BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

LUẬN VĂN THẠC SỸ KHOA HỌC

KIỂM TRA MÔ HÌNH PHẦN MỀM SỬ DỤNG LÝ THUYẾT ÔTÔMAT BUCHI VÀ LOGIC THỜI GIAN TUYẾN TÍNH

NGÀNH: CÔNG NGHỆ THÔNG TIN
MÃ SỐ:

PHẠM THỊ THÁI NINH

Người hướng dẫn khoa học: TS. HUỲNH QUYẾT THẮNG

HÀ NỘI 2006

LỜI CẢM ƠN

Trước hết tôi xin gửi lời cảm ơn đặc biệt nhất tới Thầy TS Huỳnh Quyết Thắng, người đã định hướng đề tài và tận tình hướng dẫn chỉ bảo tôi trong suốt quá trình thực hiện bản luận văn cao học này, từ những ý tưởng trong đề cương nghiên cứu, phương pháp giải quyết vấn đề cho đến những lần kiểm tra cuối cùng để hoàn tất bản luận văn.

Tôi xin chân thành bày tỏ lòng biết ơn sâu sắc tới Trung tâm Đào tạo Sau đại học và các thầy cô giáo trong khoa Công nghệ thông tin, trường Đại học Bách Khoa Hà Nội đã cho tôi nhiều kiến thức quý báu về các vấn đề hiện đại của ngành công nghệ thông tin, cho tôi một môi trường tập thể, một khoảng thời gian học cao học tuy ngắn ngủi nhưng khó quên trong cuộc đời.

Tôi xin bày tỏ lòng cảm ơn chân thành tới tất cả các bạn bè, các đồng nghiệp đã động viên tôi trong suốt thời gian thực hiện bản luận văn này.

Cuối cùng tôi xin dành một tình cảm biết ơn sâu nặng tới Bố, Mẹ và gia đình, những người đã luôn luôn ở bên cạnh tôi trong mọi nơi, mọi lúc trong suốt quá trình làm bản luận văn cao học này cũng như trong suốt cuộc đời tôi.

Hà nội, tháng 11 năm 2006

Tác giả

Phạm Thị Thái Ninh

LỜI CAM ĐOAN

Tôi xin cam đoan đây là công trình nghiên cứu của riêng tôi. Các kết quả nêu trong bản luận văn này là trung thực và chưa từng được ai công bố trong bất cứ công trình nào khác.

TÁC GIẢ LUẬN VĂN

PHẠM THỊ THÁI NINH

MỤC LỤC

DANH MỤC CÁC TỪ VIẾT TẮT

Số TT	Từ viết tắt	Giải nghĩa
1	OBDD	Ordered Binary Decision Diagrams
2	BDD	Binary Decision Diagrams
3	LTL	Linear Temporal Logic
4	LTS	Label Transition System
5	BTL	Branching Temporal Logic
6	DFS	Depth First Search
7	SPIN	Simple Promela Interpreter
8	PROMELA	Protocol / Process Meta Language

DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ

LỜI MỞ ĐẦU

Với sự phát triển nhanh tột bậc của lĩnh vực công nghệ thông tin và truyền thông trên cả các hệ thống phần cứng và phần mềm, khả năng xảy ra nhiều lỗi, đặc biệt là các lỗi tinh vi là rất cao. Những lỗi này có thể gây ra những hậu quả nghiêm trọng về tiền bạc, thời gian, thậm chí cuộc sống của con người. Nhìn chung, một lỗi càng sớm được phát hiện sẽ càng mất ít công sức để sửa lỗi đó.

- Theo thống kê của Standish Group (2000) trên 350 công ty với hơn 8000 dự án phần mềm có: 31% dự án phần mềm bị huỷ bỏ trước khi được hoàn thành. Với các công ty lớn, chỉ có khoảng 9% tổng số các dự án hoàn thành đúng tiến độ và trong ngân sách dự án (với các công ty nhỏ, tỷ lệ này vào khoảng 16%)
- Theo thống kê của PCWeek (2001) trên 365 công ty chuyên cung cấp các dự án phần mềm chuyên nghiệp có: 16% các dự án là thành công, 53% sử dụng được nhưng không thành công hoàn toàn, 31% bị huỷ bỏ.
- NIST Study (2002): Lỗi phần mềm gây thiệt hại ước tính 59.5 triệu đô la cho nền kinh tế nước Mỹ mỗi năm chiếm 0.6% GDP.
- Vệ tinh nhân tạo Ariane-5 vào ngày 4/06/1996 chỉ sau 36 giây rời khỏi bệ phóng đã bị nổ vì lý do lỗi phần mềm: người ta đã sử dụng 16 bit lưu trữ số nguyên để lưu trữ dữ liệu kiểu thực 64 bit gây thiệt hại 500 triệu USD…

Trong các ngành công nghiệp, luôn đặt ra một yêu cầu phát triển các phương pháp luận để có thể tăng độ tin cậy trong việc thiết kế và xây dựng phần mềm. Các phương pháp luận đó sẽ cải thiện chất lượng và hạ giá thành cho việc phát triển một hệ thống. Thêm nữa, về mặt lý thuyết, cần phải cung

cấp một nền tảng toán học chặt chẽ và đúng đắn cho việc thiết kế các hệ thống thông tin, để những người xây dựng và phát triển phần mềm có thể kết hợp giữa kinh nghiệm thực tiễn và lý thuyết.

Một cách tiếp cận truyền thống là xây dựng hệ thống phần mềm bằng cách đi từ xây dựng mô hình. Những mô hình đó sẽ được chỉnh sửa cho đến khi đạt được đến độ tin cậy về tính chính xác và đúng đắn. Cách tiếp cận này có ưu điểm và chi phí thấp hơn so với việc xây dựng hệ thống một cách trực tiếp. Tuy nhiên, nhược điểm của cách tiếp cận này là sự định tính nhập nhằng trong việc xây dựng mô hình.

Cách tiếp cận thứ hai là sử dụng việc xác thực hình thức (Formal Verification) bằng cách xây dựng mô hình toán học của hệ thống, sử dụng một ngôn ngữ để đặc tả các thuộc tính của một hệ thống. Đồng thời cung cấp các phương thức để chứng minh mô hình đó thoả mãn các thuộc tính đề ra. Khi phương thức đó được chứng minh bằng ôtômat, người ta gọi là xác thực tự động (Automation Verification). Tuy nhiên, các phương thức xác thực đó chưa thoả mãn các điều kiện cần có với một công cụ tự động như sau:

- Có cơ sở hình thức để xây dựng được các công cụ có tính thực thi. Công cụ hoặc phương thức đó phải dễ dàng, hữu ích cho người sử dụng. Do đó, các ký hiệu phải rõ ràng, dễ hiểu với người sử dụng, có giao diện thân thiện.
- Có khả năng liên kết giữa các giai đoạn trong vòng đời phần mềm. Dễ dàng tích hợp giữa các pha trong vòng đời phần mềm
- Tính ổn định cao, nhất là với những phần mềm phức tạp.
- Có khả năng phát hiện lỗi và sửa lỗi

Cách tiếp cận thứ 3: Kiểm tra mô hình (Model Checking) là một phương thức có thể đáp ứng được các yêu cầu trên. Các kỹ thuật truyền thống đang được sử dụng như kiểm thử (testing) hoặc mô phỏng (simulation)

thường không tự động, quá phức tạp hoặc chỉ đưa ra kết quả từng phần. Chúng có thể tìm ra rất nhiều lỗi nhưng không thể tìm ra tất cả các lỗi nhất là với những phần mềm tương tranh đa luồng, phần mềm nhúng, phần mềm thời gian thực, phần mềm hướng đối tượng...Khắc phục những nhược điểm đó, các giải thuật kiểm tra mô hình đã cung cấp một cách tiếp cận toàn diện và tự động để phân tích hê thống. *Phương pháp kiểm tra mô hình phần mềm là một kỹ thuật tự động mà: khi cho một mô hình hữu hạn trạng thái của một hệ thống và một thuộc tính hệ thống cần thoả mãn, kiểm tra xem hệ thống đó có thoả mãn thuộc tính đưa ra hay không?[1]*

Với lợi ích to lớn của kiểm tra mô hình đặc biệt là kiểm tra mô hình phần mềm, đây trở thành một vấn đề nóng hổi đang được rất nhiều người quan tâm trên thế giới. Tuy nhiên đây là một vấn đề rất rộng, cộng với tính phức tạp và mới mẻ của nó nên luận văn sẽ giới hạn nghiên cứu trong xây dựng giải thuật kiểm tra mô hình phần mềm tối ưu và có bố cục, nội dung như sau:

Chương 1: Tổng quan về kiểm tra mô hình phần mềm: giới thiệu về định nghĩa, lịch sử ra đời và phát triển của kiểm tra mô hình phần mềm, các vấn đề đang được quan tâm và cần giải quyết xung quanh kiểm tra mô hình phần mềm hiện nay.

Chương 2: Các giải thuật kiểm tra mô hình phần mềm. Trong chương này sẽ đề cập đến các giải thuật kiểm tra mô hình phần mềm đang được áp dụng hiện nay. Từ đó sẽ xem xét và đưa ra kiến trúc và giải thuật đề xuất phù hợp nhất giải quyết các vấn đề đặt ra và cho hiệu năng cao

Chương 3: Đề xuất và xây dựng giải thuật kiểm tra mô hình phần mềm: Đề cập đến các kiến thức nền tảng nhưng rất mới mẻ như hệ thống chuyển trạng thái, lôgic thời gian tuyến tính, Ôtômat Buchi… trên cơ sở lý thuyết đó, sẽ đề xuất xây dựng giải thuật kiểm tra mô hình phần mềm tối ưu nhất.

Chương 4: Xây dựng mô hình minh hoạ: Dựa vào giải thuật đề xuất, lựa chọn công cụ để xây dựng mô hình minh hoạ. Giới thiệu ngôn ngữ PROMELA để mô hình hoá hệ thống và công cụ SPIN để kiểm tra mô hình phần mềm. Đồng thời đưa ra các ví dụ về để minh hoạ cơ chế hoạt động của SPIN với các hệ thống tương tranh.

Kết luận: Tổng kết những gì đã đạt được, đóng góp khoa học của luận văn và hướng mong muốn phát triển trong tương lai của đề tài.

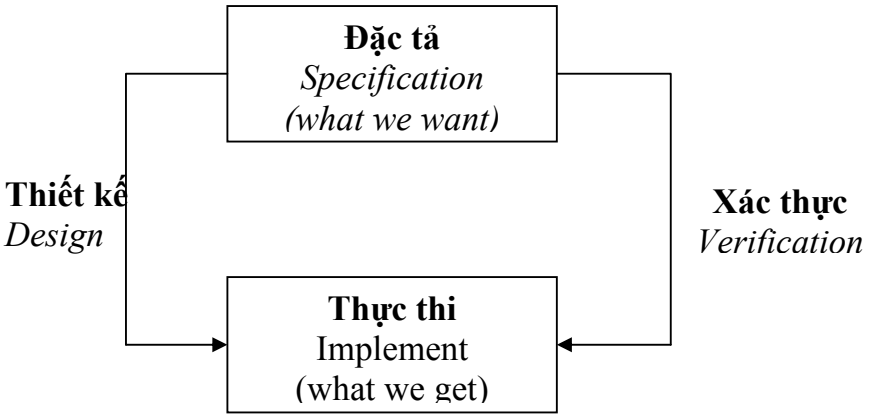
CHƯƠNG I:

TỔNG QUAN VỀ KIỂM TRA MÔ HÌNH PHẦN MỀM

1.1 LỊCH SỬ PHÁT TRIỂN

Kiểm tra mô hình phần mềm đã có lịch sử phát triển từ khá sớm với mục đích đạt được là phải tự động hoá quá trình xác thực các hệ thống, cho đến nay đã phát triển lên thành nhiều phương pháp luận. Từ những khi bắt đầu phát triển theo hướng này, người ta đã xác định được điều kiện tiên quyết của tự động hoá quá trình xác thực gồm 2 yếu tố: *ngữ nghĩa hình thức* (formal semantics) và *ngôn ngữ đặc tả* (specification language). [10]

Trước hết, xác thực là gì? Xác thực là kiểm tra tất cả các hành vi khi thực thi có phù hợp với đặc tả hay không?



Hình 1.1 Mô hình xác thực phần mềm

Thời kỳ đầu tiên, khi các hệ thống thông tin chủ yếu là các hệ thống vào ra, một hệ thống tổng thể là đúng đắn và chính xác nếu từng phần của hệ thống đó đúng và kết thúc hay đầu ra là đúng đắn. Ở thời kỳ đầu tiên này, ngữ nghĩa hình thức chính là mối quan hệ vào ra, ngôn ngữ đặc tả là logic một ngôi.

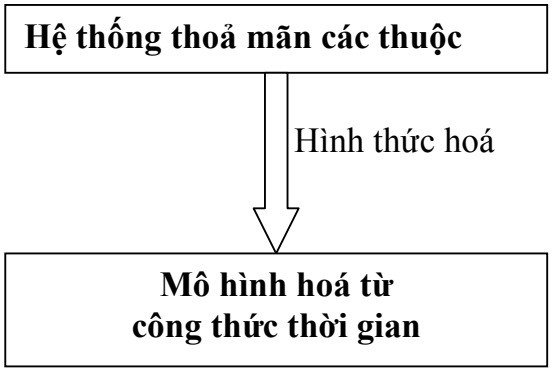
Những năm 60 của thế kỷ 19, khi các hệ thống phản hồi (reactive) xuất hiện, các hệ thống này không phải chỉ đơn thuần là để tính toán, sự kết thúc

có thể không như mong đợi hoặc dễ xảy ra hiện tượng deadlock. Do đó, hệ thống tổng thể là đúng đắn và chính xác nếu nó thoả mãn các yếu tố: an toàn, phát triển và tin cậy. Ngữ nghĩa hình thức chính là Ôtômat, các hệ thống dịch chuyển, ngôn ngữ đặc tả là logic thời gian.

Cùng với sự phát triển của các loại ngôn ngữ lập trình theo xu hướng ngôn ngữ tự nhiên, người ta cố gắng phân tích và đưa ra những kết luận mang tính thể thức và liên quan đến thời gian.

Những năm đầu thế kỷ 20: Logic thời gian được hình thức hoá với các thực thể: always (luôn luôn), sometimes (đôi khi), until (cho đến khi), since (từ khi)…theo trật tự thời gian từ quá khứ, hiện tại cho đến tương lai.

Năm 1977, Pnueli giới thiệu việc sử dụng logic thời gian như một ngôn ngữ đặc tả. Các công thức logic thời gian được thông dịch qua cấu trúc Kripke theo mô hình sau:



Hình 1.2 Mô hình logic thời gian

Trên cơ sở lý thuyết trên bao gồm mô hình hệ thống và logic thời gian, từ đó bắt đầu hình thành ý tưởng về việc tự động hoá quá trình xác thực một vấn đề. Bài toán được phát biểu như sau: Cho một hệ thống M và một công thức thời gian φ, cần tìm một giải thuật để quyết định xem liệu hệ thống M có thoả mãn công thức đó hay không?

Vào cuối những năm 70, đầu những năm 80 người ta thu nhỏ vấn đề kiểm tra một vấn đề qua các bước sau:

- Đưa ra một hệ thống chứng minh để kiểm tra tính đúng đắn dùng logic
- Phân rã hệ thống M thành tập của các công thức F
- Chứng minh rằng F thoả mãn φ bằng cách sử dụng hệ thống chứng minh

Sau đó, vấn đề kiểm tra mô hình được đưa ra gồm các bước sau:

- Xây dựng và lưu trữ mô hình hệ thống M bằng hệ thống trạng thái hữu hạn.
- Kiểm tra mô hình M có thoả mãn φ hay không thông qua định nghĩa.

Từ đó, vấn đề kiểm tra mô hình được đặt ra để giải quyết các vấn đề về bùng nổ trạng thái vì số lượng các trạng thái trong một hệ thống gia tăng với số lượng hàm mũ.

Cuối những năm 80, đầu 90 đã có những kết quả nghiên cứu về vấn đề này:

- Nén (Compress): Biểu diễn tập các trạng thái một cách ngắn gọn như: Lược đồ quyết định nhị phân (Binary decision diagrams)
- Giản lược (Reduce): Không sinh ra những trạng thái không liên quan.
- Trừu tượng (Abstract): Tập hợp các trạng thái tương đương như: biểu đồ xác thực (verification diagrams), biến đổi các tiến trình tương đương.

Cuối những năm 90, đầu những năm 2000: áp dụng kiểm tra mô hình trong các ứng dụng công nghiệp, nhất là thành công trong lĩnh vực xác thực phần cứng, tiên phong là các tập đoàn: IBM, Intel, Microsoft, Motorola,

Samsung, Siement…Có rất nhiều các công cụ thương mại và phi thương mại áp dụng kiểm tra mô hình như: Formal Check, PEP, SMV, SPIN…

Từ những năm 2000 trở lại đây, lĩnh vực kiểm tra mô hình phần mềm rất phát triển và là một chủ đề được rất nhiều các người quan tâm, và điều đặc biệt ở đây, các hệ thống đã được biểu diễn dưới dạng hệ thống vô hạn trạng thái.

1.2 KIỂM TRA MÔ HÌNH PHẦN MỀM

1.2.1 Khái niệm kiểm tra mô hình

***Khái niệm 1**:* Kiểm tra mô hình (Model Checking) là các phương thức, thuật toán để xác thực độ tin cậy và hiệu năng của các hệ thống máy tính. Các phương thức này đối lập với phương thức chứng minh lập luận sử dụng phương pháp suy diễn. [6]

***Khái niệm 2**:* Là một kỹ thuật tự động để xác thực các hệ thống tương tranh hữu hạn trạng thái. [6]

Khái niệm 3: Là một kỹ thuật tự động để xác thực các thuộc tính, hành vi của một mô hình của một hệ thống bằng cách duyệt tất cả các trạng thái của hệ thống đó. [6]

Kiểm tra mô hình được chia làm 2 loại:

- Kiểm tra mô hình phần cứng
- Kiểm tra mô hình phần mềm

Trong khuôn khổ của luận văn, sẽ chỉ xét đến kiểm tra mô hình phần mềm.

1.2.2 Kiểm tra mô hình phần mềm

Kiểm tra mô hình phần mềm (Software model checking) có hai ý nghĩa chính:

- Kiểm tra mô hình phần mềm với mục đích chính là kiểm thử, xác thực xem hệ thống có thoả mãn một số thuộc tính, tính chất nào đó hay

không. Khi đó, hệ thống được biểu diễn dưới dạng đồ thị các trạng thái, gọi là mô hình, các trạng thái này được liên kết với nhau bởi các bước chuyển trạng thái. Mỗi bước chuyển trạng thái tương ứng với một bước của chương trình được biểu diễn bằng toán học ngữ nghĩa hoặc ngôn ngữ máy. Các thuộc tính của phần mềm sẽ được kiểm tra bằng cách duyệt toàn bộ đồ thị.

- Kiểm tra mô hình phần mềm còn mang ý nghĩa logic tính toán nhằm kiểm tra xem hệ thống phần mềm có thể biểu diễn dưới dạng một mô hình công thức logic thời gian (temporal logic) hay không? Do đó, từ mô hình không chỉ mang ý nghĩa là việc đặc tả hành vi một cách trừu tượng mà còn là biểu diễn hành vi của hệ thống.

Trong kiểm tra mô hình phần mềm, các thuộc tính cần thoả mãn được biểu diễn bằng logic thời gian hoặc bằng các Ôtômat. Sau đó, sẽ thực hiện phép duyệt toàn bộ không gian trạng thái để kiểm tra xem hệ thống có thoả mãn các tính chất đó hay không, hay là một mô hình như đặc tả của nó hay không. Vì vậy người ta gọi đó là kiểm tra mô hình. Khi hệ thống và đặc tả của hệ thống được mô hình hoá bằng Ôtômat hữu hạn trạng thái, hệ thống sẽ được so sánh với đặc tả để kiểm tra xem các hành vi của hệ thống có phù hợp với đặc tả hay không.

Do đó, kiểm tra mô hình phần mềm còn được định nghĩa là một kỹ thuật tự động mà: khi cho một mô hình hữu hạn trạng thái của một hệ thống và một thuộc tính hệ thống cần thoả mãn, kiểm tra xem hệ thống đó có thoả mãn thuộc tính đưa ra hay không?

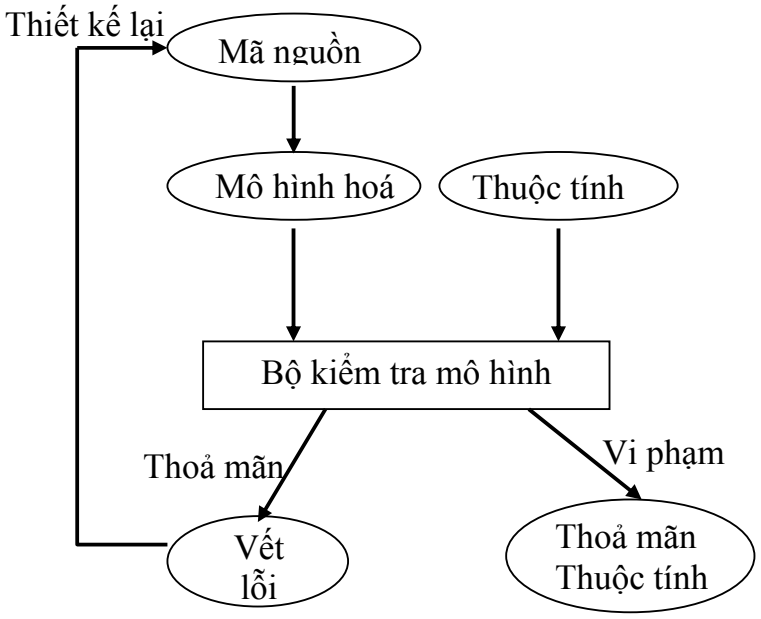
Để kiểm tra mô hình phần mềm sẽ phải qua 3 bước cơ bản sau:

- Mô hình hoá hệ thống (System Modelling): Mô hình hoá hệ thống phần mềm theo phương pháp thủ công hoặc tự động bằng cách phân rã phần

mềm bằng một số trình biên dịch nào đó để có được một mô hình đầy đủ và chính xác.

- Đặc tả các thuộc tính (Properties Specification): Sử dụng một ngôn ngữ nào đó để diễn tả đặc tả hệ thống, thông thường sử dụng logic thời gian hoặc sử dụng Ôtômat.
- Xác thực (Verification): Kiểm tra tính phù hợp, đúng đắn giữa mô hình phần mềm và đặc tả của phần mềm đó.

Các giai đoạn của việc kiểm tra mô hình phần mềm được biểu diễn như sau (hình 1.3):



Hình 1.3 Mô hình của kiểm tra mô hình phần mềm

Từ chương trình nguồn, ta sẽ mô hình hoá chương trình đó. Sau đó, sử dụng bộ kiểm tra mô hình để kiểm tra xem mô hình có thoả mãn thuộc tính đề ra hay không. Nếu không vi phạm, sẽ đưa ra kết luận hệ thống thoả mãn thuộc tính. Ngược lại, nếu không thoả mãn thuộc tính đó, bộ kiểm tra mô hình sẽ chỉ ra những chỗ lỗi và quay lại quá trình thiết kế, lập trình.

