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 "Một số cải tiến phương pháp sinh dữ liệu Kiểm thử cho các dự án C/C++ dựa trên phân tích mã nguồn" 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 - ĐHQG Hà Nội.

Hà Nội, ngày 31 tháng 05 năm 2021

Sinh viên

Nguyễn Tùng Lâm

LỜI CẢM ƠN

Lời đầu tiên cho phép em được gửi lời cảm ơn tới Khoa Công Nghệ Thông Tin – Trường Đại học Công nghệ - ĐHQG Hà Nội đã tạo điều kiện thuận lợi cho em được học tập, nghiên cứu và thực hiện đề tài tốt nghiệp này.

Em cũng xin được bày tỏ lòng biết ơn sâu sắc tới thầy Phạm Ngọc Hùng và thầy Võ Đình Hiếu đã tận tình hướng dẫn, đóng góp những ý kiến xác đáng để em có thể hoàn thành khóa luận một cách tốt nhất.

Em cũng vô cùng biết ơn những thầy cô trong trường đã tận tình giảng dạy, trang bị cho em những kiến thức quan trọng để em có hành trang vững chắc cho con đường học vấn của mình.

Cuối cùng, em xin được gửi lời cảm ơn chân thành tới anh Nguyễn Đức Anh, anh Trần Hoàng Việt 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.

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

TÓM TẮT

Tóm tắt: Kiểm thử đơn vị cho các dự án C/C++, đặc biệt là các thư viện và dự án nhúng, đã và đang trở thành một bài toán lớn. Nhằm nâng cao độ tin cậy của mã nguồn, các phương pháp kiểm thử phần mềm tự động đang nhận được đông đảo sự quan tâm và đang được áp dụng rộng rãi trong cả cộng đồng nghiên cứu lẫn các công ty phần mềm. 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 vấn đề nảy sinh trong quá trình sinh dữ liệu kiểm thử cho con trỏ void và con trỏ hàm. Khóa luận này đề xuất một phương pháp để tìm kiếm một tập các dữ liệu phù hợp cho các tham số con trỏ nhằm giải quyết các hạn chế kể trên. Phương pháp sinh dữ liệu kiểm thử cho con trỏ void và con trỏ hàm giúp tăng độ bao phủ kiểm thử và nâng cao độ tin cậy của mã nguồn. Ý tưởng chính của phương pháp đề xuất là tìm kiếm tất cả các kiểu phù hợp cho con trỏ void và các tham chiếu ứng viên của con trỏ hàm. Quá trình trên được thực hiện bằng cách phân tích mã nguồn trong hai phạm vi: nội bộ và ngoại bộ đơn vị được kiểm thử. Các kiểu và tham chiếu trên được sử dụng để hỗ trợ sinh các dữ liệu khởi tạo theo hướng tiếp cận của kiểm thử tượng trưng động. Phương pháp đề xuất đã được cài đặt thành một công cụ hỗ trợ kiểm thử cho nhiều thư viện và dự án nhúng viết bằng C/C++. Kết quả thực nghiệm cho thấy rằng phương pháp có khả năng sinh ra một lượng ít hơn dữ liệu kiểm thử nhưng vẫn đảm bảo nâng cao độ phủ mã nguồn đạt được so với các phương pháp truyền thống. Phương pháp đề xuất đã được 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 FGA (FPT Global Automative) để kiểm thử tự động các phần mềm điều khiển xe ô tô. Những kết quả tích cực của phương pháp đề xuất cho thấy tiềm năng ứng dụng thực tế để kiểm thử các thư viện và dự án nhúng viết bằng C/C++ có kích thước lớn.

Từ khóa: *kiểm thử tự động, kiểm thử tượng trưng động, thực thi giá trị tượng trưng, kiểm thử dòng điều khiển, con trỏ void, con trỏ hàm, bộ giải SMT*

ABSTRACT

Abstract: Unit testing for C/C++ projects, especially libraries and embedded projects has been known as a complex problem. In order to improve the software reliability, automated software testing is widely applied in both research and industrial communities. Automated software testing includes two main approaches called static testing and concolic testing. Inheriting the above two approaches, concolic testing is proposed with several improvements and certain achievements. However, while applying these approaches, it is noticed that there are some issues in generating test data for void pointers and function pointers. This thesis proposed a method to generate test data for void pointers and function pointers in order to improve the testing coverage and the software reliability. The key idea of the proposed method applying concolic testing is to analyze the given source code to find all possible types for void pointers and references for function pointers. The analysis is performed on source code both inside and outside of the unit under test. These types and references are used to generate the initial test data for concolic testing method. The proposed method is implemented in a support tool to test on various C/C++ libraries and embedded projects. The experimental results show that the method significantly improves the coverage of the generated test data. The proposed method is capable of employing a limited number of test data but remaining highly competitive in comparison with other existing methods. The proposed method has been applied into the AKAUTAUTO - a research collaboration product between Software Quality Assurance Laboratory (Faculty of Information Technology, University of Engineering and Technology, VNU) and FGA (FPT Global Automative) for automated testing for automotive software. The design of the proposed method allows various C/C+ libraries and embedded projects to be tested on a large scale.

Keywords: *Automated test data generation, concolic testing, symbolic execution, control flow testing, void pointers, function pointers, SMT Solver*

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 đầy đủ**	
AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
CFG	Control Flow Graph	Đồ thị dòng điều khiển
CFT	Control Flow Testing	Kiểm thử dòng điều khiển
SE	Symbolic Execution	Thực thi giá trị tượng trưng
SMT	Satisfiability Modulo Theories	Lý thuyết mô-đun thỏa mãn
Static testing		Kiểm thử tĩnh
Dynamic testing		Kiểm thử động
Concolic testing		Kiểm thử tượng trưng động

Chương 1

Đặt vấn đề

Chương 2

Kiến thức cơ sở

2.1 Kiểm thử tượng trưng động

2.2 Sinh giả lập mã nguồn tự động

2.3 Đồ thị dòng điều khiển

2.4 Độ phủ mã nguồn

2.5 Tập ràng buộc và bộ giải hệ ràng buộc Z3

2.6 Tổng quan về hàm thiếu định nghĩa

Chương 3

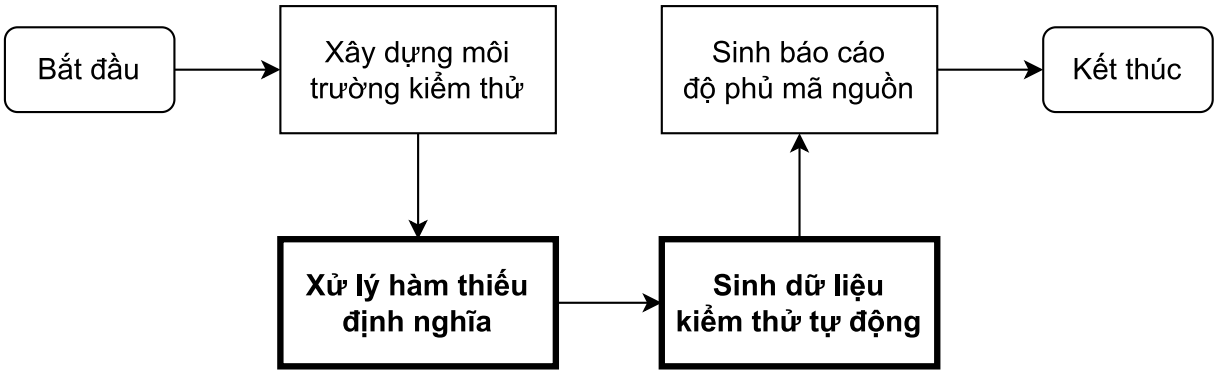
Phương pháp kiểm thử đơn vị tự động cho mã nguồn chứa hàm thiếu định nghĩa

3.1 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 [33] và phương pháp sinh giả lập mã nguồn tự động [54] 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 3.1. 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 3.2.

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 3.3.



Hình 3.1: 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 2.2). 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 3.4.

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 GCOV và công cụ sinh báo cáo LCOV để thực hiện nhiệm vụ của pha. Trong đó, GCOV là công cụ phân tích độ phủ tích hợp sẵn trong gói của trình biên dịch C/C++ GNU, LCOV là một tiện ích mở rộng từ công cụ GCOV với nhiệm vụ sinh báo cáo độ phủ dạng HTML dựa trên các thông tin độ phủ của GCOV. GCOV và LCOV là hai công cụ có sẵn trong trình biên dịch và được sử dụng phổ biến trong các dự án C/C++ [52]. 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.

3.2 Pha xây dựng môi trường kiểm thử

Như đã đề cập ở Mục 3.1, 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 (instrumentation) 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 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. 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.

- 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.

