ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Trịnh Thanh Bình

KIỂM CHỨNG CÁC THÀNH PHẦN JAVA TƯƠNG TRANH

LUẬN ÁN TIẾN SỸ CÔNG NGHỆ THÔNG TIN

Hà Nội - 2011

Lời cam đoan

Tôi xin cam đoan luận án "**Kiểm chứng các thành phần Java tương tranh**" là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả được trình bày trong luận án là hoàn toàn trung thực và chưa từng được công bố trong bất kỳ một công trình nào khác.

- Tôi đã trích dẫn đầy đủ các tài liệu tham khảo, công trình nghiên cứu liên quan ở trong nước và quốc tế. Ngoại trừ các tài liệu tham khảo này, luận án hoàn toàn là công việc của riêng tôi.
- Trong các công trình khoa học được công bố trong luận án, tôi đã thể hiện rõ ràng và chính xác đóng góp của các đồng tác giả và những gì do tôi đã đóng góp.
- Luận án được hoàn thành trong thời gian tôi làm Nghiên cứu sinh tại Bộ môn Công nghệ phần mềm, Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội.

Tác giả : _____

Hà Nội : _____

Lời cảm ơn

Trước hết, tôi muốn cảm ơn đến PGS.TS Nguyễn Việt Hà, cán bộ hướng dẫn chính, người đã trực tiếp giảng dạy và hướng dẫn tôi trong suốt thời gian học cao học, thực hiện luận văn thạc sĩ và luận án này. Một vinh dự lớn cho tôi được học tập, nghiên cứu dưới sự hướng dẫn của thầy. Tôi cũng trân trọng cảm ơn PGS.TS Nguyễn Ngọc Bình vì sự hướng dẫn tận tình và khoa học của Thầy.

Tôi gửi lời cám ơn đến TS Trương Ninh Thuận, TS Trương Anh Hoàng vì sự giúp đỡ của các Thầy cho các đề xuất và các trao đổi trong nghiên cứu của tôi.

Tôi bày tỏ sự cảm ơn đến TS Đặng Văn Hưng, TS Phạm Ngọc Hùng, TS Võ Đình Hiếu, TS Đặng Đức Hạnh, Ths Tô Văn Khánh, Ths Vũ Quang Dũng vì sự giúp đỡ của các Thầy về các đóng góp rất hữu ích cho luận án.

Tôi trân trọng cảm ơn Bộ môn Công nghệ phần mềm, Khoa Công nghệ thông tin, Phòng Đào tạo và Ban giám hiệu trường Đại học Công nghệ đã tạo điều kiện thuận lợi cho tôi trong suốt quá trình thực hiện luận án.

Tôi cũng bày tỏ sự cảm ơn đến Ban giám hiệu Trường Đại học Hải Phòng đã tạo điều kiện về thời gian và tài chính cho tôi thực hiện luận án này. Tôi muốn cảm ơn đến các cán bộ, giảng viên trong Khoa Toán tin – Trường Đại học Hải Phòng đã cổ vũ động viên và sát cánh bên tôi trong quá trình nghiên cứu.

Một phần của nghiên cứu này được thực hiện trong khuôn khổ đề tài nghiên cứu khoa học QGTD 09.02 của Đại học Quốc gia Hà Nội. Xin cảm ơn các trao đổi và trợ giúp của các thành viên đề tài.

Tôi muốn cảm ơn đến tất cả những người bạn của tôi. Những người luôn chia sẻ, động viên tôi trong những lúc khó khăn và tôi luôn ghi nhớ điều đó. Cuối cùng, tôi xin bày tỏ lòng biết ơn vô hạn đối với cha mẹ, vợ con và gia đình đã luôn ủng hộ, giúp đỡ tôi.

Mục lục

Lời cam đoan i

Lời cảm ơn ii

Từ viết tắt vii

Danh mục các hình vẽ viii

Danh mục các bảng biểu x

1 Mở đầu 1
- 1.1 Bối cảnh . . . 1
- 1.2 Một số nghiên cứu liên quan . . . 3
  - 1.2.1 Kiểm chứng thiết kế . . . 3
  - 1.2.2 Kiểm chứng mã nguồn . . . 4
- 1.3 Nội dung nghiên cứu . . . 5
- 1.4 Cấu trúc luận án . . . 7

2 Kiến thức cơ sở 9
- 2.1 Kiểm chứng phần mềm . . . 9
  - 2.1.1 Kiểm chứng hình thức . . . 9
    - 2.1.1.1 Kiểm chứng mô hình . . . 9
    - 2.1.1.2 Chứng minh định lý . . . 10
  - 2.1.2 Kiểm chứng tại thời điểm thực thi . . . 11
- 2.2 Một số vấn đề trong chương trình tương tranh . . . 11
- 2.3 Sự tương tranh trong Java . . . 12
  - 2.3.1 Mô hình lưu trữ (JMM-Java Memory Model) . . . 13
  - 2.3.2 Ngôn ngữ mô hình hóa cho Java (JML-Java Modeling Language) . . . 14
  - 2.3.3 Công cụ kiểm chứng mã Java (JPF-Java PathFinder) . . . 15
- 2.4 Phương pháp hình thức với Event-B . . . 17
  - 2.4.1 Máy và Ngữ cảnh . . . 17
  - 2.4.2 Sự kiện . . . 17

2.4.3 Phân rã và kết hợp . . . 19
2.4.4 Sinh mệnh đề cần chứng minh . . . 20
2.5 Ngôn ngữ mô hình hóa UML . . . 21
2.5.1 Biểu đồ tuần tự . . . 21
2.5.2 Máy trạng thái giao thức . . . 22
2.5.3 Biểu đồ thời gian . . . 23
2.6 Lập trình hướng khía cạnh . . . 25
2.6.1 Thực thi cắt ngang . . . 26
2.6.2 Điểm nối . . . 27
2.6.3 Hướng cắt . . . 27
2.6.4 Mã hành vi . . . 28
2.6.5 Khía cạnh . . . 29
2.7 Kết luận . . . 30

3 Ràng buộc thứ tự giữa các tiến trình tương tranh 31
3.1 Giới thiệu . . . 31
3.2 Đặc tả và kiểm chứng ràng buộc thứ tự giữa các tiến trình tương tranh . . . 33
3.2.1 Mô tả phương pháp . . . 33
3.2.2 Vùng xung đột . . . 34
3.2.3 Cung cấp và tiêu thụ . . . 36
3.2.4 Vấn đề đọc-ghi . . . 41
3.2.5 Kết quả chứng minh . . . 42
3.3 Kết luận . . . 45

4 Sự đồng thuận của hệ thống đa thành phần 46
4.1 Giới thiệu . . . 46
4.2 Một số định nghĩa và bổ đề . . . 48
4.3 Phương pháp đặc tả và kiểm chứng bản thiết kế sự đồng thuận của hệ thống đa thành phần . . . 50
4.3.1 Đặc tả kiến trúc hệ thống . . . 50
4.3.2 Giao thức tuần tự . . . 51
4.3.3 Giao thức song song . . . 53
4.3.4 Hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân . . . 54
4.3.4.1 Mô tả hệ thống . . . 54
4.3.4.2 Đặc tả hệ thống với Event-B . . . 55
4.3.4.3 Kết quả chứng minh . . . 58
4.4 Phương pháp kiểm chứng sự đồng thuận của hệ thống đa thành phần tại mức mã nguồn . . . 60
4.4.1 Mô tả phương pháp . . . 60
4.4.2 Sinh mã kiểm chứng trong JPF . . . 61
4.4.3 Hệ thống cung cấp tiêu thụ . . . 61
4.5 Kết luận . . . 62

5 Sự tuân thủ giữa thực thi và đặc tả giao thức tương tác **65**
5.1 Giới thiệu . . . 65
5.2 Bài toán kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác . . . 66
5.3 Phương pháp đặc tả và kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác . . . 67
5.3.1 Mô tả phương pháp . . . 67
5.3.2 Đặc tả giao thức tương tác . . . 67
5.3.2.1 Biểu thức chính quy mở rộng cho biểu diễn giao thức tương tác . . . 67
5.3.2.2 Biểu đồ PSM cho biểu diễn giao thức tương tác . . . 69
5.3.3 Sinh mã aspect . . . 70
5.3.4 Đan mã aspect . . . 73
5.4 Thực nghiệm . . . 73
5.5 Kết luận . . . 76

6 Ràng buộc thời gian giữa các thành phần trong chương trình tương tranh **78**
6.1 Giới thiệu . . . 78
6.2 Bài toán kiểm chứng ràng buộc thời gian giữa các thành phần tương tranh . . . 79
6.3 Phương pháp đặc tả và kiểm chứng ràng buộc thời gian . . . 80
6.3.1 Mô tả phương pháp . . . 80
6.3.2 Đặc tả ràng buộc thời gian . . . 81
6.3.2.1 Biểu thức chính quy thời gian . . . 82
6.3.2.2 Biểu đồ thời gian . . . 83
6.3.3 Sinh mã aspect . . . 84
6.4 Thực nghiệm . . . 85
6.5 Kết luận . . . 86

7 Kết luận **88**
7.1 Các đóng góp của luận án . . . 88
7.2 Hướng phát triển . . . 91

A Đặc tả ràng buộc thứ tự giữa các tiến trình tương tranh **104**
A.1 Vấn đề vùng xung đột . . . 104
A.1.1 Mô hình khởi tạo . . . 104
A.1.2 Mô hình làm mịn . . . 105
A.2 Vấn đề cung cấp tiêu thụ . . . 107
A.2.1 Mô hình khởi tạo . . . 107
A.2.2 Mô hình làm mịn . . . 108
A.3 Vấn đề đọc ghi . . . 111
A.3.1 Mô hình khởi tạo . . . 111

A.3.2 Mô hình làm mịn 112

B Đặc tả hệ thống đa thành phần thực hiện các phép toán nhị phân 116
B.1 Đặc tả phép dịch bit 116
B.1.1 Ngữ cảnh của phép dịch bit 116
B.1.2 Máy thực thi của phép dịch bit 116
B.2 Đặc tả phép nhân xâu nhị phân với một bit 118
B.2.1 Ngữ cảnh của phép nhân xâu nhị phân với một bit 118
B.2.2 Máy thực thi của phép nhân xâu nhị phân với một bit . . . 118
B.3 Đặc tả phép cộng xâu nhị phân 119
B.3.1 Ngữ cảnh của phép cộng xâu nhị phân 119
B.3.2 Máy thực thi của phép cộng hai xâu nhị phân 120
B.4 Đặc tả hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân 121
B.4.1 Ngữ cảnh của hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân 121
B.4.2 Máy thực thi của hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân 122

C Công cụ sinh mã kiểm chứng PVG 125
C.1 Giới thiệu 125
C.2 Hướng dẫn sử dụng 126
C.2.1 Các yêu cầu 126
C.2.2 Các chức năng chính 127
C.2.3 Hướng dẫn thực hiện 127
C.2.3.1 Đặc tả giao thức 127
C.2.3.2 Lưu mã Aspect 129
C.2.3.3 Đan mã aspect 129

Từ viết tắt

Dạng viết tắt	Dạng đầy đủ	Diễn giải
AOP	Aspect Oriented Programming	Lập trình hướng khía cạnh
CTL	Computation Tree Logic	Logic cây tính toán
IPC	Interaction Protocol	Giao thức tương tác
JMM	Java memory model	Mô hình lưu trữ trong Java
JML	Java modeling language	Ngôn ngữ đặc tả cho Java
JPF	Java PathFinder	Công cụ kiểm chứng mã Java
LTL	Linear Temporal Logic	Logic thời gian tuyến tính
MCS	Multi-Component System	Hệ thống đa thành phần
PSM	Protocol State Machine	Máy trạng thái giao thức
RE	Regular Expression	Biểu thức chính quy
TD	Timing Diagram	Biểu đồ thời gian
UML	Unified Modeling Language	Ngôn ngữ mô hình hóa thống nhất

Danh sách hình vẽ

1.1 Kiểm chứng mức thiết kế và cài đặt chương trình. 6
1.2 Cấu trúc luận án. 8

2.1 Kiểm chứng chương trình Java với JPF. 15
2.2 Cấu trúc tổng quát của máy và ngữ cảnh. 18
2.3 Cấu trúc tổng quát của sự kiện. 19
2.4 Sự phân rã và kết hợp. 20
2.5 Sự kiện sinh các mệnh đề chứng minh để bảo toàn bất biến. 20
2.6 Biểu đồ tuần tự biểu diễn giao thức rút tiền của hệ thống ATM. . . 22
2.7 Máy trạng thái biểu diễn giao thức tương tác truy cập cơ sở dữ liệu. 23
2.8 Dạng trạng thái của biểu đồ thời gian. 24
2.9 Dạng giá trị của biểu đồ thời gian. 25
2.10 Biểu đồ thời gian dạng kết hợp. 25

3.1 Kiến trúc tổng quát của đặc tả tương tranh với Event-B. 35
3.2 Máy truy cập vào vùng xung đột. 35
3.3 Máy được làm mịn để truy cập vào vùng xung đột. 36
3.4 Giao thức tương tác của vấn đề cung cấp tiêu thụ. 37
3.5 Máy trừu tượng cho vấn đề cung cấp-tiêu thụ. 38
3.6 Máy làm mịn thứ nhất cho vấn đề cung cấp-tiêu thụ. 39
3.7 Máy làm mịn thứ hai cho vấn đề cung cấp-tiêu thụ. 40
3.8 Máy trừu tượng cho vấn đề đọc-ghi. 41
3.9 Máy làm mịn cho vấn đề đọc-ghi. 43
3.10 Đặc tả sự kiện producer trong mô hình khởi tạo và làm mịn. 44

4.1 Sự kết hợp của máy trừu tượng và ngữ cảnh. 50
4.2 Giao thức tuần tự được biểu diễn bằng UML. 52
4.3 Giao thức song song được biểu diễn bằng UML. 53
4.4 Đặc tả phép dịch bit trong UML. 55
4.5 Máy và ngữ cảnh của hệ thống. 57
4.6 Đặc tả sự kiện ShiftLeftIf của thành phần bitshift. 59
4.7 Phương pháp kiểm chứng sự đồng thuận tại mức mã nguồn. 60
4.8 Kiểm chứng mã nguồn hệ thống cung cấp-tiêu thụ với JPF. 63

5.1 Sơ đồ hoạt động của hệ thống. 68
5.2 Ví dụ các chương trình được cài đặt đúng và sai. 74

6.1 Biểu đồ thời gian của giao thức rút tiền. 80
6.2 Sinh mã aspect từ các đặc tả ràng buộc thời gian. 84
6.3 Ví dụ ca kiểm thử đúng và sai của phương thức withdraw với ràng buộc thời gian thực thi [726082, 143658] nano giây. 85

C.1 Giao diện chính của công cụ sinh mã kiểm chứng PVG. 126
C.2 Khởi động PVG từ NetBeans. 127
C.3 Đặc tả giao thức tương tác của hàng đợi tương tranh với UML. . . 128
C.4 Đặc tả giao thức của hàng đợi tương tranh trong textbox bên trái và mã AspectJ được sinh ra bên phải. 129
C.5 Lưu mã aspect được sinh ra. 130
C.6 Đan xen mã aspect với mã Java trong Eclipse. 131

Danh sách bảng

2.1 Chứng minh định lý 10
2.2 Luật sinh mệnh đề cần chứng minh để bảo toàn bất biến 21

3.1 Kết quả chứng minh đặc tả ràng buộc thứ tự giữa các tiến trình tương tranh với RODIN 42
3.2 Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Producer đã được chứng minh tự động 44
3.3 Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Producer chưa được chứng minh tự động 45

4.1 Kết quả chứng minh sự đồng thuận của hệ thống đa thành phần với RODIN 58
4.2 Mệnh đề cần chứng minh để bảo đảm tính định nghĩa được của sự kiện BitShiftLeftIf đã được chứng minh tự động 59
4.3 Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện BitShift-LeftIf chưa được chứng minh tự động 60

5.1 Thực nghiệm kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác 75

6.1 Thực nghiệm kiểm chứng các ràng buộc thời gian 87

Chương 1

Mở đầu

1.1 Bối cảnh

Phần mềm ngày càng đóng vai trò quan trọng trong xã hội hiện đại [70, 72]. 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.

Đến nay, trong công nghiệp phần mềm đã có nhiều phương pháp khác nhau được đề xuất và phát triển để giảm lỗi phần mềm từ pha thiết kế đến cài đặt. Các phương pháp kiểm chứng như chứng minh định lý (*theorem proving*) và kiểm chứng mô hình (*model checking*) đã được ứng dụng thành công để kiểm chứng mô hình thiết kế của phần mềm [14, 19, 40]. Trong nhiều hệ thống, cài đặt thực tế thường chỉ được thực hiện sau khi mô hình thiết kế đã được kiểm chứng. Tuy nhiên, cài đặt thực mã nguồn chương trình có thể vi phạm các ràng buộc thiết kế [80]. Do đó, phần mềm có thể vẫn tồn tại lỗi mặc dù thiết kế của nó đã được kiểm chứng và thẩm định chi tiết.

Các phương pháp kiểm chứng tại thời điểm thực thi [7, 23, 50] như kiểm thử phần mềm bằng các bộ dữ liệu kiểm thử (*test suite*) thường chỉ phát hiện được các lỗi về giá trị đầu ra (*output*) nhưng không phát hiện được các lỗi vi phạm ràng buộc thiết kế. Trong khi đó, một vi phạm ràng buộc có thể gây lỗi hệ thống, đặc biệt

khi tích hợp nhiều môđun thì việc xác định chính xác vị trí gây lỗi sẽ rất khó khăn và làm chi phí sửa lỗi tăng cao.

Các phần mềm (*chương trình*) tương tranh gồm hai hoặc nhiều tiến trình cộng tác để cùng nhau thực hiện một nhiệm vụ [12]. Trong đó, mỗi tiến trình là một chương trình tuần tự thực hiện một tập các câu lệnh tuần tự. Các tiến trình thường cộng tác với nhau thông qua các biến chia sẻ (*shared variables*) hoặc cơ chế truyền thông điệp (*message passing*). Lập trình và kiểm chứng các chương trình tương tranh thường khó khăn hơn so với các chương trình tuần tự do khả năng thực hiện của các chương trình này. Ví dụ nếu chúng ta thực thi một chương trình tương tranh hai lần với cùng một đầu vào như nhau thì không thể bảo đảm nó sẽ trả về cùng một kết quả.

Đã có một vài phương pháp hình thức được đề xuất để đặc tả và kiểm chứng các chương trình tương tranh như *Petri nets* [68], *Actor model* [52], $\pi-calculus$ [66] và *CSP* [55]. Theo đó, nhiều ngôn ngữ lập trình tương tranh được xây dựng và phát triển với mục đích nghiên cứu và thử nghiệm các phương pháp được đề xuất như các ngôn ngữ ActorScript, Pict, XC. Tuy nhiên theo Yang [82] và Edmunds [42] thì hiện nay trong công nghiệp phần mềm vẫn còn thiếu các mô hình đặc tả hình thức áp dụng cho các ngôn ngữ lập trình hiện đại hỗ trợ hỗ trợ tương tranh như Java.

Các nghiên cứu gần đây [26, 42, 49, 82] trọng tâm vào kiểm chứng các vấn đề về xung đột (*interference*), tắc nghẽn (*deadlock*) trong chương trình Java tương tranh. Tuy nhiên các phương pháp này chưa kiểm chứng sự tương tác (*giao thức tương tác*) giữa các tiến trình (*thành phần*) tương tranh nhằm bảo đảm tính nhất quán của dữ liệu chia sẻ và dữ liệu đầu vào-đầu ra. Sự tương tác giữa các tiến trình được đặc tả là ràng buộc về thứ tự thực hiện của nó. Các tiến trình phải trả về kết quả mong muốn sau một số hữu hạn lần thực hiện, và thỏa mãn ràng buộc thời gian như thời điểm bắt đầu, kết thúc thực hiện của các tiến trình. Do đó, nhu cầu nghiên cứu và đề xuất các phương pháp hình thức để kiểm chứng sự tương tác giữa các tiến trình tương tranh hoàn thiện từ pha thiết kế đến cài đặt ngày càng trở lên cần thiết.

1.2 Một số nghiên cứu liên quan

Đã có một vài phương pháp, công cụ được đề xuất để đặc tả và kiểm chứng các chương trình Java tương tranh. Trong mục này chúng tôi trình bày và đánh giá một số nghiên cứu liên quan với các nội dung nghiên cứu trong luận án. Các nghiên cứu này được chia thành hai hướng kiểm chứng thiết kế và kiểm chứng mã ngồn chương trình.

1.2.1 Kiểm chứng thiết kế

Edmunds [42] 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. Ngôn ngữ đặc tả trung gian OCB sẽ che giấu các chi tiết về cơ chế khóa (*locking*) và khối (*blocking*) trong các đặc tả tương tranh và cung cấp một khung nhìn rõ ràng về tính nguyên tử của nó sử dụng các mệnh đề nguyên tử được gán nhãn (*labelled atomic clauses*). Các mệnh đề nguyên tử này được ánh xạ thành các sự kiện nguyên tử trong máy của Event-B. 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 [17] 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. Dựa vào cơ chế làm mịn dần của Event-B để đặc tả và kiểm chứng tự động sự phân rã của các biểu đồ hoạt động của UML thỏa mãn các thuộc tính như khóa chết (*deadlock*), sự công bằng (*fairness*). Đóng góp chính của nghiên cứu này là chuyển đổi từ một đặc tả trực quan sang hình thức và dựa vào công cụ của nó để 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 [15] 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.

Yang [82] đề xuất phương pháp kết hợp giữa các đặc tả hình thức với PROB [62], CSP [55] và ngôn ngữ đặc tả dựa trên trạng thái [6] để đặc tả và cài đặt các chương trình Java tương tranh. Phương pháp này xây dựng tập các luật chuyển đổi hình thức để định nghĩa sự tương ứng giữa các ngôn ngữ đặc tả B+CSP và Java/JCSPRO [82]. Các tác giả cũng xây dựng một công cụ chuyển đổi cho phép người sử dụng tự động sinh mã thực thi Java từ các đặc tả từ B+CSP trong PROB.

1.2.2 Kiểm chứng mã nguồn

J-LO (*Java Logical Observer*) [24] 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 theo Bodden [25] thì 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 [26] 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 khóa chết.

Jin [58] 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[81]. 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 tương tranh. Hơn nữa, phương pháp này cũng phải giải quyết trọn vẹn bài toán bao phủ ngôn ngữ (*language inclusion problem*).

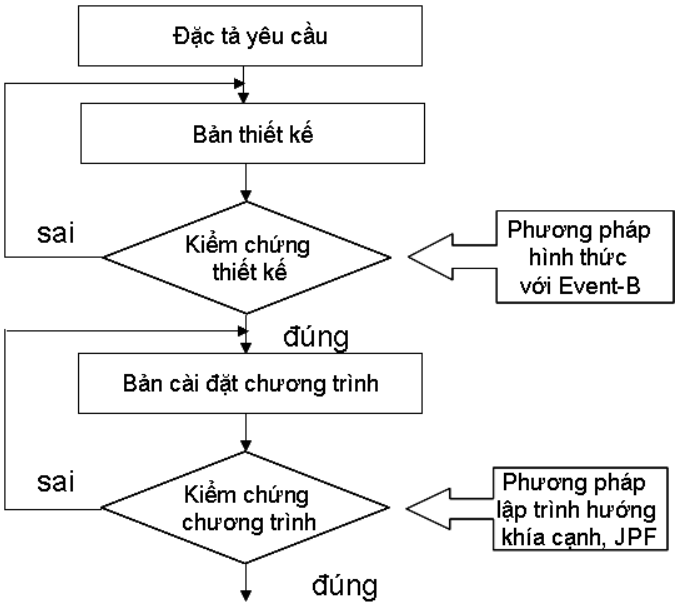
Trong các phương pháp về JML [30, 33, 34, 63], MCS phải được đặc tả dưới dạng tiền và hậu điều kiện được kết hợp với phần thân của các phương thức trong chương trình như các bất biến của vòng lặp, hay tập các câu lệnh. Các tiền và hậu điều kiện này được viết dưới một dạng chuẩn để có thể biên dịch và 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.

Trong [41] đề xuất phương pháp sử dụng biểu đồ thời gian của UML để ước lượng thời gian thực thi trong trường hợp xấu nhất của các thành phần trong hệ thống tại 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 có thể đưa ra các ước lượng thiếu chính xác do thiếu các thông tin cần thiết như kích cỡ đầu vào, môi trường thực thi,...Hơn nữa phương pháp này cũng chưa ước lượng được ràng buộc thời gian giữa các thành phần được thực hiện tương tranh với nhau.

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, luận án sử dụng phương pháp hình

thức với Event-B để kiểm chứng bản thiết kế của chương trình tương tranh nhằm phát hiện lỗi ở mức cao. Chúng tôi tập trung vào các phương pháp thiết kế định hướng đến việc cài đặt bằng Java hoặc các ngôn ngữ có tính năng tương đương. Để phát hiện lỗi ở mức thấp, chúng tôi sử dụng phương pháp lập trình hướng khía cạnh và bộ công cụ 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. Cụ thể luận án sẽ tập trung vào nghiên cứu các vấn đề sau.



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

– Sử dụng phương pháp hình thức với Event-B để đặc tả và kiểm chứng ràng buộc thứ tự giữa các tiến trình (*thành phần*) tương tranh nhằm bảo đảm tính nhất quán của đầu ra với cùng một đầu vào. Trong đó, mỗi tiến trình được biểu diễn tương ứng với một sự kiện, mỗi vấn đề được đặc tả bằng mô hình trừu tượng biểu diễn các sự kiện và mô hình làm mịn của nó đặc tả sự thực hiện tương tranh của các sự kiện. Sự thực hiện tương tranh của các sự kiện dựa trên kỹ thuật đồng bộ hóa semaphore. Tính đúng đắn của đặc tả được bảo đảm thông qua việc sinh và chứng minh tự động các mệnh đề cần chứng minh.
– Sử dụng phương pháp hình thức với Event-B để đặc tả và kiểm chứng sự đồng thuận của hệ thống đa thành phần tương tranh. Một hệ thống đa thành phần được gọi là đồng thuận nếu nó phải trả về kết quả mong muốn sau một số hữu

hạn lần thực hiện. Với bài toán này, chúng tôi sẽ đặc tả mỗi thành phần bằng một máy trừu tượng (*abstract machine*) tham chiếu đến ngữ cảnh (*context*) của nó. Các máy trừu tượng và ngữ cảnh sau đó được kết hợp với nhau theo một giao thức tương tác xác định. Sử dụng công cụ hỗ trợ của Event-B và JPF kiểm chứng tính đồng thuận của hệ thống đa thành phần tại mức thiết kế và cài đặc mã nguồn chương trình.

