Chương 1

Mở đầu

1.1 Bối cảnh

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 có thể làm tổn thất trực tiếp sinh mạng con người. Do đó, nhu cầu nghiên cứu và đề xuất các phương pháp để kiểm chứng phần mềm ngày càng trở lên cần thiết.

1.2 Một số nghiên cứu liên quan

1.2.1 Kiểm chứng thiết kế

Edmunds đề xuất ngôn ngữ đặc tả trung gian OCB (*Object-oriented Concurrent-B-OCB*) để nối liền giữa đặc tả bằng Event-B với sự cài đặt của các chương trình hướng đối tượng, tương tranh. Đặc tả OCB sẽ được chuyển tự dộng sang mô hình của Event-B và mã chương trình Java. Các chương trình Java được chuyển đổi sẽ tuân thủ theo đặc tả OCB của nó.
Ben Younes và các tác giả khác đề xuất các luật để chuyển đổi từ đặc tả bằng biểu đồ hoạt động (*Activity Diagram*) của UML sang đặc tả bằng Event-B. Đóng góp chính của nghiên cứu này là chuyển đổi từ một đặc tả trực quan sang hình thức và chứng minh tự động một mô hình thỏa mãn các thuộc tính của nó. Tuy nhiên việc chuyển đổi chưa được thực hiện tự động hoàn toàn, hơn nữa nghiên cứu này mới đưa ra một ví dụ để minh họa khả năng chuyển đổi của nó.

Ball đề xuất các mẫu thiết kế để đặc tả sự tương tác giữa các tác tử phần mềm, các mẫu thiết kế sau đó được chuyển đổi sang đặc tả bằng Event-B. Tuy nhiên, việc chuyển đổi từ mẫu thiết kế sang đặc tả bằng Event-B chưa được tự động. Giao thức tương tác tương tác được đặc tả lại với Event-B dựa vào mẫu thiết kế của nó.

1.2.2 Kiểm chứng mã nguồn

J-LO (*Java Logical Observer*) là một công cụ kiểm chứng sự tuân thủ của các chương trình Java so với các đặc tả của nó bằng logic thời gian tuyến tính (*linear temporal logic*). J-LO mở rộng trình biên dịch AspectBench để đan các mã aspect được sinh ra vào chương trình Java cần kiểm chứng nhằm phát hiện các lỗi hạt giống (*seeded errors*). Tuy nhiên, J-LO sẽ gây ra chi phí về thời gian thực thi của các chương trình cần kiểm chứng là quá lớn, do đó nó thường được sử dụng để kiểm chứng các chương trình Java có kích thước nhỏ.

Bodden và Havelund mở rộng ngôn ngữ lập trình hướng khía cạnh AspectJ với ba phương thức mới `lock()`, `unlock()` và `maybeShate()`. Các phương thức này cho phép người lập trình dễ dàng cài đặt các thuật toán phát hiện lỗi trong các chương trình Java tương tranh. Theo các tác giả thì phương pháp này có thể phát hiện tốt các lỗi tương tranh về dữ liệu (*data race*), tuy nhiên chưa phát hiện được các lỗi liên quan đến tương tranh khác như tắc nghẽn (*deadlock*).

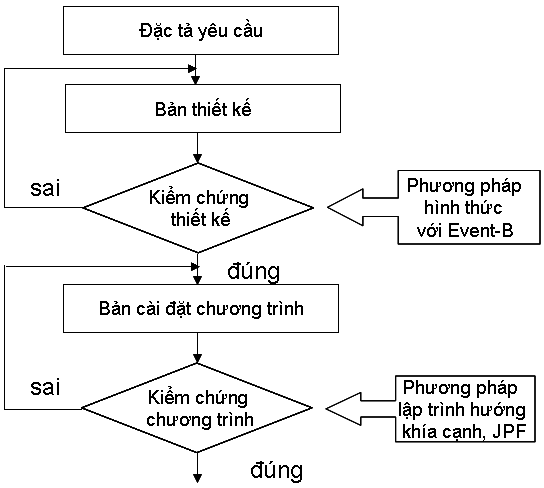
Jin đề xuất một phương pháp hình thức để kiểm chứng tĩnh sự tuân thủ giữa cài đặt mã nguồn và đặc tả thứ tự thực hiện của các phương thức (*method call sequence - MCS*) trong các chương trình Java. Ưu điểm của phương pháp này là các vi phạm có thể được phát hiện sớm, tại thời điểm phát triển hoặc biên dịch chương trình mà không cần chạy thử chương trình. Tuy nhiên, phương pháp này chưa kiểm chứng được các chương trình tương tranh.

Trong các phương pháp về JML, 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. Các tiền và hậu điều kiện này được biên dịch và chạy 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.

1.3 Nội dung nghiên cứu

Trong luận án này, chúng tôi tập trung nghiên cứu và đề xuất các phương pháp để kiểm chứng chương trình tương tranh ở các pha thiết kế và cài đặt mã nguồn chương trình (Hình 1.1). Tại mức thiết kế, chúng tôi sử dụng phương pháp hình thức với Event-B để kiểm chứng bản thiết kế của chương trình Java tương tranh nhằm phát hiện lỗi ở mức cao. Để phát hiện lỗi ở mức thấp chúng tôi sử dụng phương pháp lập trình hướng khía cạnh và bộ công cụ kiểm chứng mô hình JPF (*Java PathFinder*) để kiểm chứng sự tuân thủ giữa sự cài đặt của các chương trình Java tương tranh so với đặc tả thiết kế của nó.



Hình 1.1 – Kiểm chứng mức thiết kế và cài đặt chương trình.

1.4 Cấu trúc luận án

Luận án gồm sáu chương chính. Trong đó, Chương 2 giới thiệu một số kiến thức nền cho các đóng góp của luận án trong các chương còn lại. Chương 3 và 4 đề xuất hai phương pháp đặc tả và kiểm chứng sự tương tác giữa các thành phần tương tranh sử dụng phương pháp hình thức với Event-B. Chương 5 và 6 đề xuất hai phương pháp sử dụng lập trình hướng khía cạnh với AOP để kiểm chứng sự tuận thủ giữa bản cài đặt của chương trình so với bản thiết kế của nó.

