

Finite State Machines and Modal Models in Ptolemy II

Edward A. Lee



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-151
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-151.html>

November 1, 2009

Copyright © 2009, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force

Máy Trạng Thái Hữu Hạn và Mô hình Bộ nhớ trong Ptolemy II

Edward A. Lee
eal@eecs.berkeley.edu
Kỹ thuật Điện – Điện tử và Khoa học Máy tính
Đại học California, Berkeley
November 1, 2009

Lý thuyết

Báo cáo này miêu tả cách sử dụng và ngữ nghĩa Máy Trạng Thái Hữu Hạn cũng như Mô hình Bộ nhớ trong Ptolemy II. Máy Trạng Thái Hữu Hạn là những actor bao gồm những hành vi được miêu tả sử dụng hữu hạn các tập hợp trạng thái và chuyển trạng thái. Việc chuyển trạng thái được kích hoạt bởi các công, là những biến boolean có thể tham chiếu input tới actor và tham số. Việc chuyển trạng thái có thể cung cấp output và cập nhật giá trị của tham số. Mô hình Bộ nhớ mở rộng Máy Trạng Thái Hữu Hạn bằng cách cho phép trạng thái được tinh chỉnh, nói cách khác là mô hình Ptolemy II. Sự tinh chỉnh có thể chính là Máy Trạng Thái, Mô hình Bộ nhớ, hoặc bất kỳ actor phức tạp nào chứa những chỉ dẫn tương thích với Mô hình Bộ nhớ được sử dụng. Báo cáo này miêu tả ngữ nghĩa điều hành, sử dụng thực tế, và ngữ nghĩa về thời gian trong các Mô hình Bộ nhớ.

1 Giới thiệu

Hành vi của các actors trong Ptolmey II có thể được định nghĩa theo nhiều cách. Trong báo cáo này, chúng tôi giải thích cách gán hành vi của một actor như một cỗ máy trạng thái hữu hạn hoặc một Mô hình Bộ nhớ. Theo trực giác, trạng thái của một hệ thống hoặc hệ thống con là tình trạng của nó tại một thời điểm cụ thể. Nói chung, trạng thái ảnh hưởng đến cách hệ thống (phụ) phản ứng với đầu vào. Nói cách khác, chúng tôi xác định trạng thái là mã hóa mọi thứ về quá khứ có ảnh hưởng đến phản ứng của hệ thống đối với các đầu vào hiện tại hoặc tương lai.

LUU Y: Nếu bạn đang đọc tài liệu này trên màn hình (so với trên giấy) và bạn có kết nối mạng, thì bạn có thể nhấp vào các hình hiển thị các mô hình Ptolemy II để thực hiện và thử nghiệm trực tuyến các mô hình đó. Không cần cài đặt sẵn Ptolemy II hay bất kỳ phần mềm nào khác. Các mô hình được cung cấp trực tuyến được tóm tắt tại

<http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII8.0/jnlp-books/doc/books/design/modal/index.htm>.

Ví dụ, actor *Ramp*, tạo ra chuỗi đếm, có trạng thái. Phản ứng của nó đối với đầu vào *kích hoạt* phụ thuộc vào số lần nó đã bắn liên tục. Nó sử dụng một biến cục bộ để theo dõi xem nó đang ở đâu trong chuỗi đếm của nó. Biến cục bộ này được gọi là **biến trạng thái** và trong trường hợp này, biến trạng thái có (thường) là một giá trị số.

Trong trường hợp của actor Ramp, số trạng thái có thể có phụ thuộc vào kiểu dữ liệu của chuỗi đếm. Nếu nó là *int*, thì có 2^{32} trạng thái có thể. Nếu là *double* thì có 2^{64} . Số trạng thái có thể xảy ra là rất lớn. Nếu kiểu dữ liệu là *String*, thì số trạng thái có thể có là vô hạn. Mặc dù số lượng các trạng thái là rất lớn nhưng logic để thay đổi từ trạng thái này sang trạng thái tiếp theo lại khá đơn giản. Vì vậy, suy luận về hành vi của actor là không khó. Actor bắt đầu ở một trạng thái được cung cấp bởi tham số *init* của nó và trên mỗi lần kích hoạt, tăng trạng thái của nó bằng cách thêm vào đó giá trị của tham số *bước*. (Nếu kiểu dữ liệu là *String*, thì “thêm” có nghĩa là nối.)

Ngược lại, thông thường có các actor có số lượng trạng thái khả thi tương đối nhỏ, nhưng logic tương đối phức tạp để chuyển từ trạng thái này sang trạng thái tiếp theo. Các cơ chế được mô tả ở đây hỗ trợ thiết kế, trực quan hóa và phân tích những actor như vậy. Trước tiên, chúng tôi giải thích cơ sở hạ tầng Ptolemy II hỗ trợ các máy trạng thái hữu hạn (FSM), sau đó giải thích việc sử dụng FSM để xây dựng các Mô hình Bộ nhớ.

Phần tiếp theo dưới đây giải thích các FSM trong Ptolemy II và đưa ra một số ví dụ về cách sử dụng chúng. Phần sau đó giải thích các Mô hình Bộ nhớ, mở rộng các FSM với khả năng có các tinh chỉnh phân cấp của các trạng thái.

2 Máy Trạng Thái Hữu Hạn

Máy trạng thái là một hệ thống có đầu ra không chỉ phụ thuộc vào đầu vào hiện tại, mà còn phụ thuộc vào trạng thái hiện tại của hệ thống. Trạng thái của một hệ thống là một bản tóm tắt mọi thứ mà hệ thống cần biết về các đầu vào trước đó để tạo ra các đầu ra. Trạng thái của một hệ thống có thể được biểu diễn bằng một biến trạng thái $s \in \Sigma$, trong đó Σ là tập hợp tất cả các trạng thái có thể có của hệ thống. **Máy trạng thái hữu hạn (FSM)** là máy trạng thái trong đó Σ là tập hợp hữu hạn.

Ví dụ 1: Xét một bộ điều nhiệt điều khiển lò sưởi là một máy trạng thái với các trạng thái $\Sigma = \{sưởi ấm; làm mát\}$. Nếu trạng thái là $s = sưởi ấm$, thì máy sưởi đang bật. Nếu $s = làm mát$, thì bộ sưởi tắt. Giả sử nhiệt độ mục tiêu là 20 độ C. Nếu bật máy sưởi, thì bộ điều chỉnh nhiệt cho phép nhiệt độ tăng vượt quá mục tiêu, chẳng hạn như 22 độ. Nếu máy sưởi tắt, thì nó cho phép nhiệt độ giảm xuống mức mục tiêu, chẳng hạn như 18 độ. Do đó, hành vi phụ thuộc vào trạng thái, tóm tắt lịch sử bằng cách ghi nhớ liệu máy sưởi có đang tắt hay không. Chiến lược này tránh được sự **lộn xộn**, trong đó máy sưởi sẽ bật và tắt nhanh chóng khi nhiệt độ gần với nhiệt độ mục tiêu.

2.1 FSMAActor

FSMAActor là một actor tổng hợp trong đó quá trình tinh chỉnh bao gồm các trạng thái và chuyển tiếp hơn là các actor khác. Ptolemy II cung cấp một ký hiệu trực quan cho các trạng thái và quá trình chuyển đổi này như trong Hình 1. Trong hình đó, *FSMAActor* có hai cổng đầu vào và hai cổng đầu ra, được tạo bởi trình xây dựng mô hình. Nói chung, và *FSMAActor* có thể có bất kỳ số lượng cổng đầu vào và đầu ra nào. Actor phản ứng với các đầu vào và tạo ra các đầu ra như được chỉ định bởi một FSM, được hiển thị trực quan ở dưới cùng của hình. FSM chứa một số trạng thái hữu hạn (ba trạng thái trong hình). Một trong những trạng thái này là **trạng thái ban đầu** (được gọi là *initialState* trong hình), là trạng thái của actor khi bắt đầu thực thi mô hình. Trạng thái ban đầu được biểu thị bằng một đường viền đậm. Một số trạng thái cũng có thể là **trạng thái cuối cùng**, được biểu thị trực quan bằng một đường viền kép (thêm về các trạng thái cuối cùng sau). Các trạng thái được kết nối

bằng các **chuyển tiếp**, các chú thích xác định điều gì sẽ xảy ra khi actor kích hoạt. Trước khi đi sâu vào chi tiết, mặc dù, ví dụ về máy chỉnh nhiệt đã hỗ trợ việc hình dung.

Foundations: Model of State Machines

Các máy trạng thái thường được mô tả trong tài liệu dưới dạng năm bộ

$$(\Sigma, I, O, T, \sigma)$$

Σ là tập hợp các trạng thái và σ là trạng thái ban đầu. Các máy trạng thái không xác định có thể có nhiều hơn một trạng thái ban đầu, trong trường hợp đó $\sigma \subset \Sigma$ tự nó là một tập hợp, mặc dù khả năng cụ thể này không được hỗ trợ trong Ptolemy II FSM. I là một tập hợp các giá trị có thể có của các yếu tố đầu vào. Trong Ptolemy II

$$i: P_i \rightarrow D \cup \epsilon,$$

FSMs, I là một tập các hàm có dạng $i: P_i \rightarrow D \cup \epsilon$, trong đó P_i là tập hợp các cổng đầu vào (hoặc tên cổng đầu vào), D là tập hợp các giá trị có thể có trên các cổng đầu vào tại một lần kích hoạt cụ thể và ϵ đại diện cho các đầu vào “không có” (tức là,

$$(i.e., i(p) = \epsilon)$$

khi $p_isPresent$ đánh giá là sai). Tương tự, O là tập hợp tất cả các giá trị có thể có cho các cổng đầu ra tại một lần kích hoạt cụ thể.

Đối với một máy trạng thái xác định, T là một hàm có dạng

$$T: \Sigma \times I \rightarrow \Sigma \times O$$

, đại diện cho các mối quan hệ chuyển tiếp trong FSM. Trên thực tế, các bộ bảo vệ và các hành động đầu ra chỉ là mã hóa của chức năng này. Đối với một máy trạng thái không xác định (được hỗ trợ bởi Ptolemy II), mã miền của T là bộ quyền hạn của $\Sigma \times O$, cho phép có nhiều hơn một trạng thái đích và định giá đầu ra.

