Kiểm chứng sự tương tác giữa các thành phần trong chương trình đa luồng sử dụng lập trình hướng khía cạnh

Checking Interaction Protocol in Multi-threaded Program using AOP

Trịnh Thanh Bình, Trương Anh Hoàng, Nguyễn Việt Hà

Abstract: *Interaction protocol specifies allowed method call sequences among classes or objects in a program. We propose an approach to verify interaction protocol for multi-thread programs. Our approach processes interaction protocol specified by extended regular expressions or protocol state machines in UML 2.0 and generates aspect code to weave with the programs for runtime verification. The aspect code will monitor the execution of the program and check the conformance between the programs and their specifications. We implemented the approach as a tool for generating aspect code in AspectJ and checking Java programs. The experimental results show that our approach is convenient to use in practice.*

I. GIỚI THIỆU

Phần mềm ngày càng đóng vai trò quan trọng trong xã hội hiện đại. Tỷ trọng giá trị phần mềm trong các hệ thống ngày càng lớn. Tuy nhiên, trong nhiều hệ thống, lỗi của phần mềm gây ra các hậu quả đặc biệt nghiêm trọng, không chỉ về mặt kinh tế mà còn về con người [16], đặc biệt là các phần mềm điều khiển hệ thống và thiết bị giao thông.

Các phương pháp kiểm chứng hình thức như chứng minh định lý [8] và kiểm chứng mô hình [6, 7] đã đạt được thành công nhất định trong kiểm chứng đặc tả phần mềm. Cài đặt mã chương trình thường chỉ được thực hiện sau khi các đặc tả này đã được kiểm chứng. Tuy nhiên, cài đặt (chương trình) thường không tự sinh ra từ đặc tả nên nó có thể vẫn có lỗi mặc dù thiết kế của nó đã được kiểm chứng là đúng [16].

Để giải quyết các vấn đề này, chúng tôi đã đề xuất một phương pháp kiểm chứng sự tuân thủ của cài đặt so với đặc tả vào thời điểm thực thi [1,10]. Phương pháp này có thể kiểm chứng được sự nhất quán giữa chương trình Java và đặc tả giao thức tương tác của nó, các vi phạm được phát hiện trong bước kiểm thử.

Bài báo này, chúng tôi mở rộng các nghiên cứu trong [1,10] để kiểm chứng sự tuân thủ giữa cài đặt và đặc tả giao thức tương tác trong các chương trình đa luồng sử dụng lập trình hướng khía cạnh (*Aspect-Oriented Programming - AOP*) [5]. Trong [10] chúng tôi đã sử dụng máy trạng thái giao thức (*Protocol State Machine – PSM*) của UML 2.0 để đặc tả giao thức tương tác. Việc sử dụng biểu đồ PSM để đặc tả giao thức tương tác có ưu điểm là trực quan. Tuy nhiên, các biểu đồ này còn nhiều hạn chế như khả năng biểu diễn, và sự không tương thích giữa các công cụ của UML khi xuất các biểu đồ này sang định dạng XMI. Do đó, chúng tôi đã mở rộng biểu thức chính quy (*Regular Expression - RE*) để đặc tả giao thức tương tác. Mã aspect được tự động sinh ra từ các đặc tả này sẽ đan

với chương trình để kiểm chứng sự tuân thủ của nó so với đặc tả giao thức tương tác.

Các phần còn lại của bài báo được cấu trúc như sau. Mục II giới thiệu một số kiến thức cơ bản về AOP. Mục III thảo luận một số nghiên cứu liên quan. Mục IV trình bày các phương pháp đặc tả giao thức tương tác bằng máy trạng thái giao thức, biểu thức chính quy mở rộng và phương pháp kiểm chứng sự tuân thủ giữa chương trình và đặc tả. Mục V chỉ ra một số kết quả thực nghiệm. Các kết luận và hướng phát triển tiếp theo được trình bày trong Mục VI.

II. LẬP TRÌNH HƯỚNG KHÍA CẠNH

Phương pháp lập trình hướng khía cạnh (*Aspect-Oriented Programming - AOP*) [5,11] là phương pháp lập trình phát triển trên tư duy tách biệt các mối quan tâm khác nhau thành các môđun khác nhau. Ở đây, một mối quan tâm thường không phải là một chức năng nghiệp vụ cụ thể và có thể được đóng gói mà là một khía cạnh (*thuộc tính*) chung mà nhiều môđun phần mềm trong cùng hệ thống nên có, ví dụ như lưu vết thao tác và lỗi (*error logging*).