Chương 2

Kiến thức cơ sở

2.1 Kiểm chứng phần mềm

2.1.1 Kiểm chứng hình thức

2.1.1.1 Kiểm chứng mô hình

Kiểm chứng mô hình (*model checking*) được sử dụng để xác định tính hợp lệ của một hay nhiều tính chất mà người dùng quan tâm trong một mô hình phần mềm cho trước. Cho mô hình M và thuộc tính p cho trước, nó kiểm tra liệu thuộc tính p có thỏa mãn trong mô hình M hay không, ký hiệu $M \models p$.

2.1.1.2 Chứng minh định lý

Phương pháp chứng minh định lý (*theorem proving*) sử dụng các kĩ thuật suy luận để chứng minh tính đúng đắn của một công thức hay tính khả thỏa của một công thức F với tất cả các mô hình, ký hiệu $\models F$.

2.1.2 Kiểm chứng tại thời điểm thực thi

Kiểm chứng động (*runtime verification*) là kỹ thuật kết hợp giữa kiểm chứng hình thức và thực thi chương trình để phát hiện các lỗi của hệ thống dựa trên quá trình quan sát input/output khi thực thi chương trình. So với phương pháp kiểm chứng tĩnh thì kiểm chứng động được thực hiện trong khi thực thi hệ thống. Do đó, kiểm chứng động còn được gọi là kiểm thử bị động (*passive testing*). Kiểm chứng động nhằm bảo đảm sự tuân thủ giữa cài đặt hệ thống phần mềm so với đặc tả của nó.

2.2 Một số vấn đề trong chương trình tương tranh

Trong các chương trình tương tranh, có hai thuộc tính cơ bản cần phải bảo đảm là an toàn (*safety*) và thực hiện được (*liveness*). Một số vấn đề về tương tranh liên quan đến hai thuộc tính này được mô tả như sau. Sự xung đột (*interference*) xảy ra khi hai hoặc nhiều tiến trình đồng thời truy cập một biến chia sẻ, trong đó có ít nhất một tiến trình ghi. Khi đó giá trị của biến chia sẻ và kết quả của chương trình sẽ phụ thuộc vào sự đan xen (*interleaving*) hay thứ tự thực hiện của các tiến trình. Sự xung đột còn được gọi là cạnh tranh dữ liệu (*data race*). Tắc nghẽn xảy ra khi hệ thống (*chương trình*) không thể đáp ứng được bất kỳ tín hiệu hoặc yêu cầu nào. Có hai dạng tắc nghẽn, dạng một xảy ra khi các tiến trình dừng lại và chờ đợi lẫn nhau, ví dụ một tiến trình nắm giữ một khóa mà các tiến trình khác mong muốn và ngược lại. Dạng hai xảy ra khi một tiến trình chờ đợi một tiến trình khác không kết thúc. Sự đói (*starvation*) liên quan đến sự tranh chấp tài nguyên, vấn đề này xảy ra khi một tiến trình không thể truy cập đến các tài nguyên chia sẻ.

2.3 Sự tương tranh trong Java

Sự tương tranh trong Java được thực hiện thông qua cơ chế giám sát các tiến trình, hành vi của tiến trình được mô tả trong phương thức `run`. Sự thực thi của một tiến trình có thể được điều khiển bởi các tiến trình khác thông qua các phương thức `stop`, `suspend` và `resum`.

2.3.1 Mô hình lưu trữ (JMM-Java Memory Model)

Trong Java, các tiến trình tương tác với nhau thông qua việc đọc ghi dữ lệu chia sẻ. Mô hình lưu trữ JMM biểu diễn sự tương tác giữa các tiến trình trong bộ nhớ. JMM định nghĩa các luật về sự tương tác giữa các tiến trình và các hành vi của chương trình Java tương tranh, do đó người lập trình có thể thiết kế và cài đặt chương trình một cách đúng đắn và phù hợp. Tuy nhiên vấ đề tránh tắc nghẽn (*deadlock*), cạnh tranh dữ liệu (*data race*) vẫn chưa được giải quyết trong mô hình này.

2.3.2 Ngôn ngữ mô hình hóa cho Java (JML-Java Modeling Language)

JML là ngôn ngữ đặc tả hình thức cho Java dựa trên logic Hoare để đặc tả và kiểm chứng các tiền điều kiện (*precondition*), hậu điều kiện (*postcondition*) và các bất biến (*invariant*). Đặc tả JML được nhúng vào mã nguồn Java và bắt đầu bởi kí hiệu //@< *JML specification* > hoặc /*@ < *JML specification* > @*/ theo sau là các thuộc tính cần đặc tả. Một số từ khóa cơ bản như `requires`, `ensures` định nghĩa các biểu thức tiền điều kiện, hậu điều kiện và `invariant` định nghĩa các bất biến. Các thuôc tính được đặc tả trong JML sẽ được biên dịch và thực thi cùng với mã nguồn để kiểm chứng sự thỏa mãn nó.

2.3.3 Công cụ kiểm chứng mã Java (JPF-Java PathFinder)

Java PathFinder (JPF) là một công cụ kiểm chứng các chương trình Java tương tranh dưới dạng bytecode. JPF được sử dụng một cách linh hoạt dưới dạng giao diện dòng lệnh hoặc tích hợp vào trong các môi trường phát triển ứng dụng Java như Netbean, Eclipse. JPF là một ứng dụng Java mã nguồn mở cho phép ta cấu hình để sử dụng nó theo các cách khác nhau và mở rộng nó. Các phiên bản hiện tại hỗ trợ kiểm chứng các thuộc tính như tắc nghẽn (*deadlock*), cạnh tranh dữ liệu (*data race*) và các ngoại lệ chưa được xử lý (*unhandled exceptions*). Tuy nhiên, JPF cũng cho phép người sử dụng mở rộng để kiểm chứng các thuộc tính khác dựa trên các giao diện đã được thiết kế sẵn như giao diện `property` và `listener`.

2.4 Phương pháp hình thức với Event-B

Event-B là một phương pháp hình thức dựa trên lý thuyết tập hợp, ngôn ngữ thay thế tổng quát và logic vị từ bậc một (*first order logic*). Event-B bao gồm ký pháp, phương pháp và công cụ hỗ trợ quá trình phát triển phần mềm bằng cách làm mịn (*refinement*). Quá trình làm mịn bắt đầu bằng cách xây dựng các máy trừu tượng sau đó làm mịn dần cho đến khi nhận được một máy thực thi, tương tự như mã nguồn chương trình.

