ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Bùi Hoàng Khánh

XÂY DỰNG PHẦN MỞ RỘNG KIỂM CHỨNG THUỘC TÍNH LOGIC THỜI GIAN CHO JAVA PATHFINDER

LUẬN VĂN THẠC SĨ

HÀ NỘI - 2014

ĐẠI HỌC QUỐC GIA HÀ NỘI

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Bùi Hoàng Khánh

XÂY DỰNG PHẦN MỞ RỘNG KIỂM CHỨNG THUỘC TÍNH LOGIC THỜI GIAN CHO JAVA PATHFINDER

Ngành: Công nghệ thông tin

Chuyên ngành: Kỹ nghệ phần mềm

Mã số: 60480103

LUẬN VĂN THẠC SĨ

NGƯỜI HƯỚNG DẪN KHOA HỌC: TS. Trương Anh Hoàng

HÀ NỘI - 2014

Lời cảm ơn

Lời đầu tiên, tôi xin gửi lời cảm ơn sâu sắc nhất tới TS. Trương Anh Hoàng đã tận tình hướng dẫn tôi trong suốt quá trình thực hiện Luận văn.

Tôi chân thành cảm ơn các thầy, cô đã tạo cho tôi những điều kiện thuận lợi để tôi học tập và nghiên cứu tại trường Đại học Công Nghệ.

Cuối cùng, tôi muốn được gửi lời cảm ơn tới gia đình và bạn bè, những người thân yêu luôn bên cạnh và động viên tôi trong suốt quá trình thực hiện Luận văn.

Tôi xin chân thành cảm ơn!

Hà Nội, ngày 30 tháng 10 năm 2014

Học viên

Bùi Hoàng Khánh

Tóm tắt nội dung

Trong những năm gần đây, thực thi tượng trưng được xem là phương pháp hiệu quả trong kiểm thử phần mềm. Dựa trên thực thi tượng trưng, chúng ta có thể duyệt qua hết các dãy thực thi có thể có của một chương trình. Hơn thế nữa, thực thi tượng trưng có thể phát hiện ra các lỗi khó mà các phương pháp kiểm thử thông thường tốn nhiều tài nguyên và công sức để phát hiện. Luận văn tập trung vào việc áp dụng thực thi tượng trưng vào giải quyết các bài toán cụ thể là kiểm tra một chương trình Java có cài đặt theo đúng đặc tả công thức logic thời gian tuyến tính (LTL) hay không. Kết quả của luận văn là đã xây dựng được một công cụ kiểm chứng công thức LTL với các dãy thực thi tượng trưng vô hạn dựa trên Java PathFinder – một nền tảng phổ biến cho việc kiểm chứng mô hình các chương trình Java.

Lời cam đoan

Tôi xin cam đoan luận văn "Xây dựng phần mở rộng kiểm chứng thuộc tính logic thời gian tuyến tính cho Java Pathfinder" là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả được trình bày trong luận văn là hoàn toàn trung thực. Tôi đã trích dẫn đầy đủ các tài liệu tham khảo, công trình nghiên cứu liên quan ở trong nước và quốc tế. Ngoại trừ các tài liệu tham khảo này, luận văn hoàn toàn là công việc của riêng tôi.

Trong các công trình khoa học được công bố trong luận văn, tôi đã thể hiện rõ ràng và chính xác đóng góp của các đồng tác giả và những gì do tôi đã đóng góp.

Luận văn được hoàn thành trong thời gian tôi làm học viên tại Bộ môn Công nghệ phần mềm, Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội.

Hà Nội, ngày 30 tháng 10 năm 2014

Học viên

Bùi Hoàng Khánh

Mục lục

Bảng ký hiệu và chữ viết tắt

Chữ viết tắt	Cụm từ đầy đủ	Ý nghĩa
FSM	Finite state machine	Máy hữu hạn trạng thái
JPF	Java PathFinder	Java PathFinder
JVM	Java virtual machine	Máy ảo Java
LTL	Linear temporal logic	Logic thời gian tuyến tính
PC	Path condition	Điều kiện đường đi
SPF	Symbolic PathFinder	Symbolic PathFinder

Danh mục hình vẽ

Chương 1. Mở đầu

1.1. Kiểm thử

Sự đúng đắn của phần mềm là một trong những yếu tố quan trọng hàng đầu trong quá trình phát triển phần mềm. Kiểm thử là một trong những phương pháp đầu tiên được sử dụng để kiểm tra tính đúng đắn của phần mềm, nó có thể chiếm tới 50% chi phí phát triển phần mềm.

Hai nguyên nhân chính dẫn tới tình trạng chi phí cao của kiểm thử đó là: thiếu sự tự động hoá và thiếu độ đo tốt cho việc kiểm thử thành công. Kiểm thử tiêu tốn khá nhiều tài nguyên về bộ nhớ và CPU của hệ thống và thường ít khi có thể kiểm tra hết tất cả những kịch bản thực thi của chương trình. Trước hết, các ca kiểm thử cần được xác định một cách thủ công, tức là cần xác định các bộ dữ liệu đầu vào và dữ liệu đầu ra mong muốn tương ứng. Những bộ kiểm thử này sẽ cần được lặp đi lặp lại trong quá trình tiến hoá phần mềm. Mặc dù vậy, thậm chí khi một đội ngũ kiểm thử chuyên nghiệp thực hiện hàng triệu ca kiểm thử, lỗi vẫn xuất hiện trong sản phẩm phần mềm. Một thực tế là người kiểm thử rất khó có thể biết được họ gần hoàn thành hay đã hoàn thành việc kiểm thử hay chưa bởi vì kiểm thử khó có thể phát hiện hết các kịch bản thực thi của chương trình có thể gặp phải. Thực tế, quá trình kiểm thử thường được cho là kết thúc khi đã dùng hết tài nguyên về bộ nhớ hoặc CPU của hệ thống mà thiếu sự đảm bảo về tính đúng đắn của chương trình. Phương pháp này có thể giúp chúng ta phát hiện được những lỗi liên quan đến phần cứng, bộ biên dịch hay máy ảo nhưng phương pháp này rất khó kiểm tra tính đúng đắn khi bộ dữ liệu đầu vào không có trong bộ mẫu kiểm thử đã được chuẩn bị sẵn.

1.2. Kiểm chứng hình thức

Kiểm chứng hình thức (formal verification) [4] được dùng để chứng minh hoặc phản chứng sự đúng đắn của một hệ thống nào đó dựa trên một đặc tả hình thức, sử dụng phương pháp hình thức có trong toán học.

Quá trình kiểm tra dựa trên việc chứng minh hình thức trên một mô hình toán học của hệ thống. Các mô hình toán học thường được sử dụng như là máy hữu hạn trạng thái, các hệ thống chuyển được gán nhãn, mạng Petri, toán học đại số...

Hai phương pháp kiểm tra hình thức thường được sử dụng đó là: Kiểm chứng mô hình và suy luận lôgic.

- Kiểm chứng mô hình: là phương pháp thăm dò toàn bộ các khả năng của mô hình toán học của chương trình (chỉ áp dụng được cho các mô hình hữu hạn trạng thái).

- Suy luận lôgic: là phương pháp sử dụng các hệ thống suy luận toán học, thường là những công cụ chứng minh như HOL, ACL2, Isabelle…

Một yêu cầu cho phương pháp kiểm chứng hình thức là phải đảm bảo tính chính xác của các yếu tố khác như phần cứng, bộ biên dịch, máy ảo... Kiểm chứng hình thức sẽ không thể phát hiện được lỗi gây ra do các thành phần trên.

Thực thi tượng trưng ra đời dựa trên việc kết hợp thế mạnh của phương pháp kiểm tra hình thức và kiểm thử, đồng thời giúp người lập trình tạo được các bộ kiểm thử đạt được độ bao phủ cao và ít tốn công sức nhất. Đặc biệt với các chương trình có không gian trạng thái lớn, thực thi tượng trưng có ý nghĩa quan trọng đó là giúp giảm được không gian trạng thái, qua đó giảm các phụ thuộc vào các yếu tố tài nguyên CPU, bộ nhớ cũng như bộ dữ liệu đầu vào.

1.3. Nội dung nguyên cứu và đóng góp của luận văn

Với mục đích tạo ra một công cụ hỗ trợ kiểm chứng các chương trình có không gian trạng thái lớn, mà cụ thể là các tính chất thời gian tuyến tính của một chương trình Java, luận văn tập trung nghiên cứu các nội dung sau:

- Các kiến thức tổng quát về hệ thống chuyển trạng thái, logic thời gian tuyến tính, phương pháp mô hình hóa hệ thống, automat buchi.

- Kiến trúc, kỹ thuật mở rộng Java PathFinder và Symbolic PathFinder.

- Cài đặt và tích hợp thực thi tượng trưng vào thuật toán DDFS.

Đóng góp chính của luận văn là việc tích hợp được thực thi tượng trưng vào thuật toán DDFS đã cài đặt trước đó trong công cụ *jpf-ltl*. Công cụ *jpf-ltl* là một mở rộng của Java PathFinder cho phép kiểm chứng các chương trình Java có thỏa mãn một tính chất logic thời gian tuyến tính nào đó hay không. Nhưng với cài đặt hiện có, jpf-ltl, chưa hỗ trợ kiểm chứng trong không gian trạng thái lớn với thực thi tượng trưng, được tập trung vào giải quyết trong luận văn này. Bên cạnh đó, luận văn còn thực hiện tổ chức lại mã nguồn của *jpf-ltl*, đồng thời thêm một thành phần cần thiết còn thiếu như các biểu thức nguyên tử.

1.4. Cấu trúc luận văn

Phần còn lại của luận văn được trình bày thành các phần như sau: Chương 2 sẽ trình bày các kiến thức nền về logic thời gian tuyến tính, hệ thống chuyển trạng thái, Buchi automat và khái niệm thực thi tượng trưng. Chương 3 giới thiệu về Java Pathfinder và công cụ hỗ trợ thực thi tượng trưng Symbolic Pathfinder. Chương 4 sẽ áp dụng thực thi tượng trưng vào bài toán kiểm chứng các tính chất logic thời gian tuyến tính của chương trình Java có không gian trạng thái lớn. Cuối cùng là phần đánh giá và các hướng phát triển tiếp theo.

Chương 2. Logic thời gian tuyến tính và thực thi tượng trưng

Chương này sẽ trình bày các kiến thức nền về hệ thống chuyển trạng thái, phương pháp mô tả hệ thống chuyển trạng thái, logic thời gian tuyến tính và khái niệm thực thi tượng trưng trong kiểm chứng mô hình.

2.1. Hệ thống chuyển trạng thái (Transition system)

Các mệnh đề logic được sử dụng để mô tả một hệ thống tĩnh (static system). Hệ thống tĩnh được hiểu là không thay đổi trạng thái theo thời gian, hoặc chỉ quan tâm đến một trạng thái xác định của hệ thống. Nhưng trong thực tế có rất nhiều ứng dụng chúng ta phải quan tâm đến trạng thái của hệ thống theo thời gian, như hệ điều hành, các ứng dụng mạng, bộ lập lịch, hay các thiết bị tự động. Ví dụ, khi xem xét tính an toàn của các giao thức mã hóa, chúng ta phải quan tâm đến tất cả chuỗi hành động làm thay đổi trạng thái của hệ thống sử dụng giao thức đó.