Với AOP, chúng ta có thể cài đặt các mối quan tâm chung cắt ngang hệ thống bằng các môđun đặc biệt gọi là aspect thay vì dàn trải chúng trên các môđun nghiệp vụ liên quan. Các aspect sau đó được kết hợp tự động với các môđun nghiệp vụ khác bằng quá trình gọi là đan (*weaving*) bằng bộ biên dịch đặc biệt.

AspectJ [3] là một công cụ AOP cho ngôn ngữ lập trình Java. Trình biên dịch AspectJ sẽ đan xen chương trình Java chính với các aspect thành các tệp mã bytecode chạy trên chính máy ảo Java.

III. MỘT SỐ NGHIÊN CỨU LIÊN QUAN

Đã có một vài phương pháp được đề xuất để kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác được đề xuất.

Jin[15] đề xuất một phương pháp hình thức để kiểm chứng tĩnh sự tuân thủ giữa cài đặt mã nguồn và đặc tả thứ tự thực hiện của các phương thức (*Method Call Sequence - MCS*) trong các chương trình Java tuần tự. Phương pháp này sử dụng automat hữu hạn trang thái để đặc tả MCS, các chương trình Java được biến đổi thành các văn phạm phi ngữ cảnh (*Context Free Grammar- CFG*) sử dụng công cụ Accent[1]. Ngôn ngữ sinh ra bởi ôtômát $L(A)$ được so sánh với ngôn ngữ sinh ra bởi CFG $L(G)$, nếu $L(G) \subseteq L(A)$ thì chương trình Java tuân thủ theo đặc tả MCS. Ưu điểm của phương pháp này là các vi phạm có thể được phát hiện sớm, tại thời điểm phát triển hoặc biên dịch chương trình mà không cần chạy thử chương trình. Tuy nhiên, phương pháp này chưa kiểm chứng được các chương trình đa luồng. Hơn nữa, phương pháp này cũng phải giải quyết trọn vẹn bài toán bao phủ ngôn ngữ (*Language Inclusion Problem*).

Trong các phương pháp về JML[9,13,14], MCS phải được đặc tả dưới dạng tiền và hậu điều kiện được kết hợp với phần thân của các phương thức trong chương trình như các bất biến của vòng lặp, hay tập các câu lệnh. Các tiền và hậu điều kiện này được viết dưới một dạng chuẩn để có thể biên dịch và chạy đan cùng với chương trình nguồn. Các vi phạm sẽ được phát hiện vào thời điểm chạy chương trình. Với các phương pháp này thì người lập trình phải đặc tả rải rác mã kiểm tra ở nhiều điểm trong chương trình. Do đó sẽ khó kiểm soát, không đặc tả độc lập, tách biệt từng đặc tả MCS được.

Yoonsik và Perumandla [14] mở rộng ngôn ngữ đặc tả và trình biên dịch JML để biểu diễn MCS bằng biểu thức chính quy. Các biểu thức chính quy này được biên dịch thành mã thực thi và đan xen với mã nguồn của chương trình gốc để kiểm chứng sự tuân thủ giữa cài đặt so với đặc tả MSC. Các hành vi của chương trình gốc sẽ không bị thay đổi ngoại trừ thời gian thực thi và kích thước.

Deline và Fahndrich [12] đề xuất phương pháp kiểm chứng vào thời điểm thực thi sự tuân thủ giữa cài đặt và

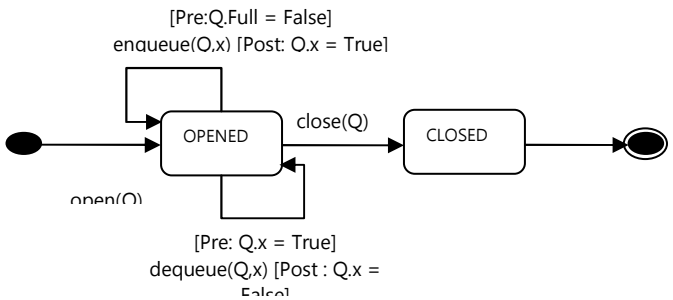
[1] http://accent.compilertools.net/Accent.html

đặc tả MCS. Phương pháp này sử dụng máy trạng thái để đặc tả MCS. Đặc tả MCS sau đó được biên dịch sang mã nguồn và đan xen với mã nguồn chương trình để kiểm chứng động sự tuân thủ của cài đặt so với đặc tả MCS. Các mệnh đề tiền và hậu điều kiện của các phương thức trong MSC cũng được đặc tả và kiểm chứng.