2.4.1 Máy và Ngữ cảnh

Các mô hình Event-B được mô tả bởi hai cấu trúc cơ bản là Máy (*machine*) và Ngữ cảnh (*context*). Trong đó, Máy dùng để mô tả phần động của mô hình bao gồm biến, bất biến, định lý và các sự kiện tương tác với môi trường. Ngữ cảnh mô tả phần tĩnh của mô hình, chứa các tập hợp, hằng, tiên đề và định lý. Một mô hình có thể chỉ gồm Máy hoặc Ngữ cảnh hoặc sự kết hợp giữa Máy và Ngữ cảnh. Một Máy có thể không hoặc tham chiếu một vài Ngữ cảnh. Các Máy và Ngữ cảnh của mô hình được làm mịn bằng cách bổ sung các hằng, biến, bất biến, định lý, sự kiện.

2.4.2 Sự kiện

Mô hình hệ thống với Event-B được bắt đầu từ các sự kiên trừu tượng quan sát được có thể xảy ra trong hệ thống, từ đó đặc tả các trạng thái và hành vi của hệ thống ở mức trừu tượng cao hơn. Một sự kiện e tác động lên (*một danh sách*) biến trạng thái v, với điều kiện $G(v)$ và hành động $A(v)$, sẽ được mô tả như sau :

$$e \mathrel{\widehat{=}} \textbf{when } G(v) \textbf{ then } A(v) \textbf{ end}$$

2.4.3 Phân rã và kết hợp

Một trong những đặc trưng quan trọng của Event-B đó là khả năng bổ sung các sự kiện mới trong quá trình làm mịn, tuy nhiên khi bổ sung các sự kiện sẽ làm tăng độ phức tạp của tiến trình làm mịn do phải xử lý nhiều sự kiện và nhiều biến trạng thái. Ý tưởng chính của sự phân rã là phân chia mô hình M thành các mô hình con $M_1, ..., M_n$, các mô hình con này dễ dàng được làm mịn hơn so với mô hình ban đầu.

2.4.4 Sinh mệnh đề cần chứng minh

RODIN (*Rigorous Open Development Environment for Complex Systems*) là một bộ công cụ mã nguồn mở dựa trên nền Eclipse để mô hình và chứng minh tự động trong Event-B. Trong luận án này chúng tôi sử dụng bộ công cụ RODIN để mô hình, làm mịn, sinh và chứng minh tự động các mệnh đề cần chứng minh để bảo đảm tính đúng đắn của mô hình.

2.5 Ngôn ngữ mô hình hóa UML

2.5.1 Biểu đồ tuần tự

Biểu đồ tuần tự (*Sequence Diagram-SD*) là một dạng biểu đồ phổ biến của UML sử dụng để biểu diễn các phần tử logic của hệ thống. Một biểu đồ tuần tự gồm hai phần chính, các trục dọc biểu diễn các đối tượng hoặc các tiến trình, các mũi tên nằm ngang biểu diễn thứ tự trao đổi thông điệp giữa các đối tượng một cách tuần tự.

2.5.2 Máy trạng thái giao thức

Biểu đồ máy trạng thái giao thức (*Protocol State Machine-PSM*) là một dạng đặc biệt của biểu đồ SD được bổ sung vào UML 2.0, PSM được sử dụng để đặc tả giao thức tương tác hay thứ tự thực hiện của các phương thức giữa các đối tượng.

2.5.3 Biểu đồ thời gian

Biểu đồ thời gian (*Timing Diagram-TD*) là một dạng biểu đồ mới được bổ sung vào UML 2.0 để mô hình hành vi của các đối tượng cùng với các ràng buộc thời gian của nó. Thông thường, TD được sử dụng để đặc tả ràng buộc thời gian trong các hệ thống thời gian thực, hệ thống nhúng, tuy nhiên nó cũng có thể dùng để mô hình các hệ thống nghiệp vụ khác.

2.6 Lập trình hướng khía cạnh

Phương pháp lập trình hướng khía cạnh (*Aspect-Oriented Programming - AOP*) là phương pháp lập trình phát triển trên tư duy tách biệt các mối quan tâm khác nhau thành các môđun khác nhau. 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 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.

Chương 3

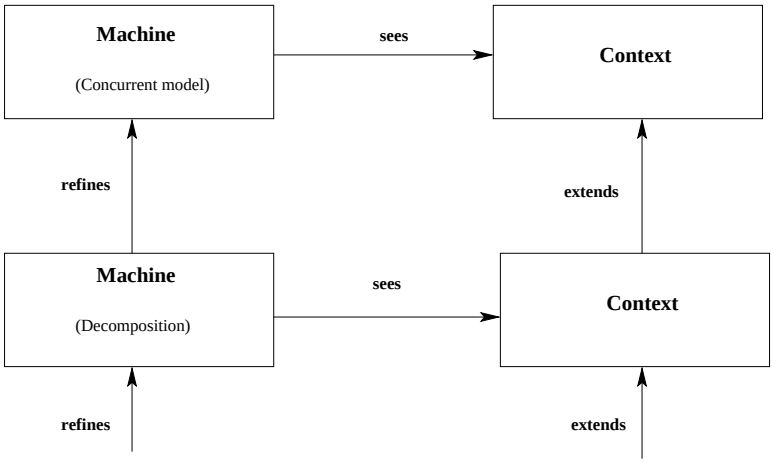
Ràng buộc thứ tự giữa các tiến trình tương tranh

3.1 Giới thiệu

Trong chương này luận án đề xuất một cách tiếp cận để đặc tả và kiểm chứng giao thức tương tác giữa các tiến trình tương tranh sử dụng phương pháp hình thức với Event-B. Chúng tôi minh họa phương pháp đề xuất cho các vấn đề vùng xung đột (*critical section*), cung cấp-tiêu thụ (*producer-consumer*) và đọc-ghi dữ liệu (*reader-writer*).

3.2 Đặc tả và kiểm chứng ràng buộc thứ tự giữa các tiến trình tương tranh

Trong Event-B, một trạng thái của mô hình được định nghĩa bởi một tập các biến biểu diễn cho bất kỳ một đối tượng toán học nào trong lý thuyết tập hợp. Ngoài các định nghĩa về biến, các bất biến là các vị từ được biểu diễn trong logic vị từ bậc một và lý thuyết tập hợp. Sự kết hợp của các biến và bất biến tạo thành trạng thái, một trạng thái của mô hình là một tập trừu tượng.
Với cơ chế làm mịn trong Event-B, chúng tôi đề xuất mô hình tổng quát để đặc tả ràng buộc về thứ tự giữa các tiến trình tương tranh như trong Hình 3.1. Trong đó, mô hình khởi tạo biểu diễn một máy trừu tượng với các sự kiện được thực hiện tương tranh nhau. Mô hình làm mịn biểu diễn giải pháp cho sự thực hiện tương tranh của các sự kiện. Mỗi tiến trình trong chương trình tương tranh được biểu diễn bằng một sự kiện.