Lợi ích của kiểm tra mô hình phần mềm:

- Kiểm tra mô hình phần mềm được ứng dụng trong nhiều lĩnh vực: phần cứng, phần mềm, các hệ thống giao thức, mang lại lợi ích kinh tế cho nhiều ngành khác nhau, đặc biệt trong ngành công nghiệp.
- Cho phép xác thực các thuộc tính với những phần liên quan nhiều nhất tới thuộc tính đó, vì vậy một thuộc tính hay điều kiện phức tạp sẽ được chia nhỏ thành nhiều phần để áp dụng vào nhánh đồ thị nào liên quan đến phần thuộc tính đó nhất nhằm nâng cao tốc độ xử lý.
- Khi thuộc tính bị vi phạm, chương trình sẽ đưa ra các biến đếm của chương trình để chỉ ra lỗi ở trong mô hình
- Không giống như kiểm thử là luôn mong muốn sinh ra các trường hợp kiểm thử để bao gồm nhiều nhất các tình huống hoặc kịch bản có thể, kiểm tra mô hình luôn đảm bảo duyệt được hết tất cả các tình huống, hay tất cả các trạng thái của mô hình.

Kiểm tra mô hình phần mềm còn có một số điểm hạn chế sau:

- Kiểm tra mô hình tập trung chủ yếu vào hướng điều khiển các ứng dụng vì vậy không hướng nhiều vào dữ liệu
- Bất cứ một phép kiểm tra và xác thực nào sử dụng kiểm tra mô hình chỉ tốt khi và chỉ khi mô hình hoá hệ thống đó tốt. Nếu hệ thống đó mô hình hoá không đầy đủ sẽ xảy ra rất nhiều sai sót khi xác thực, hoặc đưa ra các lỗi sai.

Tuy nhiên, kiểm tra mô hình phần mềm là một công cụ hữu hiệu để tìm lỗi và có khả năng tìm được những lỗi tiềm tàng khác.

1.3 PHÂN LOẠI HƯỚNG TIẾP CẬN KIỂM TRA MÔ HÌNH PHẦN MỀM

Có 2 cách tiếp cận kiểm tra mô hình phần mềm: tiếp cận động và tiếp cận tĩnh

1.3.1 Cách tiếp cận động

Thường áp dụng với ngữ nghĩa động, và được coi như sản phẩm của các tiến trình trên Linux. Các tiến trình được kết nối với nhau bằng các toán tử thực thi trên com.objects. Các toán tử trên com.object là nhìn thấy được, ngược lại các toán tử khác là bị ẩn. Khi đó, chỉ các toán tử hiện mới có thể bị khoá. Hệ thống là một trạng thái tổng thể mà các toán tử tiếp theo của mỗi tiến trình đều được hiện. Không gian trạng thái chính là hợp của trạng thái tổng thể và đường đi giữa chúng. [7]

1.3.2 Cách tiếp cận tĩnh

Lặp giữa các quá trình: Trừu tượng (Abstract) - Kiểm tra (Check) – Làm mịn (Refine): [7]

- Trừu tượng hoá (Abstract): sinh ra một máy trừu tượng dựa vào phân tích chương trình tĩnh.
- Kiểm tra (Check): Kiểm tra mô hình với máy trừu tượng
- Làm mịn (Refine): Trừu tượng hoá các vết lỗi của code, sau đó quay trở lại bước 1.

1.3.4 Kết hợp giữa hai cách tiếp cận tĩnh và động

Hai cách tiếp cận tĩnh và động như vừa đề cập có những đặc tính khác biệt nhau như sau:

- Ý tưởng

- Tiếp cận tĩnh: duyệt toàn bộ code để sinh ra một môi trường trừu tượng có thể phân tích sử dụng kiểm tra mô hình
- Tiếp cận động: điều khiển thực thi đa tiến trình

- Ngôn ngữ:
  - Tiếp cận tĩnh: Không yêu cầu thực thi nhưng ngôn ngữ là phụ thuộc vào chương trình
  - Tiếp cận động: Ngôn ngữ độc lập với yêu cầu thực thi chương trình
- Lưu vết lỗi:
  - Tiếp cận tĩnh: Có thể sinh ra các vết lỗi sai, có thể chứng minh được sự đúng đắn của mô hình trên lý thuyết, nhưng chưa chứng minh được trong thực hành
  - Tiếp cận động: Vết lỗi tăng theo khối lượng code

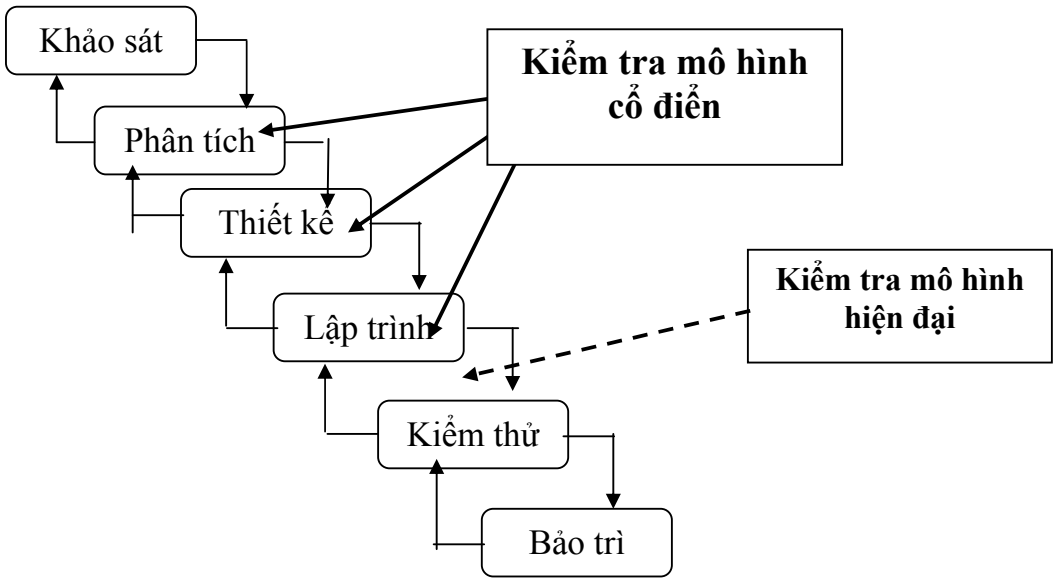
Dựa vào đó, người ta đề xuất một cách tiếp cận kết hợp giữa hai cách tiếp cận tĩnh và động trong kiểm tra mô hình phần mềm để tận dụng được những ưu điểm của cả hai cách tiếp cận đó.

Mô hình kết hợp gồm các bước sau: [7]

- Tự động triển khai giao diện của chương trình từ mã nguồn của chương trình.
- Sinh ra các trình điều khiển kiểm thử (test driver) cho việc kiểm thử ngẫu nhiên thông qua giao diện ở bước 1
- Sinh ra các kiểm thử tự động để thực thi trực tiếp thông qua các đường đi thay đổi của chương trình.

1.4 KIỂM TRA MÔ HÌNH PHẦN MỀM CỔ ĐIỂN VÀ HIỆN ĐẠI

Quy trình phát triển phần mềm theo mô hình thác nước được biểu diễn như sau: [17]



Hình 1.4 Kiểm tra mô hình phần mềm gắn với vòng đời phần mềm

Phương pháp kiểm tra mô hình cổ điển được xây dựng dựa trên 3 pha: phân tích, thiết kế và lập trình. Sau khi phân tích, thiết kế, người ta sẽ mô hình hoá hệ thống, sau đó sử dụng công cụ kiểm tra mô hình phần mềm để kiểm tra xem hệ thống đó có thoả mãn các thuộc tính đặt ra hay không? Nếu có thoả mãn, sẽ đi đến bước lập trình, nếu không, sẽ thiết kế lại mô hình hệ thống. Tuy nhiên, phương pháp kiểm tra mô hình hiện đại xây dựng dựa trên 2 pha: lập trình và kiểm thử. Sau khi lập trình, từ mã nguồn sẽ xây dựng ra mô hình hệ thống dưới dạng mô hình trạng thái, sử dụng công cụ kiểm tra mô hình để tìm ra kết luận. Biện pháp này sẽ thay thế cho việc kiểm thử, vì nó sẽ bao quát được tất cả các trường hợp.

Trong cả hai phương pháp kiểm tra mô hình cổ điển và hiện đại, trừu tượng hoá đều được coi là một hoạt động chính. Ở phương pháp tiếp cận cổ điển từ pha thiết kế, phải trừu tượng hoá một cách thủ công, sau đó từ mô hình xác thực trừu tượng, nhờ kỹ thuật làm mịn sẽ dẫn đến pha thực thi. Ở

phương pháp tiếp cận hiện đại, từ mô hình xác thực trừu tượng, dựa vào kỹ thuật trừu tượng hoá sẽ dẫn đến pha thực thi.

1.5 KẾT LUẬN CHƯƠNG

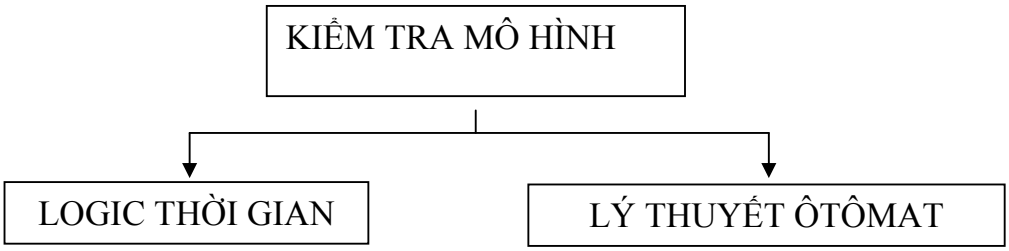
Với mục đích kiểm tra một hệ thống được mô hình hoá có thoả mãn một thuộc tính cho trước hay không, lĩnh vực kiểm tra mô hình phần mềm đã tiến xa hơn kiểm thử tự động vì có thể bao quát được tất cả các trường hợp thuộc hệ thống một cách tự động, do đó là một vấn đề đã và đang rất được quan tâm hiện nay. Kiểm tra mô hình phần mềm đều phải đi qua ba bước đó là mô hình hoá hệ thống, đặc tả các thuộc tính và xác thực tính hệ thống có thoả mãn thuộc tính đó hay không. Để giải quyết từng bước trong các pha đó, có rất nhiều các kỹ thuật kiểm tra mô hình phần mềm được đề xuất nhằm mục đích tối ưu hoá bài toán được trình bày ở chương 2 tiếp theo.

CHƯƠNG 2:

CÁC KỸ THUẬT KIỂM TRA MÔ HÌNH PHẦN MỀM

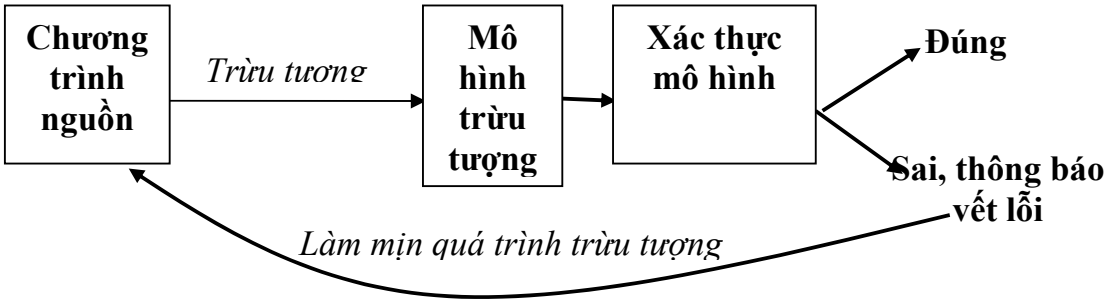
2.1 GIỚI THIỆU

Kiểm tra mô hình dựa trên việc tạo ra mô hình hữu hạn của hệ thống và kiểm tra mô hình đó với các thuộc tính đặt ra của phần mềm. Mô hình của hệ thống được biểu diễn dưới dạng máy trạng thái hữu hạn. Sau đó, ta phải tìm cách để hoàn thành việc duyệt toàn bộ không gian trạng thái để kiểm tra mô hình đó có thoả mãn với đặc tả hay không. Đặc tả hệ thống thường được biểu diễn dưới dạng logic thời gian (temporal logic) hoặc Ôtômat, do đó sẽ có 2 cách tiếp cận việc kiểm tra mô hình: đó là kiểm tra mô hình thời gian và kiểm tra mô hình theo lý thuyết ôtômat (Hình 2.1)



Hình 2.1: Các cách tiếp cận kiểm tra mô hình phần mềm

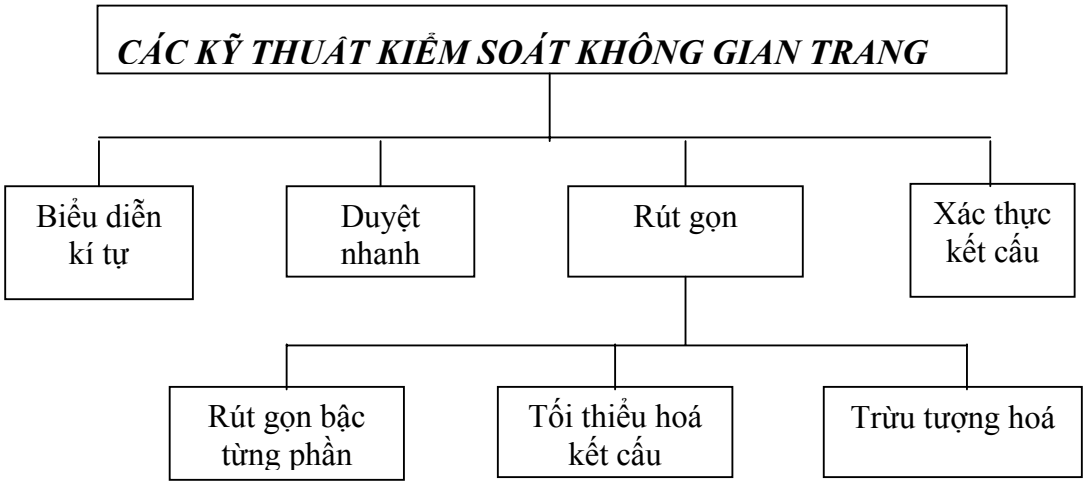
Kiểm tra mô hình phần mềm đang có xu hướng rất đang phát triển hiện nay và thông thường theo các bước sau:



Hình 2.2 Các bước cơ bản trong kiểm tra mô hình phần mềm

- Kiểm tra mô hình tuỳ chọn theo ngôn ngữ lập trình bằng quá trình trừu tượng từ động ở mức độ mã nguồn.
- Trừu tượng và dịch tự động dựa trên sự chuyển đổi sang trừu tượng kiểu mới cho kiểm tra mô hình
- Làm mịn quá trình trừu tượng hầu hết được tự động.

Với bất cứ kỹ thuật kiểm tra mô hình phần mềm nào đều phải giải quyết một vấn đề khó khăn nhất đó là: bùng nổ không gian trạng thái. Không gian trạng thái của việc kiểm tra mô hình thường là tuyến tính nhưng không gian trạng thái của hệ thống lại thường tăng theo hàm mũ (hoặc hơn thế nữa). Do đó, thách thức kỹ thuật chủ yếu trong việc kiểm tra mô hình là thiết kế các phương thức và các cấu trúc dữ liệu để giải quyết được không gian trạng thái lớn như vậy. Có một số phương pháp để có thể tránh sự bùng nổ trạng thái, trong đó có 4 phương pháp chính đó là: Biểu diễn ký hiệu (Symbolic representation), Duyệt nhanh (On the fly), Rút gọn (Reduction), Xác thực kết cấu (Compositional reasoning) (Hình 2.3). [2]

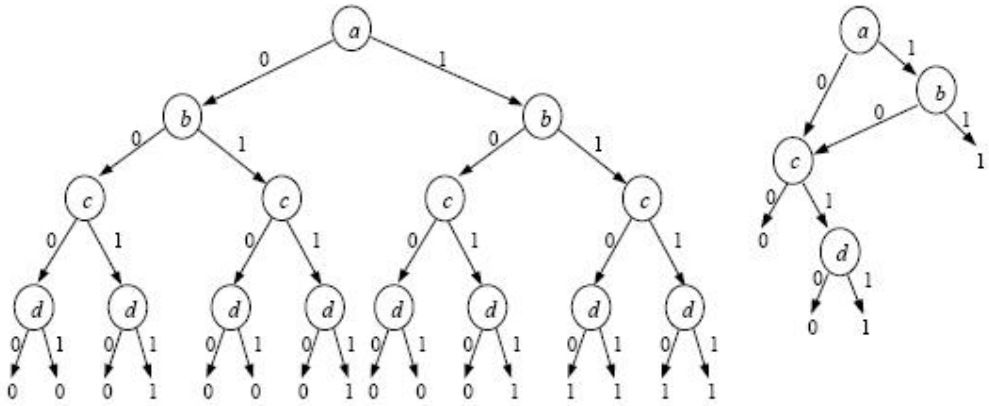


Hình 2.3: Các cách tiếp cận để điều khiển sự bùng nổ không gian trạng thái

Các kỹ thuật biểu diễn ký hiệu tránh việc bùng nổ trạng thái bằng cách thể hiện hệ thống dưới dạng chuyển trạng thái một cách hoàn toàn, sử dụng lược đồ quyết định nhị phân. Vì vậy mô hình của hệ thống được biểu diễn bằng các ký hiệu mà không cần sự xây dựng một cấu trúc dữ liệu hiệu quả. Kỹ thuật duyệt nhanh (On-the-fly) bao gồm việc xác thực hệ thống trong khi sinh ra nó. Nó mô phỏng mọi chuỗi chuyển trạng thái có thể có của hệ thống bằng cách duyệt đồ thị theo chiều sâu mà không cần lưu trữ các dịch chuyển, quá trình tìm kiếm kết thúc sau khi có một lỗi bất kỳ được tìm ra, giúp ta không phải duyệt toàn bộ hệ thống ngay từ đầu. Kỹ thuật giản lược (Reduction) dựa trên việc chuyển đổi vấn đề xác thực sang một vấn đề tương đương nhưng với không gian trạng thái nhỏ hơn. Cuối cùng, đó là kỹ thuật xác thực kết cấu (Compositional Verification) dựa trên việc định nghĩa các thuộc tính cục bộ của các hệ thống con xem có thoả mãn các tính chất đề ra của toàn bộ hệ thống. Bằng cách này, không cần phải sinh đồ thị trạng thái tổng thể, vì các thuộc tính đã được các hệ thống con kiểm tra.

2.2 PHƯƠNG PHÁP KÝ HIỆU BIỂU DIỄN

Phương pháp ký hiệu biểu diễn (Symbolic representation) dựa trên việc sử dụng hoàn toàn mô hình trạng thái hữu hạn để biểu diễn một hệ thống. Cách biểu diễn thông thường là sử dụng kết hợp những hàm và toán tử logic gọi là Lược đồ quyết định nhị phân theo bậc(Ordered Binary Decision Diagrams – OBDD). Cách biểu diễn sử dụng OBDD có 3 ưu điểm chính: phù hợp với những lớp các hàm Boolean lớn, phù hợp với yêu cầu đưa ra đảm bảo thứ tự của biến đầu vào, có thể thao tác trực tiếp để hoàn thành tất cả các toán tử Boolean cơ bản một cách có hiệu quả. [2]



Hình 2.4 : Cây quyết định nhị phân theo bậc và OBDD cho (a ∧b)∨(c∧d) với thứ tự a<b<c<d

Một OBDD tương tự như một cây nhị phân quyết định, ngoại trừ cấu trúc của nó là một đồ thị bán liên thông có hướng, không đơn thuần là một cây, và có một sự quy định chặt chẽ thứ tự xuất hiện của các biến khi cây được duyệt từ gốc tới các lá. Đặc biệt hơn, OBDD biểu diễn một hàm logic *f* bằng cách giảm đi từ cây quyết định thứ tự nhị phân một số cấu trúc liên quan (Hình 2.4). Để lấy được giá trị thực tương ứng với một dãy giá trị của các biến trong f, ta phải duyệt cây nhị phân quyết định từ gốc tới các lá. Tại mỗi nút, giá trị của biến tương ứng sẽ quyết định đường đi tiếp theo: hoặc theo con trái hoặc theo con phải nếu giá trị của các nhãn được đánh nhãn là false/true hoặc 0/1. Do đó, cách thể hiện này được gọi là ký hiệu (symbolic), và giải thuật kiểm tra mô hình làm việc thực hiện thông qua biểu diễn ký hiệu được gọi là kiểm tra mô hình ký hiệu. Các giá trị trên cây xuất hiện theo thứ tự bậc tăng dần từ gốc tới các lá. Mô hình OBDD được tinh giảm từ cây nhị phân quyết định bằng cách hợp các nhánh giống nhau trên cây thành một cây đơn, và loại bỏ bất kỳ nút nào có các con trái hoặc phải là giống nhau. (Hình 2.4)

OBDD là một cấu trúc dữ liệu để biểu diễn ký hiệu của các tập trở nên thông dụng cho việc kiểm tra mô hình bởi vì chúng có những đặc tính sau:

- Mọi hàm Boolean đều là duy nhất, biểu diễn bằng BDD. Nếu bắt buộc phải chia sẻ các nút BDD, sự tương đương giữa hai hàm có thể được quyết định trong một thời gian hằng số.
- Các toán tử Boolean như: phủ định, phép kết nối,…có thể được thực hiện từng phần để giảm tính phức tạp.
- Phép chiếu được thực hiện một cách dễ dàng, trong trường hợp xấu nhất độ phức tạp có thể lên tới hàm mũ

Mô hình trạng thái hữu hạn của một hệ thống có thể biểu diễn dưới dạng OBDD như trên. Mỗi trạng thái được mã hoá bằng một phép gán các giá trị logic cho tập các biến tương ứng của hệ thống. Quá trình xử lý này được thực hiện hoàn toàn trong suốt với người sử dụng bằng các công cụ hỗ trợ phương pháp ký hiệu biểu diễn. Chuyển quan hệ có thể diễn giải bằng các hàm Boolean dưới dạng hai tập các biến, một tập để mã hoá trạng thái hiện thời, và một tập để mã hoá trạng thái mới.

Tiếp cận theo phương pháp ký hiệu biểu diễn tránh được việc xây dựng biểu đồ trạng thái của hệ thống. Do đó, vấn đề không còn là kích cỡ của không gian trạng thái mà chính là kích cỡ của cách thể hiện OBDD. Trong những trường hợp thông thường nó có khả năng xác thực các hệ thống với quy mô lớn nhưng không toàn diện trên tất cả không gian trạng thái.

Các giải thuật dựa OBDD chưa thể thay thế hết các giải thuật khác vì nó không thể hoàn thành tốt trong mọi trường hợp. Trên thực tế, kích cỡ của OBDD chủ yếu dựa vào bậc của biến. Vấn đề ở đây là tìm ra bậc hoặc thứ tự mà trả về cây tối thiểu là một bài toán NP đầy đủ. Có một số các heuristic đã được phát triển để tìm ra một thứ tự biến tốt nếu thứ tự đó tồn tại. Tuy nhiên có rất nhiều các hàm Boolean có kích cỡ là hàm mũ với mọi bậc của biến.

