TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Trương Xuân Hiếu

PHÂN TÍCH TĨNH CHƯƠNG TRÌNH ĐA LUỒNG

KHÓA LUẬN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

Ngành Công nghệ thông tin

Cán bộ hướng dẫn: TS. Tô Văn Khánh

Hà Nội - 2023

LỜI CẢM ƠN

Lời đầu tiên, em xin tỏ lòng biết ơn chân thành và sâu sắc tới TS.Tô Văn Khánh – giảng viên làm việc tại trường Đại học Công Nghệ - Đại học Quốc Gia Hà Nội, người đã hướng dẫn tận tình, động viên và giúp đỡ em trong suốt quá trình thực hiện khóa luận tốt nghiệp này.

Em cũng xin gửi cảm ơn tới các quý thầy cô Khoa Công nghệ thông tin nói riêng và trường Đại học Công nghệ - Đại học Quốc gia Hà Nội nói chung, đã dìu dắt, bảo ban em trong thời gian em vẫn còn ngồi ghế nhà trường. Những kiến thức mà các quý thầy cô tận tình truyền đạt đã giúp em có được nền tảng vững chắc trên con đường học tập và làm việc.

Cuối cùng, em xin cảm ơn tới các anh chị và các bạn, đặc biết là tập thể lớp QH-2019-I/CQ-C-D đã đồng hành cùng em trong suốt những năm học qua. Mọi người không chỉ tận tình chia sẻ, giúp đỡ và đưa ra những góp ý chân thành mà còn là chỗ dựa tinh thần, san sẻ niềm vui nỗi buồn, giúp em vượt qua những khó khăn trong học tập cũng như là cuộc sống.

LỜI CAM ĐOAN

Em xin cam đoan rằng những nghiên cứu về phương pháp phân tích tĩnh chương trình đa luồng được trình bày trong khóa luận này là của em và chưa từng được nộp như một báo cáo khóa luận tại trường Đại học Công Nghệ - Đại học quốc gia Hà Nội hoặc bất kỳ trường đại học khác. Những gì em viết ra không sao chép từ các tài liệu, không sử dụng các kết quả của người khác mà không trích dẫn cụ thể. Em xin cam đoan cốt lõi của phương pháp mã hóa sinh ràng buộc được sử dụng trong công cụ được trình bày trong khoá luận là do em tự phát triển, không sao chép mã nguồn của người khác.

Nếu sai em hoàn toàn chịu trách nhiệm theo quy định của trường Đại Học Công Nghệ - Đại Học Quốc Gia Hà Nội.

Hà Nội, ngày..... tháng..... năm.....

Sinh viên

Trương Xuân Hiếu

TÓM TẮT

Tóm tắt: Sự phát triển bùng nổ của ngành công nghiệp phần mềm trong những năm gần đây đã kéo theo rất nhiều hệ quả. Các phần mềm càng ngày càng lớn dần, càng trở nên phức tạp, việc thực thi các phần mềm đó cũng càng ngày càng tiêu tốn nhiều thời gian. Để giảm thiểu chi phí về thời gian, khái niệm thực thi đa luồng (multi threading) đã xuất hiện. Thực thi đa luồng giúp phần mềm có thể thực hiện nhiều tác vụ cùng một lúc, giúp giảm thiểu chi phí thời gian đi rất nhiều. Tuy nhiên, bên cạnh đó, thực thi đa luồng lại khá phức tạp trong việc xác định trình tự thực thi vì câu lệnh của các luồng trong chương trình chạy đan xen (interleave) vào nhau không theo một trình tự cố định, kéo theo đó là sự gia tăng của chi phí kiểm thử. Ngoài ra, các phương pháp kiểm thử truyền thống không thể đảm bảo rằng toàn bộ các trình tự thực thi đều được kiểm thử, dẫn đến việc kiểm thử thiếu, làm kết quả kiểm thử bị sai lệch so với thực tế. Do đó, nghiên cứu này đề xuất một phương pháp kiểm chứng chương trình đa luồng có tên là All-Constraints-In-One (ACIO) để thay thế cho các phương pháp kiểm thử truyền thống. Phương pháp kiểm chứng này sử dụng đồ thị để biểu diễn các hành động đọc/ghi giá trị (Event Order Graph - EOG) của chương trình đa luồng và tìm ra điểm xung đột đọc – ghi giữa các biến dùng chung của các luồng, từ đó xây dựng nên các ràng buộc. Diễn giải về phương pháp kiểm chứng ACIO sẽ được trình bày chi tiết trong tài liệu này.

Từ khóa: *Kiểm chứng đa luồng, Event Order Graph, thực thi tượng trưng.*

MỤC LỤC

DANH SÁCH THUẬT NGỮ VIẾT TẮT

- **ACIO** All-Constraints-In-One
- **AST** Abstract Syntax Tree – Cây cú pháp trừu tượng
- **CFG** Control Flow Graph – Đồ thị luồng điều khiển
- **EOG** Event Order Graph – Đồ thị thứ tự sự kiện
- **SMT** Satifiability Modulo Theories
- **SV-COMP** Software Verification Competition

DANH SÁCH BẢNG BIỂU

DANH SÁCH HÌNH VẼ

MỞ ĐẦU

Hiện nay, nhờ vào sự phát triển của hệ thống phần cứng, hệ thống phần mềm đã có những bước tiến mạnh mẽ. Hệ thống phần cứng với tốc độ nhanh hơn, kích thước lưu trữ lớn hơn cho phép mở rộng quy mô cũng như là khả năng của hệ thống phần mềm. Tuy nhiên, song hành với sự mở rộng của quy mô phần mềm là nhu cầu rất lớn đến việc cải thiện tốc độ và cách tối ưu hóa xử lý của mã nguồn. Với chương trình thông thường nơi các câu lệnh được thực thi hoàn toàn tuyến tính, việc thực thi chương trình đó chỉ sử dụng tối đa một luồng của CPU bất kể quy mô của chương trình là lớn hay nhỏ. Đây hoàn toàn là một sự phí phạm vì với sự hỗ trợ của công nghệ CPU đa luồng hiện thời, một chương trình hoàn toàn có thể được phân chia thành các nhánh nhỏ và chia đều cho các luồng trong CPU để thực thi thay vì gán toàn bộ trách nhiệm đó cho một luồng duy nhất. Việc chia nhỏ trách nhiệm thực thi không chỉ giảm tải công việc cho từng CPU đơn lẻ mà còn tối đa hóa hiệu quả của phần cứng, làm tăng tốc độ thực thi lên rất nhiều lần. Đó là lí do thực thi đa luồng ngày một trở nên phổ biến và dần trở thành một trong những thành phần không thể thiếu của hệ thống phần mềm.

Tuy vậy, đánh đổi với tốc độ thực thi, chương trình đa luồng đặt ra một thách thức rất lớn với công đoạn đảm bảo chất lượng phần mềm. Các câu lệnh được thực thi song song trên các luồng khác nhau với thứ tự không định trước gây ra rất nhiều khó khăn trong việc tìm ra trình tự thực thi của chương trình. Một chương trình gồm vài luồng đơn giản cũng có thể có đến hàng chục hoặc hàng trăm trình tự thực thi khả thi, mỗi lần chương trình thực thi lại thao tác và trả về một bộ giá trị khác nhau dựa trên trình tự thực thi tương ứng. Việc kiểm tra hàng chục, hàng trăm thậm chí hàng nghìn trình tự thực thi khả thi trở thành một bài toán rất nan giải và cấp bách với các kiểm thử viên nói riêng và toàn ngành công nghệ phần mềm nói chung.

Cũng bởi sự phức tạp trong việc kiểm chứng chương trình đa luồng, đã có rất nhiều diễn đàn được mở ra để phân tích các thách thức và thảo luận về phương pháp giải quyết. Một trong những diễn đàn có thể kể đến là cuộc thi *SV-COMP* [6]. Mục tiêu của cuộc thi là tìm ra giải pháp tốt nhất cho việc kiểm chứng chương trình đa luồng. Các đội tham dự cần xây dựng công cụ để kiểm chứng các chương trình mẫu cho trước (*Benchmark*). Kết quả của từng đội sẽ được đánh giá và cho điểm dựa trên rất nhiều thang đo khác nhau như *ReachSafety*, *MemorySafety*, *ConcurrencySafety*,... trước khi được tổng kết và xếp hạng.

Một trong những phương pháp rất hiệu quả trong việc kiểm chứng chương trình đa luồng được các đội tham dự thi đấu sử dụng đó là mã hóa các sự kiện đọc / ghi giá trị của các biến dùng chung để kiểm tra các xung đột. Một chương trình dù đơn giản hay phức tạp thì dưới sự thực thi của bộ nhớ máy tính, chương trình đó cũng chỉ bao gồm việc đọc giá trị từ các ô nhớ, thao tác với các giá trị đọc được rồi ghi lại giá trị vào một số ô nhớ nào đó. Do đó, việc mã hóa các sự kiện đọc/ghi không chỉ đảm bảo mã hóa đủ thông tin cần thiết mà còn không mã hóa các thông tin dư thừa.

Một trong số nghiên cứu đạt thành tích cao trong cuộc thi *SV-COMP* sử dụng việc mã hóa các sự kiện đọc/ghi là công cụ kiểm chứng *Yogar-CBMC*, sử dụng kiến trúc đồ thị *EOG* [12]. Công cụ cho kết quả tốt trên bộ thực nghiệm của cuộc thi SV-COMP [7] với việc kiểm chứng đúng 1047 chương trình đa luồng trong thời gian giới hạn với điểm số cao nhất đạt được là 1293 (số liệu năm 2017). Tuy nhiên, do phương pháp *Yogar-CBMC* chỉ mã hóa một phần các sự kiện đọc/ghi thành các ràng buộc mà bỏ qua việc xử lí xung đột giữa chúng nên các trình tự thực thi được tìm ra có thể là không khả thi, cần phải được phân tích và sàng lọc lại trước khi đưa ra kết luận. Việc sinh ra các trình tự thực thi không khả thi này sẽ đặt gánh nặng lên bộ giải *SMT* - công cụ được sử dụng để tính toán tập các ràng buộc.

Trong nghiên cứu này, khóa luận đề xuất phương pháp có tên là *All-Constraints-In-One* (*ACIO*) kết hợp đồ thị thứ tự sự kiện (*EOG*). Việc sinh tất cả các ràng buộc cung cấp một phương thức để xây dựng nên toàn bộ các trình tự thực thi của chương trình cần kiểm chứng. Khác với phương pháp *Yogar-CBMC* chỉ mã hóa một phần các sự kiện đọc/ghi, phương pháp *ACIO* không chỉ mã hóa các sự kiện đọc/ghi mà còn mã hóa cả các xung đột giữa các sự kiện đó. Điều đó không chỉ đảm bảo các trình tự thực thi được tìm ra là luôn khả thi do chúng bao hàm cả các điều kiện chống xung đột mà còn giúp giảm tải lượng công việc đặt lên bộ giải *SMT* do tập các ràng buộc của chương trình tương ứng là duy nhất.

Cấu trúc của khóa luận được trình bày trong 4 phần: Chương 1 khóa luận sẽ làm rõ các lý thuyết và vấn đề trong kiểm chứng chương trình đa luồng. Tiếp đến chương 2 là các khái niệm của kỹ thuật thực thi tượng trưng và phương pháp kiểm chứng chương trình dựa trên kỹ thuật thực thi tượng trưng. Trong chương 3, khóa luận sẽ trình bày về phương pháp kiểm chứng chương trình đa luồng được nghiên cứu. Cuối cùng, chương 4 sẽ mô tả về công cụ kiểm chứng được xây dựng dựa trên phương pháp đã được trình bày, đồng thời là các kết quả thực nghiệm để chứng minh công cụ đáp ứng được các yêu cầu kiểm chứng đặt ra.

CHƯƠNG 1. KIỂM CHỨNG CHƯƠNG TRÌNH ĐA LUỒNG

1.1. Kiểm chứng

Sự phát triển của công nghệ phần cứng kéo theo sự bùng nổ của ngành công nghiệp phần mềm khiến cho nhu cầu đảm bảo chất lượng phần mềm càng ngày càng được chú ý, đặc biệt là những phần mềm phức tạp, sử dụng kiến trúc đa luồng.

Phần lớn các quy trình đảm bảo chất lượng phần mềm hiện nay được xây dựng dựa trên phương pháp kiểm thử. Có thể nói kiểm thử là một phương pháp khá hiệu quả trong việc kiểm tra tính đúng đắn của phần mềm với các đầu vào cụ thể phổ biến. Tuy nhiên, các đầu vào cụ thể phổ biến chỉ chiếm một phần rất nhỏ trong toàn bộ các đầu vào khả thi, và dù rằng xác xuất xảy ra của chúng là "phổ biến" không có nghĩa là chúng luôn luôn xảy ra. Việc kiểm thử thiếu các đầu vào ít phổ biến hơn có thể dẫn đến việc bỏ sót nhiều lỗi của chương trình, làm ảnh hưởng đến kết quả kiểm thử.

Chính bởi sự khiếm khuyết của phương pháp kiểm thử, phương pháp kiểm chứng đã được ra đời nhằm mục tiêu kiểm tra toàn bộ các đầu vào khả thi của chương trình để đảm bảo chương trình hoàn toàn không có lỗi.

1.1.1. Kiểm chứng chương trình

Kiểm chứng chương trình (software verification) là quy trình xác minh xem chương trình hoạt động đúng như mong đợi hay không. Trong quy trình kiểm chứng, các kỹ sư phần mềm sẽ sử dụng các kỹ thuật khác nhau để tìm ra các lỗi hoặc sai sót trong chương trình, từ đó cải thiện chất lượng và độ tin cậy của chương trình.

Kiểm chứng chương trình có thể bao gồm các phương pháp kiểm thử (testing) để đảm bảo chương trình hoạt động đúng trong các trường hợp sử dụng thực tế, kiểm tra mã nguồn (code review) để tìm lỗi cú pháp hoặc thiếu sót trong logic của chương trình, hay sử dụng các công cụ phân tích tĩnh (static analysis) để phát hiện các lỗi tiềm ẩn trong mã nguồn.