Hình 3.1 – Kiến trúc tổng quát của đặc tả tương tranh với Event-B.

3.2.1 Vùng xung đột

Trong mô hình khởi tạo của vấn đề này, mỗi tiến trình được biểu diễn bởi một sự kiện tương ứng. Giả sử các điều kiện $G1 \cap G2 \cap \ldots \cap Gn \neq \varnothing$ để các sự kiện được thực hiện tương tranh nhau. Không mất tính tổng quát, chúng tôi giả thiết thứ tự thực hiện của các sự kiện tuân theo giao thức $\Gamma \mathrel{\widehat{=}} [init, ec_1, ec_2, ec_3, ..., ec_n]$. Trong mô hình làm mịn của nó, chúng tôi sử dụng kỹ thuật đồng bộ hóa với biến semaphore *turn* để điều khiển sự thực thi của các sự kiện theo giao thức Γ.

3.2.2 Cung cấp và tiêu thụ

Mô hình khởi tạo của vấn đề cung cấp-tiêu thụ được biểu diễn như tương tự như vấn đề vùng xung đột, trong đó các điều kiện $G1 \cap G2 \neq \varnothing$ để bảo đảm các sự kiện của producer và consumer có thể được thực hiện tương tranh. Tại thời điểm ban đầu, sự kiện khởi tạo *init* sẽ được thực hiện để đồng thời kích hoạt các sự kiện *producer* và *consumer*, khi hàng đợi đã đầy sự kiện *producer* sẽ kích hoạt sự kiện *close* theo giao thức $\Gamma \mathrel{\widehat{=}} [init, producer \mid\mid consumer, close]$. Trong mô hình làm mịn của nó, chúng tôi sử dụng cơ chế đồng bộ hóa với biến semaphore là *Count* để kiểm tra các trạng thái đầy và rỗng của hàng đợi.

3.2.3 Vấn đề đọc-ghi

Mô hình khởi tạo của vấn đề này tương tự như mô hình khởi tạo của bài toán cung cấp-tiêu thụ và vùng xung đột.

Bảng 3.1 – Thực nghiệm đặc tả ràng buộc thứ tự giữa các tiến trình tương tranh với RODIN

Ràng buộc	Số sự kiện	Số Mệnh đề cần chứng minh	Số mệnh đề đã chứng minh	Số mệnh đề Còn lại
Vùng xung đột	4	10	10	0
Cung cấp-tiêu thụ	10	52	36	16
Đọc-ghi	10	14	14	0

Bảng 3.2 – Sinh mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Producer

$Fornt \in 0..Queue\_Size$ $x \in dom(Queue)$ g=TRUE ⊢ $Front{+}1 \in 0..Queue\_Size$	Producer/inv2/INV

Trong mô hình làm mịn của nó chúng tôi bổ sung các biến *readers* biểu diễn số tiến trình reader, biến *writers* biểu diễn số tiến trình writer. Biến điều kiện *OKtoRead* được sử dụng để khóa các tiến trình đọc reader cho đến khi điều kiện của nó được thỏa mãn cho phép đọc. Tương tự với biến điều kiện *OKtoWrite* được sử dụng để khóa các tiến trình writer cho đến khi điều kiện của nó được thỏa mãn cho phép ghi write.

3.2.4 Kết quả chứng minh

Chúng tôi đã cài đặt và đặc tả các vấn đề vùng xung đột, cung cấp-tiêu thụ và đọc-ghi bằng công cụ RODIN của Event-B. Bảng 3.1 mô tả việc sinh và chứng minh tự động các mệnh đề cần chứng minh bằng bộ chứng minh của RODIN. Trong đó, số mệnh đề cần chứng minh được sinh ra tự động để bảo đảm tính đúng đắn của đặc tả, một số mệnh đề đã được chứng minh tự động. Các mệnh đề còn lại được chứng minh bằng cách làm mịn mô hình hoặc chứng minh thủ công.
Với ràng buộc cung cấp-tiêu thụ thì các mệnh đề còn lại có thể được chứng minh tự động bằng cách làm mịn mô hình hoặc chứng minh thủ công bởi người sử dụng. Bảng 3.2 mô tả một mệnh đề chưa được chứng minh tự động bằng công cụ RODIN, với mệnh đề này được chúng tôi chứng minh bằng cách bổ sung thêm điều kiện $Front < Queue\_Size$. Các mệnh đề còn lại được chứng minh tương tự.

Chương 4

Sự đồng thuận của hệ thống đa thành phần

4.1 Giới thiệu

Để bảo đảm tính đúng đắn của hệ thống đa thành phần từ mức thiết kế đến cài đặt mã chương trình. Trong chương này, chúng tôi đề xuất phương pháp kiểm chứng sự đồng thuận của hệ thống tại mức thiết kế sử dụng Event-B và tại mức mã nguồn Java (*bytecode*) sử dụng JPF (*Java PathFinder*).

4.2 Một số định nghĩa và bổ đề

Định nghĩa 4.1 (Sự kiện hội tụ)**.** $\hat{e} \mathrel{\widehat{=}} \{e = \langle g, a\rangle \mid [a]_{g \rightsquigarrow false}\}$

Một sự kiện lặp (*iterative event*) $e = \langle g, a\rangle$ được gọi là hội tụ (*dừng*) khi và chỉ khi điều kiện g được thỏa mãn và tập các hành động a của nó được thực hiện đến khi điều kiện g không còn thỏa mãn sau một số hữu hạn bước thực hiện.

Bổ đề 4.2 (Sự kiện lấy giá trị hội tụ)**.** *Nếu* $\hat{e} = \langle g, a\rangle$ *thì* $\exists\, e' = \langle g', a'\rangle$ *sao cho* $g \vee g' = true$

Bổ đề 4.2 cho biết khi một sự kiện lặp dừng thì giá trị trả về của nó có thể nhận được bằng cách bổ sung thêm một sự kiện mới.