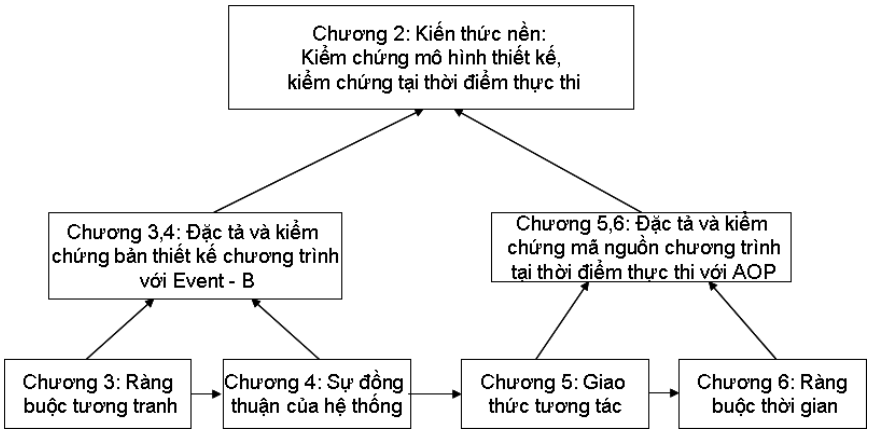
– Sử dụng phương pháp lập trình hướng khía cạnh để kiểm chứng sự tuân thủ giữa thực thi và đặc tả thứ tự thực hiện (*giao thức tương tác*) của các thành phần tương tranh, các vi phạm được phát hiện trong bước kiểm thử, tại thời điểm thực thi chương trình. Với nghiên cứu này, chúng tôi sẽ sử dụng máy 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.

– Sử dụng phương pháp lập trình hướng khía cạnh để kiểm chứng ràng buộc thời gian giữa các thành phần tương tranh. 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 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.

1.4 Cấu trúc luận án

Luận án gồm sáu chương chính được cấu trúc như trong Hình 1.2. Trong đó, Chương 2 giới thiệu một số kiến thức nền cho các đóng góp của luận án trong các chương còn lại. Theo cách tiếp cận kiểm chứng ở mức mô hình thiết kế, luận án đã để 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 được trình bày trong các Chương 3 và 4.

Theo cách tiếp cận kiểm chứng tại thời điểm thực thi, luận án đề xuất hai phương pháp sử dụng lập trình hướng khía cạnh với AOP để kiểm chứng sự tuận thủ giữa chương trình và đặc tả của nó, các kết quả được trình bày trong các Chương 5 và 6. Chương 5 trình bày phương pháp kiểm chứng sự tuân thủ giữa sự cài đặt của chương trình tương tranh so với đặc tả giao thức tương tác của nó. Chương 6 trình bày phương pháp kiểm chứng các ràng buộc thời gian giữa các thành phần tuần tự và song song trong chương trình tương tranh.



Hình 1.2 – Cấu trúc luận án.

Chương 2

Kiến thức cơ sở

2.1 Kiểm chứng phần mềm

Kiểm chứng phần mềm (*software verification*) là tập các nguyên lý, phương pháp và công cụ để bảo đảm tính đúng đắn của các sản phẩm phần mềm. Trong mục này chúng tôi giới thiệu tổng quan về hai phương pháp kiểm chứng phần mềm là các phương pháp kiểm chứng hình thức và kiểm chứng tại thời điểm thực thi chương trình.

2.1.1 Kiểm chứng hình thức

2.1.1.1 Kiểm chứng mô hình

Phương pháp 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 $M \models p$ [19]. Về mặt thực thi, kiểm chứng mô hình sẽ duyệt qua các trạng thái, các đường thực thi có thể có trong mô hình M để xác định tính khả thỏa của p. Trong đó, các thuộc tính được đặc tả bằng logic thời gian LTL hoặc CTL [19]. Mô hình M là

Bảng 2.1 – Chứng minh định lý

$P_1, P_2, ..., P_n$ $\vdash$ C	name

một cấu trúc Kripke gồm bốn thành phần $M = (S, S_0, L, R)$ với S là một tập hữu hạn các trạng thái, $S_0 \in S$ là trạng thái đầu, $R \subset S \times S$ là quan hệ chuyển trạng thái, $L : S \to 2^{AP}$ là hàm gán nhãn với AP là tập hữu hạn các mệnh đề nguyên tử được xây dựng từ hệ thống.

Một bộ kiểm chứng mô hình [16, 56] (*model checker*) sẽ kiểm tra tất cả các trạng thái có thể có của mô hình để tìm ra tất cả các đường thực thi có thể gây ra lỗi. Do đó không gian trạng thái thường là vô cùng lớn nếu không muốn nói là vô hạn. Vì vậy việc phải duyệt qua tất cả các trạng thái là bất khả thi. Để đối phó với bài toán bùng nổ không gian trạng thái đã có một vài nghiên cứu liên quan đến các kỹ thuật làm giảm không gian trạng thái như *Abstraction*, *Slicing* [35, 80].

2.1.1.2 Chứng minh định lý

Phương pháp chứng minh định lý (*theorem proving*) [20] 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 $\models F$. Một hệ chứng minh bao gồm các luật suy diễn có dạng như trong Bảng 2.1. Trong đó, P_i với $i = 1..n$ là tập các tiên đề, C là tập các định lý. Một hệ thống được gọi là đúng (*sound*) nếu tất cả các định lý của nó đều được chứng minh.

Các phương pháp chứng minh định lý như B [5], Event-B [8] đã được sử dụng thành công để đặc tả và kiểm chứng tính đúng đắn của mô hình thiết kế phần mềm. Trong Mục 2.4 chúng tôi trình bày chi tiết về một phương pháp chứng minh định lý với Event-B, phương pháp này sẽ được sử dụng để kiểm chứng tính đúng đắn của bản thiết kế các chương trình tương tranh.

2.1.2 Kiểm chứng tại thời điểm thực thi

Kiểm chứng tại thời điểm thực thi [18] (*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. Các hành vi quan sát được như bản ghi của các vết (*log of traces*) được theo dõi và kiểm tra tính khả thỏa với đặc tả yêu cầu hệ thống. Các yêu cầu có thể được đặc tả bằng logic thời gian (*linear temporal logic*), automát,... So với phương pháp kiểm chứng tĩnh thì kiểm chứng tại thời điểm thực thi được thực hiện trong khi thực thi hệ thống. Do đó, phương pháp này còn được gọi là kiểm thử bị động (*passive testing*). Kiểm chứng tại thời điểm thực thi nhằm bảo đảm sự tuân thủ giữa cài đặt hệ thống phần mềm so với đặc tả thiết kế của nó. Các lý do sau được lựa chọn khi sử dụng phương pháp kiểm chứng tại thời điểm thực thi.

1. Không thể bảo đảm tính đúng đắn giữa sự cài đặt của chương trình so với đặc tả thiết kế của nó,
2. Nhiều thông tin chỉ sẵn có hoặc thuận tiện ở thời điểm thực thi chương trình,
3. Các hành vi của hệ thống có thể phụ thuộc vào môi trường khi nó được thực thi,
4. Với các hệ thống an toàn và bảo mật cao thi việc giám sát các hành vi hoặc thuộc tính đã được thử nghiệm hoặc chưng minh bằng các phương pháp tĩnh là cần thiết.

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*) [65, 73]. Thuộc tính an toàn bảo đảm chương trình luôn thỏa mãn (*luôn đúng*) các ràng buộc của nó. Ví dụ như ràng buộc về sự xung đột (*interference*) giữa các tiến trình. Thuộc tính thực hiện được bảo đảm chương trình cuối cùng sẽ thỏa mãn (*sẽ đúng*) các ràng buộc của nó. Ví dụ như tính dừng của các tiến trình (*process termination*) và các tiến trình

khi muốn truy cập vào tài nguyên chia sẻ (*shared resource*) thì cuối cùng nó sẽ truy cập được. 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 và các tiến trình khác không có cơ chế rõ ràng để ngăn chặn sự truy cập đồng thời này. 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. Một thuộc tính khác tương tự như khóa chết khi các tiến trình liên tục thay đổi trạng thái của nó để đáp ứng với những thay đổi của các tiến trình khác mà không đạt được mục đích cuối cùng. 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

Java là ngôn ngữ lập trình hướng đối tượng hỗ trợ lập trình tương tranh với cơ chế đồng bộ hóa giữa các tiến trình, mô hình bộ nhớ chia sẻ, môi trường lập trình trực quan và thư viện phong phú với nhiều giao diện lập trình tương tranh khác nhau. Java được biết đến như một ngôn ngữ lập trình tương tranh được sử dụng rộng rãi trong công nghiệp phần mềm.

Sự tương tranh trong Java [82] 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`. Tuy nhiên, với các hệ thống lớn gồm nhiều tiến trình thì sử dụng các phương thức này để điều khiển sự thực thi của các tiến trình là không an toàn, do chúng ta khó kiểm soát tất cả các tiến trình. Do đó, Java cung cấp một mô hình tương tranh để giải quyết sự đồng bộ hóa giữa các tiến trình. Khi nhiều tiến trình cùng muốn truy cập vào dữ liệu chia sẻ trong một vùng giới hạn được đánh dấu bằng từ khóa `synchoronized` thì tại một thời điểm khóa của vùng xung đột chỉ cho phép một tiến trình được phép truy cập. Một tiến trình sẽ sử dụng phương thức `wait` để chờ khi nó không thể truy cập vào vùng xung đột mà đã bị chiếm giữ bởi tiến trình khác. Các tiến trình có thể được đánh thức bằng các phương thức `notify` hoặc `notifyAll`. Chiến lược giám sát ở mức thấp của Java không tránh được các lỗi liên quan về tương tranh như khóa chết, xung đột. Trong các mục tiếp theo của chương này, luận án trình bày một số mô hình và công cụ để đặc tả và kiểm chứng các thành phần Java tương tranh.

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 [46] biểu diễn sự tương tác giữa các tiến trình trong bộ nhớ. Trong đó, mỗi tiến trình sẽ gọi các hành động sau.

- `use` đọc một vùng nhớ (*region*) trong bộ nhớ làm việc (*working memory*),
- `assign` ghi vào vùng nhớ đã được đọc trong bộ nhớ làm việc,
- `read` bắt đầu đọc dữ liệu từ bộ nhớ chính của vùng nhớ,
- `load` kết thúc việc đọc dữ liệu từ bộ nhớ chính của vùng nhớ,
- `store` bắt đầu ghi dữ liệu từ bộ nhớ làm việc vào bộ nhớ chính của vùng nhớ,
- `write` kết thúc việc ghi dữ liệu từ bộ nhớ làm việc vào bộ nhớ chính của vùng nhớ,
- `lock` lấy giá trị trong bộ nhớ chính và chuyển giao cho một tiến trình đang làm việc trong bộ nhớ thông qua các hành động `read` và `load`,
- `unlock` đẩy các giá trị của một tiến trình đang lắm giữ trong bộ nhớ làm việc vào bộ nhớ chính thông qua các hành động `store` và `write`.

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 [64] 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. Danh sách 2.1 biểu diễn một đặc tả JML với các biểu thức tiền và hậu điều kiện để kiểm tra trạng thái cho phương thức `producer`. 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.

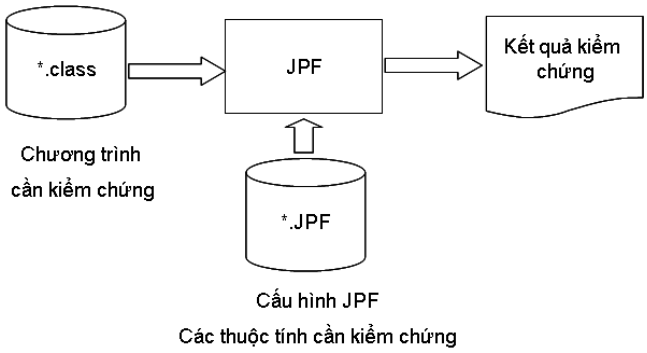
```
//@ requires state == CONSUMER || state == OPEN;
//@ ensures state == PRODUCER;
   public void  producer() {
        // ...
   }
```

Danh sách 2.1 – Dạng đặc tả JML cho phương thức producer.

Hiện nay, đã có nhiều công cụ được xây dựng và phát triển để kiểm chứng mã Java cùng với đặc tả JML của nó như ESC/Java2 [44], RAC [31].

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

Java PathFinder (JPF) [80] là một công cụ kiểm chứng các chương trình Java tương tranh dưới dạng bytecode (*Hình 2.1*). 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.



Hình 2.1 – Kiểm chứng chương trình Java với JPF.

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]. Giao diện `property` như trong Danh sách 2.2 cho phép định nghĩa các thuộc tính cần kiểm chứng bằng cách tạo ra các lớp thực thi sau đó gắn vào đối tượng `search` của JPF bằng cách gọi `jpf.getSearch().addProperty()`.

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

Danh sách 2.2 – Giao diện của lớp property.

Các `listener` cho phép, tương tác và mở rộng việc thực thi của JPF tại thời điểm chạy chương trình. JPF cho phép chúng ta tạo ra các `listener` và gắn nó vào

thể hiện của JPF. Khi thực thi chương trình thì mỗi khi có một sự kiện nào đó xảy ra nó sẽ báo cho `listener` biết. Các sự kiện này bao quát gần như toàn bộ quá trình thực thi của JPF từ những sự kiện ở mức thấp như thực thi các chỉ thị (*instructionExecuted*) cho tới những sự kiện ở mức cao như khi đã hoàn thành việc tìm kiếm (*searchFinished*). Để tạo ra các `listener` này, JPF đưa ra hai giao diện tương ứng với hai đối tượng nguồn của các sự kiện là `SearchListener` và `VMListener` như trong Danh sách 2.3 [2]. Hai giao diện này đưa ra các phương thức ứng với từng sự kiện sẽ báo cho `listener` biết. `SearchListener` được dùng để báo cáo về các sự kiện diễn ra trong suốt quá trình tìm kiếm trên không gian trạng thái còn `VMListener` sẽ báo cáo các sự kiện là các hoạt động của máy ảo.

```
public class MyPropertyListener extends PropertyListenerAdapter{
  boolean propertyHolds = true;
  ..
  /********************** Property interface **************/
  public boolean check (Search search, VM vm) {
    return propertyHolds;
  }
  public String getErrorMessage () {
    return "My property violated " + ...;
  }
  /********************** VMListener interface *************/
  public void instructionExecuted (VM vm) {
    JVM jvm = (JVM)vm;
    Instruction insn = jvm.getLastInstruction();
    if (insn instanceof ...) {
       .. propertyHolds = false; ..
    }
  }
}
```

Danh sách 2.3 – Giao diện của lớp listener.

2.4 Phương pháp hình thức với Event-B

Event-B [8] 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. Tính nhất quán của các mô hình được bảo đảm thông qua việc sinh các mệnh đề cần chứng minh (*proof obligations*).

2.4.1 Máy và Ngữ cảnh

Các mô hình Event-B [8] 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 (*set*), 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. Hình 2.2 biểu diễn cấu trúc tổng quát của Máy bên trái và Ngữ cảnh bên phải.

2.4.2 Sự kiện

Mô hình hệ thống với Event-B được bắt đầu từ các sự kiện 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 :

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

< *machine_identifier* >
refines
< *machine_identifier* >
sees
< *context_identifier_list* >
variables
< *variable_identifier_list* >
invariants
< *label* >:< *predicate* >
...
theorems
< *label* >:< *predicate* >
variant
< *variant* >
events
< *event_list* >
end

(a) machine

< *context_identifier* >
extends
< *context_identifier_list* >
sets
< *set_identifier_list* >
constants
< *constant_identifier_list* >
axioms
< *label* >:< *predicate* >
...
theorems
< *label* >:< *predicate* >
...
end

(b) context

Hình 2.2 – Cấu trúc tổng quát của máy và ngữ cảnh.

Vì thế, khi trạng thái v thỏa mãn điều kiện $G(v) = true$, hành động $A(v)$ sẽ được thực hiện. Hình 2.3 biểu diễn cấu trúc tổng quát của một sự kiện. Trong đó :

1. Mệnh đề *status* có thể là *ordinary*, *convergent* (*sự kiện phải làm tăng giá trị các biến của nó*), *anticipated* (*sự kiện không được làm tăng giá trị các biến của nó*),
2. Mệnh đề *refine* hiển thị danh sách các sự kiện trừu tượng được làm mịn,
3. Mệnh đề *any* hiển thị các tham số nếu có,
4. Mệnh đề *where* chứa các điều kiện để sự kiện được kích hoạt,
5. Mệnh đề *with* chứa các tham số phải nhận giá trị trả về của sự kiện trừu tượng tương ứng,
6. Mệnh đề *then* chứa các hành động của sự kiện. Các hành động này sẽ được thực thi khi các điều kiện của sự kiện được thỏa mãn.

Một hành động có thể là đơn định (*deterministic*) hoặc không đơn định (*non-deterministic*), các hành động được thực hiện tuần tự. Dạng thông thường của hành động đơn định được biểu diễn là < *variable_identifier* >:=< *expression* >, ví dụ $act1 : x := x+y$. Các hành động không đơn định có dạng :
< *variable_identifier_list* >:|< *before_after_predicate* > hoặc
< *variable_identifier* >:∈< *set_expression* >, ví dụ $act : x :\in A \cup \{y\}$. Biến x

< *event_identifier* >
status
{*ordinary*, *convergent*, *anticipated*}

refines
< *event_identifier_list* >
any
< *parameter_identifier_list* >
where
< *label* >:< *predicate* >
...
with
< *label* >:< *witness* >
...
then
< *label* >:< *action* >
...
end

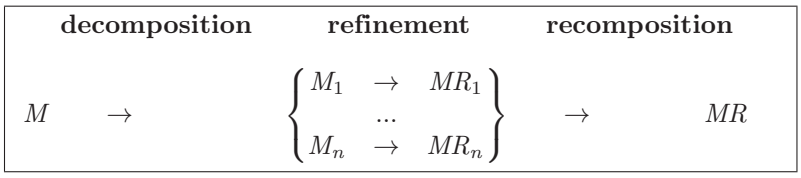
Hình 2.3 – Cấu trúc tổng quát của sự kiện.

trong hành động *act* trở thành thành viên của tập $A \cup \{y\}$, các biến A và y không thay đổi.

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 [9, 54]. Sự phân rã của mô hình M được định nghĩa như sau (Hình 2.4).

1. M được phân rã thành các mô hình con $M_1, ..., M_n$,
2. Các sự kiện của M được phân hoạch thành các sự kiện của các mô hình con,
3. Các mô hình con sau đó được độc lập làm mịn một số lần $MR_1, ..., MR_n$,
4. Các mô hình làm mịn được kết hợp lại thành mô hình MR,
5. Mô hình kết hợp phải bảo đảm là được làm mịn từ M.



Hình 2.4 – Sự phân rã và kết hợp.

2.4.4 Sinh mệnh đề cần chứng minh

RODIN [1, 10, 11] (*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.

RODIN sẽ kiểm tra tĩnh các Máy và Ngữ cảnh để sinh ra các mệnh đề bằng bộ sinh mệnh đề cần chứng minh (*Proof Obligation Generator*). Ví dụ một sự kiện *evt* có dạng như trong Hình 2.5, khi đó luật sinh mệnh đề tính chất bảo toàn của các bất biến như trong Bảng 2.2. Trong đó, s, c, v và x lần lượt là các tập hợp, hằng và biến của Máy. Tiên đề-định lý, bất biến-định lý lần lượt được biểu diễn bằng các vị từ $A(s, c)$, $I(s, c, v)$. Các điều kiện của sự kiện được biểu diễn bằng vị từ $G(s, c, v, x)$, $BA(s, c, v, x, v')$ là vị từ trước sau của một sự kiện.

$< evt >$
any x **where**
$G(s, c, v, x)$
then
$v :| BA(s, c, v, x, v')$
end

Hình 2.5 – Sự kiện sinh các mệnh đề chứng minh để bảo toàn bất biến.

Các kết quả được chứng minh một cách tự động bằng bộ chứng minh (*prover*) hoặc bán tự động thông qua tương tác với người dùng.

Bảng 2.2 – Luật sinh mệnh đề cần chứng minh để bảo toàn bất biến

Axioms and theorems	$A(s, c)$	
Invariants and theorems	$I(s, c, v)$	
Guards of the event	$G(s, c, v, x)$	evt/inv/INV
Before–after predicate of the event	$BA(s, c, v, x, v')$	
$\vdash$	$\vdash$	
Modified specific invariant	$inv(s, c, v')$	

2.5 Ngôn ngữ mô hình hóa UML

Ngôn ngữ mô hình hóa thống nhất UML [3, 71] (*Unified Modeling Language*) là một chuẩn của tổ chức OMG (*Object Management Group*). UML gồm một tập các ký pháp đồ họa thống nhất được sử dụng rộng rãi và hiệu quả trong việc đặc tả và thiết kế các hệ thống phần mềm theo phương pháp hướng đối tượng [27, 45]. Mô hình hóa hệ thống bằng UML là trực quan, dễ dàng xác định được các thành phần chính, cấu trúc tĩnh cũng như các hành vi động của hệ thống.

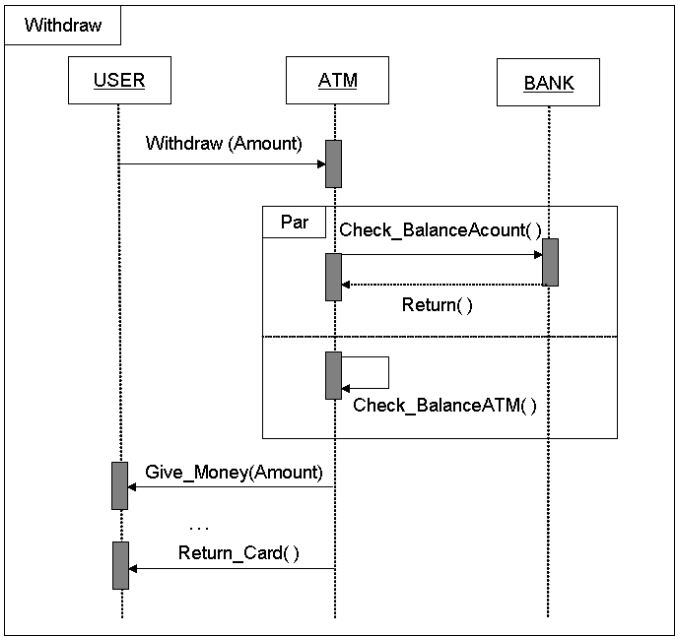
UML 2.0 gồm mười ba biểu đồ được sử dụng để đặc tả các khía cạnh khác nhau của hệ thống. Trong chương này, chúng tôi giới thiệu một số biểu đồ liên quan được sử dụng trong các Chương 5, 6 của luận án.

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. Hình 2.6 mô tả một biểu đồ tuần tự của giao thức rút tiền của hệ thống ATM. Một số quy ước khi biểu diễn bằng biểu đồ tuần tự như sau.

1. Các thông điệp song song được mô tả trong khung *Par*,
2. Các biểu thức tiền và hậu điều kiện phải được biểu diễn trong cặp dấu ngoặc [],

3. Các biểu tiền điều kiện phải được thỏa mãn trước khi đối tượng chuyển tiếp từ trạng thái này sang trạng thái khác,
4. Biểu thức hậu điều kiện phải thỏa mãn khi đối tượng kết thúc và chuyển trạng thái mới.

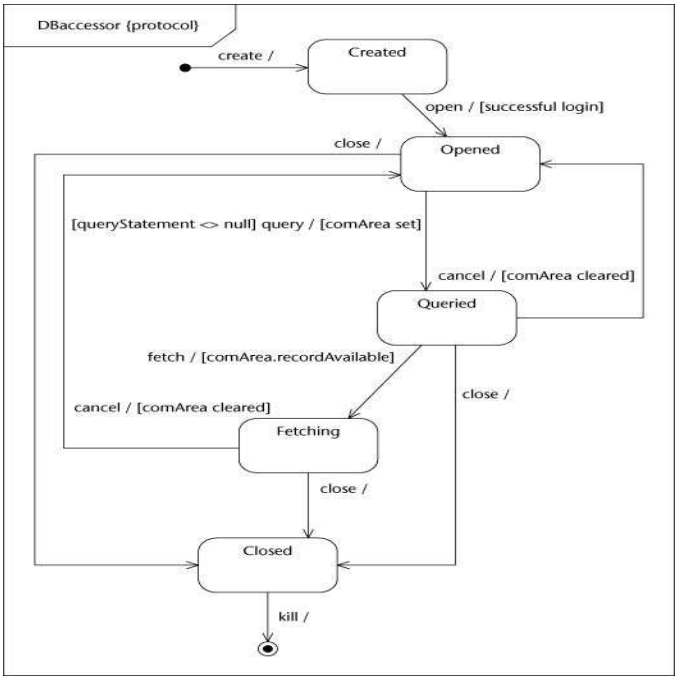


Hình 2.6 – Biểu đồ tuần tự biểu diễn giao thức rút tiền của hệ thống ATM.

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. Ví dụ giao thức truy cập một cơ sở dữ liệu với các phương thức `Open(..)`, `Close(..)`, `Query(..)`, `fetch(..)`, `cancel(..)`, `create(..)` và `kill(..)` được biểu diễn bằng biểu đồ PSM trong Hình 2.7. Thứ tự thực hiện của các phương thức được biểu diễn bằng các cung trong biểu đồ. Khi đặc tả một giao thức tương tác bằng PSM chúng ta cần tuân thủ các nguyên lý sau.

1. Các trạng thái có thể được biểu diễn bằng tên của các sự kiện, nhưng không thể biểu diễn các hành động vào, ra và các hành động bên trong hoặc sự thực thi của các hành động,



Hình 2.7 – Máy trạng thái biểu diễn giao thức tương tác truy cập cơ sở dữ liệu.

2. Các phép chuyển trạng thái chỉ thể hiện các phép toán, không biểu diễn các hành động hoặc sự kiện,
3. Các biểu thức tiền và hậu điều kiện phải được biểu diễn trong cặp dấu ngoặc [], ví dụ [queryStatement <> null] query / [comArea set] (Hình 2.7),
4. Các biểu tiền điều kiện phải được thỏa mãn trước khi đối tượng chuyển tiếp từ trạng thái này sang trạng thái khác. Ví dụ trong Hình 2.7 khi tiền điều kiện `queryStatement<>null` thỏa mãn và đối tượng ở trạng thái `Opened` thì sẽ chuyển sang trạng thái `Queried`.
5. Biểu thức hậu điều kiện phải được thỏa mãn khi đối tượng kết thúc và chuyển trạng thái mới.