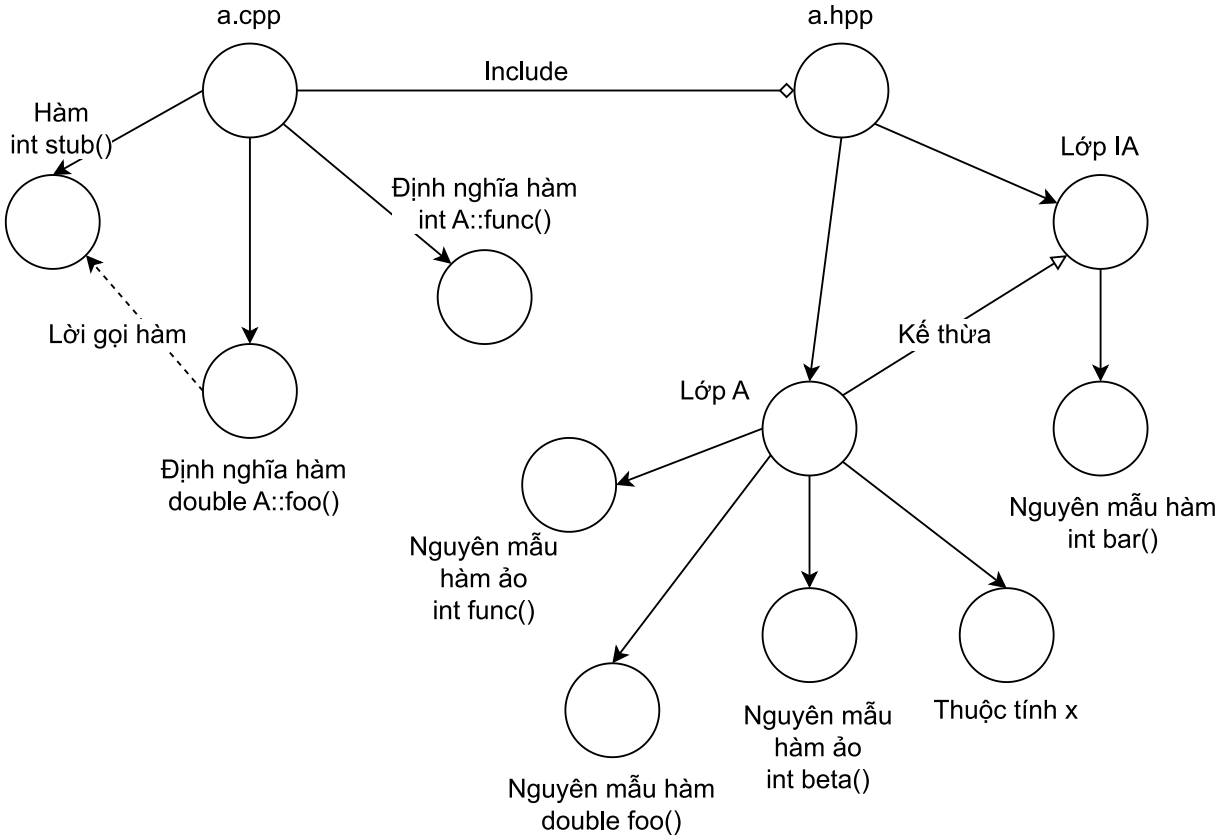
Ví dụ về đồ thị cấu trúc mã nguồn

```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 3.1: 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 3.2: Mã nguồn tệp *a.hpp* minh họa đồ thị cấu trúc mã nguồn.



Hình 3.2: Ví dụ minh hoạ về đồ thị cấu trúc mã nguồn xây dựng trên mã nguồn tệp *a.hpp* và *a.cpp*.

Hình 3.2 minh hoạ đồ thị cấu trúc mã nguồn của hai tệp mã nguồn a.hpp và a.cpp trong Đoạn mã 3.1 và 3.2. 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 3.1. 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.

3.3 Pha 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 các nguyên mẫu hàm thiếu định nghĩa tìm được bởi trình biên dịch và xử lý các 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 3.3.1 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 3.3.2.

3.3.1 Phương pháp xử lý nguyên mẫu hàm 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 3.1 mô tả chi tiết quá trình tìm kiếm và xử lý các nguyên mẫu hàm thiếu định nghĩa. Trước hết, thuật toán bắt đầu bằng việc thu thập danh sách các nguyên mẫu hàm 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 3.2 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 3.1: 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 3.2 (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 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 2.6. Đoạn mã 3.3 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*[1] với các cờ *-u* - trích xuất các hàm thiếu định nghĩa, *-C* -

[1]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 3.4: 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 3.5: 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 3.3: Các câu lệnh tìm nguyên mẫu hàm thiếu định nghĩa cần quan tâm sử dụng trình biên dịch.

Đoạn mã 3.4 và 3.5 minh họa cách áp dụng phương pháp đề xuất trên tập đơn vị kiểm thử cần xử lý các nguyên mẫu hàm thiếu định nghĩa với bốn tệp mã nguồn. Kiểm thử viên muốn kiểm thử tự động cho hai hàm có lôgic đầy đủ `foo` và `gamma`. Hai hàm này

```
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 3.6: Danh sách các nguyên mẫu hàm thiếu định nghĩa tìm bởi trình biên dịch.

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 3.4 nhận đầu vào là một biến con trỏ đến đối tượng của lớp `A` và một biến kiểu nguyên `x`. Lôgic 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`.

Đoạn mã 3.6 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 3.5). 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 3.5).

Lọc tìm ứng viên hợp lệ

Như đã mô tả trong ví dụ minh hoạ ở Đoạn mã 3.6, mỗi nguyên mẫu hàm trong danh sách trả về bởi trình biên dịch sẽ tương ứng với một số nguyên mẫu hàm trong mã nguồn gốc. Để ánh xạ từng nguyên mẫu hàm cần quan tâm tới đúng nguyên mẫu hàm 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 3.2 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++[23] 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.

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

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

Thuật toán 3.2: Thuật toán lọc tìm ứng viên hợp lệ

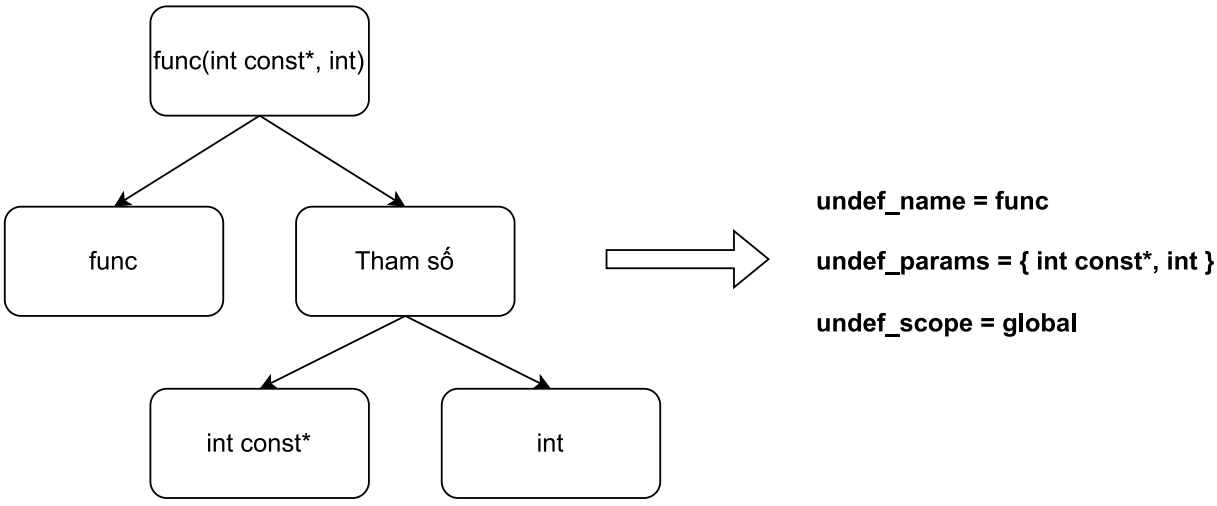
Input: *undef_params* - list of parameters extracted from the string *undef*
undef_scope - the scope qualifier of *undef*
candidates - list of function prototypes having the same name

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 hơn cách hoạt động của Thuật toán 3.2, ta xét ví dụ áp dụng Thuật toán 3.1 trên danh sách các hàm trả về ở Đoạn mã 3.6. 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 3.3. 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 3.5, 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 3.2 với hai ứng viên cùng các thông tin trích xuất được. Bảng 3.1 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.