Mục tiêu của kiểm chứng chương trình là đảm bảo rằng chương trình đáp ứng được yêu cầu và hoạt động đúng, đáng tin cậy và an toàn. Nó là một phần quan trọng trong quá

trình phát triển phần mềm và đảm bảo rằng chương trình sẽ không gây ra sự cố hoặc tổn thất.

1.1.2. Kiểm chứng mô hình

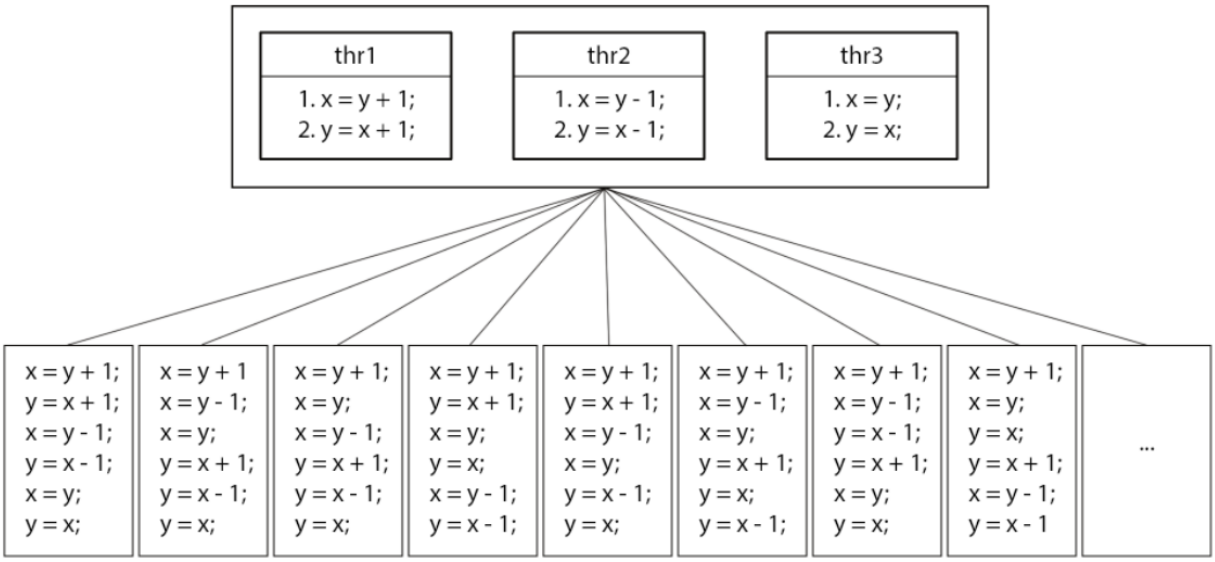
Trong thực tế thiết kế và phát triển phần mềm, công việc kiểm chứng tiêu tốn nhiều thời gian và công sức hơn là việc xây dựng hệ thống. Do đó, nhu cầu tìm kiếm một kỹ thuật kiểm chứng hiệu quả và giảm thiểu tối đa chi phí đang ngày càng trở nên cấp thiết. Một trong những kỹ thuật kiểm chứng cho hiệu quả cao có thể kể đến là kỹ thuật kiểm chứng mô hình.

Kỹ thuật kiểm chứng mô hình là quy trình sử dụng các phương pháp, công cụ và kỹ thuật mô hình hóa để đánh giá tính đúng đắn, hiệu quả và đáng tin cậy của mô hình. Kỹ thuật này dựa trên cơ sở mô tả các hành vi của hệ thống bằng các công thức toán học. Phương pháp kiểm chứng mô hình sẽ kiểm tra tất cả các kịch bản khả thi bằng cách vét cạn toàn bộ các trạng thái của hệ thống. Nhờ đó, kiểm chứng mô hình hoàn toàn có thể khẳng định rằng hệ thống có thật sự thỏa mãn các thuộc tính nhất định nào đó.

Áp dụng lý thuyết về kiểm chứng mô hình, các công cụ kiểm chứng mô hình sẽ kiểm tra tất cả các trạng thái có liên quan của hệ thống. Trong trường hợp một tập các trạng thái nào đó vi phạm đặc tính mà đặc tả mong muốn, một phản ví dụ (counter example) cho thấy cách mà hệ thống hoạt động để tạo ra lỗi sẽ được sinh ra. Nhờ đó, người thiết kế và cài đặt có thể dễ dàng hơn trong việc kiểm tra và gỡ lỗi, đồng thời tích hợp các mô hình phù hợp.

1.2. Vấn đề kiểm chứng chương trình đa luồng

Trong quá trình đảm bảo chất lượng phần mềm, việc kiểm chứng các chương trình đa luồng là một trong những vấn đề thách thức nhất. Về mặt lý thuyết, các luồng trong chương trình được thực thi một cách “song song”. Tuy nhiên trong thực tế, các câu lệnh trong các luồng khi thao tác lên các biến dùng chung (global variables) sẽ đan xen vào nhau theo một thứ tự không cố định. Việc đan xen không theo thứ tự cố định này khiến việc xác định trình tự thực thi trở nên vô cùng phức tạp. Có thể phát biểu rằng kiểm chứng chương trình đa luồng bằng việc xây dựng các trình tự thực thi dựa trên quy trình xen kẽ các câu lệnh là một công việc vô cùng khó khăn và yêu cầu rất nhiều về mặt chi phí lẫn công sức.



Hình 1.1. Một số trình tự thực thi của chương trình đa luồng

Xét một chương trình đơn giản chỉ gồm ba luồng chạy song song *thr1, thr2 và thr3*, mỗi luồng bao gồm hai câu lệnh gán giá trị (*Hình 1.1.*) Giả thiết rằng dòng lệnh *thr1.1* sẽ được thực thi đầu tiên, khi đó tập các dòng lệnh tiếp theo có thể được thực thi là *{thr1.2, thr2.1, th3.1}.* Nếu trong ba dòng lệnh đó, *thr2.1* được chọn để thực thi thì tập các dòng lệnh tiếp theo có thể được thực thi sẽ trở thành *{thr1.2, thr2.2, th3.1}.* Tổng quát hơn, mỗi khi một dòng lệnh trong một luồng được thực thi, luồng điều khiển có thể tiếp tục thực thi dòng lệnh tiếp theo trong cùng luồng hoặc chuyển sang thực thi các luồng còn lại. Nói cách khác, với chương trình có *n* luồng, mỗi bước lựa chọn dòng lệnh thực thi tiếp theo sẽ có tối đa *n* lựa chọn. Cho rằng mỗi luồng trong chương trình đó có *m* dòng lệnh, tức là toàn bộ chương trình sẽ có *m*n* dòng lệnh, đồng thời coi các trường hợp đặc biệt (bước lựa chọn dòng lệnh thực thi tiếp theo có ít hơn n lựa chọn) cũng có n lựa chọn, việc xác định trình tự thực thi của chương trình sẽ là *m*n* lần chọn *1* trong *n* khả năng khả thi. Điều đó có nghĩa là một chương trình *n* luồng song song, mỗi luồng có *m* dòng lệnh ước tính sẽ có n^{m*n} trình tự thực thi khác nhau đôi một.

Công thức trên có sai số khá lớn khi áp dụng vào ví dụ đang xét với $3^{2*3} = 729$ luồng thực thi trên tính toán và chỉ 90 luồng thực thi trong thực tế. Nguyên nhân là do các luồng trong chương trình chỉ bao gồm hai câu lệnh khiến các trường hợp đặc biệt chiếm đa số. Dù vậy, có thể thấy rằng 729 hay 90 đều là một con số tương đối lớn so với con số 6 của tổng số dòng lệnh được thực thi.

CHƯƠNG 2. KIỂM CHỨNG DỰA TRÊN KỸ THUẬT THỰC THI TƯỢNG TRƯNG

2.1. Kỹ thuật thực thi tượng trưng

Thực thi tượng trưng (Symbolic Execution - SE) [4] là một kỹ thuật trong lĩnh vực kiểm thử phần mềm, cho phép tự động tìm kiếm các lỗi trong mã nguồn của chương trình bằng cách phân tích các đường đi có thể của chương trình dựa trên các biểu thức và ràng buộc logic.

Ý tưởng chính của thực thi tượng trưng là thay thế việc sử dụng tham số đầu vào là các giá trị cụ thể (*concrete value*) bằng các giá trị tượng trưng (*symbolic value*), khi đó đầu ra sẽ được tính toán dựa trên các giá trị tượng trưng này và được biểu diễn dưới dạng một biểu thức tượng trưng.

Lấy ví dụ với câu lệnh sau:

if a then b else c

Đối với việc sử dụng tham số đầu vào là các giá trị cụ thể, chỉ có duy nhất một luồng điều khiển được thực thi. Mỗi một lần thực thi cụ thể, luồng điều khiển chỉ có thể rẽ vào một trong hai nhánh ***b*** hoặc ***c***, tùy theo giá trị cụ thể của ***a***. Vì vậy, trong hầu hết các trường hợp, thực thi cụ thể chỉ có thể đánh giá tương đối độ thỏa mãn của chương trình với đặc tả yêu cầu.

Ngược lại, đối với thực thi tượng trưng, việc đi theo nhánh ***b*** hay ***c*** không được cố định vì ***a*** không cụ thể. Tại điểm rẽ nhánh ***a***, cả hai nhánh ***b*** và ***c*** cùng đồng thời được xem xét nhằm định hướng cho bước kế tiếp của luồng điều khiển. Điều đó đảm bảo tất cả các luồng điều khiển của chương trình đều được kiểm tra một cách đầy đủ và chính xác.

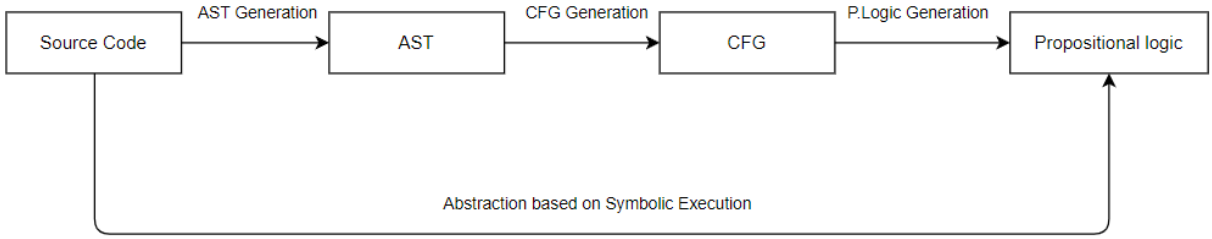
2.2. Trừu tượng hóa chương trình dựa trên kỹ thuật thực thi tượng trưng

Một vấn đề mà phần lớn các công cụ kiểm chứng đều mắc phải, điển hình là các công cụ kiểm chứng dựa trên kỹ thuật kiểm chứng mô hình đó là việc bùng nổ số trạng thái. Việc áp dụng phương pháp thực thi tượng trưng để trừu tượng hóa chương trình vào kiểm chứng

sẽ giúp giảm thiểu nguy cơ bùng nổ trạng thái, từ đó có thể sử dụng để kiểm chứng cho các hệ thống lớn và phức tạp.

Thay vì biểu diễn cụ thể từng đường thi hành, chương trình sẽ được trừu tượng hóa thành các biểu thức logic vị từ. Các biểu thức logic đó sẽ được sử dụng để phân tích, kiểm tra mã nguồn và sự thỏa mãn của chương trình với các điều kiện được đặc tả bởi người dùng.

Tuy nhiên, việc tổng quát hóa trực tiếp mã nguồn của một chương trình thành tập các biểu thức logic vị từ trong mã máy là một việc rất khó khăn. Do đó, mã nguồn của chương trình cần phải được biểu diễn thành các công thức trung gian là cây cú pháp trừu tượng AST và đồ thị luồng điều khiển CFG (*Hình 2.1.*) Các công thức trung gian này cung cấp khả năng trừu tượng hóa một phần mã nguồn chương trình thành mã máy bán trừu tượng, giúp việc trừu tượng hóa hoàn toàn chương trình thành biểu thức logic vị từ trong mã máy trở nên đơn giản và dễ dàng hơn.



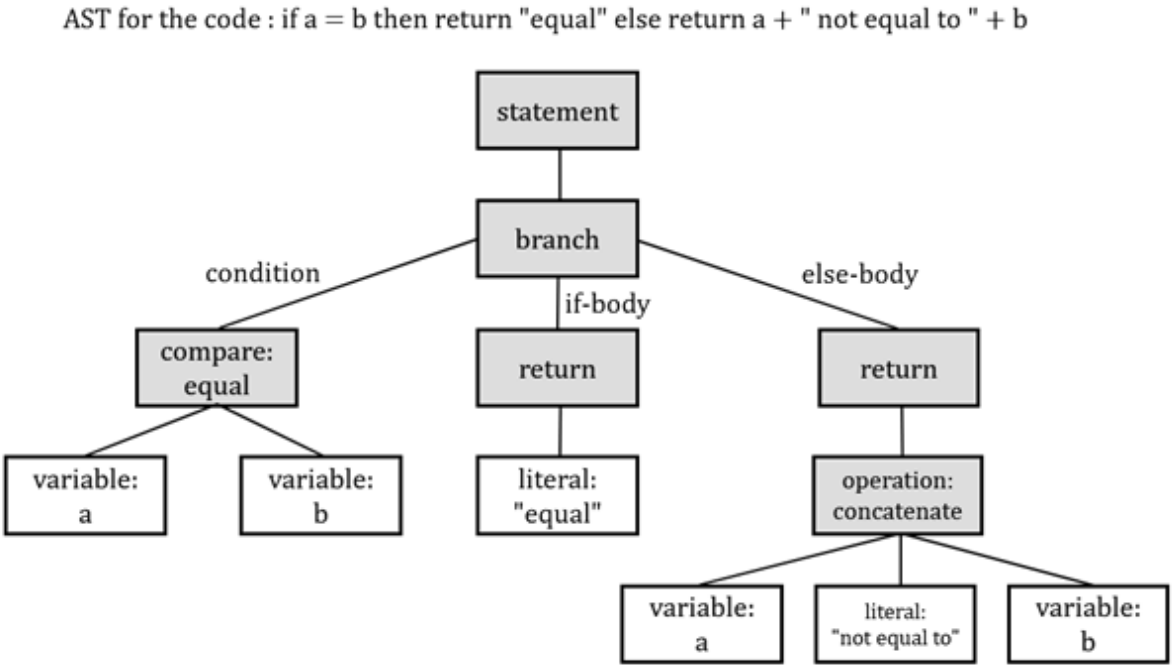
Hình 2.1. Trừu tượng hóa dựa trên thực thi tượng trưng

2.2.1. Cây cú pháp trừu tượng - AST

Cây cú pháp trừu tượng (*Abstract Syntax Tree - AST*) là một cấu trúc dữ liệu phân cấp biểu diễn cú pháp trừu tượng của một đoạn mã (*code*) trong một ngôn ngữ lập trình nhất định. Nó được sử dụng như một công cụ phân tích cú pháp (*parser*) trong quá trình biên dịch hoặc thông dịch mã nguồn. *AST* được tạo ra bằng cách duyệt cây cú pháp của một đoạn mã và tạo ra các đối tượng phân tích cú pháp (*parse nodes*) biểu diễn cho mỗi phần của cú pháp. Các đối tượng này được sắp xếp theo thứ tự phân cấp tương ứng với cấu trúc cú pháp của đoạn mã. *AST* được sử dụng trong các công cụ phân tích mã nguồn như trình biên dịch hoặc trình thông dịch để thực hiện các tác vụ phân tích mã như tối ưu hóa, dò lỗi, tạo mã trung gian, hoặc thậm chí là tự động sinh mã.