Chứng minh. Giả sử $e = \langle g, a\rangle$ là một sự kiện hội tụ. Khi đó theo Định nghĩa 4.1 thì điều kiện g sẽ không còn thỏa mãn sau một số hữu hạn lần thực hiện các hành động a. Do đó, ta có thể định nghĩa một sự kiện mới $e' = \langle g', a'\rangle$ với điều kiện $g' = \neg\ g$ để nhận giá trị của trả về của sự kiện hội tụ e.□

Định nghĩa 4.3 (Hệ thống đa thành phần)**.** *Một hệ thống đa thành phần (*multi-component system-MCS*) là một bộ bốn* $Mult = \langle Co, Mact, \alpha, \Gamma \rangle$ *Trong đó :*

- Co *: tập hữu hạn các thành phần,*
- $Mact$ *: tập hữu hạn các chức năng có thể trong* $Mult$*,*
- $\alpha : Mact \rightarrow Co$ *hàm gán mỗi chức năng của* $Mact$ *mà thành phần thực hiện hành vi đó,*
- Γ *: giao thức thực hiện của các thành phần để thực hiện một công việc.*

Định nghĩa 4.4 (Sự đồng thuận của hệ thống đa thành phần)**.** *Giả sử* $\hat{E} \mathrel{\widehat{=}} \{S(e_i = \langle g_i, a_i \rangle) \mid i \in [1..n]\}$ *biểu diễn thứ tự thực hiện của các sự kiện trong một hệ thống đa thành phần* M *với giao thức tương tác* Γ*.* M *được gọi là đồng thuận khi và chỉ khi :*

1. $\hat{E} \vdash \Gamma$ *: thứ tự thực hiện của các sự kiện tuân thủ giao thức,*
2. $[S(e_i)]_{\bigvee g_i \rightsquigarrow false}$ *: tuyển các mệnh đề điều kiện của tất cả các sự kiện trong giao thức không còn thỏa mãn sau một số hữu hạn lần thực hiện.*

Khi phép tuyển các điều kiện của tất cả các sự kiện không thỏa mãn thì hệ thống bị tắc nghẽn. Do đó, chúng tôi đưa vào sự kiện mới $e' = \langle g', a' \rangle$ sao cho $g_1 \vee g_2 \vee \ldots g_n \vee g'$ được thỏa mãn.

Bổ đề 4.5 (Sự kiện lấy giá trị đồng thuận)**.** *Nếu* $\hat{E} = S(e_i)$ *thì* $\exists\, e' = \langle g', a' \rangle$ *sao cho* $\bigvee g_i \vee g' = true$

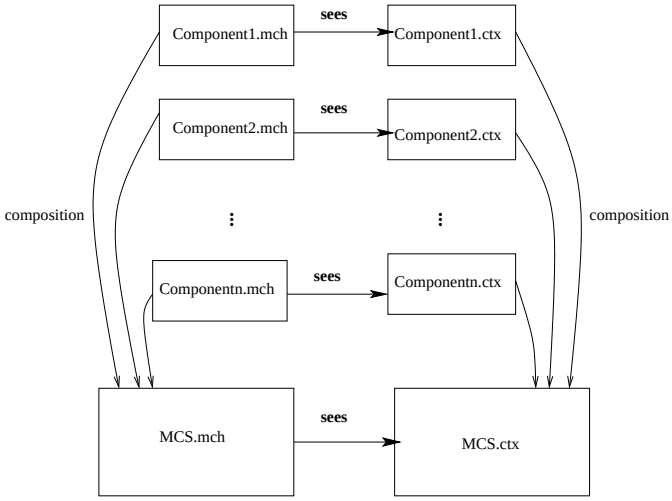
Bổ đề 4.5 cho biết khi sự tương tác của các sự kiện là đồng thuận thì chúng ta có thể nhận được giá trị trả về của nó bằng cách bổ sung thêm sự kiện mới. Việc chứng minh Bổ đề 4.5 tương tự như chứng minh Bổ đề 4.2.

4.3 Phương pháp đặc tả và kiểm chứng bản thiết kế sự đồng thuận của hệ thống đa thành phần

4.3.1 Đặc tả kiến trúc hệ thống

Chúng tôi xây dựng đặc tả kiến trúc hệ thống với Event-B trong Hình 4.1. Trong đó, ngữ cảnh và máy trừu tượng của các thành phần khác nhau sẽ

được kết hợp thành ngữ cảnh và máy trừu tượng duy nhất của hệ thống gọi là `MCS.ctx` và `MCS.mch`. Giả sử $M = \langle V, Init, ec, ee, ee_M \rangle$ là một máy kết hợp



Hình 4.1 – Sự kết hợp của máy trừu tượng và ngữ cảnh.

biểu diễn khả năng của các thành phần M_i, $i = 1, \ldots, n$. Phụ thuộc vào sự tương tác giữa các thành phần chỉ chứa các sự kiện tuần tự hoặc song song chúng tôi xây dựng các thành phần cho máy kết hợp trong Mục 4.3.2 và Mục 4.3.3.

4.3.2 Giao thức tuần tự

Giả sử thứ tự thực hiện của các sự kiện là $\Gamma \widehat{=} [ec_1, ec_2, ec_3, ..., ec_n]$ và đề xuất các nguyên lý để xây dựng máy kết hợp `M` trong Event-B như sau :

1. $V = \cup v_i$: danh sách các biến của máy kết hợp bao gồm các biến của các máy thành phần,
2. $ec = \cup ec_i$: máy kết hợp gồm tất cả các sự kiện của máy thành phần,
3. $Init = Init_1$: sự kiện khởi tạo $Init$ của máy kết hợp được định nghĩa là sự kiện khởi tạo $Init$ của máy thành phần đầu tiên trong giao thức,
4. $ee = \cup ee_i$: máy kết hợp bao gồm tất cả các sự kiện lấy kết quả trả về của máy thành phần,
5. ee_M là một sự kiện mới được bổ sung vào để lấy kết quả cuối cùng của quá trình tính toán trong mô hình kết hợp.

4.3.3 Giao thức song song

Giao thức song song Γ được hình thức hóa như sau :

Γ	::=	scenario	
(1)		e	event
(2)	|	$\Gamma\,;\, e$	sequence
(3)	|	$\Gamma \| e$	parallel

Khi đó máy kết hợp M được xây dựng theo các nguyên lý sau.