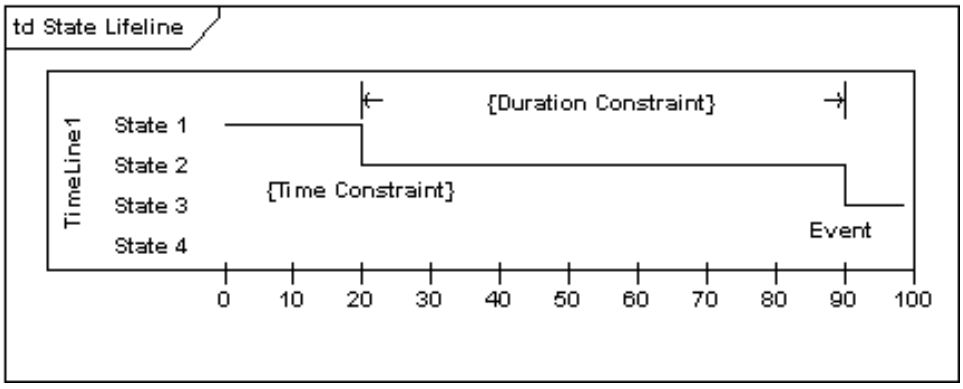
2.5.3 Biểu đồ thời gian

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

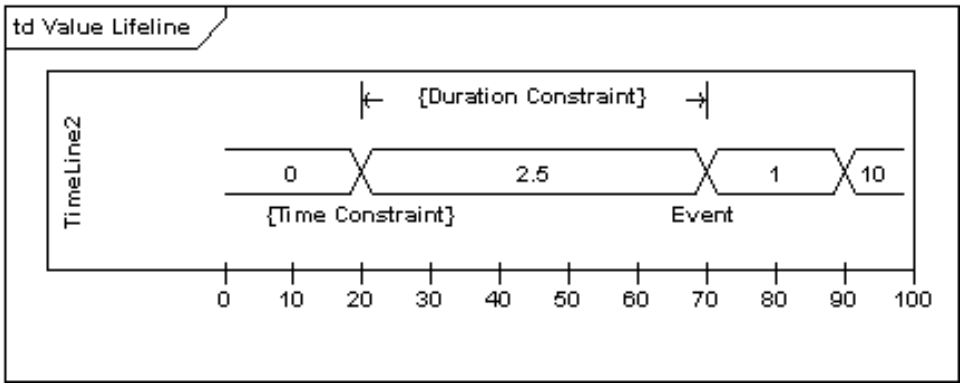
các hệ thống thời gian thực, hệ thống nhúng, tuy nhiên nó cũng có thể dùng để mô hình các hệ thống nghiệp vụ khác.

Biểu đồ thời gian biểu diễn sự thay đổi trạng thái hoặc giá trị của các sự kiện theo thời gian, nó cũng biểu diễn sự tương tác giữa các sự kiện thời gian và ràng buộc thời gian của các sự kiện này. Có ba dạng biểu đồ thời gian là biểu đồ giá trị (*value lifeline*- Hình 2.8), biểu đồ trạng thái (*state lifeline*-Hình 2.9) và dạng kết hợp giữa biểu đồ giá trị và biểu đồ trạng thái (Hình 2.10). Trong đó :

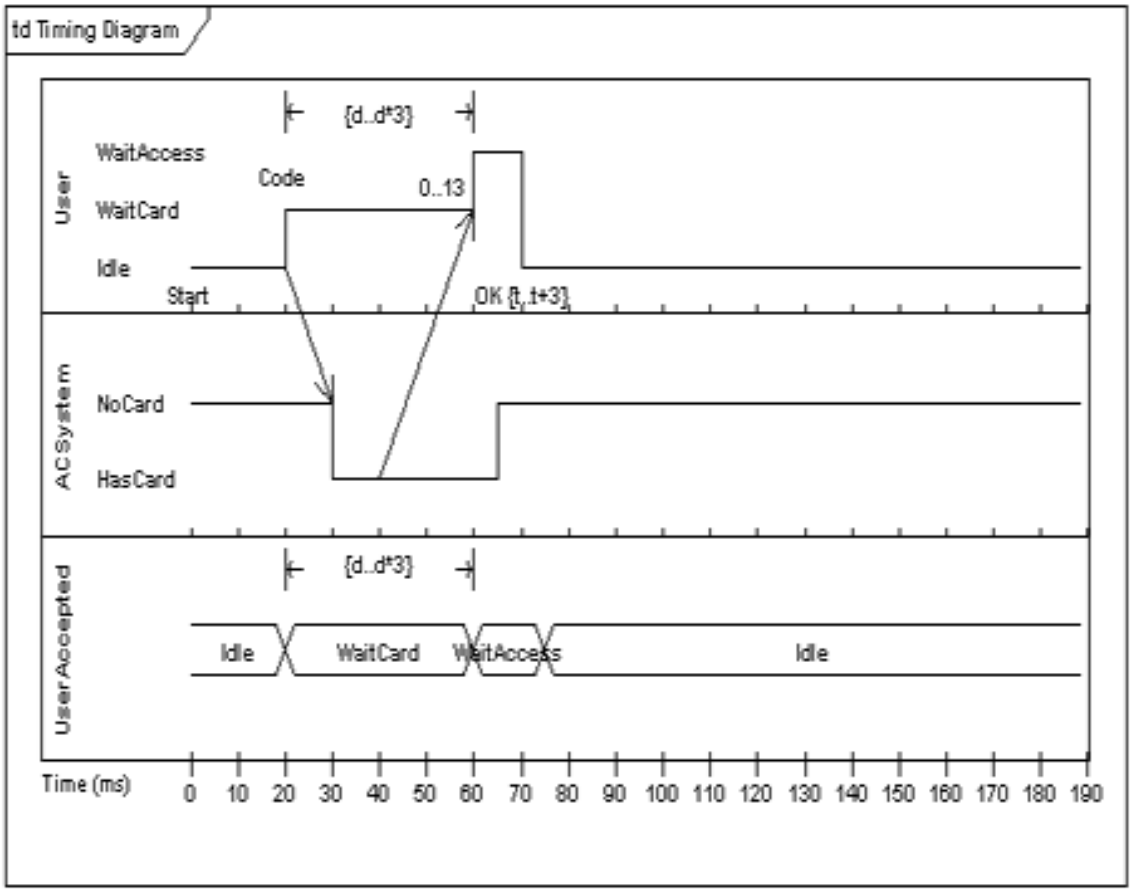
1. Biểu đồ trạng thái biểu diễn sự thay đổi trạng thái của các sự kiện theo thời gian, trục x biểu diễn các đơn vị thời gian, trục y biểu diễn danh sách các trạng thái,
2. Biểu đồ giá trị biểu diễn sự thay đổi giá trị của các sự kiện theo thời gian, trục x biểu diễn các đơn vị thời gian, giá trị của các sự kiện được hiển thị giữa cặp đường thằng song song nằm ngang. Sự thay đổi của cặp đường thẳng này biểu diễn sự thay đổi giá trị của các sự kiện,
3. Biểu đồ kết hợp giữa biểu đồ trạng thái và biểu đồ giá trị, trong đó trục x biểu diễn các đơn vị thời gian, các thông điệp có thể truyền từ biểu đồ trạng thái sang biểu đồ giá trị và ngược lại từ biểu đồ giá trị sang biểu đồ trạng thái. Mỗi phép chuyển trạng thái hoặc sự kiện có thể có một sự kiện được định nghĩa, một ràng buộc thời gian để sự kiện phải xảy ra và một đoạn thời gian xác định cho mỗi sự kiện.



Hình 2.8 – Dạng trạng thái của biểu đồ thời gian.



Hình 2.9 – Dạng giá trị của biểu đồ thời gian.



Hình 2.10 – Biểu đồ thời gian dạng kết hợp.

2.6 Lập trình hướng khía cạnh

Phương pháp lập trình hướng khía cạnh (*Aspect-Oriented Programming - AOP*) [43, 59, 60] 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 khía cạnh (*aspect*) thay vì dàn trải chúng trên các môđun nghiệp vụ liên quan. Các khía cạnh 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 [36, 53, 60] là một công cụ AOP cho ngôn ngữ lập trình Java. Trình biên dịch AspectJ sẽ đan xen chương trình Java chính với các khía cạnh thành các tệp mã bytecode chạy trên chính máy ảo Java. Trong mục này chúng tôi trình bày một số khái niệm liên quan về AspectJ được sử dụng trong các chương 5 và 6 của luận án.

2.6.1 Thực thi cắt ngang

Trong AspectJ, quá trình trình biên dịch thực thi các quy tắc đan để đan kết các môđun cắt ngang vào môđun nghiệp vụ chính được gọi là thực thi cắt ngang (*crosscutting*). AspectJ định nghĩa hai loại thực thi cắt ngang là thực thi cắt ngang động và thực thi cắt ngang tĩnh : Thực thi cắt ngang động (*dynamic crosscutting*) là việc đan các hành vi mới vào quá trình thực thi một chương trình. Trình biên dịch sẽ dựa vào tập các quy tắc đan để xác định điểm đan và chèn thêm hoặc thay thế luồng thực thi chương trình chính bằng môđun cắt ngang, từ đó làm thay đổi hành vi của hệ thống. Hầu hết việc thực thi cắt ngang trong AspectJ đều là thực thi cắt ngang động.

Thực thi cắt ngang tĩnh (*static crosscutting*) là quá trình đan một sửa đổi vào cấu trúc tĩnh của lớp, giao diện hay các khía cạnh hệ thống. Chức năng chính của thực thi cắt ngang tĩnh là hỗ trợ cho thực thi cắt ngang động. Ví dụ như thêm dữ liệu và phương thức mới vào lớp đã có nhằm định nghĩa trạng thái và hành vi của lớp đó để sử dụng trong các hành vi cắt ngang động. Thực thi cắt ngang tĩnh

còn được sử dụng nhằm khai báo các cảnh báo và lỗi tại thời điểm biên dịch cho nhiều môđun.

Trong AOP, mỗi khía cạnh được coi là biểu diễn của một quy tắc đan, nó chỉ ra môđun cần được đan, vị trí đan trong môđun đó, hành vi sẽ được đan vào thông qua các định nghĩa của điểm nối, hướng cắt và mã hành vi.

2.6.2 Điểm nối

Điểm nối (*joinpoint*) là một điểm có thể xác định trong quá trình thực thi một chương trình, điểm này chính là vị trí mà các hành động thực thi cắt ngang được đan vào. Trong AspectJ có một số loại điểm nối được chia thành hai loại thực thi và triệu hồi phương thức sau.

1. Điểm nối thực thi phương thức (*method execution joinpoint*) : điểm này nằm trên chính phần thân của phương thức thực thi, nó gồm toàn bộ các lệnh nằm trong thân phương thức,
2. Điểm nối triệu gọi phương thức (*method call joinpoint*) : điểm này nằm trên phần chương trình gọi phương thức đang xét, nó chính là điểm mà phương thức đang xét được gọi.

2.6.3 Hướng cắt

Trong AspectJ, điểm nối thường được sử dụng trong một hướng cắt (*pointcut*), mỗi hướng cắt chứa một nhóm các điểm nối cùng với ngữ cảnh của nó. Ta có thể khai báo hướng cắt trong một khía cạnh, một lớp hoặc một giao diện. Giống như phương thức, có thể sử dụng định danh truy cập (*public, private*) để giới hạn quyền truy cập đến hướng cắt. Các hướng cắt có thể có tên hoặc không tên. Các hướng cắt không tên, giống như các lớp không tên, được định nghĩa tại nơi sử dụng. Các hướng cắt được đặt tên thì có thể được tham chiếu từ nhiều nơi khác. Cú pháp khai báo hướng cắt như trong Danh sách 2.4 [61]. Trong đó *access_specifier* khai báo định danh truy cập có thể là `public` hoặc `private`, *pointcut_name* khai báo tên của

hướng cắt, *argument_list* danh sách các tham số, và cuối cùng *pointcut_definition* là định nghĩa hướng cắt (*các lệnh*).

```
<access_specifier> <pointcut> <pointcut_name>(<argument_list>):
 pointcut_definition;
```

Danh sách 2.4 – Cấu trúc cơ bản của hướng cắt.

2.6.4 Mã hành vi

Mã hành vi (*advice*) là đoạn mã định nghĩa hành vi được đan vào mã nguồn tại điểm điểm nối [53]. Mã hành vi được thực thi trước (*before advice*), sau (*after advice*), hoặc xung quanh (*around advice*) một điểm nối. Mã hành vi có thể được sử dụng để đưa ra thông báo trước khi đoạn mã tại điểm nối được thực thi lưu vết hoặc kiểm tra lỗi. Trong AspectJ định nghĩa ba loại mã hành vi sau.

1. Mã hành vi trước là loại mã được thực thi trước các điểm nối,
2. Mã hành vi sau là loại mã được thực thi ngay sau các điểm nối,
3. Mã hành vi xung quanh là loại mã mạnh nhất, nó bao gồm cả mã hành vi trước và sau. Mã hành vi xung quanh có thể chỉnh sửa đoạn mã tại điểm nối, nó có thể thay thế, thậm chí bỏ qua sự thực thi của điểm nối đó. Mã hành vi xung quanh phải khai báo giá trị trả về cùng kiểu với kiểu trả về của điểm nối. Trong mã hành vi xung quanh, sự thực thi của điểm nối được thể hiện thông qua `proceed()`. Nếu trong mã hành vi không gọi `proceed()` thì sự thực thi của điểm nối sẽ bị bỏ qua.

Danh sách 2.6 biểu diễn một khía cạnh với mã hành vi trước và sau. Như vậy điểm nối và mã hành vi kết hợp với nhau tạo nên quy tắc thực thi cắt ngang động : hướng cắt xác định các điểm cắt cần thiết còn mã hành vi cung cấp các hành động sẽ xảy ra tại điểm nối đó.

2.6.5 Khía cạnh

Trong AspectJ, khía cạnh (*aspect*) đóng vai trò là đơn vị trung tâm giống như lớp là đơn vị trung tâm của Java. Nó là sự kết hợp của hướng cắt, mã hành vi, điểm nối và các dữ liệu, phương thức khác. Một khía cạnh thông thường có các đặc điểm sau.

1. Có thể sử dụng các định danh phạm vi truy cập (*public, private, protect*) cho nó. Ngoài ra khía cạnh còn có thể có định danh phạm vi truy cập "privileged", định danh này cho phép khía cạnh có thể truy cập đến các thành viên riêng của các lớp mà chúng cắt ngang,
2. Có thể khai báo khía cạnh trừu tượng bằng từ khóa `abstract`,
3. Khía cạnh không thể được thể hiện hóa trực tiếp,
4. Khía cạnh có thể kế thừa các lớp và các khía cạnh trừu tượng khác, có thể thực thi các giao diện nhưng không thể kế thừa từ các khía cạnh cụ thể (*không trừu tượng*),
5. Khía cạnh có thể nằm trong các lớp và các giao diện.

Cấu trúc cơ bản của khía cạnh như trong Danh sách 2.5 [61].

```
<access_type>[privilege] [static] aspect <aspect_name>
 [instantiation]
{
// pointcut
//advice
//attribute/method
}
```

Danh sách 2.5 – Cấu trúc cơ bản của một khía cạnh.

Trong đó, *access_type* để chỉ phạm vi truy cập (*public, private*), *privilege* từ khóa này có thể có hoặc không, nếu có nó cho phép khía cạnh truy cập được vào các phần tử riêng của lớp mà chúng cắt ngang, *aspect_name* là tên khía cạnh, *instantiation* có thể có hoặc không, xác định dạng thể hiện của khía cạnh (*singleton, perthis, pertarget, percflow, perflowbelow*).

```
public aspect ProtocolCheck {
 public final static int ST_START = 0;
 public final static int ST_init = 2;
 public int Applet.state = ST_START;
 pointcut pc_init(Applet o):target(o)&&call(void init());
   before(Applet o):pc_init(o){
      if (!(o.state==ST_START))
        log(thisjoinpoint);
    }
  after(Applet o):pc_init(o) {
    o.state = ST_init;
  }
}
```

Danh sách 2.6 – Khía cạnh với mã hành vi trước và sau để kiểm tra trạng thái khởi tạo của phương thức `init()`.

2.7 Kết luận

Trong chương này chúng tôi trình bày tổng quan về một số kiến thức nền được sử dụng cho các đóng góp của luận án. Mục 2.1 giới thiệu tổng quan về các phương pháp kiểm chứng hình thức và kiểm chứng tại thời điểm thực thi. Các kết quả của luận án được trình bày theo hai hướng tiếp cận này. Mục 2.4 giới thiệu các thành phần cơ bản của ngôn ngữ đặc tả hình thức với Event-B, ngôn ngữ này được sử dụng để đặc tả và kiểm chứng các bài toán trong các Chương 3 và 4. Mục 2.5, 2.6 giới thiệu một số biểu đồ của UML, phương pháp lập trình hướng khía cạnh AOP được sử dụng để đặc tả và kiểm chứng sự tuân thủ của chương trình so với đặc tả của nó, các kết quả được trình bày trong các Chương 5 và 6. Mục 2.2, 2.3 trình bày một số vấn đề trong các chương trình Java tương tranh. Mục 2.3.3 giới thiệu bộ công cụ JPF để kiểm chứng mã Java.

Chương 3

Ràng buộc thứ tự giữa các tiến trình tương tranh

3.1 Giới thiệu

Các tiến trình (*process*) trong một hệ thống tương tranh thường phải được đồng bộ hóa. Sự đồng bộ hóa giữa các tiến trình được phân thành hai loại cộng tác hoặc cạnh tranh. Một trong những ví dụ điển hình về sự cộng tác giữa các tiến trình là vấn đề cung cấp-tiêu thụ (*producer-consumer*) với hai tiến trình producer và consumer. Trong đó tiến trình producer cung cấp các phần tử dữ liệu, sau đó tiến trình consumer sẽ tiêu thụ nó.

Cấp phát tài nguyên cho các tiến trình phải giải quyết vấn đề xung đột khi nhiều tiến trình cùng muốn sử dụng tài nguyên chia sẻ như dữ liệu, tệp, máy in,...Ví dụ, tính nhất quán của dữ liệu sẽ bị phá vỡ nếu hai tiến trình đọc-ghi cùng cập nhật chung một tệp tại cùng một thời điểm. Danh sách 3.1 biểu diễn một chương trình Java tương tranh được cài đặt cho vấn đề cung cấp tiêu thụ. Trong chương trình này các tiến trình không cộng tác với nhau, vì thế dễ phát sinh các lỗi tương tranh dữ liệu. Do đó vấn đề đặt ra là phải đặc tả ràng buộc về thứ tự thực hiện [38, 67] (*giao thức tương tác*) giữa các tiến trình nhằm bảo đảm tính nhất quán của dữ liệu chia sẻ và dữ liệu đầu vào-đầu ra.

```
...
class Producer implements Runnable {
     public void run() {
       while (true) {
         Object o = createNewObject();
          push(o);
       }
     }
 }
 class Consumer implements Runnable {
     public void run() {
       while (true) {
            Object o = pop();
            doSomethingWith(o);
        }
     }
 }
...
```

Danh sách 3.1 – Chương trình Java cho vấn đề cung cấp tiêu thụ.

Nhằm phát hiện lỗi ở mức thiết kế, 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 sử dụng phương pháp hình thức với Event-B. Phương pháp được đề xuất đã giải quyết được đặc tả cho các vấn đề như vùng xung đột (*critical section*), cung cấp-tiêu thụ (*producer-consumer*) và đọc-ghi dữ liệu (*reader-writer*). Các đặc tả này sẽ được cài đặt (*phát sinh*) mã của các chương trình Java tương tranh hoặc tương đương. Các kết quả chính của phương pháp được trình bày trong [77].

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

3.2.1 Mô tả phương pháp

Chúng tôi định nghĩa một hệ thống tương tranh với giao thức tương tác Γ như sau.

Định nghĩa 3.1 (Hệ thống tương tranh)**.** *Một hệ thống tương tranh (*Concurrent system-CS*) là một bộ bốn* $CS = \langle Pc, Br, \alpha, \Gamma \rangle$*. Trong đó :*

- Pc *: tập hữu hạn các tiến trình,*
- Br *: tập hữu hạn các hành vi có thể trong* CS*,*
- $\alpha : Br \rightarrow Pc$ *hàm gán mỗi chức năng của* CS *mà tiến trình thực hiện hành vi đó,*
- Γ *: giao thức tương tác đặc tả thứ tự thực hiện của các tiến trình.* $\Gamma \triangleq p \mid \Gamma, p \mid \Gamma \parallel p$*. Với* $p \in Pc$*, các kí hiệu "," và* $\parallel$ *lần lượt biểu diễn các tiến trình được thực hiện tuần tự và song song.*

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.

Như vậy, một sự kiện của Máy có thể được biểu diễn trừu tượng bằng một quan hệ nhị phân trong tập các trạng thái. Quan hệ này biểu diễn sự kết nối giữa hai trạng thái kế tiếp trước và sau khi một sự kiện được thực hiện. Trong ký pháp của Event-B một sự kiện có thể được chia thành hai phần là điều kiện và hành động. Các hành động của một sự kiện được giả thiết là thực hiện đồng thời trên các biến khác nhau. Các biến không được gán giá trị thì nó sẽ không thay đổi [8].

Các điều kiện của sự kiện biểu diễn điều kiện cần và đủ để cho một sự kiện được kích hoạt. Khi một sự kiện được kích hoạt thì sự chuyển đổi trạng thái tương ứng

sẽ được thực hiện và ngược lại. Giả sử chúng ta có hai sự kiện $e1$ và $e2$ được mô tả như sau :

$$e1 \mathrel{\widehat{=}} \textbf{when } g1 \textbf{ then } a1 \textbf{ end}$$

$$e2 \mathrel{\widehat{=}} \textbf{when } g2 \textbf{ then } a2 \textbf{ end}$$

Nếu điều kiện $g1$ được thỏa mãn thì hành động $a1$ sẽ được thực hiện, và nếu điều kiện $g2$ được thỏa mãn thì $a2$ sẽ được thực hiện. Thông thường, với hai điều kiện $g1$ và $g2$ sao cho $g1 \cap g2 = \varnothing$ thì các sự kiện $e1$ và $e2$ sẽ được thực hiện tuần tự.

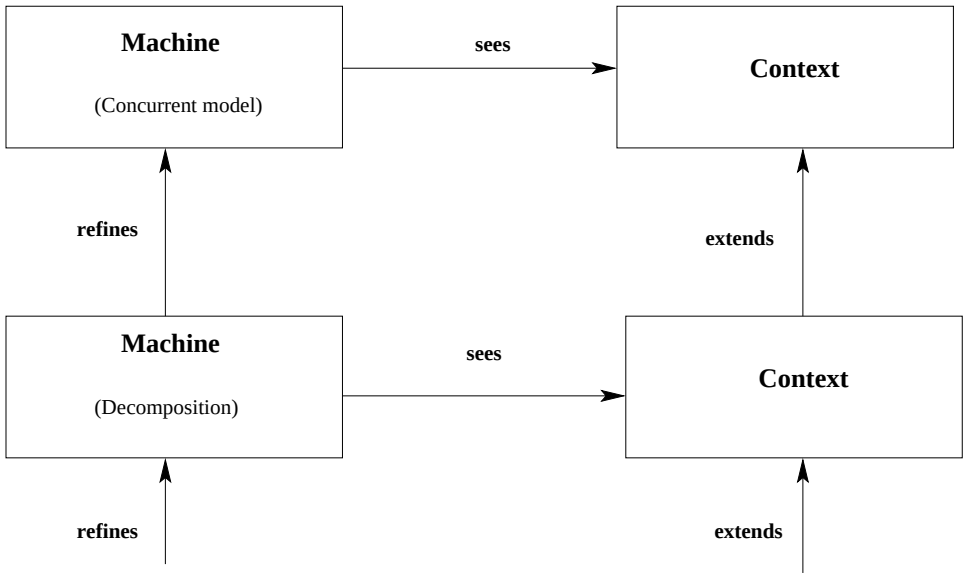
Để đặc giao thức tương tác giữa các tiến trình tương tranh, trước hết các tiến trình này được mô tả bằng các sự kiện trong mô hình Event-B. Trong đó các điều kiện của các sự kiện này là không rời nhau để bảo đảm các sự kiện sẽ được thực hiện tương tranh với nhau theo cơ chế đan xen. Khi đó để đặc tả thứ tự thực hiện của các sự kiện này, chúng tôi sử dụng một biến lgic đóng vai trò như biến semaphore trong chương trình tương tranh nhằm điều khiển sự thực thi của các sự kiện.

Cùng 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.

Từ mô hình tổng quát này, chúng tôi sẽ trình bày giải pháp để đặc tả thứ tự thực hiện của các sự kiện áp dụng cho các vấn đề về vùng xung đột (*critical section problem*), cung cấp-tiêu thụ (*producer-consumer problem*) từ bộ đệm dữ liệu và đọc-ghi (*reader-writer problem*) dữ liệu từ bộ nhớ chia sẻ (*shared memory*).

3.2.2 Vùng xung đột

Vấn đề về vùng xung đột (*critical section problem*) được mô tả như sau. Giả sử một chương trình tương tranh có n tiến trình cùng sử dụng chung một vùng dữ



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

machine CriticalSection
init
...

event1	**event2**	**eventn**
when	**when**	**when**
grd1 : $G1$	grd1 : $G2$	grd1 : Gn
then	**then**	**then**
act1 : $CS1$	act1 : $CS2$	act1 : $CS3$
end	**end**	**end**

Hình 3.2 – Máy truy cập vào vùng xung đột.

liệu chia sẻ, mỗi tiến trình có một phân đoạn mã gọi là `critical section` để truy cập vào vùng dữ liệu chia sẻ. Vấn đề đặt ra là khi một tiến trình đang thực hiện các lệnh trong phân đoạn mã `critical section` để truy cập dữ liệu thì phải bảo đảm không một tiến trình nào khác được phép thực hiện lệnh trong phân đoạn `critical section` của nó.

Không mất tính tổng quát, chúng tôi giả thiết thứ tự thực hiện của các sự kiện tuân theo giao thức $\Gamma \mathrel{\widehat{=}} [init, ec_1, ec_2, ec_3, ..., ec_n]$. Khi đó mô hình làm mịn của vấn đề vùng xung đột được biểu diễn trong Hình 3.3. Trong đó 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. Trong mô hình làm mịn này,

machine CriticalSectionR
refines CriticalSection
invariant
inv1 : $turn \in NAT$
...
init
act1 : $turn := 0$
...

event1	**event2**	**eventn**
refines event1	**refines** event2	**refines** eventn
when	**when**	**when**
grd1 : $G1$	grd1 : $G2$	grd1 : Gn
grd2 : $turn = 0$	grd2 : $turn = 1$	grd2 : $turn = n-1$
then	**then**	**then**
act1 : $CS1$	act1 : $CS2$	act1 : CSn
act2 : $turn := 1$	act2 : $turn := 2$	act2 : $turn := 0$
end	**end**	**end**

Hình 3.3 – Máy được làm mịn để truy cập vào vùng xung đột.