Các phương pháp nói trên đều chưa kiểm chứng được các chương trình đa luồng, giao thức được kiểm chứng đơn thuần chỉ là thứ tự thực hiện của các phương thức. Trong bài báo này chúng tôi đề xuất một cách tiếp cận mới trong việc kiểm chứng sự nhất quán giữa cài đặt so với thiết kế ở thời điểm thực thi. Trong đó các phương thức trong giao thức có thể được thực hiện song song với nhau và phải thỏa mãn các mệnh đề tiền và hậu điều kiện.

IV. PHƯƠNG PHÁP KIỂM CHỨNG SỰ TUÂN THỦ GIỮA THỰC THI VÀ ĐẶC TẢ GIAO THỨC TƯƠNG TÁC

Giả sử một giao thức tương tác của một hàng đợi tương tranh (*Concurrent Queue - CQ*) với bốn phương thức được cài đặt cho phép gọi cùng lúc bởi một luồng cung cấp Producer đẩy các phần tử vào hàng đợi, và nhiều luồng Consumer cùng thao tác với các phần tử trong hàng đợi (Hình 1). Tại trạng thái trừu tượng OPENED, các luồng Consumer có thể gọi các phương thức enqueue() hoặc dequeue() để bổ sung hoặc loại bỏ các phần tử của hàng đợi. Khi luồng Producer gọi phương thức close() để chuyển sang trạng thái trừu tượng CLOSED thì các phần tử khác sẽ không được bổ sung hoặc loại bỏ từ hàng đợi.



Hình 1. Giao thức tương tác của hàng đợi tương tranh.

Khi đó bài toán kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác trong các chương trình đa luồng được đặc tả như sau:

1. Thứ tự thực hiện của các phương thức trong chương trình phải tuân thủ theo các cung trong Hình 1 là một đường đi từ trạng thái đầu đến trạng thái kết thúc. Trong đó, hai phương thức dequeue(Q,x) và enqueue(Q,x) có thể được gọi đồng thời bởi các luồng khác nhau.
2. Khi phương thức enqueue(Q,x) được thực hiện thì tiền điều kiện là hàng đợi chưa đầy và hậu điều kiện là x phải được đẩy vào hàng đợi. Với phương thức dequeue(Q,x) thì tiền điều kiện là x thuộc hàng đợi và hậu điều kiện là x được loại bỏ khỏi hàng đợi.

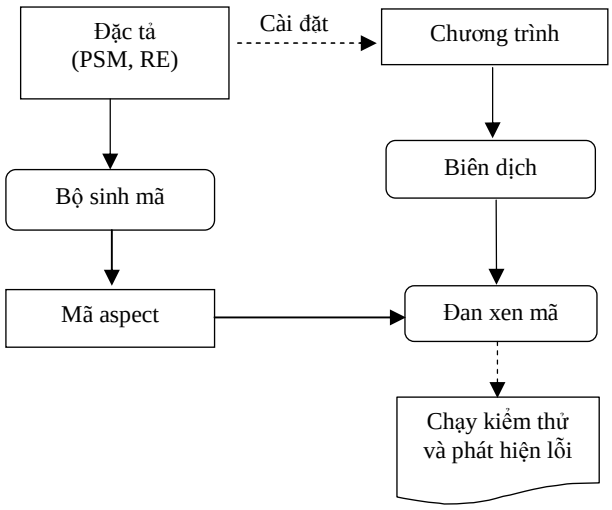
Giả sử đặc tả thiết kế giao thức này là đúng đắn. Tuy nhiên, cài đặt mã nguồn chương trình có thể vi phạm các đặc tả thiết kế của giao thức. Thông thường các vi phạm này khó được phát hiện trong bước kiểm thử bằng các bộ dữ liệu đầu vào và đầu ra.

Do đó chúng tôi đã đề xuất phương pháp kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác trong các chương trình đa luồng như sau (Hình 2).

1. Sử dụng biểu thức chính quy mở rộng (RE) hoặc máy trạng thái giao thức (PSM) để đặc tả giao thức tương tác (IP),
2. Người lập trình cài đặt các ứng dụng dựa trên các đặc tả IP,
3. Tự động sinh các mã aspect từ các đặc tả IP,
4. Các mã aspect sinh ra được tự động đan với mã của các chương trình ứng dụng để kiểm chứng động sự tuân thủ giữa thực thi và đặc tả IP.

Kết quả thực nghiệm trong Mục V cho thấy khi các chương trình được thực hiện thì các mã đan xen vào có thể phát hiện được chính xác vị trí của các vi phạm nếu có của chương trình với đặc tả IP. Trong khi đó, các hành vi của

chương trình gốc sẽ không bị thay đổi ngoại trừ thời gian thực hiện và kích thước của chương trình.