2. 3 PHƯƠNG PHÁP DUYỆT NHANH

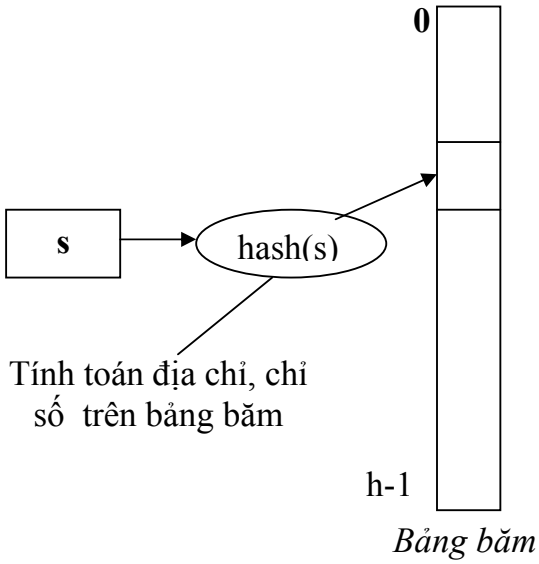
Kỹ thuật duyệt nhanh (On the fly) thực hiện bằng cách hoàn thành tất cả các phép duyệt đến tất cả các trạng thái hoặc các chuyển trạng thái. Do đó, không cần thiết phải lưu trữ toàn bộ đồ thị trạng thái của toàn hệ thống. Trên thực tế, sự bùng nổ không gian trạng thái có thể làm cho hầu hết các hệ thống khó có thể thực thi được. Kỹ thuật này mô phỏng tất cả các chuyển trạng thái có thể của hệ thống có thể thực hiện được. Sau đó, sử dụng giải thuật truyền thống tìm kiếm theo chiều sâu để phân rã, khảo sát hệ thống để thực hiện kỹ thuật duyệt nhanh mà không phải lưu trữ các chuyển trạng thái trong quá trình tìm kiếm. Kỹ thuật này sẽ làm giảm yêu cầu bộ nhớ khi thực hiện. [2]

Trong lần duyệt đồ thị theo chiều sâu đầu tiên, đường đi hiện thời sẽ yêu cầu bộ nhớ nhỏ nhất. Kỹ thuật phải đáp ứng được yêu cầu của bài toán là giảm khối lượng bộ nhớ yêu cầu trong khi vẫn đảm bảo duyệt toàn bộ các trạng thái. Mỗi trạng thái chỉ được lưu trữ một lần khi nó được đến thăm. Do vậy, với các đồ thị phức tạp, sẽ không thể lưu trữ được tất cả các trạng thái. Có rất nhiều các biện pháp được đề nghị để cố gắng dung hoà giữa hai chiến lược trên.

Cộng với việc lưu trữ đường đi hiện thời, bộ nhớ đệm không gian trạng thái (state space caching) tạo ra một bộ đệm gồm các trạng thái đã được đến thăm. Ban đầu tất cả các trạng thái đã đến thăm được lưu trữ cho đến khi nó điền đầy bộ nhớ đệm. Khi đó, các trạng thái cũ sẽ được thay dần dần bằng các trạng thái mới. Hiệu quả của việc sử dụng bộ đệm lưu trữ không gian trạng thái phụ thuộc vào kích cỡ của bộ đệm và phụ thuộc vào cấu trúc của không gian trạng thái. Một nhiệm vụ hết sức phức tạp và không thể đoán trước đó là tìm ra cách thiết lập một bộ đệm tối ưu vì nếu không sẽ làm tăng thời gian thực thi cực nhanh. Như trên đã trình bày, thời gian thực thi cần thiết tăng vọt là do sự bùng nổ gấp nhiều lần của các phần trong không gian trạng thái

không được lưu trữ. Tận dụng tối đa lợi ích đạt được từ việc sử dụng bộ nhớ đệm không gian trạng thái sẽ tránh việc bùng nổ không gian trạng thái và thời gian do việc lưu trữ nhiều lần cùng một phần giống nhau.

Để cùng giải quyết vấn đề bùng nổ không gian trạng thái, kỹ thuật này còn sủ dụng bit trạng thái băm (bit state hashing) hoặc siêu vết (super trace) để thực hiện tìm từng phần trên không gian trạng thái (Hình 2.5). Các trạng thái đã thăm được lưu trữ trong một bảng băm H có kích cỡ phụ thuộc vào bộ nhớ còn trống. Với mỗi trạng thái s sẽ sử dụng một bit đơn h(s), khi đó h là một hàm băm trả về giá trị các bit đánh dấu trong H. Nếu h(s) = 1 có nghĩa là s đã được thăm. Do đó, sẽ không xảy ra hiện tượng đụng độ vì tìm kiếm trên từng phần. Khi thực hiện giải thuật, không gian trạng thái sẽ được bao phủ tăng dần đáng kể với một dãy các bit trạng thái băm.Giải thuật này kết hợp việc chạy song song với hàm băm độc lập tĩnh cho đến khi tất cả các mức của đồ thị đều được đạt tới. Ưu điểm của việc sử dụng bảng băm là tất cả các trạng thái đều được lưu trữ và được tra cứu, tìm kiếm rất nhanh, tiết kiệm được tối đa dung lượng bộ nhớ.



Hình 2.5 Quản lý không gian trạng thái trong kỹ thuật duyệt nhanh

Ưu điểm của kỹ thuật xác thực duyệt nhanh là nó chỉ tiến hành cho đến khi có một lỗi bị phát hiện, trong trường hợp đó, một vết lỗi (counterexample) được sinh ra để chỉ định lỗi và sửa lỗi. Thông thường, lỗi tìm được khá sớm trong quá trình tìm kiếm, do đó tránh được việc phải đi thăm toàn bộ đồ thị. Mặt khác, khi hệ thống chạy đúng, việc tìm kiếm sẽ phải diễn ra trên toàn bộ không gian trạng thái. Vì thế cách tiếp cận này đặc biệt thích hợp với những hệ thống có thể xảy ra rất nhiều lỗi.

2.4. PHƯƠNG PHÁP RÚT GỌN

Các kỹ thuật rút gọn (Reduction) tập trung vào việc xây dựng từng phần, hoặc trừu tượng không gian trạng thái của một chương trình, trong khi phải chứng tỏ được đầy đủ khả năng thoả mãn các thuộc tính của hệ thống. Ta tập trung đi sâu vào rút gọn không gian trạng thái. [2]

2.4.1 Rút gọn bậc từng phần

Mục đích: Giảm số lượng các kết nối độc lập xen vào trong mô hình

Trong hầu hết các tiếp cận kiểm tra mô hình, tính tương tranh được mô hình hoá bởi sự xen nhau, đó là vấn đề chính của sự bùng nổ trạng thái. Rút gọn bậc từng phần (Partial order reduction) được dựa trên việc quan sát các hệ thống tương tranh, hiệu quả tuyệt đối của một tập các hành động thường độc lập với bậc của chúng. Do đó, sẽ tránh sự lãng phí phải sinh ra tất cả các trường hợp xen nhau giữa chúng. Một số phương pháp dựa trên ý tưởng này đã được đề xuất, bằng cách phân rã một đồ thị giản lược của hệ thống mà vẫn đảm bảo được các thuộc tính cần đáp ứng.

Phương thức rút gọn bậc từng phần thực hiện một phép tìm kiếm lựa chọn của hệ thống không gian trạng thái. Với mỗi trạng thái s đến được trong khi tìm kiếm, ta tính một tập con T của tập các chuyển trạng thái tại s, và khảo sát chỉ những chuyển trạng thái trong T. Phương pháp này khác với cách tìm

kiếm truyền thống đó là, ở cách tìm kiếm truyền thống với mỗi trạng thái s, khảo sát tất cả các chuyển trạng thái từ s. Hai kỹ thuật chính này được đề nghị trong các tài liệu về nhận biết tập con của chúng, được tính toán dựa trên các tập liên tục và các tập ngủ (sleep sets).

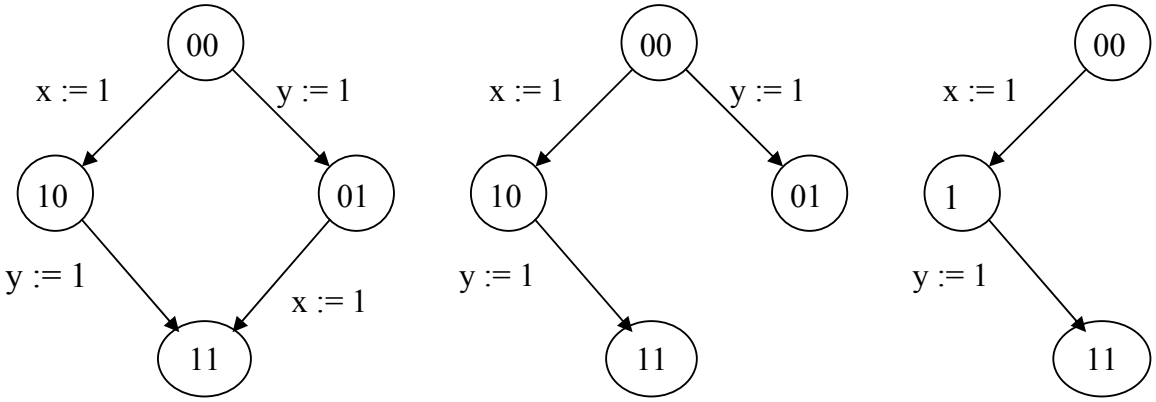
Một tập liên tục (persistent set)T với một số các trạng thái s có chứa các chuyển trạng thái từ trạng thái s, sẽ có một số đặc trưng sau: với bất cứ chuyển trạng thái nào được đến từ trạng thái s bằng việc thực hiện loại trừ các chuyển trạng thái không trong T đều được gọi là các chuyển trạng thái độc lập trong T. Một trong những kỹ thuật cơ bản của tập liên tục là dựa trên việc tính toán của các tập cố định (stubborn). Trong khi giảm sự bùng nổ không gian trạng thái của hệ thống, chỉ các chuyển trạng thái trong tập cố định của mỗi trạng thái được lựa chọn. Nó đã chứng minh rằng sự thực thi của toàn bộ các chuyển trạng thái còn lại có thể trì hoãn không cần kết quả của sự xác thực có hiệu quả hay không. Giải thuật trên được tính toán các tập cố định trong suốt các quá trình duyệt không gian trạng thái và được thực hiện bởi kỹ thuật duyệt nhanh.

Kỹ thuật tập ngủ (sleep set) khai thác thông tin về việc tìm kiếm trong quá khứ. Nếu sử dụng riêng lẻ, nó giảm số lượng các chuyển trạng thái được duyệt nhưng không giảm số lượng các trạng thái. Như đã đề cập, đây là một kỹ thuật rất hữu ích khi các tập ngủ được kết hợp với kỹ thuật bộ đệm. Trong quá trình tìm kiếm theo chiều sâu trên đồ thị của hệ thống, mỗi trạng thái s tương ứng với một tập ngủ, đó là tập các chuyển trạng thái tại s nhưng sẽ không được thực thi từ s. Tập ngủ có thể kết hợp với tập liên tục để giảm không gian trạng thái cần duyệt. Thực tế, kỹ thuật tập liên tục không thể tránh các lựa chọn của các chuyển trạng thái độc lập trong một trạng thái, các tập ngủ không thể tránh được các chuyển trạng thái chèn lên nhau.

Khi kết hợp với kiểm tra mô hình, kỹ thuật rút gọn từng phần cũng biến đổi theo thuộc tính cần phải xác thực. Nó thường trong trường hợp các kỹ thuật rút gọn bậc từng phần được tính toán, hầu hết trong khi tìm kiếm, những phần này của các đồ thị trạng thái là thừa và có thể bỏ qua.

Ví dụ:

x := 1 || y := 1 khởi tạo x = y = 0



Không rút gọn Rút gọn các dịch chuyển Rút gọn trạng thái

Hình 2.6 Minh hoạ phương pháp rút gọn bậc từng phần

2.4.2 Tối thiểu hoá kết cấu

Nhiệm vụ của xác thực hệ thống bao gồm thiết lập một hệ thống S thoả mãn một số thuộc tính f. Gọi R là vùng ngữ nghĩa tương đương với thuộc tính f. Do đó, S thoả mãn f nếu S' thoả mãn f, trong đó S'là một máy trạng thái tối thiểu sao cho (S, S') $\in$ R. Quá trình xây dựng S' từ S được gọi là quá trình tối thiểu hoá. Khi R tương ứng với ứng dụng của một trừu tượng của S, thì S' có chứa ít trạng thái hơn S.

Kỹ thuật phân tích máy trạng thái tối thiểu tương ứng với một số các hệ thống sẽ tốt hơn chính hệ thống đó. Rõ ràng là, mục tiêu để tối thiểu hoá đồ

thị S’ không sinh ra đồ thị đầy đủ của hệ thống. Tối thiểu hoá kết cấu (Compositional minimisation) cung cấp các phương pháp để đạt được điều đó.

Giả sử một hệ thống được mô tả bởi một cây phân cấp. Tối thiểu hoá kết cấu hoàn thành tối thiểu hoá trong các bước, từ mức thấp nhất cho đến mức cao nhất trong cây phân cấp. Biểu thức kết cấu của mỗi mức sẽ định nghĩa những máy trạng thái nào phải được kết hợp để tạo thành các máy trạng thái của những hệ thống con tại mức đó. Kết quả là mỗi kết cấu đều được tối thiểu hoá. Một số các ký hiệu tương đương được sử dụng trong cách tiếp cận này được gọi là một sự tương đẳng với các toán tử trong các biểu thức kết cấu. Điều này đảm bảo các thành phần có thể tạo thành một cách an toàn bằng cách tối thiểu hoá các biểu thức.

Trong quá trình đã được mô tả ở trên, đồ thị trạng thái cho các hệ thống con trung gian được xây dựng bằng cách phân tích những tình huống có thể. Vì vậy, cách tiếp cận tối thiểu hoá kết cấu này có những đặc tính rất phù hợp sau:

- Kết hợp từ mức thấp tới mức cao hơn của các hệ thống: bằng cách tạo thành hành vi thành phần, che giấu chi tiết từ hành vi đối tượng mà toàn bộ hệ thống không cần đến, đặt tên lại cho các hành động của các giao diện trong các thành phần sử dụng với các ngữ cảnh khác nhau.
- Ký hiệu tương đương: Các ký hiệu tương đương thường được dùng để đơn giản hoá các hệ thống trung gian, phải thoả mãn việc bảo vệ các thuộc tính cần quan tâm và giảm được không gian trạng thái.
- Giải thuật rút gọn: Giảm kích cỡ của hệ thống con, để sinh ra các máy trạng thái càng nhanh và nhỏ càng tốt.

2.4.3 Trừu tượng hoá

Hầu hết các chiến lược rút gọn đều dựa trên ứng dụng của một số dạng trừu tượng hoá hệ thống thông qua các phép phân tích. Trong thực tế tối thiểu hoá kết cấu có thể được coi như một kỹ thuật trừu tượng hoá (abstraction). Nó

trừu tượng chi tiết từ hành vi của hệ thống dựa trên mô tả về cấu trúc hệ thống và mô tả các thành phần của nó tương tác với nhau như thế nào.

Rút gọn cục bộ (Localisation Reduction) là một quá trình tự động, thuộc tính phụ thuộc vào kỹ thuật rút gọn được Kurshan đề nghị . Đây là một quá trình động khi ngôn ngữ kiểm tra bao gồm các phương thức xác thực tự động. Ngôn ngữ bao gồm các thuộc tính được bảo vệ khi các tiến trình khác được thêm vào mô hình. Giải thuật được khởi tạo bằng cách trừu tượng hoá hệ thống có chứa chỉ một tập con của các tiến trình hệ thống, sau đó sẽ thực hiện một cách đệ quy cho đến khi các biến đếm chương trình tương ứng với một sự thực thi đúng của hệ thống. Phép lựa chọn của các tiến trình bao gồm quá trình xấp xỉ dựa trên một đồ thị có hướng biểu diễn sự phụ thuộc giữa các tiến trình của hệ thống.

Một cách tiếp cận được đề xuất bởi Bharadwaj và Heitmeyer [6] để kiểm tra các thuộc tính bất biến trên một hệ thống đã được trừu tượng hoá. Những sự trừu tượng hoá này được sinh trực tiếp từ đặc tả hệ thống bằng cách khử các biến trạng thái không ảnh hưởng đến thuộc tính quan tâm. Hệ thống trừu tượng chỉ chứa những biến có liên quan đến bao đóng của tập các biến xuất hiện trong thuộc tính, phụ thuộc vào mối quan hệ giữa các biến hệ thống.

Với những chương trình có hành vi phụ thuộc dữ liệu, Clarke đề xuất thực hiện kiểm tra mô hình dựa trên xấp xỉ không gian trạng thái của chúng, khi đó không gian trạng thái sẽ rất lớn (hoặc có thể là vô hạn). Sự xấp xỉ dựa trên ánh xạ tập trên dãy các biến của chương trình lên tập các giá trị trừu tượng. Chúng được xây dựng trực tiếp từ chương trình mà không xây dựng đầu tiên hệ thống chuyển trạng thái ban đầu.

Một cách tiếp cận khác để trừu tượng hoá bao gồm khai thác sự đối xứng trong hệ thống sinh không gian trạng thái và cho việc kiểm tra mô hình. Nhìn chung, các kỹ thuật cho những chương trình hành vi độc lập dữ liệu

được ứng dụng không nhiều trong các hệ thống tương tranh, ở đó tập trung chủ yếu cho sự tương tác giữa các tiến trình.

2. 5. KỸ THUẬT XÁC THỰC KẾT CẤU

Kỹ thuật xác thực kết cấu (Compositional verification) khai thác dựa trên sự phân rã một hệ thống phức tạp thành các thành phần đơn giản hơn. Các thuộc tính của các thành phần hệ thống được xác thực đầu tiên. Những thuộc tính này sau đó được hợp lại với nhau để suy ra các thuộc tính của hệ thống tổng thể. Rõ ràng là, cách tiếp cận này không phải đối mặt với khó khăn về bùng nổ không gian trạng thái vì nó không yêu cầu phải xây dựng trên không gian trạng thái của hệ thống. Một vấn đề nữa đó là những trạng thái của các hệ thống con chỉ được thoả mãn chỉ khi các giả định được đặt ra trên môi trường đó. Một cách tiếp cận cho vấn dề này là sử dụng giao diện các tiến trình để mô hình hoá môi trường của các hệ thống con. [2]

Một số lượng lớn các nghiên cứu đều dành cho xác thực kết cấu, đưa lại những hi vọng khả quan về việc ngăn chặn sự bùng nổ không gian trạng thái. Giải thuật rút gọn cục bộ có thể coi như một phương pháp xác thực kết cấu đơn giản vì nó sẽ chứng minh các thuộc tính của hệ thống tổng thể bằng cách kiểm tra xem nó có thoả mãn một số các thành phần của hệ thống. Thuận lợi của việc rút gọn cục bộ đó là nó có thể tự động được.

Nhìn chung, đó là một nhiệm vụ phức tạp để phân rã các thuộc tính của một hệ thống tổng thể thành các thuộc tính cục bộ của các thành phần của hệ thống. Hơn nữa, nó phải chứng minh rằng sự phân rã đó là đúng đắn, đó là: phải thoả mãn các thuộc tính cục bộ của các hệ thống con và các thuộc tính tổng thể của hệ thống. Cách tiếp cận này được hỗ trợ bởi các công cụ tự động ở mức độ cao để được sử dụng một cách rộng rãi bởi các kỹ sư phần mềm. Theo các kết quả nghiên cứu, tìm ra một heuristic có ích để quyết định sự

phân rã các thuộc tính của hệ thống tổng thể thành các thuộc tính cục bộ của các hệ thống con là một trong những vấn đề mở trước tiên của lĩnh vực này.

2.6 KẾT LUẬN CHƯƠNG

Để kiểm tra mô hình phần mềm, các kỹ thuật đưa ra đều tuân theo một nguyên tắc chung đó là phải trừu tượng hoá mô hình hệ thống và thuộc tính hệ thống cần thoả mãn. Sau đó, sử dụng bộ kiểm tra mô hình để kiểm tra xem hệ thống có thoả mãn thuộc tính đó hay không. Nếu thoả mãn, đưa ra thông báo thành công, nếu không thoả mãn, đưa ra các vết lỗi để thiết kế lại. Điểm khác nhau cơ bản giữa 4 kỹ thuật đề xuất: biểu diễn ký hiệu, duyệt nhanh, rút gọn, xác thực kết cấu đó là cách xử lý để tránh sự bùng nổ không gian trạng thái của hệ thống. Trong 4 kỹ thuật trên, điều khiển không gian trạng thái hiệu quả nhất là kỹ thuật duyệt nhanh (On the fly). Bằng cách thức sử dụng hàm băm để lưu trữ toàn bộ không gian trạng thái, nhưng quá trình duyệt và tìm kiếm trạng thái lại rất nhanh. Mặt khác kỹ thuật duyệt nhanh không yêu cầu phải lưu trữ các chuyển trạng thái, sử dụng kỹ thuật bộ nhớ cache để tiết kiệm dung lượng bộ nhớ, tăng tốc độ tìm kiếm. Đồng thời với việc dựa trên các ưu điểm lưu trữ của kỹ thuật duyệt nhanh, luận văn sẽ đi sâu nghiên cứu tìm ra giải thuật để giải quyết bài toán kiểm tra mô hình phần mềm sử dụng kỹ thuật duyệt nhanh sẽ được đề cập ở chương 3 tiếp theo.

CHƯƠNG 3:

KỸ THUẬT KIỂM TRA MÔ HÌNH PHẦN MỀM SỬ DỤNG LÝ THUYẾT LOGIC THỜI GIAN TUYẾN TÍNH VÀ ÔTÔMAT BUCHI

3.1. BÀI TOÁN KIỂM TRA MÔ HÌNH PHẦN MỀM

Bài toán đặt ra: Cho một hệ thống chuyển trạng thái T và một thuộc tính f. Cần kiểm tra xem hệ thống T có thoả mãn thuộc tính f hay không?

Ý tưởng giải quyết: [5]

- Chuyển đổi hệ thống chuyển trạng thái T về dạng Ôtômat Buchi, ký hiệu A_T
- Đặc tả thuộc tính f dưới dạng Logic thời gian tuyến tính (LTL – Linear Temporal Logic)
- Lấy phủ định của thuộc tính LTL f là $\neg f$ và chuyển $\neg f$ sang dạng Ôtômat Buchi $A_{\neg f}$
- Kiểm tra giao của các ngôn ngữ được đoán nhận bởi A_T và $A_{\neg f}$ có là rỗng hay không, tức là:
  - $L(A_T) \cap L(A_{\neg f}) = \varnothing$
  - Nếu $L(A_T) \cap L(A_{\neg f}) \neq \varnothing$ chứng tỏ hệ thống chuyển trạng thái T đã vi phạm thuộc tính f, đưa ra vết lỗi.
- **Chú ý:**
  - $L(A_T) \cap L(A_{\neg f}) = \varnothing$ nếu và chỉ nếu $L(A_T) \subseteq L(A_{\neg f})$
  - Cho hai Ôtômat Buchi A_T và $A_{\neg f}$, xây dựng tích chập của hai Ôtômat $A_T \times A_{\neg f}$ như sau:

    $L(A_T \times A_{\neg f}) = L(A_T) \cap L(A_{\neg f})$

- Sau đó, ta kiểm tra ngôn ngữ được đoán nhận bởi Ôtômat Buchi $A_T \times A_{\neg f}$ có bằng rỗng (empty) hay không.