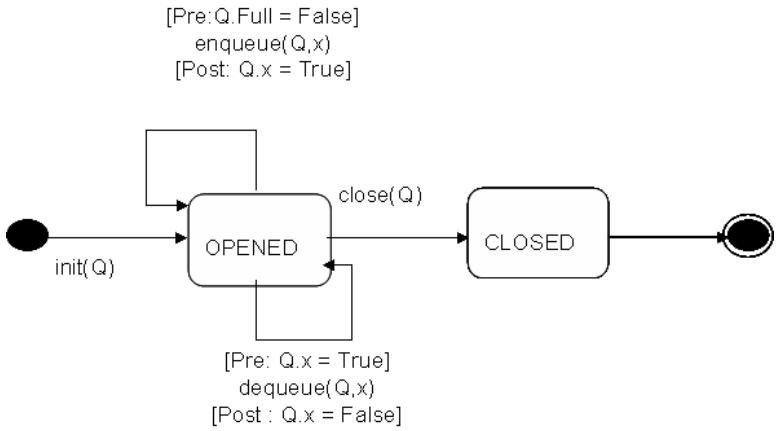
chúng tôi sử dụng kỹ thuật đồng bộ hóa với biến semaphore *turn* để điều khiển sự thực thi của các sự kiện theo giao thức Γ. Ban đầu sự kiện *init* được kích hoạt, biến *turn* được khởi tạo bằng 0 để khích hoạt sự kiện $event1$ trong giao thức khi điều kiện $G1$ của nó đồng thời được thỏa mãn. Các sự kiện khác được ngăn chặn không cho phép thực hiện, sau khi thực hiện xong, biến semaphore *turn* sẽ thay đổi giá trị để giải phóng vùng xung đột và cho phép sự kiện tiếp theo trong giao thức được thực hiện. Mục A.1 của Phụ lục A trình bày đặc tả cho vấn đề vùng xung đột với ba tiến trình được biểu diễn bằng các sự kiện $evt1$, $evt2$ và $evt3$ làm thay đổi giá trị của các biến x, y và z.

3.2.3 Cung cấp và tiêu thụ

Vấn đề cung cấp-tiêu thụ (*producer-consumer problem*) là một ví dụ điển hình về sự đồng bộ hóa giữa các tiến trình tương tranh. Chúng tôi mô tả vấn đề này bởi hai tiến trình thực hiện theo các nguyên tắc như sau (Hình 3.4).

1. Producer : tạo các phần tử dữ liệu và đẩy vào bộ đệm bằng phương thức `enqueue()`, bộ đệm được biểu diễn bằng một hàng đợi với kích thước hữu hạn và cố định,

2. Consumer : lấy các phần tử dữ liệu từ hàng đợi qua phương thức `dequeue()`,
3. Các tiến trình được thực hiện đồng thời và lặp,
4. Thứ tự thực hiện của các tiến trình tuân theo giao thức $\Gamma \widehat{=} [init, producer \mid\mid consumer, close]$ tiến trình khởi tạo init được thực hiện trước, sau đó các tiến trình producer và consumer được thực hiện đồng thời. Tại trạng thái trừu tượng `OPENED`, các tiến trình producer và consumer có thể thực hiện các phương thức `enqueue()` hoặc `dequeue()` để bổ sung hoặc loại bỏ các phần tử của hàng đợi. Khi tiến trình 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,
5. Các tiến trình producer và consumer phải bảo đảm không đẩy phần tử dữ liệu vào hàng đợi nếu nó đã đầy và không lấy dữ liệu nếu nó rỗng.



Hình 3.4 – Giao thức tương tác của vấn đề cung cấp tiêu thụ.

Mô hình khởi tạo của vấn đề cung cấp-tiêu thụ được biểu diễn như trong Hình 3.5, 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 song song. 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 $\Gamma \widehat{=} [init, producer \mid\mid consumer, close]$.

Do bộ đệm dữ liệu được giả thiết là hữu hạn và có kích thước cố định. Chúng tôi sử dụng cơ chế đồng bộ hóa với biến semaphore là *Count* được khởi tạo bằng

machine ProducerConsumer
init
...

producer	**consumer**
when	**when**
grd1 : $G1$	grd1 : $G2$
then	**then**
act1 : produce the database	act1 : consume the database
end	**end**

close
...

Hình 3.5 – Máy trừu tượng cho vấn đề cung cấp-tiêu thụ.

không để biểu diễn số phần tử của bộ đệm. Biến *Count* sẽ tăng lên một khi một phần tử dữ liệu mới được bổ sung vào bộ đệm và giảm khi một phần tử dữ liệu được loại bỏ từ bộ đệm. Biến *Buffer_Size* biểu diễn kích thước (*số phần tử tối đa*) của bộ đệm, bộ đệm sẽ đầy khi biến semaphore *Count* bằng kích thước của nó và ngược lại rỗng khi *Count* bằng không. Khi bộ đệm đã đầy sự kiện *Producers* sẽ kích hoạt sự kiện *close* thông qua biến semaphore *isClose* để chuyển sang trạng thái trừu tượng `CLOSED`. Mô hình làm mịn được đặc tả như trong Hình 3.6. Mục A.2 của Phụ lục A trình bày đặc tả cho vấn đề cung cấp-tiêu thụ với hai tiến trình và bộ đệm dữ liệu được biểu diễn bằng một hàng đợi có kích thước hữu hạn, cố định.

Mô hình làm mịn trong Hình 3.6 chỉ thực hiện tốt khi chúng ta có một tiến trình cung cấp và một tiến trình tiêu thụ. Trong trường hợp có nhiều tiến trình cung cấp đồng thời đẩy các phần tử dữ liệu, hoặc nhiều tiến trình cùng tiêu thụ dữ liệu thì sẽ gây ra trường hợp cạnh tranh dữ liệu (*data race*) do các tiến trình cùng nhau đọc/ghi dữ liệu tại một ô nhớ của bộ đệm ở cùng một thời điểm. Để giải quyết vấn đề này chúng ta cần một cơ chế loại trừ sao cho tại cùng một thời điểm chỉ cho phép một tiến trình producer và một tiến trình consumer được thực hiện song song nhau. Hình 3.7 đặc tả mô hình làm mịn thứ hai với n tiến trình producer và m tiến trình consumer được thực hiện song song. Trong mô hình làm mịn này, chúng tôi bổ sung hai tập hợp *Producers* gồm n phần tử và *Consumers* gồm m phần tử để biểu diễn số tiến trình producer và consumer của mô hình. Hai biến semaphore là *TurnP* và *TurnC* được khởi tạo là một số ngẫu nhiên thuộc tập

machine ProducerConsumerR1
refines ProducerConsumer
invariant
  inv1 : $Buffer\_size \in NAT$
  inv2 : $Count \in 0..Buffer\_size$
  ...
init
  act1 : $Count := 0$
  ...

producer
refines producer
when
  grd1 : $G1$
  grd2 : $Count < Buffer\_Size$
then
  act1 : create d
  act2 : $Count := Count+1$
  act3 : *if* $Count = Buffer\_Size$ *then* $isClose = True$
end

consumer
refines consumer
when
  grd1 : $G2$
  grd2 : $Count > 0$
then
  act1 : consume d
  act2 : $Count := Count-1$
end

close
when
  grd1 : $isClose = True$
then
  act1 : $G1 := False$
  act2 : $G2 := False$
end

Hình 3.6 – Máy làm mịn thứ nhất cho vấn đề cung cấp-tiêu thụ.

Producers và *Consumers* để khích hoạt các tiến trình producer và consumer tương ứng khi các điều kiện của nó đồng thời được thỏa mãn. Các tiến trình khác được ngăn chặn không cho phép thực hiện, sau khi thực hiện xong các biến semaphore sẽ thay đổi giá trị để giải phóng và cho phép tiến trình khác được thực hiện. Mô hình làm mịn này bảo đảm tại một thời điểm chỉ cho phép một tiến trình producer và consumer thực hiện song song, các tiến trình còn lại được thực hiện đan xen nhau nếu kích thước của bộ đệm khác rỗng và chưa đầy.

machine ProducerConsumerR2
refines ProducerConsumer1
invariant
...
init
...

producer1
when
grd1 : $G1$
grd2 : $Count < Buffer\_Size$
grd3 : $TurnP = 1$
then
act1 : create d
act2 : $Count := Count{+}1$
act3 : $TurnP :\in producers \setminus \{TurnP\}$
act4 : *if* $Count = Buffer\_Size$ *then* $isClose = True$
end

consumer1
when
grd1 : $G2$
grd2 : $Count > 0$
grd3 : $TurnC = 1$
then
act1 : consume d
act2 : $Count := Count{-}1$
act3 : $TurnC :\in producers \setminus \{TurnC\}$
end

producer2
when
grd1 : $G1$
grd2 : $Count < Buffer\_Size$
grd3 : $TurnP = 2$
then
act1 : create d
act2 : $Count := Count{+}1$
act3 : $TurnP :\in producers \setminus \{TurnP\}$
act4 : *if* $Count = Buffer\_Size$ *then* $isClose = True$
end

consumer2
when
grd1 : $G2$
grd2 : $Count > 0$
grd3 : $TurnC = 2$
then
act1 : consume d
act2 : $Count := Count{-}1$
act3 : $TurnC :\in producers \setminus \{TurnC\}$
end

...

producern
when
grd1 : $G1$
grd2 : $Count < Buffer\_Size$
grd3 : $TurnP = n$
then
act1 : create d
act2 : $Count := Count{+}1$
act3 : $TurnP :\in producers \setminus \{TurnP\}$
act4 : *if* $Count = Buffer\_Size$ *then* $isClose = True$
end

consumerm
when
grd1 : $G2$
grd2 : $Count > 0$
grd3 : $TurnC = m$
then
act1 : consume d
act2 : $Count := Count{-}1$
act3 : $TurnC :\in producers \setminus \{TurnC\}$
end

close
when
grd1 : $isClose = True$
then
act1 : $G1 := False$
act2 : $G2 := False$
end

Hình 3.7 – Máy làm mịn thứ hai cho vấn đề cung cấp-tiêu thụ.

machine ReaderWriter	
init	
...	
reader	**writer**
when	**when**
grd1 : $G1$	grd1 : $G2$
then	**then**
act1 : read the database	act1 : write the database
end	**end**

Hình 3.8 – Máy trừu tượng cho vấn đề đọc-ghi.

3.2.4 Vấn đề đọc-ghi

Chúng tôi mô tả vấn đề đọc-ghi (*readers-writers problem*) bằng hai tiến trình hoạt động theo giao thức được mô tả như sau :

1. Reader : các tiến trình đọc *reader* cần phải loại trừ các tiến trình ghi *write*, nhưng không loại trừ các tiến trình đọc *reader* khác,
2. Writer : các tiến trình ghi cần phải loại trừ tất cả các tiến trình đọc *reader* và ghi *writer* khác.

Để đặc tả bài toán này, chúng tôi giả thiết mỗi tiến trình được biểu diễn bằng một sự kiện của mô hình khởi tạo (Hình 3.8). Tương tự như mô hình khởi tạo của bài toán cung cấp-tiêu thụ, các điều kiện được giả thiết là thỏa mãn : $G1 \cap G2 \neq \varnothing$. để các sự kiện kiện của nó được thực hiện tương tranh với nhau.

Chúng tôi đặc tả các tiến trình tương tranh reader-writer như trong Hình 3.9. Trong đặc tả này biến *readers* biểu diễn số tiến trình reader được đọc song song nhau từ cơ sở dữ liệu sau khi thực hiện thành công sự kiện *StartRead* và trước khi thực hiện sự kiện *EndRead*. Tương tự, biến *writers* biểu diễn số tiến trình writer được ghi song song vào cơ sở dữ liệu sau khi thực hiện xong sự kiện *StartWrite* và trước khi thực hiện sự kiện *EndWrite*. 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.

Các biến *readers* và *writers* được tăng dần trong sự kiện *startRead* và giảm dần trong sự kiện *endRead*. Tại thời điểm ban đầu, sự kiện *startRead* sẽ kiểm tra biểu

Bảng 3.1 – Kết quả chứng minh 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

thức điều kiện xem liệu một tiến trình sẽ bị khóa hay không, tại thời điểm kết thúc sự kiện *endRead* sẽ mở khóa một tiến trình nếu biểu thức điều kiện của nó được thỏa mãn. Một tiến trình *reader* sẽ bị khóa nếu một vài tiến trình khác đang ghi write ($writers \neq 0$) hoặc đang chờ để ghi *write*. Một tiến trình ghi *write* sẽ bị khóa khi và chỉ khi ở tại cùng thời điểm tồn tại một vài tiến trình khác đang đọc ($readers \neq 0$) hoặc ghi ($writers \neq 0$) (Hình 3.9). Mục A.3 của Phụ lục A trình bày đặc tả cho vấn đề đọc-ghi với hai tiến trình đọc và ghi làm thay đổi giá trị của các biến *rd* và *wt*.

3.2.5 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, chi tiết của đặc tả được trình bày trong phần Phụ lục A. Bảng 3.1 thống kê kết quả của 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.

Với vấn đề vùng xung đột và đọc-ghi thì tất cả các mệnh đề cần chứng minh được sinh ra đều đã được chứng minh tự động. Với vấn đề 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. Hình 3.10 mô tả một sự kiện producer trong mô hình khởi tạo bên trái và mô hình làm mịn của nó bên phải. Bảng 3.2 mô tả một mệnh đề cần chứng minh để thỏa mãn bất biến của sự kiện trong Hình 3.10 được sinh ra và đã được chứng minh tự động. Bảng 3.3 mô tả một mệnh đề

machine ReaderWriterR
refines ReaderWriter
variables
readers
writers
OKtoRead
OKtoWrite
...
invariant
inv1 : $readers \in NAT$
inv2 : $writers \in NAT$
inv3 : $OKtoRead \in BOOL$
inv4 : $OKtoWrite \in BOOL$
...
init
act1 : $readers := 0$
act2 : $writers := 0$
act3 : $OKtoRead := true$
act4 : $OKtoWrite := false$
...

startRead
when
grd1 : $G1$
grd2 : $writers \neq 0$
grd3 : $OKtoWrite = false$
grd4 : $OKtoRead = true$
then
act1 : $readers := readers+1$
act2 : $isRead := true$
end

endRead
when
grd1 : $endOfRead = true$
then
act1 : $readers := readers-1$
act2 :**if** $readers = 0$
then $OKtoWrite := true$
endif
end

reader
when
grd1 : $isRead = true$
then
act1 : read the database
act2 : $endOfRead := true$
act3 : $OktoRead := false$
end

startWrite
when
grd1 : $G2$
grd2 : $writers \neq 0 \vee readers \neq 0$
grd3 : $OKtoWrite = true$
grd4 : $OKtoRead = false$
then
act1 : $writers := writers+1$
act2 : $isWrite := true$
end

endWrite
when
grd1 : $endOfWrite = true$
then
act1 : $writers := writers-1$
act2 : **if** $OKtoRead = false$
then $OKtoWrite := true$
else $OKtoRead := true$ **endif**
end

writer
when
grd1 : $isWrite = true$
then
act1 : write to the database
act2 : $endOfWrite := true$

end

Hình 3.9 – Máy làm mịn cho vấn đề đọc-ghi.

MACHINE ProducerConsumerI
Event *Producer* $\widehat{=}$
any
x
where
grd1 : $x \in dom(Queue)$
grd2 : $g = TRUE$
then
act1 : $Front := Front{+}1$
act2 : $Queue(Front) := x$
end

(a) Sự kiện producer trong mô hình khởi tạo

MACHINE ProducerConsumerR
REFINES ProducerConsumerI
Event *Producer2R* $\widehat{=}$
refines *Producer*
any
x
where
grd1 : $x \in dom(Queue)$
grd2 : $TurnP = 1$
grd4 : $g = TRUE$
grd5 : $Count < Queue\_Size$
then
act2 : $Queue(Front) := x$
act3 : $Count := Count{+}1$
act4 : $TurnP := 0$
end

(b) Sự kiện producer trong mô hình làm mịn

Hình 3.10 – Đặc tả sự kiện producer trong mô hình khởi tạo và làm mịn.

Bảng 3.2 – Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Producer đã được chứng minh tự động

$Fornt \in 0..Queue\_Size$ $Rear \in 0..Queue\_Size$ $x \in dom(Queue)$ $TurnP \in producer$ $producer \subseteq N1$ $g = TRUE$ $Queue \in N \rightarrow N$ $Count < Queue\_Size$ $\vdash$ $Queue \triangleleft \{Front \mapsto x\} \in N \rightarrow N$	Producer2R/inv2/INV

cần chứng minh được sinh ra để thỏa mãn bất biến của sự kiện trong Hình 3.10 và chưa được chứng minh tự động bằng công cụ RODIN. Mệnh đề này được chứng minh được bằng cách bổ sung thêm điều kiện $Front < Queue\_Size$. Các mệnh đề còn lại được chứng minh tương tự.

Bảng 3.3 – Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Producer chưa được chứng minh tự động

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

3.3 Kết luận

Trong chương này chúng tôi đã đề xuất một phương pháp để đặc tả và kiểm chứng ràng buộc về thứ tự thực hiện của các tiến trình tương tranh và áp dụng cho các vấn đề như vùng xung đột, đọc-ghi dữ liệu và cung cấp-tiêu thụ sử dụng phương pháp hình thức với Event-B. Trong phương pháp này, mỗi tiến trình được biểu diễn tương ứng với một sự kiện, mỗi vấn đề được đặc tả bằng mô hình trừu tượng biểu diễn các sự kiện và mô hình làm mịn của nó đặc tả sự thực hiện tương tranh của các sự kiện. Sự thực hiện tương tranh của các sự kiện dựa trên kỹ thuật đồng bộ hóa semaphore. Tính đúng đắn của đặc tả được bảo đảm thông qua việc sinh và chứng minh các mệnh đề cần chứng minh.

Phương pháp đã được đề xuất trong chương này tương tự như các phương pháp của Edmunds [42], Ben Younes [17] và Ball [15] về cách tiếp cận cùng sử dụng phương pháp hình thức với Event-B. Điểm khác nhau là các phương pháp của chúng tôi được áp dụng cho bài toán kiểm chứng ràng buộc về thứ tự thực hiện giữa các tiến trình. Vấn đề này đều chưa được giải quyết bởi các phương pháp nói trên. Hơn nữa, với phương pháp trong luận án thì mô hình hệ thống được đặc tả trực tiếp bằng Event-B, không phải xây dựng các đặc tả trung gian. Tuy nhiên, phương pháp này còn một số hạn chế như chưa tự động sinh mã chương trình Java từ đặc tả, sự song song của các sự kiện trong Event-B được thực hiện qua cơ chế đan xen.

Chương 4

Sự đồng thuận của hệ thống đa thành phần

4.1 Giới thiệu

Với các hệ thống phần mềm đa thành phần, mỗi thành phần thực hiện một vài hành vi xác định. Tập các thành phần trao đổi thông tin với nhau theo một giao thức tương tác (*interaction protocol*) xác định tạo nên hành vi tổng thể của hệ thống. Một hệ thống đa thành phần được gọi là đồng thuận (*consensus*) nếu thứ tự thực hiện của các thành phần tuân thủ các luật được định nghĩa trước (*giao thức tương tác*), các thành phần phải trả về kết quả mong muốn sau một số hữu hạn lần thực hiện.

Hiện nay đã có một vài phương pháp được đề xuất để đặc tả và kiểm chứng sự đồng thuận của hệ thống đa thành phần như các phương pháp Event-B [28, 79], JML. Danh sách 4.1 [39] biểu diễn đặc tả JML được nhúng vào mã Java cho thành phần `producer` của vấn đề cung cấp tiêu thụ được mô tả trong Chương 3. Với phương pháp đặc tả bằng JML yêu cầu chúng ta cần phải biết mã nguồn Java, sau khi hệ thống được cài đặt do đó bản thiết kế hệ thống có thể chưa được kiểm chứng.

```
public class Producer {
```

```
    /*@ spec_public rep readonly @*/ Product[] buf;
    /*@ spec_public @*/ int n;
    /*@ spec_public peer @*/ Consumer con;
    /*@ invariant buf != null && 0 <= n && n < buf.length &&
    @ (\forall int i; 0 <= i && i < buf.length;
    @ buf[i] == null || buf[i].owner == this.owner);
    @*/
 public Producer() {
   buf = new /*@ rep readonly @*/ Product[10];
 }
    /*@ requires con != null && con.n != n;
    @ ensures n == \old((n+1) % buf.length);
    @*/
 public void produce(/*@ peer @*/ Product p) {
    buf[n] = p;
    n = (n+1) % buf.length;
 }
}
```

Danh sách 4.1 – Đặc tả JML với mã Java của lớp Producer.

Trong chương này, chúng tôi đề xuất các phương pháp kiểm chứng sự đồng thuận của hệ thống đa thành phần từ pha thiết kế đến cài đặt mã Java tương tranh hoặc tương đương. Các kết quả chính của phương pháp được trình bày trong [76, 79].

Đề xuất phương pháp kiểm chứng sự đồng thuận của hệ thống tại mức thiết kế sử dụng phương pháp hình thức với Event-B. Trong phương pháp này, mỗi thành phần được đặc tả bằng một máy trừu tượng (*abstract machine*) tham chiếu đến ngữ cảnh (*context*) của nó. Các máy trừu tượng và ngữ cảnh sau đó được kết hợp với nhau theo một giao thức tương tác xác định. Sử dụng công cụ hỗ trợ của Event-B chúng tôi chứng minh tính đồng thuận của hệ thống đa thành phần này. Phương pháp đề xuất được minh họa qua một hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân. .

Đề xuất phương pháp kiểm chứng sự đồng thuận của hệ thống tại mức mã nguồn Java (*bytecode*). Với phương pháp này, chúng tôi xây dựng lớp `VMListener` trong JPF để kiểm chứng sự tuân thủ giữa cài đặt so với đặc tả thiết kế của nó. Phương pháp đề xuất được minh họa qua một hệ thống cung cấp tiêu thụ.

4.2 Một số định nghĩa và bổ đề

Trước khi trình bày các phương pháp đặc tả và kiểm chứng sự đồng thuận của hệ thống đa thành phần. Chúng tôi giới thiệu một số định nghĩa và bổ đề liên quan sau.

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. Khi điều kiện g của một sự kiện hội tụ ê không thỏa mãn thì chưa chứng minh được tính đồng thuận của hệ thống đa thành phần (*hay tính chất không bị tắc nghẽn - Deadlock*). Do đó, để chứng minh sự đồng thuận của hệ thống chúng tôi bổ sung một sự kiện mới $e' = \langle g', a'\rangle$ sao cho điều kiện $g \vee g'$ luôn được thỏa mãn với tập các hành động a hoặc a'.

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.

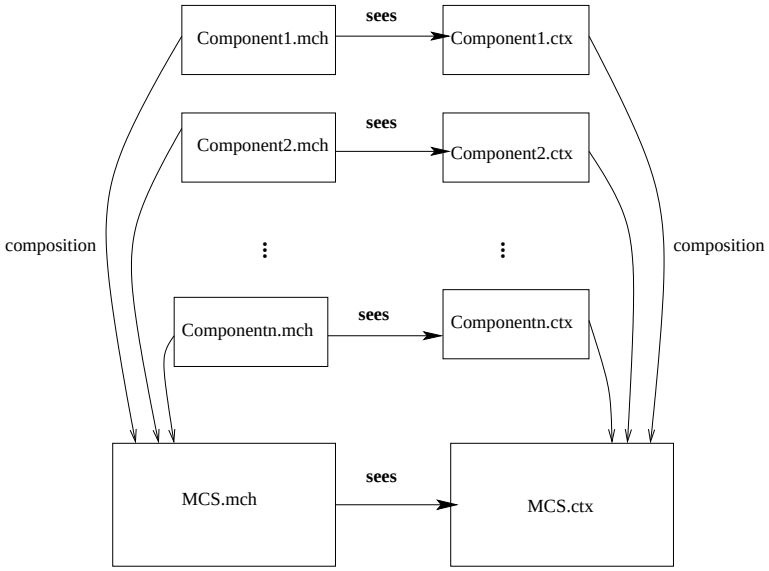
Bổ đề 4.5 (Sự kiện lấy giá trị đồng thuận)**.** *Nếu* $\hat{E} = S(e_i)$ *thì* $\exists\, e' = \langle g', a' \rangle$ *sao cho* $\bigvee g_i \vee g' = true$

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

Để phân tích sự đồng thuận của của hệ thống đa thành phần, trước hết chúng tôi xây dựng đặc tả hệ thống với Event-B, kiến trúc của đặc tả được biểu diễn 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`.



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

Sự kết hợp của các máy và ngữ cảnh trong Event-B đã được đề xuất trong [69] như sau. Giả sử M là mô hình kết hợp của hai mô hình $M1$ và $M2$. Danh sách biến v trong $M1$ gồm tập các biến hành động (*action variables*) x và biến nhảy (*skip variables*) y của mỗi sự kiện. Tương tự, danh sách biến w trong $M2$ gồm các biến hành động z và các biến nhảy a của mỗi sự kiện. Các biến hành động xz và biến nhảy ya của M được định nghĩa tương ứng là $xz = x \cap z$, $ya = y \cap a$ với $xz \cap ya = \phi$.

Ngoài ra, các kỹ thuật kết hợp trong Event-B phải bảo đảm tất cả các hành vi của mô hình trừu tượng phải được bảo toàn, và mô hình kết hợp không rơi vào các trạng thái sau.

1. Phân kỳ (*Divergence*) : xảy ra khi có các sự kiện bị hủy bỏ dẫn đến hành vi của hệ thống bị hỗn loạn,
2. Tắc nghẽn (*Deadlock*) : xảy ra khi không có sự kiện nào được kích hoạt. Trạng thái của hệ thống sẽ không bao giờ thay đổi khi bị tắc nghẽn.