Hệ thống chuyển trạng thái (transition system) là hệ thống mà trạng thái của nó chuyển từ trạng này sang trạng thái khác theo thời gian dưới tác dụng của các hành động khác nhau.

Một hệ thống chuyển trạng thái thường có các tính chất đặc trưng sau:

- Tại một thời điểm cụ thể, hệ thống có một trạng thái xác định.
- Trạng thái của hệ thống có thể thay đổi, thường là dưới tác động của một số loại hành động (actions). Các loại hành động này có thể xuất hiện từ bên trong hoặc bên ngoài hệ thống.

Với hai tính chất này, chúng ta có thể xây dựng mô hình toán học của hệ thống dựa trên các khái niệm trừu tượng sau:

- Biến (variable) để mô tả các thuộc tính của hệ thống và giả sử rằng trạng thái được xác định bằng cách gán các giá trị vào các biến. Các biến này cũng được xem là biến trạng thái của hệ thống.
- Hình thức hóa các hành động (action) bằng cách xác định xem trạng thái hiện tại của hệ thống có thay đổi hay không và các biến trạng thái thay đổi như thế nào sau hành động đó.

Từ những tính chất và khái niệm ở trên, chúng ta có định nghĩa hệ thống chuyển trạng thái như sau:

Hệ thống chuyển trạng thái là một tập $S = (X, D, Dom, In, T)$, trong đó

- X là tập hữu hạn các biến trạng thái.

- *D* là tập không rỗng, còn gọi là miền (*domain*). Các thành phần của *D* gọi là các giá trị.
- *Dom* là một ánh xạ từ *X* đến các tập con không rỗng của *D*. Với mỗi biến trạng thái $x \epsilon X$, tập *Dom(x)* được gọi là miền của *x*.
- *In* là tập trạng thái bắt đầu.
- *T* là tập trạng thái kết thúc.

Một biến đổi *t* được áp dụng cho trạng thái *s* nếu tồn tại một trạng thái *s'* mà (*s*; *s'*) ∈ *t*. Biến đổi *t* được gọi là đơn định nếu ứng với mỗi trạng thái *s* thì tồn tại nhiều nhất một trạng thái *s'* mà (*s*; *s'*) ∈ *t*. Ngược lại, *t* được gọi là không đơn định.

Một hệ thống chuyển trạng thái *S* là hữu hạn trạng thái nếu *X* là hữu hạn và vô hạn trạng thái nếu *X* vô hạn.

2.2. Logic thời gian tuyến tính (LTL)

Logic thời gian tuyến tính được sử dụng để biểu diễn các tính chất của một hệ thống cho kiểm tra mô hình [1]. Cho trước một tập các mệnh đề nguyên tử (atomic proposition) *P*, một công thức LTL được định nghĩa bằng cách sử dụng các toán tử logic chuẩn và các toán tử thời gian X (next) và U (strong until) như sau:

- Mỗi thành phần $p \in P$ là một công thức
- Nếu ϕ và ψ là các công thức thì $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $X\phi$, ϕ U ψ cũng là các công thức

Một thể hiện cho một công thức LTL là một từ vô hạn $w = x_0x_1x_2\ldots$ trên 2^P. Nói cách khác, một thể hiện tương ứng với một thời điểm thời gian, mà tại đó có một tập các mệnh đề nguyên tử đúng. Chúng ta viết w_1 là phần tử đầu tiên của từ *w* bắt đầu tại x_1. Như vậy, ngữ nghĩa của LTL được định nghĩa như sau [9]:

- $w \models p$ khi và chỉ khi $p \in x_0$, với $p \in P$
- $w \models \neg\phi$ khi và chỉ khi $w \models \phi$ không đúng
- $w \models \phi \vee \psi$ khi và chỉ khi $w \models \phi$ hoặc $w \models \psi$
- $w \models \phi \wedge \psi$ khi và chỉ khi $w \models \phi$ và $w \models \psi$
- $w \models \phi$ U ψ khi và chỉ khi $\exists i \geq 0\ w_i \models \psi$ và $\forall 0<j<i\ w_j \models \phi$
- $w \models X\phi$ khi và chỉ khi $w_1 \models \phi$

Ngoài ra còn có hai công thức rút gọn sau "true ≡ $\phi \vee \neg\phi$" và "*false* ≡ ¬*true*".

2.2.1. Các toán tử

- Toán tử *global* (toàn thể)

Toán tử *global* kí hiệu là □. Giả sử ϕ là một biểu thức logic vị từ, khi đó biểu thức □ϕ có giá trị đúng nếu ϕ đúng trong mọi thời điểm.

Toán tử *global* thường được kí hiệu bằng chữ cái G.

- Toán tử *next* (tiếp theo)

Toán tử *next* kí hiệu là ○. Giả sử ϕ là một biểu thức logic. Có thể coi ϕ như một dãy trạng thái và trạng thái hiện tại đang xét đến là trạng thái thứ n. Khi đó biểu thức ○ϕ có giá trị đúng khi và chỉ khi phần tử ngay sau phần tử hiện tại trong dãy trạng thái ϕ (phần tử thứ n+1) có giá trị đúng.

Toán tử *next* thường được kí hiệu bằng chữ cái X.

- Toán tử *eventually* (cuối cùng cũng xảy ra)

Toán tử *eventually* kí hiệu là ◊. Giả sử ϕ là một biểu thức logic vàϕ được coi như một dãy trạng thái mà mỗi phần tử chỉ có giá trị bằng 0 hoặc 1. Khi đó giá trị biểu thức ◊ϕ bằng 1 khi và chỉ khi ϕ có ít nhất một phần tử có giá trị bằng 1. Toán tử ◊ được định nghĩa thông qua toán tử □ như sau:

$$\Diamond\phi \equiv \neg\Box\neg\varphi$$

Toán tử *eventually* thường được kí hiệu bằng chữ cái F.

2.2.2. Các tính chất

- Safety (tính an toàn)

Tính an toàn của một chương trình đảm bảo rằng sẽ không bao giờ xảy ra tình huống xấu trong chương trình (something bad never happen).

Tính an toàn có thể được biểu diễn bằng logic thời gian như sau:

$$G\phi$$

Trong đó ϕ là một biểu thức logic.

Ví dụ của tính an toàn:

- Nhiệt độ của phản ứng không bao giờ quá 100 độ C.
- Bất kì lúc nào chìa khóa xe chưa vặn tới vị trí khởi động, xe sẽ không nổ máy.

- Liveness (tính sống)

Tính sống của một chương trình đảm bảo rằng nó có thể thực thi được một chức năng "tốt" nào đó đã đặt ra (something good will happen eventually).

Tính sống có thể được biểu diễn bằng các phép kết hợp AF hoặc F trong logic thời gian như sau:

$$F\phi$$

$$G(\phi \rightarrow F\varphi)$$

$$GF\phi$$

Ví dụ:

- Khi chìa khóa xe vặn tới vị trí khởi động, xe sẽ nổ máy.
- Bóng đèn sẽ chuyển sang màu xanh.

- Fairness (tính công bằng)

Tính công bằng đảm bảo rằng nếu một sự kiện nào đó ở trạng thái sẵn sàng được thực thi thì đến một lúc nào đó nó sẽ được thực thi.

Thuộc tính công bằng có thể được biểu diễn bằng các toán tử AF và phép suy ra:

$$AG(\phi \rightarrow AF\varphi)$$

Ví dụ:

- Trong một hệ thống truyền-nhận tin là khi một gói tin được gửi đi thì đến một lúc nào đó nó sẽ đến được đích.

2.3. Buchi automat

Buchi giới thiệu automat trên các đầu vào vô hạn [2]. Một automat Buchi là một automat hữu hạn trạng thái không đơn định và đầu vào các từ vô hạn. Một từ được đoán nhận bởi Buchi nếu như trong khi đọc từ đó Buchi đi qua một vài trạng thái đặc biệt nào đó thường xuyên và vô hạn.

Một cách hình thức Buchi được định nghĩa như một tập $A = (\Sigma, S, \Delta, s_0, F)$, trong đó

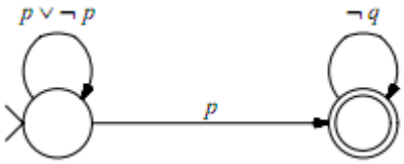
- Σ là một bộ chữ cái
- S là một tập các trạng thái
- $\Delta \subseteq S \times \Sigma \times S$ là một hàm chuyển
- $s_0 \in S$ là trạng thái ban đầu
- $F \subseteq S$ tập hợp các trạng thái chấp nhận

Như vậy, một automat Buchi là một automat được thêm vào một tập F bao gồm các trạng thái chấp nhận. Buchi được sử dụng để định nghĩa ngôn ngữ trên các từ w hay tức là các hàm từ w sang bộ chữ cái Σ.

Chúng ta định nghĩa một đường chạy σ của A trên một từ $w = a_1a_2\ldots$ như là một dãy $\sigma = s_0, s_1, \ldots$, đó là một hàm từ w sang S, ở đây $(s_{i-1}, a_i, s_i) \in \Delta$, với mọi $i \geq 1$. Một đường chạy $\sigma = s_0,s_1,\ldots$ được gọi là được chấp nhận nếu như có một vài trạng thái trong F được lặp lại thường xuyên và vô hạn, tức là có một vài trạng thái $x \in F$ mà ở

đó có nhiều vô hạn $i \in \omega$ để cho $s_i = x$. Từ w được đoán nhận bởi A nếu như có một đường chạy được chấp nhận của A trên từ w đó.

Việc xây dựng Buchi A_f từ công thức f trong trường hợp xấu nhất thì độ phức tạp tính toán là hàm mũ của độ dài của công thức. Tuy nhiên trên thực tế, hầu hết các công thức thường rất ngắn và trường hợp xấu nhất rất hiếm khi xảy ra.



Hình 2.1 Buchi tương đương với công thức ¬((p∨◊q)

Hình 2.1 chỉ ra một automat Buchi đoán nhận tất cả các từ vô hạn thỏa mãn công thức $\neg f$ với $f \equiv (p \vee \Diamond q)$, nghĩa là tất cả các dãy trạng thái trong đó bao gồm một trạng thái p đúng và từ đó q không bao giờ đúng trong suốt phần còn lại của dãy.

2.4. Thực thi tượng trưng

2.4.1. Thực thi tượng trưng

Thực thi tượng trưng [5] được James C. King đề xuất năm 1976, với tư tưởng chủ đạo là thực thi chương trình sử dụng các dữ liệu đầu vào là các giá trị tượng trưng (symbolic values) thay vì các giá trị cụ thể (concrete values).

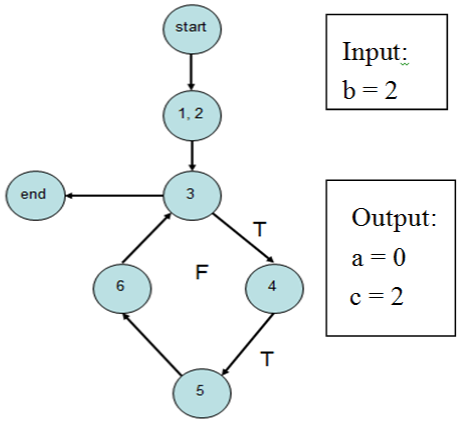
Ta xem một chương trình P như là một hàm P: $I \rightarrow O$, trong đó:

- I là tập hợp các đầu vào (input).
- O là tập hợp các đầu ra (output) có thể có.

