Tạp chí Tin học và Điều khiển học, T.26, S.2 (2010), 173–184

KIỂM CHỨNG SỰ TUÂN THỦ VỀ RÀNG BUỘC THỜI GIAN TRONG CÁC ỨNG DỤNG PHẦN MỀM

TRỊNH THANH BÌNH, TRƯƠNG NINH THUẬN, NGUYỄN VIỆT HÀ

Trường Đại học Công nghệ - Đại học Quốc Gia Hà Nội

Abstract. Timing constraints play an important role in software development, particularly in real time, embedded systems. This paper proposes an approach for checking the compliance of execution of components in an application with their timing constraints. In this approach, timing constraints are specified by UML Timing Diagrams and Timed Regular Expressions. Aspect Oriented Programming technology is used to observe the execution of the program and to check if the execution of the components satisfies their timing constraint specification. We have implemented our approach and verified this tool with several practical Java components. The initial results promise that our approach can be applied in verification of real time software.

Tóm tắt. Ràng buộc thời gian giữa các thành phần đóng vai trò quan trọng trong các hệ thống phần mềm đặc biệt với các hệ thống thời gian thực, hệ thống nhúng. Bài báo này đề xuất một phương pháp kiểm chứng sự tuân thủ về ràng buộc thời gian thực thi giữa các thành phần phần mềm so với đặc tả sử dụng lập trình hướng khía cạnh. Trong đó, ràng buộc thời gian giữa các thành phần được đặc tả bằng biểu đồ thời gian của UML (Unified Modeling Language) và biểu thức chính quy thời gian. Từ các đặc tả này mã kiểm chứng aspect sẽ được tự động sinh ra và đan với mã của các thành phần để tính thời gian thực thi từ đó kiểm chứng sự tuân thủ so với đặc tả. Phương pháp này đã được thực nghiệm với nhiều thành phần phần mềm khác nhau. Kết quả thực nghiệm cho thấy phương pháp được đề xuất có thể có thể phát hiện được các vi phạm ràng buộc thời gian giữa các thành phần phần mềm so với đặc tả.

Keywords: Model checking, Timed regular expression, Timing diagram, aspect, AspectJ.

1. 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 thiệt hại về mặt kinh tế mà còn làm tổn thất trực tiếp sinh mạng con người [17], đặc biệt với các phần mềm thời gian thực như phần mềm điều khiển hệ thống giao thông và thiết bị giao thông.

Trong công nghiệp phần mềm, phương pháp chủ đạo để đảm bảo chất lượng vẫn là kiểm thử phần mềm bằng các bộ dữ liệu test *(test suite)*. Tuy nhiên, việc kiểm thử ở mức đơn vị *(unit testing)* bằng các bộ dữ liệu test thường chỉ phát hiện được các lỗi về giá trị đầu ra *(output)*, không thể phát hiện lỗi vi phạm các ràng buộc thiết kế như ràng buộc về thời gian, thứ tự

thực hiện giữa các thành phần,... Các vi phạm ràng buộc này sẽ gây lỗi hệ thống trong một ngữ cảnh đặc biệt khi tích hợp nhiều thành phần và chạy với một tập dữ liệu đặc biệt nào đó. Khi đó việc xác định chính xác vị trí gây lỗi sẽ rất khó khăn và làm chi phí sửa lỗi tăng cao.

Các phương pháp kiểm chứng hình thức như chứng minh định lý *(theorem proving)* [13] và kiểm chứng mô hình *(model checking)* [11, 15, 16] đã được ứng dụng thành công để kiểm chứng mô hình đặc tả phần mềm. Cài đặt thực tế thường chỉ được thực hiện sau khi mô hình hệ thống đã được kiểm chứng. Tuy nhiên, cài đặt thực mã nguồn chương trình có thể vi phạm các ràng buộc thiết kế. Do đó, phần mềm có thể vẫn tồn tại lỗi mặc dù thiết kế đã được kiểm chứng và thẩm định chi tiết [17].

Để giải quyết các vấn đề này, chúng tôi đã đề xuất các phương pháp kiểm chứng sự tuân thủ của cài đặt so với thiết kế vào thời điểm thực thi [2, 3, 10] sử dụng lập trình hướng khía cạnh *(AOP - Aspect-Oriented Programming)* [6, 8]. Với AOP, chúng ta có thể cài đặt các mô đun đặc biệt gọi là aspect. Các aspect sẽ được kết hợp tự động với chương trình bằng bộ biên dịch đặc biệt để giám sát sự hoạt động và phát hiện vi phạm giữa chương trình và ràng buộc thiết kế trong bước kiểm thử.