Lý thuyết cổ điển về máy trạng thái (Hopcroft và Ullman, 1979) tạo ra sự khác biệt giữa **máy Mealy** và **máy Moore**. Máy Mealy liên kết đầu ra với các chuyển tiếp. Máy Moore liên kết đầu ra với các trạng thái. Ptolemy II hỗ trợ cả hai, sử dụng các hành động đầu ra cho máy Mealy và tinh chỉnh trạng thái trong các Mô hình Bộ nhớ cho máy Moore.

Các máy trạng thái Ptolemy II thực sự là các FSM mở rộng, đòi hỏi một mô hình phong phú hơn mô hình đã nêu ở trên. Các máy trạng thái mở rộng thêm vào mô hình trên một tập V các giá trị biến, là các hàm có dạng $v: N \rightarrow D$, trong đó N là tập tên biến và D là tập giá trị mà biến có thể nhận. Máy trạng thái mở rộng là một bộ sáu bộ

$$(\Sigma, I, O, T, \sigma, V)$$

trong đó hàm chuyển tiếp hiện có dạng

$$T: \Sigma \times I \times V \rightarrow \Sigma \times O \times V$$

(đối với máy trạng thái xác định). Chức năng này được mã hóa bởi các chuyển tiếp, bảo vệ, hành động đầu ra và tập hợp các hành động của FSM.

Example 2: Mô hình của bộ điều nhiệt và lò sưởi được minh họa trong Hình 2. Người đọc có thể đọc hình đó mà không cần trợ giúp nhiều. Trong hình đó, *FSMActor* có đầu vào *nhiệt độ* và đầu ra *nhiệt*. Hành vi của nó được đưa ra bởi FSM được hiển thị trong hộp màu xám. Có hai trạng thái, $\Sigma = \{nung\ nồng; làm\ mát\}$. Có bốn chuyển tiếp. Bộ bảo vệ trên mỗi quá trình chuyển đổi đưa ra các điều kiện theo đó quá trình chuyển đổi được thực hiện. Các hành động đầu ra trên mỗi quá trình chuyển đổi cung cấp các giá trị được tạo ra trên các cổng đầu ra khi quá trình chuyển đổi được thực hiện. Đọc sơ đồ, ta thấy khi ở trạng thái *gia nhiệt*, nếu nhiệt độ đầu vào nhỏ hơn *heatOffThreshold* (22.0) thì giá trị đầu ra là *heatRate* (0.1). Khi đầu vào *nhiệt độ* lớn hơn hoặc bằng *heatOffThreshold*, thì FSM sẽ chuyển sang trạng thái *làm mát* và tạo ra giá trị đầu ra được cung cấp bởi *coolingRate* (-0,05). Một ví dụ thực thi được vẽ trong Hình 4.

2.2 Execution Policy for an *FSMActor*

Một *FSMActor* chứa một tập hợp các trạng thái và chuyển tiếp. Một trong các trạng thái là trạng thái ban đầu và bất kỳ số lượng trạng thái nào cũng có thể là trạng thái cuối cùng. Mỗi quá trình chuyển đổi có một **biểu thức bảo vệ**, bất kỳ số lượng **hành động đầu ra** nào và bất kỳ **số lượng hành động** đã đặt nào. Khi bắt đầu thực hiện, trạng thái của actor được đặt thành trạng thái ban đầu. Sau đó, mỗi lần kích hoạt actor là một chuỗi các bước như sau.

Trong phương thức *fire()*, actor

1. đọc đầu vào;
2. đánh giá các bảo vệ về các chuyển tiếp đi của trạng thái hiện tại;
3. chọn một quá trình chuyển đổi mà bộ bảo vệ đánh giá là đúng; và
4. thực hiện các hành động đầu ra trên quá trình chuyển đổi đã chọn, nếu có.

Trong phương thức *postfire()*, actor

5. thực hiện các hành động đã đặt của quá trình chuyển đổi đã chọn; và
6. thay đổi trạng thái hiện tại thành đích của quá trình chuyển đổi đã chọn.

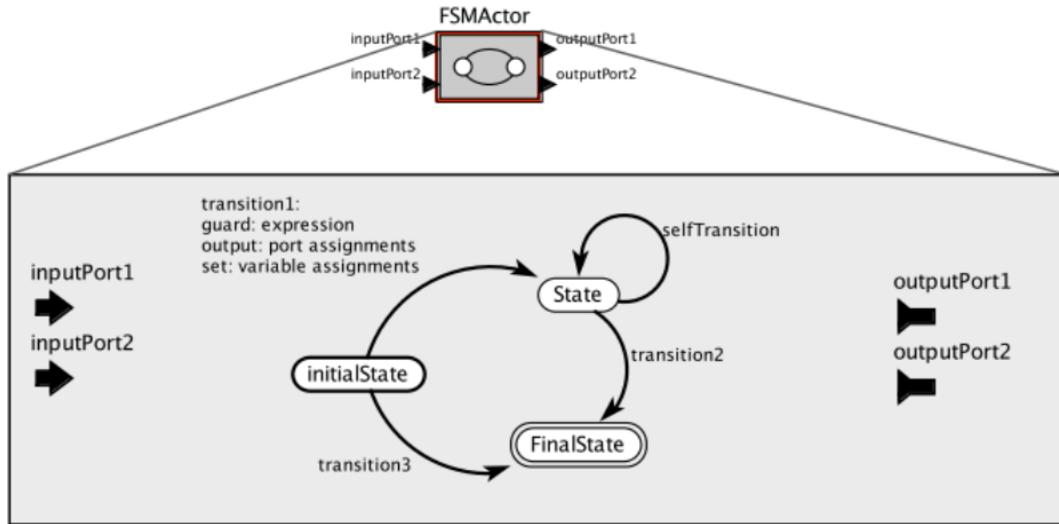


Figure 1: Visual notation for state machines in Ptolemy II.

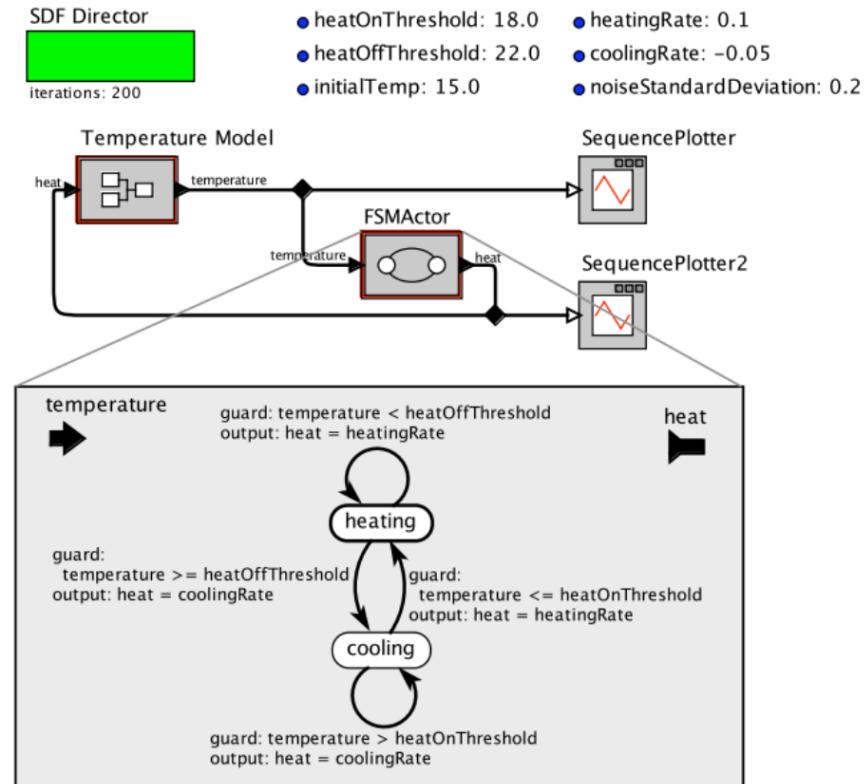


Figure 2: A model of a thermostat with hysteresis. The Temperature Model actor is shown in Figure 3.

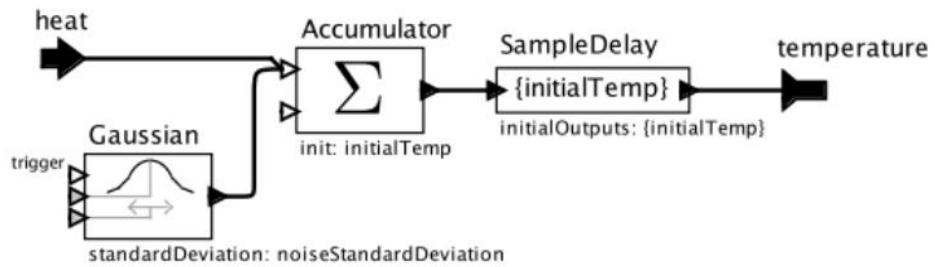


Figure 3: The Temperature Model composite actor of Figure 2.

Chúng được tách thành hai phương pháp riêng biệt để hỗ trợ việc sử dụng actor này trong các miền thực hiện phép lặp điểm cố định (chẳng hạn như SR và Liên tục), như được giải thích bên dưới trong Phần 2.9. Trong các miền không thực hiện điều đó (chẳng hạn như PN, SDF và DDF), các bước từ 1 đến 6 có thể thực hiện theo trình tự trong mỗi lần lặp và sự khác biệt giữa *fire()* và *postfire()* là không quan trọng.

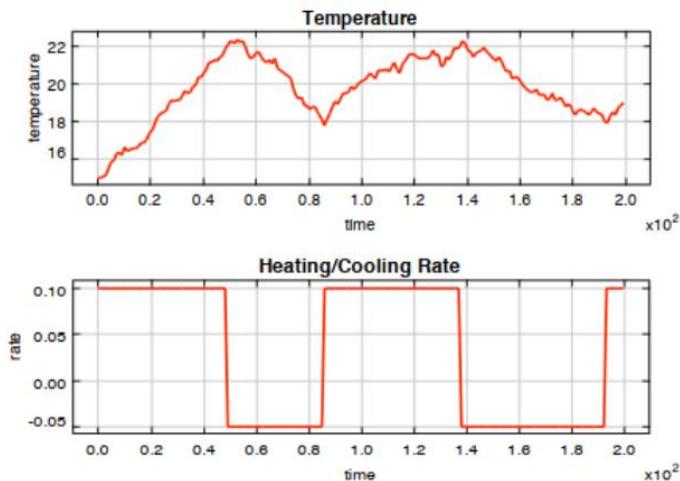


Figure 4: Two plots generated by Figure 2, showing the temperature (above) and whether the heater is on or off (below), both as a function of time.

