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

TÓM TẮT LUẬN VĂN THẠC SĨ

HÀ NỘI - 2014

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.

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.

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

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.

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*. 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

2.1. Hệ thống chuyển trạng thái (Transition system)

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.

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 $(s; s') \in t$. 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 $(s; s') \in t$. 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. 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 *next* (tiếp theo)
- Toán tử *eventually* (cuối cùng cũng xảy ra)

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).

- 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).

- 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.

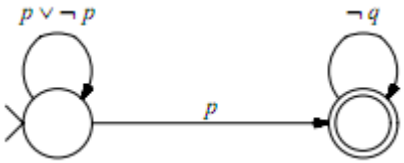
2.3. Buchi automat

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.

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



Hình 2.1 Buchi tương đương với công thức $\neg((p \vee \Diamond q)$

2.4. Thực thi tượng trưng

2.4.1. Thực thi tượng trưng

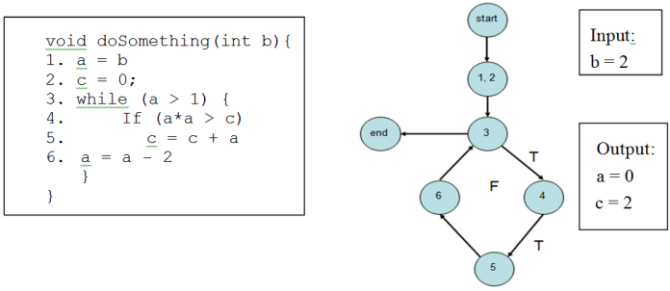
Thực thi tượng trưng [5] được James C. King đề xuất năm 1976, với tư tưởng chủ đạo là thực thi chương trình sử dụng các dữ liệu đầu vào là các giá trị tượng trưng (symbolic values) thay vì các giá trị cụ thể (concrete values).

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 $0_i = P(i)$.



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.

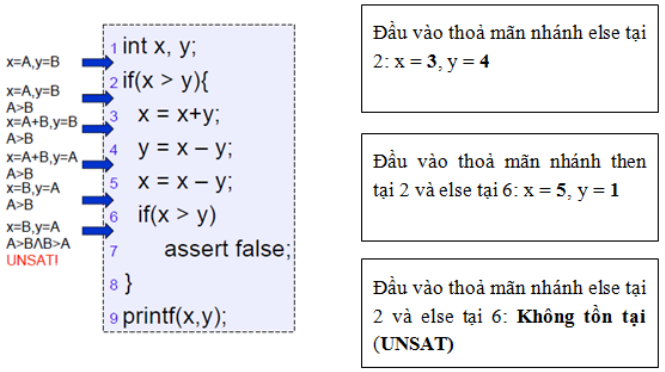
Mỗi nút (khối cơ bản) là một dãy liên tục các chỉ thị sao cho luồng điều khiển không có sự rẽ nhánh hoặc ngừng lại từ khi đi vào nút đó cho tới khi đi ra khỏi nút đó. Ví dụ luồng điều khiển ở *Hình 2.2*

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 $(p_i, p_{i+1}) \in E$ $(1 \leq i \leq n-$

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).
- $\circ$ là một toán tử thuộc tập hợp $\{\leq, \neq, =, <, >, \geq\}$.



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

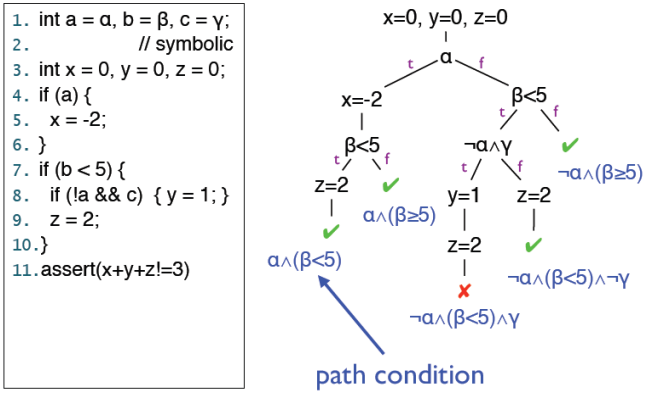
Ý tưởng chính của thực thi tượng trưng là thực thi chương trình với các giá trị tượng trưng thay vì các giá trị cụ thể của các tham số đầu vào. Với mỗi tham số đầu vào, một giá trị tượng trưng được đưa ra để kết hợp với nó. Mỗi biến trong chương trình *P* mà giá trị của nó phụ thuộc vào giá trị của các 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.

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.



Hình 2.4 Biểu thức đường đi (PC)

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).