Phương pháp kiểm chứng ràng buộc thời gian *(Timing Constraints - TC)* giữa sự cài đặt các thành phần phần mềm so với đặc tả biểu đồ thời gian *(Timing Diagram - TD)* của UML được đề xuất trong [2] còn nhiều hạn chế trong đặc tả như khả năng biểu diễn của TD, tính khả chuyển giữa các công cụ UML. Hơn nữa, phương pháp này cũng chưa kiểm chứng được ràng buộc thời gian giữa các thành phần tương tranh. Do đó bài báo này chúng tôi mở rộng với TC được đặc tả bằng biểu thức chính quy thời gian *(Timed Regular Expressions - TRE)*, và kiểm chứng ràng buộc thời gian giữa các thành phần tương tranh.

Các phần còn lại của bài báo được cấu trúc như sau. Mục 2 trình bày một số nghiên cứu liên quan. Phương pháp kiểm chứng ràng buộc thời gian giữa các thành phần sử dụng AOP được trình bày trong Mục 3. Mục 4 trình bày một số kết quả thực nghiệm, cuối cùng là các kết luận và hướng phát triển tiếp theo.

2. MỘT SỐ NGHIÊN CỨU LIÊN QUAN

Đã có một vài phương pháp được đề xuất để kiểm chứng ràng buộc thời gian trong các hệ thống phần mềm.

SACRES [1] là một môi trường kiểm chứng cho các hệ thống nhúng, cho phép người sử dụng đặc tả ràng buộc thời gian bằng các biểu đồ thời gian dạng kí hiệu *(symbolic timing diagrams)*. Các đặc tả thiết kế được dịch sang máy hữu hạn trạng thái *(finite state machine)* được tối ưu và kiểm chứng bằng mô hình kí hiệu *(symbolic model checking)*. Tuy nhiên phương pháp này chỉ kiểm chứng ở mức mô hình, không phải ở mức cài đặt.

Wegener [18] đề xuất phương pháp để kiểm chứng ràng buộc thời gian trong các hệ thống thời gian thực dựa trên kĩ thuật kiểm thử tiến hóa *(evolutionary testing)*. Trong đó, vi phạm ràng buộc thời gian được định nghĩa là đầu ra *(output)* được đưa ra quá nhanh hoặc quá chậm so với đặc tả. Do đó, nhiệm vụ của người kiểm thử là thiết kế các đầu vào *(input)* với thời gian thực hiện nhanh nhất hoặc chậm nhất để phát hiện các vi phạm. Việc thiết kế các đầu vào được quy về bài toán tối ưu trong tính toán tiến hóa để tự động tìm đầu vào với thời gian thực hiện nhanh nhất hoặc chậm nhất. Tuy nhiên, phương pháp này chưa kiểm chứng được ràng buộc thời gian giữa các thành phần như phương pháp được đề xuất trong bài báo này.

Guo và Lee [9] đề xuất phương pháp kết hợp giữa đặc tả và kiểm chứng ràng buộc thời gian cho các hệ thống thời gian thực. Trong đó, ràng buộc thời gian cùng với yêu cầu hệ thống được đặc tả và kiểm chứng bằng môđun TER nets [14]. Giống như [1], phương pháp này chỉ kiểm chứng ở mức mô hình, không phải ở mức cài đặt.

Trong [7] phương pháp sử dụng biểu đồ thời gian của UML được đề xuất để ước lượng thời gian thực thi trong trường hợp xấu nhất của các thành phần trong hệ thống ở thời điểm thiết kế. Thời gian thực thi được ước lượng dựa trên biểu đồ ca sử dụng kết hợp với các thông tin bổ sung về hành vi của người sử dụng hệ thống trong tương lai. Phương pháp này cũng không kiểm chứng ràng buộc thời gian thực thi giữa các thành phần so với đặc tả bằng biểu đồ thời gian.

Jin [12] đề xuất một phương pháp hình thức để kiểm chứng tĩnh thứ tự thực hiện của các phương thức *(method call sequence - MSC)* trong chương trình Java tuần tự. Phương pháp này sử dụng ôtômát hữu hạn trang thái để đặc tả giao thức, các chương trình Java được biến đổi thành các văn phạm phi ngữ cảnh *(context free grammar- CFG)* sử dụng công cụ Accent . 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 theo đặc tả giao thức, và ngược lại. Ưu điểm của phương pháp này đó là các vi phạm có thể được phát hiện sớm, tại thời điểm phát triển hoặc biên dịch chương trình. Do đó sự thực thi của chương trình không bị ảnh hưởng.

Deline và Fahndrich [14] đề xuất phương pháp kiểm chứng vào thời điểm thực thi sự tuân thủ giữa cài đặt và 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. Tuy nhiên, các phương pháp này chưa kiểm chứng ràng buộc thời gian giữa các thành phần.