1. Từ sự kiện tuần tự được thực hiện trước đó, kích hoạt điều kiện của tất cả các sự kiện song song để cho các sự kiện này được thực hiện tại cùng một thời điểm,
2. Với mỗi sự kiện ee_i được thực hiện song song. Do các sự kiện này là hội tụ nên chúng tôi bổ sung một sự kiện ee_{is} để lấy kết quả trả về,
3. Bổ sung một sự kiện ee_P để nhận kết quả cuối cùng của tiến trình song song, sự kiện này sẽ được kích hoạt bởi sự kiện ee_{is},
4. Sự kiện nhận kết quả ee_P có nhiệm vụ kích hoạt sự kiện tuần tự tiếp theo trong giao thức.

4.3.4 Hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân

4.3.4.1 Mô tả hệ thống

Giả sử chúng ta cần đặc tả một hệ thống đa thành phần thực hiện các phép toán nhân hai số nhị phân dựa vào các phép toán nhân số nhị phân với một bit, dịch trái và phép cộng hai số nhị phân.

4.3.4.2 Đặc tả hệ thống với Event-B

Thuật toán 4.1 Nhân hai số nhị phân

ar ← Multiplication2BinaryNumber(aa,bb)

1: **for** each $ii \leq size\_bb$ **do**
2: $modr \leftarrow multiplyWithOneDigit(bb[ii], aa)$
3: $slr \leftarrow shiftLeft(modr, ii)$
4: $cc \leftarrow addition(slr, cc)$
5: **end for**
6: $ar \leftarrow cc$

Dựa vào Thuật toán 4.1 và các nguyên lý được đề xuất trong Mục 4.3.2, chúng tôi đã xây dựng máy kết hợp từ các máy thành phần và chứng minh sự đồng thuận của hệ thống này.

Bảng 4.1 – Thực nghiệm đặc tả sự đồng thuận của hệ thống đa thành phần với RODIN

Thành phần	Số sự kiện	Số Mệnh đề cần chứng minh	Số mệnh đề đã chứng minh	Số mệnh đề Còn lại
Bitshiftt	4	9	6	3
MultiDigit	3	7	3	4
Sum	4	11	4	7
MSC	10	32	15	17

4.3.4.3 Kết quả chứng minh

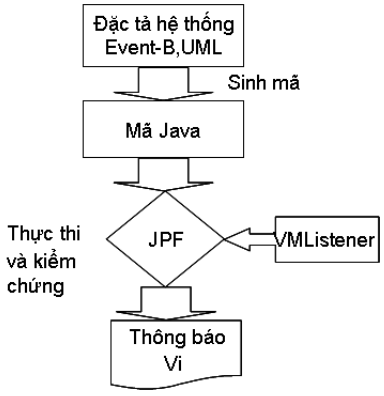
Hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân được đặc tả bằng công cụ RODIN của Event-B. Bảng 4.1 thống kê kết quả của việc sinh và chứng minh tự động các mệnh đề cần chứng minh. Trong đó, số mệnh đề cần chứng minh được sinh ra tự động để bảo đảm tính đúng đắn của đặc tả, một số mệnh đề đã được chứng minh tự động.

4.4 Phương pháp kiểm chứng sự đồng thuận của hệ thống đa thành phần tại mức mã nguồn

4.4.1 Mô tả phương pháp

Phương pháp kiểm chứng sự đồng thuận của hệ thống đa thành phần tại mức mã nguồn được mô tả như sau (Hình 4.2).

– Bản thiết kế hệ thống được đặc tả bằng các biểu đồ UML hoặc Event-B,
– Người lập trình cài đặt (*sinh mã*) Java dựa trên các đặc tả hệ thống,
– Sinh mã cho giao diện của lớp `VMListener` trong JPF để kiểm chứng sự tuân thủ của hệ thống đa thành phần so với đặc tả của nó.



Hình 4.2 – Phương pháp kiểm chứng sự đồng thuận tại mức mã nguồn.

4.4.2 Sinh mã kiểm chứng trong JPF

Chúng tôi xây dựng thuật toán để sinh mã kiểm tra cho lớp khuôn mẫu *VMListener* trong JPF như sau.

Thuật toán 4.2 Sinh mã cho lớp VMListener trong JPF

Require: Đặc tả hệ thống đa thành phần với giao thức Γ
Ensure: Mã kiểm chứng cho *VMListener* trong JPF
  Khởi tạo biến trang thái khởi tạo và kết thúc *St_Start*, *St_Final*
  State = *St_Start* { Biến State kiểm tra thứ tự thực hiện theo Γ }
  while *State* $\neq$ *St_Final* **do**
    for mỗi phương thức khởi tạo m thuộc giao thức Γ **do**
      if *State*! = *St_Start* **then**
        dừng và thông báo vi phạm đặc tả giao thức
      end if
      State = *St_m*
    end for
    for mỗi phương thức n có thứ tự thực hiện liền sau các phương thức trong Γ' thuộc giao thức Γ **do**
      if *State* == *St_Final* **then**
        return Sự đồng thuận của hệ thống
      else if *State* $\notin S(\Gamma')$ **then**
        $\{S(\Gamma')$ tập các trạng thái của các phương thức liền trước trước phương thức $n\}$
        dừng và thông báo vi phạm đặc tả giao thức
      end if
      State = *St_n*
      {Sau khi thực xong phương thức n thì chuyển sang trạng thái nó}
    end for
  end while

4.4.3 Hệ thống cung cấp tiêu thụ

Vấn đề cung cấp-tiêu thụ với giao thức song song $\Gamma \mathrel{\widehat{=}} [init, producer \mid\mid consumer, close]$ được đặc tả trong Chương 3. Dựa vào thuật toán sinh mã và giao diện của lớp *VMListener*, chúng tôi xây dựng mã để kiểm chứng sự tuân thủ giữa bản cài đặt chương trình và đặc tả của nó. Chúng tôi đã thử nghiệm với các chương trình Java được cài đặt đúng tuân thủ theo giao thức Γ và sai không tuân thủ theo giao thức. Kết quả cho thấy phương pháp này đã phát hiện được các vi phạm của chương trình so với đặc tả thiết kế của nó.