3.2. MÔ HÌNH HOÁ HỆ THỐNG PHẦN MỀM

3.2.1 Vấn đề đặt ra

Ta luôn mong muốn tìm được cách biểu diễn mô hình phần mềm để đáp ứng các vấn đề đặt ra:

- **Có khả năng biểu diễn tương tranh**: Làm thế nào để mô hình hoá các hệ thống trong đó phép chuyển trạng thái có thể được thực hiện bởi các tiến trình khác nhau, các tiến trình tương tranh. Chuyển trạng thái có thể chỉ là một phép chuyển tại một thời điểm hoặc có thể có rất nhiều khả năng chuyển trạng thái tại một thời điểm.
- **Các phép chuyển được mô tả ở mức độ nào là thích hợp nhất?**
  - Mỗi phép chuyển được mô tả bởi một vài câu lệnh
  - Mỗi phép chuyển được mô tả bởi một phép gán hoặc một xác định chắc chắn và cụ thể
  - Mỗi phép chuyển được mô tả bởi một câu lệnh mã máy
  - Mỗi phép chuyển được mô tả bởi một sự thay đổi vật lý
- **Lựa chọn mô hình thực thi:** Mô hình tuyến tính hay mô hình phân nhánh?
  - Mô hình tuyến tính: Tập hợp tất cả các phép thực thi hoàn chỉnh (còn gọi là vết) của hệ thống
  - Mô hình phân nhánh: Phân biệt các cách khác nhau tại mọi điểm trong khi thực thi hệ thống.
- **Các trạng thái hệ thống**: Sử dụng các trạng thái toàn cục hay cục bộ cho các hệ thống tương tranh hoặc phân tán?

- Các trạng thái toàn cục: thể hiện miêu tả tức thì của toàn bộ hệ thống.
- Các trạng thái cục bộ: Thể hiện phép gán các giá trị cho các biến của một tiến trình xử lý đơn lẻ.

Trạng thái hệ thống: Trạng thái để mô tả hệ thống một cách hình thức, để cung cấp một số thông tin tại một thời điểm bất kỳ trong quá trình thực thi hệ thống.

Trạng thái hệ thống sử dụng một trong các thành phần sau: các thực thể trừu tượng như đợi tín hiệu vào (waiting for input) hoặc đang chạy (running), giá trị của các biến chương trình, giá trị của các bộ đếm chương trình, nội dung của dãy các thông điệp, các cờ tiến trình, thông tin lập lịch…

Từ đó, yêu cầu phải có những mô hình toán học để làm cơ sở định nghĩa ngữ nghĩa của logic thời gian, đó chính là hệ thống đánh nhãn dịch chuyển (LTS – Label Transition System)

3.2.2. Hệ thống đánh nhãn dịch chuyển

3.2.2.1 Các định nghĩa

Với mục đích thoả mãn các vấn đề đặt ra như trên, ta sẽ biểu diễn các hành vi của hệ thống bằng đồ thị hữu hạn hoặc vô hạn trong đó các nút là các trạng thái của hệ thống và các cạnh để biểu thị sự dịch chuyển trạng thái.

Định nghĩa hệ thống đánh nhãn dịch chuyển: [1] Hệ thống đánh nhãn dịch chuyển bao gồm bộ bốn : $K = (S, S_0, L, \rightarrow)$

trong đó:

S: tập các trạng thái

S_0: tập các trạng thái khởi đầu

L: tập các nhãn

$\rightarrow$: một quan hệ dịch chuyển $\subseteq S \times L \times S$

Nếu (s, l, s') $\in \rightarrow$ thì sẽ viết là: $s \xrightarrow{l} s'$

Định nghĩa phép thực thi trong LTS: [1]

Một phép thực thi của LTS (S, S_0, L, $\rightarrow$) là một đường đi vô hạn hoặc hữu hạn có dạng:

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} s_3 \ldots$$

với $s_0 \in S_0$ và $(s_i, l_i, s_{i+1}) \in \rightarrow$ với mọi i

Chú ý:

- Các trạng thái có thể được đánh nhãn bằng một tập các biến, mỗi biến biểu thị cho một thuộc tính trạng thái.
- Hệ thống đánh nhãn dịch chuyển hữu hạn được coi như ôtômat hữu hạn không có những trạng thái kết thúc.

Định nghĩa đường đi trong LTS: [1]

Nếu $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \xrightarrow{l_3} \ldots$ là một phép thực hiện vô hạn của T, thì

$\sigma := s_0 s_1 s_2 \ldots \in S^\omega$ được gọi là một đường đi trong T. Tập hợp tất cả các đường đi của T được ký hiệu Path(T).

Định nghĩa biểu diễn dãy trạng thái: [1]

Với mọi k $\in$ N, σ^k biểu thị dãy các trạng thái từ thứ k trở đi của σ:

$\sigma^k := s_k s_{k+1} s_{k+2} \ldots \in S^\omega$

3.2.2.2 Áp dụng mô hình hoá chương trình

- Gọi V là tập hợp các biến của chương trình. Tập các trạng thái của chương trình được cho bởi giá trị của V:

  S := {s | s : V $\rightarrow$N}

- Nếu V = {v_1, …, v_n}, thì s thường được viết là:

  $[v_1 \ \alpha \ s(v_1), \ldots, v_n \ \alpha \ s(v_n)]$

- Những trạng thái khởi đầu trong S_0 có thể được khởi tạo giá trị $S_0 \subseteq S$

Ví dụ như $S_0 := \{s_0\}$ với $s_0(v) := 0$ với mọi $v \in V$

- Các nhãn dịch chuyển trong L được ký hiệu bởi các phép gán có dạng:

  $g \rightarrow (v_1,\ldots,v_n) := (e_1,\ldots,e_n)$

  trong đó:

  - $g \in BExp$ là một biểu thức logic trên V và N
  - $n \geq 1$ và với mọi $i \in \{1,\ldots,n\}$
  - $v_i \in V$
  - $e_i \in AExp$ là một biểu thức toán học trên V và N
- Cho $s \in S$, $s(e) \in N$ và $s(g) \in B$ lần lượt biểu thị giá trị của e và g trong trạng thái s. Do đó, s được mở rộng thành:

  $s: AExp \cup BExp \rightarrow N \cup B$
- $g \rightarrow (v_1,\ldots,v_n) := (e_1,\ldots,e_n)$ thực hiện được trong s nếu $s(g) = true$
- Tập các dịch chuyển được ký hiệu:

  $\rightarrow := \{(s, l, s') \mid l$ thực hiện được trong $s\}$

  với $s' = s[v_i \ \alpha \ s(e_i) \mid i \in \{1,\ldots,n\}]$ và $l = g \rightarrow (v_1, \ldots, v_n) := (e_1,\ldots, e_n)$

Ví dụ 1: Phép chia số tự nhiên

Lập chương trình tuần tự tính kết quả y1:= x1/x2 và phần dư y2 := x1 mod x2 là:

```
1:    y1 :=0;
2:    y2 :=x1;
3:    while  y2 ≥ x2 do
4:          y1 := y1+1;
5:          y2 := y2 - x2
6:    end
```

Tập các biến của chương trình: V := {pc, x1, x2, y1, y2} trong đó pc là biến đếm của chương trình (program counter) để quản lý các bước của chương trình, như ví dụ trên s(pc) ∈ {1,…,6}

Các trạng thái khởi đầu: S_0 := {s ∈ S | s(pc) =1, s(x2) > 0}

Các phép chuyển:

L := { (l_1) pc = 1 → (pc, y1) := (2, 0),
(l_2) pc = 2 → (pc, y2) := (3, x1),
(l_3) pc = 3 ∧ y2 ≥ x2 → pc := 4,
(l_4) pc = 3 ∧ y2 < x2 → pc := 6,
(l_5) pc = 4 → (pc, y1) := (5, y1+1),
(l_6) pc = 5 → (pc, y2) := (3,y2 - x2) }

Ví dụ 2: Kết hợp tính toán

Lập chương trình song song tính:

$$\binom{n}{k} := \frac{n*(n-1)*...*(n-k+1)}{1*2*...*k}$$

```
//Tử số                    ||  //Mẫu số và tổng hợp kết quả
1 :   x1 := n;             ||  6 :   x2 := 0;
2 :   while x1 > n - k do  ||  7 :   while x2 < k do
3 :   x3 := x3  * x1;      ||  8 :   x2 := x2 + 1;
4 :   x1 := x1  - 1        ||  9 :   await n - x1  ≥ x2;
5 :   end                  ||  10 :  x3 := x3/x2
                           ||  11 :  end
```

S_0 := { s ∈ S | s(pc_l) = 1, s(pc_r) = 6, s(n) ≥ s(k) >0, s(x_3) =1}

L := { (l_1) pc_l = 1 → (pc_l, x1) := (2,n),
(l_2) pc_l = 2 ∧ x1 > n - k → pc_l := 3:
(l_3) pc_l = 2 ∧ x1 ≤ n - k → pc_l := 5:

(l_4) $pc_l = 3$ $\rightarrow (pc_l, x3) := (4, x3 \_ x1)$,

(l_5) $pc_l = 4$ $\rightarrow (pc_l, x1) := (2, x1 \_ 1)$,

(l_6) $pc_r = 6$ $\rightarrow (pc_r, x2) := (7, 0)$,

$(l7)$ $pc_r = 7 \wedge x2 < k$ $\rightarrow pc_r := 8$,

$(l8)$ $pc_r = 7 \wedge x2 \geq k$ $\rightarrow pc_r := 11$,

$(l9)$ $pc_r = 8$ $\rightarrow (pc_r, x2) := (9, x2 + 1)$,

$(l10)$ $pc_r = 9 \wedge n - x1 \geq x2$ $\rightarrow pc_r := 10$,

$(l11)$ $pc_r = 10$ $\rightarrow (pc_r, x3) := (7, x3/x2)\}$

3.3 ĐẶC TẢ HÌNH THỨC CÁC THUỘC TÍNH CỦA HỆ THỐNG

3.3.1. Vấn đề đặt ra

- Ta đã biết: Hệ thống được mô hình hoá dưới dạng hệ thống dịch chuyển trạng thái được giới thiệu ở trên.
- Làm thế nào để đặc tả các thuộc tính mà hệ thống cần thoả mãn?

Phép đặc tả đó phải thoả mãn các điều kiện sau:

- Tính chính xác: cú pháp rõ ràng, ngữ nghĩa chính xác, do đó không thể đặc tả theo ngôn ngữ tự nhiên.
- Tính tiện lợi: dễ hiểu, dễ sử dụng với những người như: phân tích yêu cầu, thiết kế hệ thống, lập trình viên, kiểm thử
- Ngắn gọn: đặc tả phải ngắn gọn, súc tích nhưng dễ hiểu.
- Tính hiệu quả: có khả năng kiểm tra hệ thống và các thuộc tính có nhất quán hay không?
- Khả năng diễn giải: Có khả năng diễn giải các thuộc tính
- Dễ chuyển đổi: có thể tự động sinh code từ đặc tả (sử dụng làm mịn từng bước)

Nhận thấy, không có một đặc tả hình thức sẵn có nào có thể đáp ứng được các yêu cầu trên. Người ta đề xuất sử dụng logic thời gian (temporal logic) thay cho việc sử dụng logic tĩnh để biểu diễn mối quan hệ giữa các trạng thái khác nhau trong quá trình thực thi hệ thống.

3.3.2. Logic thời gian

Kiểm tra mô hình thời gian là mô hình phát triển theo cách tiếp cận: các thuộc tính cần đạt được của hệ thống được biểu diễn dưới dạng mệnh đề logic thời gian. Logic thời gian (Temporal Logic) đã được chứng minh là rất hữu ích cho việc đặc tả các hệ thống tương tranh, thời gian thực, hướng đối tượng bởi vì nó có thể mô tả thứ tự của các sự kiện theo thứ tự thời gian mà không cần phải giới thiệu một cách rõ ràng về thời gian. Trong logic thời gian, không sử dụng các toán tử chỉ thời gian quá khứ để xác thực chương trình mà chỉ cần các toán tử liên quan đến hiện tại và tương lai.

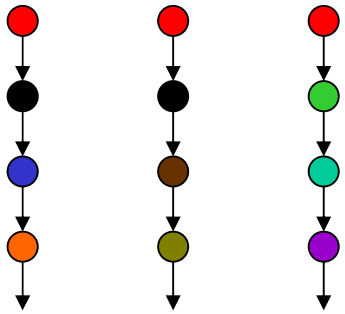
Một khung thời gian (temporal frame) là một cặp (S, R) trong đó S là tập thời gian tại nhiều thời điểm và R là một quan hệ trên S liên quan mỗi thời điểm với bộ xử lý tức thì của nó. Bao đóng phản thân của R, ký hiệu là ≤, biểu diễn thứ tự thời gian: s ≤ t nghĩa là s xảy ra trước t hoặc s và t xảy ra đồng thời trong khoảng thời gian giống nhau. Nguồn gốc của quan hệ R đã nảy sinh ra hai hướng phát triển, hai mô hình thời gian và logic: logic thời gian nhánh và logic thời gian tuyến tính. [2]

3.3.3. Logic thời gian tuyến tính (Linear Temporal Logic - LTL)

Các thuộc tính của hệ thống thường được chuẩn hoá dưới dạng Logic thời gian tuyến tính (LTL). Bất cứ biểu thức logic nào của trạng thái của một hệ thống và các giá trị tương ứng của nó được gọi là một công thức trạng thái.

LTL là một dạng logic hình thức để mô tả mối quan hệ giữa các trạng thái trong quá trình thực thi của hệ thống.

Trong logic thời gian tuyến tính, thời gian là một tập hợp thứ tự tuyến tính, thường được đo bằng các số tự nhiên. Trong một khung tuyến tính (S, R), R là một quan hệ hàm để chỉ định mỗi thời điểm chỉ có một phép kế tiếp. Thứ tự thời gian trong LTL được sử dụng là ký hiệu ≤, ví dụ cho hai khoảng thời gian bất kỳ s, t ∈ S thì hoặc s ≤ t hoặc t ≤ s.



Hình 3.1 : Mô hình Logic thời gian tuyến tính (LTL)

3.3.3.1 Thuộc tính trạng thái

Định nghĩa thuộc tính trạng thái:

- Cho V là một tập các biến của chương trình và S là một tập các trạng thái của chương trình S:= {s | s: V → N}
- Tập hợp các phép toán và các biểu thức logic trên V và N ký hiệu là: *AExp* và *BExp* được định nghĩa như sau:
  - V, N ⊆ *AExp*
  - Nếu $e_1, e_2 \in AExp$ thì $e_1 + e_2, e_1 - e_2, e_1 * e_2 \in AExp$
  - True, False ∈ *BExp*
  - Nếu $e_1, e_2 \in AExp$ thì $e_1 < e_2, e_1 > e_2, e_1 = e_2 \in BExp$
  - Nếu $b_1, b_2 \in BExp$ thì $b_1 \wedge b_2, b_1 \vee b_2, \neg b_1 \in BExp$

  Các phần tử trong *BExp* còn được gọi là các thuộc tính trạng thái.

- Một thuộc tính trạng thái $b \in BExp$ được gọi là hợp lệ trong một trạng thái $s \in S$ nếu $s(b)$ = true (hoặc còn gọi s thoả b, hoặc s là trạng thái b, ký hiệu $s \models b$)
- $b \in BExp$ được gọi là thoả mãn nếu tồn tại một trạng thái $s \in S$ sao cho $s \models b$, b được gọi là một phép lặp thừa nếu $s \models b$ với mọi $s \in S$

3.3.3.2. Cú pháp LTL

Các công thức LTL cơ bản được định nghĩa qui nạp như sau: [1]

- $BExp \subseteq$ LTL
- Nếu $\varphi, \psi \in$ LTL, thì

  $\varphi \wedge \psi \in$ LTL (phép hợp)

  $\varphi \vee \psi \in$ LTL (phép tuyển)

  $\neg \varphi \in$ LTL (phép phủ định)

  $o\ \varphi \in$ LTL (tiếp theo- next)

  $\Diamond \varphi \in$ LTL (tồn tại - eventually)

  $\Box \varphi \in$ LTL (luôn luôn - always)

  φ U $\psi \in$ LTL (cho đến khi - until)

Ý nghĩa:

$b \in BExp$ được biểu diễn là trạng thái đầu tiên

$o\varphi$ đúng nếu φ đúng sau trạng thái đầu tiên

$\Diamond\varphi$ đúng nếu φ đúng ở một số trạng thái phía trước nào đó

$\Box\varphi$ đúng nếu φ đúng với mọi trạng thái phía trước trạng thái nào đó

φ U ψ đúng nếu φ sẽ đúng cho đến một số điểm mà ψ đúng.

3.3.3.3. Ngữ nghĩa của LTL

a) Khái quát về ngữ nghĩa của LTL

- Giả sử có một dãy các trạng thái vô hạn:

$\sigma = s_0 s_1 s_2 ... \in S^\omega$

- Với mọi k ∈ N, σ^k biểu thị dãy các trạng thái từ thứ k trở đi của σ: $\sigma^k := s_k s_{k+1} s_{k+2} ... \in S^\omega$ do đó, $\sigma^0 = \sigma$
- Công thức φ ∈ LTL được thoả mãn ở dãy trạng thái σ^k (hay σ^k thoả σ, kí hiệu $\sigma^k \models \sigma$) được định nghĩa như sau:

  $\sigma^k \models b$ nếu $s_k \models b$

  $\sigma^k \models \varphi \wedge \psi$ nếu $\sigma^k \models \varphi$ và $\sigma^k \models \psi$

  $\sigma^k \models \varphi \vee \psi$ nếu $\sigma^k \models \varphi$ hoặc $\sigma^k \models \psi$

  $\sigma^k \models \neg\varphi$ nếu $\sigma^k \models \varphi$ (σ^k không thoả φ)

  $\sigma^k \models o\varphi$ nếu $\sigma^{k+1} \models \varphi$

  $\sigma^k \models \Diamond\varphi$ nếu tồn tại i ≥ k thoả mãn $\sigma^i \models \varphi$

  $\sigma^k \models \Box\varphi$ nếu với mọi i ≥ k đều thoả mãn $\sigma^i \models \varphi$

  $\sigma^k \models \varphi \, U \, \psi$ nếu tồn tại i ≥ k sao cho $\sigma^i \models \psi$ và $\sigma^j \models \varphi$ với mọi $k \le j < i$
- Hai công thức φ, ψ ∈ LTL được gọi là tương đương (kí hiệu: $\varphi \equiv \psi$) nếu với mọi $\sigma \in S^\omega$: $\sigma \models \varphi$ nếu $\sigma \models \psi$
- Một LTS T là một mô hình của (hoặc thoả) một công thức φ∈LTL nếu $\sigma \models \varphi$ với mọi $\sigma \models Path(T)$. Kí hiệu $T \models \varphi$

Xét một cách cụ thể về ngữ nghĩa của LTL như sau:

b) Thuộc tính trạng thái (State Properties)

$\sigma^k \models b$ nếu $s_k \models b$

Ví dụ: x, y ∈ V, b := (x=y):

σ	s_0	s_1	s_2	s_3	s_4	…
x	1	2	3	4	5	…
y	5	4	3	2	1	…
b	false	false	true	false	false	…

$\Rightarrow \sigma^0 \not\models b, \sigma^1 \not\models b, \sigma^2 \models b$

c) Toán tử tiếp theo o (Next)

$\sigma^k \models o\varphi$ nếu $\sigma^{k+1} \models \varphi$

Ví dụ:

$x, y \in V, \varphi := (x=y)$:

σ	s_0	s_1	s_2	s_3	s_4	…
x	1	2	3	4	5	…
y	5	4	3	2	1	…
φ	false	false	true	false	false	…

$\Rightarrow \sigma^0 \not\models b, \sigma^1 \models ob, \sigma^2 \not\models b$

d) Toán tử tồn tại ◊ (Eventually)

$\sigma^k \models \Diamond\varphi$ nếu tồn tại $i \geq k$ sao cho $\sigma^i \models \varphi$

Ví dụ:

$x \in V, \varphi := (x=2)$:

σ	s_0	s_1	s_2	s_3	s_4	…
x	1	2	3	4	5	…
φ	false	true	false	false	false	…

$\Rightarrow \sigma^0 \models \Diamond\varphi, \sigma^1 \models \Diamond\varphi, \sigma^2 \not\models \Diamond\varphi$

Từ đó rút ra:

- Nếu $\sigma^k \models \Diamond\varphi$ thì $\sigma^i \models \Diamond\varphi$ với mọi $i<k$
- $\Diamond\varphi \equiv \varphi \vee o\Diamond\varphi$

e) Toán tử luôn luôn □ (Always)

$\sigma^k \models \Box\varphi$ nếu với mọi $i \geq k$ luôn có $\sigma^i \models \varphi$

Ví dụ: $x \in V, \varphi := (x > 2)$:

σ	s_0	s_1	s_2	s_3	s_4	…
x	1	2	3	4	5	…
φ	false	false	true	true	true	…

$\Rightarrow \sigma^0 \not\models \Box\varphi, \sigma^1 \not\models \Box\varphi, \sigma^2 \models \Box\varphi$

Từ đó rút ra:

- Nếu $\sigma^k \models \Box\varphi$ thì $\sigma^i \models \Box\varphi$ với mọi $i > k$
- $\Box\varphi \equiv \varphi \wedge o\Box\varphi$
- $\Box$ và $\Diamond$ là hai toán tử đối ngẫu: $\Box\varphi \equiv \neg\Diamond\neg\varphi$

$$\Diamond\varphi \equiv \neg\Box\neg\varphi$$

f) Toán tử cho đến khi U (Until)

$\sigma^k \models \varphi \, U \, \psi$ nếu tồn tại $i \geq k$ sao cho $\sigma^i \models \psi$ và $\sigma^j \models \varphi$ với mọi $k \leq j < i$

Ví dụ:

$x, y \in V$, $\varphi := (x=1)$, $\psi := (y = 2 \vee y = 4)$:

σ	s_0	s_1	s_2	s_3	s_4	…
x	1	2	3	4	5	…
φ	true	false	false	false	false	…
y	5	4	3	2	1	…
ψ	false	true	false	true	false	…

$\Rightarrow \sigma^0 \models \varphi \, U \, \psi, \sigma^1 \models \varphi \, U \, \psi, \sigma^2 \not\models \varphi \, U \, \psi$

Từ đó rút ra:

- $\Diamond\varphi \equiv \text{true} \, U \, \varphi$
- Nếu $\varphi \, U \, \psi$ thoả mãn thì $\Diamond\psi$ cũng thoả mãn

Ví dụ: Biểu diễn tín hiệu đèn giao thông dưới dạng cú pháp LTL

- Tín hiệu đèn giao thông được chuyển đổi giữa các màu xanh, vàng và đỏ như sau:

Giả sử ký hiệu như sau: xanh: gr, vàng: ye, đỏ: re

$gr \rightarrow ye \rightarrow re \rightarrow gr \rightarrow ...$

Thuộc tính thứ nhất: tại một thời điểm chỉ có một tín hiệu đèn duy nhất, do đó:

$\varphi := \Box((gr \vee ye \vee re) \wedge \neg(gr \wedge ye) \wedge \neg(gr \wedge re) \wedge \neg(ye \wedge re))$

- Thuộc tính thứ hai: thứ tự chuyển màu tín hiệu phải đúng, do đó sẽ biểu diễn như sau:

  $\psi := \Box((gr \ U \ ye) \vee (ye \ U \ re) \vee (re \ U \ gr))$

- Đặc tả đầy đủ:

  $gr \wedge \varphi \wedge \psi$

g) Một số phép biến đổi tương đương

Ta có một số phép biến đổi tương đương như sau:

$\neg\Box f \quad \Leftrightarrow \quad \Diamond\neg f$

$\neg\Diamond f \quad \Leftrightarrow \quad \Box\neg f.$

$\Box(f \wedge g) \quad \Leftrightarrow \quad \Box f \wedge \Box g$

$\Diamond(f \vee g) \quad \Leftrightarrow \quad \Diamond f \vee \Diamond g$

$\Box\Diamond(f \vee g) \quad \Leftrightarrow \quad \Box\Diamond f \vee \Box\Diamond g$

$\Diamond\Box(f \wedge g) \quad \Leftrightarrow \quad \Diamond\Box f \wedge \Diamond\Box g$

$f \ U \ (g \vee h) \quad \Leftrightarrow \quad (f \ U \ g) \vee (f \ U \ h)$

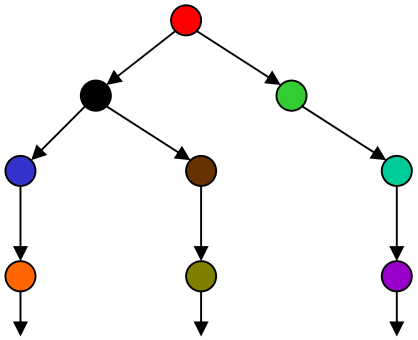
$(f \wedge g) \ U \ h \quad \Leftrightarrow \quad (f \ U \ h) \wedge (g \ U \ h)$

$\neg(f \ U \ g) \quad \Leftrightarrow \quad (\neg g) \ U \ (\neg f \wedge \neg g)$

3.3.4 Logic thời gian nhánh (Branching Temporal Logic - BTL)

Trong LTL, tại mỗi thời điểm chỉ có chính xác một sự kiện kế tiếp. Trong BTL, tại mỗi thời điểm có một hoặc nhiều thời điểm kế tiếp (Hình 3.2), và được biểu diễn dưới dạng hình cây. Do đó, LTL là một trường hợp đặc biệt của BTL.