Giả sử đầu vào của chương trình có n tham số $p_1, p_2, \ldots, p_n$ khi đó ta có thể biểu diễn I dưới dạng như sau: $I = (p_1, p_2, \ldots, p_n)$.

Một bộ tham số cụ thể $i = (x_1, x_2, \ldots, x_n)$ biểu thị một đầu vào cụ thể cho chương trình P, với x_k $(1 \leq k \leq n)$ là một giá trị cụ thể của biến p_k. Giả sử chương trình với đầu vào i có kết quả là 0_i, ta biểu diễn $0_i = P(i)$.

```
void doSomething(int b){
1. a = b
2. c = 0;
3. while (a > 1) {
4.       If (a*a > c)
5.          c = c + a
6.  a = a - 2
    }
}
```



Hình 2.2 Minh hoạ biểu đồ luồng điều khiển

Với mỗi chương trình *P* ta có thể xây dựng được một đồ thị có hướng *G*, với *G* = (*V*, *E*) trong đó:

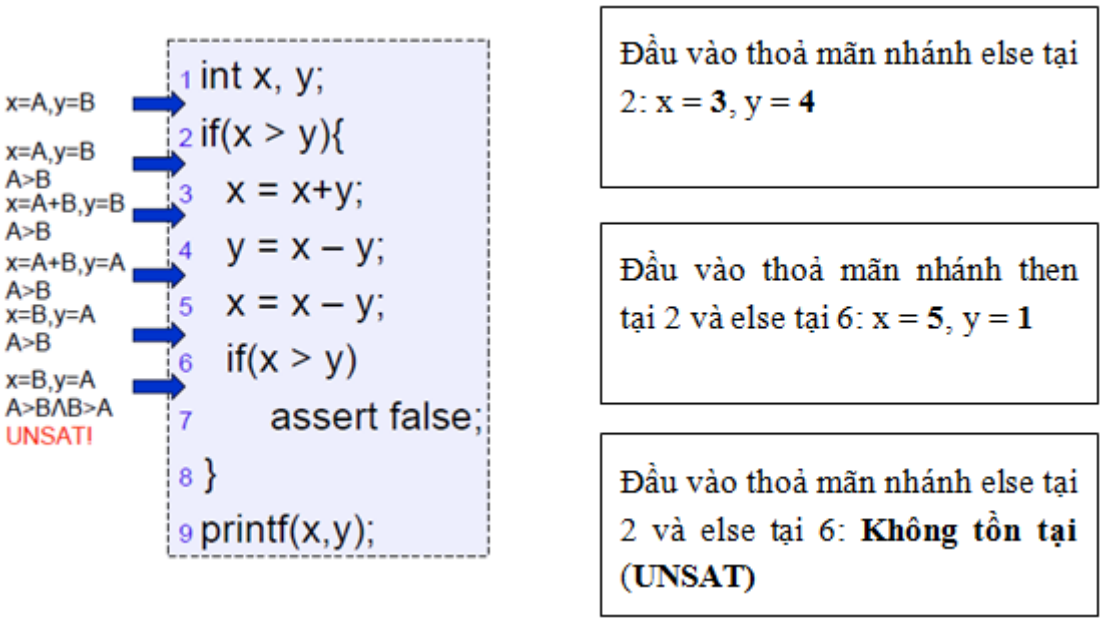
- *V* là tập hợp các nút (các khối cơ bản).
- *E* là tập của các cạnh.

Mỗi nút (khối cơ bản) là một dãy liên tục các chỉ thị sao cho luồng điều khiển không có sự rẽ nhánh hoặc ngừng lại từ khi đi vào nút đó cho tớ khi đi ra khỏi nút đó. Ví dụ luồng điều khiển ở *Hình 2.2*

Một đường đi (path) cụ thể là dãy các nút: $p = (p_1, p_2, \ldots, p_n)$ với p_n là nút cuối của đường đi *p* và $(p_i, p_{i+1}) \in E$ $(1 \le i \le n\text{-}1)$. Nếu tồn tại $i \in S$ sao cho sự thực thi $P(i)$ đi theo đường đi *p* thì *p* gọi là đường đi khả thi, ngược lại *p* là đường đi không khả thi. Một đường đi bắt đầu tại nút vào và kết thúc tại nút ra gọi là đường đi đầy đủ, ngược lại nếu kết thúc tại nút không phải là nút ra thì gọi là đường đi không đầy đủ (path segment).

Điều kiện đường đi (PC) là một biểu thức kết hợp của các ràng buộc mà các giá trị đầu thoả mãn để chương trình thực thi theo đường đi tương ứng với PC đó. Mỗi ràng buộc là một biểu thức tượng trưng dạng $x \circ y$ trong đó:

- Ít nhất một trong hai giá trị *x* và *y* là giá trị tượng trưng (hoặc cả hai).
- ○ là một toán tử thuộc tập hợp {≤, ≠, =, <, >, ≥}.



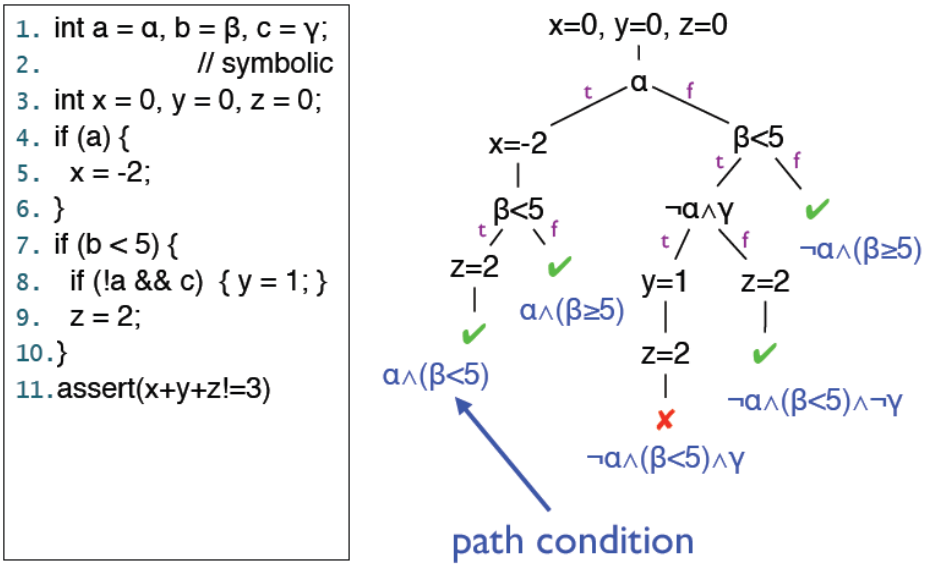
Hình 2.3 Ví dụ về đường đi không khả thi

Các ràng buộc đó chính là biểu thức của điều kiện rẽ nhánh và biểu thức phủ định của điều kiện rẽ nhánh tương ứng với nhánh *true* và *false*. Tại mỗi câu lệnh rẽ nhánh, các ràng buộc được tạo ra. Các ràng buộc này được biểu thị bởi biểu thức của các giá trị tượng trưng hay biểu thức của giá trị tượng trưng và giá trị cụ thể phụ thuộc

vào biến xuất hiện trong biểu thức điều kiện của câu lệnh rẽ nhánh có giá trị tượng trưng được tính toán để kết hợp với nó hay không.

2.4.2. Thực thi tượng trưng tĩnh

Ý tưởng chính của thực thi tượng trưng là thực thi chương trình với các giá trị tượng trưng thay vì các giá trị cụ thể của các tham số đầu vào. Với mỗi tham số đầu vào, một giá trị tượng trưng được đưa ra để kết hợp với nó. Mỗi biến trong chương trình *P* mà giá trị của nó phụ thuộc vào giá trị của các tham số đầu vào thì trong quá trình thực thi tượng trưng một giá trị tượng trưng sẽ được tính toán để kết hợp cùng với nó. Mỗi giá trị tượng trưng biểu thị cho một tập hợp các giá trị cụ thể mà một biến hoặc một tham số đầu vào có thể nhận. Kết quả trả về của một chương trình được thực thi tương trưng nếu có cũng được biểu thị bởi biểu thức của các giá trị tượng trưng.



Hình 2.4 Biểu thức đường đi (PC)

Giá trị tượng trưng của biến *x* có thể được biểu thị bởi một trong các biểu thức sau:

(a) Một ký hiệu đầu vào (input symbol).

(b) Một biểu thức kết hợp giữa các giá trị tượng trưng bởi các toán tử.

(c) Một biểu thức kết hợp giữa giá trị tượng trưng và giá trị cụ thể bởi toán tử.

Một ký hiệu đầu vào biểu thị cho giá trị tượng trưng của một tham số đầu vào lúc bắt đầu thực thi chương trình. Các tham số đầu vào khác nhau của *P* được biểu thị bởi các ký hiệu đầu vào khác nhau. Các toán tử (operator) là các phép toán như cộng (+), trừ (–), nhân (*), chia (/).

Nếu giá trị của một biến *x* không phụ thuộc vào các giá trị đầu vào thì không có giá trị tượng trưng nào được tính toán để kết hợp với nó. Giá trị tượng trưng của các

biến và các tham số đầu vào được cập nhật như các giá trị cụ thể của nó trong quá trình thực thi. Gán một giá trị cụ thể từ một biến tới biến khác dẫn đến giá trị tượng trưng cũng được sao chép nếu biến được gán tới một biến khác có một giá trị tượng trưng. Giả sử với một câu lệnh gán $x=e$, nếu e là một tham số đầu vào, thì giá trị tượng trưng được gán cho x sẽ có dạng (a). Nếu e là một biểu thức tính toán gồm các toán hạng. Các toán hạng đó có thể là biến, tham số đầu vào hoặc hằng thì giá trị tượng trưng của biến x sẽ là một biểu thức tượng trưng dạng (b) nếu mỗi toán hạng trong biểu thức có một giá trị tượng trưng kết hợp với nó, hoặc là một biểu thức tượng trưng dạng (c) nếu có toán hạng là hằng số hoặc không có giá trị tượng trưng kết hợp với nó. Giá trị cụ thể của một hằng hoặc một biến cũng được sử dụng trong biểu thức tượng trưng nếu như hằng hoặc biến đó không có giá trị tượng trưng kết hợp với nó.

Trạng thái của một chương trình được thực thi tương trưng bao gồm các giá trị của các biến trong chương trình, điều kiện đường đi (PC) và biến đếm chương trình (program counter). Biến đếm chương trình xác định chỉ thị (câu lệnh) tiếp theo sẽ được thực thi. Mỗi PC là một biểu thức kết hợp bởi các ràng buộc mà các giá trị đầu vào chương trình cần thỏa mãn để chương trình được thực thi theo đường đi tương ứng với PC đó. Mỗi ràng buộc là một biểu thức tượng trưng dạng $x \circ y$ trong đó x là giá trị tượng trưng, y là giá trị tượng trưng hoặc giá trị cụ thể và $\circ \in \{\leq, \neq, =, <, >, \geq\}$. Các ràng buộc đó chính là biểu thức của điều kiện rẽ nhánh và biểu thức phủ định của điều kiện rẽ nhánh tương ứng với nhánh *true* và *false*. Tại mỗi câu lệnh rẽ nhánh, các ràng buộc được tạo ra. Các ràng buộc này được biểu thị bởi biểu thức của các giá trị tượng trưng hay biểu thức của giá trị tượng trưng và giá trị cụ thể phụ thuộc vào biến xuất hiện trong biểu thức điều kiện của câu lệnh rẽ nhánh có giá trị tượng trưng được tính toán để kết hợp với nó hay không.