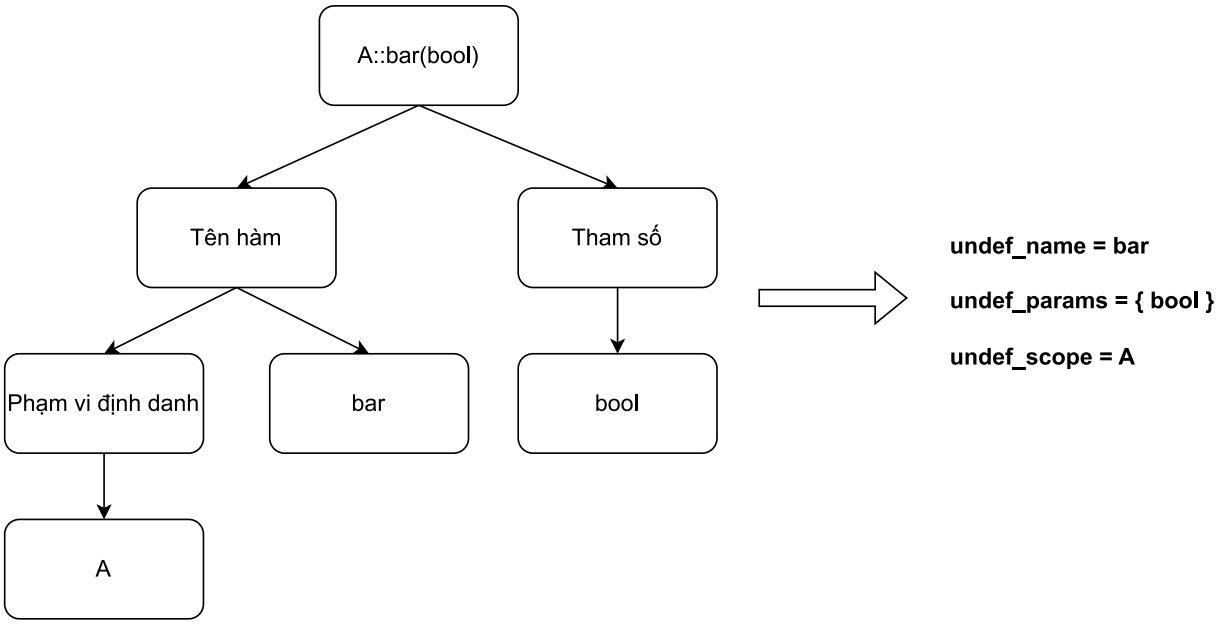


Hình 3.3: AST (bên trái) và các thông tin trích xuất được (bên phải) của nguyên mẫu hàm `func(int const*, int)` trong Đoạn mã 3.6.

Bảng 3.1: 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 3.4. 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 3.2 và được minh hoạ qua Bảng 3.2. Như vậy, khi xét ứng viên `bar(bool b)`, Thuật toán 3.2 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 3.7. 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 3.4: AST (bên trái) và các thông tin trích xuất được (bên phải) của nguyên mẫu hàm `A::bar(bool)` trong Đoạn mã 3.6.

3.3.2 Phương pháp xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Như đã đề cập ở Mục 2.6, 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 3.3.1 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 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 3.3. 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

Bảng 3.2: 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

hợp T các cây cấu trúc có đỉnh tệp có cạnh Include tới đỉnh tệp của $file\_tree$ (dòng 2-4). 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 3.2 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

Đoạn mã 3.1 và 3.2 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 3.3 để xử lý hai nguyên mẫu hàm ảo trên. Trước hết, thuật toán xét cây cấu trúc tệp có đỉnh tệp *a.hpp*

```
// 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 3.7: Mã nguồn tệp a.hpp, b.hpp và c.hpp sau khi áp dụng phương pháp xử lý nguyên mẫu hàm thiếu định nghĩa.

(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 3.2, 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 3.2 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

Thuật toán 3.3: Thuật toán xử lý nguyên mẫu hàm ảo thiếu định nghĩa

Input: *project_graph* - the project structure graph

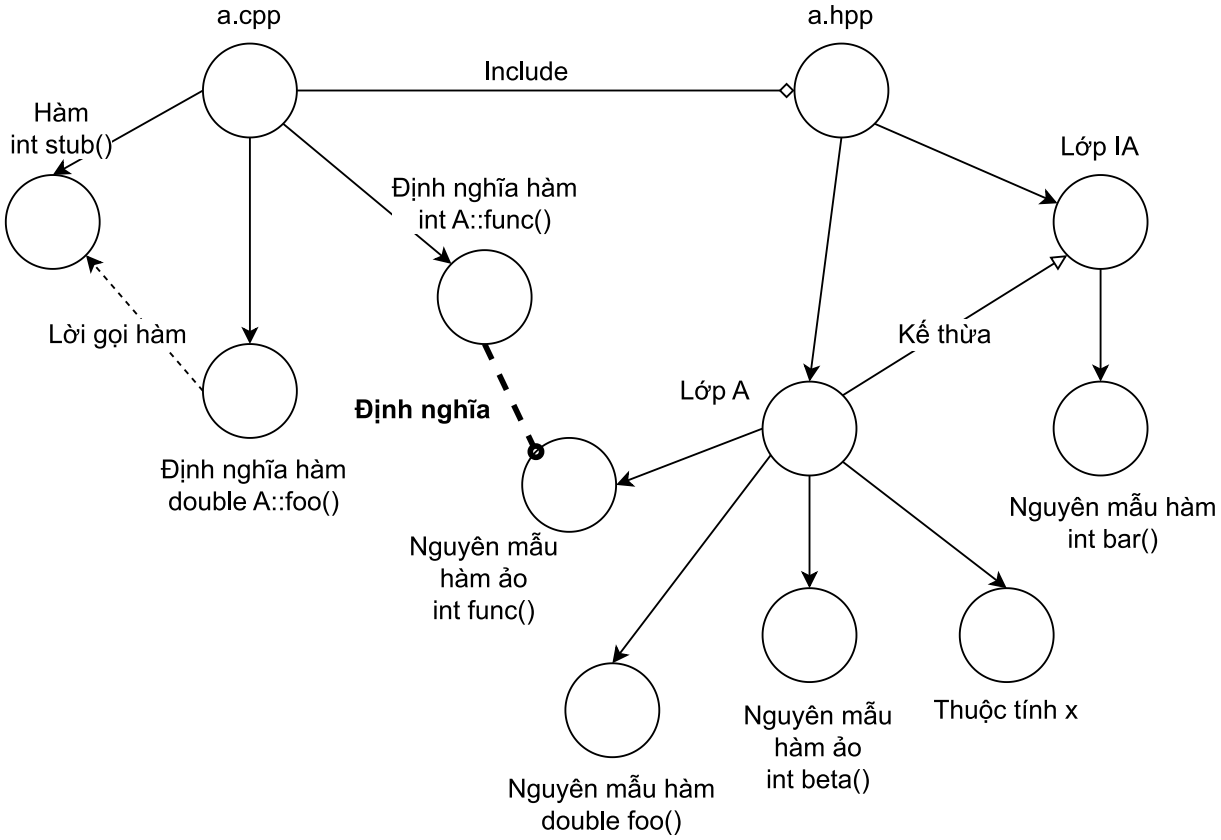
Output: *project_graph* - the modified graph with new definition edges