Probing Further: Hysteresis

Bộ điều nhiệt trong ví dụ 2 thể hiện một dạng đặc biệt của hành vi phụ thuộc vào trạng thái được gọi là **hiện tượng trễ**. Một hệ thống có độ trễ có đặc tính là thang thời gian tuyệt đối không liên quan. Giả sử đầu vào là

một hàm của thời gian $x: \mathbb{R} \rightarrow \mathbb{R}$ (đối với bộ đSiêu nhiệt, $x(t)$ là nhiệt độ tại thời điểm t). Giả sử rằng đầu vào x gây ra đầu ra

$y: \mathbb{R} \rightarrow \mathbb{R}$, cũng là một hàm của thời gian. Ví dụ, trong Hình 4, x là tín hiệu trên và y là tín hiệu dưới. Đối với hệ thống này, nếu thay vì x là đầu vào là x' được cho bởi

$$x'(t) = x(\alpha \cdot t)$$

đối với hằng số không âm α , thì đầu ra là y' được cho bởi

$$y'(t) = y(\alpha \cdot t) .$$

Chia tỷ lệ trực thời gian ở đầu vào dẫn đến tỷ lệ trực thời gian ở đầu ra, do đó, tỷ lệ thời gian tuyệt đối là không liên quan.

Một cách triển khai thay thế cho bộ điều nhiệt sẽ sử dụng một ngưỡng nhiệt độ duy nhất, nhưng thay vào đó sẽ yêu cầu bộ sưởi duy trì bật hoặc tắt trong ít nhất một khoảng thời gian tối thiểu, bất kể nhiệt độ là bao nhiêu. Thiết kế này sẽ không có thuộc tính trễ.

Probing Further: Internal Structure of FSMActor

FSMActor là một lớp con của *CompositeEntity*, giống như *CompositeActor*. Về mặt nội tại, nó chứa một số lượng thể hiện của *Trạng thái* và *Chuyển tiếp*, tương ứng là các lớp con của *Thực thể* và *Quan hệ*. Cấu trúc đơn giản hiển thị dưới đây:



được thể hiện trong MoML như sau:

```
<entity name="FSMActor" class="...FSMActor">
    <entity name="State1" class="...State">
        <property name="isInitialState" class="...Parameter"
            value="true"/>
    </entity>
    <entity name="State2" class="...State"/>
    <relation name="relation" class="...Transition"/>
    <relation name="relation2" class="...Transition"/>
    <link port="State1.incomingPort" relation="relation2"/>
    <link port="State1.outgoingPort" relation="relation"/>
    <link port="State2.incomingPort" relation="relation"/>
    <link port="State2.outgoingPort" relation="relation2"/>
</entity>
```

Cấu trúc tương tự có thể được chỉ định trong Java như sau:

```
import ptolemy.domains.modal.kernel.FSMActor;
import ptolemy.domains.modal.kernel.State;
import ptolemy.domains.modal.kernel.Transition;

FSMActor actor = new FSMActor();
State state1 = new State(actor, "State1");
State state2 = new State(actor, "State2");
Transition relation = new Transition(actor, "relation");
Transition relation2 = new Transition(actor, "relation2");
state1.incomingPort.link(relation2);
state1.outgoingPort.link(relation);
state2.incomingPort.link(relation);
state2.outgoingPort.link(relation2);
```

Như vậy, ở trên, chúng ta thấy ba cú pháp cụ thể riêng biệt cho cùng một cấu trúc.

Sau khi đọc các đầu vào, actor này kiểm tra các chuyển đổi đi ra của trạng thái hiện tại, đánh giá các biểu thức bảo vệ của chúng. Quá trình chuyển đổi được *kích hoạt* nếu biểu thức bảo vệ của nó đánh giá là đúng. Một biểu thức bảo vệ trống được hiểu là luôn luôn đúng. Biểu thức bảo vệ có thể đề cập đến bất kỳ công đầu vào nào và bất kỳ biến nào trong phạm vi.

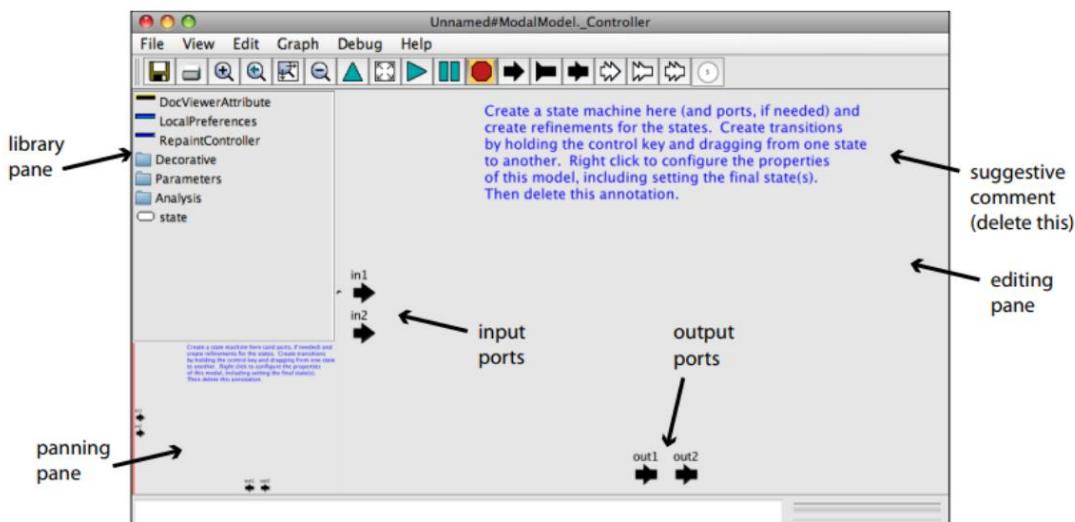


Figure 5: Editor for FSMs in Vergil, showing two input and two output ports, before being populated with an FSM.

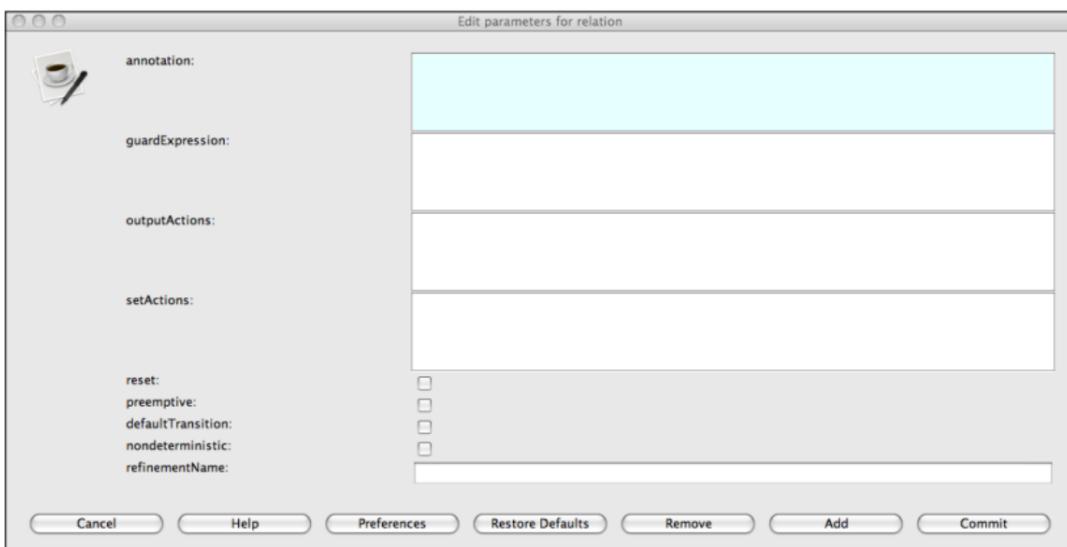


Figure 6: Dialog box for configuring a transition in an FSM.

2.3 Referencing Input Values of an FSMAActor

Nếu tên cổng đầu vào *portName* được sử dụng trong biểu thức bảo vệ, thì nó đề cập đến đầu vào hiện tại trên cổng đó trên kênh 0. Ví dụ, trong Hình 2, trong biểu thức bảo vệ “*temperature < heatOffThreshold*”, biến *temperature* đề cập đến giá trị hiện tại trong cổng đầu vào *temperature* của cổng đầu vào và *heatOffThreshold* đề cập đến tham số có tên *heatOffThresh-old*.

Trong nhiều mô hình tính toán, đầu vào có thể *không có*. Nếu không có cổng p, thì một biểu thức như “*p < 10*” sẽ được đánh giá là sai. Tuy nhiên, một biểu thức như “*p < 10 || true*” đánh giá là true. Một kỹ thuật rõ ràng hơn dẫn đến các mô hình máy trạng thái dễ đọc hơn là sử dụng ký hiệu *portName_isPresent* trong các biểu thức bảo vệ. Đây là một boolean đúng nếu có đầu vào trên cổng *portName*.

Nhớ lại rằng một multiport có thể có nhiều kênh. Để chỉ một kênh cụ thể, một biểu thức bảo vệ có thể sử dụng ký hiệu *portName_channellIndex*, trong đó *channellIndex* là một số nguyên từ 0 đến n - 1, trong đó n là số kênh được kết nối với cổng. Biểu tượng này sẽ đánh giá giá trị nhận được trên cổng trên kênh đã cho. Tương tự, một biểu thức bảo vệ có thể tham chiếu đến *portName_channellIndex-isPresent*.

Mechanics: Creating FSMs in Vergil

FSMActor trong Vergil nằm trong *MoreLibraries/Automata*. Bạn cũng có thể sử dụng *ModalModel* từ thư viện *Utilities*. Chúng tôi khuyên bạn nên sử dụng *ModalModel* vì nó là một actor tổng quát hơn và có thể làm mọi thứ mà *FSMActor* có thể làm.