Hình 2. Sơ đồ hoạt động của hệ thống.

1. Đặc tả giao thức tương tác

1.1. Biểu thức chính quy mở rộng cho biểu diễn giao thức tương tác

RE được mở rộng để biểu diễn IP độc lập với mã nguồn để sinh ra mã aspect được chúng tôi định nghĩa như sau.

Định nghĩa 1 (Biểu thức chính quy mở rộng). Regular Expression - RE là một bộ năm RE = <M, O, S, Pre, Post>. Trong đó,

1. M = $\{m_1,m_2,...,m_n\}$ là bảng chữ cái Sigma gồm một tập hữu hạn các phương thức,
2. O = $\{o_1,o_2,...,o_p\}$ là tập hữu hạn các đối tượng,
3. Pre, Post là tập hữu hạn các tiền và hậu điều kiện,
4. S = $\{s_1,s_2,...,s_k\}$ là tập hạn các biểu thức biểu diễn các phương thức,
5. s ::= [Pre]o.m[Post]|s->s|s|s| s||s |s^*|s^+|(s). Trong đó: m ∈ M, s ∈ S, và o ∈ O. s→s là sự kết hợp của hai hoặc nhiều biểu thức tuần tự, s|s: phép hoặc, s||s: phép song song, s^*: không hoặc nhiều phép lặp, s^+: một hoặc nhiều phép lặp, (s): biểu thức kết hợp.

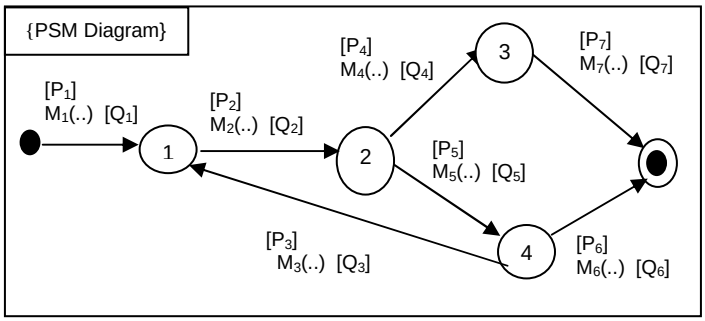
Ví dụ RE:

$[p_1]o_1.m_1()[q_1]\rightarrow([p_2]o_2.m_2()[q_2]||[p_3]o_3.m_3()[q_3])^+\rightarrow(o_1.m_1()||o_4.m_4())\rightarrow[p_5]o_5.m_5()[q_5]$ biểu diễn một IP. Trong đó, nếu tiền điều kiện p_1 được thỏa mãn thì phương thức $o_1.m_1()$ được thực hiện trước và thỏa mãn hậu điều kiện q_1, sau đó là một hoặc nhiều lần thực hiện phương thức $o_2.m_2()$ hoặc $o_3.m_3()$ với các tiền điều kiện p_2, p_3 và hậu điều kiện q_2, q_3. Tiếp theo là các phương thức $o_1.m_1()$ và $o_4.m_4()$ được thực hiện song song. Cuối cùng là $o_5.m_5()$ với các điều kiện là p_5 và q_5.

1.2. Biểu đồ PSM cho biểu diễn giao thức tương tác

Biểu đồ PSM trong UML2.0 biểu diễn thứ tự thực hiện của các phương thức cùng với ràng buộc về các mệnh đề tiền và hậu điều kiện được sử dụng để đặc tả IP. Chúng tôi định nghĩa hình thức như sau:

Định nghĩa 2 (Máy trạng thái giao thức). Protocol State Machine - PSM là một bộ bẩy thành phần PSM = <S; σ; M; Pre; Post; s_0;f>. Trong đó, S là tập hữu hạn các trạng thái, M là tập các phương thức, Pre, Post là tập các tiền điệu kiện và hậu điều kiện. $\sigma \subseteq S\times Pre\times M\times Post\rightarrow S$ là hàm chuyển trạng thái. $s_0,f\in S$ lần lượt là các trạng thái đầu và kết thúc.



Hình 3. Biểu đồ PSM cho một giao thức tương tác.

Hình 3 biểu diễn biểu đồ PSM cho một IP, thứ tự thực hiện của các phương thức được thể hiện bằng các cung trong biểu đồ. Trong đó:

- $S=\{1,2,3,4\}\cup\{s_0,f\}$,
- $Pre=\{P_1..P_7\};Post=\{Q_1..Q_7\};M=\{M_1..M_7\}$,
- $\sigma=\{s_0P_1M_1Q_1\rightarrow1;1P_2M_2Q_2\rightarrow2;...;3P_7M_7Q_7\rightarrow f;\ 4P_6M_6Q_6\rightarrow f\}$.