Yoonsik và Perumandla [4, 5] mở rộng ngôn ngữ đặc tả, và trình biên dịch JML để biểu diễn giao thức tương tác bằng biểu thức chính quy. Sau đó, biểu thức chính quy được biên dịch thành mã thực thi để chạy đan xen với chương trình gốc để kiểm chứng sự tuân thủ giữa cài đặt so với đặc tả giao thức tương tác. Các hành vi của chương trình gốc sẽ không

bị thay đổi ngoại trừ thời gian và kích thước. Như[12], phương pháp này chưa kiểm chứng các ràng buộc về thời gian giữa các thành phần so với đặc tả.

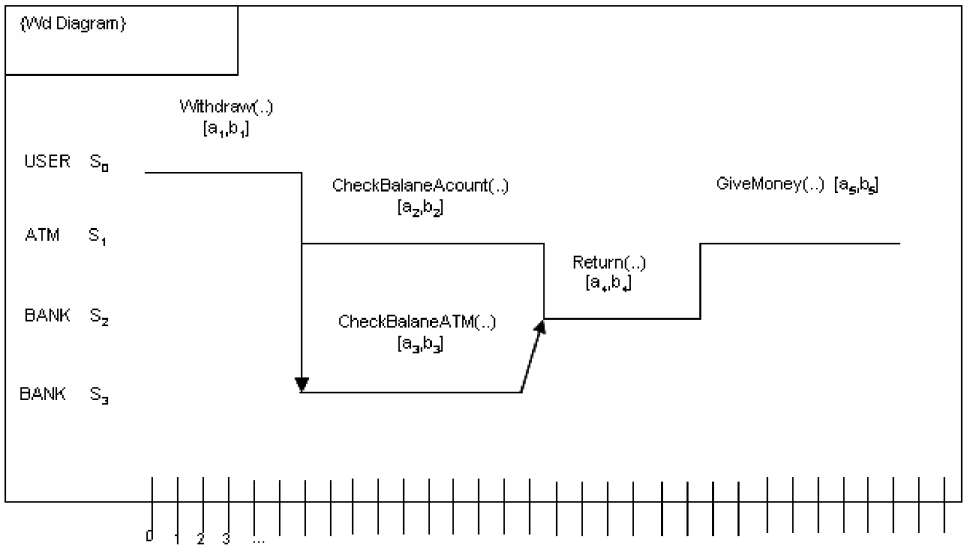
3. PHƯƠNG PHÁP KIỂM CHỨNG RÀNG BUỘC THỜI GIAN GIỮA CÀI ĐẶT THÀNH PHẦN PHẦN MỀM SO VỚI ĐẶC TẢ

Giả sử hệ thống rút tiền tự động của máy ATM *(ATM - Automatic Teller Machine)* gồm ba thành phần khách hàng được biểu diễn bằng đối tượng user, bộ điều khiển ATM được biểu diễn bằng đối tượng ATM, và thành phần cuối cùng máy chủ ngân hàng được biểu diễn bằng đối tượng Bank. Khi đó, bài toán kiểm chứng các ràng buộc thời gian thực thi giữa các thành phần của hệ thống ATM được đặc tả như sau, Hình 1.

1. Thời gian thực thi của phương thức `Withdraw(..)` được thực hiện với đoạn thời gian đáp ứng cho phép là $[a_1, b_1]$. Sau đó lần lượt đến các phương thức `CheckBalanceAccount(..)`, `CheckBalanceATM(..)` và `Return(..)` được thực hiện với đoạn thời gian cho phép tương ứng là $[a_2, b_2], [a_3, b_3]$ và $[a_4, b_4]$. Cuối cùng, phương thức `GiveMoney(..)` được thực hiện với thời gian đáp ứng là $[a_5, b_5]$.

2. Tổng thời gian thực hiện của các phương thức trên không được vượt qua ngưỡng θ cho phép.

3. Các phương thức `CheckBalanceATM(..)` phải kết thúc trước phương thức `CheckBalanceAccount(..)`. Hai phương thức này được thực hiện song song với nhau.