Đầu tiên, kéo actor vào mô hình của bạn từ thư viện. Điền các cổng đầu vào và đầu ra cho actor bằng cách nhấp chuột phải (hoặc bấm điều khiển trên máy Mac) và chọn [*Open Actor*]. Cửa sổ kết quả được hiển thị trong Hình 5. Nó tương tự như các cửa sổ Vergil khác, nhưng có một thư viện tùy chỉnh bao gồm chủ yếu là *Trạng thái*, thư viện tham số và thư viện các yếu tố trang trí để chú thích thiết kế của bạn. Nó cũng bao gồm một chú thích văn bản với một bình luận gợi ý mà bạn sẽ muốn xóa sau khi đọc. Kéo trong một hoặc nhiều trạng thái. Để tạo chuyển tiếp giữa các trạng thái, **hãy giữ phím điều khiển** (hoặc phím Command trên máy Mac) và nhấp và kéo từ trạng thái này sang trạng thái khác. Tay nắm trên các phần chuyển tiếp có thể được sử dụng để kiểm soát độ cong và vị trí của các phần chuyển tiếp. Nhấp đúp chuột (hoặc nhấp chuột phải và chọn [*Configure*]) trên quá trình chuyển đổi để đặt bảo vệ, hành động đầu ra và đặt hành động bằng cách nhập văn bản vào hộp thoại như trong Hình 6. Bạn cũng có thể chỉ định một chú thích liên quan đến quá trình chuyển đổi có không ảnh hưởng đến việc thực thi và do đó hoạt động như một nhận xét.

2.4 Output Actions

Khi một quá trình chuyển đổi được chọn, các **hành động đầu ra** của nó sẽ được thực thi. Hành động đầu ra được chỉ định bởi tham số *outputActions* của quá trình chuyển đổi. Hình thức của một hành động đầu ra thường là *portName = biểu thức*, trong đó biểu thức có thể tham chiếu đến các giá trị đầu vào như trên hoặc bất kỳ tham số nào trong phạm vi. Ví dụ, trong Hình 2, dòng

```
output: heat = coolingRate
```

chỉ định rằng công đầu ra có tên *heat* sẽ tạo ra giá trị được cung cấp bởi tham số *coolingRate*. Điều này mang lại hành vi của **máy Mealy**, là một kiểu máy trạng thái trong đó đầu ra được tạo ra bởi các chuyển đổi chứ không phải bởi các trạng thái. Hành vi của **máy Moore** cũng có thể đạt được bằng cách sử dụng các tinh chỉnh trạng thái tạo ra đầu ra, như được giải thích bên dưới trong Phần 3.

Có thể đưa ra nhiều hành động đầu ra bằng cách phân tách chúng bằng dấu chấm phẩy, như trong *port1 = expression; port1 = expression1*.

2.5 Set Actions and Extended Finite State Machines

Các **hành động** đã đặt trên quá trình chuyển đổi có thể được sử dụng để đặt giá trị của các tham số của máy trạng thái. Một ứng dụng thực tế cho điều này là tạo ra một **FSM mở rộng**, là một máy trạng thái hữu hạn được mở rộng với một biến trạng thái số. Nó được gọi là “mở rộng” bởi vì số trạng thái bây giờ phụ thuộc vào số lượng giá trị riêng biệt mà biến có thể nhận. Nó thậm chí có thể là vô hạn.

Example 3: Một ví dụ đơn giản về FSM mở rộng được hiển thị trong Hình 7, một lần nữa có thể đọc được mà không cần trợ giúp nhiều. Trong ví dụ này, *FSMActor* có bộ đếm tham số, được hiển thị với dấu đầu dòng nhỏ màu xanh bên cạnh. Quá trình chuyển đổi từ trạng thái ban đầu *init* sang trạng thái *đếm* khởi tạo *đếm* thành 0 trong hành động thiết lập của nó. Trạng thái *đếm* có hai quá trình chuyển tiếp đi, một là tự chuyển đổi và trạng thái còn lại là chuyển tiếp *cuối cùng*. Quá trình tự chuyển đổi được thực hiện miễn là số *đếm* nhỏ hơn 5. Quá trình chuyển đổi đó tăng giá trị của số *đếm* lên một. Khi giá trị của *đếm* đạt đến 5, quá trình chuyển đổi sang *cuối cùng* được thực hiện. Trước khi thực hiện quá trình chuyển đổi đó, đầu ra được đặt bằng đầu vào. Khi thực hiện quá trình chuyển đổi đó, đầu ra từ đó trở đi không đổi. Do đó, đầu ra của mô hình này là chuỗi 0, 1, 2, 3, 4, 5, 5, 5...

2.6 Final States

Một FSM có thể có các **trạng thái cuối cùng**, là các trạng thái mà khi được nhập vào, cho biết sự kết thúc thực thi của máy trạng thái.

Example 4: Một biến thể của Ví dụ 3 được hiển thị trong Hình 8. Biến thể này có tham số *isFinalState* của trạng thái *cuối cùng* được đặt thành *true*. Điều này được biểu thị bằng đường viền kép xung quanh trạng thái. Khi vào trạng thái đó, *FSMActor* chỉ ra cho chỉ dẫn kèm theo rằng nó không muốn thực thi nữa (nó thực hiện điều này bằng cách trả về *false* từ phương thức *postfire()* của nó). Kết quả là, đầu ra được gửi đến actor *Hiển thị* là chuỗi hữu hạn 0, 1, 2, 3, 4, 5, 5.

Trong lần lặp mà *FSMActor* đi vào trạng thái được đánh dấu là cuối cùng, phương thức *postfire()* của *FSMActor* trả về *false*. Điều này cho chỉ dẫn kèm theo biết rằng *FSMActor* không muốn bị kích hoạt lần nữa. Hầu hết các chỉ dẫn sẽ chỉ đơn giản là tránh kích hoạt actor đó một lần nữa, nhưng sẽ tiếp tục thực hiện mô hình. Tuy nhiên, chỉ dẫn SDF thì khác. Vì nó giả định tỷ lệ tiêu thụ và sản xuất thường xuyên cho tất cả các actor và vì nó xây dựng lịch trình của mình một cách cố định, nên nó không thể chấp nhận những actor từ chối kích hoạt. Do đó, chỉ dẫn SDF sẽ ngừng thực thi mô hình hoàn toàn nếu bất kỳ actor nào trả về *false* từ *postfire()*.

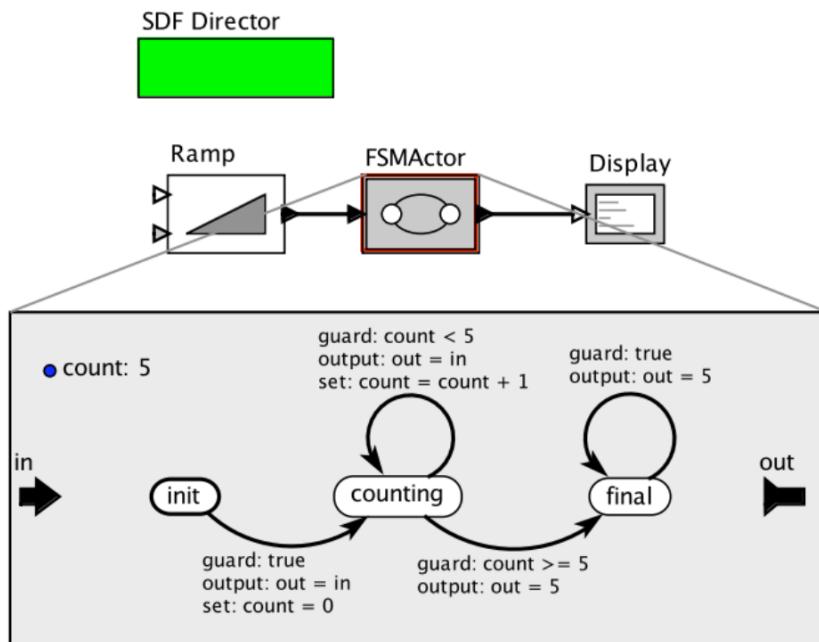


Figure 7: An extended FSM, where the *count* variable is part of the state of the system.

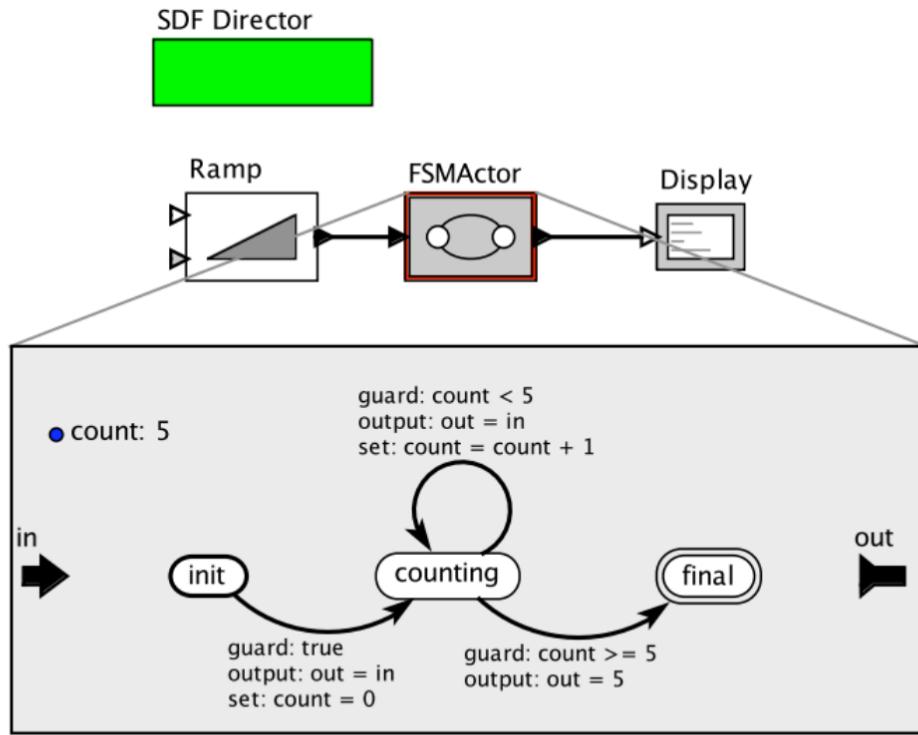


Figure 8: A state machine with a final state, which indicates the end of execution of the state machine.