Hình 2.2. dưới đây biểu diễn cho cấu trúc *AST* của đoạn mã

if (a == b) then return "equal"; else return a + "not equal" + b;



Hình 2.2. Trừu tượng hóa chương trình thành Abstract Syntax Tree

2.2.2. Đồ thị luồng điều khiển – CFG

Đồ thị luồng điều khiển (Control flow graph - CFG) là một đồ thị biểu diễn luồng điều khiển hoặc luồng thực thi của một chương trình máy tính. CFG được sử dụng để trực quan hóa cách chương trình thực thi và phân tích hành vi của chương trình đó.

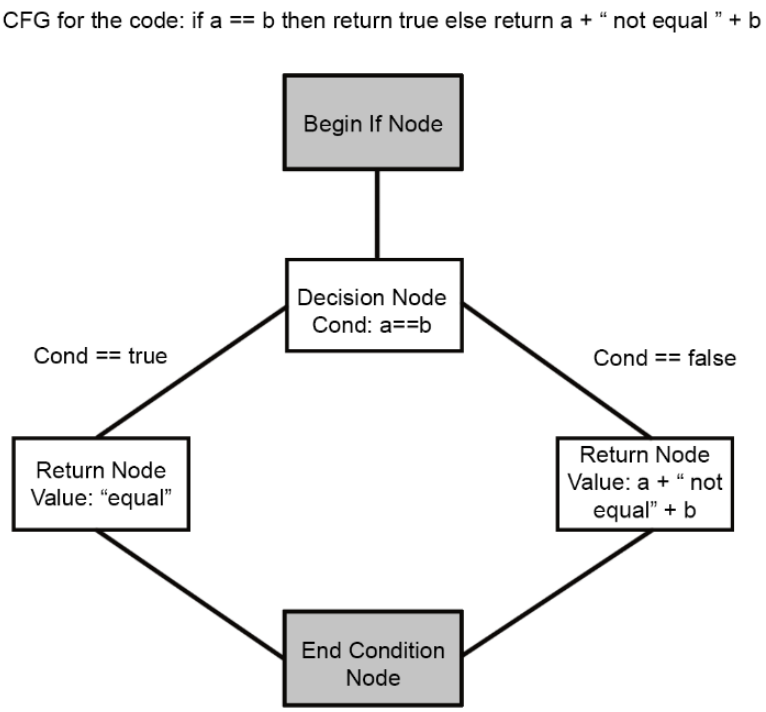
Đồ thị CFG bao gồm các đỉnh và cạnh. Các đỉnh đại diện cho các khối cơ bản của mã lệnh thực hiện một chuỗi các hoạt động mà không có bất kỳ thay đổi luồng điều khiển nào, trong khi các cạnh đại diện cho luồng điều khiển giữa các đỉnh. Một cạnh có thể đại diện cho một điểm quyết định trong mã lệnh, chẳng hạn như một câu lệnh if-else hoặc vòng lặp, hoặc nó có thể đại diện cho một dòng lệnh thực thi thẳng.

Trong nghiên cứu này, CFG sẽ được phân tích từ AST xây dựng dựa trên mã lệnh, các khối cơ bản của mã lệnh sẽ là các đỉnh nằm trên AST. Mỗi khối cơ bản được đại diện bởi một đỉnh trong biểu đồ CFG, và các câu lệnh điều khiển được đại diện bởi các cạnh.

CFG là một công cụ hữu ích cho phân tích chương trình, tối ưu hóa và kiểm tra. Nó có thể được sử dụng để xác định các lỗi tiềm ẩn, xác định thời gian thực thi của chương trình, tối ưu hóa chương trình bằng cách xác định các điểm quan trọng và tạo ra các trường hợp kiểm tra cho tất cả các đường thực thi có thể. CFG cũng có thể biểu thị rõ ràng cấu trúc và hành vi của mã, làm cho việc bảo trì và sửa đổi mã dễ dàng hơn.

Hình 2.3. dưới đây biểu diễn cho cấu trúc *CFG* của đoạn mã

if (a == b) then return "equal"; else return a + "not equal" + b;



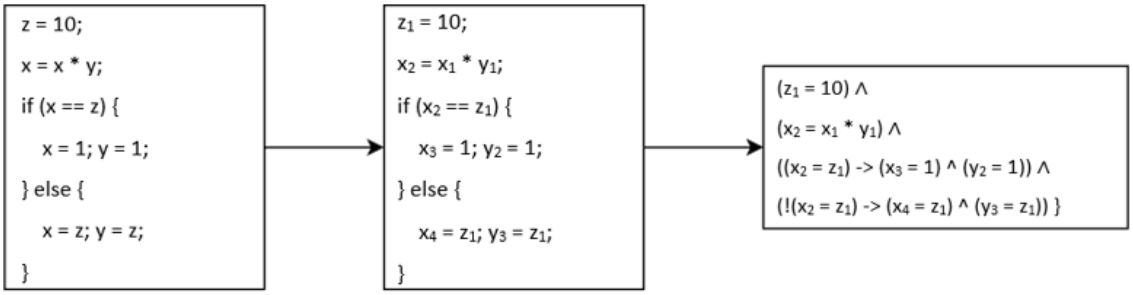
Hình 2.3. Trừu tượng hóa chương trình thành Control Flow Graph

2.2.3. Logic vị từ - FOL

Công thức logic vị từ (First Order Logic) là một biểu thức được tạo ra từ các biến mệnh đề (propositional variables) và các liên từ logic (logical connectives) như AND, OR, NOT, IF-THEN và IF-AND-ONLY-IF. Biểu thức này có thể được tính toán thành một giá trị logic duy nhất, có thể là true (đúng) hoặc false (sai), phụ thuộc vào giá trị của các biến mệnh đề. Biểu thức logic vị từ được sử dụng rộng rãi trong lý thuyết đại số Boolean và trong các ứng dụng như lập trình logic, mã hóa thông tin, xử lý ngôn ngữ tự nhiên, và trong các lĩnh vực khác của khoa học máy tính và toán học.

Trong ngữ cảnh trừu tượng hóa chương trình dựa trên thực thi tượng trưng, các biến mệnh đề là các phép gán giá trị của một biến hoặc biểu thức cho một biến còn các liên từ logic kết nối các biến mệnh đề thể hiện mối quan hệ giữa các phép gán giá trị đó.

Hình 2.4. dưới đây biểu diễn việc sinh ra các biểu thức logic vị từ từ một đoạn mã nguồn cho trước.



Hình 2.4. Trừu tượng hóa chương trình thành biểu thức logic vị từ.

2.3. Satisfiability Modulo Theories - SMT

Sau khi chương trình được trừu tượng hóa thành các biểu thức logic vị từ, Satisfiability Modulo Theories SMT sẽ được sử dụng để kiểm tra tính đúng đắn của tập các biểu thức vị từ cấp một đó.

Satisfiability Modulo Theories (SMT) là tên gọi của một phương pháp giải quyết bài toán logic được sử dụng trong lĩnh vực máy tính và khoa học máy tính. SMT kết hợp đồng thời việc tự động chứng minh định lý trong logic và kỹ thuật giải quyết bài toán tìm kiếm cục bộ để giải quyết các bài toán liên quan đến hệ thống các giả định.

Cụ thể, SMT tập trung vào việc xác định tính khả thi của một tập hợp các ràng buộc logic bằng cách sử dụng các giả định từ các lý thuyết khác nhau. Các lý thuyết này có thể bao gồm số học, giải tích, lý thuyết tập hợp, lý thuyết chuỗi, lý thuyết mảng,...

Bộ giải SMT giúp người dùng dễ dàng có thể hình dung được xu hướng lựa chọn luồng thực thi trong quá trình thực hiện chương trình dựa trên việc mô hình hóa các biểu thức điều kiện. Bằng cách thêm vào những điều kiện khác nhau, người dùng có thể sử dụng bộ giải SMT để kiểm tra những nhánh thực thi khác nhau của chương trình, đảm bảo vị trí lưu trữ nào đó luôn có một giá trị cụ thể và thỏa mãn các điều kiện đặt ra.

CHƯƠNG 3. XÂY DỰNG PHƯƠNG PHÁP KIỂM CHỨNG CHƯƠNG TRÌNH ĐA LUỒNG ACIO

Phương pháp kiểm chứng chương trình đa luồng ACIO được sử dụng trong khóa luận này được xây dựng dựa trên cơ sở trừu tượng hóa chương trình dựa trên thực thi tượng trưng kết hợp với đồ thị thứ tự sự kiện EOG. Từ cây AST và đồ thị CFG thu được, phương pháp ACIO sẽ tìm ra các sự kiện đọc/ghi các biến dùng chung (global variables) để xây dựng lên đồ thị EOG biểu diễn mối quan hệ thứ tự nhằm sinh ra các biểu thức logic vị từ mô tả cho các khả năng có thể xảy ra của luồng thực thi của chương trình.

3.1. Kiểm chứng dựa trên đồ thị thứ tự sự kiện EOG

Trong phần này, khóa luận sẽ trình bày về đối tượng nghiên cứu của phương pháp ACIO là các chương trình đa luồng đồng thời là các điều kiện đặt ra với các chương trình đa luồng đó. Ngoài ra, phần này cũng sẽ diễn giải các khái niệm, định nghĩa và công thức được sử dụng trong quá trình mô tả, xây dựng đồ thị thứ tự sự kiện EOG.

3.1.1. Chương trình đa luồng

Nghiên cứu này sẽ tập trung vào kiểm chứng chương trình C/C++ sử dụng Pthread, một trong những thư viện phổ biến nhất trong lập trình đa luồng. Thư viện Pthread sử dụng pthread_create(&t, &attrib, &f, &args) để bắt đầu chạy luồng t và pthread_join(t, &return) để tạm dừng luồng hiện tại cho đến khi luồng t hoàn tất thực thi.

Giả định rằng toàn bộ các biến trong P là nguyên tử, đồng thời toàn bộ các vòng lặp và đệ quy có chiều sâu là giới hạn (điều kiện cơ bản trong Bounded Model Checking, sẽ được trình bày trong mục **3.1.2. Bounded Model Checking**). Ngoài ra, việc mô hình hóa các cấu trúc dữ liệu phức tạp như là con trỏ, cấu trúc, mảng,... các phép tính và ký pháp phép tính phức tạp như là lũy thừa, logarit, ++, --,... và một số hàm của Pthread như pthread_mutex_lock, pthread_mutex_unlock.. sẽ bị bỏ qua trong nghiên cứu này.

Chương trình đa luồng P có V là tập các biến toàn cục. E là tập các sự kiện e biểu thị một hành động đọc hoặc ghi vào một biến toàn cục. Mỗi sự kiện $e \in E$ tương ứng với một phần tử biến var(e) $\in V$, một kiểu hành động type(e) (type(e) = read nếu sự kiện e biểu thị

cho hành động đọc và type(e) = write nếu sự kiện e biểu thị cho hành động ghi) và một giá trị value(e) là giá trị được thao tác trên var(e) [12].

Cho $\prec_P$ là một tập hợp các cặp gồm 2 sự kiện (e_i, e_j) với $e_i \subset E$ và $e_j \subset E$ ($hay \prec_P^0 \subset E \times E$). Mỗi một cặp sự kiện $(e_i, e_j) \in \prec_P$ ($hay\ e_i \prec_P e_j$) biểu thị rằng "sự kiện e_i sẽ xảy ra trước sự kiện ej trong luồng thực thi chương trình P" [12].

Cho $\prec_P^0$ là một tập hợp các cặp gồm 2 sự kiện (e_i, e_j) với $e_i \subset E$ và $e_j \subset E$ ($hay \prec_P^0 \subset E \times E$). Mỗi một cặp sự kiện $(e_i, e_j) \in \prec_P^0$ ($hay\ e_i \prec_P^0 e_j$) biểu thị rằng "sự kiện e_i sẽ xảy ra trước sự kiện e_j trong luồng thực thi chương trình P, dựa trên mã nguồn của P" [12]. Nếu $e_i \prec_P^0 e_j$ thì $e_i \prec_P e_j$, hay nói cách khác $\prec_P^0 \subset \prec_P$. Một cặp sự kiện $(e_i, e_j) \in \prec_P^0$ có thể được tạo ra từ các trường hợp sau:

- Với hai sự kiện e_i và e_j trong cùng một luồng, nếu e_i xảy ra trước e_j dựa trên thứ tự câu lệnh trong mã nguồn thì $e_i \prec_P^0 e_j$.
- Nếu *pthread_create()* được dùng để tạo một luồng t tại một điểm p trong luồng hiện tại, với mọi sự kiện e_i xảy ra trước p trong luồng hiện tại và mọi sự kiện e_j trong luồng t thì $e_i \prec_P^0 e_j$.
- Nếu *pthread_join()* được dùng để tạm dừng luồng hiện tại tại thời điểm p để đợi luồng t thực thi xong, với mọi sự kiện e_i trong luồng t và mọi sự kiện e_j xảy ra sau p trong luồng hiện tại thì $e_i \prec_P^0 e_j$.