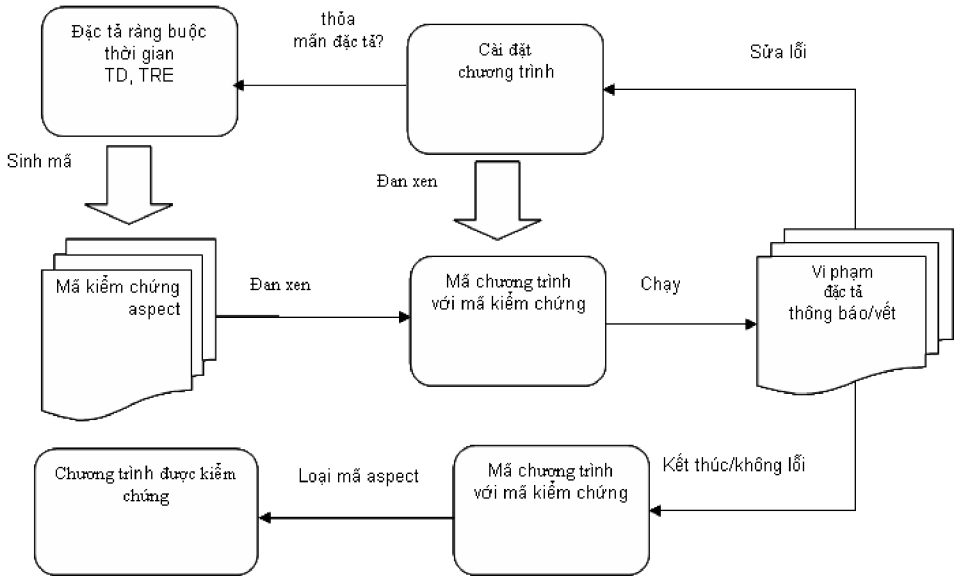
Hình 1. Biểu đồ thời gian của giao thức rút tiền

Bài báo đề xuất phương pháp kiểm chứng sự tuân thủ về ràng buộc thời gian trong các ứng dụng phần mềm như sau, Hình 2.

1. Sử dụng biểu đồ thời gian *(Timing Diagram-TD)* hoặc biểu thức chính quy thời gian *(Timed Regular Expression TRE)* để đặc tả ràng buộc thời gian *(Timing constraint TC)*,

2. Tự động sinh mã aspect từ đặc tả TC,

3. Mã aspect sinh ra được tự động đan vào trước và sau mã thực thi của mỗi thành phần trong chương trình để kiểm chứng động sự tuân thủ với các TC. Khi các chương trình được thực hiện thì các mã đan xen vào có thể phát hiện được chính xác các thành phần vi phạm với đặc tả TC. Trong khi đó, các hành vi của chương trình, và thời gian thực thi của các thành phần sẽ không bị thay đổi.



Hình 2. Phương pháp kiểm chứng sự tuân thủ về ràng buộc thời gian

3.1. Đặc tả các ràng buộc thời gian

Phần này sẽ định nghĩa hình thức các ràng buộc thời gian, sau đó là phương pháp đặc tả các ràng buộc này dựa trên biểu đồ thời gian và biểu thức chính quy thời gian.

Định nghĩa 1. (Ràng buộc thời gian thực thi) Ràng buộc thời gian thực thi của một thành phần TC là đoạn thời gian đáp ứng cho phép của nó khi được thực thi, được biểu diễn bằng một bộ hai thành phần $TC = [a, b]$ trong đó $a, b \in N$ và $a < b$.

Ví dụ trong Hình 1, giả sử $a_1 = 10ms$, $b_1 = 30ms$, và τ `(Withdraw(..))` là thời gian

thực thi của thành phần `Withdraw(..)`. Khi đó ràng buộc thời gian thực thi của thành phần này là đoạn thời gian $[10, 30]$ với $10ms \leq \tau(Withdraw(..)) \leq 30ms$.

Định nghĩa 2. (Ràng buộc thời gian giữa các thành phần tuần tự) Giả sử r_i, và c_i lần lượt là thời điểm bắt đầu và kết thúc thực hiện của một thành phần TC_i, thời gian thực thi $t_i = c_i - r_i, t_i \in [a_i, b_i]$, với i=1,...,n ($t_i$ thỏa mãn ràng buộc thời gian của một thành phần, định nghĩa 1). Khi đó ràng buộc thời gian giữa các thành phần là tổng thời gian thời gian thực thi không được vượt qua của các thành phần $\sum_{i=1}^{n} t_i \leq \theta$, với $\theta \in N$.

Giả sử $\tau(\alpha)$ là thời gian thực thi của thành phần α, và tổng thời gian thực thi của các thành phần tuần tự `Withdraw(..)`, `CheckBalanceAccount(..)`, `Return(..)`, `GiveMoney(..)` không vượt quá ngưỡng $\theta = 55ms$. Khi đó ta có ràng buộc thời gian giữa các thành phần tuần tự này như sau *(Định nghĩa 2)*.

τ `(Withdraw(..))` + τ `(CheckBalanceAccount(..))` + τ `(Return(..))` + τ `(GiveMoney(..))` ≤ 55.