```
for file_tree : project_graph do
    cur_file_node ← Get file node of file_tree;
    T ← file_tree;
    T ← T ∪ Get all file struct tree of file nodes that include cur_file_node;
    for undef_virtual : file_tree do
        undef_name ← Get the name of the function from undef_virtual;
        candidates ← Find all definition functions having the same name with
          undef_name in T;
        undef_params ← Extract a list of parameters from undef_virtual;
        undef_scope ← Extract function scope qualifier from undef_virtual;
        qualified_cans ← call Algorithm 3.2
          (undef_params, undef_scope, candidates);
        if qualified_cans = ∅ then
            Generate a simple function body for undef_virtual;
        else
            qc ← Get the first qualified candidate;
            Generate definition edge for qc and undef_virtual on project_graph;
        end
    end
end
return project_graph
```

mẫu hàm thiếu định nghĩa ở Mục 3.3.1 Thuật toán tiếp tục xử lý tương tự với cây cấu trúc *a.cpp*. Hình 3.5 mô tả sự thay đổi của đồ thị cấu trúc mã nguồn sau quá trình xử lý. Cạnh đậm, nét đứt, đầu hình tròn thể hiện cạnh định nghĩa được bổ sung sau quá trình.

3.4 Pha 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

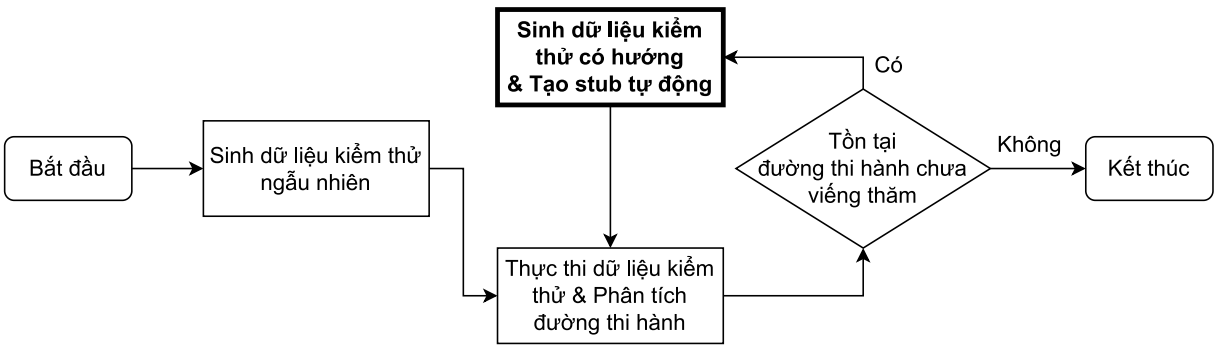


Hình 3.5: Ví dụ minh hoạ sự thay đổi đồ thị cấu trúc mã nguồn *a.hpp* và *a.cpp* sau khi xử lý hàm thiếu định nghĩa.

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 [23, 25, 54] và phương pháp sinh giả lập mã nguồn (stub) tự động AS4UT [53] 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.

3.4.1 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 3.6. 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 [54]. Trong đó, bước sinh dữ liệu kiểm thử có hướng được bổ sung phương pháp sinh stub tự động AS4UT và phương pháp xử lý lời gọi phương thức của đối tượng do khóa luận đề xuất. 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 [53] được tích hợp trong quá trình thực thi tượng trưng nhằm xử lý các lời gọi hàm trong đơn vị kiểm thử. Cụ thể,



Hình 3.6: Tổng quan quá trình sinh dữ liệu kiểm thử tự động.

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 3.4.2 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 hệ 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 được lặp lại cho đến khi không thể tìm thấy đường thi hành chưa thử viếng thăm.

3.4.2 Phương pháp 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 có sự thay đổi 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 3.4 để xử lý lời gọi phương thức. Ý tưởng chính của thuật toán đề xuất 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 3.4 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 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.

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

Đoạn mã 3.8 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 3.7. 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. Hình 3.8 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 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

Thuật toán 3.4: 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

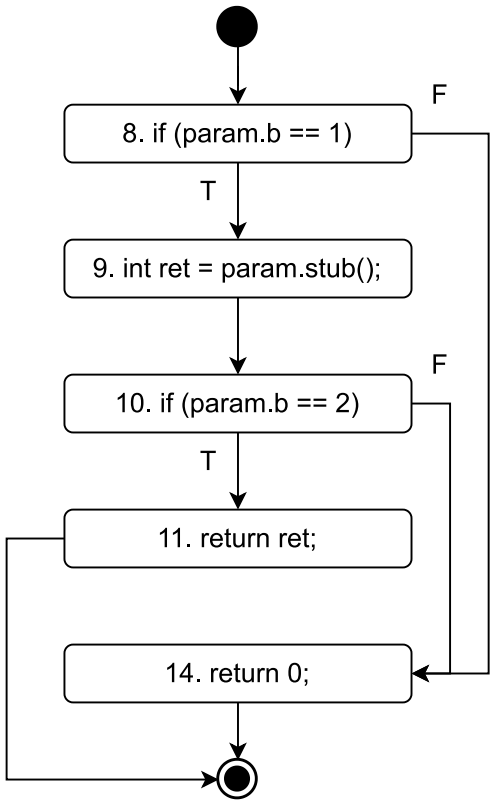
Output: MAP - the updated map table
REF - the updated reference table
stm - the refactored statement

$function \leftarrow$ 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)

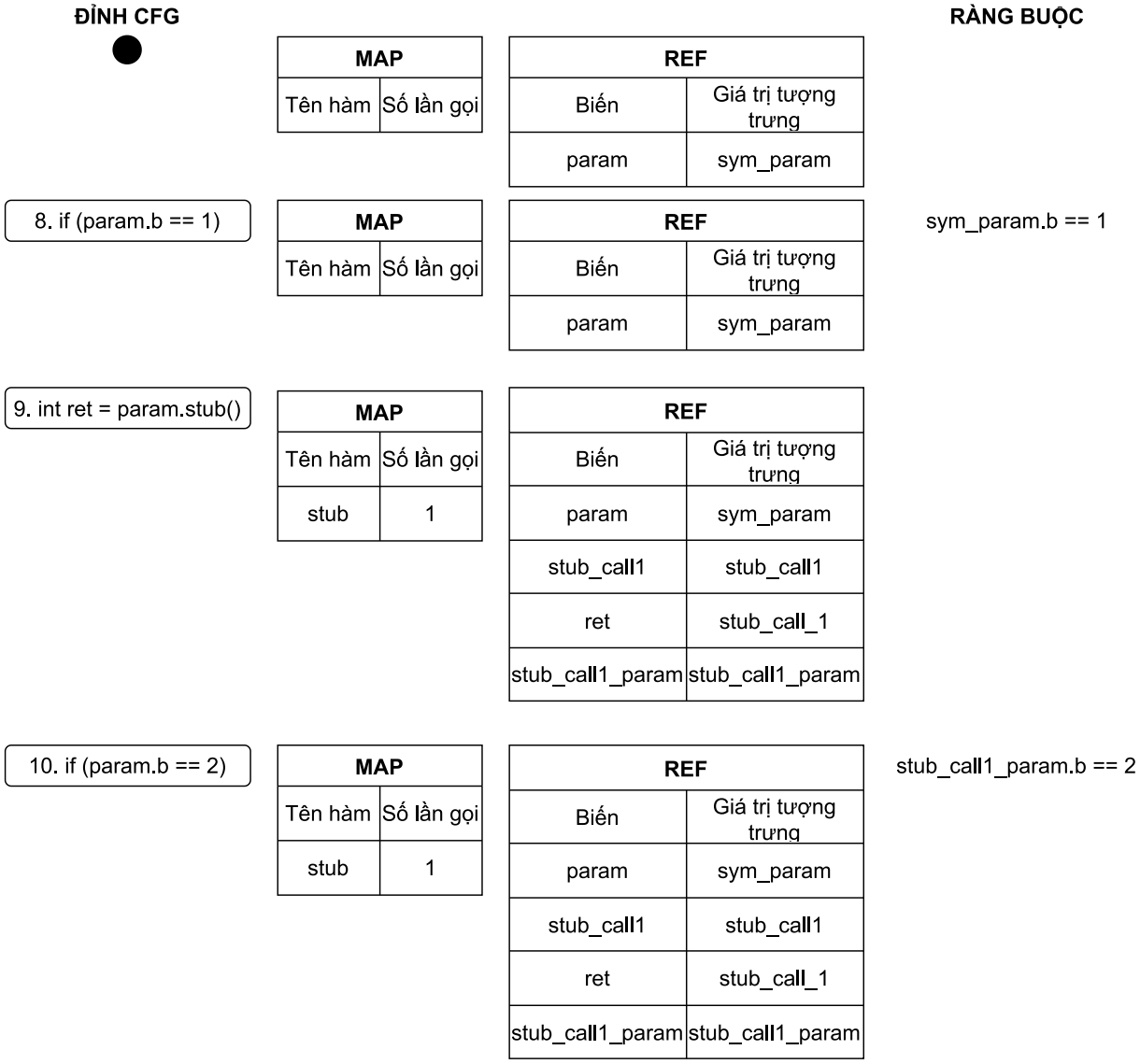
của câu điều kiện `if (param.b == 1)`, dựa trên bảng *REF* quá trình thực thi tượng trưng tạo ra ràng buộc mới `sym_param.b == 1`. Xét tiếp đỉnh CFG của câu lệnh `int ret = param.stub()`, lúc này, Thuật toán 3.4 được áp dụng để xử lý lời gọi phương thức của đối tượng `param`. Trước hết thuật toán cập 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 SMTLib (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. Đoạn mã 3.9 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.

```
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 3.8: Mã nguồn minh họa phương pháp tạo stub tự động cho phương thức của đối tượng.



Hình 3.7: Đồ thị dòng điều khiển của hàm `foo(B param)` trong Đoạn mã 3.8.



Hình 3.8: Quá trình thực thi tượng trưng kết hợp Thuật toán 3.4.

```
class B {
public:
    int b;
    B() {}
    int stub() {
        static int cur_call = 0;
        cur_call++;
        if (TEST_CASE_NAME == "foo") {
            if (cur_call == 1) {
                int stub_call1_param_attr_b = 2;
                this->b = stub_call1_param_attr_b;
                int stub_call1 = 0;
                return stub_call1;
            }
        }
    }
};
int foo(B param) {
    if (param.b == 1) {
        int ret = param.stub();
        if (param.b == 2) {
            return ret;
        }
    }
    return 0;
}
```

Đoạn mã 3.9: Sự thay đổi của hàm `stub` với bộ dữ liệu kiểm thử mới.