Trong mô hình thời gian nhánh, mỗi thời điểm t có thể có rất nhiều khả năng tương lai. Với những khả năng tương lai tương ứng là một đường đi được tổ chức từ t. Do đó, một đường đi biểu diễn một khả năng xảy ra trong tương lai. Các toán tử thường biểu thị hoặc là "A" nghĩa là với mọi khả năng trong tương lai diễn tả luôn luôn, chắc chắn hoặc là "E" nghĩa là có thể tồn tại khả năng tương lai diễn tả sự có thể, không chắc chắn. [3]



Hình 3.2: Mô hình cây BTL

Trong khung logic thời gian nhánh, các toán tử là: G(Generally - thông thường), F(Future - tương lai), X (Next - tiếp theo), U (Until - Cho đến khi) ký hiệu tương ứng là □, ◊, o, U. Tuy nhiên, do tính chất phức tạp mô hình BTL ít được sử dụng, các công cụ kiểm tra mô hình phần mềm chủ yếu sử dụng LTL.

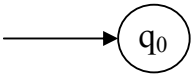
3.4 ÔTÔMAT ĐOÁN NHẬN CÁC XÂU VÔ HẠN

3.4.1 Một số khái niệm ôtômat cổ điển:

Định nghĩa Ôtômat hữu hạn đơn định (Deterministic Finite Automation): Ôtômat hữu hạn đơn định là bộ năm $M = (\Sigma, Q, \Delta, Q_0, F)$ trong đó

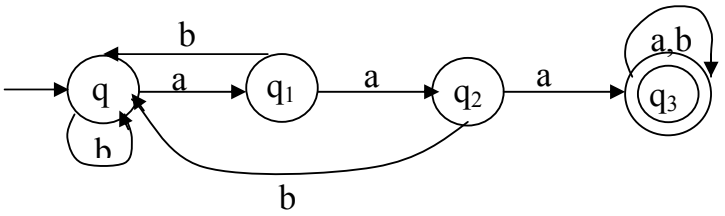
- Σ: bảng chữ vào, là tập hữu hạn các ký hiệu.
- Q: một tập hữu hạn các trạng thái, giả sử $\Sigma \cap Q = \phi$
- Δ: hàm chuyển, là một hàm ánh xạ từ $Q \times \Sigma \rightarrow Q$
- $Q_0 \subseteq Q$ là tập các trạng thái đầu

- F là tập hợp các trạng thái kết thúc $\subseteq$ Q

- Xâu: dãy các ký hiệu ghép liền với nhau
- Đoán nhận xâu: Xuất phát từ trạng thái đầu sau khi đọc hết xâu thì ôtômat chuyển đến một trong những trạng thái kết thúc.
- Biểu diễn ôtômat hữu hạn bằng đồ thị có hướng (sơ đồ trạng thái, sơ đồ chuyển)
    - Tập các đỉnh của đồ thị sẽ biểu diễn các trạng thái và có nhãn là tên của trạng thái đó
    - Trạng thái đầu: 

    - Trạng thái kết thúc: 

Ví dụ: Biểu diễn hình thức đối với ôtômat hữu hạn đơn định



Khi đó:

$\Sigma = \{a, b\}$

$Q = \{q_0, q_1, q_2, q_3\}$

$F = \{q_3\}$

Δ:

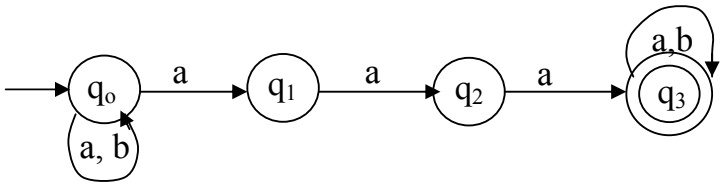
Trạng thái \ Kí hiệu vào	a	b
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_3	q_0
q_3	q_3	q_3

Định nghĩa Ôtômat hữu hạn không đơn định (Nondeterministic Finite Automaton):

Ôtômat hữu hạn không đơn định là bộ 5 phần tử M = (Σ, Q, Δ, Q_0, F) trong đó:

Σ, Q, Q_0, F: tương tự ôtômat hữu hạn đơn định

Δ: là một ánh xạ $Q \times \Sigma \rightarrow 2^Q$



Khi đó:

Σ = {a, b}

Q = {q_0, q_1, q_2, q_3}

F = {q_3}

Δ:

Kí hiệu vào / Trạng thái	a	b
q_0	{q_0,q_1}	{q_0}
q_1	{q_2}	ϕ
q_2	{q_3}	ϕ
q_3	{q_3}	{q_3}

3.4.2 Ôtômat Buchi

Để đặc tả hệ thống thực thi như thế nào, ta không thể chỉ áp dụng lý thuyết ôtômat cổ điển vì ôtômat cổ điển chỉ đoán nhận được những xâu hữu hạn. Giải pháp đặt ra: đưa ra lý thuyết ôtômat có thể đoán nhận các xâu vô

hạn. Như vậy, sau khi đặc tả các thuộc tính của hệ thống bằng LTL, ta chuyển biểu thức LTL đó sang dạng ôtômat đoán nhận xâu vô hạn gọi là Ôtômat Buchi.

Định nghĩa Ôtômat Buchi [1]:

Ôtômat Buchi gồm năm phần tử $A = (\Sigma, Q, \Delta, Q_0, F)$ trong đó

- Σ: bảng chữ vào
- Q: một tập hữu hạn các trạng thái
- Δ: là một hàm ánh xạ từ $Q \times \Sigma \rightarrow 2^Q$, hàm chuyển
- $Q_0 \subseteq Q$ là tập các trạng thái đầu
- $F = \{F_1,\ldots,F_k\} \subseteq 2^Q$ là tập hợp các tập trạng thái kết thúc (tập các trạng thái được chấp nhận)

Ôtômat Buchi tương tự như Ôtômat hữu hạn không đơn định, chỉ khác ở ký hiệu F là tập hợp các tập trạng thái kết thúc.

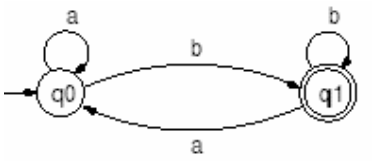
Định nghĩa về điều kiện được đoán nhận:

- Một xâu vô hạn $\sigma \in \sum^w$ được đoán nhận nếu ôtômat chuyển đến ít nhất một trạng thái kết thúc vô hạn lần khi đọc xâu σ đó.
- Ký hiệu $\sigma = s_0s_1s_2\ldots \in \sum^w$ là một từ vô hạn của các biểu tượng đầu vào
- Ký hiệu $\rho = q_0q_1q_2\ldots$là một dãy các trạng thái của A trong đó $q_0 \in Q_0$ và $q_{i+1} = \Delta(q_i, a_i)$ với mọi $i \in N$; ρ được gọi là một đường đi trên σ
- Ký hiệu $Inf(\rho) := \{q \in Q$ sao cho q xuất hiện vô hạn lần trong $\sigma\}$
- Một đường đi ρ trên σ được gọi là chấp nhận được nếu với mọi $1 \le i \le k$ ta có:

$$Inf(\rho) \cap F_i \neq \phi$$

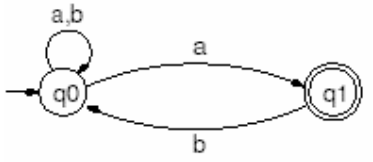
nghĩa là tồn tại F_i xuất hiện một số lần vô hạn trên ρ

Ví dụ 1:



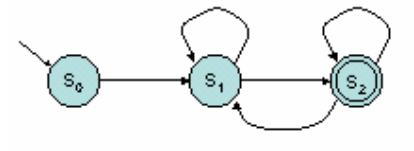
b được xuất hiện vô hạn lần

Ví dụ 2:



a,b được xuất hiện vô hạn lần

Ví dụ 3:



Dựa vào các điều kiện trên ta thấy:

Nếu $\rho_1 = S_0S_1S_2S_2S_2S_2\ldots$ thì ρ_1 được gọi là đường đi được đoán nhận

Nếu $\rho_2 = S_0S_1S_2S_1S_2S_1\ldots$ thì ρ_2 được gọi là đường đi được đoán nhận

Nếu $\rho_3 = S_0S_1S_2S_1S_1S_1\ldots$ thì ρ_3 không được đoán nhận

Ngôn ngữ được đoán nhận bởi ôtômat Buchi: là tập các xâu được ôtômat Buchi đoán nhận $L(A) \subseteq \sum^w$

3.5 CHUYỂN ĐỔI TỪ LTL SANG ÔTÔMAT BUCHI

3.5.1 Tổng quan

Gọi φ là một công thức LTL trên các biểu thức logic thông thường AP (Atomic Proposition)

Từ φ, sinh một Ôtômat Buchi B (trên bảng chữ vào là tập các tập con của AP, ký hiệu 2^{AP}) sao cho L(B) = L(φ). Sau đó, B phải tiếp tục chuyển sang một Ôtômat Buchi chuẩn.

Quá trình xây dựng B khá phức tạp hơn nhiều so với việc chuyển từ những biểu thức thông thường sang ôtômat hữu hạn.

Các bước chuyển đổi:

- Chuyển φ sang dạng chuẩn thông thường.
- Gọi Sub(φ) là tập các biểu thức con của công thức chuẩn hoá đó.
- Các trạng thái của b sẽ là từng cặp tập con của Sub(φ)

3.5.2 Chuẩn hoá về dạng LTL chuẩn

Gọi φ là một công thức LTL trên các biểu thức logic thông thường. Ta nói rằng φ được chuẩn hoá nếu và chỉ nếu:

φ chỉ chứa các biểu thức nguyên tố, toán tử logic thông thường như $\wedge$, $\vee$, $\neg$, các hằng số logic true và false và các toán tử thời gian: o, U, R**,** và toán tử phủ định $\neg$ chỉ xuất hiện phía trước các biểu thức nguyên tố thuộc AP.

Chú ý: Mọi công thức φ đều có thể chuyển sang dạng chuẩn tương ứng φ' (thoả mãn L(φ) = L(φ')) sử dụng các phép biến đổi tương đương ví dụ như:

$\neg(\varphi_1 \, R \, \varphi_2) \equiv \neg\varphi_1 \, U \, \neg\varphi_2$

$\neg(\varphi_1 \, U \, \varphi_2) \equiv \neg\varphi_1 \, R \, \neg\varphi_2$

$\neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2$

$\neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2$

$\neg(o\varphi) \equiv o\neg\varphi$

$\neg\neg\varphi \equiv \varphi$

3.5.3 Biểu thức con

Gọi φ là một biểu thức LTL. Ta định nghĩa tập các biểu thức con Sub(φ) là tập hợp nhỏ nhất của các biểu thức LTL có chứa φ và thoả mãn các điều kiện sau:

Nếu $\varphi_1 \vee \varphi_2 \in Sub(\varphi)$ thì $\varphi_1, \varphi_2 \in Sub(\varphi)$

Nếu $\varphi_1 \wedge \varphi_2 \in Sub(\varphi)$ thì $\varphi_1, \varphi_2 \in Sub(\varphi)$

Nếu $o\varphi_1 \in Sub(\varphi)$ thì $\varphi_1 \in Sub(\varphi)$

Nếu $\varphi_1 U \varphi_2 \in Sub(\varphi)$ thì $\varphi_1, \varphi_2 \in Sub(\varphi)$

nếu $\varphi_1 R \varphi_2 \in Sub(\varphi)$ thì $\varphi_1, \varphi_2 \in Sub(\varphi)$

3.5.4 Chuyển đổi từ LTL sang Ôtômat Buchi

3.5.4.1 Giải thuật chuyển đổi từ LTL sang Ôtômat Buchi

Với 1 biểu thức LTL φ trên tập các biểu thức logic thông thường AP, xây dựng 1 máy Buchi B sao cho mọi xâu được chấp nhận bởi B đều thỏa φ.

Ta gọi $B = (\Sigma, S, \Delta, S_0, F)$ là Ôtômat Buchi được chuyển đổi từ công thức φ với:

$\Sigma = 2^{AP}$ là bảng chữ vào và ở đây chính là tập các tập con của AP

S_0 = {init}, Trạng thái đầu chỉ gồm 1 trạng thái thêm vào

$S = \{init\} \cup (2^{Sub(\varphi)} \times 2^{Sub(\varphi)})$

(các trạng thái tiếp đó là từng cặp các biểu thức con)

Δ là tập các luật chuyển trạng thái $(s, \sigma, t) : s, t \in S, \sigma \in \sum$

F: Tập các trạng thái kết thúc

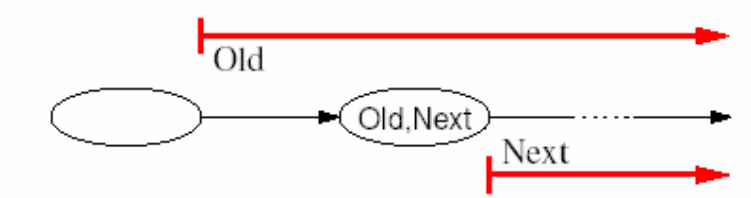
Cụ thể giải thuật chuyển đổi từ LTL sang Ôtômat Buchi gồm những bước sau:

Bước 1: Tìm bảng chữ vào cho Ôtômat Buchi mới được sinh ra, là tập các tập con của AP

Bước 2: Đặt trạng thái S_0 = init

Bước 3: Tìm tập các trạng thái S của Ôtômat Buchi

S là tập các trạng thái của B, mỗi trạng thái s được đặc trưng bởi 2 tập Old và Next là các tập con của Sub(φ) trong đó Old chứa các biểu thức được thỏa bởi 1 xâu chạy đạt đến s, Next chứa các biểu thức được thỏa bởi xâu chạy từ s về sau.



Hình 3.3 Tập các trạng thái của Ôtômat Buchi

- Để đưa ra được tập các trạng thái S và các hàm chuyển Δ, ta định nghĩa hàm Expand có dạng Expand (Old, New, Next)
- Các trạng thái được sinh ra 1 cách đệ quy (hàm expand(Old, New, Next)). Giả sử hiện tại ta có trạng thái s = (Old, Next). Khi đó từ s sẽ có thể dịch chuyển đến các trạng thái được sinh ra bởi hàm expand(∅, Next, ∅).
- Hàm expand(Old, New, Next) được định nghĩa 1 cách đệ quy: ban đầu sẽ gọi hàm expand(∅, { φ }, ∅). Mỗi trạng thái s = (Old, Next) được sinh ra ta lại gọi tiếp hàm expand(∅, Next, ∅) để sinh ra các hàm tiếp theo (các trạng thái không được trùng lặp). Tại mỗi bước đệ quy, hàm Expand(Old, New, Next) sẽ chuyển các biểu thức từ New sang Old để tiếp tục tính toán.
- Từ trạng thái {init} có thể dịch chuyển đến các trạng thái được sinh ra bởi

  expand(∅, {φ}, ∅)

Hàm Expand:

Trường hợp kết thúc:

Expand (Old, {}, Next) = {(Old, Next)}

Một số trường hợp đơn giản:

Expand (Old, New ∪ {p}, Next) = Expand (Old ∪ {p}, New, Next)

nếu p ∈ AP và ¬p ∉ Old

Expand (Old, New $\cup$ {$\neg$p}, Next) = Expand (Old $\cup$ {$\neg$p}, New, Next)
nếu p $\in$ AP và p $\notin$ Old

Expand (Old, New $\cup$ {p}, Next) = {} nếu p $\in$ AP và $\neg$p $\in$ Old

Expand (Old, New $\cup$ {$\neg$p}, Next) = {} nếu p $\in$ AP và p $\in$ Old

Expand (Old, New $\cup$ {true}, Next) = Expand (Old, New, Next)

Expand (Old, New $\cup$ {false}, Next) = {}

Một số trường hợp đơn giản khác:

Expand (Old, New $\cup$ {oφ_1}, Next) = Expand (Old $\cup$ {oφ_1}, New, Next {φ_1})

Expand (Old, New $\cup$ {$\varphi_1 \wedge \varphi_2$}, Next)

= Expand (Old $\cup${$\varphi_1 \wedge \varphi_2$}, New $\cup${$\varphi_1 \wedge \varphi_2$},Next)

Trường hợp phức tạp: Phép hoặc

Expand (Old, New $\cup$ {$\varphi_1 \vee \varphi_2$}, Next)

= Expand (Old $\cup$ {$\varphi_1 \vee \varphi_2$}, New $\cup$ {φ_1}, Next)

$\cup$ Expand (Old $\cup$ {$\varphi_1 \vee \varphi_2$}, New $\cup$ {φ_2}, Next)

Đối với toán tử Until U, xuất phát từ công thức:

φ_1 U $\varphi_2 \equiv \varphi_2 \vee (\varphi_1 \wedge$ o (φ_1 U φ_2))

Do đó, chúng ta sẽ áp dụng hàm Expand cho toán tử Until như sau:

Expand (Old, New $\cup$ {φ_1 U φ_2}, Next)

= Expand (Old $\cup$ {φ_1 U φ_2}, New $\cup$ {φ_2}, Next)

$\cup$ Expand (Old $\cup${φ_1 U φ_2}, New $\cup$ {φ_1}, Next)

Tương tự, với toán tử Release R ta có:

φ_1 R $\varphi_2 \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge$ o (φ_1 R φ_2))

Do đó:

Expand (Old, New $\cup$ {φ_1 R φ_2}, Next)

= Expand (Old $\cup$ {φ_1 R φ_2}, New $\cup$ {φ_1, φ_2}, Next)

$\cup$ Expand (Old $\cup${φ_1 R φ_2}, New $\cup$ {φ2}, Next $\cup$ {φ_1 R φ_2})

Bước 4: Tìm các chuyển trạng thái Δ

Từ trạng thái s = (Old, Next) có thể dịch chuyển đến trạng thái s' = (Old', Next') với ký hiệu vào $\sigma \in \sum$ phải thỏa mãn điều kiện sau:

- σ phải chứa tất cả các biểu thức atomic p ($p \in AP$) thuộc Old'.
- σ không chứa các biểu thức atomic p ($p \in AP$) mà $\neg p$ thuộc Old'.

Từ đó ta xây dựng hàm chuyển Δ chính là bộ (s, σ, t)

Bước 5: Tìm tập các trạng thái kết thúc

Nếu Sub(φ) chứa các biểu thức dạng $\varphi_1 U \varphi_2$ thì F chứa các trạng thái s = (Old, Next) sao cho hoặc $\varphi_2 \in$ Old hoặc $\varphi_1 U \varphi_2 \notin$ Old.

3.5.4.2. Ví dụ

a) Ví dụ 1 Xây dựng Ôtômat Buchi cho p

Ta thấy p là biểu thức nguyên tố, AP = {p}

Bước 1. $\sum = 2^{AP} = \{\{\}, \{p\}\}$.

Bước 2. $S_0 = \{init\}$

Bước 3. Sinh ra các trạng thái.

- Đầu tiên, gọi hàm Expand ({},{φ},{}) tức gọi hàm Expand({},{p}, {})

  Theo định nghĩa đệ quy ở trên:

  Expand(Old, New ∪ {p}, Next} = Expand(Old∪{p}, New, Next)

  Áp dụng ta có:

  Expand({}, {p}, {}) = Expand({p}, {}, {}) = ({p}, {})

  (ứng với trường hợp kết thúc)

  Lúc này B có 2 trạng thái s_0 = (init) và s_1 = ({p},{}) và sự chuyển trạng thái $s_0 \rightarrow s_1$.

- Theo luật từ trạng thái s = (Old, Next) gọi tiếp đến hàm expand({}, Next, {})

Do đó, từ trạng thái $s_1 = (\{p\}, \{\})$ (Old = {p} và Next = {}), gọi tiếp hàm expand({}, {}, {})

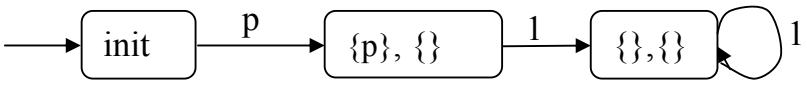
Ta có: Expand({}, {}, {}) = ({},{}), sẽ sinh ra trạng thái *s_2 = ({}, {})* và sự chuyển trạng thái $s_1 \rightarrow s_2$.

- Từ trạng thái $s_2 = (\{\}, \{\})$ gọi đến hàm expand({}, {}, {}) = ({},{}), không sinh ra trạng thái mới (vì trạng thái $s^2 = (\{\}, \{\})$ đã được sinh ra).

Bước 4. Tìm các chuyển trạng thái

- (s_0, σ, s_1), với $\sigma = \{p\}$ vì $Old(s_1) = \{p\}$
- (s_1, σ, s_2) với $\sigma = \varnothing, \{p\}$ vì $Old(s_2) = \varnothing$ nên σ không có ràng buộc nào.
- (s_2, σ, s_2) với $\sigma = \varnothing, \{p\}$ vì $Old(s_2) = \varnothing$ nên σ không có ràng buộc nào.