Định nghĩa 3. (Ràng buộc thời gian giữa các thành phần tương tranh) Giả sử $\tau(\alpha_1)$, $\tau(\alpha_2), ..., \tau(\alpha_n)$ là thời gian thực thi tương ứng của n thành phần tương tranh $\alpha_1, \alpha_2, ...$ $, \alpha_n$. Khi đó ràng buộc thời gian giữa các thành phần này được định nghĩa như sau $\tau(\alpha_i) \bowtie \tau(\alpha_j)$ với $\bowtie \in \{<, \leq, >, \geq, \neq, =\}$ và $i, j = 1..n, i \neq j$.

Giả sử hai thành phần `CheckBalanceAccount(..)` và `CheckBalanceATM(..)` được thực hiện song song tại cùng một thời điểm, Hình 1. Với ràng buộc là thành phần `CheckBalanceATM(..)` phải kết thúc trước thành phần `CheckBalanceAccount(..)`. Khi đó ta có ràng buộc thời gian giữa hai thành phần tương tranh như sau *(Định nghĩa 3)*. τ `(CheckBalanceAccount(..))` > τ `(CheckBalanceATM(..))`.

3.1.1. *Biểu thức chính quy thời gian*

Biểu thức chính quy thời gian *(Timed Regular Expression -TRE)* là sự mở rộng của biểu thức chính quy để đặc tả các ràng buộc thời gian độc lập với mã nguồn chương trình để sinh ra mã aspect. Chúng tôi định nghĩa như sau.

Định nghĩa 4. (Biểu thức chính quy thời gian) TRE là một bộ ba TRE = ¡C, M, S¿. Trong đó, $C = \{c_1, c_2, ..., c_n\}$ là tập hữu hạn các thành phần, $M = \{m_1, m_2, ..., m_m\}$ là tập hữu hạn các phương thức, $S = \{s_1, s_2, ..., s_k\}$ là tập hữu hạn các biểu thức biểu diễn mối liên hệ giữa các thành phần được định nghĩa như sau $s \triangleq c.m[a, b] \mid s|s \mid s \parallel s \mid s^* \mid s^+$, trong đó, $m \in M$; $a, b \in N$; $c \in C$;$s, s_i, s_j \in S$ với $i, j = \{1..k\}$; $s_i \rightarrow s_j$ là sự kết hợp của hai hoặc nhiều biểu thức tuần tự; $s_i \mid s_j$: phép hoặc; $s_i \parallel s_j$: phép song song (các phương thức trong s_i và s_j có thể được thực hiện song song); s^*: không hoặc nhiều phép lặp; s^+: một hoặc nhiều phép lặp.

Ví dụ biểu đồ thời gian trong Hình 1 được biểu diễn bằng một biểu thức chính quy thời

gian TRE:
$USER.Withdraw(..)[a_1, b_1] \rightarrow (ATM.CheckBalaneAccount(..)[a_2, b_2] \rightarrow BANK.CheckBalaneATM(..)[a_3, b_3]) \rightarrow BANK.Return(..)[a_4, b_4] \rightarrow ATM.GiveMoney(..)[a_5, b_5]$, trong đó, thành phần `USER.Withdraw(..)` được thực hiện trước với ràng buộc thời gian thuộc đoạn $[a_1, b_1]$, sau đó là thành phần `ATM.CheckBalaneAccount(..)` và `BANK.CheckBalaneATM(..)` được thực hiện song song nhau với ràng buộc thời gian lần lượt thuộc các đoạn $[a_2, b_2]$ và $[a_3, b_3]$. Tiếp theo là các thành phần `BANK.Return(..)` và `ATM.GiveMoney(..)` được thực hiện tuần tự với các ràng buộc tương ứng thuộc các đoạn $[a_4, b_4]$ và $[a_5, b_5]$.

3.1.2. *Biểu đồ thời gian*

Biểu đồ thời gian *(Timing Diagram - TD)* trong UML2.0 đặc tả thứ tự thực hiện của các phương thức cùng với ràng buộc về thời gian. Chúng tôi định nghĩa hình thức như sau.

Định nghĩa 5. (Biểu đồ thời gian) TD là một bộ sáu $TD =< S, S_0, C, M, \delta, F >$. Trong đó, S là tập hữu hạn các trạng thái, C là tập các thành phần, M là tập các phương thức. $\delta \subseteq S \times C.M[a, b] \rightarrow S$ là hàm chuyển trạng thái với $a, b \in N$ và $a \leq b$ là ràng buộc thời gian. $S_0, F \in S$ lần lượt là các trạng thái đầu và kết thúc.