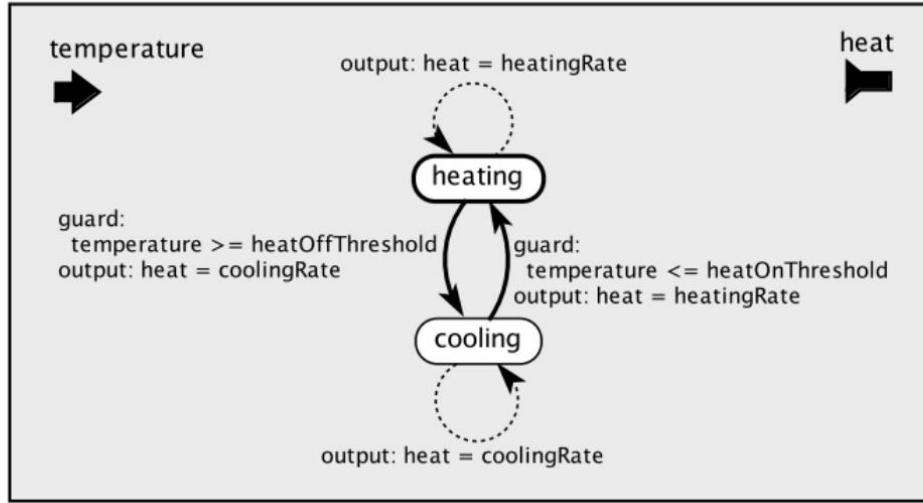


Figure 9: An FSM equivalent to the one in Figure 2, but using default transitions, indicated with dashed lines.

2.7 Default Transitions

Một FSM có thể có các **chuyển đổi mặc định**, là các chuyển đổi có tham số *mặc định* được đặt thành true (xem Hình 6). Các chuyển đổi này được bật nếu không có chuyển đổi đi nào khác của trạng thái hiện tại được bật. Các chuyển tiếp mặc định được hiển thị dưới dạng các cung có nét đứt thay vì các cung liền nết.

Example 5: FSM của Hình 2 có thể được triển khai tương đương bằng cách sử dụng các chuyển đổi mặc định như trong Hình 9. Ở đây, các chuyển đổi mặc định chỉ cần xác định rằng quá trình chuyển đổi đi sang trạng thái khác không được kích hoạt, sau đó FSM sẽ giữ nguyên trạng thái và tạo ra một đầu ra.

Nếu quá trình chuyển đổi mặc định cũng có biểu thức bảo vệ, thì quá trình chuyển đổi đó chỉ được bật nếu trình bảo vệ đánh giá là đúng và không có quá trình chuyển đổi không mặc định nào khác được bật. Lưu ý rằng việc sử dụng các chuyển đổi mặc định với các mô hình tính toán theo thời gian có thể hơi phức tạp. Xem Phần 3.7 bên dưới.

2.8 Nondeterministic State Machines

Nếu có nhiều hơn một bộ bảo vệ đánh giá là đúng tại bất kỳ thời điểm nào, thì FSM là **không xác định**. Các chuyển đổi có các bộ bảo vệ đánh giá đồng thời thành true được gọi là **các chuyển đổi không xác định**. Theo mặc định, các chuyển đổi không được phép là không xác định, vì vậy nếu có nhiều hơn một bộ bảo vệ đánh giá là đúng, bạn sẽ thấy một ngoại lệ giống như sau:

```
Nondeterministic FSM error: Multiple enabled transitions found but not
all of them are marked nondeterministic.
```

```
in ... name of a transition not so marked ...
```

Để cho phép chúng không xác định, hãy đặt tham số *không xác định* thành *true* trên mọi chuyển đổi có thể được bật trong khi một chuyển đổi khác được bật.

Example 6: Một mô hình của bộ điều nhiệt bị lỗi được hiển thị trong Hình 10. Khi FSM ở trạng thái *heating*, cả hai quá trình chuyển đổi đi ra đều được kích hoạt (cả hai bộ bảo vệ của chúng đều *đúng*), vì vậy có thể thực hiện một trong hai. Cả hai quá trình chuyển đổi đều được đánh dấu là không xác định, một thực tế được biểu thị trực quan bằng cách hiển thị các quá trình chuyển đổi bằng màu đỏ. Sơ đồ thực hiện được hiển thị trong Hình 11. Lưu ý rằng máy sưởi hiện đang bật trong khoảng thời gian khá ngắn, khiến nhiệt độ dao động quanh 18 độ, ngưỡng mà máy sưởi được bật.

Trong một FSM không xác định, nếu có nhiều hơn một quá trình chuyển đổi được bật và tất cả chúng đều được đánh dấu là không xác định, thì một chuyển đổi được chọn ngẫu nhiên trong phương thức *fire()* của actor *FSM*. Nếu phương thức *fire()* được gọi nhiều lần trong một lần lặp (xem Phần 2.9 bên dưới), thì các lần gọi tiếp theo trong cùng một lần lặp sẽ luôn chọn cùng một quá trình chuyển đổi.

Nếu có nhiều hơn một quá trình chuyển đổi mặc định rời khỏi trạng thái, thì những quá trình chuyển đổi này cũng phải được đánh dấu là không xác định nếu không sẽ dẫn đến một ngoại lệ. Các chuyển đổi mặc định không xác định được hiển thị dưới dạng các cung có nét đứt màu đỏ.

2.9 Fixed-Point Iterations

Phần này có thể được bỏ qua trong lần đọc đầu tiên trừ khi bạn đặc biệt tập trung vào các miền điểm cố định như SR và Liên tục.

Trong Phần 2.2 ở trên, chúng tôi giải thích rằng việc thực thi một *FSMActor* được chia thành hai phân đoạn, bước 1-4, được thực hiện trong phương thức *fire()* và bước 5-6, được thực hiện trong phương thức *postfire()*. Sự tách biệt này rất quan trọng trong các miền thực hiện phép lặp điểm cố định, chẳng hạn như SR và Liên tục. Trong các miền có phép lặp điểm cố định, phương thức *fire()* có thể được gọi nhiều lần trong một phép lặp trong khi giám đốc tìm kiếm giải pháp. Đối với các miền như vậy, điều quan trọng là phương thức *fire()* không bao gồm bất kỳ thay đổi trạng thái liên tục nào. Các bước 1-4 đọc đầu vào, đánh giá các bộ bảo vệ, chọn chuyển đổi và tạo đầu ra, nhưng chúng không cam kết chuyển đổi trạng thái hoặc thay đổi giá trị của bất kỳ biến cục bộ nào.

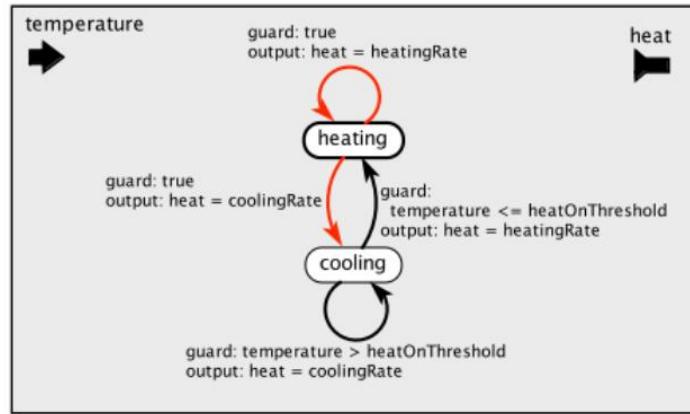


Figure 10: A model of a faulty thermostat that nondeterministically switches from heating to cooling.

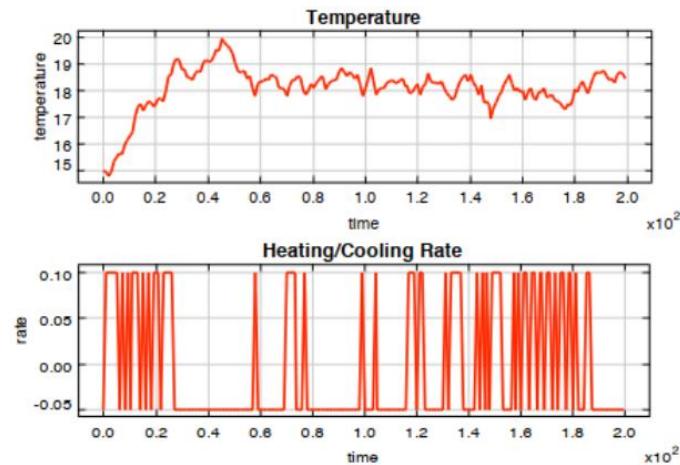


Figure 11: Plot of the thermostat FSM of Figure 10 replacing that in Figure 2.

Có thể hiểu lý do của sự tách biệt này bằng cách nghiên cứu ví dụ trong Hình 12. Việc thực thi mô hình SR liên quan đến việc tìm giá trị cho từng tín hiệu tại mỗi lần đánh dấu của đồng hồ toàn cục. Ở lần đánh dấu đầu tiên, mỗi actor *NonStrictDelay* đặt giá trị được hiển thị trong biểu tượng của nó trên cổng đầu ra của nó (các giá trị lần lượt là 1 và 2). Điều này xác định giá trị *in1* cho *FSMActor1* và giá trị *in2* cho *FSMActor2*. Nhưng các cổng đầu vào khác vẫn

chưa được xác định. Giá trị của $in2$ của $FSMActor1$ được chỉ định bởi $FSMActor2$ và giá trị của $in1$ của $FSMActor2$ được chỉ định bởi $FSMActor1$. Điều này dường như tạo ra một vòng lặp nhân quả, nhưng kiểm tra kỹ các máy trạng thái cho thấy rằng không có vòng lặp nhân quả nào.