Trong quá trình thực thi tượng trưng, việc chương trình được thực thi theo một đường đi cụ thể nào đó không phụ thuộc vào các giá trị cụ thể của các biến và các tham số đầu vào. Tại các điểm rẽ nhánh, cả hai nhánh ra sẽ được xem xét để điều hướng sự thực thi hiện thời đi theo. Thực thi tượng trưng chủ yếu liên quan tới việc thực thi hai loại câu lệnh đó là câu lệnh gán (assignment statments) và câu lệnh rẽ nhánh. Tại các câu lệnh gán thì giá trị tượng trưng của các biến chương trình cũng như các tham số đầu vào có liên quan tới câu lệnh gán đó được cập nhật. Tại các câu lệnh rẽ nhánh, chương trình sẽ được điều hướng để thực thi theo cả hai nhánh. Hai ràng buộc đường đi tương ứng với hai nhánh sẽ được tạo ra. Một ràng buộc là biểu thức điều kiện tại câu lệnh rẽ nhánh. Còn ràng buộc kia là phủ định của biểu thức điều kiện rẽ nhánh. Các ràng buộc này sẽ được cập nhật vào điều kiện đường đi tương ứng với các nhánh đó. Đồng thời các PC này sẽ được đánh giá để xác định đường đi tương ứng với PC đó có khả thi. Nếu PC được đánh giá trở thành *false* thì thực thi tượng trưng sẽ quay lui và chỉ thực thi theo nhánh khả thi. Các PC được tạo ra bằng cách thu gom các

ràng buộc trên các đường đi tương ứng và giải quyết các ràng buộc này sẽ sinh ra các giá trị cụ thể cho các tham số đầu vào.

Ví dụ:

```
public void doSomething(int a, int b)
```

Trong quá trình thực thi, thay vì cung cấp các giá trị cụ thể *S* như

```
doSomething(3, 5)
```

ta sử dụng các giá trị tượng trưng cho *a* và *b* là *a1* và *b1*. Và khi đó lời gọi hàm sẽ là:

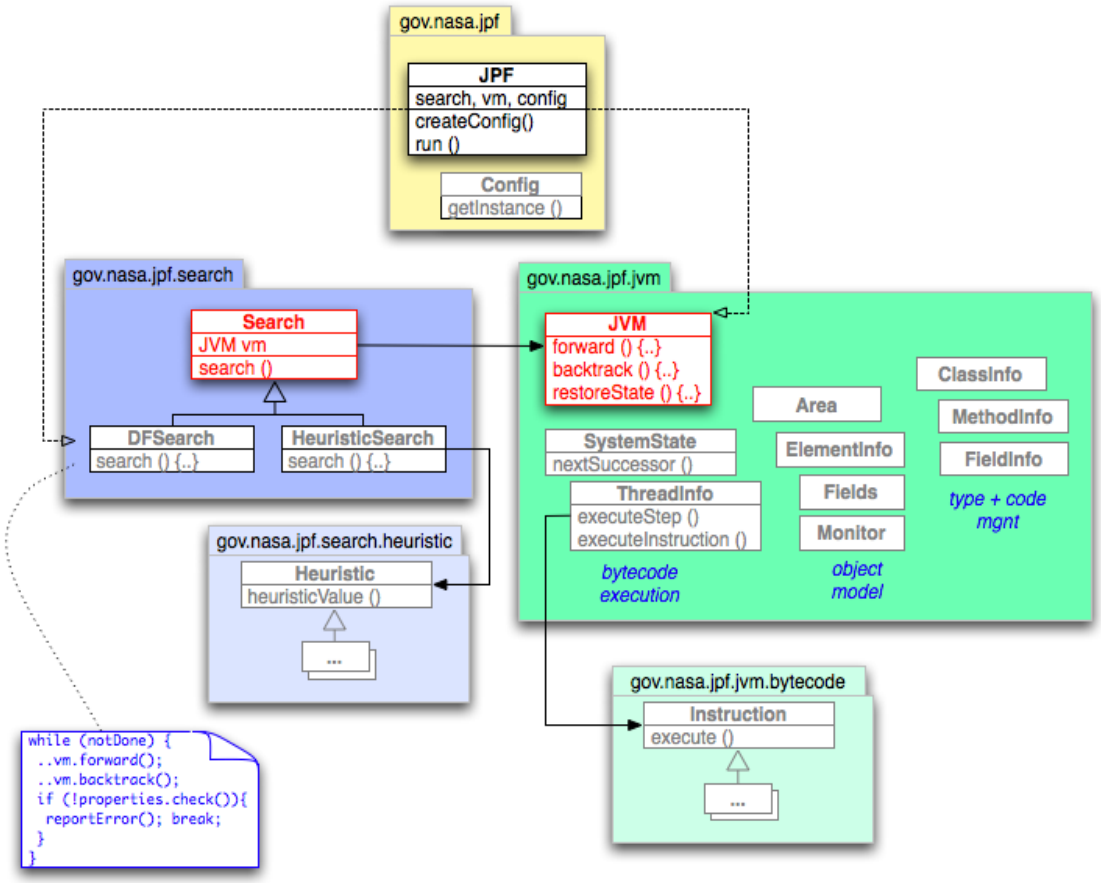
```
doSomething(a1, b1)
```

Chương 3. Java PathFinder (JPF)

Chương này sẽ giới thiệu về Java PathFinder và một mở rộng của nó giúp chúng ta có thể thực thi tượng trưng một chương trình viết bởi ngôn ngữ Java là Symbolic PathFinder.

3.1. Java PathFinder

Java PathFinder [3] là một môi trường hỗ trợ việc kiểm tra mô hình các chương trình Java ở dạng tệp Java Bytecode, gồm một máy ảo JVM và các kỹ thuật khác để giảm thiểu không gian trạng thái. Cho tới nay, JPF đã phát triển thêm rất nhiều ứng dụng và phần mở rộng khác nhau nhưng chức năng chính của JPF vẫn là kiểm chứng mô hình.



Hình 3.1 Thiết kế chính của JPF [1]

3.1.1. Cấu trúc chính của JPF

[1] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/design

Toàn bộ framework này được thiết kế xoay quanh hai thành phần chính là máy ảo JVM và đối tượng Search.

Máy ảo JVM chạy trên nền máy ảo Java thật, thực thi các chỉ thị Java Bytecode và đồng thời sinh ra các trạng thái cùng với đó là các lớp để quản lý không gian trạng thái này. JVM có thể kiểm tra việc trùng lặp các trạng thái xem trạng thái đó đã được thăm hay chưa, có thể lưu lại trạng thái để sau này có thể quay lại trạng thái đó và thực thi theo một đường khác.

Đối tượng Search được xem như là bộ điều khiển hoạt động của JVM. Nó có trách nhiệm hướng dẫn JVM khi nào phải tạo ra trạng thái mới hay khi nào nên quay lại một trạng thái cũ đã được sinh ra trước đó. Cứ như thế cho tới khi JVM thăm hết tất cả các không gian trạng thái thì thôi. Ngoài việc sử dụng thuật toán tìm kiếm theo độ sâu đơn giản, còn có các kiểu tìm kiếm kinh nghiệm khác. Chúng ta hoàn toàn có thể tự tạo ra đối tượng Search cho mình bằng cách kế thừa lớp Search này và cài đặt phương thức *search()* theo các thuật toán tìm kiếm mà ta mong muốn.

JPF áp dụng rất nhiều kỹ thuật khác nhau trong cài đặt của mình mà không thể nêu hết trong phạm vi báo cáo này. Do đó ta chỉ nêu ra ba kỹ thuật đã dùng trong luận văn này đó là Choice Generator, Property và Listener.

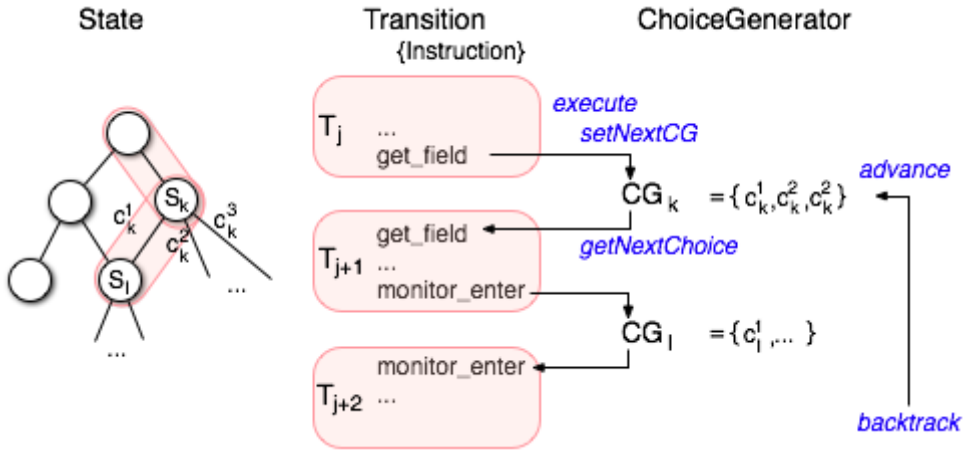
3.1.2. Choice Generator

Kiểm chứng mô hình thì điều cần thiết nhất là đưa ra được các lựa chọn chính xác với những ràng buộc về tài nguyên và môi trường thực thi của hệ thống để dẫn tới những trạng thái mà ta mong muốn. Choice Generator chính là công nghệ sinh ra các lựa chọn mà JPF sử dụng để đi qua toàn bộ không gian trạng thái. Lựa chọn ở đây có thể hiểu là tới một thời điểm nào đó chương trình phải chọn ra một giá trị tiếp theo trong một bộ giá trị có thể. Ví dụ như một giá trị ngẫu nhiên tiếp theo trong trường hợp sinh số ngẫu nhiên hay một luồng tiếp theo trong trường hợp lập lịch ở chương trình đa luồng. Nhưng trước hết ta phải hiểu một số thuật ngữ của JPF trước khi đi vào cụ thể công nghệ này:

- State - lưu giữ trạng thái thực thi hiện tại của chương trình và quá trình dẫn tới trạng thái này (path). Mọi State có một số định danh duy nhất và được đóng gói trong một thể hiện của lớp SystemState bao gồm ba thành phần sau:
    + KernelState – trạng thái thực thi hiện tại của chương trình.
    + Trail – Transition cuối cùng.
    + ChoiceGenerator hiện tại và kế tiếp – các đối tượng lưu giữ các lựa chọn để tạo ra các Transition khác nhau.

- Transition là một dãy các chỉ thị dẫn từ state này tới state tiếp theo. Mọi transition đều nằm trọn trong một thread. Có thể có nhiều transition đi ra từ một state nhưng không nhất thiết tới một thread khác.
- Choice (lựa chọn) là cái bắt đầu một transition mới.