Chương 5

Sự tuân thủ giữa thực thi và đặc tả giao thức tương tác

5.1 Giới thiệu

Trong chương này, chúng tôi đề xuất một cách tiếp cận kiểm chứng động sự tương tác giữa các thành phần trong chương trình tương tranh sử dụng lập trình hướng khía cạnh. Các vi phạm được phát hiện trong bước kiểm thử, tại thời điểm thực thi chương trình.

5.2 Phương pháp đặc tả và kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác

5.2.1 Mô tả phương pháp

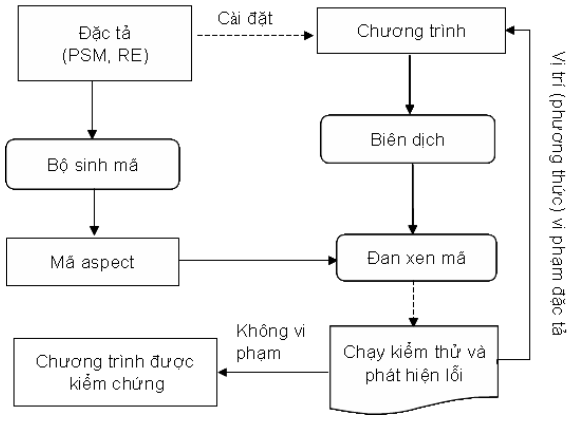
Chúng tôi đề xuất phương pháp kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác trong các chương trình tương tranh như sau (Hình 5.1).

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

5.2.2 Đặc tả giao thức tương tác

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

Định nghĩa 5.1 (Biểu thức chính quy mở rộng)**.** *Regular Expression - RE là một bộ năm* $RE =< M, O, S, Pre, Post >$*. Trong đó,* $M =$



HÌNH 5.1 – Sơ đồ hoạt động của hệ thống.

$\{m_1, m_2, ..., m_n\}$ là bảng chữ cái Sigma gồm một tập hữu hạn các phương thức, $O = \{o_1, o_2, ..., o_p\}$ là tập hữu hạn các đối tượng, Pre, Post là tập hữu hạn các tiền và hậu điều kiện, $S = \{s_1, s_2, ..., s_p\}$ là tập hạn các biểu thức biểu diễn các phương thức. $s \triangleq [Pre]o.m[Post] \mid s \rightarrow s \mid s \mid s \mid s \parallel s \mid s^{\cdot} \mid s^{+} \mid (s)$ Với $m \in M, s \in S$ và $o \in O$. $s \rightarrow s$ là sự kết hợp của hai hoặc nhiều biểu thức tuần tự, $s \mid s$ phép hoặc, $s \parallel s$ phép song song, $s^{\cdot}$ không hoặc lặp lại nhiều lần, s^{+} một hoặc lặp lại nhiều lần, (s) biểu thức kết hợp.

5.2.2.2 Biểu đồ PSM cho biểu diễn giao thức tương tác

Định nghĩa 5.2 (Máy trạng thái giao thức)**.** *Protocol State Machine - PSM là một bộ bẩy thành phần $PSM =< S, \delta, M, Pre, Post, s_0, f >$. 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. $\delta \subset S \times Pre \times M \times Post \rightarrow S$ là hàm chuyển trạng thái. $s_0, f \in S$ lần lượt là các trạng thái đầu và kết thúc.*

5.2.3 Sinh mã aspect

Với đặc tả dạng PSM chúng tôi sinh ra đồ thị có hướng để biểu diễn IPC bằng thuật toán trong Thuật toán 5.1. 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 $a \in \Sigma$ của biểu thức RE chuẩn. Từ dạng RE chuẩn chúng tôi chuyển sang máy trạng thái hữu hạn mã aspect sau đó được sinh ra tự động từ đặc tả này.

Thuật toán 5.1 Sinh đồ thị biểu diễn IPC từ đặc tả PSM

Require: đặc tả PSM
Ensure: Đồ thị G = <V, M> đặc tả giao thức.
1. Tạo hàm song ánh $\mu\, M \rightarrow \{1.. \mid M \mid\}$, $\mid M \mid$ 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 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 từ $\mu(m)$ tới $\mu(m')$ và gán nhãn là $\{pre_{m'}\}m'\{post_{m'}\}$.
4. Tạo một đỉnh kết thúc, với mỗi $m \rightarrow \odot$ thuộc đỉnh kết thúc trong PSM, tạo một cung từ $\mu(m)$ tới đỉnh kết thúc vừa tạo.

5.2.4 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. 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.

5.3 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. Kết quả thực nghiệm cho thấy :

1. 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,
2. Các aspect không làm thay đổi hành vi của chương trình gốc ngoại trừ thời gian chạy và kích thước của chương trình,
3. 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 IPC,
4. 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.

Chương 6

Ràng buộc thời gian giữa các thành phần trong chương trình tương tranh

6.1 Giới thiệu

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. Chương này chúng tôi đề xuất một phương pháp kiểm chứng sự tuân thủ về ràng buộc thời gian giữa các thành phần phần tương tranh so với đặc tả sử dụng lập trình hướng khía cạnh.

6.2 Phương pháp đặc tả và kiểm chứng ràng buộc thời gian

6.2.1 Mô tả phương pháp

Chúng tôi đề xuất phương pháp kiểm chứng sự tuân thủ về ràng buộc thời gian trong các ứng dụng phần mềm như sau (Hình 5.1, Chương 5).

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.

6.2.2 Đặc tả ràng buộc thời gian

Định nghĩa 6.1 (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* $a < b$.

Định nghĩa 6.2 (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. 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* $\theta \in N$.

Định nghĩa 6.3 (Ràng buộc thời gian giữa các thành phần tương tranh)**.** *Giả sử* $\tau(\alpha_1), \tau(\alpha_2), \ldots, \tau(\alpha_n)$ *là thời gian thực thi tương ứng của n thành phần tương tranh* $\alpha_1, \alpha_2, \ldots, \alpha_n$*. 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* $i, j = 1..n, i \neq j$.