2. Sinh mã aspect

Mục này trình bày thuật toán tự động sinh mã kiểm chứng aspect từ đặc tả IP. Với đặc tả dạng PSM chúng tôi sinh ra đồ thị có hướng để biểu diễn IP bằng thuật toán trong Bảng 1. Với đặc tả RE mở rộng được đưa về dạng RE chuẩn bằng phép biến đổi mỗi s=[Pre]o.m[Post] thành một ký tự $a \in \Sigma$ (*một ký tự thuộc bảng chữ cái của biểu thức RE chuẩn*). Từ dạng RE chuẩn chúng tôi chuyển sang máy trạng thái hữu hạn (*Finite State Machine-FSM*) bằng thuật toán trong [2]. Mã aspect sau đó được sinh ra tự động từ các đặc tả PSM và FSM.

Bảng 1. Sinh đồ thị biểu diễn IP từ đặc tả PSM

Đầu vào: đặc tả PSM

Đầu ra: Đồ thị G = <V, M> đặc tả giao thức. Trong đó:

- V là tập các đỉnh của đồ thị (được biểu diễn bằng tập các số nguyên),
- M là tập các cung của đồ thị, $M = \{[Pre_1]m_1[Post_1],[Pre_2]m_2[Post_2],..., [Pre_n]m_n[Post_n]\}$ là tập các phương thức với các tiền và hậu điều kiện thuộc giao thức,
- Các cung của đồ thị được gán nhãn là các phương thức thuộc M, các đỉnh được gán nhãn là các số nguyên. Các cung này thể hiện mối quan hệ phụ thuộc giữa các phương thức trong IP.

1. Tạo hàm song ánh μ: $M \rightarrow \{1..|M|\}$,|M| lực lượng của tập M, các số nguyên này là tập các đỉnh của đồ thị.
2. Tạo một đỉnh vào và gán nhãn bằng 0, với mỗi m thuộc M_0 (tập các đỉnh vào của máy trạng thái ($o \rightarrow M_0$) tạo một cung từ đỉnh vào đến đỉnh $\mu(m)$, gán nhãn là $[pre_m]m[post_m]$.
3. Với mỗi cung dạng $m \rightarrow m'$ thuộc PSM tạo một nút $\mu(m)$ tới $\mu(m')$ và gán nhãn là $\{pre_{m'}\}m'\{post_{m'}\}$.
4. Tạo một đỉnh kết thúc, với mỗi $m \rightarrow \Theta$ thuộc đỉnh kết thúc trong PSM, tạo một cung từ $\mu(m)$ tới đỉnh kết thúc vừa tạo.

Quá trình tự động sinh mã aspect gồm ba bước chính sau.

Bước 1. Khởi tạo mẫu aspect sẽ được sinh ra từ đặc tả gao thức tương tác như sau.

```
static final String aspectTemplate =
"import org.aspectj.lang.JoinPoint;\n" +
"public aspect ProtocolCheck {\n" + "#CONSTS#\n" +
"#ADVICES#\n\n" + " void log(JoinPoint jp); \n";
```

Trong aspect mẫu trên xâu "#CONST#" sẽ được thay thế bằng các trạng thái của mỗi phương thức trong giao thức. Xâu "# ADVICES#" sẽ được thay thế bằng các điều kiện kiểm tra trước và sau (*pointcut*) của phương thức khi nó được thực hiện. Phương thức log(JoinPoint jp) sẽ thông báo các phương thức và vị trí của nó khi vi phạm đặc tả.

Bước 2. Khởi tạo mẫu pointcut sẽ được sinh ra từ đặc tả gao thức tương tác như sau.

```
static String pointcutTemplate =
"\n" +" pointcut pc_#SIG_NM#(#CLS_NM# o):\n"+" target(o)\n"+
" &&call(#SIG#);\n"+" before(#CLS_NM#
o):pc_#SIG_NM#(o){\n"+
" if (!(#PRE_COND#))\n" +" log(thisJoinPoint);\n"+" }\n"+
" after(#CLS_NM# o):pc_#SIG_NM#(o) {\n"+" o.state =
ST_#SIG_NM#;\n"+"#POST_COND# "+" }\n";
```

Trong pointcut mẫu trên xâu "#SIG_NM#" sẽ được thay thế bằng tên của mỗi phương thức trong giao thức, "# CLS_NM#" sẽ được thay thế bằng tên của lớp tương ứng. Xâu "#PRE_COND#" và "#POST_COND# sẽ được thay thế bằng các biểu thức tiền và hậu điều kiện.