Do đó, Ôtômat Buchi cho p được biểu diễn như sau:



b)Ví dụ 2 Xây dựng Ôtômat Buchi cho $p \wedge oq$

AP = {p, q}

Bước 1. $\sum = 2^{AP} = \{\varnothing, \{p\}, \{q\}, \{p,q\}\}$

Bước 2. $S_0 = \{s_0\} = \{init\}$

Bước 3. Sinh ra các trạng thái.

- Đầu tiên, gọi hàm Expand($\varnothing$, {$p \wedge Xq$} , $\varnothing$)

Theo định nghĩa

Expand(Old, New$\cup${$p \wedge q$}, Next) = Expand(Old$\cup${$p \wedge q$}, New$\cup${p, q}, Next)

Do đó, Expand($\varnothing$, {$p \wedge Xq$} , $\varnothing$)

= Expand({$p \wedge Xq$}, {p, Xq}, $\varnothing$)

= Expand({$p \wedge Xq$, p}, { Xq}, $\varnothing$)

= Expand({$p \wedge Xq$, p, Xq}, $\varnothing$, {q})

vì Expand(Old, New ∪ {Xq}, Next) = Expand(Old ∪ {Xq}, New, Next∪{q}) = ({p ∧ Xq, p, Xq },{q}) (trường hợp kết thúc)

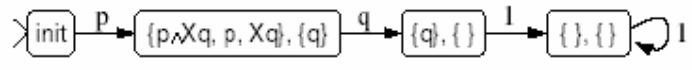
Như vậy sinh ra trạng thái s_1 = ({p ∧ Xq, p, Xq },{q}) và sự chuyển trạng thái từ s_0 sang s_1.

- Từ trạng thái s_1, gọi hàm Expand(∅, {q}, ∅) = expand({q}, ∅, ∅) -> sinh ra trạng thái s_2 = ({q}, ∅) và sự chuyển trạng thái s_1 sang s_2.
- Từ trạng thái s_2 = ({q}, ∅) gọi hàm expand(∅,∅,∅) sinh ra trạng thái s_3 = (∅,∅) và sự chuyển trạng thái s_2 sang s_3.
- Từ trạng thái s_3 = ({}, {}) gọi đến hàm expand({}, {}, {}) = ({},{}), không sinh ra trạng thái mới (vì trạng thái s_3 = ({}, {}) đã được sinh ra).

Bước 4. Chuyển trạng thái

- (s_0, σ, s_1) với σ = {p} vì Old(s_1) = {p ∧ Xq, p, Xq } chứa p thuộc AP.
- (s_1, σ, s_2) với σ = {q} vì Old(s_2) = {q} chứa q thuộc AP.
- (s_2, σ, s_3) với σ = ∅, {q}, {p}, {p,q} vì Old(s_3) = ∅, không chứa biểu thức nào thuộc AP nên không có ràng buộc gì cho σ.
- (s_3, σ, s_3) với σ = ∅, {q}, {p}, {p,q} vì Old(s_3) = ∅, không chứa biểu thức nào thuộc AP nên không có ràng buộc gì cho σ.

Do đó, Ôtômat Buchi cho p ∧ oq được biểu diễn như sau:



c) Ví dụ 3: Xây dựng Ôtômat Buchi cho pUq

AP = {p, q}

Bước 1. $\Sigma = 2^{AP}$ = {{}, {p}, {q}, {p,q}}

Bước 2. S_0 = {s_0} = {init}

Bước 3. Sinh ra các trạng thái.

- Đầu tiên, gọi hàm Expand({}, {p U q} , {})

Theo định nghĩa:

Expand({},{pUq},{}) = Expand ({pUq},{q},{})∪Expand({pUq},{p},{pUq})

Ta có:

Expand ({pUq}, {q}, {}) = Expand ({p U q} ∪ {q}, {}, {})

= ({p U q} ∪ {q}, {})

= ({p U q, q}, {})

Expand ({p U q},{p},{pUq}) = Expand ({pUq} ∪ {p}, {}, {pUq})

= ({p U q} ∪ {p}, {p U q})

= ({p U q, p}, {pUq})

Do đó:

Expand ({}, {pUq},{}) = ({p U q, p}, {pUq}) ∪ ({p U q, q}, {})

Như vậy sinh ra các trạng thái s_1 = ({pUq, p}, {pUq}) và s_2 = ({p U q, q}, {}) và sự chuyển trạng thái từ s_0 sang s_1 và từ s_0 sang s_2

- Từ trạng thái s_1 = ({p U q, p},{pUq}) , gọi hàm Expand ({},{pUq},{}) (là bài toán ban đầu) sẽ sinh ra các trạng thái s_1 và s_2 và sự chuyển trạng thái s_1 đến s_2 và s_1 đến chính nó.
- Từ trạng thái s_2 = ({p U q, q}, {}) , gọi hàm Expand ({},{},{}) sinh ra trạng thái s_3 = ({},{}) và sự chuyển trạng thái từ s_2 đến s_3
- Từ trạng thái s_3 = ({}, {}) gọi đến hàm expand({}, {}, {}) = ({},{}), không sinh ra trạng thái mới (vì trạng thái s_3 = ({}, {}) đã được sinh ra).

Bước 4. Chuyển trạng thái:

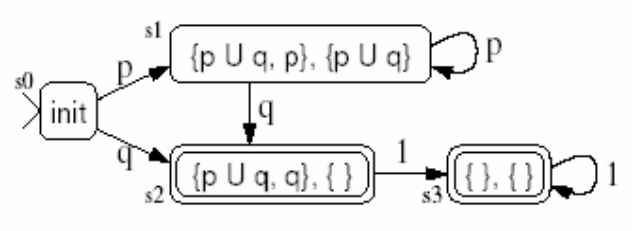
- (s_0, σ, s_1) với σ = {p} vì Old(s_1) = {p U q, p } chứa p thuộc AP.
- (s_0, σ, s_2) với σ = {q} vì Old(s_2) = {p U q, q} chứa q thuộc AP.
- (s_1, σ, s_2) với σ = {q} vì Old(s_2) = {p U q, q} chứa q thuộc AP.
- (s_1, σ, s_1) với σ = {p} vì Old(s_1) = {p U q, p }chứa p thuộc AP.

- (s_2, σ, s_3) với σ = {}, {q}, {p}, {p,q} vì $Old(s_3)$ = {}, không chứa biểu thức nào thuộc AP nên không có ràng buộc gì cho σ.
- (s_3, σ, s_3) với σ = {}, {q}, {p}, {p,q} vì $Old(s_3)$ = {}, không chứa biểu thức nào thuộc AP nên không có ràng buộc gì cho σ.

Bước 5. Trạng thái kết thúc

Theo định nghĩa tính trạng thái kết thúc F, ta thấy F = $\{\{s_2,s_3\}\}$

Do đó, Ôtômat Buchi cho p U q được biểu diễn như sau:



3.6. CHUYỂN TỪ HỆ THỐNG CHUYỂN TRẠNG THÁI SANG ÔTÔMAT BUCHI

Cho một hệ thống chuyển trạng thái T = (S, S_0, R) bao gồm: tập các biểu thức nguyên tố AP và một hàm đánh nhãn L: S × AP → {True, False}

Ôtômat Buchi được sinh ra bởi hệ thống chuyển trạng thái T có ký hiệu là $A_T = (\Sigma_T, S_T, \Delta_T, S_{0T}, F_T)$ trong đó: [14]

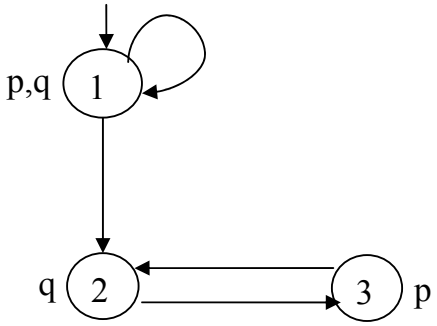
- $\Sigma_T = 2^{AP}$ là bảng chữ vào và ở đây chính là tập các tập con của AP
- S_{0T} = {init}, Trạng thái đầu chỉ gồm 1 trạng thái thêm vào, init không thuộc S
- S_T= {init} ∪ S
- F_T = {init} ∪ S Tất cả các trạng thái của A_T đều là các trạng thái kết thúc
- Δ_T là tập các luật chuyển trạng thái (s, σ, s') : s, t ∈ S_T, σ ∈ $\sum_T$ được tính như sau:

$(s, \sigma, s') \in \Delta_T$ nếu hoặc $(s, s') \in R$ và $L(s',\sigma) = true$

hoặc $s = i$ và $s' \in S_0$ và $L(s',a) = true$

Ví dụ:

Chuyển từ hệ thống chuyển trạng thái sau sang dạng Ôtômat Buchi:



Mỗi trạng thái đều được đánh nhãn tương ứng với các biểu thức điều kiện tại trạng thái đó.

Dựa vào cách chuyển đổi trên ta xây dựng được Ôtômat Buchi gồm có:

Ta có:

AP = {p,q}

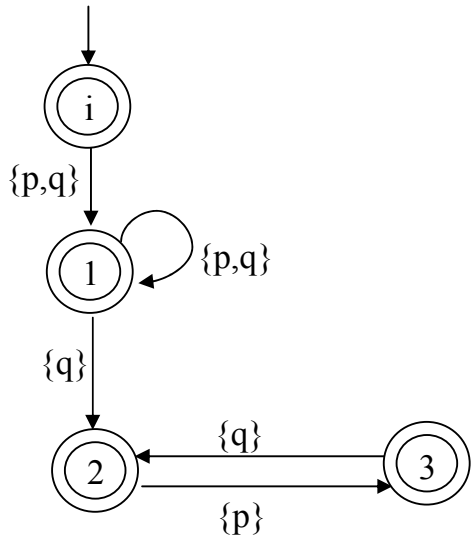
$\Sigma_T = 2^{AP}$ = {{},{p},{q},{p,q}}

S_0 = {init} Thêm trạng thái init

$S_T = F_T$ = {{1},{2},{3},{init}}

Δ_T gồm các luật chuyển trạng thái sau: ({init},{p,q},{1}), ({1},{p,q}{1}), ({1},{q},{2}), ({2}, {p},{3}), ({3},{q},{2})

Ôtômat Buchi mới được sinh ra như sau:



3.7. TÍCH CHẬP CỦA HAI ÔTÔMAT BUCHI

3.7.1 Ôtômat Buchi dẫn xuất

Định nghĩa Ôtômat Buchi dẫn xuất:

Ôtômat Buchi dẫn xuấtgồm năm phần tử $A= (\Sigma, S, \Delta, S_0, F)$ trong đó [8]

- Σ: bảng chữ vào
- S: một tập hữu hạn các trạng thái
- Δ: là hàm chuyển $S \times \Delta \times S$
- $S_0 \subseteq S$ là tập các trạng thái đầu
- $F = \{F_1,\ldots,F_k\} \subseteq 2^S$ là các tập hợp các tập trạng thái kết thúc (tập các trạng thái được chấp nhận) trong đó $F_i \subseteq S$ với mọi $1 \leq i \leq k$ (đây chính là điểm khác với Ôtômat Buchi thông thường)

Chú ý: Cho Ôtômat Buchi dẫn xuất A, một đường đi r được gọi là chấp nhận được nếu và chỉ nếu: với mọi $1 \leq i \leq k$, $\inf(r) \cap F_i \neq \varnothing$, tức là ít nhất một trong số các trạng thái trong tập Fi đều được thăm vô hạn lần

3.7.2 Nguyên tắc thực hiện

Cho hai Ôtômat Buchi $A_1 = (\Sigma, S_1, \Delta_1, S_{01}, F_1)$ và $A_2 = (\Sigma, S_2, \Delta_2, S_{02}, F_2)$

Tích chập của hai Ôtômat Buchi $A_1 \times A_2 = (\Sigma, S_2, \Delta_2, S_{02}, F_2)$ được định nghĩa như sau:

$S = S_1 \times S_2$

$S_0 = S_{01} \times S_{02}$

$F = \{F_1 \times S_2, S_1 \times F_2\}$ (Ôtômat Buchi dẫn xuất)

Δ được xây dựng như sau:

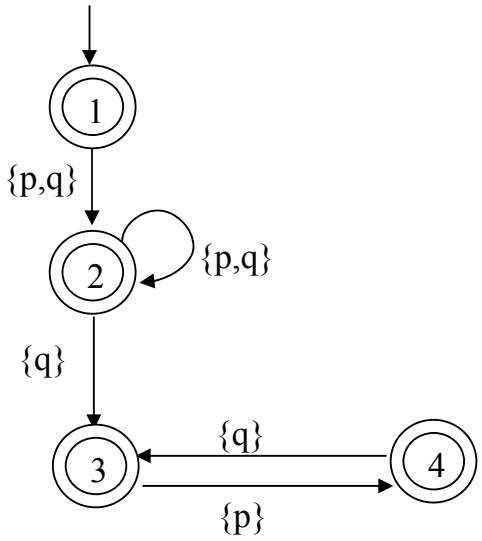
$((s_1, s_2), \sigma, (s_1', s_2')) \in \Delta$ nếu $(s_1, \sigma, s_1') \in \Delta$ và $(s_2, \sigma, s_2') \in \Delta$

Do đó, ta xây dựng được

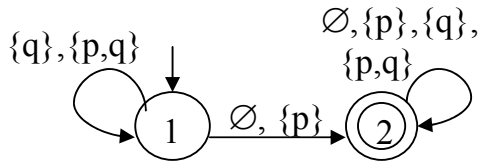
$L(A_1 \times A_2) = L(A_1) \cap L(A_2)$

Ví dụ:

Tính tích chập của 2 ôtômat Buchi sau:



Ôtômat Buchi 1



Ôtômat Buchi 2

$S_0 = S_{01} \times S_{02} = (1,1)$

S = (1,1), (2,1), (3,1), (4,1), (1,2), (2,2), (3,2), (4,2)

Xây dựng các hàm chuyển Δ theo công thức trên, các trạng thái không tới được là (1,2), (2,2), (4,1)

Tìm tập trạng thái kết thúc $F = \{F_1 \times S_2, S_1 \times F_2\}$

Đặt $F_a = F_1 \times S_2 = \{(1,1), (2,1), (3,1), (4,1), (1,2), (2,2), (3,2), (4,2)\}$

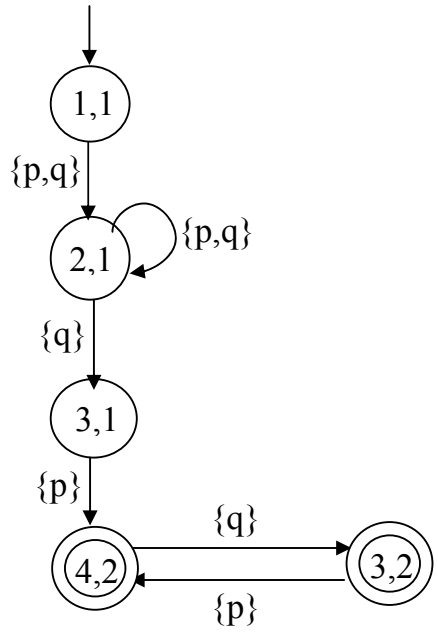
loại các trạng thái không đến được, ta có $F_a = \{(1,1), (2,1), (3,1), (3,2), (4,2)\}$

Đặt $F_b = S_1 \times F_2 = (1,2), (2,2), (3,2), (4,2)$

loại các trạng thái không đến được, ta có $F_b = \{(3,2), (4,2)\}$

Theo định nghĩa đường đi chấp nhận được của Ôtômat Buchi dẫn xuất sẽ phải đi qua một trong số các trạng thái của F_a vô hạn lần và đồng thời đi qua một trong số các trạng thái của F_b vô hạn lần, do đó: $F = \{(3,2), (4,2)\}$

Ta có kết quả như sau:



3.8 KIỂM TRA TÍNH RỖNG CỦA NGÔN NGỮ ĐƯỢC ĐOÁN NHẬN BỞI ÔTÔMAT BUCHI

Cho một Ôtômat Buchi $A = (\Sigma, Q, \Delta, Q_0, F)$, kiểm tra xem ngôn ngữ được đoán nhận bởi A: $L(A) = \varnothing$?

$L(A) \neq \varnothing$ nếu và chỉ nếu tồn tại một đường đi $r = q_0, q_1, q_2, \ldots$sao cho

- $q_0 \in Q_0$,

- $\inf(r) \cap F \neq \varnothing$ và
- Với mọi $i \geq 0$, luôn tồn tại một $a_i \in \Sigma$ sao cho $(q_i,a_i,q_{i+1}) \in \Delta$

Nói cách khác, ngôn ngữ được đoán nhận bởi Ôtômat Buchi là rỗng nếu không tồn tại một đường đi nào chấp nhận được. Một đường đi chấp nhận được nếu và chỉ nếu tồn tại một trạng thái kết thúc (trạng thái chấp nhận được) $q \in F$ sao cho:

- q có thể tới được từ một trạng thái khởi tạo thuộc Q_0 và
- q có thể tự đến được chính nó

Dựa vào tiêu chí đó, đề xuất giải thuật kiểm tra tính rỗng của ngôn ngữ được đoán nhận bởi Ôtômat Buchi như sau:

Bước 1: Tìm kiếm theo chiều sâu lần thứ nhất: Tìm tất cả những đường đi bắt đầu từ trạng thái ban đầu thuộc Q_0 và kết thúc bởi trạng thái nằm trong tập F.

Bước 2: Từ những trạng thái thuộc tập F được đến từ trạng thái khởi tạo thuộc Q_0 đó, tìm xem có tồn tại một đường đi nào đến được chính trạng thái thuộc tập F đó không. Nếu tồn tại chứng tỏ $L(A) \neq \varnothing$. Ngược lại nếu bước 1 hoặc bước 2 không tìm thấy kết quả thì $L(A) = \varnothing$

Giải thuật kiểm tra tính rỗng của ngôn ngữ được đoán nhận bởi Ôtômat Buchi: [5]

Dfs_B(s,d) là thủ tục tìm kiếm theo chiều sâu từ trạng thái s đến trạng thái d. Trong đó, acc(s) = true nếu và chỉ nếu s là trạng thái chấp nhận được. Đầu tiên, quá trình tìm kiếm bắt đầu với dfs_B(s_0, s_0)

```
dfs_B(s, d)
{
     /* Tìm các đường đi từ s0 đến trạng thái kết thúc */
     add s to visited
     push s onto stack
     for each successor s' of s
     {
```

```
            if s' not in visited
            {
                  dfs_B(s', d)
            }
            else if s' = seed and d = s1
            {
                  report acceptance cycle
                  stop
            }
      }
      /* Tìm xem từ những trạng thái kết thúc của các đường đi
      tìm được có đường đến chính nó hay không */
      if d = s0 and acc(s)
      {
            // nhớ nút gốc của lần tìm kiếm thứ hai
            seed = s
            //thực hiện lần tìm kiếm thứ hai bằng cách duyệt
            xem //có đường đi đến nút gốc đó hay không
            dfs_B(s,s1)
      }
      pop s from stack
}
```

Trong trường hợp xấu nhất ta phải duyệt toàn bộ đồ thị để đáp ứng hai lần tìm kiếm theo chiều sâu.

3.9 KẾT LUẬN CHƯƠNG

Kỹ thuật kiểm tra mô hình phần mềm đề xuất dựa trên hai vấn đề khá mới mẻ đó là: lý thuyết Ôtômat Buchi với đặc tính có khả năng đoán nhận xâu vô hạn và lý thuyết Logic thời gian tuyến tính LTL có khả năng biểu diễn về mặt thời gian đối với các thuộc tính của hệ thống. Từ đó, đề cập đến rất nhiều các định nghĩa, khái niệm, giải thuật xung quanh Ôtômat Buchi và Logic thời

gian tuyến tính theo trình tự để mô hình hoá hệ thống, thuộc tính của hệ thống, đồng thời giải quyết triệt để bài toán đặt ra. Dựa vào các giải thuật đề xuất, ta tiếp tục xét đến một bộ kiểm tra mô hình phần mềm cụ thể được cài đặt sử dụng các giải thuật đề xuất để minh hoạ tính tự động và các ưu điểm của kiểm tra mô hình phần mềm.

CHƯƠNG 4: XÂY DỰNG HỆ THỐNG ĐỂ KIỂM TRA MÔ HÌNH PHẦN MỀM

4.1 GIỚI THIỆU VỀ MÔ HÌNH SPIN

SPIN (Simple Promela Interpreter) là một hệ thống xác thực chung, hay một bộ kiểm tra mô hình (Model Checker) để hỗ trợ việc thiết kế và xác thực các hệ thống tương tranh đa luồng, đa tiến trình, đặc biệt là các giao thức truyền dữ liệu. [13] SPIN sử dụng ngôn ngữ PROMELA (Protocol Meta Language), sẽ được đề cập sau, để mô hình hóa phần mềm. SPIN xác thực các mô hình phần mềm bằng cách chứng minh tính đúng đắn của các sự tương tác giữa các tiến trình và cố gắng trừu tượng hoá đến mức tối đa từ việc tính toán tuần tự bên trong. Sự tương tác giữa các tiến trình được đặc tả trong SPIN có thể là theo cơ chế gặp mặt (rendezvous) nghĩa là các tiến trình đến cùng một lúc và có thể trao đổi dữ liệu trực tiếp với nhau, hoặc theo cơ chế vùng đệm (buffer) khi hai tiến trình không gặp nhau tại một thời điểm, tiến trình đến trước sẽ gửi gói dữ liệu vào một vùng đệm để tiến trình khác đến lấy. Tập trung nhiều cho việc điều khiển không đồng bộ trong phần mềm hơn là điều khiển đồng bộ trong phần cứng, SPIN phân biệt với các phần mềm khác trong việc tiếp cận kiểm tra mô hình, cụ thể là kiểm tra mô hình phần mềm.

Là một công cụ áp dụng các phương pháp hình thức, SPIN có những đặc tính cơ bản như sau:

- SPIN là một công cụ trực quan, chương trình sử dụng các ký hiệu để thiết kế đặc tả rất rõ ràng, không thực thi chi tiết.
- SPIN là một công cụ mạnh, các ký hiệu ngắn gọn, súc tích để diễn giải các yêu cầu cần kiểm tra tính đúng đắn.
- Cung cấp một phương pháp luận để thiết lập tính nhất quán logic khi thiết kế các lựa chọn hoặc kết hợp các điều kiện cần thoả mãn.