6.2.2.1 Biểu thức chính quy thời gian

Định nghĩa 6.4 (Biểu thức chính quy thời gian)**.** *TRE là một bộ ba* $TRE = <C, M, S>$*. Trong đó,* $C = \{c_1, c_2, \ldots, c_n\}$ *là tập hữu hạn các thành phần,* $M = \{m_1, m_2, \ldots, m_m\}$ *là tập hữu hạn các phương thức,* $S = \{s_1, s_2, \ldots, s_k\}$ *là tập hữu hạn các biểu thức biểu diễn mối liên hệ giữa các thành phần được định nghĩa như sau* $s \triangleq c.m[a, b] \mid s \circ s \mid s \parallel s \mid s^{\cdot} \mid s^{+}$*. 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;* $s_i \circ s_j$ *: 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) ;* $s^{\cdot}$ *: không hoặc lặp lại nhiều lần ;* s^{+} *: một hoặc lặp lại nhiều lần.*

6.2.2.2 Biểu đồ thời gian

Định nghĩa 6.5 (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.*

6.2.3 Sinh mã aspect

Chúng tôi định nghĩa một mẫu để biểu diễn các aspect được sinh ra từ các đặc tả ràng buộc thời gian như trong Hình 6.1. 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. Aspect sinh ra sẽ được tự động đan xen với với các chương trình để kiểm chứng sự tuân thủ về ràng buộc thời gian giữa các thành phần.

```
import org.aspectj.lang.joinpoint;
variables
Variables are declared here;
...
aspect AspectName{
before() : (execution(* *.*(..))) && !within(AspectName){
 1. st = 0;
 2. Get τ1; // the current system time;
}
after() : (execution(* *.*(..)))&& !within(AspectName){
 3. Get τ2; // 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. τ = τ2−τ1; // Calculate the execution time of the method;
 7. st+ = τ; // the execution time of sequential method;
 8.if (ξ(τ, r1, r2, ...) = false)// Timing constraint conditions;
 9. Produce violation reports;
}
```

Hình 6.1 – Sinh mã aspect từ các đặc tả ràng buộc thời gian.

6.3 Thực nghiệm

Chúng tôi đã cài đặt phương pháp này và tích hợp với bộ công cụ kiểm chứng PVG *(Chương 5)*. Đầu vào của công cụ PVG 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. Qua các kết quả thực nghiệm cho thấy phương pháp được đề xuất đã phát hiện được các vi phạm ràng buộc thời gian giữa các thành phần. Chúng tôi đã cài đặt phương pháp này thành một công cụ kiểm chứng và chạy thử nghiệm với một số ứng dụng hướng đối tượng viết trên ngôn ngữ lập trình Java. Kết quả thử nghiệm cho thấy phương pháp được đề xuất đã phát hiện được vi phạm ràng buộc thời gian của các thành phần so với đặc tả.

Chương 7

Kết luận

7.1 Các đóng góp của luận án

Luận án này đã đóng góp hai kết quả chính trong việc kiểm chứng các chương trình Java tương tranh, kết quả cụ thể như sau.

1. Đề xuất các phương pháp kiểm chứng tính đúng đắn của bản thiết kế các chương trình tương tranh sử dụng Event-B,
2. Đề xuất các phương pháp kiểm chứng sự tuân thủ giữa bản cài đặt chương trình so với mô hình thiết kế của nó sử dụng AOP và JPF.

Các kết quả cho thấy các đề xuất của chúng tôi có ý nghĩa trong việc bổ sung và hoàn thiện các phương pháp đặc tả và kiểm chứng phần mềm từ pha thiết kế đến cài đặt.

7.2 Hướng phát triển

Tiếp tục phát triển và đề xuất các phương pháp kiểm chứng với Event-B. Cài đặt công cụ hỗ trợ đặc tả song song và plugin vào bộ công cụ kiểm chứng mã nguồn mở RODIN, tự động sinh mã Java từ đặc tả bằng Event-B. Hoàn thiện môi trường kiểm chứng dựa trên lập trình hướng khía cạnh để kiểm chứng sự tuân thủ giữa thiết kế với cài đặt mã nguồn chương trình.

Danh mục các công trình khoa học đã công bố

Tạp chí

1. Trịnh Thanh Bình, Trương Ninh Thuận và Nguyễn Việt Hà. Kiểm chứng sự tuân thủ về ràng buộc thời gian trong các ứng dụng phần mềm, *Tạp chí Tin học và Điều khiển học*, Vol.26, No.2, pp.173–184, 2010.
2. Trịnh Thanh Bình, Trương Anh Hoàng và Nguyễn Việt Hà.Kiểm chứng sự tương tác giữa các thành phần trong chương trình đa luồng sử dụng lập trình hướng khía cạnh. *Chuyên san Các công trình nghiên cứu, phát triển và ứng dụng CNTT-TT, Tạp chí Công nghệ thông tin & Truyền thông*, Vol.24, No.4(24), pp.36-45, 2010.
3. Trinh Thanh Binh, Truong Anh Hoang, Nguyen Viet Ha. A Dynamic Birthmark to Detect the Theft of Java Programs, *Tạp chí Khoa học Tự nhiên và Công nghệ*, Đại học Quốc gia Hà Nội, Vol. 24, No. 3S, pp. 123-130, 2008.

Hội nghị

1. Thanh-Binh Trinh, Ninh-Thuan Truong, and Viet-Ha Nguyen. Refining undetermined events for specifying concurrent programs, *3nd Intern. Conf. on Knowledge and Systems Engineering (KSE 2011)*, Hanoi, Vietnam, 2011.
2. Thanh-Binh Trinh, Quang-Thap Pham, Ninh-Thuan Truong, and Viet-Ha Nguyen. A Runtime Approach to Verify Scenario in Multi-agent Systems, *2nd Intern. Conf. on Knowledge and Systems Engineering (KSE 2010)*, pp. 161-166, Hanoi, Vietnam, Oct 8-9, 2010.
3. Trinh Thanh Binh, Truong Anh Hoang, Nguyen Viet Ha. Checking Protocol-Conformance in Component Models using Aspect Oriented Programming, *IASTED Intern. Conf. on Advances in Computer Science and Engineering*, pp. 150-155, Phuket, Thailand, March 16-18, 2009.
4. 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)*, pp. 220-225, Hanoi, Vietnam, Oct 14-15, 2009.
5. Ninh-Thuan Truong, Thanh-Binh Trinh, and 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)*, pp. 201-209, Hanoi, Vietnam, 23-27 November 2009.

6. Hoang Truong, Thanh-Binh Trinh, Viet-Ha Nguyen, Trang Nguyen Thi Thu, Hung Dang Van and Hung Pham Dinh. Specifying and checking interface protocols using aspect-oriented programming, *6th IEEE Intern. Conf. on Software Engineering and Formal Method (SEFM 2008)*, pp. 382-386, Cape Town, South Africa, 10-14 November 2008.