Bước 3. Các biểu thức tiền và hậu điều kiện được chia làm hai loại. Loại một kiểm tra thứ tự thực hiện của các phương thức trong giao thức. Loại hai đặc tả các điều kiện trước và sau của mỗi phương thức phải thỏa mãn khi nó được thực hiện.

Bước 3.1. Với biểu thức tiền và hậu điều kiện loại một thì mỗi phương thức trong giao thức chúng tôi tự động sinh ra một biến trạng thái có tiền tố là ST_, theo sau là tên các phương thức. Mỗi khi phương thức được thực hiện thì biến trạng thái được gán bằng trạng thái của phương thức đó. Hàm sinh biểu thức tiền điều kiện được cài đặt như sau.

```
static String genCondition( Entry<String, Set<String>> e,
Set<String> entrySigs) {
  String src = "";
```

```
  if (entrySigs.contains(e.getKey()))
    src += "o.state==ST_START";
  for (String s: e.getValue()){
    if (s.equals("START"))  continue;
    if (src.length() > 0) src += "";
    src += "o.state==ST_" +getMethodName(s);
   }
  return src;
}
```

Bước 3.2. Với các biểu thức tiền và hậu điều kiện loại hai, chúng tôi giới hạn được đặc tả dưới dạng các biểu thức logic của Java (*bước 2 và 3, thuật toán trong Bảng 2*). Các biểu thức này được đọc trực tiếp từ đặc tả và đưa vào pointcut mẫu trong bước 2.

3. Đan mã aspect

AspectJ cho phép đan xen mã aspect với các chương trình Java ở ba mức khác nhau: mức mã nguồn, mã bytecode và tại thời điểm nạp chương trình khi chương trình gốc chuẩn bị được thực hiện.

Đan ở mức mã nguồn, AspectJ sẽ nạp các mã aspect và Java ở mức mã nguồn *(.aj và .java)*, sau đó thực hiện biên dịch để sinh ra mã đã được đan xen bytecode, dạng *.class*. Đan xen ở mức mã bytecode, AspectJ sẽ dịch lại và sinh mã dạng *.class* từ các các mã aspect và *Java* đã được biên dịch ở dạng (*.class*). Đan xen tại thời điểm nạp chương trình (*load time weaving*), các mã của aspect và Java dạng *.class* được cung cấp cho máy ảo Java (*JVM*). Khi JVM nạp chương trình để chạy, bộ nạp lớp của *AspectJ* sẽ thực hiện đan mã và chạy chương trình.

Với việc đan xen ở mức mã bytecode và tại thời điểm nạp chương trình thì phương pháp này có thể được sử dụng mà không yêu cầu phải có mã nguồn. Khi thay đổi đặc tả thì mới phải sinh và biên dịch lại mã aspect.

V. THỰC NGHIỆM

Chúng tôi đã cài đặt phương pháp này thành một công cụ kiểm chứng PVG (*Protocol Verification Generator - PVG*). Đầu vào của công cụ PVG là các FSM hoặc đồ thị có hướng biểu diễn giao thức tương tác. Đầu ra là các mã kiểm chứng aspect của AspectJ.

Thực nghiệm được tiến hành trên lớp StreamBuffer với ba phương thức open(), read() và close(). Giao thức tương tác được đặc tả bằng biểu thức chính quy open()->(read())*->close() mô tả phương thức open() được thực hiện trước sau đó là một hoặc nhiều lần gọi phương thức read(), cuối cùng là phương thức close() được gọi để giải phóng tài nguyên.

Bảng 2 minh họa một chương trình được cài đặt tuân thủ theo đúng giao thức bên trái và sai bên phải (*do phương thức close(..) không được gọi*). Các chương trình đa luồng này được xây dựng để tính tổng các số nguyên trong file. Thực hiện kiểm thử các chương trình trên với các input/output khác nhau, các kết quả cho thấy cả hai chương trình đều cho kết quả đúng như nhau. Tuy nhiên khi đan mã của các chương trình trên với mã aspect được sinh ra từ công cụ PVG chúng tôi đã phát hiện được vi phạm ràng buộc của chương trình được cài đặt sai bên phải.