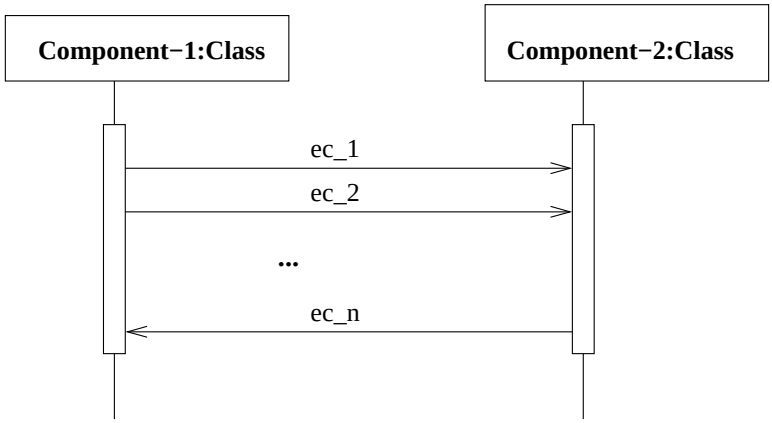
Ràng buộc thứ nhất (*không phân kỳ*) được bảo đảm bằng cách đưa vào một biến có cấu trúc V (*ví dụ N,$\leq$ đã được chứng minh là một quan hệ có cấu trúc*). Ràng buộc thứ hai (*không tắc nghẽn*) được chứng minh bằng các mệnh đề chứng minh, do các mệnh đề này được xây dựng từ phép tuyển các điều kiện của sự kiện nên luôn luôn được thỏa mãn theo các tính chất của hằng và bất biến. Hai ràng buộc này có thể được chứng minh tự động bằng công cụ hỗ trợ của Event-B.

Để tránh nhập nhằng, với các thành phần có nhiều chức năng được chúng tôi phân rã thành nhiều mô hình máy, mỗi chức năng tương ứng với một mô hình. Giả sử trong mô hình phân rã này, một máy trừu tượng đặc tả một chức năng của một thành phần được biểu diễn bằng một bộ bốn $M_i = \langle v_i, Init_i, ec_i, ee_i \rangle$. Trong đó, v_i là danh sách các biến, $Init_i$ là sự kiện khởi tạo của thành phần, ee_i, ec_i là danh sách các sự kiện đặc tả một chức năng của thành phần và ee_i là một sự kiện được sử dụng để lấy kết quả trả về của thành phần (Bổ đề 4.2). Giả sử $M = \langle V, Init, ec, ee, ee_M \rangle$ là một máy kết hợp biểu diễn khả năng của các thành phần M_i, $i = 1, \ldots, n$. Phụ thuộc vào giao thức tương tác giữa các thành phần chỉ chứa các sự kiện tuần tự hoặc có các sự kiện 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

Trong trường hợp giao thức chỉ gồm các sự kiện được thực hiện tuần tự. Không mất tính tổng quát, chúng tôi giả thiết thứ tự thực hiện của các sự kiện là $\Gamma \mathrel{\widehat{=}}$

$[ec_1, ec_2, ec_3, ..., ec_n]$ (*Hình 4.2 biểu diễn một giao thức tuần tự gồm n sự kiện tương tác với nhau bằng biểu đồ tuần tự trong UML*).



Hình 4.2 – Giao thức tuần tự được biểu diễn bằng UML.

Chúng tôi đề xuất các nguyên lý để xây dựng máy kết hợp `M` trong Event-B như sau :

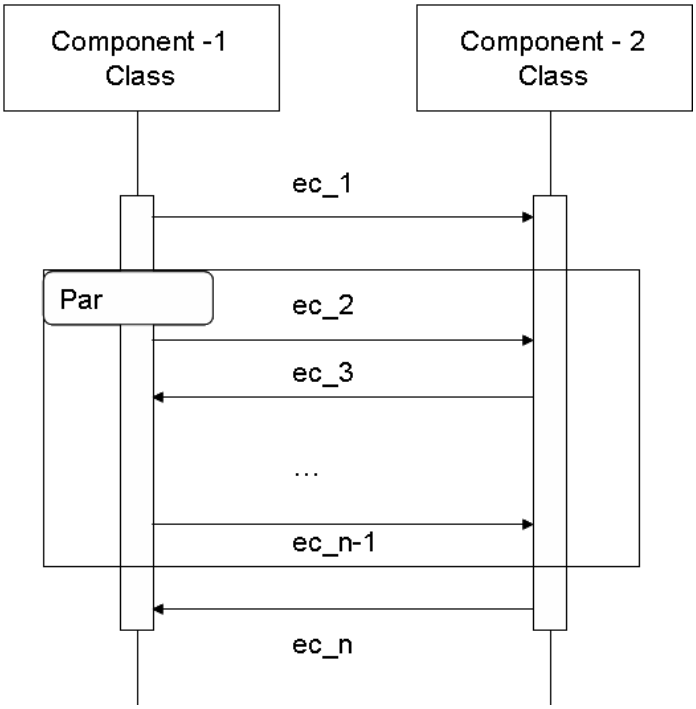
1. $V = \cup v_i$: danh sách các biến của máy kết hợp bao gồm các biến của các máy thành phần,
2. $ec = \cup ec_i$: máy kết hợp bao gồm tất cả các sự kiện của máy thành phần,
3. $Init = Init_1$: sự kiện khởi tạo *Init* của máy kết hợp được định nghĩa là sự kiện khởi tạo *Init* của máy thành phần đầu 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.

Sau khi kết hợp, chúng tôi tối ưu lại mô hình bằng cách loại bỏ các hằng và biến dư thừa hoặc không cần thiết. Các nguyên lý được đề xuất trên là hoàn toàn đúng đắn bởi vì các sự kiện được thực hiện tuân thủ theo giao thức tương tác. Sự kiện ec_1 được thực hiện trước thông qua định nghĩa sự kiện *Init* của máy kết hợp, sau đó các sự kiện tiếp theo sẽ được thực hiện thông qua các hành động được định nghĩa trong sự kiện lấy kết quả trả về của sự kiện trước đó.

4.3.3 Giao thức song song

Trong trường hợp giao thức tương tác có các sự kiện được thực hiện song song với nhau. Giả thiết giao thức của các thành phần Γ hình thức hóa như sau (*Hình 4.3 biểu diễn một giao thức song song gồm n sự kiện tương tác với nhau bằng biểu đồ tuần tự trong UML.*).

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



Hình 4.3 – Giao thức song song được biểu diễn bằng UML.

Trong mô hình Event-B các sự kiện sẽ được thực hiện song song nếu điều kiện của nó đồng thời được thỏa mãn. Do đó, một hệ thống đa thành phần sẽ đồng thuận khi mỗi sự kiện trong giao thức hội tụ (*dừng*)[1]. Tương tự như giao thức tuần tự, chúng tôi cũng bổ sung vào máy kết hợp một sự kiện để nhận kết quả trả về của giao thức khi nó. Quá trình xây dựng máy kết hợp M trong trường hợp này là sự

1. một sự kiện là hội tụ nếu nó trả về kết quả sau một số hữu hạn lần thực hiện các hành động

kết hợp giữa giao thức tuần tự và song song. Phần tuần tự được thực hiện tương tự như đã trình bày ở trên, phần song song được xây dựng theo nguyên lý sau.

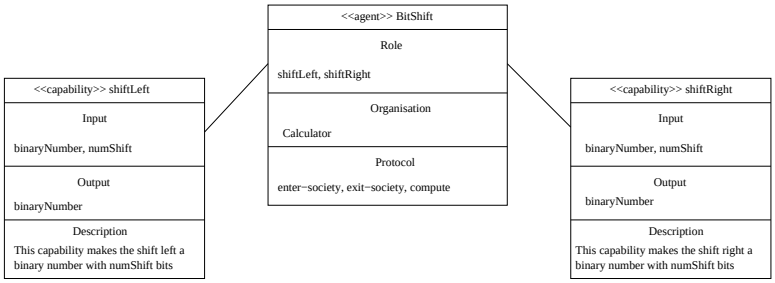
1. Từ sự kiện tuần tự được thực hiện trước đó, kích hoạt điều kiện của tất cả các sự kiện song song để cho các sự kiện này được thực hiện tại cùng một thời điểm,
2. Với mỗi sự kiện ee_i được thực hiện song song. Do các sự kiện này là hội tụ nên chúng tôi bổ sung một sự kiện ee_{is} để lấy kết quả trả 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.

Sự hội tụ của mỗi sự kiện và sự tương tác giữa các sự kiện sẽ được chứng minh tự động bởi công cụ hỗ trợ của Event-B.

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

Gỉa sử một hệ thống đa thành phần với các thành phần thực hiện các phép toán dịch bit `BitShift`, cộng và nhân hai số nhị phân `Sum`, `MultiDigit`. Trong đó phép nhân hai số nhị phân được thực hiện tương tự như phép nhân ở hệ đếm 10. Giả thiết hệ thống đa thành phần được xây dựng với ba phép toán `multiplyWithOneDigit` thuộc thành phần `MultiDigit`, `shiftLeft` thuộc thành phần `BitShift` và `addition` thuộc thành phần `Sum`. Với phép toán dịch bit `BitShift` thì các bit được dịch sang trái hoặc phải. Phép toán dịch trái `shiftLeft` thì một số bit sẽ được dịch sang trái và tương ứng là các bit 0 được chèn vào bên phải của số nhị phân bị dịch. Ví dụ nếu ta áp dụng phép dịch trái một bit với số 00011011 thì nhận được kết quả là 00110110. Toán tử `BitShift` được đặc tả bằng biểu đồ UML như trong Hình 4.4



Hình 4.4 – Đặc tả phép dịch bit trong UML.

Thành phần `Sum` được sử dụng để công hoặc trừ các số nhị phân, đầu vào của thành phần này là hai số nhị phân, đầu ra là số kết quả khi thực hiện phép toán. Ví dụ :

$$\begin{array}{rl} & 00011011 \\ + & \underline{00110110} \\ & 01010001 \end{array}$$

Thành phần `MultiDigit` được sử dụng để nhân một số nhị phân với một bit 0 hoặc 1. Đầu vào là một số nhị phân và một bit 0 hoặc 1, đầu ra là số kết quả.

4.3.4.2 Đặc tả hệ thống với Event-B

Để đặc tả hệ thống đa thành phần trên với Event B, trước hết chúng tôi sử dụng tập hợp để biểu diễn số nhị phân. Tập hợp này là một bộ hai thành phần, một phần dùng để biểu diễn vị trí của các bit trong số nhị phân, phần còn lại biểu diễn giá trị của các bit tương ứng với vị trí của nó. Ví dụ xâu nhị phân 00011011 được đặc tả qua tập hợp như sau :

$$\{8 \mapsto 0, 7 \mapsto 0, 6 \mapsto 0, 5 \mapsto 1, 4 \mapsto 1, 3 \mapsto 0, 2 \mapsto 1, 1 \mapsto 1\}$$

Với cách biểu diễn này và giao thức tương tác :
$\Gamma \mathrel{\widehat{=}} [multiplyWithOneDigit, shiftLeft, addition]$, (*nhân hai số nhị phân được biểu diễn bằng Thuật toán 4.1*). Chúng tôi đặc tả các thành phần `MultiDigit` bằng máy trừu tượng `MultiDigit.mch` biểu diễn phần động của hệ thống, máy trừu tượng này tham chiếu ngữ cảnh `MultiDigit.ctx` (xem phụ lục B).

Phép toán `shiftLeft` của thành phần `BitShift`, phép toán `addition` của thành phần `Sum` được trình bày trong phụ lục B. Với chú ý là mỗi sự kiện tương ứng với một phép toán, chúng tôi định nghĩa một sự kiện mới [*event_name*]_*result* để lấy giá trị hội tụ (*Bổ đề 4.2*).

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, Máy kết hợp của các máy thành phần trong hệ thống được biểu diễn trong Hình 4.5(a). Hình 4.5(b) biểu diễn ngữ cảnh kết hợp của các ngữ cảnh thành phần. Trong đó, chúng tôi bổ sung sự kiện *multiply2BinaryNumbers* để nhận giá trị trả về của hệ thống (Bổ đề 4.5).

Sự thực hiện của các sự kiện trong máy kết hợp MCS như sau. Trước hết, sự kiện `Init` được thực hiện. Trong sự kiện này, chúng tôi khởi tạo giá trị cho biến `jj := 1` để kích hoạt sự thực hiện của sự kiện `multiplyWithOneDigit`. Do sự kiện này đã được chứng minh là sự kiện hội tụ trong máy thành phần `MultiDigit` nên nó thực hiện đến khi điều kiện $jj \leq size\_aa$ không còn thỏa mãn. Từ đó suy ra điều kiện `jj = size_aa + 1` được thỏa mãn và do đó sự kiện `multiplyWithOneDigit_result` được kích hoạt. Theo định nghĩa của máy thành phần `MultiDigit` thì điều kiện sẽ không thay đổi với bất kỳ một sự kiện nào.
Tuy nhiên, sự kiện `multiplyWithOneDigit_result` trong máy `MCS.mch` bao hàm một phần của sự kiện `Init` của máy thành phần `BitShift`, như định nghĩa trong nguyên lý kết hợp ở trên. Định nghĩa này cho phép kích hoạt sự kiện `shiftLeft`,...

Quá trình này được lặp lại đến khi tất cả các sự kiện trong giao thức được kích hoạt. Nếu các sự kiện này không hội tụ thì nó sẽ lặp vô tận do đó việc chứng minh thuộc tính không phân kỳ bị vi phạm. Ngược lại, nếu nó hội tụ thì tuyển các điều kiện của các sự kiện liên quan sẽ không được thỏa mãn sau một số hữu

machine MCS.mch
sees MCS.ctx
variables
ii
jj
...
invariant
$ii \in NAT$
$jj \in NAT$
...
events
INIT
$ii := 1$
$jj := 1$
...
end
multiply2BinaryNumbers
when $(ii = size\_bb{+}1)$ **then**
$res := cc$
end
multiplyWithOneDigit
when $(jj \leq size\_aa)$ **then**
$pp(jj) := bb(ii) * aa(jj)$
$jj := jj{+}1$
end
multiplyWithOneDigit_result
when $jj = size\_aa{+}1$ **then**
$modr := pp$
$kk := size\_pp$ //activate shiftLeft
end
shiftLeft
when $(kk > 0 \wedge kk \leq size\_pp)$ **then**
if $(kk > numShift)$ **then**
$modr(kk) := modr(kk{-}numShift)$
else if $(kk \leq numshift)$ **then**
$modr(kk) := 0$
end
$kk := kk{-}1$
end
shiftLeft_result
when $(kk = 0)$ **then**
$slr := modr$
$hh := 1$ //activate the addition event
end
addition
when $(hh \leq size\_ar)$ **then**
if $(hh = size\_ar \wedge carry \neq 0)$ **then**
$cc(hh{+}1) := 1$
else
$cc(hh) := (cc(hh){+}slr(hh){+}carry)$ **mod** 2
$carry := (cc(hh){+}slr(hh){+}carry)$ **div** 2
$hh := hh{+}1$
end
end
addition_result
when $hh = size\_ar{+}1$ **then**
$ar := cc$
$jj := 1$
$ii := ii{+}1$
end
end

(a) MCS machine

context MCS.ctx
constants
aa
bb
$size\_aa$
$size\_bb$
$size\_res$
axioms
$aa \in NAT_1 \rightarrow 0..1$
$bb \in NAT_1 \rightarrow 0..1$
$0 < size\_aa$
$0 < size\_bb$
$size\_aa < size\_res$
$size\_bb < size\_res$
theorems
$\mathbf{ran}(aa) \neq \varnothing$
$\mathbf{ran}(bb) \neq \varnothing$
end

(b) MCS context

Hình 4.5 – Máy và ngữ cảnh của hệ thống.

Bảng 4.1 – Kết quả chứng minh 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

hạn bước thực hiện, từ đó suy ra điều kiện của sự kiện `multiply2BinaryNumbers` sẽ luôn được thỏa mãn dẫn đến các hành động của sự kiện này được thực hiện. Do các hành động này không chứa bất kỳ một biến (variant) nào nên điều kiện của sự kiện `multiply2BinaryNumbers` luôn thoả mãn. Kết quả là nếu các mệnh đề cần chứng minh của máy kết hợp được chứng minh thì sự thực hiện của các sự kiện sẽ hội tụ.

4.3.4.3 Kết quả chứng minh

Chúng tôi đã đặc tả và cài đặt hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân bằng công cụ RODIN của Event-B, chi tiết của đặc tả được trình bày trong phần Phụ lục B. Bảng 4.1 thống kê kết quả của 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. Ví dụ thành phần *Bitshiftt* được đặc tả bằng 4 sự kiện sẽ sinh ra 9 mệnh đề cần chứng minh, trong đó có 6 mệnh đề đã được chứng minh tự động bằng công cụ, các mệnh đề còn lại đã được chứng minh thủ công (*manual proving*). Các thành phần khác trong bảng được mô tả tương tự.

Hình 4.6 mô tả một sự kiện ShiftLeftIf của thành phần Bitshift. Bảng 4.2 mô tả một mệnh đề cần chứng minh để bảo đảm tính định nghĩa được của một hành động (*well-definedness of an event Action*) trong sự kiện ShiftLeftIf được sinh ra và đã được chứng minh tự động.

MACHINE BitShiftmch
SEES BitShiftctx
Event *ShiftLeftIf* $\widehat{=}$
when
grd1 : kk > 0
grd2 : kk > numShift
then
act1 : ppr(kk) := ppr(kk−numShift)
act2 : kk := kk−1
end

Hình 4.6 – Đặc tả sự kiện ShiftLeftIf của thành phần bitshift.

Bảng 4.2 – Mệnh đề cần chứng minh để bảo đảm tính định nghĩa được của sự kiện BitShiftLeftIf đã được chứng minh tự động

$kk-numShift \geq 0$ $kk-numShift \leq 1$ $numShift > 0$ $ppr \in N1 \rightarrow 0..1$ $kk > 0$ $kk > numShift$ $\vdash$ $kk-numShift \in dom(ppr)$	ShiftLeftIf/act1/WD

Bảng 4.3 mô tả một mệnh đề cần chứng minh được sinh ra để thỏa mãn bất biến và chưa được chứng minh tự động bằng công cụ RODIN. Mệnh đề này được chứng minh bằng cách bổ sung thêm tiên đề $kk-numShift \in dom(ppr)$. Kết quả chứng minh cho thấy hiện tại công cụ RODIN chưa hỗ trợ chứng minh tự động hoàn toàn, tuy nhiên các mệnh đề chưa được chứng minh tự động có thể được chứng minh bằng cách bổ sung các tiên đề hoặc chứng minh thủ công.

Bảng 4.3 – Mệnh đề cần chứng minh để bảo toàn bất biến của sự kiện Bit-ShiftLeftIf chưa được chứng minh tự động

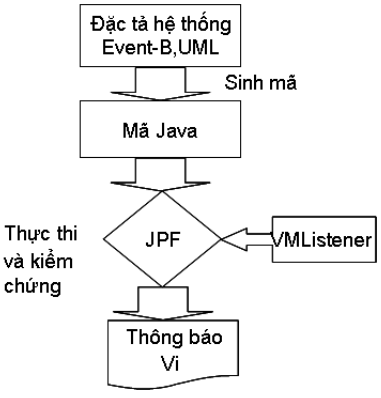
$ppr \in N1 \rightarrow 0..1$ $size\_pp > 0$ $numShift > 0$ $numShift < size\_pp$ $kk \in N$ $kk > 0$ $kk > numShift$ $\vdash$ $ppr \lhd \{kk \rightarrow ppr(kk - numShift)\} \in N1 \rightarrow 0..1$	ShiftLeftIf/inv4/INV

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 4.7).

- Bản thiết kế hệ thống đa thành phần 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 4.7 – Phương pháp kiểm chứng sự đồng thuận tại mức mã nguồn.

Giả sử bản thiết kế hệ thống là đúng đắn, người lập trình sau đó cài đặt mã Java theo bản thiết kế. Tuy nhiên, việc cài đặt mã Java có thể phát sinh các lỗi không tuân thủ theo đặc tả thiết kế của nó. Để giải quyết vấn đề này, chúng tôi sử dụng bộ công cụ JPF đóng vai trò như một máy ảo để thực thi các chương trình Java, lớp `VMListener` được thiết kế để đặc tả các thuộc tính cần kiểm tra. JPF sẽ phân tích tất cả các đường đi có thể trong chương trình để tìm ra các vi phạm nếu có.

4.4.2 Sinh mã kiểm chứng trong JPF

Chúng tôi đề xuất thuật toán 4.2 để sinh mã kiểm tra cho lớp khuôn mẫu *VMListener* trong JPF. Trong đó, các biến *St_Start* và *St_Final* được sử dụng để đánh dấu phương thức khởi tạo và phương thức kết thúc trong giao thức Γ. Thuật toán sử dụng một biến *State* và tự động sinh ra một biến trạng thái có tiền tố là *St_* theo sau là tên các phương thức khi nó được thực hiện. Để kiểm tra thứ tự thực hiện trước khi một phương thức được thực hiện thì biến trạng thái *State* sẽ được so sánh với biến trạng thái của phương thức được thực hiện trước đó. Nếu nó không thoải mãn thì dừng và thông báo vi phạm đặc tả, ngược lại sau khi thực hiện xong biến *State* được thay đổi thành biến trạng thái của phương thức đã được thực hiện. Thuật toán sẽ dừng khi biến *State* đạt được trạng thái đích và thông báo sự đồng thuận của hệ thống hoặc gặp trạng thái lỗi.

4.4.3 Hệ thống cung cấp tiêu thụ

Vấn đề cung cấp-tiêu thụ (*producer-consumer problem*) với giao thức song song $\Gamma \widehat{=} [init, producer \mid\mid consumer, close]$ được đặc tả trong Hình 3.4, Chương 3. Dựa vào thuật toán 4.2 và giao diện của lớp *VMListener* [4], chúng tôi xây dựng mã để kiểm chứng sự tuân thủ giữa bản cài đặt chương trình và đặc tả của nó. Hình 4.8 mô tả kết quả kiểm chứng cho hệ thống cung cấp tiêu thụ với bộ kiểm chứng mô hình JPF được thực thi trên nền Eclipse. Trong đó, tại cột bên phải mô tả mã chương trình Java của hệ thống. Kết quả kiểm chứng được biểu diễn trong các cột bên trái bao gồm các thông tin về số lỗi vi phạm đặc tả, vết của lỗi (*trace*) và

Thuật toán 4.2 Sinh mã cho lớp VMListener trong JPF

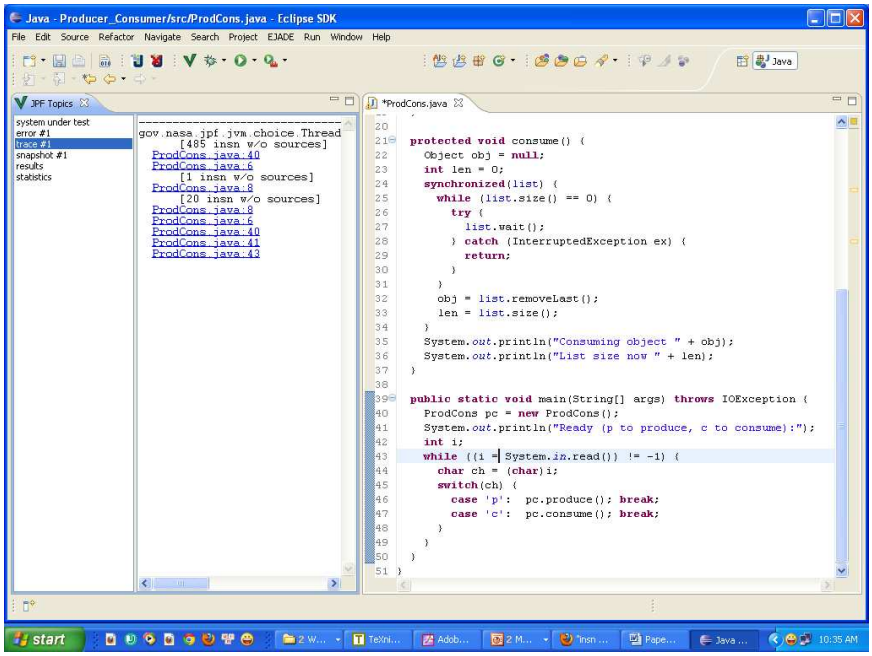
Require: Đặc tả hệ thống đa thành phần với giao thức Γ
Ensure: Mã kiểm chứng cho *VMListener* trong JPF
  Khởi tạo biến trang thái khởi tạo và kết thúc *St_Start*, *St_Final*
  State = *St_Start* { Biến State kiểm tra thứ tự thực hiện theo Γ }
  while *State* $\neq$ *St_Final* **do**
    for mỗi phương 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

không gian trạng thái được duyệt. 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.

4.5 Kết luận

Trong chương này, luận án đã đề xuất các phương pháp đặc tả và kiểm chứng sự đồng thuận của hệ thống đa thành từ mức thiết kế đến cài đặt mã nguồn chương trình.

Phương pháp kiểm chứng thiết kế sử dụng phương pháp hình thức với Event-B, mỗi thành phần được đặc tả bằng máy trừu tượng tham chiếu đến một ngữ cảnh



Hình 4.8 – Kiểm chứng mã nguồn hệ thống cung cấp-tiêu thụ với JPF.

của nó. Sự tương tác giữa các thành phần được đặc tả qua một giao thức tương tác hoặc thuật toán xác định. Các giao thức hoặc thuật toán này sẽ làm thay đổi trạng thái của các sự kiện. Các máy thành phần và ngữ cảnh của nó sau đó sẽ được kết hợp lại thành một dạng máy tổng quát của hệ thống. Sau đó chúng tôi sử dụng công cụ của Event-B để hình thức và phân tích sự đồng thuận của hệ thống thông qua máy kết hợp. Chúng tôi đã đề xuất các quy tắc để đặc tả giao thức tương tác (*bao gồm giao thức tuần tự và song song*). Phương pháp này được minh họa thông qua một hệ thống đa thành phần thực hiện các phép toán trên tập các số nhị phân. Trong hệ thống này, kết quả của phép nhân hai số nhị phân được thực hiện qua sự tương tác giữa các thành phần nhân một bit `multiplyWithOneDigit`, dịch trái `ShiftLeft` và cộng `Sum`. Chúng tôi đã chứng minh sự đồng thuận của hệ thống đa thành phần này với sự hỗ trợ của công cụ chứng minh RODIN .

Trong phương pháp kiểm chứng mức mã nguồn, chúng tôi mở rộng bộ công cụ kiểm chứng mô hình JPF để kiểm chứng sự tuân thủ giữa bản cài đặt của chương trình so với đặc tả thiết kế của nó. Phương pháp này sử dụng các biểu đồ UML hoặc Event-B để đặc tả hệ thống, dựa vào các đặc tả này, chúng tôi sinh mã cho giao diện của lớp `VMListener` trong JPF để kiểm chứng sự tuân thủ giữa mã

chương trình Java và đặc tả thiết kế của nó. Phương pháp này đã được minh họa qua hệ thống cung cấp-tiêu thụ.