Khi một chỉ thị thực thi và có yêu cầu phải sinh lựa chọn, nó sẽ làm các công việc sau: Tạo ra một đối tượng ChoiceGenerator mới chứa tất cả các khả năng lựa chọn có thể sau đó đặt đối tượng này là ChoiceGenerator cho Transition tiếp theo thông qua việc gọi phương thức *SystemState.setNextChoiceGenerator()*. Ở thời điểm này, JPF kết thúc Transition hiện tại, lưu lại trạng thái hiện thời của chương trình và bắt đầu một Transition mới bằng cách gọi phương thức *ChoiceGenerator.advance()* trên đối tượng ChoiceGenerator vừa mới tạo ở cuối Transition trước. Như vậy, mọi state sẽ chỉ có duy nhất một ChoiceGenerator tương ứng với nó và mỗi transition sẽ ứng với một giá trị choice sinh ra từ ChoiceGenerator đó. Mọi transition kết thúc khi có một chỉ thị phát sinh ra một ChoiceGenerator mới.



Hình 3.2 Trình tự của ChoiceGenerator khi thực thi chỉ thị get_field [2]

Hình 3.2 là một ví dụ về kỹ thuật này khi thực thi chỉ thị *get_field* trong đó choice ở đây là các thread. Nếu như không còn chỉ thị nào nữa hoặc đối tượng Search nhận thấy trạng thái này đã được thăm trước đó rồi, nó sẽ báo cho JVM quay lại một trạng thái cũ gần nhất và kiểm tra xem đối tượng ChoiceGenerator tương ứng với trạng thái đó có còn lựa chọn nào chưa dùng đến hay không bằng cách gọi *ChoiceGenerator.hasMoreChoice()*. Nếu còn nó lại lấy một giá trị mới bằng cách gọi *ChoiceGenerator.advance()* và bắt đầu một Transition mới đi ra từ trạng thái đó. Khi nào không còn một lựa chọn nào nữa thì JVM lại quay lại trạng thái trên nữa. Cứ như

[2] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/choicegenerator

vậy, thủ tục này được lặp lại cho tới khi tất cả các ChoiceGenerator đều đã dùng hết các lựa chọn của mình. Khi đó, quá trình tìm kiếm kết thúc.

3.1.3. Property

Property là các tính chất hay ràng buộc mà chương trình phải tuân theo. Có ba cách để kiểm tra các tính chất này với JPF.

Cách thứ nhất là đối với các tính chất đơn giản và chỉ phụ thuộc vào giá trị của các dữ liệu trong phạm vi chương trình. Nếu như có sự vi phạm các tính chất này xảy ra thì nó sẽ được bắt bởi lớp *NoUncaughtExceptionProperty*. Nhược điểm của phương pháp này là nó yêu cầu truy cập vào mã nguồn của chương trình thực thi và có thể tăng không gian trạng thái lên đáng kể.

Cách thứ hai là sử dụng giao diện (interface) *gov.nasa.jpf.Property*. Interface này có chứa phương thức *Property.check()* được kiểm tra sau khi kết thúc mỗi transition. Trong phương thức này chúng ta lưu giữ tính chất cần kiểm tra. Nếu như có vi phạm nó sẽ trả về giá trị *false* khi đó quá trình tìm kiếm sẽ bị ngưng lại và các vi phạm sẽ được in ra. Bản thân JPF cũng có một số tính chất như *NoUncaughtExceptionsProperty* hay *NotDeadlockedProperty*. Ngoài ra ta có thể tự định nghĩa thêm các tính chất khác cho mình bằng cách tạo ra các lớp thực thi interface này:

```
public interface Property extends Printable {
      boolean check (Search search, VM vm);
      String getErrorMessage();
}
```

Sau đó gắn nó vào đối tượng Search của JPF bằng cách gọi

jpf.getSearch().addProperty()

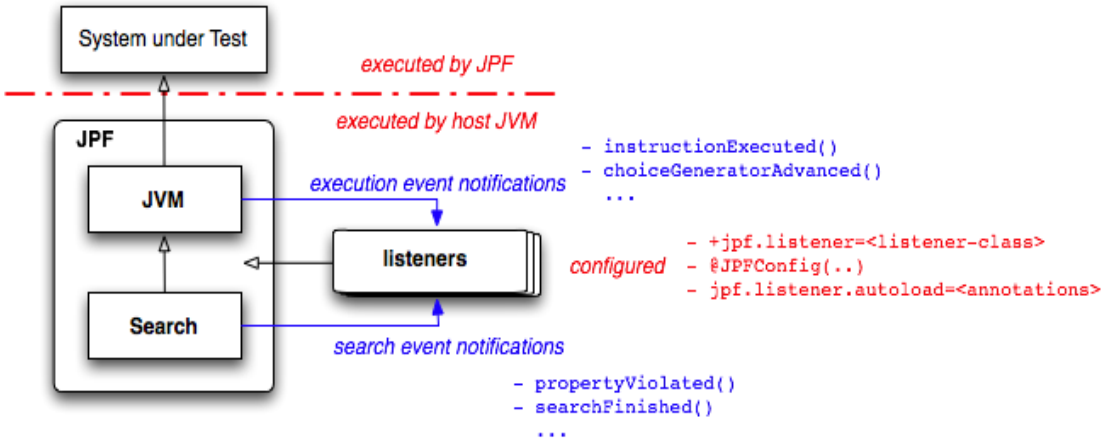
Cách thứ ba là sử dụng các listener và sẽ được nói kỹ hơn ở phần tiếp theo đây.

3.1.4. Listener

Đây là kỹ thuật vô cùng quan trọng để viết các ứng dụng mở rộng cho JPF. Các listener cho phép theo dõi, tương tác và mở rộng việc thực thi của JPF ở thời điểm chạy chương trình nên nó không làm thay đổi phần nhân của JPF.

JPF sử dụng Observer pattern [11] để hỗ trợ việc tạo ra các listener. Kỹ thuật này có thể hiểu là JPF cho phép chúng ta tạo ra các listener và gắn nó vào thể hiện của JPF. Sau đó, khi chương trình thực thi thì mỗi khi có một sự kiện nào đó xảy ra nó sẽ báo cho listener biết. Các sự kiện này bao quát gần như toàn bộ quá trình thực thi của JPF từ những sự kiện ở mức thấp như thực thi các chỉ thị (*instructionExecuted*) cho tới

những sự kiện ở mức cao như khi đã hoàn thành việc tìm kiếm chẳng hạn (*searchFinished*).



Hình 3.3 JPF Listeners [3]

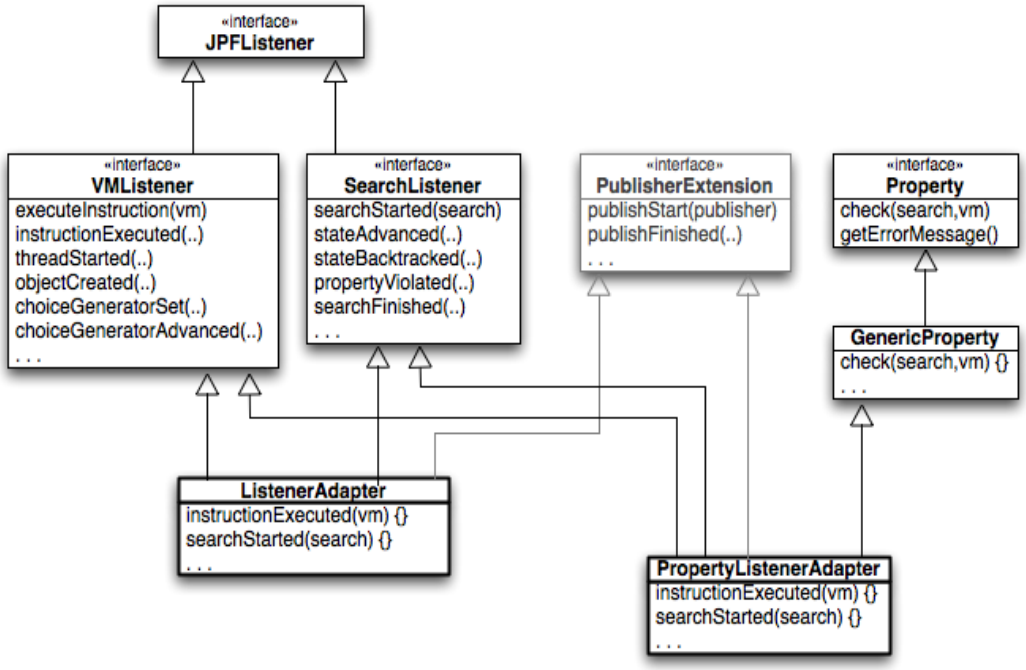
Mỗi sự kiện tương ứng với một nguồn sinh ra sự kiện đó, như đã nói ở trên là đối tượng JVM và Search, do đó có hai loại listener đó là VMListener và SearchListener. Trong tập sự kiện này, chúng ta phải xác định được mình cần biết khi có những sự kiện nào và ghi rõ trong listener của mình. Từ các nguồn tương ứng (JVM hoặc Search) JPF cũng cung cấp các API để truy xuất rất nhiều thông tin về sự kiện cũng như về trạng thái bên trong của JPF mà ta có thể sử dụng để làm những gì mình muốn.

Để tạo ra các listener này, JPF đưa ra hai interface tương ứng với hai đối tượng nguồn của các sự kiện là SearchListener và VMListener. Hai interface đưa ra các phương thức ứng với từng sự kiện sẽ báo cho listener biết. SearchListener thường được dùng để báo cáo về các sự kiện diễn ra trong suốt quá trình tìm kiếm trên không gian trạng thái còn VMListener thì thường báo cáo các sự kiện là các hoạt động của máy ảo. Tuy nhiên listener do ta tạo ra thường phải thực thi cả hai interface này và do đó phải cài đặt tất cả các phương thức ứng với các sự kiện trong đó. Mà ta chỉ muốn theo dõi một số sự kiện trong số đó thôi, do đó JPF tạo ra các lớp Adapter. Lớp này thực thi cả hai interface và cài đặt tất cả các phương thức trong hai interface đó nhưng với một thân phương thức rỗng không làm bất kỳ công việc nào cả. Việc tạo ra listener cho một mục đích cụ thể bây giờ đơn giản hơn nhiều đó là tạo ra một lớp kế thừa các lớp Adapter này và cài đặt chồng (override) phương thức ứng với sự kiện muốn theo dõi. Có hai lớp Adapter như vậy thường được sử dụng:

[3] http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/listener

ListenerAdapter: Lớp này thực thi VMListener và SearchListener, ngoài ra còn thêm một interface nữa đó là PublisherExtension để ta đưa thêm các thông tin báo cáo vào hệ thống báo cáo của JPF.

PropertyListenerAdapter: Lớp này cũng giống như lớp ListenerAdapter, ngoài ra còn thực thi thêm interface Property đã đề cập ở phần trên. Sở dĩ có thêm interface Property là vì khi ta viết các ứng dụng mở rộng cho JPF ngoài việc viết các listener ta thường muốn định nghĩa các tính chất mới cho chương trình của mình.