Bảng 2– Chương trình được cài đặt đúng – sai

```
public class Mytest extends
Thread
{
  private int num;
  public void run()  {
   try {
     InputStream f = new
FileInputStream("file.txt");
//open()
      int c, s=0;
      while ((c=f.read())!=-1){

System.out.print((char)c)
        s=s+c;
       }
       f.close();
       System.out.print(s);
      }catch(Exception e) {
        e.printStackTrace(); }
}
public     Mytest     (int     n)
           {
   super();
   num=n;
}
public static void main(String[]
args) {
    Mytest t1=new Mytest (1);
    t1.start();
}}
```

```
public class Mytest extends Thread
{
  private int num;
  public void run()  {
   try {
     InputStream f = new
FileInputStream("file.txt"); //open()
      int c, s=0;
      while ((c=f.read())!=-1){
        System.out.print((char)c)
         s=s+c;
       }
   // phương thức f.close() không
dược gọi
        System.out.print(s);
      }catch(Exception e) {
        e.printStackTrace(); }
}
public Mytest (int n)            {
   super();
   num=n;
}
public static void main(String[] args)
{
    Mytest t1=new Mytest (1);
    t1.start();
}
}
```

Bên cạnh giao thức này, chúng tôi cũng đã thử nghiệm với các giao thức khác trong [1,4,12,13,15]. Các giao thức này được đặc tả bằng các RE và PSM. Với mỗi đặc tả này chúng tôi sử dụng công cụ PVG để sinh các mã aspect của AspectJ và đan tự động với các chương trình Java mô phỏng để kiểm chứng sự tuân thủ giữa sự cài đặt đối với đặc tả giao thức. Các kết quả thực nghiệm trong Bảng 3.

Trong đó, mỗi lớp trong cột 1 bên trái của Bảng 3 tương ứng với số các phương thức của giao thức trong cột 2. Chúng tôi xây dựng chương trình mô phỏng cho từng lớp với số các ca kiểm thử đúng và sai khác nhau trong cột 3, kết qủa phát hiện trong cột 4. Với các ca kiểm thử đúng thì các lớp được cài đặt tuân thủ đúng đặc tả giao thức. Ngược lại, với các ca kiểm thử sai thì sẽ có ít nhất một phương thức thực hiện không đúng đặc tả (*các vi phạm về thứ tự thực hiện, tiền và hậu điều kiện*). Các giao thức đều được đặc tả dưới cả hai dạng RE và PSM.

Các chương trình mô phỏng trước và sau khi đan mã AspectJ được chạy 20 lần với mỗi lần chạy thì số luồng được tăng dần từ 1 đến 20 luồng. Để đánh giá thời gian thực hiện của các chương trình trước khi đan mã aspect so với thời gian thực hiện sau khi đan mã chúng tôi tính tỷ lệ gia tăng thời gian trung bình bằng công thức sau:

$$\tau = \frac{\sum_{i=1}^{n} |ts_i - tt_i|}{n} \times 100\%.$$

Trong đó, ts_i và tt_i lần lượt là thời gian thực hiện của chương trình trước và sau khi đan mã aspect ở lượt chạy thứ i, n là tổng số lượt chạy của chương trình trước và sau khi đan mã. Thời gian thực hiện của các chương trình trước và sau khi đan mã được tính bằng hiệu của thời gian hiện tại của hệ thống trước khi chương trình được thực hiện với thời gian hiện tại của hệ thống sau khi chương trình thực hiện xong. Kết quả thực nghiệm trong Bảng 3.

Đối với các giao thức mô tả trong cột 1, Bảng 3 thì kết quả thực nghiệm cho thấy: (i) các aspect được sinh ra đúng so với các đặc tả giao thức, nhất quán giữa biểu thức chính quy và máy trạng thái giao thức, (ii) các aspect không làm thay đổi hành vi của chương trình gốc ngoại trừ thời gian chạy và kích thước của chương trình, (iii) đã phát hiện được các vi phạm tương tác (*thứ tự thực hiện*), tiền và hậu điều kiện của các phương thức được cài đặt mà không tuân thủ theo đặc tả IP, (iv) thời gian chạy sau khi đan mã aspect sẽ tăng tỷ lệ thuận với số luồng trong chương trình và số phương thức được mô tả trong giao thức.

Bảng 3- Kết quả thực nghiệm

Lớp (.java)	Số phương thức	Số test đúng/sai	Phát hiện đúng/sai	Tỷ lệ gia tăng thời gian (%s)
Applet	5	10/20	10/20	0.915
StreamReader	6	5/15	5/15	0.923
ReadWrite	4	6/10	6/10	0.974
Iterator	3	2/3	2/3	0.533
Stack	5	2/5	2/5	0.915
LinkedList	9	5/15	5/15	1.542
ConcurrentQueue	4	3/5	3/5	0.974
Roster	2	2/2	2/2	0.323

VI. KẾT LUẬN

Giao thức tương tác đặc tả các ràng buộc về thứ tự thực hiện của các phương thức trong các lớp hoặc các thành phần phần mềm, các biểu thức tiền và hậu điều kiện của mỗi phương thức khi nó được thực hiện. Sự vi phạm giữa cài đặt và đặc tả giao thức này tại thời điểm thực thi có thể gây ra các lỗi hệ thống. Tuy nhiên, thiết kế giao diện của thành phần phần mềm chỉ đặc tả các ràng buộc về kiểu dữ liệu và giá trị trả về của mỗi phương thức. Hơn nữa, các trình biên dịch cũng không kiểm tra các ràng buộc của giao thức này.