Cho liên kết đọc-ghi $S(e_i, e_j)$ là một cặp gồm 2 sự kiện e_i và e_j biểu thị rằng "hành động đọc e_j sẽ đọc được giá trị ghi bởi hành động ghi e_i". Điều đó có nghĩa là $e_i \prec_P e_j$, type(e_j) == read, type(e_i) == write, var(e_j) == var(e_i) và value(e_j) == value(e_i) [12]. Thêm vào đó, nếu $S(e_i, e_j)$ tồn tại thì với mọi hành động ghi $e_k \neq ei$:

- $S(e_k, e_j)$ không tồn tại. Nói cách khác, nếu e_j đọc được giá trị ghi vào bởi e_i, e_j sẽ không thể đọc được giá trị ghi vào bởi e_k. Điều đó đảm bảo hành động đọc sẽ trả về kết quả là duy nhất.
- $e_k \prec_P e_i$ hoặc $e_j \prec_P e_k$. Nếu vẫn tồn tại e_k thỏa mãn $e_i \prec_P e_k \prec_P e_j$ thì khi đó, tuy rằng var(e_i) == var(e_j) = var nhưng value(e_i) ≠ value(e_j) do var đã bị ghi đè giá trị của hành động ghi e_k, hay value(e_k) = value(e_j). Do đó $S(e_i, e_j)$ là không tồn tại. Điều đó phủ định lại giả thuyết $S(e_i, e_j)$ được đặt ra ban đầu.

Cho liên kết đọc ghi $S(e_i, e_j)$. Liên kết đọc-ghi $S(e_i, e_j)$ sẽ được biểu diễn tượng trưng bằng một biến boolean gọi là chữ ký (signature). Tùy vào luồng thực thi cụ thể mà chữ ký của liên kết có thể nhận giá trị là true hoặc false. Chữ ký của liên kết đọc-ghi sẽ có giá trị là true nếu liên kết đọc-ghi đó có tồn tại trong luồng thực thi cụ thể thỏa mãn điều kiện đang xét, ngược lại sẽ có giá trị là false.

3.1.2. Bounded Model Checking

Bounded model checking [2] là một trong những kỹ thuật được áp dụng nhiều nhất để giảm thiểu vấn đề bùng nổ không gian trạng thái trong kiểm định chương trình đa luồng. Cho rằng hầu hết các lỗi có thể xác định với một số lượng giới hạn các vòng lặp, khi đó các vòng lặp/đệ quy có thể được trải phẳng thành các đoạn mã tuyến tính với độ dài giới hạn [8]. Thay vì liệt kê rõ ràng toàn bộ các xen kẽ luồng, BMC sử dụng ký hiệu tượng trưng để thể hiện cho việc mã hóa vấn đề kiểm chứng, thứ sẽ được giải bằng một bộ giải SMT.

Trong BMC, kiến trúc trừu tượng của một chương trình đa luồng thường được thể hiện dưới dạng

$$\alpha := \phi_{init} \wedge \rho \wedge \zeta \wedge \xi$$

trong đó, ϕ_{init} là các trạng thái khởi đầu của chương trình, ρ mã hóa từng luồng một cách riêng biệt với nhau, ζ biểu thị cho ràng buộc "mỗi hành vi đọc trên một biến v có thể đọc được giá trị được ghi bởi bất kì hành vi ghi trên biến v khả thi" và ξ biểu thị cho ràng buộc "với mỗi cặp $\langle w, r \rangle$, hành động đọc r đọc được giá trị của biến v được ghi bởi hành động ghi w, ngoài ra các hành động ghi khác nằm giữa chúng không có bất kì hành động ghi trên biến v nào" [1].

3.1.3. Đồ thị thứ tự sự kiện EOG

3.1.3.1. Đồ thị thứ tự sự kiện tổng quát

Đồ thị thứ tự sự kiện tổng quát G của một chương trình là biểu diễn đồ thị của kiến trúc trừu tượng α không bao gồm điều kiện khởi đầu và các ràng buộc về liên kết đọc-ghi của chương trình đó:

$$G = graph(\rho) = \langle E, \pounds_n^0 \rangle.$$

với E là tập các đỉnh đọc hoặc ghi tương ứng với các sự kiện đọc hoặc ghi xảy ra trong α và $\pounds_n^0$ là tập các cạnh thể hiện mối quan hệ thứ tự tuyến tính giữa các đỉnh tương ứng với

các ràng buộc trong ρ. Mỗi một đỉnh trong G được biểu diễn bằng một nút trắng hoặc đen tương ứng với sự kiện ghi hoặc đọc. Mỗi một cạnh trong G được biểu diễn bằng một đường nối kép đi từ trên xuống dưới.

Việc xây dựng đồ thị thứ tự sự kiện tổng quát G yêu cầu chương trình phải được chuyển về dạng tập hợp các câu lệnh static single assignment (SSA), nghĩa là các biến trong chương trình cần được đổi tên thành các trạng thái giá trị sử dụng một lần. Yêu cầu này sẽ đảm bảo mỗi đỉnh và mỗi cạnh được thêm vào trong G là duy nhất.

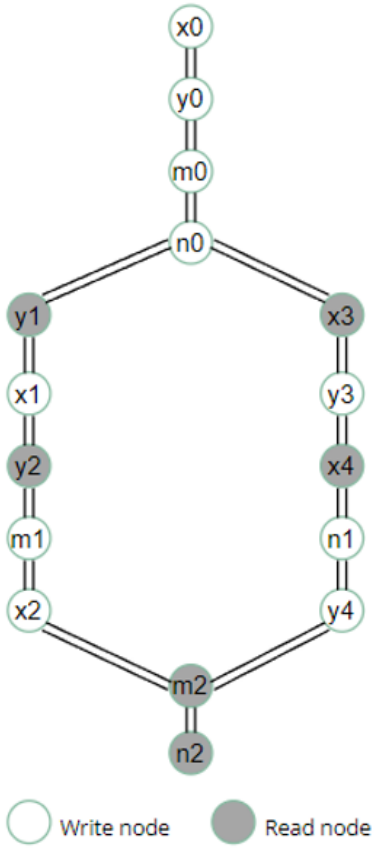
Hình 3.1. dưới đây mô tả phương pháp chuyển đổi mã nguồn thành mã SSA rồi từ đó xây dựng nên đồ thị thứ tự sự kiện tổng quát của một chương trình C hoàn chỉnh.

```
int x = 1, y = 1, m = 0, n = 0;
void* thr1(void * arg) {
    x = y + 1;
    m = y;
    x = 0;
}
void* thr2(void * arg) {
    y = x + 1;
    n = x;
    y = 0;
}
void main() {
    pthread_t t1, t2;
    pthread_create(&t1, 0, thr1, 0);
    pthread_create(&t2, 0, thr2, 0);
    pthread_join(t1, 0);
    pthread_join(t2, 0);
    assert (!(m == 2 && n == 3));
}
```

(1) Chương trình gốc

```
int x0 = 1, y0 = 1, m0 = 0, n0 = 0;
void* thr1(void * arg) {
    x1 = y1 + 1;
    m1 = y2 ;
    x2 = 0;
}
void* thr2(void * arg) {
    y3 = x3 + 1;
    n1 = x4 ;
    y4 = 0;
}
void main() {
    pthread_t t1, t2;
    pthread_create(&t1, 0, thr1, 0);
    pthread_create(&t2, 0, thr2, 0);
    pthread_join(t1, 0);
    pthread_join(t2, 0);
    assert (!(m2 == 2 && n2 == 3));
}
```

(2) Mã SSA của chương trình



Hình 3.1. Xây dựng đồ thị thứ tự sự kiện tổng quát

3.1.3.2. Phản ví dụ

Phản ví dụ π của kiến trúc trừu tượng α tương ứng với điều kiện kiểm chứng ϕ_{verify} là một trình tự thực thi $£_\pi$ của α vi phạm ϕ_{verify}, hay nói cách khác là tập hợp các phép gán giá trị cho các biến trong α thỏa mãn ϕ_{err} với ϕ_{err} là các trạng thái lỗi: $\phi_{err} := \neg\, \phi_{verify}$. Trình tự thực thi đó được mô tả bằng cách thiết lập một tập các mối quan hệ đọc-ghi giữa các hành động đọc và ghi xảy ra trong α.

$$\pi := \alpha \wedge £_\pi \wedge \phi_{err} \quad \Leftrightarrow \quad \pi := \phi_{init} \wedge \rho \wedge \zeta \wedge \xi \wedge £_\pi \wedge \phi_{err}$$

Một chương trình có thể không có, có một hoặc có nhiều phản ví dụ, tùy vào số lượng trình tự thực thi $£_\pi$ vi phạm điều kiện kiểm chứng mà nó sở hữu.

3.1.3.3. Đồ thị thứ tự sự kiện của phản ví dụ

Đồ thị thứ tự sự kiện G_π của phản ví dụ π là sự kết hợp của đồ thị thứ tự sự kiện tổng quát G và biểu diễn đồ thị μ_π của trình tự thực thi $£_\pi$ được mô tả trong π.

$$G_\pi = G \wedge £_\pi = \langle E, £_n^0, \mu_\pi \rangle.$$

G_π được biểu diễn bằng cách biểu diễn đồng thời G và $£_\pi$ trên cùng một đồ thị.

μ_π là tập hợp các cạnh thể hiện mối quan hệ đọc-ghi giữa một đỉnh ghi n_1 và một đỉnh đọc n_2 tương ứng đảm bảo rằng $n_1 \prec_P n_2$, var(n_1) = var(n_2), type(n_1) = write, type(n_2) = read, value(n_1) = value(n_2), giữa chúng không có bất kì đỉnh nào ghi đè giá trị và không có bất kì cạnh nào khác được định nghĩa bởi μ_π có đỉnh thứ hai là n_2. Mỗi một cạnh trong μ_π được biểu diễn bằng một đường mũi tên nối từ đỉnh ghi xuống đỉnh đọc.

3.2. Phương pháp biểu diễn ràng buộc

Với đồ thị thứ tự sự kiện $G_\pi = \langle E, £_n^0, \mu_\pi \rangle$, các ràng buộc sẽ được xây dựng dựa trên các cạnh của G_π, hay nói cách khác là dựa trên các quan hệ tuyến tính $£_n^0$ và các liên kết đọc-ghi μ_π. Mỗi ràng buộc sẽ có phương pháp phân tích và xây dựng khác nhau, dựa trên mục tiêu của các ràng buộc đó trong tập biểu diễn trừu tượng. Do đó, nghiên cứu này phân chia các ràng buộc thành ba nhóm chính dựa trên vai trò của chúng trong việc biểu diễn các sự kiện và mối quan hệ.

- Ràng buộc độc lập trên các luồng: Biểu diễn các sự kiện gán giá trị.
- Ràng buộc cho các biến đọc/ghi dữ liệu: Biểu diễn các sự kiện đọc giá trị.
- Ràng buộc đảm bảo thứ tự thực thi: Biểu diễn mối quan hệ giữa các sự kiện đọc/ghi.

3.2.1. Ràng buộc độc lập trên các luồng

Ràng buộc độc lập trên các luồng (gọi tắt là ràng buộc độc lập) là ràng buộc được xây dựng dựa trên quá trình lấy các giá trị từ tập các biến r_i, thực hiện tính toán trên tập giá trị này rồi ghi kết quả vào một biến w_i nào đó, hay nói ngắn gọn là dựa trên các phép gán giá trị. Ràng buộc này được gọi là độc lập giữa các luồng vì mỗi một phép gán giá trị chỉ nằm trong một luồng duy nhất, không ảnh hưởng tới các luồng còn lại.

Với mỗi một thao tác ghi sẽ có một phép toán đầu vào đi kèm để xác định giá trị được ghi. Phép toán đầu vào này chứa thao tác đọc giá trị của một số biến nào đó (hoặc có thể không chứa thao tác đọc nếu phép toán chỉ bao gồm các số). Khi biểu diễn thành các đỉnh đọc/ghi trên EOG, mỗi đỉnh ghi trên EOG sẽ có liên hệ với một tập các đỉnh đọc được sử dụng trong phép toán đầu vào tương ứng. Từ mối liên hệ đó, ràng buộc độc lập trên các luồng được sinh ra sử dụng quan hệ ngang bằng giữa giá trị của biến ghi và giá trị của phép toán đầu vào.

Algorithm 1 được trình bày dưới đây mô tả cho thuật toán sinh ràng buộc độc lập trên các luồng của chương trình.

Algorithm 1. Thuật toán sinh ràng buộc độc lập trên các luồng

Input: EOG $G_\pi := \langle E, £_n^0, \mu_\pi \rangle$.

Output: Tập các ràng buộc độc lập trên các luồng của G_π.

Cho E_{write} := Tập các đỉnh ghi của G_π.

Cho $\beta_{independent}$ là tập rỗng dùng để lưu các ràng buộc độc lập.

foreach (đỉnh ghi W trong E_{write}) **do**

    Cho R_s := tập các đỉnh đọc tương ứng với W.

    Cho C_{Rs} := phép tính của indexedVar(R_s), tương ứng với W.

    $\beta_{independent}$.add(indexedVar(W) = C_{Rs}).

end.

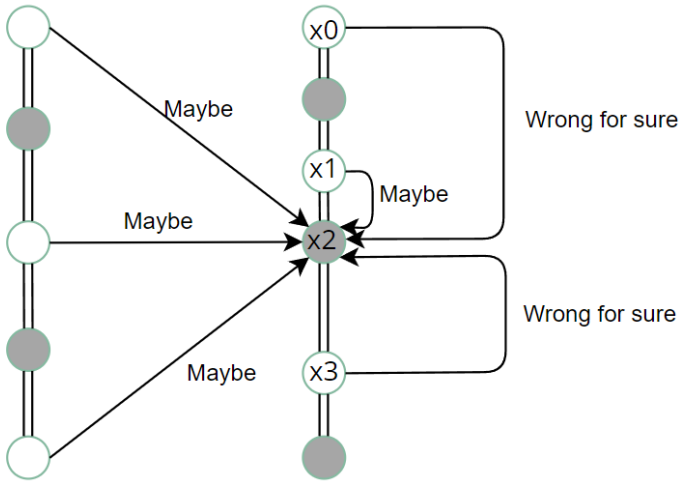
return $\beta_{independent}$.