Chương 4

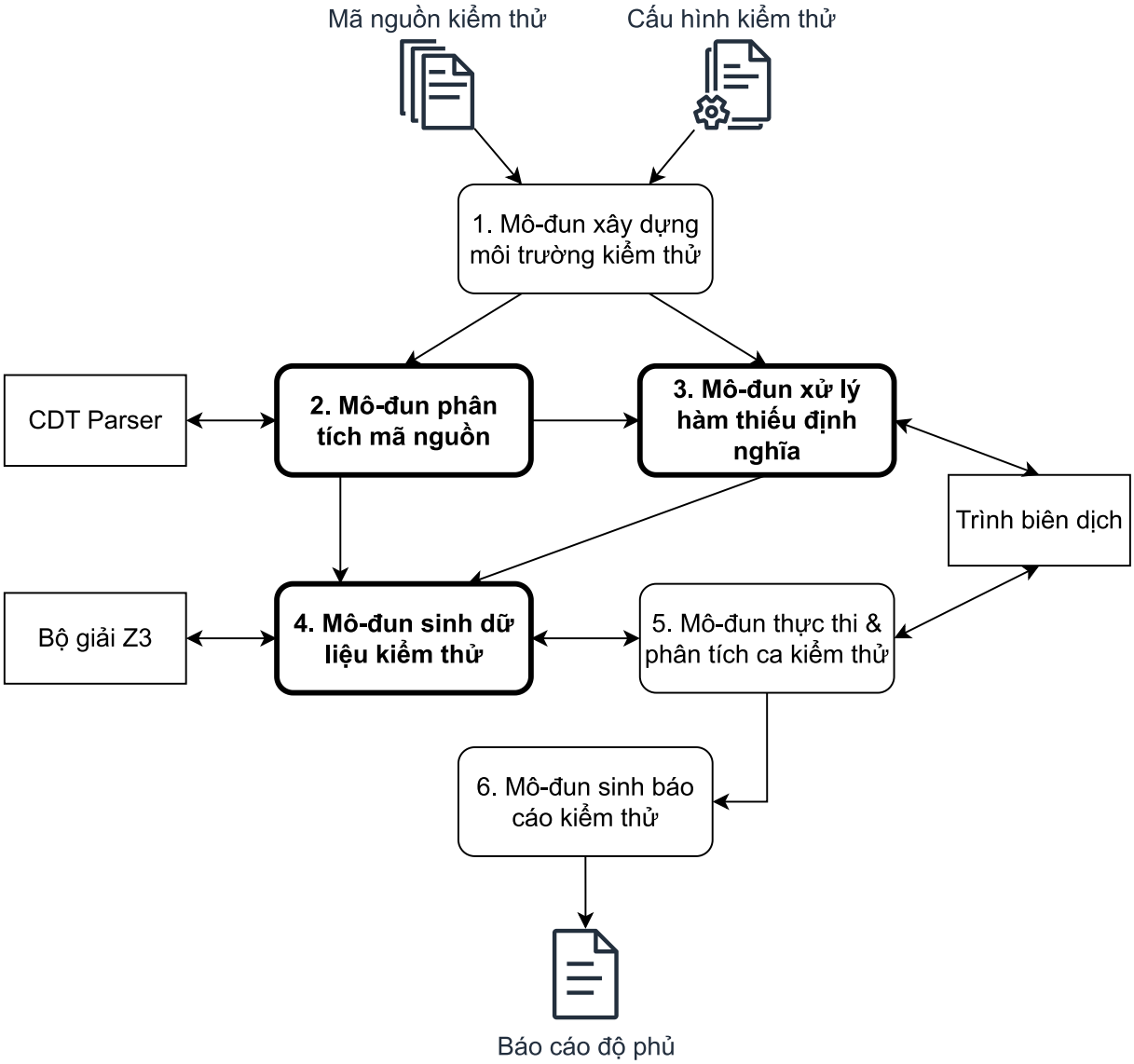
Cài đặt công cụ và thực nghiệm

4.1 Công cụ thực nghiệm

Nhằm đánh giá tính hiệu quả của phương pháp đề xuất, khóa luận đã cài đặt 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 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).

4.1.1 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 4.1 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 đó, khóa luận đã bổ sung mô-đun xử lý hàm thiếu định nghĩa so với kiến trúc gốc, và cải



Hình 4.1: Kiến trúc công cụ AKAUTAUTO.

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 3.2 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 4.1.2 và Mục 4.1.4. 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 4.1.3.

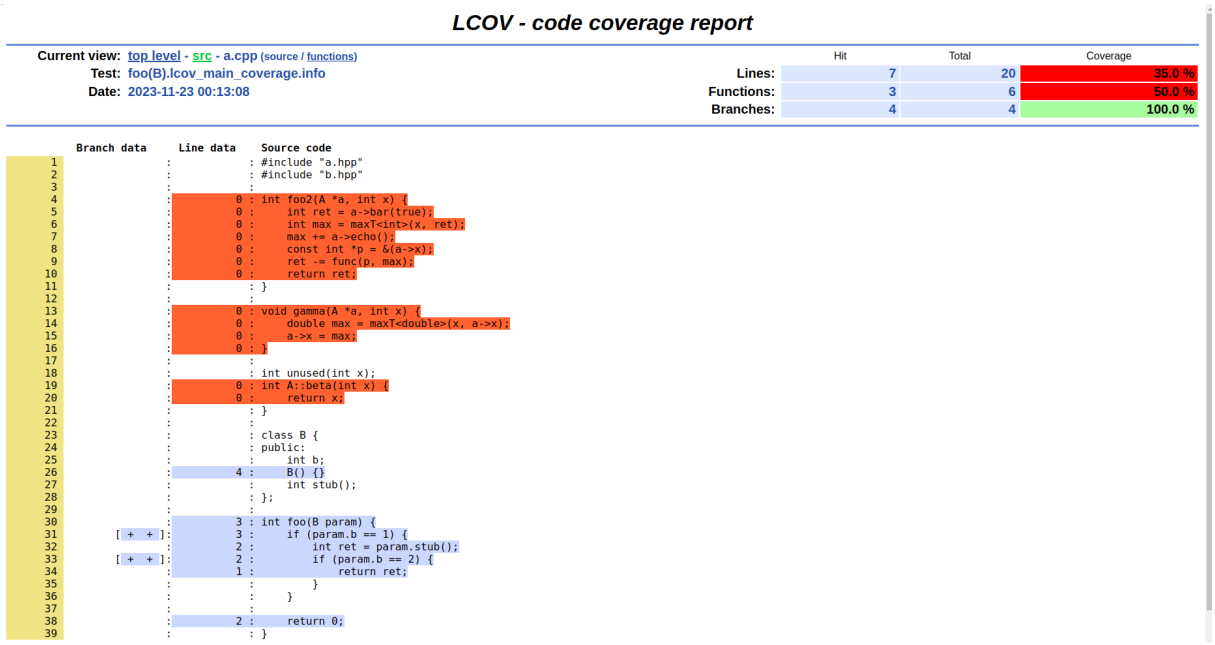
Khóa luận kế thừa mô-đun thực thi và phân tích ca kiểm thử từ kiến trúc gốc của công cụ. Mô-đun này đóng vai trò sinh trình điều khiển kiểm thử, thực thi trình điều khiển và phân tích đường thi hành thu được sau khi chạy ca kiểm thử. Cấu trúc của mô-đun gồm ba thành phần chính với nhiệm vụ chi tiết của 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 phân 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 4.2 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.

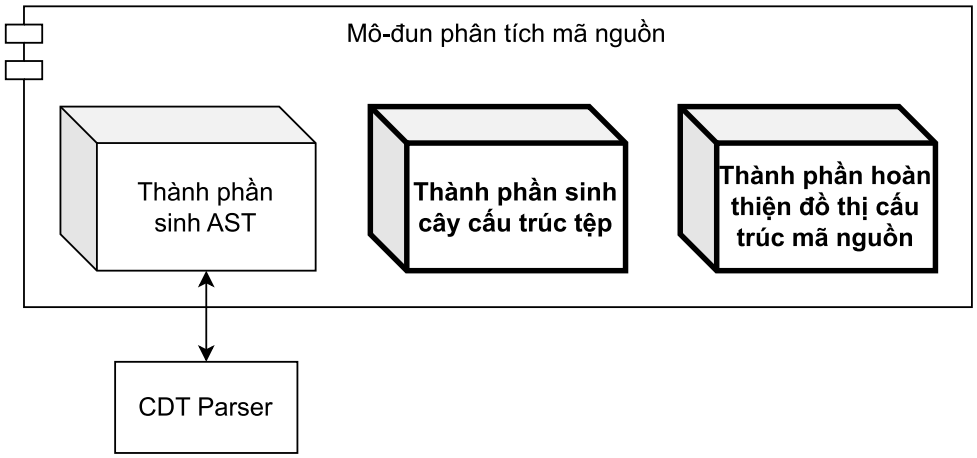


Hình 4.2: Báo cáo kiểm thử LCOV của hàm `foo`.

4.1.2 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 và xây dựng đồ thị cấu trúc mã nguồn. Hình 4.3 mô tả cấu

trúc của mô-đun với ba thành phần lần lượt là thành phần sinh AST, thành phần sinh cây cấu trúc tệp và thành phần hoàn thiện đồ thị cấu trúc mã nguồn. Trong đó, khóa luận kế thừa thành phần AST từ kiến trúc gốc và cải tiến hai thành phần còn lại. Thành phần sinh AST có nhiệm vụ chuyển đổi mã nguồn kiểm thử thành AST thông qua sử dụng thư viện phân tích mã nguồn CDT Parser. Thành phần sinh cây cấu trúc tệp được bổ sung so với mô-đun gốc với nhiệm vụ sinh cây cấu trúc tệp thông qua việc mở rộng đỉnh tệp và tập cạnh cha con tương ứng với đỉnh đó. Thành phần hoàn thiện đồ thị cấu trúc mã nguồn được cải tiến từ thành phần phân tích kiểu dữ liệu từ mô-đun gốc. Trong mô-đun gốc, thành phần này chịu trách nhiệm xác định các kiểu định nghĩa bởi người dùng. Để công cụ có thể khai thác các đặc trưng liên quan đến hướng đối tượng trong ngôn ngữ C++, khóa luận đã bổ sung quá trình xây dựng cạnh kế thừa, cạnh lời gọi hàm.