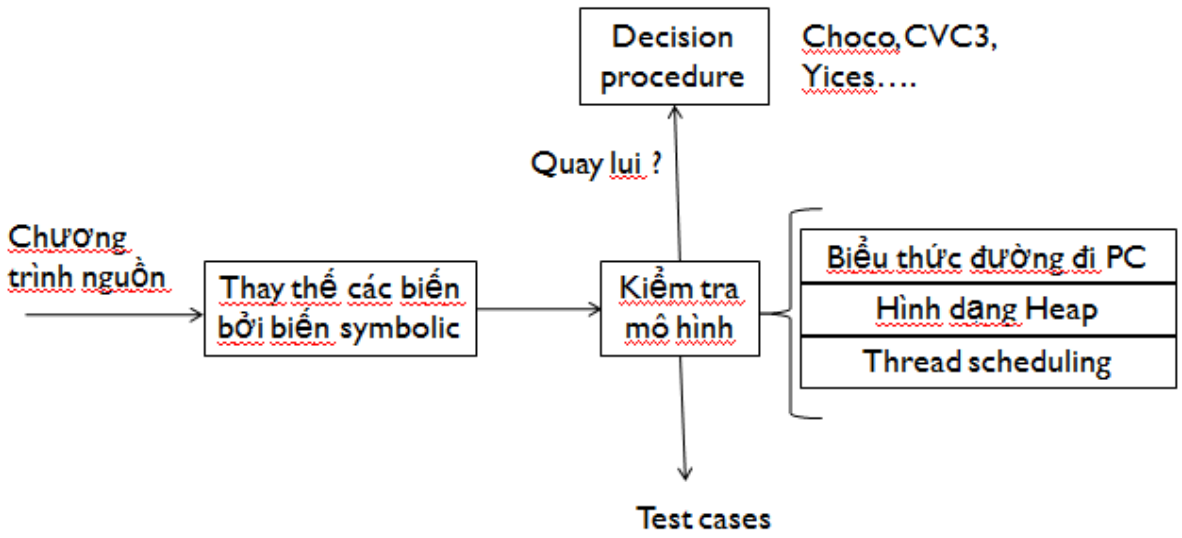
Hình 3.4 Các loại Listener

Thông thường, việc chỉ được báo cáo rằng có sự kiện nào đó vừa xảy ra là chưa đủ. Ta thường muốn biết thêm thông tin khác về chương trình và trạng thái hiện thời của hệ thống. Do đó trong mỗi phương thức của SearchListener và VMListener đều có một tham số là đối tượng nguồn tương ứng của sự kiện đó (Search hoặc JVM) để từ đối tượng này ta có thể truy xuất thêm các thông tin cần thiết từ JPF.

3.2. Symbolic PathFinder (SPF)

Symbolic PathFinder là một ứng dụng mở rộng của JPF. Framework bao gồm một bộ thông dịch Jave Bytecode không chuẩn và sử dụng JPF để thực thi tượng trưng. Sở dĩ gọi là không chuẩn bởi vì nó không giống với ý nghĩa thông thường của thực thi cụ thể (concrete excution) dựa trên mô hình ngăn xếp được miêu tả trong đặc tả của JVM. Làm được điều này là vì JPF đọc và phân tích tệp .class và thông dịch mã Java Bytecode ra các chỉ thị. Mặc định thì các chỉ thị này được thông dịch theo đúng như đặc tả của JVM nhưng có thể thay đổi ý nghĩa đó bằng cách sử dụng lớp

InstructionFactory. SPF tạo ra lớp SymbolicInstructionFactory bao gồm các chỉ thị giúp cho việc thông dịch mã Java Bytecode để thực thi tượng trưng.



Hình 3.5 Kiến trúc tổng quát của Symbolic PathFinder

Khi thực thi tượng trưng tới những biểu thức điều kiện thì nó sẽ tạo ra một lựa chọn sinh ra bởi lớp PCChoiceGenerator để rẽ nhánh chương trình và đồng thời nó thêm vào PathCondition điều kiện vừa mới gặp. Mỗi một PC tương ứng với một giá trị lựa chọn được sinh ra bởi PCChoiceGenerator. PC được kiểm tra bởi một constraint solver xem có thể thực hiện được không. Nếu không thì JPF sẽ quay lại trạng thái trước đó gần nhất (backtrack).

SPF sử dụng hai constraint solver là Choco [13] để giải các ràng buộc với số nguyên và số thực và CVC3 [12] để giải các hàm logic vị từ. SPF cũng đồng thời cung cấp các interface để người sử dụng có thể chọn một trong hai solver này.

SPF dùng JPF để thực thi tượng trưng; Tuy nhiên nó không yêu cầu JPF phải thực hiện kiểm tra việc trùng lặp trạng thái (một trong những đặc điểm của các công cụ kiểm thử mô hình như JPF nhằm làm giảm không gian trạng thái loại đi các trạng thái giống nhau). Nhưng đối với những chương trình có vòng lặp hoặc đệ quy thì điều này có thể dẫn tới bùng nổ trạng thái. Để giới hạn điều này, SPF đặt một giới hạn về độ sâu tìm kiếm cho JPF hoặc là giới hạn số ràng buộc có thể trong một PC.

Chương 4. Cài đặt

Chương này trình bày các cài đặt để mở rộng công cụ *jpf-ltl* để thực hiện kiểm chứng công thức logic thời gian tuyến tính các chương trình Java vô hạn trạng thái; Bao gồm: cài đặt thêm các mệnh đề nguyển tử, kết hợp thuật toán DDFS với kiểm tra xếp gộp và thêm các ví dụ kiểm chứng.

4.1. Kiểm chứng công thức LTL

LTL được sử dụng rộng rãi để đặc tả các tính chất của những hệ thống song song và kiểm thử mô hình được xem như là một kỹ thuật được sử dụng rộng rãi để kiểm tra một hệ thống có tuân theo đặc tả LTL hay không. Vấn đề kiểm tra có thể được phát biểu như sau: Cho trước một chương trình P và một công thức logic thời gian tuyến tính F, kiểm tra rằng tất cả những dãy thực thi vô hạn của P thoả mãn F. Courcoubetis và cộng sự đã đề xuất một giải pháp giải quyết vấn đề này và đã được ứng dụng trong một số mô hình kiểm chứng nổi tiếng [14]. Giải pháp này gồm các bước chính như sau:

- Xây dựng máy trạng thái Buchi $A(\neg F)$ cho công thức nghịch đảo của F. $A(\neg F)$ sẽ đón nhận tất cả các dãy vô hạn mà vi phạm F.
- Xây dựng một máy trạng thái Buchi $A(G)$ là tích của $A(P)$ và $A(\neg F)$ với $A(P)$ là một máy trạng thái được gán nhãn thể hiện chương trình P.
- Kiểm tra tính rỗng của $A(G)$. Chương trình P thoả mãn công thức F khi và chỉ khi $A(G)$ rỗng. Chúng ta sử dụng thuật toán thăm sâu trước lồng nhau (DDFS – Double depth first search) [4] để kiểm tra tính rỗng của $A(G)$.

Công cụ *jpf-ltl* là một mở rộng của JPF để kiểm tra mô hình các tính chất logic thời gian tuyến tính các chương trình Java. Tuy nhiên, công cụ này chưa hỗ trợ việc kiểm tra tượng trưng đối với các dãy thực thi vô hạn. Luận văn này trình bày phương pháp mở rộng công cụ nói trên để có thể tích hợp thực thi tượng trưng vào việc kiểm tra mô hình các tính chất logic thời gian tuyến tính cho các dãy thực thi vô hạn.

4.2. Công cụ *jpf-ltl*

Công cụ *jpf-ltl* được xây dựng trên JPF để cho phép kiểm chứng các chương trình Java có thỏa mãn các công thức LTL hay không. Công cụ này cùng với toàn bộ mã nguồn có thể được tìm thấy từ trên trang web phát triển JPF của trung tâm NASA Ames[4]. *jpf-ltl* đọc trực tiếp các công thức LTL từ các chú thích (annotation) trong mã

[4] http://babelfish.arc.nasa.gov/trac/jpf

nguồn Java, ví dụ như `@LTLSpec("[]!(w0 && w1 && w2)")` hoặc từ một tệp riêng biệt với chú thích `"@LTLSpec("fileName")`. Các công thức này sau đó được cho vào bộ phân tích ngữ nghĩa để dịch sang một automat. Thêm vào đó, *jpf-ltl* thêm một đối tượng tìm kiếm mới DDFS vào JPF để kiểm tra các chương trình có không gian trạng thái lớn. Các trường hợp lỗi có thể được đưa ra bởi *jpf-ltl* theo chuẩn bằng cách in ra ngăn xếp của các trạng thái đã thăm.

4.2.1. Cú pháp của các công thức LTL trong công cụ *jpf-ltl*

Trong công trình này, công cụ ANTLR[5] được sử dụng để tự động sinh ra bộ phân tích ngữ nghĩa (parser) và bộ phân tích cú pháp (lexer) cho các công thức LTL. Việc phải làm là định nghĩa một bộ ngữ pháp (LTL grammar) biểu diễn chính xác các công thức này. Sau khi đưa phân tích ngữ pháp bằng parser kể trên, chúng ta sẽ thu được một đối tượng của lớp Formula<String> tượng trưng cho một công thức LTL. Sau đây là đặc tả của các công thức LTL được đoán nhận bởi bộ phân tích ngữ pháp trong công trình này:

4.2.2. Các toán tử LTL được hỗ trợ

AND	:	/\	&&
OR	:	\/	||
UNTIL	:	U	
WEAK_UNTIL	:	W	
RELEASE	:	V	
WEAK_RELEASE	:	M	
NOT	:	!	
NEXT	:	X	
ALWAYS	:	[]	
EVENTUALLY	:	<>	
IMPLIES	:	->	

4.2.3. Các mệnh đề nguyên tử (atomic proposition) được hỗ trợ

Với mục đích kiểm chứng chương trình viết bằng Java nên ngoài các biến và biểu thức logic (boolean variable và boolean expression), chúng ta cần phải kiểm tra

[5] http://www.antlr.org/

xem một lời gọi hàm có được gọi ở một thời điểm xác định hay không. Do đó, tên phương thức (method signature) cũng được hỗ trợ như là một mệnh đề nguyên tử.

- Tên của một phương thức (method signature)

Tên của một phương thức có thể là một mệnh đề nguyên tử với điều kiện nó phải đầy đủ (bao gồm tên của gói – package – nếu có, tên của lớp tiếp đến là tên của phương thức và kiểu của các đối số của nó. Ví dụ: *packageName.ClassName.methodName(int, float, String, T)* hoặc *ClassName.methodName(T)*

- Biến logic (boolean variable)

Một mệnh đề nguyên tử cũng có thể là một biến logic.