3.2.2. Ràng buộc cho các biến đọc/ghi dữ liệu

Ràng buộc cho các biến đọc/ghi (gọi tắt là ràng buộc đọc-ghi) là ràng buộc được tạo nên bởi một hành động đọc và nhiều hành động ghi. Ràng buộc đọc-ghi biểu thị rằng với mỗi một hành động đọc, nó có thể đọc được giá trị ghi bởi bất kì các hành động ghi khả thi, đồng thời chỉ có thể đọc được duy nhất giá trị ghi bởi hành động ghi đó. Điều đó có nghĩa là ràng buộc đọc ghi đảm bảo hành động đọc đọc được giá trị là duy nhất.

Với mỗi đỉnh đọc trong EOG sẽ có một tập các đỉnh ghi tương ứng (thao tác lên cùng một biến và có thể xảy ra trước đỉnh đọc). Đỉnh đọc này sẽ liên kết với các đỉnh ghi khả thi để tạo thành các liên kết đọc-ghi $S(e_i, e_j)$. Các liên kết này cần đảm bảo là khả thi. Sau đó, từ tập các liên kết vừa sinh ra, cần đảm bảo rằng trong đó có một và chỉ một liên kết có tồn tại, dựa vào tính chất duy nhất trong mối quan hệ giữa đỉnh đọc và đỉnh ghi trong liên kết đọc-ghi.

Trong *Hình 3.2.* , đỉnh đọc x_2 cùng thao tác lên một biến với đỉnh ghi x_0, x_1, x_3 và một vài đỉnh trong các luồng khác. Tuy nhiên, khi thực thi hành động đọc x_2 đọc giá trị của biến x thì giá trị x_0 và x_3 đều không tồn tại vì x_1 đã ghi đè giá trị của x_0 trong khi x_3 chưa được thực thi, có nghĩa là x_2 không thể liên kết với x_0 hoặc x_3 để tạo thành liên kết đọc ghi. Do đó, $S(x_0, x_2)$ và $S(x_3, x_2)$ là không tồn tại và sẽ không được sử dụng để sinh ràng buộc cho các biến đọc/ghi dữ liệu.



Hình 3.2. Một số hành động ghi tương ứng với hành động đọc

Algorithm 2 được trình bày dưới đây mô tả cho thuật toán sinh ràng buộc cho các biến đọc/ghi dữ liệu của chương trình.

Algorithm 2. Thuật toán sinh ràng buộc cho các biến đọc/ghi dữ liệu

Input: EOG $G_\pi := \langle E, \pounds_n^0, \mu_\pi \rangle$.

Output: Tập ràng buộc của các biến đọc/ghi dữ liệu của G_π.

Cho E_{read} := tập các đỉnh đọc của G_π

Cho β_{rw} là tập rỗng, dùng để lưu trữ các ràng buộc.

foreach (đỉnh đọc R trong E_{read}) **do**

    Cho W_s := tập các đỉnh ghi tương ứng với R và có thể thực thi trước R.

    Cho RW_s là tập rỗng, dùng để lưu trữ các ràng buộc.

    foreach (đỉnh ghi W trong W_s) **do**

        Cho isValid := true là trạng thái khởi đầu của liên kết đọc-ghi R-W.

        foreach (đỉnh ghi $W_{other} \neq W$ trong W_s) **do**

            if ($W \prec_P^0 W_{other}$ và $W_{other} \prec_P^0 R$) **do**

                isValid := false.

                break.

            end.

        end.

        if (!isValid) **do**

            continue.

        end.

        Cho signature := mã định danh của liên kết R-W.

        Cho constraint := (signature ⇒ (indexedVar(R) == indexedVar(W)).

        RW_s.add(constraint).

        β_{rw}.add(constraint).

    end.

    Cho atLeastOne := ít nhất một ràng buộc trong RW_s là đúng.

    Cho atMostOne := nhiều nhất một ràng buộc trong RW_s là đúng.

    β_{rw}.add(atLeastOne).

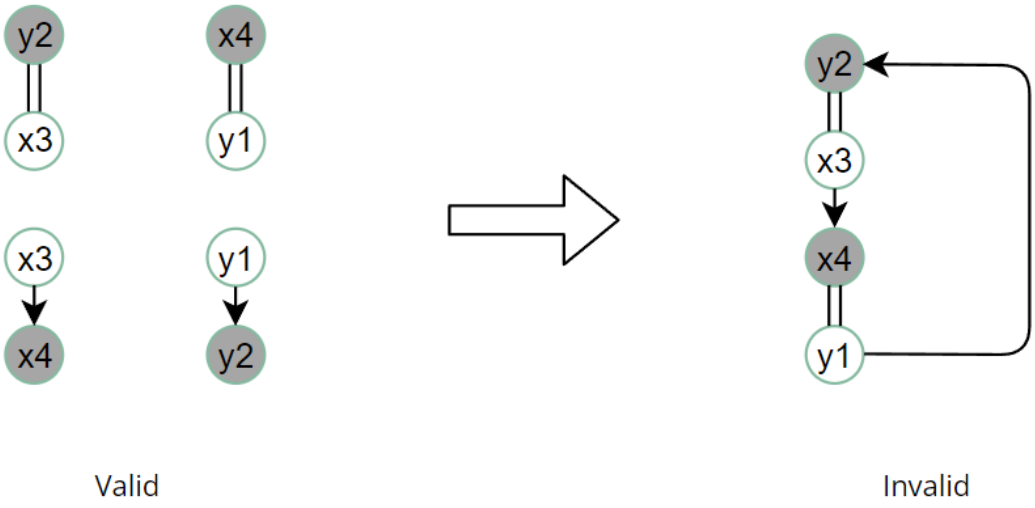
    β_{rw}.add(atMostOne).

end.

return β_{rw}.

3.2.3. Ràng buộc đảm bảo thứ tự thực thi

Có thể thấy rằng ràng buộc cho các biến đọc/ghi dữ liệu chỉ quan tâm đến ràng buộc nội hàm giữa các liên kết đọc-ghi riêng biệt có cùng đỉnh đọc mà không quan tâm đến quan hệ giữa các liên kết đọc-ghi không cùng đỉnh đọc và ràng buộc độc lập trên từng luồng. Bỏ qua các quan hệ đó có thể dẫn tới việc một đỉnh đọc có thể đọc được đỉnh ghi chưa từng xảy ra hoặc đã bị ghi đè bởi đỉnh ghi khác.



Hình 3.3. Đỉnh đọc đọc được giá trị của một đỉnh ghi chưa xảy ra

Xét ví dụ trong *Hình 3.3.*, cho rằng hành động Read x_4 có thể đọc được giá trị của hành động Write x_3, hành động Read y_2 có thể đọc được giá trị của hành động Write y_1. Tuy nhiên khi kết hợp với các ràng buộc độc lập trên các luồng vào, hành động Read y_2 lúc này sẽ xảy ra trước hành động Write y_1. Do đó y_1 không thể đọc được giá trị của y_2, liên kết đọc ghi Read y_2 – Write y_1 không tồn tại.

Để khắc phục thiếu sót này, ràng buộc đảm bảo thứ tự thực thi được thêm vào trong quá trình biểu diễn ràng buộc. Ràng buộc đảm bảo thứ tự thực thi phát biểu rằng sự tồn tại của liên kết đọc-ghi này sẽ dẫn đến sự không tồn tại của một số liên kết đọc-ghi khác để đảm bảo rằng các liên kết đọc-ghi không tạo ra xung đột đồng thời thỏa mãn ràng buộc độc lập trên từng luồng.

3.2.3.1. Thuật toán sinh ràng buộc đảm bảo thứ tự thực thi.

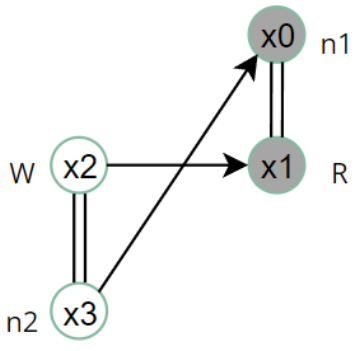
Về tổng quan, ràng buộc này đưa ra các điều kiện để các liên kết đọc-ghi là khả thi trong mối quan hệ với toàn bộ các liên kết đọc ghi khác không cùng đỉnh đọc (có thể hiểu là . Với một liên kết đọc-ghi được tạo ra bởi một đỉnh đọc *R* và một đỉnh ghi *W* cho trước, thuật toán sinh ràng buộc sẽ xem xét 4 nhóm đỉnh dựa trên hai thuật toán:

- *Algorithm 3*: xem xét tập hợp 1 là các đỉnh xảy ra trước *R* và các đỉnh xảy ra sau *W*.
- *Algorithm 4*: xem xét tập hợp 2 là các đỉnh xảy ra sau *R* và các đỉnh xảy ra trước *W*.

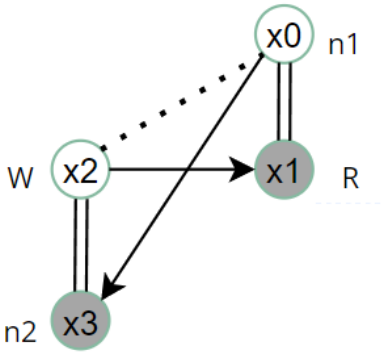
3.2.3.1.1. Thuật toán sinh ràng buộc đảm bảo thứ tự thực thi thứ nhất – Algorithm 3

Xét liên kết đọc-ghi được tạo ra bởi từng cặp gồm 2 đỉnh, đỉnh thứ nhất n_1 được chọn trong tập các đỉnh xảy ra trước đỉnh đọc *R* và đỉnh thứ hai n_2 được chọn trong tập các đỉnh xảy ra sau đỉnh ghi *W*.

- Nếu type(n_1) = type(n_2) thì không có xung đột nào do không thể tạo liên kết đọc-ghi giữa hai đỉnh cùng loại.
- Nếu *W* là đỉnh ghi x_2, *R* là đỉnh đọc x_1, n_1 là đỉnh đọc x_0, n_2 là đỉnh ghi x_3, $var(n_1) = var(n_2) = var(R) = var(W) = x$. Khi đó, $R \prec_P W \prec_P^o n_2$. Kết hợp với điều kiện $n1 \prec_P^o R$ sinh ra ràng buộc $n1 \prec_P^o R \prec_P W \prec_P^o n_2$. Lúc này, $S(n_2, n_1)$ là không tồn tại vì n_2 chưa xảy ra khi n_1 được thực thi (*Hình 3.4.*).
- Nếu *W* là đỉnh ghi x_2, *R* là đỉnh đọc x_1, n_1 là đỉnh ghi x_0, n_2 là đỉnh đọc x_3, $var(n_1) = var(n_2) = var(R) = var(W) = x$. Khi đó, $S(W, R) \rightarrow n_1 \prec_P W \prec_P^o R$. Kết hợp với điều kiện $W \prec_P^o n_2$ sinh ra ràng buộc $n_1 \prec_P W \prec_P^o n_2$. Lúc này, $S(n_1, n_2)$ là không tồn tại do giữa n_1 và n_2 có hành động *W* ghi giá trị vào biến *x* (*Hình 3.5.*).



Hình 3.4. Xung đột thứ nhất của tập hợp 1



Hình 3.5. Xung đột thứ hai của tập hợp 1

Algorithm 3. Thuật toán sinh ràng buộc đảm bảo thứ tự thực thi cho tập hợp 1.

Input: EOG $G_\pi := \langle E, £_n^0, \mu_\pi \rangle$, liên kết đọc-ghi S(W, R).

Output: Tập ràng buộc đảm bảo thứ tự thực thi của tập hợp 1 tương ứng với liên kết S(W, R) trong G_π.

Cho E_{pR} := tập các đỉnh xảy ra trước R.

Cho E_{aW} := tập các đỉnh xảy ra sau W.

Cho $\beta_{guaranteed\_1}$ là tập rỗng dùng để lưu trữ các ràng buộc.

foreach (đỉnh N_1 trong E_{pR}) **do**

    foreach (đỉnh N_2 trong E_{aW}) **do**

        if (N_1 là đỉnh đọc) **do**

            if (N_2 là đỉnh ghi) **do**

                if (var(N_1) = var(N_2) = var(W) = var(R)) **do**

                    $\beta_{guaranteed\_1}$.add(¬S(N_2, N_1)).

                end.

            end.

        else if (N_1 là đỉnh ghi) **do**

            if (N_2 là đỉnh đọc) **do**

                if (var(N_1) = var(N_2) = var(W) = var(R)) **do**

                    $\beta_{guaranteed\_1}$.add(¬S(N_1, N_2)).

                end.

            end.

        end.

    end.

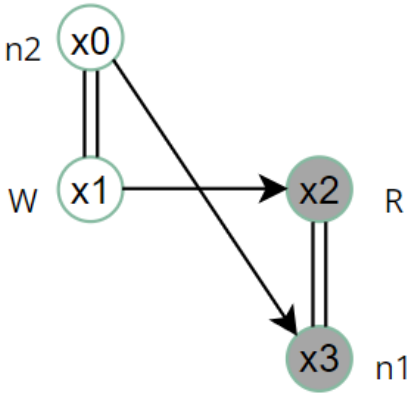
end.

return $\beta_{guaranteed\_1}$.