Hình 4.3: 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.

4.1.3 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 4.4 mô tả các thành 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 3.3. Chi tiết về các thành phần như sau.

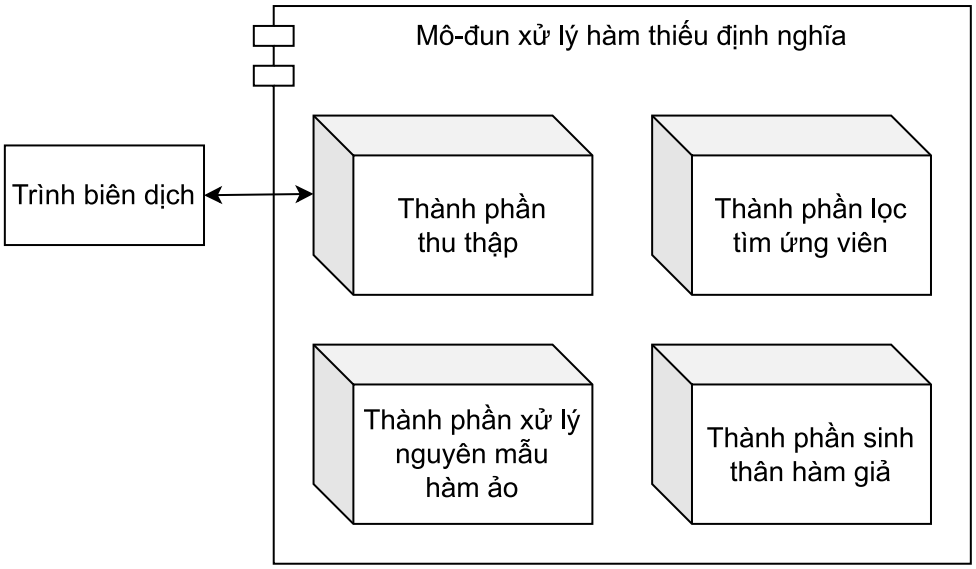
- Thành phần thu thập: Đóng vai trò thu thập danh sách các nguyên mẫu hàm thiếu định nghĩa cần sinh thân hàm giả. Thành phần thu thập sử dụng các công cụ tiện ích của trình biên dịch để thu thập danh sách này. Phương pháp thu thập đã được trình bày ở Mục 3.3.1.

- 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 3.3.2. 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 3.3. 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 3.2 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.



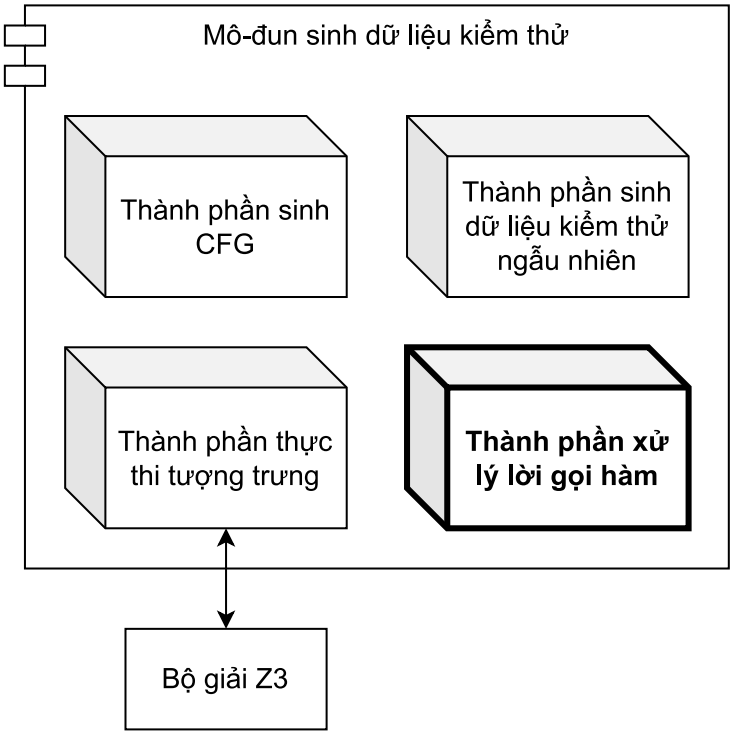
Hình 4.4: Các thành phần trong mô-đun xử lý hàm thiếu định nghĩa.

4.1.4 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 4.5 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 được cải tiến bởi phương pháp đề xuất ở Mục 3.4.2, 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 đồ thị dòng điều khiển (mô tả ở Mục 2.3) 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 3.4.
- 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 (mô tả ở Mục 2.5) 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 3.4.2. 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 4.5: Các thành phần trong mô-đun sinh dữ liệu kiểm thử.

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

Khóa luận tiến hành một số thực nghiệm trên hai phiên bản của công cụ AKAUTAUTO nhằm 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 3.3) 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.

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 3.4.2 và phương pháp truyền thống. Để so

sánh giữa hai phương pháp, khóa luận sử dụng các độ đo về độ phủ mã nguồn C1, C3 [1], số lượng ca kiểm thử sinh ra, thời gian sinh trung bình 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 gồm:

- C-plus-plus[1] (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[2] (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.
- BT_BTServiceProxy (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.
- BT_Telephony (35891 LOC): Mô-đun cung cấp dịch vụ quản lý danh bạ, cuộc gọi trên trình điều khiển đa phương tiện, thuộc một dự án phát triển phần mềm xe ô tô của đơn vị FPT-GAM.

4.3 Đánh giá khả năng xử lý hàm thiếu định nghĩa

4.3.1 Cách thức tiến hành thực nghiệm

Nhằm đảm bảo tính khách quan khi đánh giá sự hiệu quả về mặt thời gian giữa hai phương pháp, quá trình thực nghiệm được tiến hành bởi lập trình viên[3] trực thuộc đơn vị GAM. Để mô phỏng môi trường chứa hàm 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 BT_BTServiceProxy và mô-đun BT_Telephony.

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

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).

4.3.2 Kết quả thực nghiệm

Bảng 4.1 trình bày kết quả thực nghiệm trên hai mô-đun BT_BTServiceProxy và BT_Telephony. 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).

Kết quả thực nghiệm trên hai mô-đun 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 BT_BTServiceProxy 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

Bảng 4.1: Kết quả thời gian chuẩn bị môi trường trên hai mô-đun

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

giây trên mô-đun collision, 14 phút 20 giây trên mô-đun BT_BTServiceProxy 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.

4.3.3 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 cũng 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.
- 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.

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, ta cũng có thể 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.

4.4 Đánh giá khả năng sinh dữ liệu kiểm thử tự động

4.4.1 Cách thức tiến hành thực nghiệm

Nhằm đánh giá khả năng 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 truyền thống 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 BT_BTServiceProxy. Khóa luận chưa thể đánh giá khả năng sinh dữ liệu kiểm thử tự động trên mã nguồn BT_Telephony do mã nguồn này được bảo mật bởi đơn vị FPT-GAM.

4.4.2 Kết quả thực nghiệm

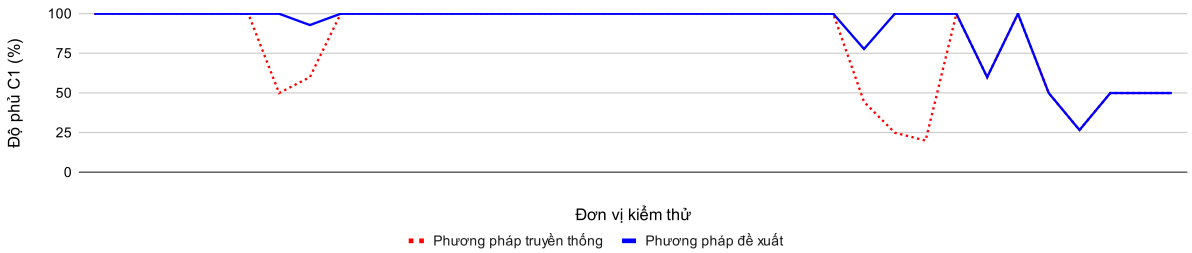
Bảng 4.2 trình bày kết quả thực nghiệm sinh dữ liệu kiểm thử tự động các mã nguồn Box2d và BT_BTServiceProxy giữa phương pháp truyền thống và phương pháp đề xuất. Trong đó, ba tệp đầu lấy từ mã nguồn Box2d và năm tệp sau lấy từ mô-đun BT_BTServiceProxy. Các cột trong bảng có ý nghĩa như sau.

- "File": Tên tệp mã nguồn đuôi .cpp được kiểm thử.
- "Unit": Số lượng hàm trong tệp mã nguồn được kiểm thử.
- "C1": Độ phủ C1 của tệp mã nguồn.
- "C3": Độ phủ C3 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.