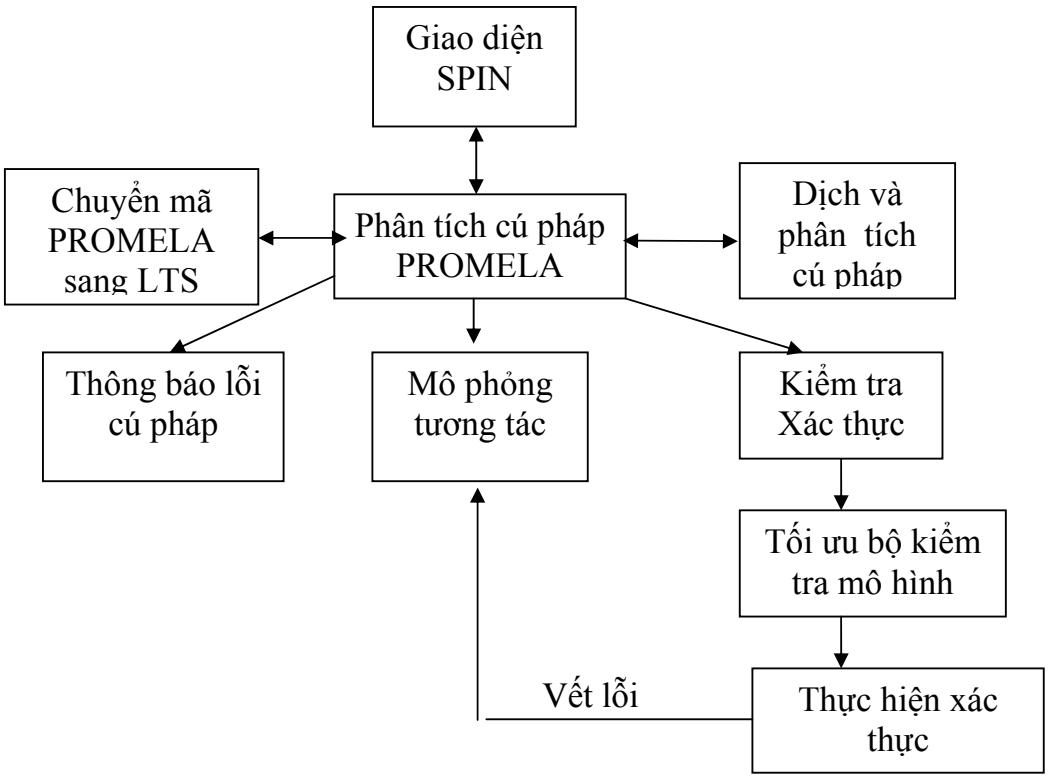
SPIN sử dụng các thiết kế đặc tả phần mềm sử dụng ngôn ngữ mô tả mô hình hệ thống PROMELA và sử dụng Logic thời gian tuyến tính LTL chuẩn để đặc tả các thuộc tính cần thoả mãn.

Không có các nguyên tắc chung cho các hệ thống phần mềm khác nhau, do đó, ta luôn cố gắng đặc tả phần mềm theo một chuẩn nào đó. SPIN sử dụng ngôn ngữ PROMELA, vì thế luôn yêu cầu theo một số các quy tắc hữu hạn nào đó để diễn tả rất nhiều các hành vi khác nhau. Điều đó có nghĩa là tất cả các thuộc tính cần thoả mãn của hệ thống đều trở lên đều phải có tính hình thức với các ràng buộc về phạm vi vấn đề, nguồn tài nguyên cần thiết để tính toán để kiểm tra mô hình có thể chứng minh được thuộc tính đó.

Tất cả các hệ thống xác thực, đều có các ràng buộc vật lý như tập hợp kích thước bài toán, dung lượng bộ nhớ và thời gian chạy tối đa mà hệ thống yêu cầu. Do đó, ta luôn tìm cách đưa ra các chiến lược để giải quyết vấn đề này.

4.2 CẤU TRÚC SPIN

Cấu trúc cơ bản của bộ kiểm tra mô hình Spin được minh hoạ ở hình 4.1. Mô hình SPIN được bắt đầu bằng việc đặc tả hệ thống tương tranh đa luồng hoặc các giải thuật phân tán dưới dạng mô hình bậc cao, sử dụng giao diện đồ hoạ XSPIN. Sau khi sửa các lỗi cú pháp, SPIN sẽ thực hiện việc mô phỏng tương tác cho đến khi đạt được độ tin cậy cơ bản để thiết kế các hành vi. Bước thứ 3, SPIN sinh ra một chương trình xác thực theo giải thuật on-the-fly từ đặc tả mức cao. Bộ xác thực này được biên dịch, với sự lựa chọn thời gian biên dịch hợp lý để thực hiện giải thuật giản lược (reduction algorithm). Nếu có lỗi được phát hiện qua các biến xác nhận lỗi của chương trình (counterexamples), nó sẽ thông báo lại cho bộ mô phỏng và kiểm tra chi tiết khi thực hiện và loại bỏ các nguyên nhân gây ra lỗi.



Hình 4.1 Cấu trúc của bộ mô hình kiểm tra SPIN

SPIN là công cụ hỗ trợ việc nhúng ngôn ngữ C để cùng mô hình hoá hệ thống. SPIN có thể xác thực trực tiếp đặc tả phần mềm ở mức thực thi, sử dụng SPIN như là một trình điều khiển và một máy logic để xác thực các thuộc tính thời gian ở mức cao. SPIN hoạt động dựa vào phương thức duyệt nhanh (on the fly), có nghĩa là tránh được việc xây dựng lại đồ thị trạng thái toàn cục như là một điều kiện tiên quyết để thực hiện việc xác thực các thuộc tính của hệ thống.

SPIN được coi như một hệ thống kiểm tra mô hình LTL đầy đủ, hỗ trợ tất cả các yêu cầu cần kiểm tra tính đúng đắn dưới dạng logic thời gian tuyến tính LTL đồng thời là một bộ xác thực hiệu quả với các thuộc tính cần đảm bảo tính an toàn và tính hoạt động.

Thuộc tính mà hệ thống cần phải thoả mãn có thể được đặc tả như một hệ thống hoặc một tiến trình bất biến, được biểu diễn như các yêu cầu logic

thời gian tuyến tính LTL, hoặc là Ôtômat buchi hình thức, hoặc rộng hơn có thể là các thuộc tính thông thường chưa được đề cập.

SPIN là một công cụ hỗ trợ động sự gia tăng số lượng các tiến trình hoặc sự rút gọn số lượng các tiến trình sử dụng kỹ thuật vector trạng thái động. SPIN hỗ trợ các quá trình mô phỏng dựa trên các việc chứng minh cục bộ và tổng thể, dựa trên tìm kiếm theo chiều sâu để có thể kiểm soát được kích cỡ bài toán lớn có hiệu quả. Để tối ưu hoá trong quá trình xác thực, SPIN đã khai thác kỹ thuật giản lược thứ tự từng phần và lưu trữ theo kiểu BDD. Để xác thực một mô hình phần mềm, mô hình đó phải là một mô hình hình thức được xây dựng bằng ngôn ngữ PROMELA, là một ngôn ngữ đầu vào của hệ thống SPIN.

SPIN có thể được sử dụng như 1 trong 3 cách cơ bản sau:

- Là một bộ mô phỏng, cho phép các mẫu thử nhanh ngẫu nhiên, theo chỉ dẫn hoặc mô phỏng tương tác.
- Là một bộ xác thực tổng thể, có khả năng chứng minh một cách chặt chẽ tính đúng đắn so với yêu cầu đặc tả của người sử dụng (dùng lý thuyết giản lược thứ tự từng phần để tối ưu hoá khi tìm kiếm).
- Là một hệ thống chứng minh xấp xỉ có thể chứng minh các mô hình hệ thống lớn với việc bao phủ lớn nhất có thể không gian trạng thái.

Các bước thực hiện khi kiểm tra mô hình với SPIN được diễn ra như sau:

Bước 1: Sử dụng ngôn ngữ PROMELA để trừu tượng hoá mô hình hệ thống

Bước 2: Sử dụng biểu thức LTL để biểu diễn thuộc tính mà hệ thống cần thoả mãn.

Bước 3: Chạy giải thuật kiểm tra mô hình SPIN, SPIN sẽ thực hiện một cách tự động các công việc sau:

- Chuyển ngôn ngữ PROMELA mô hình hoá hệ thống về dạng LTS
- Dịch và phân tích cú pháp LTL
- Tự động chuyển LTS và LTL sang Ôtômat Buchi. Sau đó, áp dụng các thuật toán với Ôtômat Buchi như tích chập hai Ôtômat Buchi, kiểm tra tính rỗng của Ôtômat Buchi… để kiểm tra xem hệ thống chuyển trạng thái LTS có thoả mãn thuộc tính LTL được mô tả hay không.

Bước 4: Kết quả sẽ thuộc một trong 3 dạng sau:

- Mô hình thoả mãn thuộc tính
- Mô hình vi phạm thuộc tính, đưa ra các counterexample để ghi nhận lỗi
- Không đủ tài nguyên để giải quyết bài toán, khi đó phải xây dựng lại mô hình trừ u tượng hơn nữa

Bước 5: Duyệt lại bước 1 hoặc 2 và lặp lại các bước từ 3 đến 5 cho đến khi mô hình thoả mãn thuộc tính.

4.3 NGÔN NGỮ PROMELA

4.3.1 Giới thiệu chung về Promela

Promela (PROTOCOL/ PROCESS META LANGUAGE) là một ngôn ngữ dùng để mô hình hoá phần mềm. Promela cung cấp một phương tiện để trừu tượng hoá các giao thức đặc biệt trong các hệ thống phân tán và loại bỏ những chi tiết không tương tác với các tiến trình. Quá trình xác thực đầy đủ bao gồm việc thực hiện một chuỗi các bước bằng cách xây dựng mô hình bằng Promela để tăng dần độ chi tiết. Mỗi mô hình có thể xác thực với SPIN dưới nhiều dạng khác nhau của môi trường giả định (như mất các gói tin, gói tin bị lặp…). [11]

Các chương trình Promela chứa các tiến trình (processes), các kênh thông tin (message channels) và các biến (variables). Các tiến trình chính là những đối tượng toàn cục, các kênh thông tin và các biến có thể được khai báo toàn cục hoặc cục bộ trong một tiến trình. Các tiến trình đặc tả hành vi, các kênh và các biến tổng thể định nghĩa môi trường mà các tiến trình chạy trên đó.

4.3.2 Mô hình một chương trình Promela

```
mtype = {MSG, ACK}; /* Khai báo kiểu */
chan  toS =…
chan  toR =…  /* Khai báo kênh */
bool  flag;        /* Khai báo biến  */
proctype  Sender()
{
     …
}    /*Khai báo tiến trình */
init
{
…
} /* Tiến trình khởi tạo (tuỳ chọn) */
```

4.3.4 Khai báo tiến trình (Process declaration)

Một tiến trình bao gồm tên, danh sách các tham số, khai báo các biến cục bộ và thân tiến trình. [

Ví dụ:

```
proctype A(byte state)
     {    byte tmp;
     (state==1) -> tmp = state; tmp = tmp+1; state = tmp
     }
```

Thân của tiến trình chứa một dãy các câu lệnh . Để ngăn cách giữa các câu lệnh, Promela sử dụng dấu ; hoặc dấu - >. Hai dấu hiệu phân cách này là tương đương trong đó trong một số trường hợp dấu - > để biểu thị mối quan hệ nếu…thì giữa hai câu lệnh. Ở đây vì là dấu hiệu ngăn cách mà không phải là dấu hiệu kết thúc câu nên ở câu lệnh cuối cùng sẽ không có dấu ;.

Một tiến trình được thực hiện đồng thời với tất cả các tiến trình khác, với các hành vi độc lập. Các tiến trình liên kết với nhau qua việc sử dụng chung các biến toàn cục hoặc các kênh. Mỗi tiến trình đều có các trạng thái cục bộ riêng như biến trong tiến trình hoặc nội dung của các biến cục bộ. Các tiến trình có thể được khởi tạo bằng cách thêm từ khoá active đằng trước proctype.

4.3.5 Tiến trình khởi tạo

Định nghĩa proctype chỉ là khai báo tiến trình, không phải là thực thi tiến trình đó. Trong ngôn ngữ mô hình hoá hệ thống Promela, chỉ có một tiến trình được thực thi được gọi là tiến trình khởi tạo: init và bắt buộc phải khai báo cụ thể trong mọi đặc tả Promela.

Ví dụ:

```
init
     {
          run A(); run B()/* Hai tiến trình A, B đã được khai
báo */
     }
```

Câu lệnh Run có thể được sử dụng ở bất cứ tiến trình nào, không chỉ riêng ở tiến trình khởi tạo. Các tiến tình được tạo ra sau khi gặp câu lệnh run. Các tiến trình có thể thực hiện xen kẽ nhau, không nhất thiết kết thúc tiến trình này mới đến tiến trình tiếp theo. Với câu lệnh Run, ta có thể tạo được một số các tiến trình copy của nó.

4.3.6 Khai báo biến và kiểu

Các kiểu dữ liệu cơ bản:

```
bit  turn = 1;      [0..1]
bool flag;          [0..1]
byte counter;       [0..255]
short     s;        [-2^16-1..2^16-1]
int  msg;      [-2^32-1..2^32-1]
```

Các kiểu dữ liệu cơ bản có giá trị khởi tạo ngầm định bằng 0

Dữ liệu kiểu mảng được bắt đầu đánh chỉ số từ 0

```
byte a[27];
bit    flag[4];
```

Khai báo kiểu bản ghi

```
typedef   Record
{
     short f1;
     byte f2;
}
```

Các biến có thể là các biến toàn cục hoặc các biến cục bộ tuỳ thuộc vào vị trí khai báo. Nếu các biến được khai báo ngoài tất cả các tiến trình thì đó là biến toàn cục, và nếu biến được khai báo trong tiến trình sẽ là biến cục bộ.

Gán giá trị cho biến có thể dùng một trong 3 cách sau:

- Sử dụng câu lệnh gán

  ```
  bb = 1;
  ```

- Sử dụng khai báo kết hợp với khởi tạo

  ```
  short     s = -1;
  ```

- Sử dụng câu lệnh kiểm tra điều kiện bằng

  ```
  i * s + 27 == 23;
  ```

4.3.7 Câu lệnh trong Promela

Trong Promela không có sự phân biệt giữa lệnh điều kiện và câu lệnh, thậm chí cả các điều kiện boolean cũng có thể được sử dụng như các câu lệnh. Câu lệnh có thể là câu lệnh thi hành (executable) hoặc lệnh bị tạm thời trì

hoãn (blocked) tuỳ thuộc vào trạng thái tổng thể của hệ thống. Khi điều kiện được thoả mãn, thì câu lệnh điều kiện chính là lệnh thi hành. Lệnh thi hành là các lệnh có thể được thi hành ngay lập tức, lệnh blocked là lệnh chưa được thi hành ngay. Câu lệnh gán luôn là câu lệnh thi hành.

Một biểu thức logic có thể là một câu lệnh thực thi nếu nó có giá trị khác 0

2 < 3 : luôn là lệnh thi hành

x < 27 chỉ là lệnh thi hành nếu giá trị của x < 27

3 + x chỉ là lệnh thi hành nếu x khác –3

Một số lệnh thường dùng:

- Lệnh *skip*: không làm gì cả (does nothing), chỉ dùng để thay đổi biến đếm tiến trình của tiến trình), luôn là lệnh thi hành.
- Lệnh *run*: chỉ là lệnh thi hành nếu một tiến trình mới được tạo ra, chú ý số tiến trình được tạo ra là có giới hạn.
- Lệnh *printf*: luôn là lệnh thi hành, nhưng không dùng để đánh giá trong suốt quá trình xác thực.

```
int x;
proctype Aap()
{
int y=1;
skip;
run Noot(); /*Là lệnh thi hành nếu Noot được tạo */
x=2;
x>2 && y==1;
/* Chỉ thi hành nếu một số tiến trình khác làm x>2 */
skip;
}
```

- Lệnh assert (<expr>): luôn luôn là lệnh thi hành. Nếu <expr> có giá trị sai, SPIN sẽ thoát và ghi nhận một lỗi là <expr> đã bị vi phạm

(violated). Câu lệnh assert thường được sử dụng trong các mô hình Promela để kiểm tra xem các điều kiện có được thoả mãn hay không

4.3.8 Cấu trúc atomic

Khi thực hiện, các tiến trình có thể xen kẽ nhau (các câu lệnh của các tiến trình khác nhau có thể không xuất hiện cùng một thời điểm). Nếu sử dụng cấu trúc:

```
atomic {
/*dãy các lệnh */
}
```

mọi lệnh thuộc cấu trúc này sẽ được thực hiện liên tiếp mà không bị xen vào bởi các tiến trình khác. Dãy các câu lệnh atomic là một công cụ rất quan trọng để giảm độ phức tạp khi xác thực mô hình.

4.3.9 Các cấu trúc điều khiển thường gặp

4.3.9.1 Câu lệnh điều kiện IF

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

Nếu luôn có ít nhất một *choice*$_i$ (điều kiện) được thoả mãn, câu lệnh IF sẽ được thực thi và SPIN không đơn định sẽ lựa chọn một trong những số nhánh đó để thi hành. Nếu không tồn tại một choice$_i$ nào được thoả mãn, câu lệnh if sẽ là lệnh blocked.

Ví dụ:

```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0) -> n=n-2
:: (n % 3 == 0) -> n=3
:: else -> skip
fi
```

Câu lệnh else sẽ làm cho câu lệnh if luôn là lệnh thi hành, vì nếu không có điều kiện nào phía trên thoả mãn, chương trình sẽ thực thi công việc sau else.

4.3.9.2 Câu lệnh lặp DO

```
do
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
od;
```

Về cấu trúc tổng quát, câu lệnh do cũng giống như câu lệnh if. Tuy nhiên, khác với lệnh if, câu lệnh do sẽ lặp các lựa chọn cho đến khi gặp câu lệnh break để thoát khỏi vòng lặp.

Ví dụ:

```
do
:: (count != 0) -> :: count = count – 1;
:: (count == 0) -> break
od;
```

Bài toán đèn giao thông, câu lệnh do luôn luôn lựa chọn được một điều kiện tại một thời điểm, tức luôn đơn định:

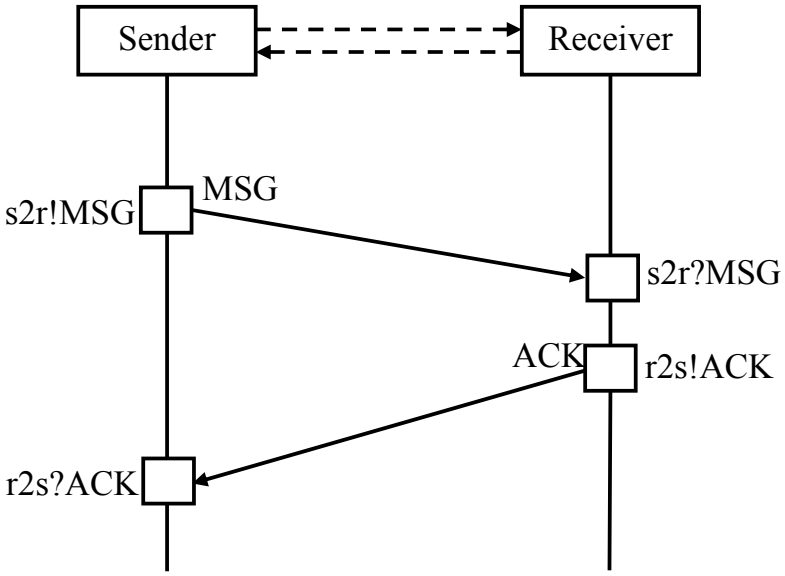
```
mtype = { RED, YELLOW, GREEN } ;
active proctype TrafficLight() {
byte state = GREEN;
do
```

```
      :: (state == GREEN) -> state = YELLOW;
      :: (state == YELLOW) -> state = RED;
      :: (state == RED) -> state = GREEN;
od;
}
```

4.3.10 Giao tiếp giữa các tiến trình

4.3.10.1 Mô hình chung

Mô hình chung của quá trình giao tiếp giữa bên gửi và bên nhận được minh hoạ như sau: Bên gửi (Sender) sẽ gửi thông điệp (MSG) cho bên nhận (Receiver). Sau khi nhận được thông điệp, bên nhận sẽ gửi lại một tin xác thực để bên gửi biết thông điệp có được gửi đến và chính xác hay không.



Trong đó:

Sender: bên gửi

Receiver: bên nhận

MSG: thông điệp

ACK: xác thực

!: quá trình gửi

?: quá trình nhận

s2r: bên gửi đến bên nhận

r2s: bên nhận đến bên gửi

Hình 4.2 Mô hình giao tiếp giữa hai tiến trình

Sự giao tiếp, truyền thông giữa các tiến trình qua các kênh (channel) được thực hiện bằng một trong hai cách sau: truyền thông điệp (message passing), tức là một tiến trình truyền dữ liệu đến một tiến trình khác, hoặc theo cơ chế bắt tay tức cả hai tiến trình cùng thực hiện quá trình gửi và nhận một cách đồng bộ (render vous). Với cả hai cách này, đều được khai báo channel theo cách sau:

chan <name> = [<dim>] of {$<t_1>$,$<t_2>$,…,$<t_n>$}

Trong đó:

name: tên của channel, channel được hiểu như một FIFO buffer

dim: số lượng các phần tử trong channel, trong trường hợp render vous thì dim = 0

$<t_1>$,$<t_2>$,…,$<t_n>$: kiểu của các phần tử được truyền trên kênh.

Ví dụ:

```
chan  c = [1] of {bit}
chan  toR = [2] of {mtype, bit}
chan line[2]  = [1] of {mtype, record}
```

Gửi (Sending) dữ liệu đến một kênh: sử dụng ký hiệu !

ch ! $<expr_1>$, $<expr_2>$,…,$<expr_n>$;

Giá trị của biểu thức $<expr_i>$ phải tương ứng với kiểu dữ liệu của kênh được khai báo

Câu lệnh gửi dữ liệu là câu lệnh thi hành nếu kênh không bị tràn.

Nhận (Receiving) dữ liệu từ kênh: sử dụng ký hiệu ?

ch ? <var1>, <var2>,…,<varn>;

ch ? <const1>, <const2>,…,<constn>;

Nếu kênh không rỗng, dữ liệu có thể được gửi từ kênh thông qua các phần độc lập của message được lưu trữ trong danh sách các biến hoặc các hằng, các biến và các hằng số có thể trộn lẫn với nhau trong danh sách.

```
proctype A(chan q1)
     {    chan q2;
          q1?q2;
          q2!123
     }
     proctype B(chan qforb)
     {    int x;
          qforb?x;
          printf("x = %d\n", x)
     }
     init
     {
          chan qname = [1] of { chan };
          chan qforb = [1] of { int };
          run A(qname);
          run B(qforb);
          qname!qforb
     }
```

Giá trị được in là 123

4.3.10.2 Giao tiếp giữa các tiến trình kiểu bắt tay

Khi số lượng phần tử trong channel = 0, các tiến trình sẽ giao tiếp với nhau theo kiểu bắt tay (render vous). <dim> ==0 có nghĩa là cổng channel có thể qua, nhưng không thể lưu trữ thông điệp. Nếu tiến trình gửi ch! được thực hiện và nếu có một tiến trình nhận ch? tương ứng được thực thi đồng bộ và kết hợp với nhau, vì thế cả 2 quá trình đều thực hiện. Hai quá trình đó sẽ bắt tay (hand-shake) hay gặp nhau (render vous) và cũng trao đổi dữ liệu.

Ví dụ:

```
chan ch = [0] of {bit, byte};
```

P muốn thực hiện ch ! 1, 3+7

Q muốn thực hiện ch ? x

Sau khi giao tiếp, x sẽ nhận giá trị 10

4.4 CÚ PHÁP CỦA LTL TRONG SPIN

Để diễn tả các toán tử logic thời gian tuyến tính LTL, SPIN quy ước sử dụng như sau: [9]

[]	Toán tử always □
<>	Toán tử tồn tại ◊
!	Toán tử phủ định
U	Toán tử cho đến khi U
&&	Toán tử AND
||	Toán tử OR
->	Toán tử suy ra
<->	Toán tử tương đương

Dựa vào những ký hiệu đó, SPIN có thể diễn tả các thuộc tính mà hệ thống cần thoả mãn.