Chương 3. Java PathFinder (JPF)

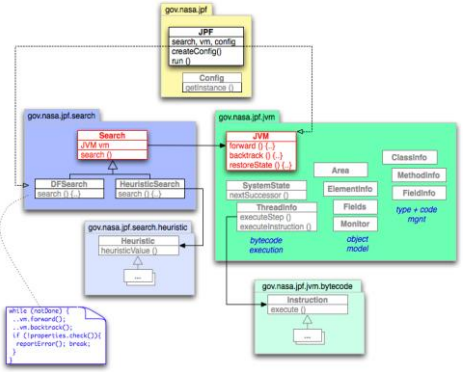
3.1. Java PathFinder

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.

3.1.1. Cấu trúc chính của JPF

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.



Hình 3.1 Thiết kế chính của JPF [1]

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.

Ngoài ra, JPF áp dụng rất nhiều kỹ thuật khác nhau trong cài đặt của mình đó là Choice Generator, Property và Listener.

3.1.2. Choice Generator

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

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

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.

3.1.3. Property

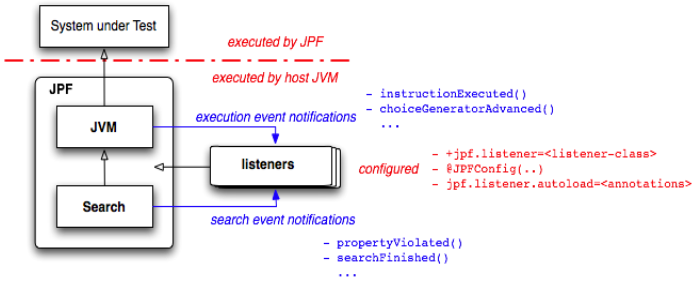
Property là các tính chất hay ràng buộc mà chương trình phải tuần theo. Có ba cách để kiểm tra các tính chất này với JPF.

Cách thứ nhất là sử dụng lớp *NoUncaughtExceptionProperty* đối với các tính chất đơn giản và chỉ phụ thuộc vào giá trị của các dữ liệu trong phạm vi chương trình.

Cách thứ hai là sử dụng giao diện *gov.nasa.jpf.Property*. Interface này có chứa phương thức *Property.check()* được kiểm tra sau khi kết thúc mỗi transition.

Cách thứ ba là sử dụng các listener

3.1.4. Listener



Hình 3.3 JPF Listeners [2]

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

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.

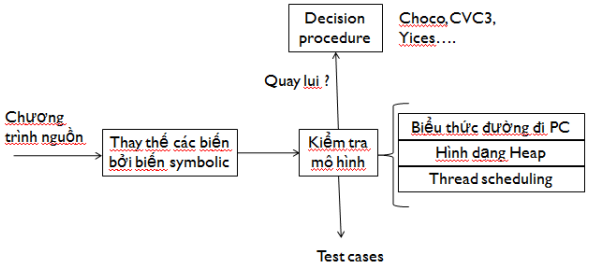
Để tạo ra các listener này, JPF đưa ra hai interface tương ứng với hai đối tượng nguồn của các sự kiện là SearchListener và VMListener. SearchListener thường được dùng để báo cáo về các sự kiện diễn ra trong suốt quá trình tìm kiếm trên không gian trạng thái còn VMListener thì thường báo cáo các sự kiện là các hoạt động của máy ảo.

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

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.



Hình 3.5 Kiến trúc tổng quát của Symbolic PathFinder

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.

Chương 4. Cài đặt

4.1. Kiểm chứng công thức LTL

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 $A(G)$.

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. *jpf-ltl* đọc trực tiếp các công thức LTL từ các chú thích (annotation) trong mã 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.

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

Công cụ ANTLR[3] 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.

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

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

- Tên của một phương thức (method signature)
- Biến logic (boolean variable)
- Mệnh đề quan hệ (relation)

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.

```
f :       binf;

binf :    orf (IMPLIES binf)?;

orf :     andf (OR orf)?;

andf :    untilf (AND andf)?;

untilf :  releasef ((UNTIL | WEAK_UNTIL) untilf)?;

releasef :propf ((RELEASE | WEAK_RELEASE) releasef)?;

propf :   TRUE | FALSE
          | atom
          | unf propf
          | '(' f ')'
          ;

unf :     | NOT | ALWAYS | EVENTUALLY
    ;
/** lexer rules **/
AND           : '/\\';
ROBBYJO_AND   : '&&';
OR            : '\\/';
ROBBYJO_OR    : '||';
UNTIL         : 'U';
WEAK_UNTIL    : 'W';
RELEASE       : 'V';
WEAK_RELEASE  : 'M';
NOT           : '!';
ALWAYS        : '[]';
EVENTUALLY    : '<>';
IMPLIES       : '->';
```

Hình 4.1 Cú pháp LTL

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

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.

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.

- 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. 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

Trình bày 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())*.

- Thuộc tính liveness

Kiểm chứng chương trình có thỏa mãn công thức LTL *[](<>done() && <>foo())* hay không.

- Thuộc tính fairness

Kiểm chứng chương trình có thỏa mãn công thức LTL *[]((y!=1) -> <>foo())* hay không.

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. [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.