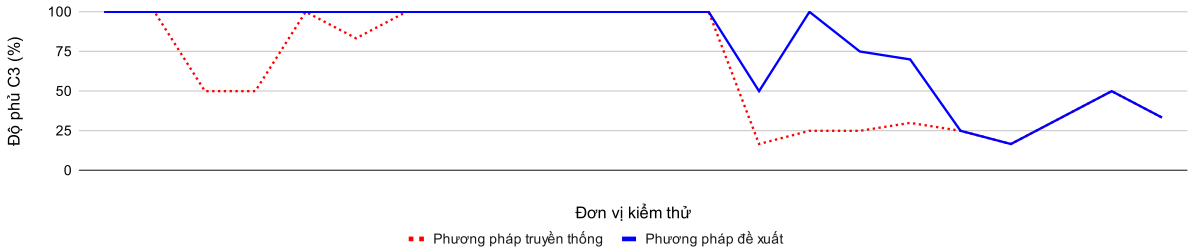
Bảng 4.2: Kết quả thực nghiệm sinh dữ liệu kiểm thử tự động cho một số mã nguồn

File	**Unit**	**Traditional 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
ServiceProxy.cpp	186	89.26	60.67	668	442.53	268.36	**98.43**	**96.05**	703	188.49	46.28
ResponseDispatcher.cpp	17	46.40	29.29	40	1353.71	54.60	**95.41**	**86.55**	75	23.79	63.33
SessionContext.cpp	1	66.67	16.67	7	21.34	72.75	**100**	**100**	7	15.20	54.20
SessionClient.cpp	4	63.07	16.67	13	108.48	278.04	**63.07**	**16.67**	10	6.68	21.35
TokenIdNumbering.cpp	2	92.86	50	4	23.34	10.91	**92.86**	**50**	4	4.37	9.36

Kết quả thực nghiệm ở Bảng 4.2 cho thấy phương pháp đề xuất sinh dữ liệu kiểm thử có độ phủ C1, C3 cao hơn phương pháp truyền thống ở 3/5 tệp mã nguồn được kiểm

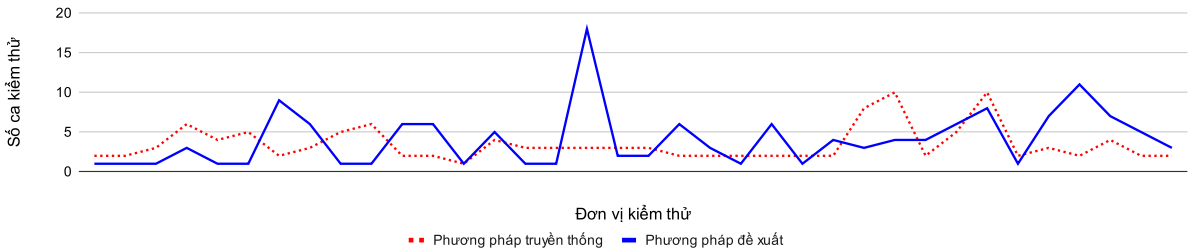


Hình 4.6: So sánh độ phủ C1 giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong C-plus-plus.

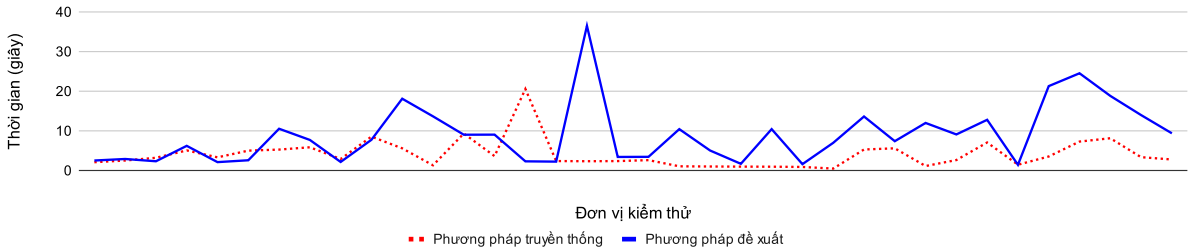


Hình 4.7: So sánh độ phủ C3 giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong C-plus-plus.

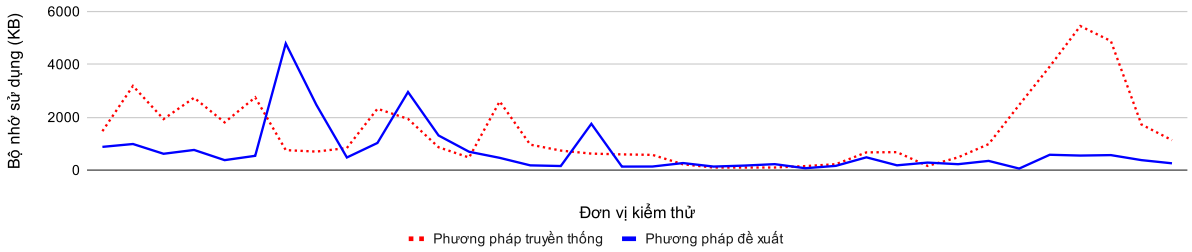
thử. Đồng thời, thời gian sinh và bộ nhớ sử dụng trung bình cho từng hàm trong tệp mã nguồn của phương pháp đề xuất ít hơn so với phương pháp truyền thống. Tuy nhiên, phương pháp đề xuất cần sinh nhiều ca kiểm thử hơn để đạt được độ phủ cao hơn so với phương pháp truyền thống.



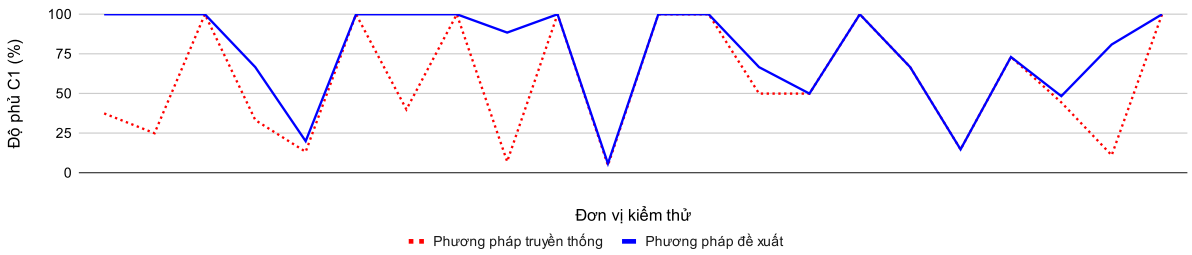
Hình 4.8: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong C-plus-plus.



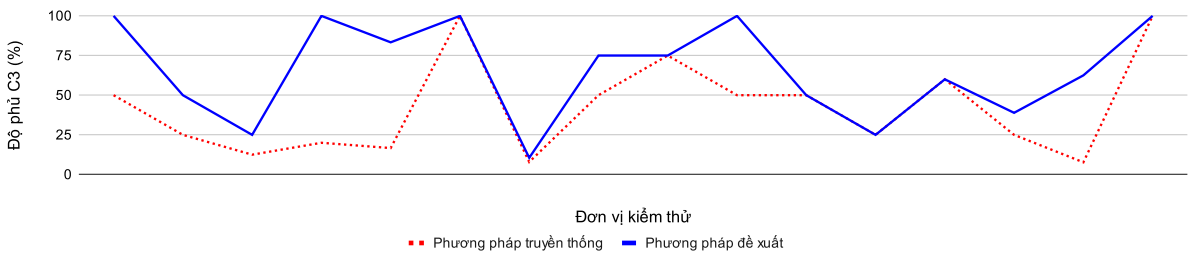
Hình 4.9: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong C-plus-plus.



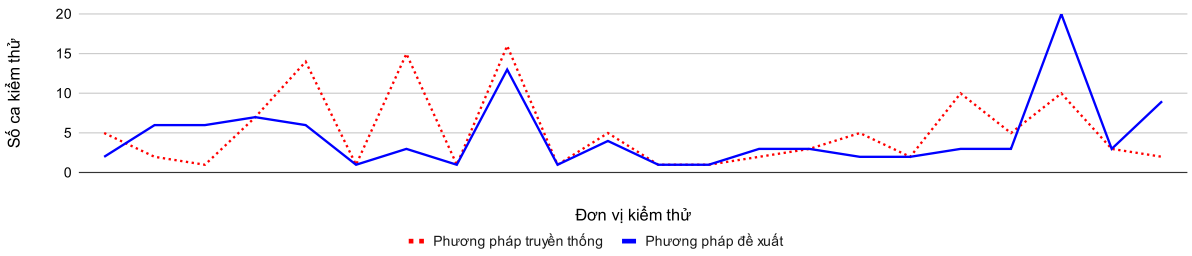
Hình 4.10: So sánh bộ nhớ sử dụng ra giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong C-plus-plus.