- Một trường trong một lớp: Cũng giống như tên của phương thức, nó phải là một tên đầy đủ bao gồm tên gói, tên lớp rồi mới tới tên trường. Ví dụ: *packageName.ClassName.field*
- Một biến cục bộ trong một phương thức: Nó phải bao gồm tên đầy đủ của phương thức mà trong đó nó được định nghĩa tiếp theo là tên của biến cục bộ đó. Ví dụ: *packageName.ClassName.methodName(T).var*

- Mệnh đề quan hệ (relation)

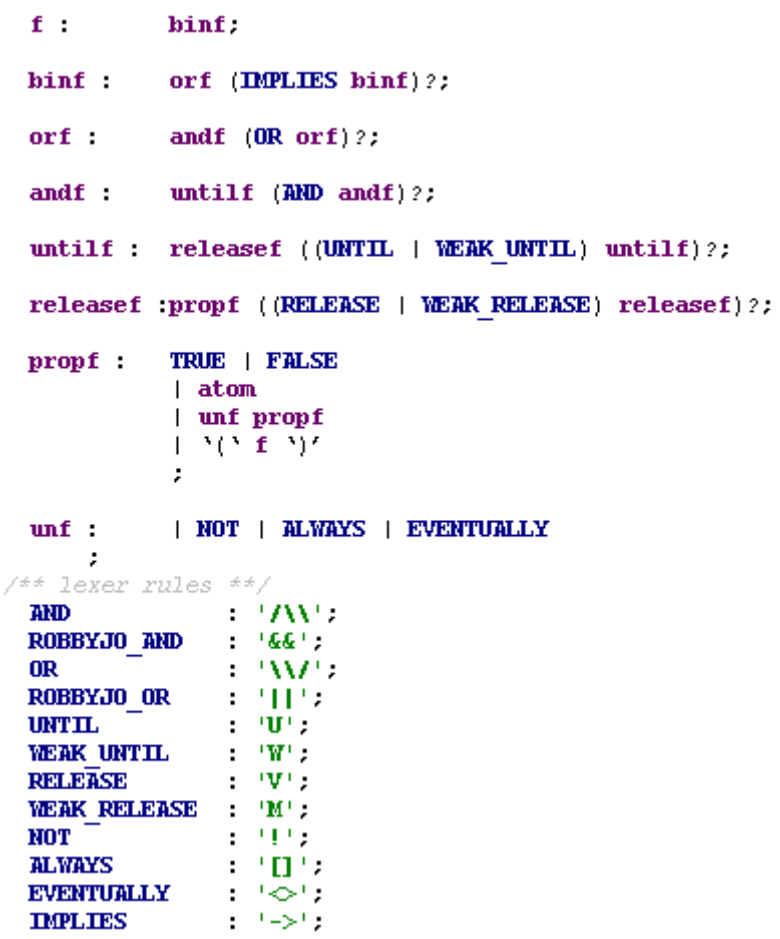
Một mệnh đề nguyên tử cũng có thể là bất kỳ mệnh đề quan hệ nào giống như biểu thức logic trong ngôn ngữ Java. Ví dụ như: *(x+y)*z - 3.0 > u/5*

Trong đó: *x, y, z* là các biến có cú pháp được định nghĩa giống như biến logic ở trên.

Ngữ nghĩa của các công thức LTL được định nghĩa theo chuẩn: Các biến logic Java và các biểu thức logic được tính từ các giá trị lấy trong các trạng thái của JPF. Một lời gọi phương thức là đúng trong một trạng thái nếu như phương thức tương ứng với nó được gọi trong một trạng thái cụ thể.

4.2.4. Cú pháp LTL

Một công thức LTL *f* có thể chứa bất kì mệnh đề nguyên tử nào và được kết hợp bởi các toán tử logic và toán tử thời gian.



Hình 4.1 Cú pháp LTL

Trong đó, *atom* là mệnh đề nguyên tử có cú pháp như *Hình 4.2*

```
atom
:
| method
| var
| exp ('==' | '!=' | '>=' | '<=' | '>' | '<') exp
;

method
:ID ('.' ID)* (
( '('(type ',' )* type ')')
|
'(' ')'
);

type
: ID ('[]')*;

exp
: mult ( '+' mult | '-' mult )*;

mult
: factor ('*' factor | '/' factor)*;

factor
: '(' exp ')'
| var
| number
;

var
: (
( ID('.' ID)*)
| (method '.' ID)
)
('[' INT ']')* ( '#' INT )?
      ;
number
  :  INT | FLOAT | '+' factor | '-' factor;


ID  : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;

INT : '0'..'9'+ ;

FLOAT
  :   ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
  |   '.' ('0'..'9')+ EXPONENT?
  |   ('0'..'9')+ EXPONENT
  ;


EXPONENT : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;
```

Hình 4.2 Cú pháp mệnh đề nguyên tử

4.3. Kiểm chứng mô hình các chương trình có không gian trạng thái lớn

Kiểm chứng mô hình các công thức LTL với các dãy thực thi vô hạn được thực hiện bằng cách tính tích của Buchi biểu diễn hệ thống với Buchi biểu diễn phủ định của công thức LTL cần kiểm tra. Tích này được xây dựng ngay trong thời gian thực thi

của hệ thống sử dụng thuật toán DDFS. Thuật toán DDFS được cài đặt bằng cách phát triển một chiến lược tìm kiếm mới cho JPF để gán nhãn cho các trạng thái của tích hai automat và để cài đặt thủ tục quay lui của tích này [6]. Trong trường hợp các chương trình có không gian trạng thái lớn, chúng ta có thể thực thi tượng trưng và khi đó, automat yêu cầu việc kiểm tra sự tương đương giữa các trạng thái tượng trưng của chương trình ở mỗi bước kiểm tra. Hay nói cách khác, chúng ta phải kiểm tra xem một trạng thái tượng trưng đã được thăm hay chưa sử dụng kỹ thuật kiểm tra xếp gộp [11].

4.3.1. DDFS

DDFS được cài đặt để làm chiến lược tìm kiếm mới cho JPF (*gov.nasa.jpf.ltl.infinite.DDFSearch*). Tư tưởng của DDFS là tìm kiếm một vòng mà trong đó có chứa trạng thái kết thúc :

- Giai đoạn 1 (*dfs1( )*) : Duyệt qua các trạng thái để tìm một trạng thái kết thúc *E*.
- Giai đoạn 2 (*dfs2()*) : Từ trạng thái kết thúc *E* này, duyệt tiếp các trạng thái để tìm một vòng quay lại chính trạng thái kết thúc *E*. Khi đó ta sẽ có một phản ví dụ, hệ thống vi phạm đặc tả.

Cụ thể, chúng ta sẽ dùng hai tập *dfs1Table* và *dfs2Table* để lưu các trạng thái đã được duyệt tương ứng bởi *dfs1()* và *dfs2()*.

- *dfs1()* được cài đặt như *Hình 4.3*:
    + Đánh dấu trạng thái hiện tại đã được duyệt vào *dfs1Table*.
    + Trong khi còn có thể chuyển đến trạng thái tiếp theo (*forward()*), kiểm tra trạng thái tiếp theo, nếu đã được duyệt bởi *dfs1()* (tức là đã có trong *dfs1Table*) thì quay lui (*backtrack()*), ngược lại thì đánh dấu trạng thái đó, và thực hiện *dfs1()*.
    + Nếu không thể chuyển sang thái tiếp theo và trạng thái hiện tại là trạng thái kết thúc, thực hiện *dfs2()* để tìm một vòng lặp chứa trạng thái kết thúc này.

```
75      public void dfs1() {
76          recordVisit(dfs1Stack);
77          recordVisit(dfs1Table);
78
79          while (true) {
80              if (forward()) {
81                  if (seenBefore(dfs1Table, 1)) {
82                      backtrack();
83                  }
84                  else {
85                      recordVisit(dfs1Stack);
86                      recordVisit(dfs1Table);
87
88                      if (isPropertyViolated()) {
89                          break;
90                      }
91                  }
92              }
93              else {
94                  if (inAcceptingState() && dfs2()) {
95                      prop.setViolated();
96                      break;
97                  }
98                  if (depth == 0) {
99                      terminate();
100                     break;
101                 }
102
103                 dfs1Stack.pop();
104                 backtrack();
105             }
106         }
107     }
```

Hình 4.3 Cài đặt của dfs1()

- *dfs2()* được cài đặt như *Hình 4.4*:
    + Nếu trạng thái hiện tại đã được duyệt bởi *dfs2()* (đã có trong *dfs2Table*), quay lui về trạng thái trước. Ngược lại, đánh dấu vào *dfs2Table*.
    + Nếu trạng thái hiện tại đã đc duyệt bởi *dfs1()* thì chúng ta có một vòng lặp ở đây. Nghĩa là hệ thống vi phạm đặc tả.
    + Ngược lại, duyệt sang trạng thái tiếp theo.

```
108
109     protected boolean dfs2() {
110         final int startDepth = depth;
111         if (vm.getChoiceGenerator() != null) {
112             vm.getChoiceGenerator().reset();
113         }
114
115         while (true) {
116             if (seenBefore(dfs2Table, 2)) {
117                 if (depth == startDepth) {
118                     return false;
119                 }
120                 backtrack();
121             }
122             else {
123                 recordVisit(dfs2Table);
124             }
125
126             if (seenBefore(dfs1Stack, 1) && depth > startDepth) {
127                 return true;
128             }
129
130             while (!forward()) {
131                 if (depth == startDepth) {
132                     return false;
133                 }
134                 backtrack();
135             }
136         }
137     }
```

Hình 4.4 Cài đặt của dfs2()

4.3.2. Thực thi tượng trưng cho các dãy thực thi vô hạn

- Chia nhỏ các trạng thái tượng trưng

Khi ta đánh giá một biểu thức nguyên tử (atomic proposition) ở chế độ thực thi bình thường, nó luôn luôn trả về một giá trị nhất định *true* hoặc *false*. Do mỗi trạng thái tượng trưng có thể đại diện cho một nhóm các trạng thái, nên kết quả trả về khi đánh giá giá trị của một biểu thức tượng trưng có thể không trả về một giá trị nhất định. Để giải quyết vấn đề này, chúng ta cần chia nhỏ các trạng thái tượng trưng để đảm bảo rằng mỗi biểu thức nguyên tử luôn luôn có một giá trị xác định.

Chúng ta hãy cùng xem xét ví dụ sau: Giả sử chúng ta cần đánh giá giá trị của biểu thức tượng trưng $x > 0$ với biểu thức đường đi tương ứng có chứa $x < 10$. Trong khi thực thi, biểu thức đường đi nói trên sẽ được đánh giá hoặc nhận giá trị *true*, hoặc nhận giá trị *false* tương ứng với mỗi đường đi tương ứng có thể. Do đó chúng ta chia nhỏ trạng thái tượng trưng này thành 2 trạng thái, một trạng thái đặc trưng bởi giá trị *true* của biểu thức đường đi, trạng thái còn lại đặc trưng bởi giá trị *false*, tức là $x \geq 10$ như trong ví dụ trên.

Khi chương trình thực thi theo đường đi ứng với $x < 10$, giá trị của biểu thức nguyên tử x > 0 cũng không xác định duy nhất: nó nhận giá trị *true* với $0 < x < 10$ và *false* với $x \geq 10$. Do đó chúng ta cần tiếp tục phân chia trạng thái tượng trưng bằng cách kết hợp giữa ràng buộc của biểu thức nguyên tử với biểu thức đường đi.

- Kiểm tra xếp gộp (Subsumption checking)

Một vấn đề đặt ra khi thực thi tượng trưng trên các chương trình có vòng lặp đó là có thể gây ra các quá trình lặp vô hạn. Có hai giải pháp chính để giải quyết vấn đề này:

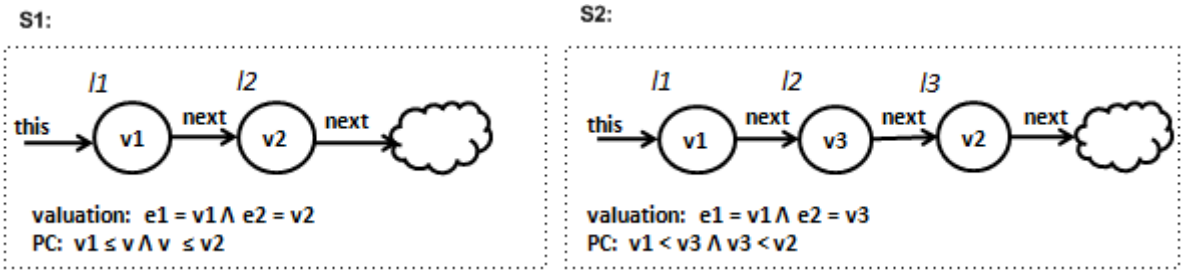
- Giới hạn độ sâu tìm kiếm: Phương pháp này dễ thực hiện nhưng đôi khi cho độ chính xác không cao.
- Sử dụng phương pháp kiểm tra xếp gộp để kiểm tra khi nào một trạng thái tượng trưng đã được thăm hay chưa.