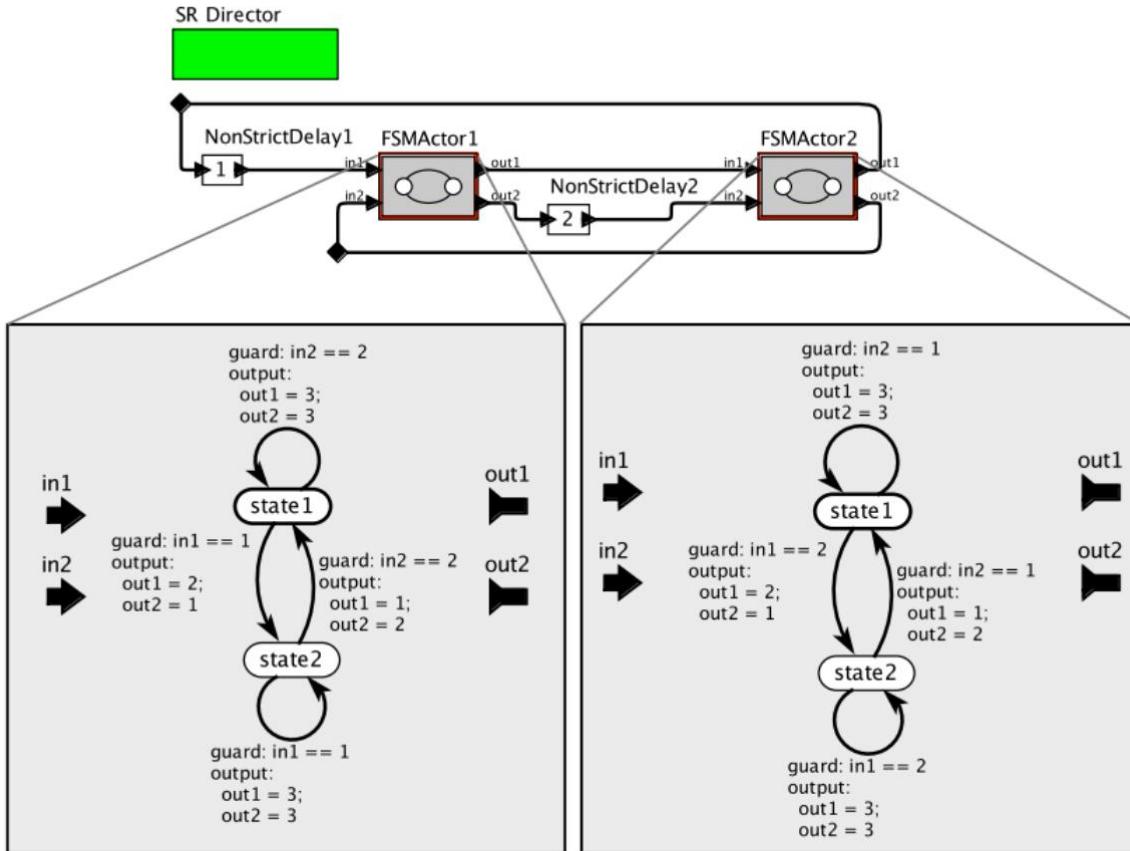


Figure 12: A model that requires separation of actions between the `fire()` method and the `postfire()` method in order to be able to converge to a fixed point.

Trong Hình 12, lưu ý đối với tất cả các trạng thái của $FSMActors$, mỗi cổng đầu vào có một bộ bảo vệ phụ thuộc vào giá trị của nó. Do đó, có vẻ như cả hai đầu vào cần phải được biết trước khi bất kỳ giá trị đầu ra nào có thể được khẳng định, điều này một lần nữa cho thấy một vòng lặp. Tuy nhiên, nhìn kỹ vào FSM bên trái, chúng ta thấy rằng quá trình chuyển đổi từ *trạng thái 1* sang *trạng thái 2* sẽ được kích hoạt ở lần tích tắc đầu tiên của đồng hồ vì $in1$ có giá trị 1, được đưa ra bởi $NonStrictDelay1$. Nếu máy trạng thái là xác định, thì đây phải là quá

trình chuyển đổi được kích hoạt duy nhất. Vì không có quá trình chuyển đổi không xác định trong máy trạng thái, nên chúng ta có thể cho rằng đây sẽ là quá trình chuyển đổi được chọn. Khi chúng tôi đưa ra giả định đó, chúng tôi có thể xác nhận cả hai giá trị đều ra như được hiển thị trên quá trình chuyển đổi (*out1* là 2 và *out2* là 1).

Khi chúng tôi xác nhận các giá trị đều ra đó, thì cả hai đều vào của *FSMActor2* đều được biết và nó có thể kích hoạt. Đầu vào của nó là *in1* = 2 và *in2* = 2, vì vậy ở máy trạng thái phù hợp, quá trình chuyển đổi từ *trạng thái 1* sang *trạng thái 2* được kích hoạt. Quá trình chuyển đổi này khẳng định rằng *out2* của *FSMActor2* có giá trị 1, vì vậy bây giờ cả hai đều vào của *FSMActor1* đều có giá trị 1. Điều này khẳng định lại rằng *FSMActor1* có chính xác một chuyển đổi được kích hoạt, chuyển đổi từ *trạng thái 1* sang *trạng thái 2*.

Thật dễ dàng để xác minh rằng tại mỗi lần tích tắc của clock, cả hai đầu vào của mỗi máy trạng thái đều có cùng giá trị, do đó, không có trạng thái nào có nhiều hơn một chuyển đổi ra ngoài được kích hoạt. Quyết định được bảo toàn. Hơn nữa, các giá trị của các đầu vào này thay thế giữa 1 và 2 trong các tích tắc tiếp theo. Cho *FSMActor1*, the inputs are 1, 2, 1,... trong tick 1, 2, 3... Cho *FSMActor2*, đầu vào là 2, 1, 2,... trong tick 1, 2, 3,...

Như đã giải thích ở phần 2.2 ở trên, trong phương thức *fire()*, actor

1. đọc đầu nhập;
2. đánh giá các bảo vệ về các chuyển tiếp đi của trạng thái hiện tại;
3. chọn một quá trình chuyển đổi mà bộ bảo vệ đánh giá là đúng; và
4. thực hiện các hành động đầu ra trên quá trình chuyển đổi đã chọn, nếu có.

Trong một lần lặp lại điểm cố định, điều này có thể xảy ra nhiều lần và có những điểm tinh tế liên quan đến từng bước. Đặc biệt:

1. *reads inputs*: Một số đầu vào có thể không được biết đến. Đầu vào không xác định không thể đọc được, vì vậy actor đơn giản là không đọc chúng.
2. *evaluates guards on outgoing transitions of the current state*: Một số bộ phận bảo vệ này có thể phụ thuộc vào đầu vào không xác định.

Những người bảo vệ này có thể hoặc không thể được đánh giá. Ví dụ:

nếu biểu thức bảo vệ là “*true || in1*” thì có thể đánh giá xem đầu vào *in1* có được biết hay không. Nếu một bộ bảo vệ không thể được đánh giá, thì nó không được đánh giá.

3. *chooses a transition whose guard evaluates to true:* Nếu chính xác một quá trình chuyển đổi có một bộ bảo vệ đánh giá là đúng, thì hãy chọn quá trình chuyển đổi đó. Nếu một quá trình chuyển đổi đã được chọn trong lần gọi phương thức *fire()* trước đó trong cùng một lần lặp, thì actor sẽ kiểm tra xem lần chuyển đổi *tương tự* có được chọn hay không. Nếu không, nó ném ra một ngoại lệ và quá trình thực thi bị tạm dừng. FSM không được phép thay đổi quá trình chuyển đổi nào nó sẽ chọn một trong các quá trình chuyển đổi. Các lần gọi tiếp theo của phương thức *fire()* trong cùng một lần lặp lại sẽ chọn cùng một quá trình chuyển đổi.
4. *executes the output actions on the chosen transition, if any:* Nếu một quá trình chuyển đổi được chọn, thì tất cả các giá trị đầu ra đều có thể được xác định. Một số trong số này có thể được chỉ định khi chuyển đổi chính nó. Nếu chúng không được chỉ định, thì chúng được khẳng định là *không có* tại dấu tích này. Nếu tất cả các quá trình chuyển đổi bị vô hiệu hóa (tất cả các bộ bảo vệ đánh giá là sai), thì tất cả các đầu ra được đặt thành *không có*. Nếu không có quá trình chuyển đổi nào được chọn nhưng ít nhất một quá trình chuyển đổi vẫn còn bộ bảo vệ mà không thể đánh giá, thì kết quả đầu ra vẫn chưa xác định.

Trong phương thức *postfire()*, actor

5. thực hiện các hành động đã đặt của quá trình chuyển đổi đã chọn; và
6. thay đổi trạng thái hiện tại thành đích của quá trình chuyển đổi đã chọn.

Các hành động này được thực hiện chính xác một lần sau khi phép lặp điểm cố định đã xác định tất cả các giá trị tín hiệu. Nếu bất kỳ giá trị tín hiệu nào vẫn chưa được xác định ở cuối quá trình lặp, thì một ngoại lệ sẽ được đưa ra. Mô hình bị lỗi.

Các FSM không xác định trong một miền điểm cố định có một số điểm tinh tế. Có thể xây dựng một mô hình trong đó có một điểm cố định có hai chuyển tiếp được kích hoạt nhưng ở đó việc lựa chọn giữa các chuyển tiếp không thực sự ngẫu nhiên. Có thể là chỉ một trong số các quá trình chuyển đổi được

chọn. Điều này xảy ra khi có nhiều lời gọi phương thức *fire()* trong phép lặp điểm cố định và trong lần đầu tiên của những lời gọi này, một trong các bộ bảo vệ không thể được đánh giá vì nó phụ thuộc vào một điều kiện không xác định. Nếu bộ bảo vệ khác có thể được đánh giá trong lần gọi *fire()* đầu tiên, thì quá trình chuyển đổi khác sẽ luôn được chọn. Kết quả là, đối với các máy trạng thái không xác định, hành vi có thể phụ thuộc vào thứ tự kích hoạt trong một phép lặp điểm cố định.

Lưu ý rằng các chuyển đổi mặc định cũng có thể được đánh dấu là không xác định. Tuy nhiên, quá trình chuyển đổi mặc định sẽ không được chọn trừ khi tất cả các quá trình chuyển đổi không mặc định đều có bộ bảo vệ đánh giá là *false*. Cụ thể hơn, nó sẽ không được chọn nếu bất kỳ quá trình chuyển đổi không mặc định nào có một bộ bảo vệ chưa thể được đánh giá do điều kiện không xác định. Nếu tất cả các chuyển đổi không mặc định đều có bộ bảo vệ đánh giá là sai và có nhiều hơn một chuyển đổi mặc định, tất cả được đánh dấu là không xác định, thì một chuyển đổi được chọn ngẫu nhiên.

