). *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press. ISBN: 978–0–12–701610–8.

[21] Lam Nguyen Tung et al. (2022). "An automated test data generation method for void pointers and function pointers in C/C++ libraries and embedded projects". In: *Information and Software Technology* 145, p. 106821. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2022.106821. URL: https://www.sciencedirect.com/science/article/pii/S0950584922000027.

[22] Nicky Williams et al. (2005). "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In: *Dependable Computing - EDCC 5*. Ed. by Mario Dal Cin, Mohamed Kaâniche, and András Pataricza. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 281–292. ISBN: 978–3–540–32019–7.