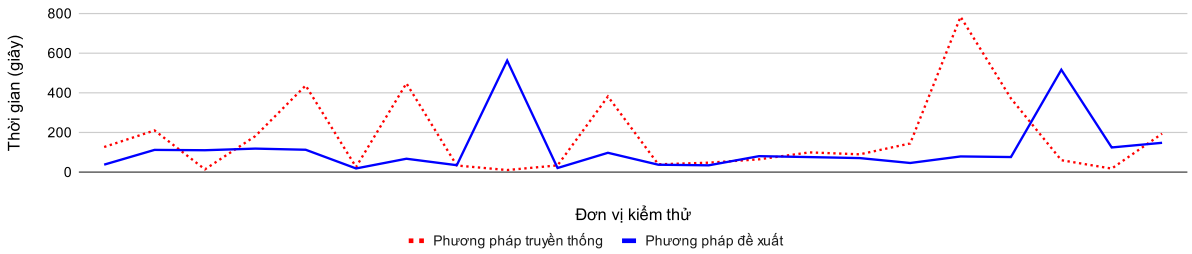
Hình 4.11: So sánh độ phủ C1 giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong Box2d.



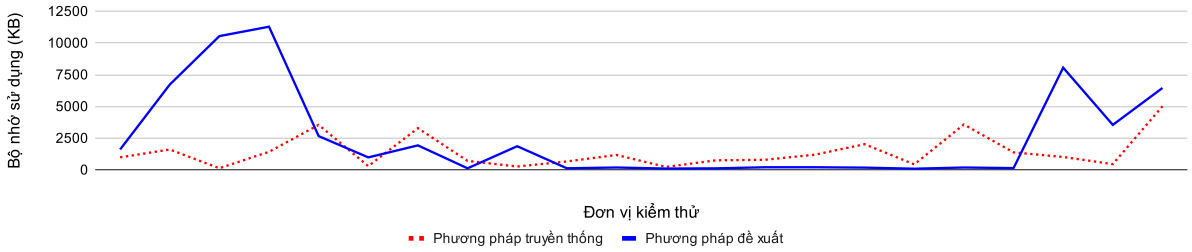
Hình 4.12: So sánh độ phủ C3 giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong Box2d.



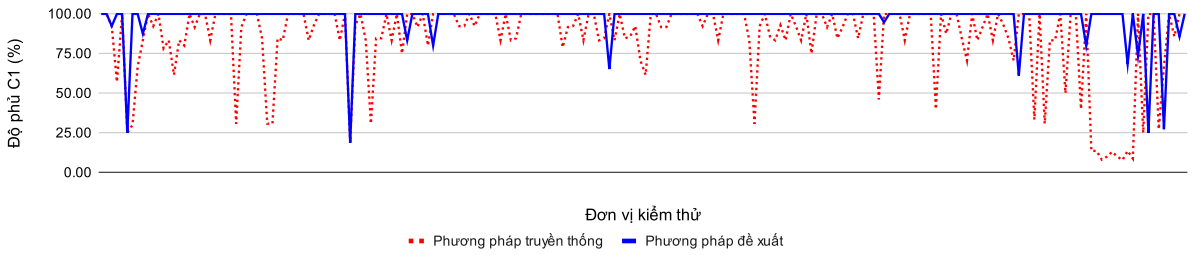
Hình 4.13: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong Box2d.



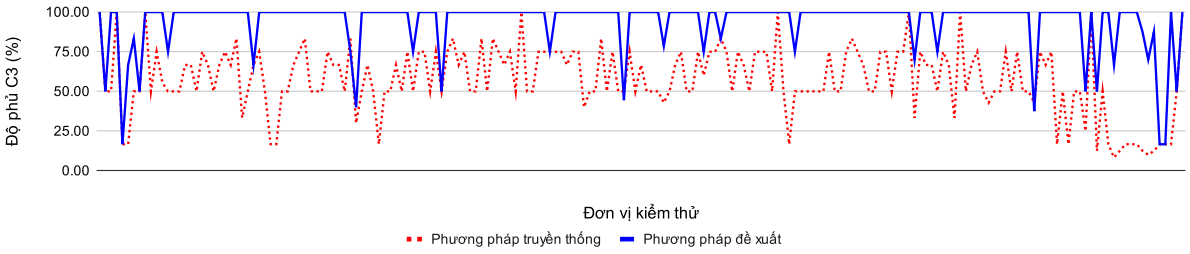
Hình 4.14: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong Box2d.



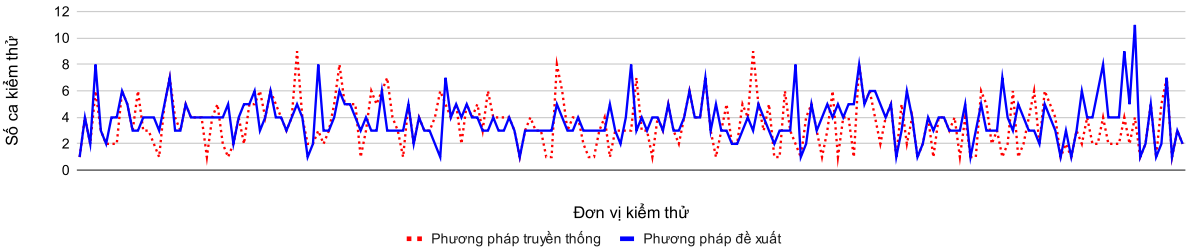
Hình 4.15: So sánh bộ nhớ sử dụng ra giữa phương pháp đề xuất và phương pháp truyền thống trên một số hàm trong Box2d.



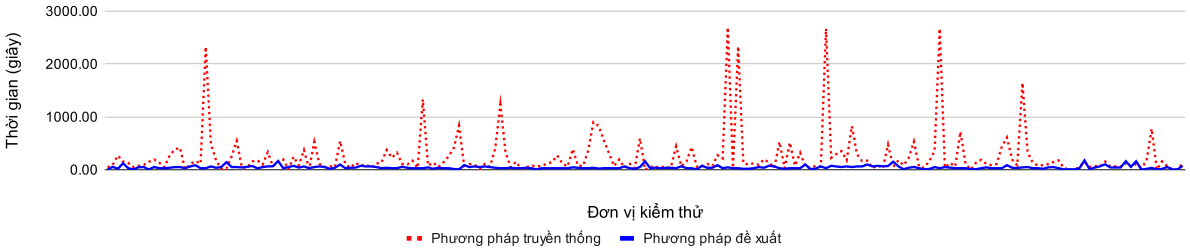
Hình 4.16: So sánh độ phủ C1 giữa phương pháp đề xuất và phương pháp truyền thống trên các hàm trong mô-đun BT_BTServiceProxy.



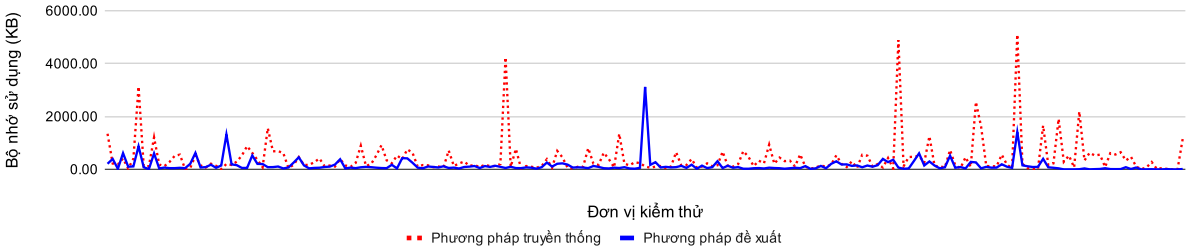
Hình 4.17: So sánh độ phủ C3 giữa phương pháp đề xuất và phương pháp truyền thống trên các hàm trong mô-đun BT_BTServiceProxy.



Hình 4.18: So sánh số ca kiểm thử sinh ra giữa phương pháp đề xuất và phương pháp truyền thống trên các hàm trong mô-đun BT_BTServiceProxy.



Hình 4.19: So sánh thời gian chạy giữa phương pháp đề xuất và phương pháp truyền thống trên các hàm trong mô-đun BT_BTServiceProxy.



Hình 4.20: So sánh bộ nhớ sử dụng giữa phương pháp đề xuất và phương pháp truyền thống trên các hàm trong mô-đun BT_BTServiceProxy.

4.4.3 Đánh giá

Từ kết quả thực nghiệm, ta có thể thấy rằng độ phủ mã nguồn của phương pháp đề xuất luôn lớn hơn hoặc bằng độ phủ mã nguồn của phương pháp truyền thống khi áp dụng lên dự án thiếu định nghĩa hàm và chứa nhiều lời gọi phương thức, đặc biệt ở độ phủ C3. 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 truyền thống.
- Độ phủ mã nguồn có sự tăng đột phá bởi cải tiến trong quá trình xử lý lời gọi hàm. Dự án BT_BTServiceProxy 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, ta có thể nhận thấy rằng một số tệp mã nguồn không tăng kết quả độ phủ so với phương pháp truyền thống. Điều này có thể 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.
- 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 truyền thống áp dụng AS4UT.

- 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.

Chương 5

Kết luận

Chương 6

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

[23] 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.

[25] 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.

[33] 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.

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

[53] 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.

[54] 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.