Hình 1 biểu diễn biểu đồ thời gian cho một giao thức rút tiền của hệ thống ATM, thứ tự thực hiện của các phương thức được thể hiện bằng các cung trong biểu đồ, trong đó:

- $S = \{S_0, S_1, S_2, S_3, F\}$,
- M={Withdraw,CheckBalaneAccount,CheckBalaneATM,Return, GiveMoney},
- C = {USER,ATM,BANK},
- $\delta = \{S_0.USER.Withdraw[a_1, b_1] \rightarrow S_1, S_0.USER.Withdraw[a_1, b_1] \rightarrow S_3, S_1.ATM.CheckBalaneAccount[a_2, b_2] \rightarrow S_2, S_3.BANK.CheckBalaneATM[a_3, b_3] \rightarrow S_2, S_2.BANK.Return[a_4, b_4] \rightarrow S_1, S_1.ATM.GiveMoney[a_5, b_5] \rightarrow F\}$.

3.2. Sinh mã aspect

Ta định nghĩa một mẫu để biểu diễn các aspect được sinh ra từ các đặc tả ràng buộc thời gian như trong Hình 3. Trong đó, các biến địa phương được định nghĩa để tính thời gian thực thi của mỗi phương thức khi nó được thực hiện, và tính tổng thời gian thực hiện của các phương thức *(dòng 2, 3, 6, và 7)*. Đặc tả ràng buộc thời gian dưới dạng biểu đồ thời gian được kết xuất ra tệp dạng xmi hoặc dạng txt đối với biểu thức chính quy. Trong thực nghiệm chúng tôi đã xây dựng thuật toán đọc tên các phương thức và ràng buộc thời gian tương ứng từ các đặc tả này *(dòng 4 và 5)*. Các ràng buộc thời gian đọc được sẽ được đưa vào điều kiện để so sánh với thời gian thực thi *(dòng 8)*, và thông báo các vi phạm nếu có

(dòng 9). Ràng buộc thời gian trong các định nghĩa 1, 2 và 3 được dịch thành các biểu thức điều kiện trong aspect mẫu *(dòng 8)*.

Bảng 1. Sinh mã aspect từ các đặc tả ràng buộc thời gian

import org.aspectj.lang.joinpoint ;
variables
Variables are declared here ;
...
aspect *AspectName*{
before() : (execution(* *.*(..))) && !within(*AspectName*){
1. $st = 0$;
2. *Get* τ_1 ; // the current system time ;
}
after() : (execution(* *.*(..)))&& !within(*AspectName*){
3. *Get* τ_2 ; // the current system time ;
4. *Get method name from XMI flie*$(task1, task2, ...)$;
5. *Get lower and upper bound on timing from XMI file*$(r1, r2, ...)$;
6. $\tau = \tau_2 - \tau_1$; // Calculate the execution time of the method ;
7. $st+ = \tau$; // the execution time of sequential method ;
8.**if** $(\xi(\tau, r1, r2, ...) = false)$// *Timing constraint conditions* ;
9. *Produce violation reports* ;
}

3.3. 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 *(source code weaving)*, 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 *(byte code weaving)*, 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 xen 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 phải sinh và biên dịch lại mã aspect.

4. THỰC NGHIỆM

Phương pháp này đã được cài đặt thành một công cụ kiểm chứng *(TCVG Timing Constraint Verification Generator)*. Đầu vào của công cụ TCVG là các đặc tả ràng buộc thời

gian cho dưới dạng tệp có phần mở rộng là txt biểu diễn biểu thức chính quy thời gian, và dạng xmi biểu diễn biểu đồ thời gian của UML. Đầu ra là các mã kiểm chứng aspect của AspectJ.

Thực nghiệm được tiến hành với các chương trình mô phỏng của hệ thống ATM, Hình 1. Cấu hình máy tính sử dụng vi xử lý 1500MHz, RAM 512, hệ điều hành Windows XP. Từ đặc tả ràng buộc thời gian của giao thức này chúng tôi sử dụng công cụ TCVG để sinh ra mã aspect và đan với chương trình ATM mô phỏng để kiểm chứng sự tuân thủ về ràng buộc thời gian trong các Định nghĩa 1, 2 và 3. Với mỗi thành phần chúng tôi xây dựng các test đúng, sai khác nhau, trong đó, các test đúng thì các thành phần được cài đặt tuân thủ theo đặc tả ràng buộc thời gian và ngược lại.