Tuy nhiên, các phương pháp đã đề xuất chưa được thực nghiệm với các hệ thống lớn gồm nhiều thành phần. Trong tương lai chúng tôi sẽ tiếp tục thử nghiệm với các hệ thống lớn, cài đặt công cụ hỗ trợ sinh mã Java từ đặc tả Event-B.

Chương 5

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

5.1 Giới thiệu

Trong các chương trình hướng đối tượng, tương tranh giao thức tương tác (*interaction protocol*) đặc tả các ràng buộc về thứ tự thực hiện của các phương thức trong các lớp hay các thành phần phải được thỏa mãn tại thời điểm thực thi chương trình. Mỗi khi giao thức tương tác bị vi phạm thì có thể gây ra các lỗi hệ thống. Tuy nhiên, các giao thức này được định nghĩa ẩn và không được kiểm tra tại quá trình biên dịch (Định nghĩa 3.1, Chương 3).

Hiện nay, có hai phương pháp kiểm chứng tĩnh [58] và động (*kiểm chứng tại thời điểm thực thi*) [32, 34, 50] để kiểm tra sự tuân thủ của giao thức tương tác. Mỗi phương pháp đều có các ưu và nhược điểm riêng. Trong đó, các phương pháp phân tích tĩnh thường tốt hơn các phương pháp động do các vi phạm được phát hiện sớm mà không phải thực thi chương trình. Tuy nhiên, các phương pháp động có thể phát hiện được các vi phạm giao thức trong các chương trình tương tranh mà các phương pháp tĩnh chưa thể phát hiện được.

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. Các kết quả chính của phương pháp được trình bày trong [21, 75, 78].

Các phần còn lại của chương này đươc cấu trúc như sau. Mục 5.2 giới thiệu bài toán kiểm chứng sự tương tác giữa các thành phần trong chương trình tương tranh. Mục 5.3 trình bày phương pháp giải quyết bài toán sử dụng AOP. Các kết quả thực nghiệm và kết luận được trình bày trong Mục 5.4 và 5.5.

5.2 Bài toán 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 3.4, Chương 3). 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. Khi đó bài toán kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác trong các chương trình tương tranh 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 3.4 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.

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

5.3.1 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).

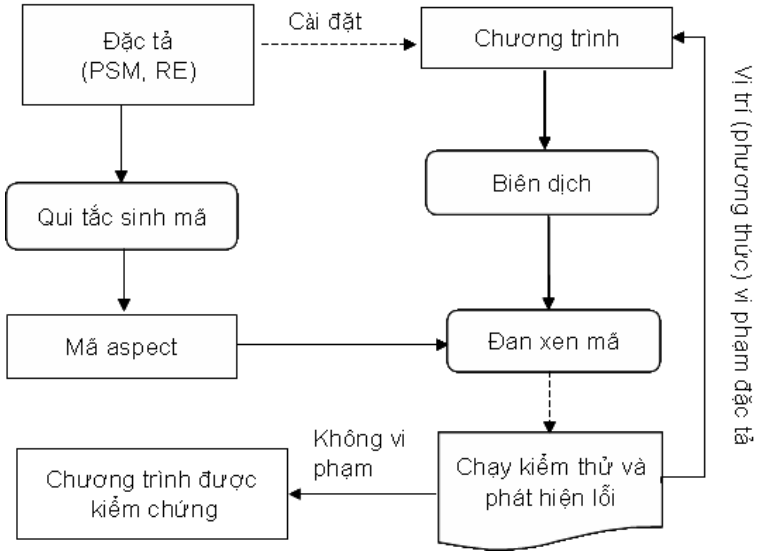
1. Sử dụng biểu thức chính quy mở rộng (RE) hoặc máy trạng thái giao thức (PSM) để đặc tả giao thức tương tác (IPC),
2. Người lập trình cài đặt các ứng dụng dựa trên các đặc tả IPC,
3. Các mã aspect sinh ra được tự động đan với mã của các chương trình ứng dụng để kiểm chứng động sự tuân thủ giữa thực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

5.3.2 Đặc tả giao thức tương tác

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

Chúng tôi mở rộng biểu thức chính qui [29] để biểu diễn IPC được định nghĩa như sau.

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,* $M = \{m_1, m_2, ..., m_n\}$ *là bảng chữ cái Sigma gồm một tập hữu hạn các phương thức,* $O = \{o_1, o_2, ..., o_p\}$



Hình 5.1 – Sơ đồ hoạt động của hệ thống.

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.*

Ví dụ, giao thức của hàng đợi tương tranh trong Hình 3.4 được biểu diễn bằng biểu thức chính quy mở rộng sau : $init(Q) \rightarrow ([Q.x = True]dequeue(Q, x)[Q.x = False] \mid [Q.Full = False]enqueue(Q, x)[Q.x = True]) \rightarrow close(Q)$.

Với AOP chúng ta có thể sử dụng các ký tự đại diện cho phương thức cùng với các tham số của nó. Trong đó, `"*"` đại diện cho một ký tự bất kì và `".."` đại diện cho tất cả các tham số. Ví dụ `st*(..)` sẽ đại diện cho các phương thức bắt đầu bởi hai ký tự `st`, các tham số là bất kì.

Dựa vào đặc trưng này, chúng tôi đề xuất sử dụng ký tự đại diện trong các biểu thức chính quy suy rộng để đặc tả giao thức tương tác. Dễ thấy các đặc tả này sẽ ngắn gọn hơn so với các phương pháp khác như trong [33, 37, 58].

Ví dụ giao diện của lớp `BytesMessage` trong thư viện J2EE của Java có khoảng mười hai phương thức đọc và ghi ứng với các kiểu dữ liệu khác nhau. Giao thức tương tác của nó được đặc tả bằng biểu thức chính quy [33], automát [58] sẽ dài hơn so với biểu thức chính quy suy rộng sử dụng các ký tự đại diện. Một phần của đặc tả cho giao thức trên bằng biểu thức chính quy như trong Danh sách 5.1.

```
void acknowledge():
{
 boolean readBoolean() |
 byte readByte() |
 int readBytes(byte[]) |
 int readBytes(byte[], int) |
 char readChar() |
 double readDouble() |
 float readFloat() |
 int readInt() |
 long readLong() |
 short readShort() |
 int readUnsignedByte() |
 int readUnsignedShort() |
 java.lang.String readUTF()
}+
:void clearBody()
```

Danh sách 5.1 – Đặc tả RE cho giao thức của lớp J2EE.

Đặc tả bằng biểu thức chính quy suy rộng với các ký tự đại diện như sau.

```
* a*(): {* r*(..)}* : * c*().
```

5.3.2.2 Biểu đồ PSM cho biểu diễn giao thức tương tác

Biểu đồ PSM trong UML2.0 [27, 45] 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 IPC. Chúng tôi định nghĩa hình thức như sau :

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.*

Hình 3.4 biểu diễn biểu đồ PSM cho giao thức tương tác của hàng đợi tương tranh, thứ tự thực hiện của các phương thức được thể hiện bằng các cung trong biểu đồ. Trong đó $S = \{OPEN, CLOSE\} \cup \{s_0, f\}$, $Pre = Q.x = True, Q.Full = False$, $Post = \{Q.x = False, Q.x = True\}$, $M = \{init(Q), dequeue(Q, x), enqueue(Q, x),$

$close(Q)\}$, $\delta = \{s_0\ init(Q) \rightarrow OPEN, OPEN[Q.x = True]dequeue(Q, x)[Q.x = False] \rightarrow OPEN,\ OPEN[Q.Full = False]enqueue(Q, x)[Q.x = True] \rightarrow OPEN,$

$OPEN\ close(Q) \rightarrow f\}$.

5.3.3 Sinh mã aspect

Mục này trình bày thuật toán tự động sinh mã kiểm chứng aspect từ đặc tả IPC. 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 $a \in \Sigma$ (*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 [51]. Mã aspect sau đó được sinh ra tự động từ các đặc tả PSM và FSM.

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";
```

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. 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 IPC.

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 (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 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.

Danh sách 5.2 – Khởi tạo mẫu aspect.

Với aspect mẫu trong Danh sách 5.2, chúng tôi khai báo tên của aspect là *ProtocolCheck* và *import* các thư viện của AspectJ để đan xen mã, 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";
```

Danh sách 5.3 – Khởi tạo mẫu pointcut.

Trong pointcut mẫu Danh sách 5.3 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;
}
```

Danh sách 5.4 – Sinh biểu thức tiền và hậu điều kiện.

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 Thuật toán 5.1*). Các biểu thức này được đọc trực tiếp từ đặc tả và đưa vào pointcut mẫu trong bước 2.

5.3.4 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.

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 mã (.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.

5.4 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. Chi tiết về công cụ được trình bày trong Phụ lục C.

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 $open() \rightarrow (read()^{\cdot} \rightarrow close())$ 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.

Hình 5.2 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 tương tranh 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.

```
public class Mytest extends thread
{
 private int num;
 public void run() {
   try {
     InputStream f=new FileInputS-
tream("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();
 }
}
```

(a) Chương trình đúng

```
public class Mytest extends thread
{
 private int num;
 public void run() {
   try {
     InputStream f=new FileInputS-
tream("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 được 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();
 }
}
```

(b) Chương trình sai

Hình 5.2 – Ví dụ các chương trình được cài đặt đúng và sai.

Bên cạnh giao thức này, chúng tôi cũng đã thử nghiệm với các giao thức khác trong [34, 37, 57, 58, 78]. 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 5.1.

Bảng 5.1 – Thực nghiệm kiểm chứng sự tuân thủ giữa thực thi và đặc tả giao thức tương tác

Lớp (Java)	Số p.thức	Số test đúng/sai	Tỷ lệ phát hiện(%)	Tỷ lệ gia tăng thời gian(%s)
Applet	5	10/20	100	0.915
StreamReader	6	5/15	100	0.923
ReadWrite	4	6/10	100	0.974
Iterator	3	2/3	100	0.533
Stack	5	2/5	100	0.915
LinkedList	9	5/15	100	1.542
ConcurrentQueue	4	3/5	100	0.974
Roster	2	2/2	100	0.323

Trong đó, mỗi lớp trong cột 1 bên trái của Bảng 5.1 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, xem Hình 5.2*). 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 $\tau = \frac{\sum_{i=1}^{n} \mid ts_i - tt_i \mid}{n} \times 100\%$. 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 5.1. Đối với các giao thức mô tả trong cột 1 của Bảng 5.1 thì 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.

5.5 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 chương này, luận án 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 tương tranh. 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 tương tranh. 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.

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. 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 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 đã phát hiện được các vi phạm ràng buộc thời gian giữa các thành phần phần mềm so với đặc tả. Các kết quả chính của phương pháp này được trình bày trong [22, 74].

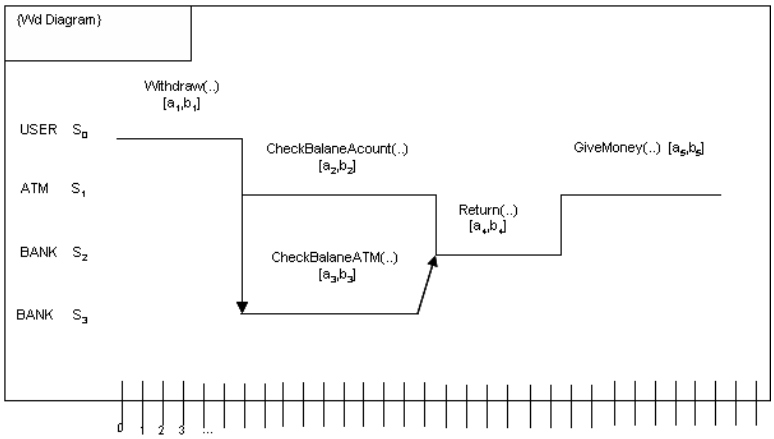
Các phần còn lại của chương này được cấu trúc như sau. Mục 6.2 trình bày bài toán kiểm chứng ràng buộc thời gian giữa các thành phần tương tranh. Phương pháp kiểm chứng ràng buộc thời gian giữa các thành phần sử dụng AOP được trình bày trong Mục 6.3. Mục 6.4 trình bày một số kết quả thực nghiệm, cuối cùng là các kết luận và hướng phát triển tiếp theo của phương pháp được đề xuất.

6.2 Bài toán kiểm chứng ràng buộc thời gian giữa các thành phần tương tranh

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 6.1).

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 vợt 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.

Một cách tổng quát mỗi sự kiện E_i được gán với đoạn thời gian thực thi $[t_{i_1}, t_{i_2}]$ với $t_{i_1} \leq t_{i_2}$. Thứ tự thực hiện của các sự kiện này được định nghĩa như sau.



HÌNH 6.1 – Biểu đồ thời gian của giao thức rút tiền.

Định nghĩa 6.1 (Hai sự kiện tuần tự)**.** *Hai sự kiện E_1 và E_2 được thực hiện tuần tự nhau khi và chỉ khi $t_{2_1} \geq t_{1_2}$ hoặc $t_{1_1} \geq t_{2_2}$.*

Định nghĩa 6.2 (Hai sự kiện đan xen)**.** *Hai sự kện E_1 và E_2 đan xen nhau khi và chỉ khi $t_{1_1} \in [t_{2_1}, t_{2_2}]$ hay $t_{1_2} \in [t_{2_1}, t_{2_2}]$ hoặc ngược lại $t_{2_1} \in [t_{1_1}, t_{1_2}]$ hoặc $t_{2_2} \in [t_{1_1}, t_{1_2}]$*

Định nghĩa 6.3 (Hai sự kiện song song và phủ nhau)**.** *E_1 và E_2 song song và phủ nhau khi và chỉ khi $t_{1_1}, t_{1_2} \in [t_{2_1}, t_{2_2}]$ hoặc $t_{2_1}, t_{2_2} \in [t_{1_1}, t_{1_2}]$.*

Định nghĩa 6.4 (Hai sự kiện song song tại cùng một thời điểm)**.** *Hai sự kiện E_1 và E_2 được thực hiện song song tại cùng một thời điểm khi và chỉ khi $t_{1_1} = t_{2_1}$.*

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

6.3.1 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. 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.

6.3.2 Đặc tả ràng buộc thời gian

Trong mục này chúng tôi 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 6.5 (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 6.1 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 6.6 (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 6.5). 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 6.6)*. τ`(Withdraw(..))` + τ`(CheckBalanceAccount(..))` + τ`(Return(..))` + τ`(GiveMoney(..))` ≤ 55.

Định nghĩa 6.7 (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* $i, j = 1..n, i \neq j$*.*

Giả sử hai thành phần `CheckBalanceAccount(..)` và `CheckBalanceATM(..)` được thực hiện song song tại cùng một thời điểm (*Hình 6.1*). 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 6.7)*. τ`(CheckBalanceAccount(..))` $>$ τ`(CheckBalanceATM(..))`.

6.3.2.1 Biểu thức chính quy thời gian

Biểu thức chính quy thời gian *(Timed Regular Expression -TRE)* [13] 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 6.8 (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.*

Ví dụ biểu đồ thời gian trong Hình 6.1 được biểu diễn bằng một biểu thức chính quy thời gian TRE sau :

$USER.Withdraw(..)[a_1, b_1] \rightarrow (ATM.CheckBalaneAccount(..)[a_2, b_2] \parallel BANK.CheckBalaneATM(..)[a_3, b_3]) \rightarrow BANK.Return(..)[a_4, b_4] \rightarrow ATM.GiveMoney(..)[a_5, b_5]$. 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 $[a_5, b_5]$.

6.3.2.2 Biểu đồ thời gian

Biểu đồ thời gian *(Timing Diagram - TD)* trong UML2.0 [27, 45] 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 6.9 (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 6.1 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},
- $\delta = \{S_0.USER.Withdraw[a_1, b_1] \rightarrow S_1, S_0.USER.Withdraw[a_1, b_1] \rightarrow S_3, S_1.ATM.CheckBalaneAccount[a_2, b_2] \rightarrow S_2, S_3.BANK.CheckBalaneATM[a_3, b_3] \rightarrow S_2, S_2.BANK.Return[a_4, b_4] \rightarrow S_1, S_1.ATM.GiveMoney[a_5, b_5] \rightarrow F\}$.

6.3.3 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 6.2. 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, xem phụ lục C)*. 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 6.5, 6.6 và 6.7 được dịch thành các biểu thức điều kiện trong aspect mẫu *(dòng 8)*.

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* ;

}

Hình 6.2 – Sinh mã aspect từ các đặc tả ràng buộc thời gian.

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. Aspect có thể được đan ở ba mức khác nhau : mức mã nguồn, mã bytecode hoặc 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 (*load time weaving*) 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 (xem Mục 5.3.4, Chương 3).

6.4 Thực nghiệm

Chúng tôi đã cài đặt phương pháp này và tích hợp với bộ công cụ kiểm chứng PVG *(Protocol Verification Generator-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. Chi tiết về công cụ được trình bày trong Phụ lục C. Thực nghiệm được tiến hành với các chương trình mô phỏng của hệ thống ATM, Hình 6.1. 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 PVG 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 6.5, 6.6 và 6.7. Với mỗi thành phần chúng tôi xây dựng các ca kiểm thử đúng, sai khác nhau. Trong đó, các ca kiểm thử đúng thì các thành phần được cài đặt tuân thủ theo đặc tả ràng buộc thời gian và ngược lại.

```
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

Hình 6.3 – Ví dụ ca kiểm thử đúng và sai của phương thức withdraw với ràng buộc thời gian thực thi [726082, 143658] nano giây.

Hình 6.3 mô tả một ca kiểm thử 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 ca kiểm thử 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 ca kiểm thử 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 ca kiểm thử đúng và sai với các đặc tả ràng buộc thời gian thực thi theo Định nghĩa 6.5 (*cột 2, Bảng 6.1*). Trong đó, ràng buộc thời gian của các thành phần tuần tự tăng dần từ 50ns đến 500ns (*theo Định nghĩa 6.6*). Ràng buộc thời gian kết thúc trước/sau giữa hai thành phần song song CheckBalanceAccount và CheckBalanceATM ở thời điểm bắt đầu t và $t \pm \xi ns$(*theo Định nghĩa 6.7*), với $\xi = 5, 10, 15, \ldots, 50$. Kết quả thực nghiệm cho thấy phương pháp đã phát hiện được đầy đủ các ca kiểm thử đúng và sai *(cột 4, Bảng 6.1)* với 25 ca kiểm thử đúng các vi phạm về ràng buộc thời gian được phát hiện chính xác (Bảng 6.1).

Qua các kết quả thực nghiệm cho thấy :

1. các aspect được sinh ra đúng so với các đặc tả ràng buộc thời gian, nhất quán giữa biểu thức chính quy và biểu đồ thời gian,
2. các aspect không làm thay đổi hành vi của chương trình gốc,
3. đã phát hiện được các vi phạm ràng buộc thời gian giữa các thành phần.

6.5 Kết luận

Trong nhiều hệ thống phần mềm, đặc biệt như với các hệ thống an toàn-bảo mật, hệ thống thời gian thực thì 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 cải tiến sự tin cậy về ràng buộc thời gian trong các ứng dụng phần mềm. Trong chương này, luận án đề xuất một phương pháp kiểm chứng sự tuân thủ giữa sự

Bảng 6.1 – Thực nghiệm kiểm chứng các ràng buộc thời gian

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

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 *(Timed Regular Expression)* để đặc tả các ràng buộc thời gian. 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 ràng buộc thời gian giữa các thành phần, trong đó có các thành phần được thực hiện song song. Các vi phạm được phát hiện trong bước kiểm thử tích hợp hệ thống.

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 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 thời gian thực thi và kích cỡ của chương trình được kiểm chứng.

Trong tương lai, chúng tôi sẽ kết hợp phương pháp này với phương pháp của Dymek [41] 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.

Chương 7

Kết luận

7.1 Các đóng góp của luận án

Các chương trình tương tranh gồm nhiều tiến trình cộng tác với nhau để cùng thực hiện một nhiệm vụ. Sự cộng tác giữa các tiến trình thường được thực hiện thông qua các biến chia sẻ hoặc cơ chế truyền thông điệp. Thiết kế, cài đặt và kiểm chứng các chương trình tương tranh thường khó khăn hơn so với các chương trình tuần tự do khả năng thực hiện của các chương trình này. Trong luận án này, chúng tôi đã đóng góp hai kết quả chính trong việc kiểm chứng các chương trình Java tương tranh từ pha thiết kế đến cài đặt mã nguồn chương trình, 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 phương pháp hình thức với 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 phương pháp lập trình hướng khía cạnh.

Với phương pháp kiểm chứng thiết kế, chúng tôi đã đề xuất phương pháp đặc tả và kiểm chứng ràng buộc về thứ tự (*giao thức*) thực hiện của các tiến trình tương tranh sử dụng phương pháp hình thức với Event-B. Với kiến trúc này mỗi tiến trình được đặc tả bằng một sự kiện của máy trừu tượng, ràng buộc về thứ tự

giữa các tiến trình được điều khiển theo cơ chế semaphore và làm mịn dần của các máy trừu tượng. Tính đúng đắn của đặc tả được bảo đảm bằng việc sinh và chứng minh tự động các mệnh đề cần chứng minh. Chúng tôi đã thử nghiệm đặc tả và cài đặt phương pháp này cho các vấn đề trong chương trình tương tranh như vùng xung đột, cung cấp-tiêu thụ và đọc ghi dữ liệu từ bộ nhớ chia sẻ. Phương pháp được đề xuất có ý nghĩa trong việc bảo đảm tính đúng đắn của các mô hình đặc tả tương tranh bằng một phương pháp hình thức, qua đó sinh ra mã của các chương trình Java tương tranh. Tuy nhiên, phương pháp này còn tồn tại một số vấn đề chưa giải quyết được như sinh mã Java tự động từ đặc tả của nó, Event-B chưa hỗ trợ ký pháp để đặc tả song song, sự tương tranh của các tiến trình được thực hiện theo cơ chế đan xen. Hơn nữa phương pháp này mới chỉ giải quyết được một lớp các bài toán về tương tranh đó là điều khiển sự thực thi của các tiến trình tuân theo một giao thức xác định.

Với các hệ thống phần mềm đa thành phần, mỗi thành phần thực hiện một vài hành vi xác định. Tập các thành phần trao đổi thông tin với nhau theo một giao thức tương tác (*interaction protocol*) xác định tạo nên hành vi tổng thể của hệ thống. Một hệ thống đa thành phần được gọi là đồng thuận (*consensus*) nếu thứ tự thực hiện của các thành phần tuân thủ các luật được định nghĩa trước (*giao thức tương tác*), các thành phần phải trả về kết quả mong muốn sau một số hữu hạn lần thực hiện. Chúng tôi đã đề xuất một phương pháp để đặc tả và kiểm chứng sự đồng thuận của hệ thống đa thành phần sử dụng phương pháp hình thức Event-B. Trong phương pháp này, mỗi thành phần được đặc tả bằng một máy trừu tượng (*abstract machine*) tham chiếu đến ngữ cảnh (*context*) của nó. Các máy trừu tượng và ngữ cảnh sau đó được kết hợp với nhau theo một giao thức tương tác xác định thành máy và ngữ cảnh tổng quát của hệ thống. Sự đồng thuận của hệ thống đa thành phần được chứng minh tự động thông qua máy và ngữ cảnh kết hợp của hệ thống. Phương pháp đề xuất được minh họa qua một hệ thống đa thành phần thực hiện các phép toán trên tập số nhị phân. Để kiểm chứng sự đồng thuận của hệ thống tại mức mã nguồn, chúng tôi bổ sung phương pháp kiểm chứng sự tuân thủ giữa bản cài đặt của chương trình so với thiết kế của nó sử dụng bộ công cụ kiểm chứng mô hình JPF. Phương pháp này được thử

nghiệm cho các chương trình Java minh họa một hệ thống cung cấp tiêu thụ. Kết quả cho thấy phương pháp được đề xuất có thể bảo đảm tính đồng thuận của hệ thống đa thành phần từ pha thiết kế đến cài đặt mã nguồn chương trình. Tuy nhiên, phương pháp này còn hạn chế như chưa được thử nghiệm với các hệ thống đa thành phần có quy mô lớn, tự động sinh mã nguồn chương trình từ đặc tả thiết kế của nó.

Để giải quyết bài toán kiểm chứng sự tuân thủ giữa mã thực thi của chương trình so với đặc tả ràng buộc thiết kế của nó, đặc biệt là các ràng buộc về thứ tự thực hiện của các phương thức (*giao thức tương tác*) và ràng buộc thời gian trong các chương trình hướng đối tượng, tương tranh. Chúng tôi đã đề xuất hai phương pháp động sử dụng lập trình hướng khía cạnh để kiểm chứng sự tuân thủ của chương trình so với đặc tả giao thức tương tác và đặc tả ràng buộc thời gian của nó. Các phương pháp này sử dụng máy trạng thái giao thức, biểu đồ tuần tự của UML và biểu thức chính quy để đặc tả các ràng buộc. 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 chương trình và đặc tả của nó, các vi phạm được phát hiện tại bước kiểm thử. Với các phương pháp kiểm chứng động chúng tôi đã cài đặt thành công cụ kiểm chứng PVG và chạy thực nghiệm với một số ứng dụng viết trên ngôn ngữ lập trình Java. Với việc sử dụng ngôn ngữ lập trình hướng đối tượng hiện đại như Java [47, 48] và ngôn ngữ mô hình hóa trực quan UML làm đặc tả cho thấy giá trị thực tiễn của các phương pháp được đề xuất. Hạn chế của các 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.

Các đóng góp của luận án 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 mã nguồn chương trình.

7.2 Hướng phát triển

Trong luận án này, chúng tôi đã đề xuất các phương pháp và xây dựng công cụ để đặc tả và kiểm chứng các chương trình tương tranh từ pha thiết kế đến cài đặt chương trình. Để kiểm chứng ở mức thiết kế chúng tôi sử dụng phương pháp hình thức với Event-B và lập trình hướng khía cạnh với AspectJ để kiểm chứng ở mức cài đặt. Hiện tại, các phương pháp này được áp dụng để kiểm chứng ràng buộc thời gian và thứ tự thực hiện của các thành phần tương tranh. Do đó, trong tương lai chúng tôi tiếp tục nghiên cứu, phát triển và đề xuất các phương pháp kiểm chứng với Event-B. Cụ thể là xây dựng và phát triển các phương pháp đặc tả và kiểm chứng các hệ thống song song, hệ thống an toàn, hệ thống phản ứng lại (*reactive system*). Cài đặt công cụ hỗ trợ đặc tả song song để plugin vào bộ công cụ kiểm chứng mã nguồn mở RODIN của Event-B và tự động sinh mã Java từ đặc tả bằng Event-B.