3 Modal Models

Hầu hết các hệ thống thú vị đều có hành vi thay đổi theo thời gian. Chẳng hạn, những thay đổi trong hành vi có thể được kích hoạt bởi điều kiện của người dùng, lỗi phần cứng hoặc dữ liệu cảm biến. Một **Mô hình Bộ nhớ** là một đại diện rõ ràng của một tập hợp hữu hạn các hành vi và các quy tắc chỉ phối quá trình chuyển đổi giữa các hành vi. Việc chuyển đổi giữa các hành vi được điều chỉnh bởi một FSM.

ModalModel là một actor phân cấp, giống như một actor tổng hợp, nhưng có nhiều cải tiến thay vì chỉ một. Mỗi tinh chỉnh cho một hành vi. Máy trạng thái xác định tinh chỉnh nào đang hoạt động tại bất kỳ thời điểm nào. Actor *ModalModel* là một dạng tổng quát hơn của *FSMActor* được mô tả trong phần trước. *FSMActor* không hỗ trợ tinh chỉnh trạng thái. Bạn có thể luôn sử dụng *ModalModel* thay vì *FSMActor* và không tạo các tinh chỉnh trạng thái. Do đó, không có lý do thực sự để sử dụng *FSMActor*.

Example 7: Mô hình được hiển thị trong Hình 13 có một actor được gắn nhãn “Mô hình Bộ nhớ” có hai chế độ, *clean* và *noisy*. Mô hình này là một kênh truyền thông với hai chế độ hoạt động. Ở chế độ *clean*, nó

chuyển đầu vào đến đầu ra không thay đổi. Ở chế độ *noisy*, nó thêm một số ngẫu nhiên Gaussian vào mỗi token đầu vào. Mô hình cấp cao nhất cung cấp tín hiệu sự kiện đến từ actor *PoissonClock*. Actor đó tạo ra các *sự kiện* vào các thời điểm ngẫu nhiên theo quy trình Poisson. Một thực thi mẫu của mô hình này trong đó actor *Signal Source* cung cấp kết quả sóng hình sin trong biểu đồ được hiển thị trong Hình 14.

Trong quy trình Poisson, thời gian giữa 2 sự kiện được cung cấp độc lập và cung cấp biến ngẫu nhiên tương tự nhau với sự phân bố số mũ

3.1 The Structure of Modal Models

Mô hình chung của một Mô hình bộ nhớ được thể hiện trong Hình 15. Hành vi của một Mô hình Bộ nhớ được điều chỉnh bởi một máy trạng thái, trong đó mỗi trạng thái được gọi là một **chế độ**. Trong Hình 15, mỗi chế độ được biểu thị bằng một bong bóng, giống như một trạng thái trong máy trạng thái, nhưng được tô màu xanh lam nhạt để gợi ý rằng đó là một chế độ chứ không phải trạng thái bình thường. Một chế độ, không giống như một trạng thái thông thường, có một **tinh chỉnh chế độ**, là một actor tổng hợp nền xác định hành vi khi chế độ đang hoạt động. Ví dụ trong Hình 13 cho thấy hai tinh chỉnh, mỗi tinh chỉnh là một mô hình SDF biến đổi các token đầu vào để tạo ra các token đầu ra.

Lưu ý rằng điều cần thiết là tinh chỉnh phải chứa chỉ dẫn, và chỉ dẫn được chứa có thể sử dụng được với chỉ dẫn chi phối việc thực hiện actor Mô hình Bộ nhớ. Ví dụ trong Hình 13 có một chỉ dẫn SDF bên trong mỗi chế độ và một chỉ dẫn DE bên ngoài Mô hình Bộ nhớ. SDF thường có thể được sử dụng bên trong DE, vì vậy sự kết hợp này là hợp lệ.

Giống như các trạng thái trong máy trạng thái thông thường, các chế độ được nối với nhau bằng các cung thể hiện các **chuyển tiếp**. Mỗi quá trình chuyển đổi có một **bộ bảo vệ**, là một vị từ (một biểu thức có giá trị boolean) chỉ định thời điểm thực hiện quá trình chuyển đổi.

Example 8: Trong Hình 13, quá trình chuyển đổi được bảo vệ bởi biểu thức *event-isPresent*, đánh giá là đúng khi input *sự kiện* có sự kiện. Do cổng đầu vào đó được kết nối với actor *PoissonClock*, nên quá trình chuyển đổi sẽ được thực hiện theo các thời điểm ngẫu nhiên, với hàm

phụ biến ngẫu nhiên cho biết khoảng thời gian giữa các lần chuyển đổi.

Một biến thể của mẫu trong Hình 15 được hiển thị trong Hình 16, trong đó hai chế độ có cùng cách tinh chỉnh. Điều này có ích khi hành vi trong các chế độ khác nhau chỉ khác nhau bởi các giá trị tham số. Để xây dựng một mô hình trong đó nhiều chế độ có cùng cách tinh chỉnh, hãy thêm một phần tinh chỉnh vào một trong các trạng thái, đặt tên cho phần tinh chỉnh (theo mặc định, tên gợi ý cho phần tinh chỉnh giống với tên của trạng thái, nhưng người dùng có thể chọn bất kỳ tên nào để tinh chỉnh). Sau đó ở trạng thái khác, thay vì chọn [Add Refinement], hãy chọn [Configure] (hoặc chỉ cần nhấp đúp vào trạng thái) và chỉ định tên tinh chỉnh làm giá trị cho tham số *refinementName*. Cả hai chế độ bây giờ sẽ có cùng một tinh chỉnh.

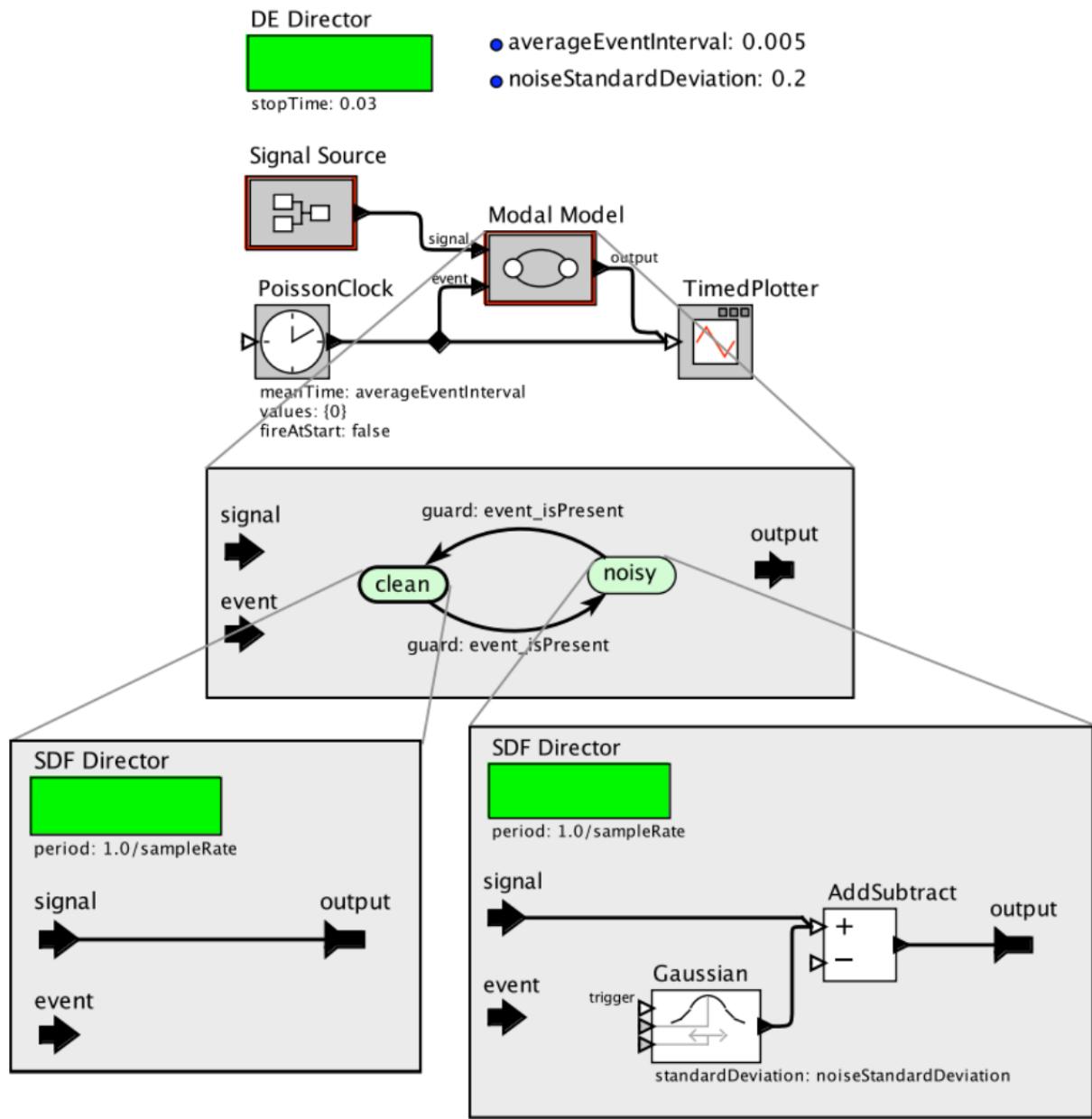


Figure 13: Simple modal model that has a normal (clean) operating mode, in which it passes inputs to the output unchanged, and a faulty mode, in which it adds Gaussian noise. It switches between these modes at random times determined by the PoissonClock actor.

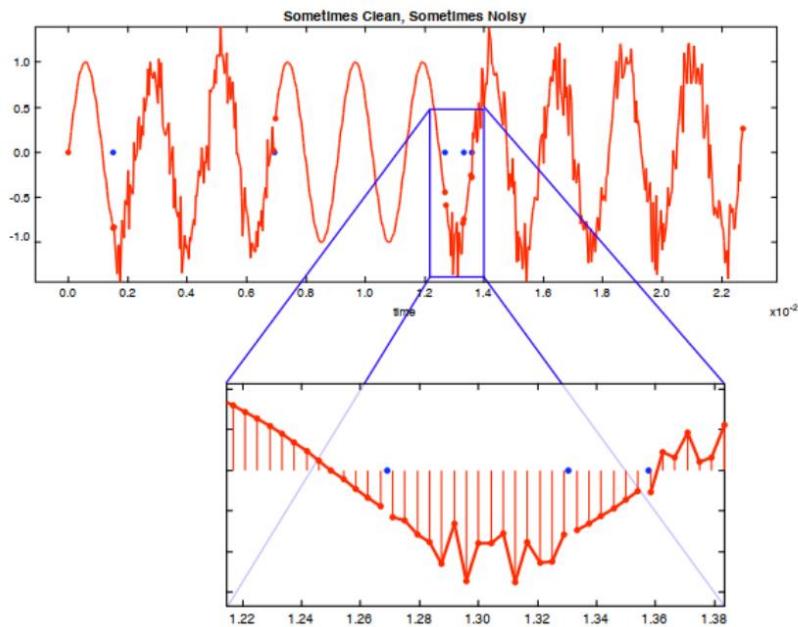


Figure 14: Plot generated by the model in figure 13.