Bảng 2. Ca kiểm thử đúng/sai của phương thức withdraw với ràng buộc thời gian thực thi [726082, 143658] nano giây

```
public static long
correctTestWithdraw(long n){
long  max=5000000;
//Your amount is not greater than
5000000
  while (n > max) {
     n = n−100;
  }
  return n;
}
```

(a) Ca kiểm thử đúng

```
public static long
wrongTestWithdraw(long n){
long  max=5000000;
// Your amount is not greater than
5000000
  if (n<=max) return n
  else
     return wrongTest(n-100)
}
```

(b) Ca kiểm thử sai

Bảng 3. Kết quả thực nghiệm

Thành phần	Ràng buộc thời gian thực thi (nano seconds)	Số test đúng/sai	Phát hiện đúng/sai
Withdraw	[10, 20], [20,30],. . . , [90,100]	25/25	25/25
CheckBalanceAccount	[10, 20], [20,30],. . . , [90,100]	25/25	25/25
CheckBalanceATM	[10, 20], [20,30],. . . , [90,100]	25/25	25/25
Return	[10, 20], [20,30],. . . , [90,100]	25/25	25/25
GiveMoney	[10, 20], [20,30],. . . , [90,100]	25/25	25/25
Ràng buộc thời gian của các thành phần tuần tự			
Tổng thời gian thực hiện ($\leq$)	50,100,150...,500	25/25	25/25
Ràng buộc thời gian giữa hai thành phần tương tranh			
CheckBalanceAccount	thời điểm bắt đầu t	25/25	25/25
CheckBalanceATM	5, 10, 15,. . . , 50	25/25	25/25

Hình 4 mô tả một test đúng bên trái và sai bên phải của thành phần withdraw với đặc tả thời gian đáp ứng là [726082, 143658] nano giây. Trong thực nghiệm chúng tôi truyền tham số n bằng 6000000, với test đúng được viết dưới dạng lặp thì thời gian thực thi là 825524

nano giây thỏa mãn ràng buộc thời gian thực thi trong đoạn [726082, 143658]. Ngược lại với Test sai được viết dưới dạng đệ quy thì thời gian thực thi là 2111442 nano giây không thỏa mãn ràng buộc thời gian thực thi trong đoạn [726082, 143658].

Các chương trình mô phỏng được chạy 25 lần cho mỗi test đúng và sai với các đặc tả ràng buộc thời gian thực thi trong cột 2, Bảng 1 *(theo Định nghĩa 1)*, trong đó, ràng buộc thời gian của các thành phần tuần tự tăng dần từ 50ns đến 500ns *(theo Định nghĩa 2)*. Ràng buộc thời gian kết thúc trước/sau giữa hai thành phần song song *(theo Định nghĩa 3)* CheckBalanceAccount và CheckBalanceATM ở thời điểm bắt đầu t và $t \pm \xi ns$, với $\xi = 5, 10, 15, ..., 50$. Kết quả thực nghiệm cho thấy phương pháp đã phát hiện được đủ các test đúng và sai *(cột 4, Bảng 3)* với 25 test đúng các vi phạm về ràng buộc thời gian được phát hiện chính xác, Bảng 1.

Qua các kết quả thực nghiệm cho thấy:

(i) Các aspect được sinh ra đúng so với các đặc tả ràng buộc thời gian và nhất quán giữa biểu thức chính quy thời gian và biểu đồ thời gian.

(ii) Các aspect không làm thay đổi hành vi của chương trình gốc. Và

(iii) Đã phát hiện được các vi phạm ràng buộc thời gian giữa các thành phần.

KẾT LUẬN

Nhiều hệ thống an toàn-bảo mật là các hệ thống thời gian thực, trong các hệ thống này ràng buộc thời gian khi bị vi phạm sẽ gây ra các lỗi hệ thống. Các kỹ thuật truyền thống như mô phỏng, kiểm thử thường chỉ ước lượng được thời gian thực thi của các thành phần hệ thống với một độ tin cậy nào đó.

Để tăng cường sự tin cậy về ràng buộc thời gian trong các hệ thống thời gian thực. Bài báo này đề xuất một phương pháp kiểm chứng sự tuân thủ giữa sự cài đặt của các thành phần phần mềm so với đặc tả các ràng buộc thời gian. Phương pháp này sử dụng biểu đồ thời gian *(Timing Diagram)* của UML và biểu thức chính quy thời gian *(Timied Regular Expression)* để đặc tả các ràng buộc thời gian. Các mã aspect được tự động sinh ra từ các đặc tả này sẽ đan tự động với mã của các thành phần để kiểm chứng sự tuân thủ giữa sự cài đặt của các thành phần so với các đặc tả ràng buộc thời gian ở thời điểm thực thi.