Tiếp tục xây dựng và mở rộng các phương pháp được trình bày trong các Chương 5 và 6 để kiểm chứng các bất biến đối tượng/lớp (*object/class invariants*) và các ràng buộc khác trong chương trình tương tranh. Tương ứng mở rộng bộ công cụ kiểm chứng PVG để kiểm chứng các ràng buộc này và thực nghiệm cho các hệ thống phần mềm thực tế, các hệ thống mã nguồn có quy mô lớn. Tiến tới 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)*, pp. 143-148, Hanoi, Vietnam, Oct 14-17, 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.

Tài liệu tham khảo

[1] URL http://event-b.org.

[2] URL http://javapathfinder.sourceforge.net/How_to_Implement_Properties.html.

[3] URL http://www.uml.org.

[4] URL http://javapathfinder.sourceforge.net/Search_and_VMListeners.html.

[5] J.-R. Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

[6] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

[7] Jean-Raymond Abrial. Formal methods in industry : achievements, problems, future. In *ICSE*, pages 761–768, 2006.

[8] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2010.

[9] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models : Application to Event-B. *Fundam. Inf.*, 77(1-2) :1–28, 2007. ISSN 0169-2968.

[10] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, and Laurent Voisin. A Roadmap for the Rodin Toolset. In *ABZ*, page 347, 2008.

[11] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin : an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6) :447–466, 2010.

[12] Greg Andrews. *Concurrent Programming : Principles and Practice*. Addison-Wesley, 1991. ISBN 0805300864.

[13] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49 :2002, 2001.

[14] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN 026202649X, 9780262026499.

[15] Elisabeth Ball and Michael Butler. *Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction*. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-00866-5.

[16] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. First edition, 2008. ISBN 1846287693, 9781846287695.

[17] A. Ben Younes and L.J. Ben Ayed. From UML Activity Diagrams to Event B for the Specification and the Verification of Workflow Applications. In *COMPSAC '08 : Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 643–648, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3262-2.

[18] Saddek Bensalem and Doron Peled, editors. *Runtime Verification : 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-04693-3.

[19] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification : model-checking techniques and tools*. Springer-Verlag New York, Inc., New York, NY, USA, 1999. ISBN 3-540-41523-8.

[20] Sergey Berezin. *Model checking and theorem proving : a unified framework*. PhD thesis, Pittsburgh, PA, USA, 2002. AAI3051019.

[21] Trịnh Thanh Bình, Trương Anh Hoàng, and Nguyễn Việt Hà. Kiểm chứng sự tương tác giữa các thành phần trong chương trình đa luồng sử dụng lập

trình hướng khía cạnh. *Tạp chí Bưu chính Viễn thông và Công nghệ thông tin, Chuyên san Các công trình nghiên cứu triển khai Viễn thông và Công nghệ thông tin*, Số 4 :36–45, 2010.

[22] Trịnh Thanh Bình, Trương Ninh Thuận, and 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*, Số 2 :173–184, 2010.

[23] Eric Bodden. A lightweight LTL runtime verification tool for Java. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 306–307, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4.

[24] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, 2005. URL http://www.bodden.de/pubs/bodden05jlo.pdf.

[25] Eric Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, 2009. URL http://www.bodden.de/pubs/bodden09phdthesis.pdf. Available through ProQuest.

[26] Eric Bodden and Klaus Havelund. Aspect-oriented race detection in Java. *IEEE Trans. Softw. Eng.*, 36(4) :509–527, 2010. ISSN 0098-5589. URL http://dx.doi.org/10.1109/TSE.2010.25.

[27] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN 0321267974.

[28] Jeremy W Bryans. Developing a consensus algorithm using stepwise refinement. *Newcastle University,Technical Report*. URL http://deploy-eprints.ecs.soton.ac.uk/285/.

[29] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11 :481–494, October 1964. ISSN 0004-5411. URL http://doi.acm.org/10.1145/321239.321249.

[30] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3) :212–232, June 2005.

[31] Patrice Chalin and Frédéric Rioux. Jml runtime assertion checking : Improved error reporting and efficiency using strong validity. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 246–261, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-68235-6. URL http://dx.doi.org/10.1007/978-3-540-68237-0_18.

[32] Feng Chen and Grigore Roşu. Mop : an efficient and generic runtime verification framework. In *OOPSLA '07 : Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 569–588, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5.

[33] Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences in JML. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 511–516. CSREA Press, 2005. ISBN 1-932415-50-5. URL http://dblp.uni-trier.de/db/conf/serp/serp2005-2.html.

[34] Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Control*, 15(1) :7–25, 2007. ISSN 0963-9314. URL http://dx.doi.org/10.1007/s11219-006-9001-4.

[35] Edmund Clarke, Orna Grumberg, and David Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5) :1512–1542, 1994. ISSN 0164-0925. URL http://doi.acm.org/10.1145/186025.186051.

[36] A. Colyer and A. Clement. Aspect-oriented programming with AspectJ. *IBM Syst. J.*, 44(2) :301–308, 2005. ISSN 0018-8670. URL http://dx.doi.org/10.1147/sj.442.0301.

[37] R. DeLine and M. Fahndrich. The fugue protocol checker : Is your software baroque, 2004. URL citeseer.ist.psu.edu/deline04fugue.html.

[38] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP : deterministic shared memory multiprocessing. 44 :85–96, March 2009. URL http://doi.acm.org/10.1145/1508284.1508255.

[39] Werner Dietl and Peter Müller. Universes : Lightweight ownership for jml. *Journal of Object Technology*, 4(8) :5–32, 2005. URL http://dx.doi.org/10.5381/jot.2005.4.8.a1.

[40] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *2007 Future of Software Engineering*, FOSE '07, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. URL http://dx.doi.org/10.1109/FOSE.2007.6.

[41] Dariusz Dymek and Leszek Kotulski. Estimation of System Workload Time Characteristic using UML Timing Diagrams. In *DEPCOS-RELCOMEX '08 : Proceedings of the 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, pages 9–14, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3179-3. URL http://dx.doi.org/10.1109/DepCoS-RELCOMEX.2008.58.

[42] Andrew Edmunds. *Providing Concurrent Implementations for Event-B Developments*. PhD thesis, University of Southampton, March 2010. URL http://eprints.ecs.soton.ac.uk/20826/.

[43] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[44] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37 :234–245, May 2002. ISSN 0362-1340. URL http://doi.acm.org/10.1145/543552.512558.

[45] Martin Fowler and Kendall Scott. *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003. ISBN 0-321-19368-7.

[46] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition : The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000. ISBN 0201310082.

[47] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition : The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[48] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[49] Christian Haack, Marieke Huisman, and Clément Hurlin. Permission-Based Separation Logic for Multithreaded Java Programs, July 2010. URL http://www-sop.inria.fr/everest/Clement.Hurlin/papers.html. *One Paper to rule them all*. Submitted.

[50] Elnar Hajiyev, Laurie Hendren, Oege de Moor, Pavel Avgustinov, Eric Bodden, Ondrej Lhotak, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Aspects for Trace Monitoring. In Klaus Havelund, Manuel Nunez, Grigore Rosu, and Burkhart Wolff, editors, *Formal Approaches to Testing Systems and Runtime Verification (FATES/RV)*, volume 4262 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2006. URL http://www.bodden.de/pubs/abh%2B06aspects-for.pdf.

[51] Ayesha Hanif, Zaheer Ahmed, Muhammad Kashif Hanif, and Ali Aqdas. Regular expression to finite state machine, 2006.

[52] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San

Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=1624775.1624804.

[53] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. URL http://doi.acm.org/10.1145/976270.976276.

[54] Thai Son Hoang and Jean-Raymond Abrial. Event-B Decomposition for Parallel Programs. In *ASM*, pages 319–333, 2010.

[55] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978. ISSN 0001-0782. URL http://doi.acm.org/10.1145/359576.359585.

[56] Gerard Holzmann. *Spin model checker, the : primer and reference manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6.

[57] Clement Hurlin. Specifying and checking protocols of multithreaded classes. In *SAC '09 : Proceedings of the 2009 ACM symposium on Applied Computing*, pages 587–592, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. URL http://doi.acm.org/10.1145/1529282.1529407.

[58] Ying Jin. Formal verification of protocol properties of sequential Java programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1*, pages 475–482, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2870-8. URL http://dx.doi.org/10.1109/COMPSAC.2007.118.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[60] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01 : Proceedings*

of the 15th European Conference on Object-Oriented Programming, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

[61] Ramnivas Laddad. *AspectJ in Action : Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003. ISBN 1930110936.

[62] G. Leavens and Y. Cheon. Design by contract with JML, 2003.

[63] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual, 2002. URL citeseer.ist.psu.edu/leavens04jml.html.

[64] Gary T. Leavens. Tutorial on JML, the java modeling language. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 573–573, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. URL http://doi.acm.org/10.1145/1321631.1321747.

[65] Brad Long, Paul Strooper, and Luke Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures : Research articles. *Concurr. Comput. : Pract. Exper.*, 19 :281–294, March 2007. ISSN 1532-0626. URL http://portal.acm.org/citation.cfm?id=1228965.1228968.

[66] Robin Milner. *Communicating and mobile systems : the &pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1.

[67] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo : efficient deterministic multithreading in software. 44 :97–108, March 2009. URL http://doi.acm.org/10.1145/1508284.1508256.

[68] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0136619835.

[69] Michael Poppleton. The composition of Event-B models. In *ABZ '08 : Proceedings of the 1st international conference on Abstract State Machines, B*

and Z, pages 209–222. Springer-Verlag, 2008. ISBN 978-3-540-87602-1. URL http://dx.doi.org/10.1007/978-3-540-87603-8_17.

[70] Roger S. Pressman. *Software Engineering : A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. ISBN 0072496681.

[71] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

[72] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2007. ISBN 9780321493750.

[73] Paul Strooper and Luke Wildman. Testing Concurrent Java Components. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 161–162, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2892-9. URL http://dx.doi.org/10.1109/ICSECOMPANION.2007.74.

[74] Thanh-Binh Trinh, Tuan-Anh Do, Ninh-Thuan Truong, and Viet-Ha Nguyen. Checking the compliance of timing constraints in software applications. In *1nd Intern. Conf. on Knowledge and Systems Engineering*, pages 220–225, Los Alamitos, CA, USA, 2009. IEEE Computer Society. URL http://www.computer.org/portal/web/csdl/doi/10.1109/KSE.2009.38.

[75] Thanh-Binh Trinh, Anh-Hoang Truong, and Viet-Ha Nguyen. Checking protocol-conformance in component models using aspect oriented programming. In *Conf. on Advances in Computer Science and Engineering*, pages 150–155, Phuket, Thailand, 2009. URL http://www.actapress.com/Abstract.aspx?paperId=34884.

[76] Thanh-Binh Trinh, Quang-Thap Pham, Ninh-Thuan Truong, and Viet-Ha Nguyen. A runtime approach to verify scenario in multi-agent systems. In *2nd Intern. Conf. on Knowledge and Systems Engineering*, pages 161–166, Los Alamitos, CA, USA, 2010. IEEE Computer Society. URL http://www.computer.org/portal/web/csdl/doi/10.1109/KSE.2010.13.

[77] Thanh-Binh Trinh, Ninh-Thuan Truong, and Viet-Ha Nguyen. Refining undetermined events for specifying concurrent programs. In *3nd Intern. Conf. on Knowledge and Systems Engineering*, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[78] 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. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 382–386, Washington, DC, USA, 2008. IEEE Computer Society. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4685826.

[79] Ninh-Thuan Truong, Thanh-Binh Trinh, and Viet Ha Nguyen. Coordinated consensus analysis of multi-agent systems using Event-B. In *Proceedings of the 2009 seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 201–209, Washington, DC, USA, 2009. IEEE Computer Society. URL http://www.computer.org/portal/web/csdl/doi/10.1109/SEFM.2009.24.

[80] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00 : Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0710-7.

[81] Friedrich Wilhelm. A Compiler for the Entire Class of Context-free Grammars. URL http://accent.compilertools.net/Accent.html.

[82] Letu Yang. *The Automated Translation of Integrated Formal Specifications into Concurrent Programs*. PhD thesis, University of Southampton, Setember 2008. URL http://eprints.soton.ac.uk/63157/.

Phụ lục A

Đặc tả ràng buộc thứ tự giữa các tiến trình tương tranh

A.1 Vấn đề vùng xung đột

A.1.1 Mô hình khởi tạo

An Event-B Specification of CriticalSectionI
Creation Date: 14 Nov 2010 @ 09 :46 :38 AM

MACHINE CriticalSectionI
// abstract machine
VARIABLES
x // x,y,z are variables that access the critical section
y
z
g // guard of all events
INVARIANTS
inv1 : $x \in \mathbb{N}$
inv2 : $y \in \mathbb{N}$
inv3 : $z \in \mathbb{N}$
inv4 : $g \in BOOL$
EVENTS
Initialisation
begin
act1 : $x := 0$
act2 : $y := 0$
act3 : $z := 0$
act4 : $g := TRUE$

end
Event *evt1* $\widehat{=}$
when
grd1 : g = TRUE
then
act1 : x := x+1
end
Event *evt2* $\widehat{=}$
when
grd1 : g = TRUE
then
act1 : y := y+1
end
Event *evt3* $\widehat{=}$
when
grd1 : g = TRUE
then
act1 : z := z+1
end
END

A.1.2 Mô hình làm mịn

An Event-B Specification of CriticalSectionR
Creation Date: 14 Nov 2010 @ 09 :46 :35 AM

MACHINE CriticalSectionR
// concretemachine
REFINES CriticalSectionI
VARIABLES
x
y
z
g
Turn // semaphore variable
INVARIANTS
inv1 : x ∈ ℕ
inv2 : y ∈ ℕ
inv3 : z ∈ ℕ

inv4 : g ∈ BOOL

inv5 : Turn ∈ ℕ

EVENTS

Initialisation

begin

act1 : x := 0

act2 : y := 0

act3 : z := 0

act4 : g := TRUE

act5 : Turn := 0

end

Event *evt1R* ≙

refines *evt1*

when

grd1 : g = TRUE

grd2 : Turn = 0

then

act1 : x := x+1

act2 : Turn := 1

end

Event *evt2R* ≙

refines *evt2*

when

grd1 : g = TRUE

grd2 : Turn = 1

then

act1 : y := y+1

act2 : Turn := 2

end

Event *evt3R* ≙

refines *evt3*

when

grd1 : g = TRUE

grd2 : Turn = 2

then

act1 : z := z+1

act2 : Turn := 0

end

END

A.2 Vấn đề cung cấp tiêu thụ

A.2.1 Mô hình khởi tạo

An Event-B Specification of ProducerConsumerI
Creation Date: 16 Mar 2011 @ 07 :07 :26 AM

MACHINE ProducerConsumerI
VARIABLES

`Queue`
`Front`
`Rear`
`Queue_Size`
`g`
`d`

INVARIANTS

`inv1` : $Queue \in \mathbb{N} \pfun \mathbb{N}$
`inv2` : $Front \in 0 \upto Queue\_Size$
`inv3` : $Rear \in 0 \upto Queue\_Size$
`inv4` : $Queue\_Size \in \mathbb{N}$
`inv6` : $g \in BOOL$
`inv7` : $d \in \mathbb{N}$

EVENTS
Initialisation

begin

`act1` : $Queue := \varnothing$
`act2` : $Front := 0$
`act3` : $Rear := 1$
`act4` : $Queue\_Size := 100$
`act5` : $g := TRUE$
`act6` : $d := 0$

end

Event *Producer* $\widehat{=}$

any

x

where

`grd1` : $x \in dom(Queue)$
`grd2` : $g = TRUE$

then

`act1` : $Front := Front + 1$

act2 : $Queue(Front) := x$

end

Event *Consumer* $\widehat{=}$

when

grd1 : $g = TRUE$

then

act2 : $d := Queue(Rear)$

act1 : $Rear := Rear{+}1$

end

END

A.2.2 Mô hình làm mịn

An Event-B Specification of ProducerConsumerR
Creation Date: 16 Mar 2011 @ 07 :07 :32 AM

MACHINE ProducerConsumerR
REFINES ProducerConsumerI
VARIABLES

`Queue`
`Queue_Size`
`Count` semaphore variable
`g`
`Front`
`Rear`
`d`
`TurnP`
`TurnC`
`consumers`
`producers`

INVARIANTS

inv1 : $Count \in 0 \,..\, Queue\_Size$
inv2 : $Queue \in \mathbb{N} \pfun \mathbb{N}$
inv3 : $Front \in 0 \,..\, Queue\_Size$
inv4 : $Rear \in 0 \,..\, Queue\_Size$
inv5 : $Queue\_Size \in \mathbb{N}$
inv6 : $g \in BOOL$
inv7 : $d \in \mathbb{N}$
inv11 : $consumers \subseteq \mathbb{N}_1$
inv12 : $producers \subseteq \mathbb{N}_1$
inv13 : $TurnP \in producers$

inv14 : $TurnC \in consumers$

EVENTS

Initialisation

begin

act1 : $Queue := \varnothing$

act2 : $Count := 0$

act3 : $Front := 0$

act4 : $Rear := 1$

act5 : $g := TRUE$

act6 : $Queue\_Size := 100$

act7 : $d := 0$

act9 : $TurnC := 1$

act10 : $producers := 1 \ldots 100$

act11 : $TurnP := 1$

act12 : $consumers := 1 \ldots 100$

end

Event *Producer1R* $\widehat{=}$
refines *Producer*

any

x

where

grd1 : $x \in dom(Queue)$

grd2 : $g = TRUE$

grd3 : $Count < Queue\_Size$

Queue is not full

grd5 : $TurnP = 1$

then

act2 : $Queue(Front) := x$

producer data element

act3 : $Count := Count+1$

act6 : $TurnP :\in producers \setminus \{TurnP\}$

end

Event *Consumer1R* $\widehat{=}$
refines *Consumer*

when

grd1 : $g = TRUE$

grd2 : $Count > 0$

Queue is not empty

grd3 : $TurnC = 0$

then

act1 : $d := Queue(Rear)$
consumer data element
act2 : $Rear := Rear{+}1$
act3 : $Count := Count{-}1$
act4 : $TurnC := 1$

end

Event *evt1* $\widehat{=}$

when

grd1 : $g = TRUE$
grd2 : $Count < Queue\_Size$
grd3 : $Front = Queue\_Size$

then

act1 : $Front := 1$

end

Event *evt2* $\widehat{=}$

when

grd1 : $g = TRUE$
grd2 : $Count < Queue\_Size$
grd3 : $Front < Queue\_Size$

then

act1 : $Front := Front{+}1$

end

Event *evt3* $\widehat{=}$

when

grd1 : $g = TRUE$
grd2 : $Count > 0$
grd3 : $Rear = Queue\_Size$

then

act1 : $Rear := 1$

end

Event *evt4* $\widehat{=}$

when

grd1 : $g = TRUE$
grd2 : $Count > 0$
grd3 : $Rear = Front$

then

act1 : $Rear := 1$
act2 : $Front := 0$

end

Event *evt5* $\widehat{=}$
when
grd1 : $g = TRUE$
grd2 : $Count > 0$
then
act1 : $Rear := Rear{+}1$
end
Event *Producer2R* $\widehat{=}$
refines *Producer*
any
x
where
grd1 : $x \in dom(Queue)$
grd2 : $TurnP = 1$
grd4 : $g = TRUE$
grd5 : $Count < Queue\_Size$
then
act2 : $Queue(Front) := x$
act3 : $Count := Count{+}1$
act4 : $TurnP := 0$
end
Event *Consumer2R* $\widehat{=}$
refines *Consumer*
when
grd1 : $g = TRUE$
grd2 : $Count > 0$
grd3 : $TurnC = 1$
then
act1 : $d := Queue(Rear)$
act2 : $Count := Count{-}1$
act3 : $TurnC := 0$
end
END

A.3 Vấn đề đọc ghi

A.3.1 Mô hình khởi tạo

An Event-B Specification of ReaderWrite
Creation Date: 26 Nov 2010 @ 05 :22 :17 PM

MACHINE ReaderWrite
VARIABLES

g
rd
wt

INVARIANTS

inv1 : g ∈ BOOL
inv2 : rd ∈ ℕ
inv3 : wt ∈ ℕ

EVENTS
Initialisation

begin

act1 : g := TRUE
act2 : rd := 0
act3 : wt := 0

end

Event *reader* ≙

when

grd1 : g = TRUE

then

act1 : rd := rd+1

end

Event *write* ≙

when

grd1 : g = TRUE

then

act1 : wt := wt+1

end

END

A.3.2 Mô hình làm mịn

An Event-B Specification of ReaderWriterR
Creation Date: 26 Nov 2010 @ 05 :22 :21 PM

MACHINE ReaderWriterR
REFINES ReaderWrite
VARIABLES

g
wt

rd
readers
writers
OktoRead
OktoWrite
isRead
isWrite
endOfRead
endOfWrite

INVARIANTS

inv1 : g ∈ BOOL
inv2 : wt ∈ ℕ
inv3 : rd ∈ ℕ
inv4 : readers ∈ ℕ
inv5 : writers ∈ ℕ
inv6 : OktoRead ∈ BOOL
inv7 : OktoWrite ∈ BOOL
inv8 : isRead ∈ BOOL
inv9 : isWrite ∈ BOOL
inv10 : endOfRead ∈ BOOL
inv11 : endOfWrite ∈ BOOL

EVENTS

Initialisation

begin

act1 : readers := 0
act2 : writers := 0
act3 : OktoRead := TRUE
act4 : OktoWrite := FALSE
act5 : g := TRUE
act6 : wt := 0
act7 : rd := 0
act8 : isRead := FALSE
act9 : isWrite := FALSE
act10 : endOfRead := FALSE
act11 : endOfWrite := FALSE

end

Event *startRead* $\widehat{=}$

when

grd1 : g = TRUE
grd2 : writers ≠ 0
grd3 : OktoWrite = FALSE

grd4 : OktoRead = TRUE

then

act1 : readers := readers+1

act2 : isRead := TRUE

end

Event *endReadIf* $\widehat{=}$

when

grd1 : endOfRead = TRUE

grd2 : readers > 0

then

act1 : readers := readers−1

end

Event *endReadElse* $\widehat{=}$

when

grd1 : readers = 0

grd2 : endOfRead = TRUE

then

act1 : OktoWrite := TRUE

end

Event *readerR* $\widehat{=}$
refines *reader*

when

grd1 : isRead = TRUE

grd1 : g = TRUE

then

act1 : rd := rd+1

act2 : endOfRead := TRUE

end

Event *startWrite* $\widehat{=}$

when

grd1 : g = TRUE

grd2 : readers ≠ 0∨writers ≠ 0

grd3 : OktoWrite = TRUE

then

act1 : writers := writers+1

act2 : isWrite := TRUE

end

Event *endWrite* $\widehat{=}$

when

grd1 : endOfWrite = TRUE

grd1 : writers > 0

then

act1 : writers := writers−1

end

Event *endWriteIf* $\widehat{=}$

when

grd1 : endOfWrite = TRUE

grd2 : OktoRead = FALSE

then

act1 : OktoWrite := TRUE

end

Event *endWriteElse* $\widehat{=}$

when

grd1 : endOfWrite = FALSE

grd2 : OktoRead = TRUE

then

act1 : OktoRead := TRUE

end

Event *writerR* $\widehat{=}$
refines *write*

when

grd1 : isWrite = TRUE

then

act1 : wt := wt+1

act2 : endOfWrite := TRUE

end

END

Phụ lục B

Đặc tả hệ thống đa thành phần thực hiện các phép toán nhị phân

B.1 Đặc tả phép dịch bit

B.1.1 Ngữ cảnh của phép dịch bit

An Event-B Specification of BitShiftctx
Creation Date: 14 Oct 2010 @ 08 :43 :25 AM

CONTEXT BitShiftctx
CONSTANTS

size_pp

numShift

AXIOMS

axm2 : size_pp $>$ 0

axm3 : numShift $>$ 0

axm4 : numShift $<$ size_pp

END

B.1.2 Máy thực thi của phép dịch bit

An Event-B Specification of BitShiftmch
Creation Date: 14 Oct 2010 @ 08 :55 :21 AM

MACHINE BitShiftmch
SEES BitShiftctx
VARIABLES

ppr //Get temporary results

`slr` //Result of the shift left

`kk`

INVARIANTS

inv4 : $\texttt{ppr} \in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv2 : $\texttt{slr} \in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv3 : $\texttt{kk} \in \mathbb{N}$

EVENTS

Initialisation

begin

act1 : $\texttt{ppr} := \varnothing$

act2 : $\texttt{slr} := \varnothing$

act3 : $\texttt{kk} := \texttt{size\_pp}$

end

Event *ShiftLeft_result* $\widehat{=}$

when

grd1 : $\texttt{kk} = 0$

then

act1 : $\texttt{slr} := \texttt{ppr}$

end

Event *ShiftLeftIf* $\widehat{=}$

when

grd1 : $\texttt{kk} > 0$

grd2 : $\texttt{kk} > \texttt{numShift}$

then

act1 : $\texttt{ppr}(\texttt{kk}) := \texttt{ppr}(\texttt{kk} - \texttt{numShift})$

act2 : $\texttt{kk} := \texttt{kk} - 1$

end

Event *ShiftLeftElse* $\widehat{=}$

when

grd1 : $\texttt{kk} > 0$

grd2 : $\texttt{kk} \leq \texttt{numShift}$

then

act1 : $\texttt{ppr}(\texttt{kk}) := 0$

act2 : $\texttt{kk} := \texttt{kk} - 1$

end

END

B.2 Đặc tả phép nhân xâu nhị phân với một bit

B.2.1 Ngữ cảnh của phép nhân xâu nhị phân với một bit

An Event-B Specification of MultiDigitctx
Creation Date: 13 Oct 2010 @ 10 :44 :57 AM

CONTEXT MultiDigit
CONSTANTS

aa
digit
size_aa
size_modr

AXIOMS

axm1 : $aa \in \mathbb{N} \rightarrow 0 \mathinner{..} 1$
axm2 : $digit \in \mathbb{N}$
axm3 : $size\_aa > 0$
axm4 : $size\_modr > 0$
axm5 : $size\_modr \geq size\_aa$

THEOREMS

thm1 : $ran(aa) \neq \varnothing$

END

B.2.2 Máy thực thi của phép nhân xâu nhị phân với một bit

An Event-B Specification of MultiDigitmch
Creation Date: 13 Oct 2010 @ 11 :15 :17 AM

MACHINE MultiDigitMchine
SEES MultiDigitctx
VARIABLES

pp
modr // Final result
jj

INVARIANTS

inv1 : $modr \in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$
inv2 : $pp \in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$
inv3 : $jj \in \mathbb{N}$