Trong bài báo này, chúng tôi đã đề xuất một phương pháp kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác sử dụng lập trình hướng khía cạnh. Phương pháp này sử dụng máy trạng thái giao thức của UML và biểu thức chính quy để đặc tả giao thức tương tác. Các mã aspect được tự động sinh ra từ các đặc tả này sẽ đan tự động với mã của các ứng dụng để kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác.

Chúng tôi đã cài đặt phương pháp này thành một công cụ kiểm chứng và chạy thử nghiệm với ngôn ngữ lập trình Java thông qua một số giao thức thực tế. Kết quả thực nghiệm ban đầu cho thấy phương pháp được đề xuất có thể phát hiện được các vi phạm ràng buộc thiết kế của giao thức tương tác trong các chương trình đa luồng. Hạn chế của phương pháp này cũng như các phương pháp kiểm chứng động khác là phải thực thi chương trình, vị phạm chỉ được phát hiện trong bước kiểm thử. Hơn nữa, mã aspect được đan vào sẽ làm tăng kích thước và thời gian thực thi của các chương trình.

Trong tương lai, chúng tôi sẽ tiếp tục mở rộng phương pháp này để kiểm chứng các bất biến đối tượng (*object invariants*), các ràng buộc thời gian (*timing constraints*), và các ràng buộc khác trong các chương trình đa luồng. Tiến tới phát triển môi trường kiểm chứng hoàn thiện dựa trên lập trình hướng khía cạnh để kiểm chứng sự tuân thủ giữa thiết kế với cài đặt mã nguồn chương trình.

TÀI LIỆU THAM KHẢO

1. Anh-Hoang Truong, et.al. Checking interface interaction protocols using Aspect-oriented programming. In *SEFM'08: Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, pages 382–386, 2008.
2. Ayesha Hanif, et.al. Regular Expression to Finite State Machine. *Journal of Applied Sciences Research*, pages 1359-1362, 2006.
3. Colyer and A. Clement. Aspect-oriented programming with AspectJ. *IBM Syst*. J., pages 301–308, 2005.
4. Cl´ement Hurlin Specifying and Checking Protocols of Multithreaded Classes. *Proceedings of the ACM symposium on Applied Computing*, Pages 587-592, 2009.
5. Filman R. E., et.al. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.
6. Gerard J.Holzmann. *The SPIN Model Checker Primer and Reference Manual*. Addison-Wesley, 2003.
7. Joost-Pieter Katoen, Concepts, Algorithms, and Tools for Model Checking, *Lecture Notes of the Course Mechanised Validation of Parallel Systems*, 1999.
8. Jones, C.B. Theorem proving and software engineering. *Software Engineering Journal Vol.3, Digital Object Identifier*, 1998.
9. L. Burdy, Y. Cheon. An overview of JML tools and applications. *Software Tools for Technology Transfer*, pages 212–232, 2005.
10. Thanh-Binh Trinh, Anh-Hoang Truong, and Viet-Ha Nguyen. Checking protocol-conformance in component models using Aspect oriented programming. In *Advances in Computer Science and Engineering*, Actapress, pages 150–155, 2009.
11. Reade, Chris. *Elements of Functional Programming*. Addison-Wesley Longman Publishing Co, Boston, USA, 1989.
12. R. DeLine and M. Fahndrich. The fugue protocol checker: Is your software baroque. *Technical Report MSR-TR-2004-07*, Microsoft Research, 2004.
13. Y. Cheon and A. Perumandla. Specifying and checking method call sequences in JML. *Software Engineering Research and Practice*, CSREA Press, pages 511–516, 2005.
14. Y. Cheon and A. Perumandla. *Specifying and checking method call sequences of Java programs*. Software Quality Control, pages 7–25, 2007.
15. Y. Jin. *Formal verification of protocol properties of sequential Java programs*. In COMPSAC'07: Proc of the 31st Annual International Computer Software and Applications Conference, Washington, DC, USA,. IEEE CS, Vol. 1, pages 475–482, 2007.
16. Willem Visser, et.al. *Model Checking Programs*, 15th IEEE International Conference on Automated Software Engineering (ASE'00), 2000.