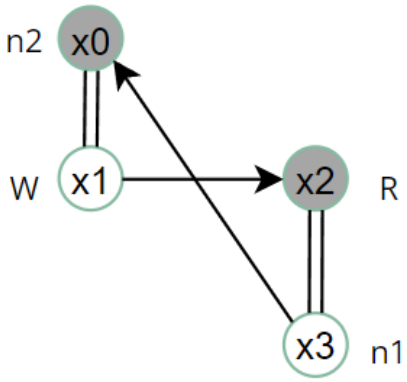
3.2.3.1.2. Thuật toán sinh ràng buộc đảm bảo thứ tự thực thi thứ hai – Algorithm 4

Xét liên kết đọc-ghi được tạo ra bởi từng cặp gồm 2 đỉnh, đỉnh thứ nhất n_1 được chọn trong tập các đỉnh xảy ra sau đỉnh đọc R và đỉnh thứ hai n_2 được chọn trong tập các đỉnh xảy ra trước đỉnh ghi W.

- Nếu type(n_1) = type(n_2) thì không có xung đột nào do không thể tạo liên kết đọc-ghi giữa hai đỉnh cùng loại.
- Nếu W là đỉnh ghi x_1, R là đỉnh đọc x_2, n_1 là đỉnh đọc x_3, n_2 là đỉnh ghi x_0, $var(n_1) = var(n_2) = var(R) = var(W) = x$. Khi đó, từ định nghĩa của liên kết đọc-ghi, $S(W, R) \rightarrow n_2 \prec_P^o W \prec_P R$. Kết hợp với điều kiện $R \prec_P^o n_1$ sinh ra ràng buộc $n_2 \prec_P^o W \prec_P R \prec_P^o n_1$. Lúc này, $S(n_2, n_1)$ là không tồn tại vì giữa n_1 và n_2 có đỉnh W ghi giá trị vào biến x (*Hình 3.6.*).
- Nếu W là đỉnh ghi x_1, R là đỉnh đọc x_2, n_1 là đỉnh ghi x_3, n_2 là đỉnh đọc x_0, $var(n_1) = var(n_2) = x$. Khi đó, $n_2 \prec_P^o W \prec_P R \prec_P^o n_1$. Lúc này, $S(n_1, n_2)$ là không tồn tại do n_1 chưa xảy ra khi n_2 được thực thi (*Hình 3.7.*).



Hình 3.6. Xung đột thứ nhất của tập hợp 2



Hình 3.7. Xung đột thứ hai của tập hợp 2

Algorithm 4. Thuật toán sinh ràng buộc đảm bảo thứ tự thực thi cho tập hợp 2.

Input: EOG $G_\pi := \langle E, £_n^0, \mu_\pi \rangle$, liên kết đọc-ghi S(W, R).

Output: Tập ràng buộc đảm bảo thứ tự thực thi của tập hợp 2 tương ứng với liên kết S(W, R) trong G_π.

Cho E_{aR} := tập các đỉnh xảy ra sau R.

Cho E_{pW} := tập các đỉnh xảy ra trước W.

Cho $\beta_{guaranteed\_2}$ là tập rỗng dùng để lưu trữ các ràng buộc.

```
foreach (đỉnh N1 trong EaR) do
    foreach (đỉnh N2 trong EpW) do
        if (N1 là đỉnh đọc) do
            if (N2 là đỉnh ghi) do
                if (var(N1) = var(N2) = var(W) = var(R)) do
                    βguaranteed_2.add(¬S(N2, N1)).
                end.
            end.
        else if (N1 là đỉnh ghi) do
            if (N2 là đỉnh đọc) do
                if (var(N1) = var(N2)) do
                    βguaranteed_2.add(¬S(N1, N2)).
                end.
            end.
        end.
    end.
end.
return βguaranteed_2.
```

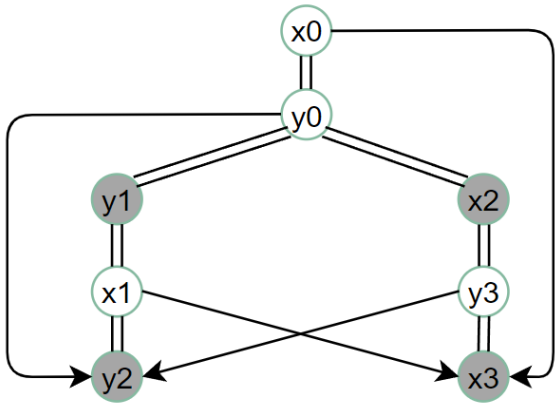
3.2.3.2. Thuật toán sinh ràng buộc suy diễn đảm bảo thứ tự thực thi.

Xét ví dụ trong

Hình 3.8., với liên kết đọc-ghi $S(y_0, y_2)$ và $S(x_0, x_3)$, thuật toán sinh ràng buộc đảm bảo thứ tự thực thi sẽ sinh ra các ràng buộc sau:

$S(y_0, y_2) \rightarrow \neg S(y_3, y_1)$

$S(x_0, x_3) \rightarrow \neg S(x_1, x_2)$



Hình 3.8. EOG không được bao phủ hết bằng thuật toán sinh ràng buộc đảm bảo thứ tự thực thi

Có thể thấy rằng, dựa trên thuật toán sinh ràng buộc đảm bảo thứ tự thực thi, hai liên kết này là độc lập, không ràng buộc với nhau. Tuy nhiên, giả sử $S(y_0, y_2)$ có tồn tại, khi đó, $y_0 \prec_P y_2$. y_3 là hành động ghi lên biến y, $y_0 \prec_P^0 y_3$ nên $y_0 \prec_P y_2 \prec_P y_3$ (theo định nghĩa của liên kết đọc-ghi). Lại có $y_3 \prec_P^0 x_3$ nên $y_2 \prec_P y_3 \prec_P^0 x_3$. Dựa vào đồ thị EOG, $x_0 \prec_P^0 x_1 \prec_P^0 y_2$ nên $x_0 \prec_P^0 x_1 \prec_P x_3$. Do x_1 ghi đè lên x_0 trước khi x_3 được thực thi nên $S(x_0, x_3)$ không tồn tại, hay $S(y_0, y_2) \rightarrow \neg\ S(x_0, x_3)$. Điều này chứng minh rằng thuật toán sinh ràng buộc đảm bảo thứ tự thực thi được trình bày ở các phần trên là chưa đủ để mã hóa toàn bộ các quan hệ giữa các sự kiện đọc/ghi.

Nguyên nhân của sự thiếu sót này là bởi vì điều kiện tiên quyết của thuật toán sinh ràng buộc đảm bảo thứ tự thực thi là sự tồn tại của liên kết đọc-ghi tương ứng. Liên kết đọc-ghi này lại là cơ sở để suy diễn thứ tự trước sau của các hành động đọc/ghi có quan hệ thứ tự với hai đầu mút của nó. Khi kết hợp liên kết đọc ghi đó với các điều kiện ban đầu, một số liên kết đọc-ghi khác lúc này, dựa vào phép suy diễn, có thể sẽ trở nên không khả

thi. Bởi vậy, thuật toán sinh ràng buộc bổ sung đảm bảo thứ tự thực thi được thêm vào để suy diễn và loại bỏ đi các liên kết đọc-ghi không khả thi mỗi khi một liên kết đọc-ghi được cho rằng là có tồn tại.

Ràng buộc bổ sung đảm bảo thứ tự thực thi phát biểu rằng với mỗi liên kết đọc-ghi *S(w,r)* thao tác lên một biến *v, w* thuộc luồng t_1, *r* thuộc luồng t_2, cho n_1 là đỉnh ghi cùng thao tác lên biến *v*, diễn ra sau *w* trong luồng t_1, khi đó toàn bộ các đỉnh đọc r_i xảy ra sau n_1 không thể đọc được giá trị ghi bởi các đỉnh ghi w_i tương ứng nằm trước *r* trong luồng t_2 ngoại trừ $w_{nearest}$ là đỉnh ghi tương ứng gần nhất so với *r*.

Ràng buộc này được chứng minh như sau:

Cho (S(w, r) and $w \prec_P^0 n_1$)

$$\rightarrow w \prec_P r \prec_P n_1$$

Có ($r \prec_P n_1$ and $n_1 \prec_P^0 r_i$)

$$\rightarrow r \prec_P r_i$$

Cho $w_1, w_2, ..., w_n$ là đỉnh ghi tương ứng với r_i thỏa mãn $w_1 \prec_P^0 ... \prec_P^0 w_n \prec_P^0 r$

Có ($r \prec_P r_i$, $w_1 \prec_P^0 ... \prec_P^0 w_n \prec_P^0 r$)

$$\rightarrow (w_1 \prec_P^0 ... \prec_P^0 w_n \prec_P^0 r \prec_P r_i)$$

Để hai hành động đọc và ghi tương ứng có thể tạo thành một liên kết đọc-ghi thì giữa chúng không được phép tồn tại bất kì hành động ghi tương ứng nào. Do đó từ suy diễn trên, r_i không tạo được bất kì liên kết đọc-ghi nào với w_j ($0 \leq j < n$), hay nói cách khác là với mọi $0 \leq j < n$, $S(w_j, r_i)$ là không tồn tại.

Algorithm 5 được trình bày dưới đây mô tả cho thuật toán sinh ràng buộc suy diễn đảm bảo thứ tự thực thi của chương trình.

Algorithm 5. Thuật toán sinh ràng buộc suy diễn đảm bảo thứ tự thực thi.

Input: EOG $G_\pi := \langle E, £_n^0, \mu_\pi \rangle$, liên kết đọc-ghi S(W, R).

Output: Tập ràng buộc suy diễn đảm bảo thứ tự thực thi của S(W, R) trong G_π.

Cho E_{fWw} := toàn bộ các đỉnh ghi xảy ra sau W, không nằm trong cùng một luồng nhưng có cùng var với R.

Cho E_{pRw} := toàn bộ các đỉnh ghi xảy ra trước R.

Cho $\beta_{deductive}$ là tập rỗng dùng để lưu các ràng buộc.

foreach (đỉnh ghi n_1 trong E_{fWw}) **do**

    Cho E_{dr} := các đỉnh đọc xảy ra sau n_1.

    foreach (đỉnh đọc n_2 trong E_{dr}) **do**

        Cho $E_{nearest}$:= đỉnh gần R nhất trong E_{pRw} mà có cùng var n_2.

        foreach (đỉnh ghi n_3 trong E_{pRw}) **do**

            if (n_3 != $E_{nearest}$ và var(n_3) == var(n_2)) **do**

                $\beta_{deductive}$ .add(¬S(n_3, n_2)).

            end.

        end.

    end.

end.

return $\beta_{deductive}$.

3.2.4. Minh họa phương pháp

Lấy chương trình trong *Hình 3.1.* làm ví dụ minh họa để xây dựng tập các ràng buộc cho bộ giải SMT. Các bước xây dựng ràng buộc được thực hiện theo trình tự như sau.

3.2.4.1. Xây dựng ràng buộc độc lập trên các luồng

Với E_{write} là tập các hành động ghi trong đồ thị EOG:

$E_{write} :=$ $\{ x_0, y_0, m_0, n_0, x_1, m_1, x_2, y_3, n_1, y_4 \}$

Mỗi đỉnh ghi trong E_{write} có biểu thức tương ứng là:

$x_0 \rightarrow x_0 = 1$ $\quad n_0 \rightarrow n_0 = 0$ $\quad x_2 \rightarrow x_2 = 0$ $\quad n_1 \rightarrow n_1 = x_4$

$y_0 \rightarrow y_0 = 1$ $\quad x_1 \rightarrow x_1 = y_1 + 1$ $\quad y_3 \rightarrow y_3 = x_3 + 1$ $\quad y_4 \rightarrow y_4 = 0$

$m_0 \rightarrow m_0 = 0$ $\quad m_1 \rightarrow m_1 = y_2$

Biểu diễn các biểu thức trên thành ràng buộc tương ứng, thu được tập ràng buộc độc lập:

$$\beta_{independent} := (x_0 = 1) \wedge (y_0 = 1) \wedge (m_0 = 0) \wedge (n_0 = 0)$$
$$\wedge (x_1 = y_1 + 1) \wedge (m_1 = y_2) \wedge (x_2 = 0)$$
$$\wedge (y_3 = x_3 + 1) \wedge (n_1 = x_4) \wedge (y_4 = 0)$$

3.2.4.2. Xây dựng ràng buộc cho các biến đọc/ghi dữ liệu

Với E_{read} là tập các hành động đọc trong đồ thị EOG:

$E_{read} :=$ $\{ y_1, y_2, x_3, x_4, m_2, n_2 \}$

Xét lần lượt các sự kiện đọc trong E_{read} thu được các ràng buộc đọc-ghi của từng sự kiện:

- y_1 có thể đọc được giá trị của $\{ y_0, y_3, y_4 \}$

$$\beta_{rw\_y_1} := (S_{y_0, y_1} \Rightarrow y_1 = y_0) \wedge (S_{y_3, y_1} \Rightarrow y_1 = y_3) \wedge (S_{y_4, y_1} \Rightarrow y_1 = y_4)$$
$$\wedge \text{atLeastOne}(S_{y_0, y_1}, S_{y_3, y_1}, S_{y_4, y_1})$$
$$\wedge \text{atMostOne}(S_{y_0, y_1}, E_{y_3, y_1}, S_{y_4, y_1})$$

- y_2 có thể đọc được giá trị của $\{ y_0, y_3, y_4 \}$

$$\beta_{rw\_y_2} := (S_{y_0, y_2} \Rightarrow y_2 = y_0) \wedge (S_{y_3, y_2} \Rightarrow y_2 = y_3) \wedge (S_{y_4, y_2} \Rightarrow y_2 = y_4)$$
$$\wedge \text{atLeastOne}(S_{y_0, y_2}, S_{y_3, y_2}, S_{y_4, y_2})$$

$\wedge$ atMostOne(S_{y_0, y_2}, S_{y_3, y_2}, S_{y_4, y_2})

- x_3 có thể đọc được giá trị của { x_0, x_1, x_2 }

  $\beta_{rw\_x_3}$:= $(S_{x_0, x_3} \Rightarrow x_3 = x_0) \wedge (S_{x_1, x_3} \Rightarrow x_3 = x_1) \wedge (S_{x_2, x_3} \Rightarrow x_3 = x_2)$

  $\wedge$ atLeastOne(S_{x_0, x_3}, S_{x_1, x_3}, S_{x_2, x_3})

  $\wedge$ atMostOne(S_{x_0, x_3}, S_{x_1, x_3}, S_{x_2, x_3})

- x_4 có thể đọc được giá trị của { x_0, x_1, x_2 }

  $\beta_{rw\_x_4}$:= $(S_{x_0, x_4} \Rightarrow x_4 = x_0) \wedge (S_{x_1, x_4} \Rightarrow x_4 = x_1) \wedge (S_{x_2, x_4} \Rightarrow x_4 = x_2)$

  $\wedge$ atLeastOne(S_{x_0, x_4}, S_{x_1, x_4}, S_{x_2, x_4})

  $\wedge$ atMostOne(S_{x_0, x_4}, S_{x_1, x_4}, S_{x_2, x_4})

- m_2 có thể đọc được giá trị của { m_1 }

  $\beta_{rw\_m_2}$:= $(S_{m_1, m_2} \Rightarrow m_2 = m_1) \wedge$ atLeastOne(S_{m_1, m_2}) $\wedge$ atMostOne(S_{m_1, m_2})

- n_2 có thể đọc được giá trị của { n_1 }

  $\beta_{rw\_n_2}$:= $(S_{n_1, n_2} \Rightarrow m_2 = m_1) \wedge$ atLeastOne(S_{n_1, n_2}) $\wedge$ atMostOne(S_{n_1, n_2})

Kết hợp tất cả ràng buộc đọc-ghi của từng sự kiện thu được ràng buộc cho các biến đọc-ghi của toàn bộ chương trình:

β_{rw} := $\beta_{rw\_y_1} \wedge \beta_{rw\_y_2} \wedge \beta_{rw\_x_3} \wedge \beta_{rw\_x_4} \wedge \beta_{rw\_m_2} \wedge \beta_{rw\_n_2}$

3.2.4.3. Xây dựng ràng buộc đảm bảo thứ tự thực thi

Những liên kết đọc-ghi không có ràng buộc đảm bảo thực thi sẽ bị bỏ qua và không được đề cập trong phần này.