Mỗi trạng thái tượng trưng S bao gồm:

- Hình dạng heap H của trạng thái đó
- Biểu thức đường đi PC

Muốn kiểm tra trạng thái tượng trưng *S2* có được xếp gộp vào trạng thái tượng trưng *S1* hay không, trước hết ta cần so sánh hình dạng heap của hai trạng thái này. Sau đó so sánh ràng buộc dữ liệu chứa trong hai trạng thái.

Ví dụ: xem xét hai trạng thái *S1* và *S2* như trong *Hình 4.5*, với *S1* và *S2* là những thể hiện của một danh sách liên kết đơn



Hình 4.5 Trạng thái S2 được xếp gộp vào S1

Hình đám mây mô tả một node chưa được khởi tạo giá tri. Ở đây ta có hình dạng shape *H2* sẽ được xếp gộp vào *H1*. Ta đánh giá ràng buộc số học của các trường trong mỗi trạng thái như sau:

S1: $e1 = v1 \wedge e2 = v2$ có nghĩa rằng mỗi giá trị của trường gán nhãn bởi *l1* (tức *e1*) sẽ có giá trị tượng trưng *v1* và giá trị của trường gán nhãn *l2* (tức *e2*) sẽ có giá trị *v2*. Biểu thức đường đi PC có thể chứa các giá tượng trưng xuất hiện trong heap hoặc không chưa tuỳ thuộc vào đường đi của chương trình. Để chứng minh *S2* được xếp gộp vào *S1*, ta cần chứng minh rằng:

$\exists v,v1,v2,v3($ *valuation2* $\wedge$ *PC2* $\Rightarrow$ *valuation1* $\wedge$ *PC1)* luôn đúng

$\Leftrightarrow \exists v,v1,v2,v3($ $e1 = v1 \wedge e2 = v3 \wedge v1 < v3 \;\; v3 < v2 \Rightarrow e1 = v1 \wedge e2 = v2 \wedge v1 \leq v \;\; \wedge v \leq v2)$ luôn đúng

$\Leftrightarrow e1 < e2 \Rightarrow e1 \leq e2$ luôn đúng (*).

Do (*) đúng nên S2 được xếp gộp vào S1.

Việc chứng minh này được thực hiện bằng cách đưa biểu thức trên vào một Solver, ở đây chúng tôi sử dụng CVC3 solver.

4.3.3. Kiểm chứng tính chất LTL

Sau đây là một số kết quả đạt được của công cụ *jpf-ltl* thông qua các ví dụ cụ thể là kiểm tra các tính chất LTL.

```
17 @LTLSpec("[](foo())")
18 public class Raimondi {
19     @Symbolic("true")
20     static int y = 0;
21
22     public static void main(String[] args) {
23         test(0);
24     }
25
26     public static void test(int x) {
27         while (true) {
28             done();
29             if (y != 1) {
30                 foo();
31             }
32         }
33     }
34
35     public static void done() {}
36     public static void foo() {}
37 }
```

Hình 4.6 Ví dụ thuộc tính safety

- Thuộc tính safety

Đoạn mã ngồn Java ở *Hình 4.6* mô phỏng một chương trình Java có dãy thực thi vô hạn. Chúng ta sẽ kiểm tra tính safety với công thức LTL cụ thể:

[](foo())

Và thực thi tượng trưng với biến *y*, với chú thích *@Symbolic("true")* ở dòng 21.

```
====================================================== error #1
gov.nasa.jpf.ltl.infinite.Property
Violated LTL property for infinite.symbolic.Raimondi:
        [](foo())

====================================================== snapshot #1
thread index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,priority=5,lockCount=0,suspendCount=0
  call stack:
        at infinite.symbolic.Raimondi.test(Raimondi.java:28)
        at infinite.symbolic.Raimondi.main(Raimondi.java:23)


====================================================== results
error #1: gov.nasa.jpf.ltl.infinite.Property "Violated LTL property for infinite.symbolic.Raimon..."

====================================================== statistics
elapsed time:       0:00:06
states:             new=9, visited=11, backtracked=2, end=0
search:             maxDepth=9, constraints=0
choice generators:  thread=18, data=3
heap:               gc=1, new=266, free=2
instructions:       2891
max memory:         81MB
loaded code:        classes=71, methods=888
```

Hình 4.7 Kết quả công thức [](foo())

Nếu bỏ chú thích ở dòng 21 thì chúng ta sẽ có một thực thi cụ thể (concrete) và điều kiện *y!=1* đúng. Nghĩa là chương trình thỏa mãn công thức LTL *[](foo())*.

```
17 @LTLSpec("[](<> done() && <> foo())")
18 public class Raimondi {
19     @Symbolic("true")
20     static int y = 0;
21
22     public static void main(String[] args) {
23         test(0);
24     }
25
26     public static void test(int x) {
27         while (true) {
28             done();
29             if (y != 1) {
30                 foo();
31             }
32         }
33     }
34
35     public static void done() {}
36     public static void foo() {}
37 }
```

Hình 4.8 Ví dụ thuộc tính liveness

- Thuộc tính liveness

Với đoạn mã ngồn ở *Hình 4.8*, chúng ta kiểm chứng chương trình có thỏa mãn công thức LTL *[](<>done() && <>foo())* hay không. Nếu bỏ dòng 21, chương trình sẽ thỏa mã với công thức LTL.

- Thuộc tính fairness

```
18 public class Raimondi {
19     @Symbolic("true")
20     static int y = 0;
21
22     public static void main(String[] args) {
23         test(0);
24     }
25
26     public static void test(int x) {
27         while (true) {
28             done();
29             if (y != 1) {
30                 foo();
31             }
32         }
33     }
34
35     public static void done() {}
36     public static void foo() {}
37 }
```

Hình 4.9 Ví dụ thuộc tính fairness

Với đoạn mã nguồn tương tự, chúng ta có kết quả tương ứng với công thức LTL *[](y!=1) -> <>foo())* như sau:

```
gov.nasa.jpf.ltl.infinite.Property
Violated LTL property for infinite.symbolic.Raimondi:
        [](y != 1) -> foo())

====================================================== snapshot #1
thread index=0,name=main,status=RUNNING,this=java.lang.Thread@0,target=null,priority=5,lockCount=0,suspendCount=0
  call stack:
        at infinite.symbolic.Raimondi.test(Raimondi.java:28)
        at infinite.symbolic.Raimondi.main(Raimondi.java:23)


====================================================== results
error #1: gov.nasa.jpf.ltl.infinite.Property "Violated LTL property for infinite.symbolic.Raimon..."

====================================================== statistics
elapsed time:       0:00:06
states:             new=9, visited=11, backtracked=2, end=0
search:             maxDepth=9, constraints=0
choice generators:  thread=18, data=3
heap:               gc=1, new=266, free=2
instructions:       2891
max memory:         81MB
loaded code:        classes=71, methods=888
```

Hình 4.10 Kết quả công thức [](y!=1) -> <>foo())

Chương 5. Kết luận

Luận văn "Xây dựng phần mở rộng kiểm chứng thuộc tính logic thời gian tuyến tính cho Java Pathfinder" đã phát triển và mở rộng công cụ *jpf-ltl* để giải quyết bài toán kiểm chứng mô hình phần mềm sau: Cho trước một chương trình *P* và một công thức logic thời gian tuyến tính *F*, kiểm tra rằng tất cả những dãy thực thi vô hạn của *P* thoả mãn *F* hay không.

Luận văn đã trình bày những cơ sở lý thuyết cơ bản liên quan đến kiểm chứng mô hình phần mềm, nghiên cứu và áp dụng kỹ thuật mở rộng Java PathFinder, một môi trường hỗ trợ kiểm chứng mô hình các chương trình Java ở dạng tệp Java Bytecode.

Luận văn đã đề cập đến các hướng lý thuyết đang được sử dụng phổ biến hiện nay như automat Buchi để đoán nhận các xâu vô hạn, Logic thời gian tuyến tính để biểu diễn các biểu thức logic có tính thời gian, sự chuyển đổi giữa Buchi và Logic thời gian tuyến tính.

Luận văn đã trình bày về thực thi tượng trưng và công cụ hỗ trợ Symbolic PathFinder, công cụ hỗ trợ thực thi tượng trưng cho Java PathFinder.

Luận văn đã tập trung vào việc nghiên cứu và cài đặt Thuật toán DDFS và Kiểm tra sự xếp gộp trạng thái để mở rộng công cụ *jpf-ltl* đã được phát triển trước đó. Ngoài ra, luận văn cũng đã cài đặt thêm các mệnh đề logic nguyên tử còn thiếu trong *jpf-ltl*, như các so sánh *!=*, >=, <=. Với công cụ này, chúng ta đã có thể kiểm chứng các tính chất logic thời gian tuyến tính như tính safety, tính liveness và tính fairness của một chương trình Java có không gian trạng thái lớn.

Do khuôn khổ có hạn về thời gian cũng như lượng kiến thức có được nên còn một số vấn đề mà luận văn phải tiếp tục hoàn thiện và phát triển trong thời gian tới như:

- Cải tiến hiệu quả của kiểm tra xếp gộp trạng thái.
- Áp dụng vào các ứng dụng thực tế.

Tài liệu tham khảo

[1] Amir Pnueli, *The temporal logic of programs*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS), 1977, pp. 46–57.

[2] J. R. Buchi, *On a decision method in restricted second order arithmetic*, Z. Math. Logik Grundlag. Math,1960, pp. 66–92.

[3] Havelund, K. and Pressburger, T., *Model Checking Java Programs Using Java PathFinder*. International Journal on Software Tools for Technology Transfer (STTT), Vol. 2(4), April 2000.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[5] J. C. King, *Symbolic execution and programtesting*. Communications of the ACM, 19(7): 385–394, 1976.

[6] Giannakopoulou, D., Flavio Lerda, *From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata*.

[7] Sarfraz Khurshid, Corina S. Pasareanu, Willem Visser, *Generalized Symbolic Execution for Model Checking and Testing*.

[8] Pietro Braione, Giovanni Denaro, *Verifying LTL Properties of Bytecode with Symbolic Execution*.

[9] Daniele, M., Giunchiglia, F., and Vardi, M.Y. *Improved Automata Generation for Linear Temporal Logic*, in Proc. of the 11th International Conference on Computer Aided Verification (CAV 1999).

[10] Giannakopoulou, D., Havelund, K., *Automata-based verification of temporal properties on running programs*. In ASE 2001, pp. 412–416 (2001)

[11] Willem Visser, Corina S. Paseranue, *Test input generation for java containers using state matching*, In Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA'06), 2006, pp. 37-48

[12] Clark Barrett and Cesare Tinelli, *CVC3*, In Proceedings of the 19th international conference on Computer aided verification (CAV'07), 2003, pp. 298-302.

[13] Choco Team, *Choco: an Open Source Java Constraint Programming Library*, Research report, 2010.

[14] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, *Memory efficient algorithms for the veri_cation of temporal properties*, Computer-Aided Verification, volume 531 of Lecture Notes in Computer Science, 1991, pp 233.