EVENTS
Initialisation

begin

act1 : modr := ∅

act2 : pp := ∅

act3 : jj := 1

end

Event *MultiplyWithOneDigit* ≙

when

grd1 : jj ≤ size_aa

then

act1 : pp(jj) := digit·aa(jj)

act2 : jj := jj+1

end

Event *MultiplyWithOneDigit_result* ≙

when

grd1 : jj = size_aa+1

then

act1 : modr := pp //Get the result

end

END

B.3 Đặc tả phép cộng xâu nhị phân

B.3.1 Ngữ cảnh của phép cộng xâu nhị phân

An Event-B Specification of Sumctx
Creation Date: 14 Oct 2010 @ 09 :12 :11 AM

CONTEXT Sumctx
CONSTANTS

aa

bb

size_aa

size_bb

size_ar

AXIOMS

axm1 : aa ∈ $\mathbb{N}_1$ → 0 . . 1

axm2 : bb ∈ $\mathbb{N}_1$ → 0 . . 1

axm3 : size_aa > 0

axm4 : size_bb > 0

axm5 : size_aa $<$ size_ar

axm6 : size_bb $<$ size_ar

axm7 : size_ar $<$ size_aa+size_bb

THEOREMS

thm1 : ran(aa) $\neq \varnothing$

thm2 : ran(bb) $\neq \varnothing$

END

B.3.2 Máy thực thi của phép cộng hai xâu nhị phân

An Event-B Specification of Summch
Creation Date: 14 Oct 2010 @ 09 :12 :04 AM

MACHINE Summch
SEES Sumctx
VARIABLES

cc // Get temprorary result

ar // Result of the addition operation

carry

hh

INVARIANTS

inv1 : cc $\in \mathbb{N}_1 \rightarrow 0\,..\,1$

inv2 : ar $\in \mathbb{N}_1 \rightarrow 0\,..\,1$

inv3 : hh $\in \mathbb{N}$

inv4 : carry $\in \mathbb{N}$

EVENTS
Initialisation

begin

act1 : cc $:= \varnothing$

act2 : ar $:= \varnothing$

act3 : hh $:= 1$

act4 : carry $:= 0$

end

Event *AdditionResult* $\widehat{=}$

when

grd1 : hh $=$ size_ar+1

then

act1 : ar $:=$ cc // Get result

end

Event *AdditionIf* $\widehat{=}$

when

grd1 : hh $\leq$ size_ar

grd2 : hh = size_ar$\wedge$carry $\neq$ 0

then

act1 : cc(hh+1) := 1

end

Event *AdditionElse* $\widehat{=}$

when

grd1 : hh $<$ size_ar

grd2 : hh $\neq$ size_ar$\wedge$carry = 0

then

act1 : cc(hh) := (aa(hh)+bb(hh)+carry)mod2

act2 : carry := (aa(hh)+bb(hh)+carry)/2

act3 : hh := hh+1

end

END

B.4 Đặc tả hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân

B.4.1 Ngữ cảnh của hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân

An Event-B Specification of Masctx
Creation Date: 14 Oct 2010 @ 11 :39 :32 AM

CONTEXT Masctx

CONSTANTS

aa

bb

size_aa

size_bb

size_res

size_pp

numShift

AXIOMS

axm1 : aa $\in \mathbb{N}_1 \rightarrow 0\,..\,1$

axm2 : bb $\in \mathbb{N}_1 \rightarrow 0\,..\,1$

axm3 : size_aa $>$ 0

axm4 : size_bb $>$ 0

axm5 : size_aa $<$ size_res

axm6 : size_bb $<$ size_res

axm7 : numShift $<$ size_pp

THEOREMS

thm1 : ran(aa) $\neq \varnothing$

thm2 : ran(bb) $\neq \varnothing$

END

B.4.2 Máy thực thi của hệ thống đa thành phần thực hiện phép nhân hai xâu nhị phân

An Event-B Specification of Masmch
Creation Date: 14 Oct 2010 @ 11 :39 :36 AM

MACHINE Masmch
SEES Masctx
VARIABLES

ii

jj

cc

res

ppr

kk

modr

slr

hh

carry

INVARIANTS

inv1 : ii $\in \mathbb{N}$

inv2 : jj $\in \mathbb{N}$

inv3 : cc $\in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv4 : res $\in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv5 : ppr $\in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv6 : kk $\in \mathbb{N}$

inv7 : modr $\in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv8 : slr $\in \mathbb{N}_1 \rightarrow 0 \mathinner{..} 1$

inv9 : hh $\in \mathbb{N}$

inv10 : carry $\in \mathbb{N}$

EVENTS
Initialisation

begin

act1 : ii := 1

act2 : jj := 1

act3 : slr := ∅

act4 : kk := size_pp

act5 : cc := ∅

act6 : hh := 1

act7 : ppr := ∅

act8 : carry := 0

act9 : modr := ∅

act10 : res := ∅

end

Event *Multiply2BinaryNumbers* $\widehat{=}$

when

grd1 : ii = size_bb+1

then

act1 : res := cc

end

Event *MultiplyWithOneDigit* $\widehat{=}$

when

grd1 : jj < size_aa

then

act1 : ppr(jj) := bb(ii)·aa(jj)

act2 : jj := jj+1

end

Event *MultiplyWithOneDigit_result* $\widehat{=}$

when

grd1 : jj < size_aa+1

then

act1 : modr := ppr

act2 : kk := size_pp //Activate shiftLeft

end

Event *ShiftLeftIf* $\widehat{=}$

when

grd1 : kk > 0∧kk ≤ size_pp

grd2 : kk > numShift

then

act1 : modr(kk) := modr(kk−numShift)

act2 : kk := kk−1

end

Event *ShiftLeftElse* $\widehat{=}$

when

grd1 : kk $>$ 0$\wedge$kk $\leq$ size_pp

grd2 : kk $\leq$ numShift

then

act1 : modr(kk) := 0

act2 : kk := kk$-$1

end

Event *ShiftLeftResult* $\widehat{=}$

when

grd1 : kk $=$ 0

then

act1 : slr := modr

act2 : hh := 1 //Activate the addition event

end

Event *AdditionIf* $\widehat{=}$

when

grd1 : hh $\leq$ size_res

grd2 : hh $=$ size_res$\wedge$carry $\neq$ 0

then

act1 : cc(hh+1) := 1

end

Event *AdditionElse* $\widehat{=}$

when

grd1 : hh $\leq$ size_res

grd2 : $\neg$ (hh $=$ size_res$\wedge$carry $\neq$ 0)

then

act1 : cc(hh) := (cc(hh)+slr(hh)+carry)mod2

act2 : carry := (cc(hh)+slr(hh)+carry)/2

act3 : hh := hh+1

end

Event *AdditionalResult* $\widehat{=}$

when

grd1 : hh $=$ size_res+1

then

act1 : res := cc

act2 : jj := 1

act3 : ii := ii+1

end

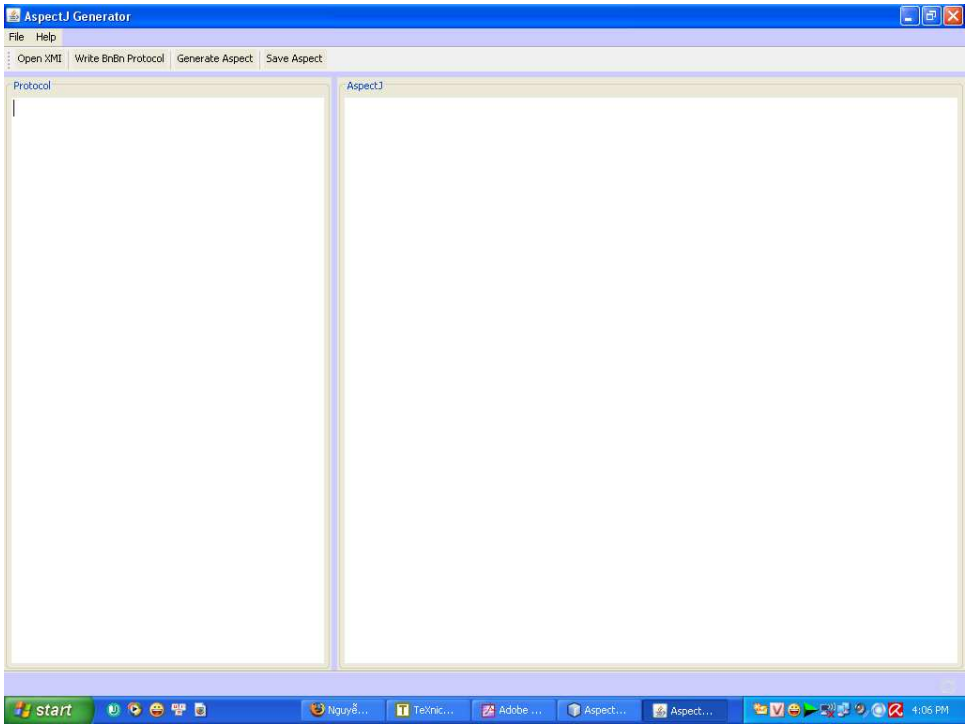
END

Phụ lục C

Công cụ sinh mã kiểm chứng PVG

C.1 Giới thiệu

PVG - Protocol Verification Generator là bộ công cụ sinh mã kiểm chứng AspectJ từ các đặc tả bằng biểu đồ UML hoặc biểu thức chính quy. Mã kiểm chứng sau đó được đan với các chương trình Java để kiểm chứng sự tuân thủ giữa chương trình và đặc tả của nó. Hiện tại PVG đã hỗ trợ kiểm chứng sự tuân thủ của chương trình so với đặc tả giao thức tương tác giữa các thành phần (*thứ tự thực hiện của các phương thức trong các lớp hoặc các thành phần*). Hoặc ràng buộc thời gian giữa các thành phần trong chương trình tương tranh. Trong đó, giao thức tương tác được đặc tả bằng máy trạng thái giao thức, biểu đồ tuần tự của UML hoặc biểu thức chính quy mở rộng. Ràng buộc thời gian được đặc tả bằng biểu đồ thời gian của UML hoặc biểu thức chính quy.



Hình C.1 – Giao diện chính của công cụ sinh mã kiểm chứng PVG.

C.2 Hướng dẫn sử dụng

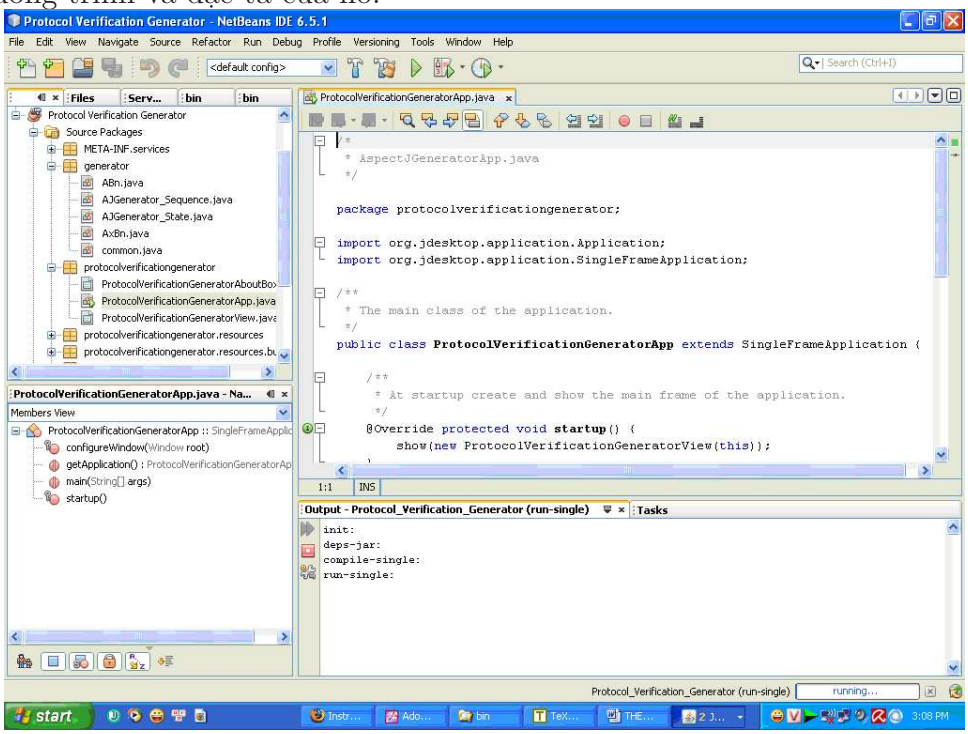
C.2.1 Các yêu cầu

Môi trường thực thi Java JRE phiên bản 1.5 hoặc cao hơn, có thể Download tại địa chỉ `http://java.sun.com`. Công cụ PVG có thể được Download tại địa chỉ : `http://www.mediafire.com/?uz9sw1u9gl0hez7`. Để khởi động bộ công cụ PVG trong hệ điều hành Window chỉ cần nhấp đúp chuột vào file có phần mở rộng .jar. Trong Unix, sử dụng lệnh : java –jar PVG.jar. PVG cũng có thể được khởi động bằng cách import mã nguồn của nó từ môi trường phát triển tích hợp IDE của NetBeans hay Eclipse (*có thể Download Netbeans, Eclipse tại địa chỉ* `http://netbeans.org/downloads/`, `http://www.eclipse.org/downloads/`) sau đó chạy file ProtocolGeneratorApp.java (Hình C.2). Giao diện chính của công cụ PVG sau khi khởi động như trong Hình C.1.

C.2.2 Các chức năng chính

Phiên bản hiện tại của công cụ PVG gồm bốn chức năng chính (Hình C.1).

- **Open XMI :** Đọc đặc tả từ các biểu đồ UML như biểu đồ máy trạng thái giao thức, biểu đồ tuần tự hoặc biểu đồ thời gian,
- **Write protocol :** Đặc tả trực tiếp giao thức tương tác hoặc ràng buộc thời gian bằng các biểu thức chính quy trong các file dạng .txt,
- **Generate Aspect :** Sinh mã aspect từ các đặc tả trên,
- **Save Aspect :** Lưu mã aspect dưới dạng các file có phần mở rộng là *.aj, các file này sẽ được đan với các chương trình Java để kiểm chứng sự tuân thủ giữa chương trình và đặc tả của nó.



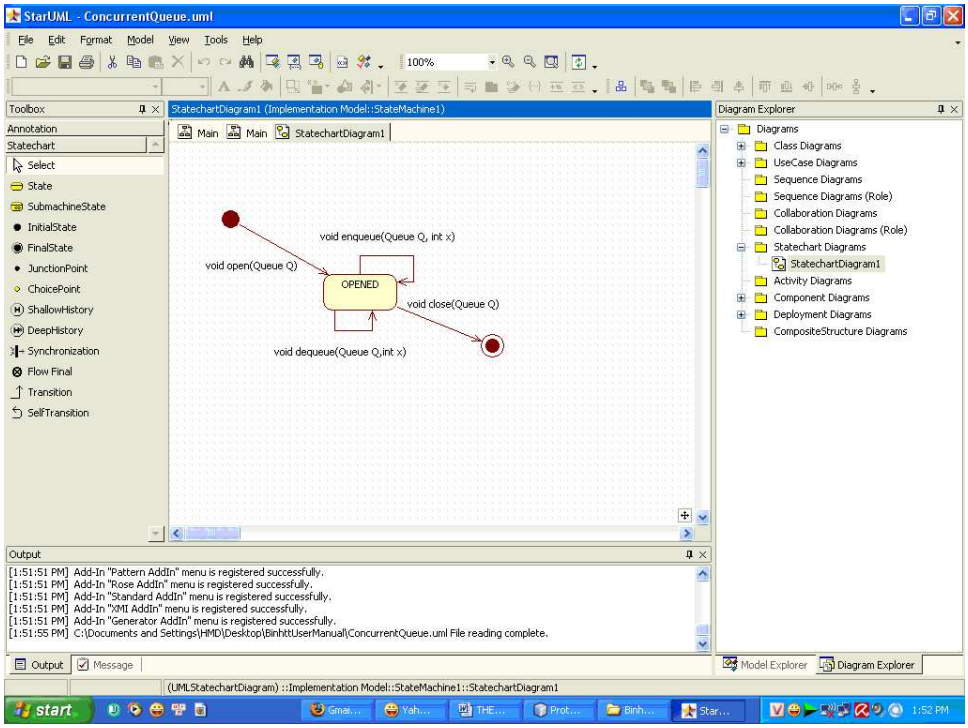
Hình C.2 – Khởi động PVG từ NetBeans.

C.2.3 Hướng dẫn thực hiện

C.2.3.1 Đặc tả giao thức

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

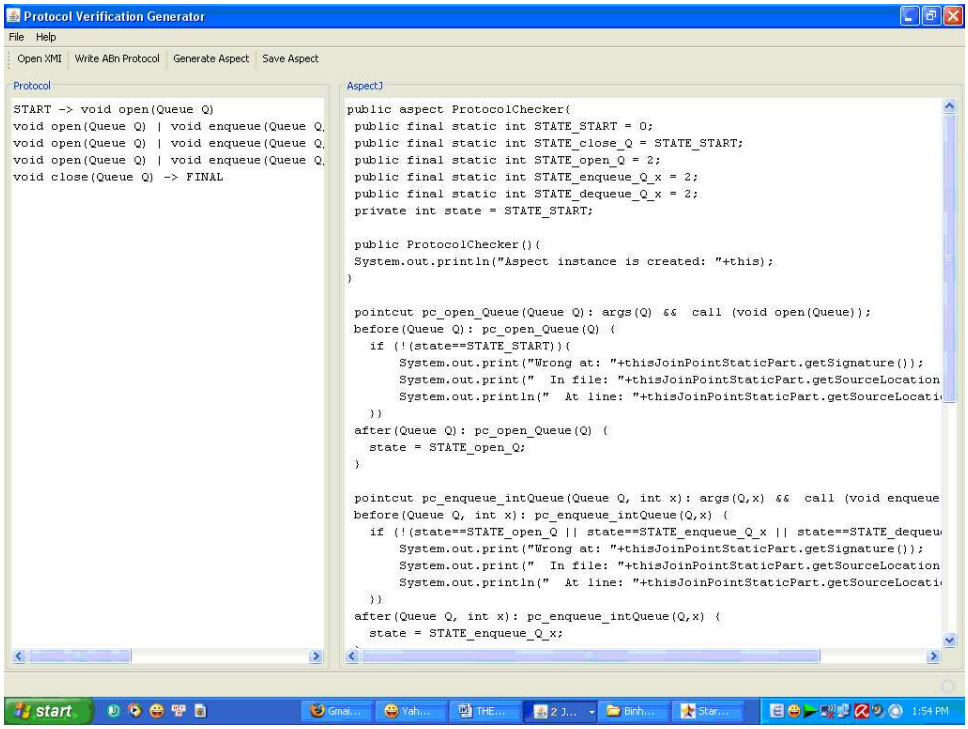
cung cấp Producer đẩy các phần tử vào hàng đợi, và nhiều luồng Consumer cùng thao tác với các phần tử trong hàng đợi (Hình 3.4 , Chương 5). Giao thức này được đặc tả bằng máy trạng thái giao thức của UML như trong Hình C.3.



Hình C.3 – Đặc tả giao thức tương tác của hàng đợi tương tranh với UML.

Hình C.4 mô tả một giao thức tương tác của hàng đợi tương tranh và mã aspect được sinh ra. Trong đó, với mỗi phương thức được đặc tả trong giao thức thì mã aspect được sinh ra sẽ chứa một biến trạng thái, và một pointcut tương ứng. Trước khi phương thức được thực hiện thì các câu lệnh trong `before(..)` của pointcut sẽ kiểm tra các trạng thái và tiền điều kiện mà nó phải thỏa mãn. Sau khi phương thức được thực hiện xong thì các câu lệnh trong `after(..)` của pointcut sẽ kiểm tra các mệnh đề hậu điều kiện và biến trạng thái được gán bằng trạng thái của phương thức hiện tại. Mỗi khi có vi phạm về giao thức thì các hàm `getSourceLocation()` và `getSignature()` của aspect được sinh ra sẽ thông báo chính xác vị trí và phương thức được gọi gây ra vi phạm. Một trạng thái đặc biệt *ST_START* của aspect được sinh ra tương ứng với trạng thái của phương thức được thực hiện đầu tiên trong giao thức. Khi phương thức cuối cùng trong giao thức được thực hiện thì

trạng thái của nó sẽ được gán bằng trạng thái đặc biệt này để bảo đảm các giao thức tiếp theo được kiểm chứng.



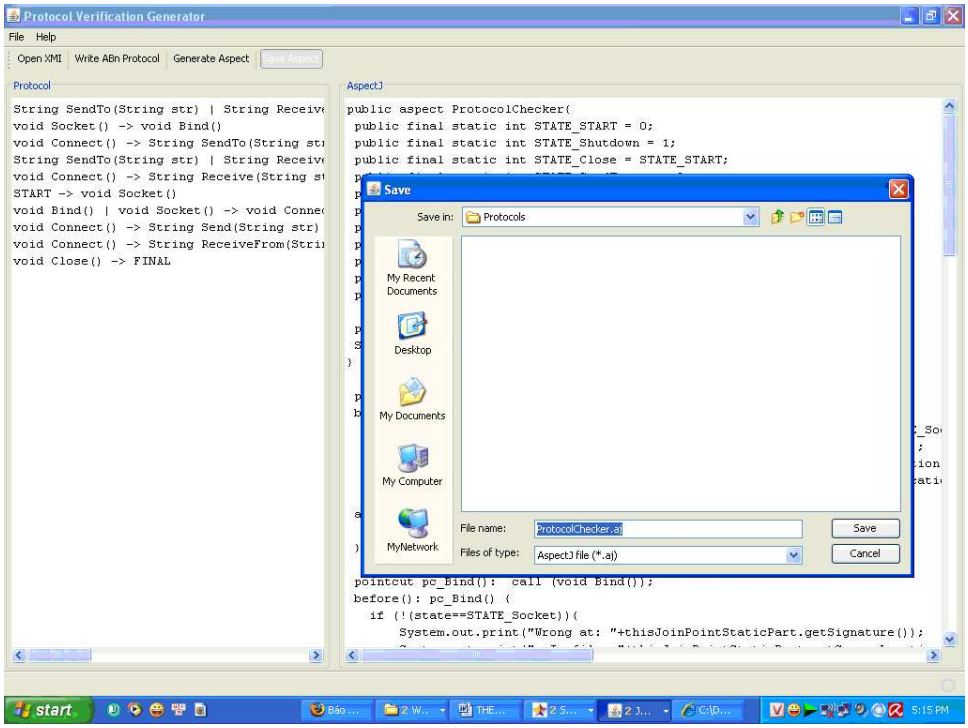
Hình C.4 – Đặc tả giao thức của hàng đợi tương tranh trong textbox bên trái và mã AspectJ được sinh ra bên phải.

C.2.3.2 Lưu mã Aspect

Sau khi sinh mã AspectJ từ đặc tả của nó, chức năng save cho phép người sử dụng lưu lại mã aspect được sinh ra trong các file riêng để đan với các chương trình Java cần kiểm chứng. Các file này có phần mở rộng là *.aj, theo định dạng của ngôn ngữ lập trình hướng khía cạnh với AOP (Hình C.5).

C.2.3.3 Đan mã aspect

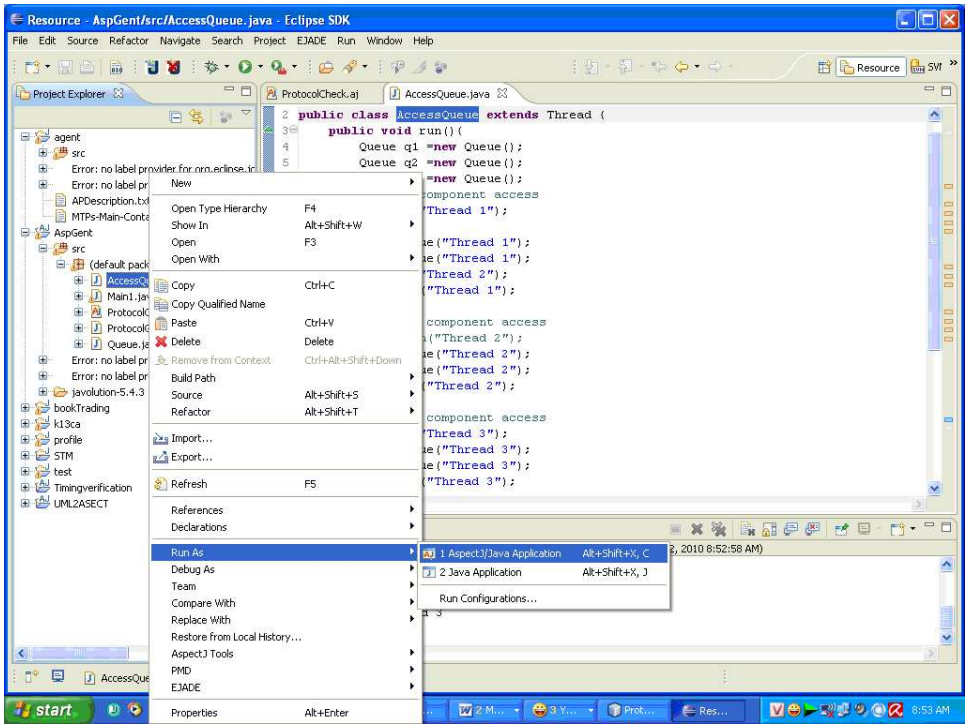
AspectJ cho phép đan xen mã aspect với các chương trình Java ở ba mức khác nhau : mức mã nguồn, mã bytecode và tại thời điểm nạp chương trình khi chương trình gốc chuẩn bị được thực hiện. Đan ở mức mã nguồn, AspectJ sẽ nạp các mã aspect và Java ở mức mã nguồn (.aj và .java), sau đó thực hiện biên dịch để sinh ra



Hình C.5 – Lưu mã aspect được 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. Ví dụ đan xen ở mức mã nguồn trong eclipse hoặc NetBeans (Hình C.6) hoặc thực hiện các câu lệnh.

- Yêu cầu :
  - **Mã nguồn :** file chương trình nguồn (*chương trình cần kiểm chứng*) *.java và file chứa mã aspect được sinh ra dạng *.aj.
  - **Thiết lập môi trường :**
    - PATH : <aspectj>/bin
    - CLASSPATH : <aspectj>/lib/aspectjrt.jar
- **Thực hiện :**
  - **Dịch và đan xen :**
    - ajc ConcurrentQueueJ.java ConcurrentQueueA.aj



Hình C.6 – Đan xen mã aspect với mã Java trong Eclipse.

– **Chạy và kiểm tra :**
  – aj ConcurrentQueueJ hoặc,
  – java ConcurrentQueueJ.