- Áp dụng *Algorithm 3* và *Algorithm 4* lên từng liên kết đọc ghi thu được:

  $\beta_{guaranteed}$:= $\beta_{guaranteed\_1} \wedge \beta_{guaranteed\_2}$

  = $(S_{x_1, x_4} \Rightarrow \neg S_{x_2, x_3}$ and $\neg S_{y_4, y_1}) \wedge (S_{x_0, x_4} \Rightarrow \neg S_{x_1, x_3}$ and $\neg S_{x_2, x_3})$

  $\wedge (S_{x_2, x_4} \Rightarrow \neg S_{y_4, y_2}$ and $\neg S_{y_4, y_1}) \wedge (S_{x_1, x_3} \Rightarrow \neg S_{y_3, y_1}$ and $\neg S_{x_0, x_4}$ and $\neg S_{y_4, y_1})$

$\wedge (S_{x_2, x_3} \Rightarrow \neg S_{y_3, y_2}$ and $\neg S_{y_3, y_1}$ and $\neg S_{x_1, x_4}$
and $\neg S_{x_0, x_4}$ and $\neg S_{y_4, y_2}$ and$\neg S_{y_4, y_1})$

$\wedge (S_{y_0, y_2} \Rightarrow \neg S_{y_3, y_1}$ and $\neg S_{y_4, y_1}) \wedge (S_{y_4, y_2} \Rightarrow \neg S_{x_2, x_4}$ and $\neg S_{x_2, x_3})$

$\wedge (S_{y_3, y_2} \Rightarrow \neg S_{y_4, y_1}$ and $\neg S_{x_2, x_3}) \wedge (S_{y_3, y_1} \Rightarrow \neg S_{x_1, x_3}$ and $\neg S_{y_0, y_2}$ and $\neg S_{x_2, x_3})$

$\wedge (S_{y_4, y_1} \Rightarrow \neg S_{x_1, x_4}$and $\neg S_{x_1, x_3}$and $\neg S_{y_3, y_2}$
and $\neg S_{y_0, y_2}$ and $\neg S_{x_2, x_4}$ and$\neg S_{x_2, x_3})$

- Áp dụng *Algorithm 5* vào từng liên kết đọc-ghi thu được:

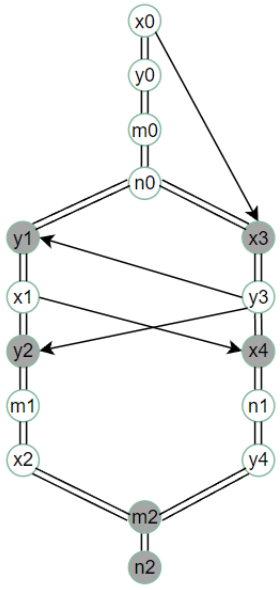
$\beta_{\text{deductive}} :=$ $(S_{x_0, x_4} \Rightarrow \neg S_{y_0, y_2}) \wedge (S_{y_0, y_2} \Rightarrow \neg S_{x_0, x_4})$

3.2.4.4. Kết quả kiểm chứng

Với điều kiện kiểm chứng ϕ_{verify} = !(*m* == 2 && *n* == 3), ϕ_{error} = (*m* == 2 && *n* == 3), bộ giải SMT Z3 Prover đưa ra kết quả SATISFIABLE cho phản ví dụ $\pi := \phi_{init} \wedge \rho \wedge \zeta \wedge \xi \wedge £_\pi \wedge \phi_{err}$ với $\rho := \beta_{\text{independent}}$, $\zeta := \beta_{rw}$ và $\xi := \beta_{guaranteed} \wedge \beta_{deductive}$. Các liên kết đọc ghi được cho là tồn tại trong π là { S_{x_0, x_3}; S_{y_3, y_1} ; S_{x_1, x_4}; S_{y_3, y_2} }

Phản ví dụ π biểu diễn cho trình tự thực thi sau:

```
x = 1;                  // x = 1
y = 1;                  // y = 1
m = 0;                  // m = 0
n = 0;                  // n = 0
y = x +1;               // y = 2
x = y + 1;              // x = 3
m = y || n = x;         // m = 2 || n = 3
x = 0 || y = 0;         // x = 0 || y = 0
assert(!(m == 2 && n == 3));        // false
```



Hình 3.9. EOG của phản ví dụ π

CHƯƠNG 4. THỰC NGHIỆM ĐÁNH GIÁ

Để chứng minh tính khả thi của phương pháp kiểm chứng được trình bày trong khóa luận, công cụ kiểm chứng có tên là ACIO đã được phát triển dựa trên phương pháp kiểm chứng này. Trong phần này, khóa luận sẽ trình bày về cấu trúc của công cụ ACIO và kết quả thực nghiệm của công cụ trên bộ dữ liệu đánh giá.

4.1. Xây dựng công cụ ACIO

4.1.1. Dữ liệu đầu vào

Công cụ này nhận dữ liệu đầu vào gồm hai thành phần: mã nguồn của chương trình và điều kiện kiểm chứng của người dùng. Điều kiện kiểm chứng của người dùng được đi kèm trong mã nguồn như là một câu lệnh kiểm tra *(assert/check)*.

4.1.2. Kết quả đầu ra

Kết quả đầu ra là một bản báo cáo hoạt động của công cụ về kết quả kiểm chứng của chương trình tương ứng với điều kiện kiểm chứng. Bản báo cáo hoạt động sẽ bao gồm các thông tin về số lượng ràng buộc được sinh ra, thời gian sinh ràng buộc, thời gian giải ràng buộc, kết quả của bộ giải SMT với tập ràng buộc đó,... Kết luận cuối cùng về kết quả kiểm chứng của chương trình được định nghĩa như sau:

- Bộ giải SMT trả về kết quả *SATISFIABLE* và thời gian giải ràng buộc nhỏ hơn thời gian giới hạn ⇒ Chương trình vi phạm điều kiện kiểm chứng, *NOT SAFE*
- Bộ giải SMT trả về kết quả *UNSATISFIABLE* và thời gian giải ràng buộc nhỏ hơn thời gian giới hạn ⇒ Chương trình không vi phạm điều kiện kiểm chứng, *SAFE*
- Bộ giải SMT trả về kết quả *UNSATISFIABLE* và thời gian giải ràng buộc vượt quá thời gian giới hạn ($timeout$) ⇒ Công cụ không chắc chắn về kết quả, *NOT SURE*

4.1.3. Các thư viện được sử dụng trong công cụ

4.1.3.1. **Eclipse Core** [9]

- Eclipse Core cung cấp cơ sở hạ tầng nền tảng cơ bản trong việc phân tích và mã hóa chương trình.
- Eclipse Core bao gồm các trình cắm (plug-in) sau:
    - org.eclipse.core.contenttype - Hỗ trợ xác định và quản lý nội dung tệp.
    - org.eclipse.core.expressions - Ngôn ngữ biểu thức dựa trên XML.

- org.eclipse.core.filesystem - API hệ thống tệp chung.
- org.eclipse.core.jobs - Cơ sở hạ tầng cho lập trình đồng thời trong Eclipse.
- org.eclipse.core.resources - Quản lý tài nguyên - dự án, thư mục và tệp.
- org.eclipse.core.runtime - Trước đây là cơ sở của nền tảng, plug-in này phần lớn đã được thay thế bởi Equinox.

- Các trình cắm này đều mang tính tổng quát: mỗi trình cắm cung cấp một dịch vụ cơ bản, API và các điểm mở rộng để quản lý và tương tác với các dịch vụ đó.

4.1.3.2. Eclipse Equinox [10]

- Equinox là một triển khai của đặc tả khung lõi OSGi (Hệ thống mô-đun động cho Java). Nó là một tập hợp các gói triển khai các dịch vụ OSGi và cơ sở hạ tầng để chạy các hệ thống dựa trên OSGi.
- Equinox bao gồm các tính năng:
    - Chịu trách nhiệm phát triển và cung cấp triển khai khung OSGi được sử dụng cho Eclipse.
    - Thực hiện tất cả các khía cạnh của đặc tả OSGi.
    - Điều tra và nghiên cứu các phiên bản tương lai của thông số kỹ thuật OSGi và các vấn đề thời gian chạy liên quan
    - Phát triển cơ sở hạ tầng phi tiêu chuẩn được coi là cần thiết để vận hành và quản lý các hệ thống dựa trên OSGi
    - Triển khai các dịch vụ khung chính và phần mở rộng cần thiết để chạy Eclipse và hữu ích cho những người sử dụng OSGi.

4.1.3.3. Java Tuples [11]

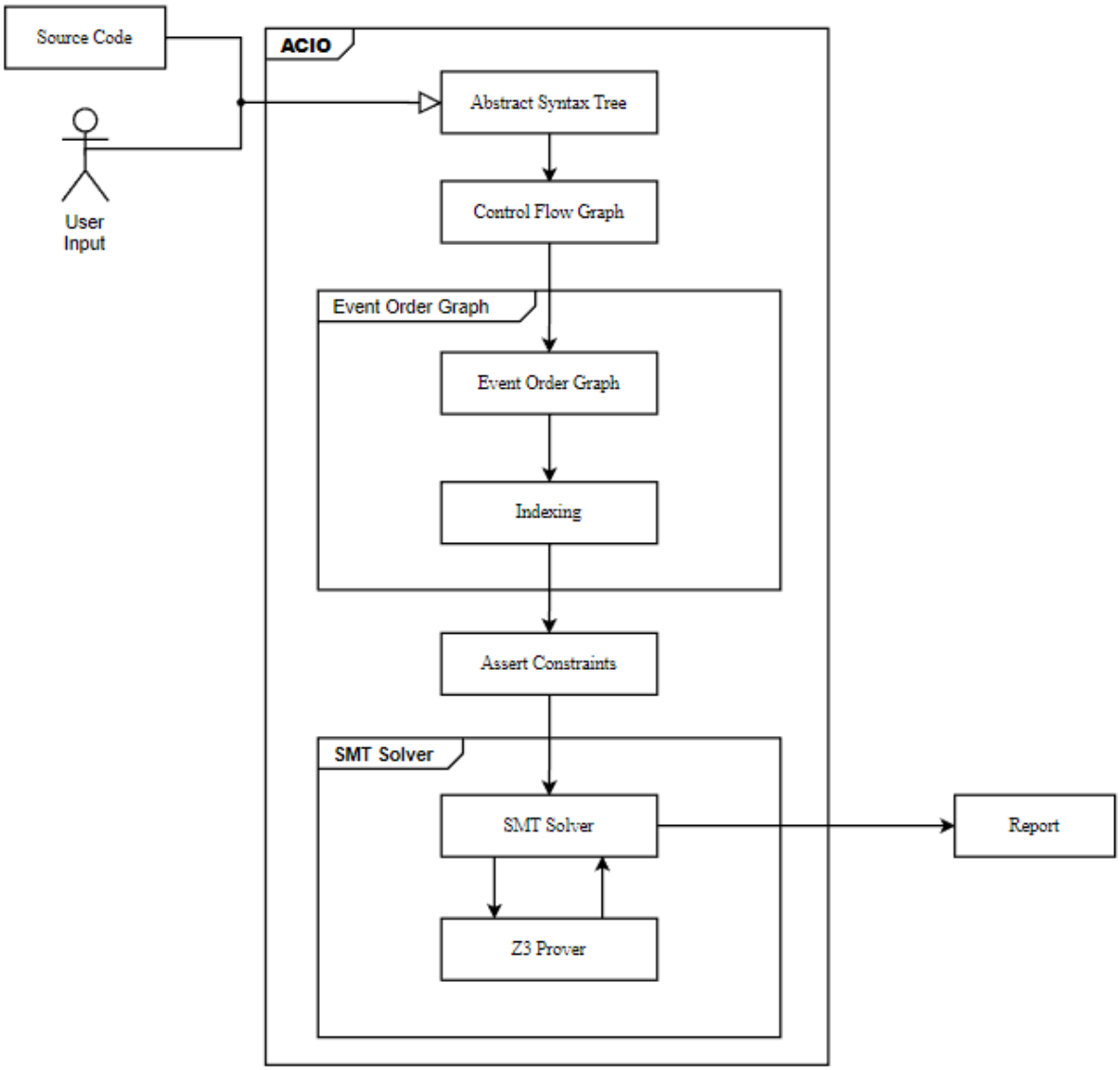
- Javatuples cung cấp một tập các thư viện cho phép người dùng làm việc với tuples.
- Tuples là một chuỗi các đối tượng không nhất thiết phải liên quan đến nhau theo bất kỳ cách nào.
    - Có thể hiểu rằng tuples tương tự với một mảng array nhưng không yêu cầu các phần tử bên trong cùng thuộc một kiểu.
    - Tuy nhiên, cần phải khai báo kiểu đối tượng của từng thành phần trong tuples trước khi sử dụng

4.1.3.4. **Z3 Prover** [3]

- Z3 Prover là một bộ chứng minh định lý được phát triển bởi Microsoft Research.
- Mục đích của Z3 Prover là giải quyết các bài toán có liên quan đến các công thức logic vị từ dựa trên các định lý, lý thuyết, phương pháp suy diễn,... đã được chứng minh tính đúng đắn.