Phương pháp này đã được cài đặt thành một công cụ kiểm chứng và chạy thử nghiệm với ngôn ngữ lập trình Java thông qua một số lớp thư viện chuẩn của Java với các bộ test khác nhau. Kết quả thử nghiệm ban đầu cho thấy phương pháp được đề xuất hoàn toàn có thể phát hiện được vi phạm ràng buộc thời gian của các thành phần so với đặc tả. Hạn chế của phương pháp này cũng như các phương pháp kiểm chứng động khác là phải thực thi chương trình, vị phạm ràng buộc thời gian chỉ được phát hiện trong bước kiểm thử, mã aspect được đan vào sẽ làm tăng kích thước của các chương trình.

Trong những nghiên cứu tiếp theo, ta sẽ kết hợp phương pháp này với phương pháp của Dymek [7], Joachim [18] để tự động xây dựng các ca kiểm thử, và kết hợp với các phương pháp kiểm chứng tĩnh khác như kiểm chứng mô hình. Tiến tới phát triển một phương pháp kiểm chứng tự động toàn diện từ mức mô hình đến mức cài đặt.

TÀI LIỆU THAM KHẢO

[1] Anh Hoang Truong, Thanh Binh Trinh, Dang Van Hung, Viet Ha Nguyen, Nguyen Thi Thu Trang, and Pham Dinh Hung, Checking interface interaction protocols using Aspect-oriented programming, *SEFM 08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods* IEEE Computer Society, 2008 , pages 382386.

[2] Thanh Binh Trinh, Tuan Anh Do, Ninh Thuan Truong, and Viet Ha Nguyen, Checking the compliance of timing constraints in software applications, *1st Intern. Conf. on Knowledge and Systems Engineering (KSE 2009)*, Hanoi, Vietnam, Oct 14-15, 2009 (p. 220–225).

[3] Colyer and A. Clement, Aspect-oriented programming with AspectJ, *IBM Syst. J.* **44** (2) (2005) 301–308.

[4] Thanh Binh Trinh, Anh Hoang Truong, and Viet Ha Nguyen, Checking protocol conformance in component models using Aspect oriented programming, *Advances in Computer Science and Engineering*, Actapress, 2009 (pages 150–155).

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit (ed.), *Aspect-Oriented Software Development*, Addison-Wesley, Boston, 2005.

[6] A. Benveniste, M. Siegel, L. Holenderski, K. Winkelmann, E. Sefton, E. Rutten, P. Le Guernic, and T. Gautier, Safety critical embedded systems design: the sacres approach, *Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT98 school*, Lyngby, Denmark, September 1998.

[7] Joachim Wegener and Matthias Grochtmann, Verifying timing constraints of real-time systems by meansof evolutionary testing, *Real-Time Syst.* **15** (3) (1998) 275298.

[8] Y. Cheon and A. Perumandla, Specifying and checking method call sequences in JML. In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, CSREA Press, 2005 (511516).

[9] Y. Cheon and A. Perumandla, Specifying and checking method call sequences of Java programs, *Software Quality Control* **15** (1) (2007) 725.

[10] Hui Guo and Woo Jin Lee, Compositional verification of timing constraints for embedded real-time systems, *ACOS07: Proceedings of the 6th Conference on WSEAS International Conference on Applied Computer Science*, Stevens Point, Wisconsin, USA, 2007 (570575).

[11] Y. Jin, Formal verification of protocol properties of sequential Java programs, *COMPSAC 07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, IEEE Computer Society, Washington, DC, USA, 2007 (475482).

[12] Joost-Pieter Katoen: "Concepts, Algorithms, and Tools for Model Checking", Lecture Notes of the Course Mechanised Validation of Parallel Systems (1999).

[13] C.B. Jones, Theorem proving and software engineering, *Software Engineering Journal* **3** (1) (Jan 1988).

[14] Dariusz Dymek and Leszek Kotulski, Estimation of system workload time characteristic using uml timing diagrams, *EPCOS-RELCOMEX 08: Proceedings of the 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, Washington, DC, USA, 2008 (914).

[15] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda: "Model Checking Programs" (2002).

[16] Paddy Nixon and Lihua Shi, Concurrent semantics for structured design methods, *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, London, UK, UK, 1996 (158169).

[17] Gerard J.Holzmann, *The SPIN Model Checker Primer and Reference Manual*, Addison-Wesley, 2003.

[18] Ninh Thuan Truong, Thanh Binh Trinh, Viet Ha Nguyen, Coordinated Consensus Analysis of Multi-agent Systems using Event-B, *7th IEEE Intern. Conf. on Software Engineering and Formal Method (SEFM 2009)*, Hanoi, Vietnam, 23-27 November 2009 (201–209).

Nhận bài ngày 3 - 3 - 2010
Nhận lại sau sửa ngày 26 - 7 -2010