4.5 MINH HOẠ KIỂM TRA MÔ HÌNH PHẦN MỀM VỚI SPIN

Bài toán quản lý tài nguyên găng: Tại một thời điểm chỉ có một tiến trình được sử dụng tài nguyên găng. Khi một tiến trình đang sử dụng tài nguyên găng, tài nguyên đó sẽ bị khoá, sau khi tiến trình đó sử dụng xong, tài nguyên mới được giải phóng để tiến trình khác sử dụng. Như vậy, cần kiểm tra xem tại một thời điểm, không được hai tiến trình cùng sử dụng một tài nguyên găng.

Do đó, thiết kế mô hình của bài toán như sau:

```
bit flag;
byte mutex;
proctype P(bit i)
{
```

```
     atomic{
          flag != 1;
          flag = 1;
     }
     mutex++;
     printf("SMC:P(%d)has enter section, mutex=
%d\n",i,mutex);
     mutex--;
     flag = 0;
}
proctype monitor()
{
     do
     ::printf("mutex value is %d\n",mutex); assert(mutex!=2);
     od
}
init
{
     atomic{
          run P(0);
          run P(1);
          run monitor();
     }
}
```

Mô hình được mô tả bằng ngôn ngữ Promela như sau:

Sử dụng biến mutex để đếm số tiến trình sử dụng tài nguyên găng, khởi tạo mutex = 0

Sử dụng biến flag để đánh dấu tài nguyên găng đang được sử dụng. Nếu flag = 1, tài nguyên găng đang được sử dụng và các tiến trình khác không vào được. Nếu flag = 0, tài nguyên găng được giải phóng, tiến trình có thể sử dụng được tài nguyên đó.

Đầu tiên, chương trình kiểm tra xem biến cờ có khác 1 (hay bằng 0) hay không. Nếu không đúng tức flag =1, tiến trình sẽ lặp đi lặp lại kiểm tra để chờ. Nếu đúng, tức flag =0, tài nguyên găng đang sẵn sàng, khi đó tiến trình có thể sử dụng tài nguyên găng. Sau đó, đặt flag = 1 để báo tài nguyên găng đang bận. Hai công việc này phải thực hiện liên tiếp, do đó đặt trong từ khoá *atomic*. Khi sử dụng tài nguyên găng, biến mutex được tăng lên 1 và thực hiện công việc (giả sử là công việc in ra màn hình). Sau khi sử dụng xong tài nguyên găng, biến mutex được giảm đi 1 và gắn lại biến flag = 0.

Tiến trình *monitor* sẽ thực hiện lệnh assert trong vòng lặp do để liên tục kiểm tra xem mutex có khác 2 hay không, tức là không thể hai tiến trình cùng sử dụng chung tài nguyên găng. Nếu vi phạm tức mô hình đã không thoả mãn thuộc tính đề ra của hệ thống. Tiến trình init kích hoạt các tiến trình P(0), P(1) và monitor().

Tạo ra file pan.c, sau đó hiển thị bảng trạng thái như sau:

```
proctype P
state 3 -> state 4  [A---G] line 6 => ((flag!=1))
state 4 -> state 5  [----G] line  11 => mutex = (mutex+1)
state 5 -> state 6  [----G] line 12 => printf('SMC: P(%d) has
enter section, mutex  = %d\n',i,mutex)
state 6 -> state 7  [----G] line 13 => mutex = (mutex-1)
state 7 -> state 8  [----G] line 14 => flag = 0
state 8 -> state 0  [--e-L] line 15 => -end-   [(257,9)]
proctype monitor
state 3 -> state 2  [----G] line 19 => printf('mutex value is
%d\n',mutex)
state 2 -> state 3  [----G] line 20 => assert((mutex!=2))
proctype init
state 4 -> state 2  [A---L] line 26 => (run P(0))
state 2 -> state 3  [A---L] line 28 => (run P(1))
state 3 -> state 5  [----L] line 29 => (run monitor())
```

```
state 5 -> state 0  [--e-L] line 32 => -end-      [(257,9)]
Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
```

Ta có thể kiểm tra mô hình bằng việc chạy một số bước hữu hạn, giả sử ta muốn chạy mô hình với số bước chạy không quá 20 lần, chương trình sẽ chỉ ra từng bước, tiến trình nào được kích hoạt và chạy câu lệnh nào, giá trị trả về bằng bao nhiêu, có vi phạm điều kiện hay không.

```
Starting :init: with pid 0
0: proc  - (:root:) creates proc  0 (:init:)
Starting P with pid 1
 1: proc 0 (:init:) creates proc  1 (P)
 1: proc 0(:init:)line 26  (state 4)[(run P(0))]
Starting P with pid 2
 2: proc 0 (:init:) creates proc  2 (P)
 2: proc 0(:init:)line 28 (state 2)[(run P(1))]
Starting monitor with pid 3
 3: proc 0 (:init:) creates proc  3 (monitor)
 3: proc 0(:init:) line 29  (state 3)
[(run monitor())]
 4: proc 1 (P) line 6   (state 3)[((flag!=1))]
 5: proc 1 (P) line 8   (state 2)[flag = 1]
 mutex value is 0
 6: proc 3 (monitor) line  19   (state 3)
 [printf('mutex value is %d\\n',mutex)]
 7: proc 3 (monitor) line  20   (state 2)
[assert((mutex!=2))]
 8: proc  3 (monitor) line  22   (state 4) [.(goto)]
 mutex value is 0
 9: proc  3 (monitor) line  19   (state 3)
```

```
 [printf('mutex value is %d\\n',mutex)]
 10: proc  1 (P) line  11   (state 4) [mutex = (mutex+1)]
 SMC: P(0) has enter section, mutex  = 1
 11: proc  1 (P) line  12   (state 5)
 [printf('SMC: P(%d) has enter section, mutex  =
%d\\n',i,mutex)]
 12: proc  3 (monitor) line 20 (state 2)[assert((mutex!=2))]
 13: proc  1 (P) line  13   (state 6)   [mutex = (mutex-1)]
 14: proc  1 (P) line  14   (state 7)   [flag = 0]
 15: proc  2 (P) line   6   (state 3)   [((flag!=1))]
 16: proc  2 (P) line   8   (state 2)   [flag = 1]
 17: proc  3 (monitor) line  22 (state 4)  [.(goto)]
 mutex value is 0
 18: proc  3 (monitor) line  19   (state 3)
 [printf('mutex value is %d\\n',mutex)]
 19: proc  3 (monitor) line 20 (state 2) [assert((mutex!=2))]
 20: proc  3 (monitor) line  22 (state 4)  [.(goto)]
 depth-limit (-u20 steps) reached
 #processes: 4
 flag = 1
 mutex = 0
 20: proc  3 (monitor) line  19   (state 3)
 20: proc  2 (P) line  11   (state 4)
 20: proc  1 (P) line  15   (state 8) <valid end state>
 20: proc  0 (:init:) line  32   (state 5) <valid end state>
4 processes created
```

Vậy, thông qua SPIN để kiểm tra mô hình của bài toán, ta có thể khẳng định mô hình đó thoả mãn thuộc tính đề ra đó là luôn đảm bảo tại một thời điểm chỉ có nhiều nhất một tiến trình được sử dụng tài nguyên găng.

Vẫn với bài toán quản lý tài nguyên găng, giả sử ta thiết kê mô hình của hệ thống như sau:

```
bit flag;
byte mutex;
proctype P(bit i)
{
     flag != 1;
     flag = 1;
     mutex++;
     printf("SMC: P(%d) has enter section, mutex  =
%d\n",i,mutex);
     flag = 0;
     mutex--;

}

proctype monitor()
{
     do
     ::assert(mutex!=2);
     od
}
init
{
     atomic{
          run P(0);
          run P(1);
          run monitor();
     }
}
```

Ở mô hình này chỉ khác ở mô hình trước bằng cách thay đổi thứ tự của hai câu lệnh `flag = 0;` và `mutex--;` đồng thời bỏ đi từ khoá atomic của hai lệnh `flag != 1;flag = 1;`

Tuy nhiên, khi thay đổi vị trí câu lệnh, sẽ làm cho mô hình của hệ thống vi phạm ngay thuộc tính đề ra.

Ta có bảng trạng thái của mô hình này như sau:

```
proctype P
state   1 -> state   2   [----G] line 6 => ((flag!=1))
state   2 -> state   3   [----G] line 7 => flag = 1
state   3 -> state   4   [----G] line 8 => mutex = (mutex+1)
state   4 -> state  5    [----G] line 9 => printf('SMC: P(%d)
has enter section, mutex  = %d\n',i,mutex)
state   5 -> state   6   [----G] line 10 => flag = 0
state   6 -> state   7   [----G] line 11 => mutex = (mutex-1)
state   7 -> state   0   [--e-L] line 13 => -end-[(257,9)]
proctype monitor
state   2 -> state   2   [----G] line 17 =>assert((mutex!=2))
proctype init
state   4 -> state   2   [A---L] line 24 => (run P(0))
state   2 -> state   3   [A---L] line 26 => (run P(1))
state   3 -> state   5  [----L] line 27 => (run monitor())
state   5 -> state   0   [--e-L] line 30 => -end-[(257,9)]
Transition Type: A=atomic; D=d_step; L=local; G=global
Source-State Labels: p=progress; e=end; a=accept;
```

Kiểm tra mô hình với số bước chạy không vượt quá 1000, được kết quả sau:

```
Starting :init: with pid 0
0:  proc  - (:root:) creates proc  0 (:init:)
//Khởi tạo tiến trình init
Starting P with pid 1
1:  proc  0 (:init:) creates proc  1 (P)
```

```
1:  proc  0 (:init:) line  24 (state 4)  [(run P(0))]
//Init: Chạy tiến trình P(0)
Starting P with pid 2
2:  proc  0 (:init:) creates proc  2 (P)
2:  proc  0 (:init:) line  26 (state 2)  [(run P(1))]
//Init: Chạy tiến trình P(1)
Starting monitor with pid 3
3:  proc  0 (:init:) creates proc  3 (monitor)
3:  proc  0 (:init:) line  27 (state 3)  [(run monitor())]
//Init: Chạy tiến trình monitor()
4:  proc  3 (monitor) line  17  (state 2) assert((mutex!=2))]
//Monitor: Kiểm tra xem mutex có khác 2 hay không?
5:  proc  2 (P) line 6  (state 1)   [((flag!=1))]
//P(1) Kiểm tra xem flag có khác 1 hay không?
6:  proc  3 (monitor) line  20  (state 3) [.(goto)]
7:  proc  3 (monitor) line  17  (state 2) assert((mutex!=2))]
//Monitor: Kiểm tra mutex có khác 2 hay không?
8:  proc  1 (P) line 6  (state 1)   [((flag!=1))]
//P(0): Kiểm tra flag có khác 1 hay không?
9:  proc  1 (P) line 7  (state 2)  [flag = 1]
//P(0): Nếu đúng, gán lại flag = 1 để tiến trình P(1) sử dụng
tài nguyên găng
10:  proc  1 (P) line 8  (state 3) [mutex = (mutex+1)]
//P(0): Tăng biến mutex lên 1
11:  proc  3 (monitor) line  20  (state 3) [.(goto)]
//Monitor: Khởi tạo tiến trình monitor
12:  proc 3(monitor) line  17  (state 2) [assert((mutex!=2))]
    SMC: P(0) has enter section, mutex  = 1
//Monitor: Kiểm tra xem mutex có khác 2 hay không?
13:  proc  1 (P) line 9  (state 4)
[printf('SMC: P(%d) has enter section, mutex  =
%d\\n',i,mutex)]
```

```
//P(0): Sử dụng tài nguyên bằng lệnh printf
14:  proc  2 (P) line 7  (state 2)   [flag = 1]
//P(1): Flag =1
15:  proc  1 (P) line  10  (state 5)  [flag = 0]
//P(0): Flag = 0
16:  proc  2 (P) line 8  (state 3) [mutex = (mutex+1)]
//P(1) : tăng mutex lên 1
17:  proc  3 (monitor) line  20  (state 3) [.(goto)]
      SMC: P(1) has enter section, mutex  = 2
//Monitor:  mutex = 2
 18:  proc  2 (P) line 9  (state 4)
[printf('SMC: P(%d) has enter section, mutex  =
%d\\n',i,mutex)]
spin: line  18 , Error: assertion violated
//Thực hiện lệnh, kiểm tra thấy mutex != 2 bị vi phạm
spin: text of failed assertion: assert((mutex!=2))
#processes: 4
      flag = 0
      mutex = 2
```

Khi bị vi phạm điều kiện mutex != 2, chương trình sẽ dừng lại và thông báo lỗi "Assertion violated"

Mô hình SPIN đã tự động tạo ra hệ thống chuyển trạng thái cho hệ thống thông qua ngôn ngữ PROMELA, sau đó tự động chuyển đổi hệ thống và các thuộc tính LTL sang dạng Ôtômat Buchi và áp dụng các giải thuật kiểm tra mô hình phần mềm với tất cả các khả năng có thể của hệ thống. Khi gặp lỗi SPIN sẽ dừng lại và đưa ra ngay vết lỗi.

KẾT LUẬN

Nội dung của đồ án được trình bày trong bốn chương về vấn đề kiểm tra mô hình phần mềm, một trong những lĩnh vực rộng và đang còn tiếp tục phát triển mạnh.

Tóm tắt luận văn:

- Giải thích tại sao cần phải nghiên cứu về kiểm tra mô hình phần mềm
- Nghiên cứu những khái niệm về kiểm tra mô hình, các định nghĩa đa dạng về kiểm tra mô hình phần mềm, vị trí của kỹ thuật kiểm tra mô hình phần mềm so với các công nghệ khác như Testing. Tiếp theo đã phân biệt được các kỹ thuật kiểm tra mô hình phần mềm và giới hạn đi sâu vào nghiên cứu một loại kỹ thuật khả thi nhất đó là theo kiểu duyệt nhanh
- Chương 3 đã nghiên cứu và đề xuất một giải pháp chung để kiểm tra mô hình phần mềm. Từ bản phác hoạ này ta nghiên cứu các vấn đề cần giải quyết khi xây dựng hệ thống kiểm tra mô hình phần mềm. Luận văn đã đề xuất giải thuật kiểm tra mô hình phần mềm sử dụng Ôtômat Buchi có thể đoán nhận được chuỗi vô hạn và Logic thời gian tuyến tính để mô hình hoá hệ thống cũng như các thuộc tính mà hệ thống cần phải thoả mãn. Luận văn đề cập đến các vấn đề lý thuyết hình thức xung quanh việc mô hình hoá hệ thống, thuộc tính và vấn đề mấu chốt của bài toán là kiểm tra xem hệ thống có thoả mãn thuộc tính hay không. Ví dụ, ngữ nghĩa, cú pháp của ngôn ngữ logic LTL, Ôtômat Buchi, hệ thống chuyển trạng thái LTS, các phép chuyển đổi LTL và LTS sang Ôtômat Buchi, các phép tích chập của hai Ôtômat Buchi, kiểm tra tính rỗng của Ôtômat Buchi v.v.

- Chương 4 cụ thể hoá việc xây dựng hệ thống kiểm tra mô hình phần mềm bằng cách giới thiệu một ngôn ngữ chuyên dụng để đặc tả mô hình hệ thống đó là ngôn ngữ PROMELA và hệ thống kiểm tra mô hình SPIN áp dụng cho những hệ thống tương tranh đa tiến trình. Ngôn ngữ PROMELA là một ngôn ngữ đơn giản và mạnh có thể diễn giải được tất cả các tình huống của mô hình phần mềm và của thuộc tính cần thoả mãn. SPIN đã giải quyết được bài toán đặt ra bằng cách sử dụng giải thuật đề xuất.
- Cuối cùng, luận văn đã sử dụng một ví dụ minh hoạ nhỏ về việc áp dụng hệ thống SPIN, ngôn ngữ PROMELA để làm rõ hơn các bước của giải thuật và tính hiệu quả của việc kiểm tra mô hình phần mềm.

Đóng góp khoa học của luận văn:

- Luận văn đã giải quyết bài toán kiểm tra mô hình phần mềm đó là: cho mô hình của một hệ thống M và thuộc tính f, cần kiểm tra xem M có thoả mãn f hay không? Luận văn đã đề cập đến các hướng lý thuyết mới mẻ và ngày càng được quan tâm rộng rãi như thuyết Ôtômat Buchi để đoán nhận xâu vô hạn và logic thời gian tuyến tính để biểu diễn các biểu thức logic có ý nghĩa cả về mặt thời gian.
- Luận văn đề xuất ra kỹ thuật kiểm tra mô hình phần mềm bằng cách tích hợp các giải thuật tối ưu nhất như: mô hình hoá hệ thống bằng hệ thống chuyển trạng thái LTS, các giải thuật chuyển đổi giữa LTS sang Ôtômat Buchi, chuyển đổi giữa logic thời gian tuyến tính LTL sang Ôtômat Buchi, các giải thuật liên quan đến Ôtômat Buchi như tính tích chập, kiểm tra tính rỗng…Tất cả các giải thuật này đã được chọn lọc và kết hợp với nhau để tạo ra một kỹ thuật kiểm tra mô hình vừa hiệu quả lại mang tính thực thi cao.

- Luận văn đã giới thiệu hệ thống SPIN để minh hoạ việc kiểm tra mô hình phần mềm có sử dụng ngôn ngữ mô hình hoá hệ thống PROMELA áp dụng với các phần mềm tương tranh đa tiến trình. Luận văn đã giới thiệu chi tiết cú pháp của PROMELA và cách mô hình hoá hệ thống cũng như các thuộc tính của hệ thống là đầu vào cho mô hình SPIN. Dựa vào SPIN ta có thể thấy được bảng dịch chuyển trạng thái của hệ thống, từng bước chạy của bộ kiểm tra mô hình, dung lượng bộ nhớ cần thiết và tổng số trạng thái của hệ thống.
- Luận văn đã giải quyết được vấn đề kiểm tra mô hình phần mềm bằng cách tiếp cận nhiều hướng lý thuyết mới đồng thời đưa ra các hướng phát triển trong tương lai nhằm tối ưu hoá hơn nữa, hỗ trợ người sử dụng nhiều vấn đề hơn đồng thời tăng tính chính xác của việc mô hình hoá hệ thống đầu vào, điều đặc biệt quan trọng với việc kiểm tra mô hình phần mềm.

Hướng phát triển tiếp theo trong tương lai của đề tài:

- Tối ưu hoá hơn nữa các biện pháp giải quyết khi bùng nổ không gian trạng thái
- Kiểm tra mô hình phần mềm bằng cách sử dụng đầu vào là các biểu đồ trạng thái trong UML, áp dụng với các phần mềm hướng đối tượng
- Nghiên cứu để kiểm tra mô hình áp dụng với từng tiến trình, cho phép người sử dụng có thể kiểm tra ở mức tiến trình hoặc thành phần nhỏ.
- Xây dựng công cụ kiểm tra mô hình phần mềm sử dụng các ngôn ngữ lập trình thông dụng để mô hình hoá phần mềm như C, Java…

TÀI LIỆU THAM KHẢO

[1] Thomas Noll (2006), "Software Model Checking", *Summer term 2006* http://www-i2.informatik.rwth-aachen.de/Teaching/Course/SMC/2006/

[2] Dimitra Giannakopoulou (1999), "Model Checking for Concurrent Software Architectures", *Ph.D Thesis,University of Twente.* http://ase.arc.nasa.gov/people/dimitra/publications/thesis.pdf

[3] Dino Distefano (2003), "On model checking the dynamics of object – based software", *Ph.D Thesis*, *University of London.* http://www.dcs.qmul.ac.uk/%7Eddino/thesis.html

[4] Stephen Merz (2000), "Model Checking: A Toturial Overview" http://spinroot.com/spin/Doc/course/mc-tutorial.pdf

[5] Gerard J. Holzmann (2005), "Software Model Checking with SPIN", *Advances in Computers, Vol. 65, Ed. M. Zelkowitz, Elsevier Publisher, Amsterdam*

[6] Willem Visser (2002), "Software Model Checking" http://ase.arc.nasa.gov/visser/ASE2002TutSoftwareMC-fonts.ppt

[7] Patrice Godefroid (2005), "Software Model Checking: Searching for Computations in the Abstract or the Concrete" http://cm.bell-labs.com/who/god/public_psfiles/talk-ifm2005.pdf

[8] Tuba Yavuz-Kahveci and Tevfik Bultan (2003), "Automated Verification of Concurrent Linked Lists with Counters", P*roceedings of the 9th International Static Analysis Symposium (SAS 2002). M. V. Hermenegildo, G. Pueble eds., LNCS 2477, Springer, Madrid, Spain, September 2002*

[9] Maurice H. ter Beek (2005), "Model Checking with SPIN" ,*Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2005)*

[10] Javier Esparza and Stephan Merz, "Model Checking" http://www.informatik.uni-stuttgart.de/fmi/szs/people/esparza/Talks/slides-

[11] **www.Spinroot.com**

[12]Joost-Pieter Katoen (2005) , "Software Modeling & Verification" http://www-i2.informatik.rwth-aachen.de/Teaching/Course/MC/2005/mc_lec13.pdf

[13]Gerard J. Holzmann (1997), "The Model Checker Spin", *IEEE Transaction on software engineering, Vol. 23, No. 5, May 1997*

[14] G.J. Holzmann (2000), "Software Model Checking", *NATO Summer School, Vol. 180, pp. 309-355, IOS Press Computer and System Sciences, Marktoberdorf Germany, Aug. 2000.*

[15]A. Lluch-Lafuente, S. Edelkamp, S. Leue (2002), "Partial Order Reduction in Directed Model Checking", *In 9th International SPIN Workshop, SPIN 2002, LNCS 2318,Springer, 2002.*

[16] Klaus Havelund, Willem Visser (2000), "Program Model Checking as New Trend", NASA Ames Research Center, USA http://ase.arc.nasa.gov/visser/sttt-spin2000.pdf

[17]Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park (2000). "Model checking programs". *Proceedings of the 15th IEEE International Conference on Automated Software Engineering, Grenoble, France, September 2000.*

[18] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezz`e (2001). "Using symbolic execution for verifying safety-critical systems". In *ESEC/FSE-9: Proc. 8th European Software Engineering Conference*, *ACM Press, 2001.*

[19] G.J. Holzmann and M.H. Smith (2000), "Automating software feature verification", *Bell Labs Technical Journal, Vol. 5, No. 2, pp. 72-87, April-June 2000. (Special Issue on Software Complexity).*

[20]I. S. W. B. Prasetya, A. Azurat, T. E. J. Vos, and A. van Leeuwen (2005), "Building Verification Condition Generators by Compositional Extensions", *IEEE International Conference in Software Engineering & Formal Method 2005*

[21] Stefan Edelkamp (2005), "Tutorial on Directed Model Checking", *The International Conference on Automated Planning and Scheduling , ICAPS-2005*

[22] Javier Esparza (2004), "An Automata-Theoretic Approach to Software Model Checking"
http://www.informatik.uni-stuttgart.de/fmi/szs/people/esparza/Talks/POPL2004.pdf

[23] Howard Barringer (2006), "Advanced Algorithms Automata-based Verification"
http://www.cs.man.ac.uk/~david/courses/advalgorithms/automata4.pdf

[24] M. Daniele, F. Giunchiglia, and M. Y. Vardi (1999). "Improved automata generation for linear temporal logic". *In Proceedings of 11th Int. Conf. On Computer Aided Verification (CAV99), number 1633 in LNCS, 1999*