4.1.4. Kiến trúc thực thi tượng trưng của công cụ

Kiến trúc thực thi của công cụ ACIO được biểu diễn trong *Hình 4.1.* dưới đây.



Hình 4.1. Kiến trúc thực thi tượng trưng của công cụ ACIO

Sau khi nhận được mã nguồn của chương trình, công cụ kiểm chứng sẽ tiến hành trừu tượng hóa chương trình dựa trên kĩ thuật thực thi tượng trưng. Mã nguồn sẽ được đưa vào bộ phiên dịch ngôn ngữ của máy tính để tạo nên cây cú pháp trừu tượng AST biểu diễn cấu trúc cú pháp của mã nguồn. Trên cơ sở cấu trúc cú pháp được biểu diễn trong AST, đồ thị luồng điều khiển CFG biểu diễn các luồng điều khiển được sinh ra. Từ đồ thị luồng điều khiển, các sự kiện đọc / ghi sẽ được sàng lọc để tạo ra đồ thị thứ tự sự kiện EOG. Các đỉnh này được đánh chỉ số và trừu tượng hóa thành các logic vị từ cấp một để tạo nên tập các ràng buộc. Tập các ràng buộc được sinh ra sẽ được đưa vào bộ giải SMT để kiểm tra tính khả thi. Báo cáo hoạt động của công cụ về kết quả kiểm chứng của chương trình sẽ được xuất ra sau khi bộ giải SMT trả về kết quả kiểm tra tính khả thi của tập các ràng buộc của chương trình tương ứng.

4.2. Thực nghiệm

4.2.1. Tiêu chuẩn của SV-COMP

Các tiêu chuẩn của Software Verification Competition (SV-COMP) được tạo ra với mục tiêu so sánh các kỹ thuật và công cụ đã được chấp nhận rộng rãi để xác minh chương trình. Một số lượng đáng kể các nghiên cứu về xác minh chương trình đa luồng đã thực hiện các thử nghiệm của họ trên chúng.

Tiêu chuẩn của SV-COMP về kiểm chứng chương trình đa luồng có 1047 ví dụ là các chương trình đa luồng được viết trên ngôn ngữ C [5]. Tuy nhiên, nhiều câu lệnh phức tạp của C chưa được phân tích và xử lý trong công cụ này, khóa luận này sẽ chỉ thực nghiệm 18 ví dụ trong số 1047 (SV – COMP 2017) ví dụ kể trên.

4.2.2. Điều kiện thực nghiệm

Toàn bộ 18 ví dụ sẽ được chạy thực nghiệm trên máy tính sử dụng CPU AMD Ryzen 5 4600H 3GHz 8GB RAM với thời gian giới hạn *(timeout)* là 900000ms (900s). Các chương trình kiểm chứng vượt quá thời gian giới hạn vẫn sẽ được trả về kết quả là UNSAT dựa trên cài đặt gốc của bộ giải SMT Z3Prover được sử dụng.

4.2.3. Kết quả thực nghiệm

Bảng 4.1. Kết quả thực nghiệm của phương pháp ACIO

Chương trình	Số vòng lặp được giới hạn (nếu có)	Kết quả của bộ giải SMT	Số dòng lệnh	Số lượng ràng buộc	Thời gian sinh ràng buộc (ms)	Thời gian giải ràng buộc (ms)	Kết luận
nondet-loop-bound-1.c	20	SAT	38	950	29	3290	NOT SAFE
nondet-loop-bound-2.c	20	UNSAT	38	950	23	Timeout	NOT SURE
read_write_lock-1b.c		UNSAT	41	156	40	20	SAFE
read_write_lock-1.c		UNSAT	43	172	18	17	SAFE
read_write_lock-2.c		UNSAT	41	180	17	14	SAFE
stateful01-1.c		SAT	27	40	9	12	NOT SAFE
stateful01-2.c		SAT	27	40	8	10	NOT SAFE
triangular-1.c	5	UNSAT	25	207	13	86	SAFE
triangular-2.c	5	SAT	25	207	13	27	NOT SAFE
triangular-longer-1.c	10	UNSAT	35	697	80	140248	SAFE
triangular-longer-2.c	10	SAT	35	697	84	9543	NOT SAFE
triangular-longest-1.c	20	UNSAT	55	2577	1313	Timeout	NOT SURE
triangular-longest-2.c	20	UNSAT	55	2577	1332	Timeout	NOT SURE
peterson-b.c		UNSAT	31	52	72	34	SAFE
reorder_c11_bad-10.c	10	SAT	39	98	15	15	NOT SAFE
reorder_c11_bad-30.c	30	SAT	81	258	17	21	NOT SAFE
reorder_c11_good-10.c	10	UNSAT	39	98	19	9	SAFE
reorder_c11_good-30.c	30	UNSAT	81	258	15	9	SAFE

Kết quả thực nghiệm của công cụ ACIO với tập các chương trình thực nghiệm được trình bày trong *Bảng 4.1.* với các tham số báo cáo là số vòng lặp được giới hạn cho các câu lệnh lặp (for, while, do-while,...), số dòng lệnh trong mã nguồn, số lượng ràng buộc được sinh ra, thời gian sinh và giải các ràng buộc kèm theo đó là kết quả của bộ giải SMT Z3Prover và kết luận cuối cùng về kết quả kiểm chứng.

4.3. Đánh giá thực nghiệm

Từ kết quả được trình bày trong *Bảng 4.1.* , có thể thấy rằng phương pháp ACIO đặc biệt hiệu quả với các chương trình nhỏ và vừa với số lượng ràng buộc sinh ra không quá lớn. Bằng phương pháp mã hóa một lần, công cụ ACIO đã giải đúng được bài toán kiểm thử trên 15/18 chương trình với thời gian tiêu hao ít nhất (bao gồm cả sinh và giải ràng buộc) là 18ms trên chương trình *stateful-01-2.c*.

Tuy nhiên, công cụ ACIO cho hiệu quả không cao trên những chương trình phức tạp có số vòng lặp giới hạn kèm theo số lượng ràng buộc lớn, đặc biệt là những chương trình không vi phạm điều kiện kiểm chứng. Việc mã hóa và giải một số lượng lớn các ràng buộc

cùng một lúc trực tiếp đặt gánh nặng lên bộ giải mã SMT, dẫn đến việc bùng nổ thời gian, đặc biệt là với các tập hợp ràng buộc cho kết quả UNSAT. Ví dụ điển hình cho sự kém hiệu quả này là kết quả kiểm chứng của hai chương trình *triangular-longer-1.c* và *triangular-longer-2.c*. *triangular-longer-2.c* với kết quả SAT - NOT SAFE chỉ mất 9534ms (9,5s) để giải các ràng buộc, trong khi *triangular-longer-1.c* với kết quả UNSAT - SAFE cần đến 140248ms (140,2s), dù rằng mã nguồn và thời gian sinh ràng buộc của hai chương trình là tương đương, chỉ khác nhau ở điều kiện kiểm chứng. Ngoài ra có thể kể đến chương trình *triangular-longest-1.c* và *triangular-longest-2.c* bị timeout và cho kết quả NOT SURE với 2577 ràng buộc được sinh ra. Tuy nhiên, trong kiểm chứng phần mềm, việc đưa ra kết quả NOT SURE không gây hậu quả quá nghiêm trọng như việc đưa ra kết quả SAFE khi chương trình có lỗi nên kết quả này có thể chấp nhận được.

KẾT LUẬN

Với sự hỗ trợ của phần cứng, xu hướng áp dụng thực thi đa luồng vào phần mềm càng ngày càng trở nên phổ biến. Điều đó mở ra một triển vọng rất lớn trong việc nâng cao tốc độ thực thi của chương trình. Tuy vậy, triển vọng đó lại làm nảy sinh một thách thức rất lớn đối với việc kiểm thử, kiểm chứng nhằm mục tiêu đảm bảo chất lượng sản phẩm. Các câu lệnh được thực thi song song với nhau tạo ra rất nhiều trình tự thực thi khác nhau không chỉ tạo nên gánh nặng cho kiểm thử, kiểm chứng về số lượng mà còn là về không gian bộ nhớ, chi phí thời gian.

Do sự cấp thiết của việc kiểm thử, kiểm chứng chương trình đa luồng, rất nhiều những đề tài nghiên cứu về việc kiểm chứng tự động đã được thực hiện và công bố công khai. Tuy vậy, số lượng nghiên cứu đạt đến kết quả hoàn hảo trong việc kiểm chứng tự động là rất ít. Đó chính là lí do mà các phương pháp kiểm chứng tự động đã, đang và sẽ tiếp tục là một bài toán thách thức các nhà nghiên cứu trong cả hiện tại và tương lai.

Trong tài liệu này, khóa luận đã tìm hiểu và trình bày về phương pháp kiểm chứng tự động chương trình bằng cách mã hóa chương trình sử dụng Đồ thị thứ tự sự kiện EOG và kết hợp EOG với thực thi tượng trưng để kiểm chứng chương trình đa luồng được viết trên ngôn ngữ C/C++. Đồng thời, công cụ kiểm chứng ACIO cũng được phát triển để chứng minh tính khả thi của phương pháp. Công cụ ACIO bao gồm các thành phần trừu tượng hóa như Cây cú pháp trừu tượng AST, Đồ thị luồng điều khiển CFG tích hợp với EOG để sinh các ràng buộc đọc-ghi. Các ràng buộc đọc ghi này được đưa vào bộ giải mã có tên là Z3 Prover để kiểm tra tính khả thi.

Khác với phương pháp Yogar-CBMC chỉ sinh một số ràng buộc liên quan đến các phép gán giá trị mà không xem xét đến các quan hệ đọc-ghi, phương pháp ACIO sinh ra toàn bộ ràng buộc liên quan đến các phép gán giá trị và các quan hệ đọc-ghi có liên hệ với chúng. Điều này mở ra một triển vọng mới trong việc giảm tải công việc cho bộ giải SMT, giúp cải thiện tốc độ của công đoạn kiểm chứng chương trình.

Với bộ thực nghiệm của SV-COMP, phương pháp ACIO cho kết quả rất tốt trên những chương trình nhỏ và vừa với thời gian tiêu hao chỉ tính bằng millisecond. Nhưng, đối với một số chương trình lớn với số lượng vòng lặp phức tạp, công cụ chưa thể cho ra được kết

quả chắc chắn trong thời gian giới hạn. Tuy vậy, có thể khẳng định rằng kết quả đạt được của công cụ ACIO trong việc kiểm chứng chương trình đa luồng là tương đối khả quan khi đã giải quyết được phần lớn các bài toán đặt ra trong thời gian giới hạn và không đưa ra kết quả sai lệch so với chương trình thực tế.

Trong tương lai, để cải thiện kết quả của ACIO thì bổ sung thêm các điều kiện hoặc chiến thuật duyệt đồ thị EOG để sinh ràng buộc là rất quan trọng. Tăng hiệu quả trong việc duyệt EOG có thể làm giảm thiểu đi các ràng buộc và liên kết đọc-ghi dư thừa, giúp giảm gánh nặng lên bộ giải SMT. Đồng thời, ACIO có thể mở rộng khả năng ứng dụng bằng cách bổ sung thêm các tính chất khác của mã nguồn C/C++ như các kiểu dữ liệu phức tạp (mảng, con trỏ, chuỗi,...) và các cấu trúc phức tạp (struct, các lệnh rẽ nhánh, các vòng lặp,...).

TÀI LIỆU THAM KHẢO

[1] J. Alglave, D. Kroening, and M. Tautschnig, “Partial Orders for Efficient Bounded Model Checking of Concurrent Software,” in *International Conference on Computer Aided Verification,* 2013, pp. 141–157.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, “Symbolic Model Checking without BDDs,” in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999, pp. 193–207.

[3] N. Bjorner, “Z3.” Z3 Theorem Prover, Apr. 26, 2023. Accessed: Apr. 27, 2023. [Online]. Available: https://github.com/Z3Prover/z3

[4] J. C. King, “Symbolic execution and Program testing,” 1973.

[5] LMU Munich, “Files · main · SoSy-Lab / Benchmarking / SV-Benchmarks · GitLab,” *GitLab*, Dec. 23, 2022. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main (accessed Apr. 27, 2023).

[6] LMU Munich, “SV-COMP - International Competition on Software Verification.” https://sv-comp.sosy-lab.org/ (accessed Apr. 27, 2023).

[7] LMU Munich, “SV-COMP 2017 - 6th International Competition on Software Verification.” https://sv-comp.sosy-lab.org/2017/results/results-verified/ (accessed Apr. 27, 2023).

[8] S. Qadeer and J. Rehof, “Context-Bounded Model Checking of Concurrent Software,” in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2005, pp. 93–107.

[9] The Eclipse Foundation, “Core | The Eclipse Foundation.” https://www.eclipse.org/eclipse/platform-core/ (accessed Apr. 27, 2023).

[10] The Eclipse Foundation, “Equinox | The Eclipse Foundation.” https://www.eclipse.org/equinox/ (accessed Apr. 27, 2023).

[11] The JAVATUPLES team, “javatuples - Main.” https://www.javatuples.org/index.html (accessed Apr. 27, 2023).

[12] L. Yin, W. Dong, W. Liu, and J. Wang, “Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification.” Wanwei Liu, Oct. 23, 2017. [Online]. Available: https://www.researchgate.net/publication/319326871_Scheduling_Constraint_Based_Abstraction_Refinement_for_Multi-Threaded_Program_Verification