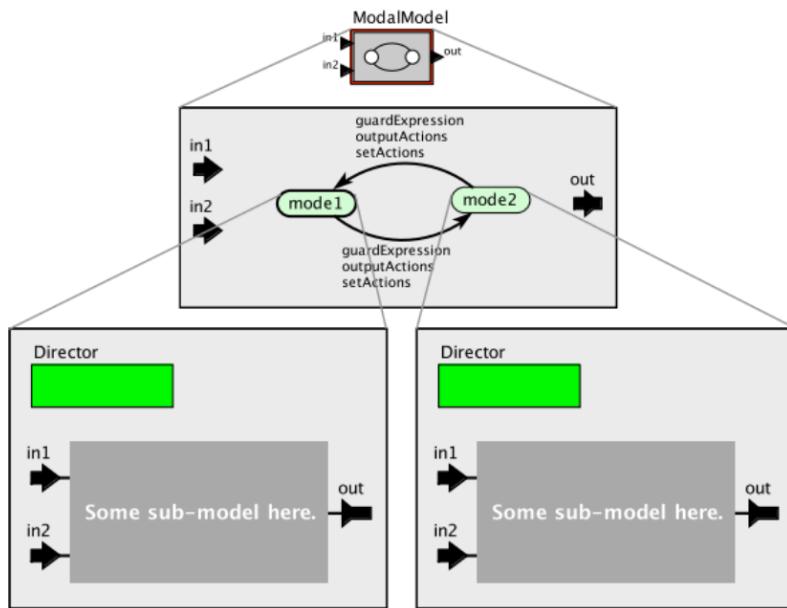


Figure 15: General pattern of a modal model with two modes, each with its own refinement.

Một biến thể khác của mẫu là khi một chế độ có nhiều tinh chỉnh. Điều này có thể được thực hiện bằng cách [Add Refinement] nhiều lần hoặc bằng cách chỉ định danh sách các tên tinh chỉnh lại được phân tách bằng dấu phẩy cho tham số *refinementName*. Những sàng lọc này sẽ thực hiện theo thứ tự mà chúng được thêm vào.

3.2 Hierarchical FSMs

Một dạng Mô hình Bộ nhớ đặc biệt hữu ích là **FSM phân cấp**. Đây là một Mô hình Bộ nhớ trong đó bản thân các tinh chỉnh trạng thái là các máy trạng thái.

Example 9: Một FSM phân cấp kết hợp bộ điều nhiệt bình thường và bộ điều nhiệt bị lỗi của Ví dụ 2 và 6 được hiển thị trong Hình 17. Trong mô hình này, actor *Bernoulli* được sử dụng để tạo tín hiệu lỗi (điều này sẽ đúng với xác suất 0,01). Khi tín hiệu *lỗi* là *đúng*, Mô hình Bộ nhớ sẽ chuyển sang trạng thái bị lỗi và duy trì ở đó trong 10 lần lặp lại trước khi trở về chế độ *bình thường*. Các tinh chỉnh trạng thái giống như trong Hình 9 và 10, cho thấy hoạt động bình thường và bị lỗi của bộ điều chỉnh nhiệt.

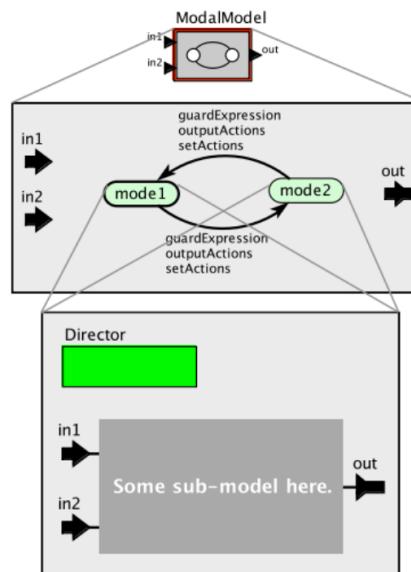


Figure 16: Variant of the pattern in Figure 15 where two modes share the same refinement.

Để tạo FSM phân cấp, khi bạn chọn [Add Refinement] trong menu của một trạng thái trong máy trạng thái của bạn, thay vì chọn [Default Refinement], bạn nên chọn [State Machine Refinement]. Máy trạng thái bên trong có thể tham chiếu các cổng đầu vào và ghi vào các cổng đầu ra, và bản thân các trạng thái của nó có thể có các tinh chỉnh (Tinh chỉnh mặc định hoặc Tinh chỉnh máy trạng thái).

Lưu ý rằng mô hình trong Hình 17 kết hợp **máy trạng thái ngẫu nhiên** với FSM không xác định. Máy trạng thái ngẫu nhiên có hành vi ngẫu nhiên, nhưng một mô hình xác suất rõ ràng được cung cấp dưới dạng actor *Bernoulli*. FSM không xác định cũng có hành vi ngẫu nhiên, nhưng không có mô hình xác suất nào được cung cấp.

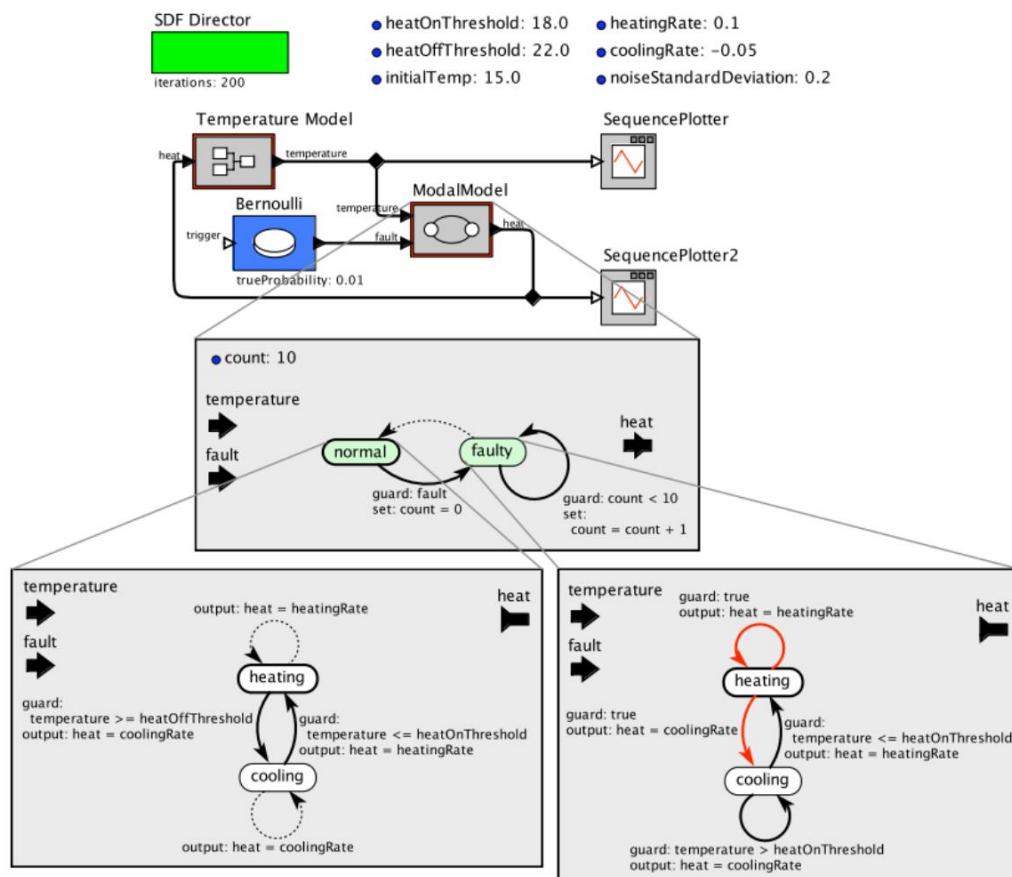


Figure 17: A hierarchical FSM that combines the normal and faulty thermostats of Examples 2 and 6.

3.3 Preemptive Transitions

Quá trình chuyển đổi ưu tiên là quá trình chuyển đổi có thể ngăn cản việc thực hiện tinh chỉnh trạng thái hiện tại. Một quá trình chuyển đổi được ưu tiên nếu tham số ưu tiên của nó đánh giá là đúng. Cú pháp trong Vergil như đã cho thấy trong Hình 19, một vòng tròn màu đỏ ở gốc của quá trình chuyển đổi. Trong hình đó, nếu trạng thái hiện tại là *state1* và biểu thức *guard1* đánh giá là *true*, thì việc tinh chỉnh *state1* sẽ không được thực thi. Hơn nữa, nếu *guard1* đánh giá là *true*, thì *guard2* sẽ không được đánh giá.

Further Reading: Concurrent Composition of State Machines

Máy trạng thái có một lịch sử lâu dài và nổi bật trong lý thuyết tính toán ([Hopcroft và Ullman, 1979](#)). Thành phần đồng thời của các máy trạng thái là một lĩnh vực nghiên cứu gần đây hơn, và tiếp tục trải qua sự thay đổi đáng kể. Một mô hình ban sơ cho thành phần đồng thời như vậy là **Statecharts**, do David Harel ([Harel, 1987](#)). Với Statecharts, Harel đã đưa ra khái niệm về **và trạng thái**, trong đó máy trạng thái có thể ở cả hai trạng thái *A* và *B* cùng một lúc. Khi kiểm tra cẩn thận, mô hình Statecharts là sự kết hợp đồng thời của các FSM phân cấp theo mô hình tính toán SR. Do đó, các biểu đồ trạng thái (gần như) tương đương với các Mô hình bộ nhớ kết hợp các FSM phân cấp và chỉ dẫn SR trong Ptolemy II ([Eker et al., 2003](#)). Statecharts đã được thực tiễn hóa trong một công cụ phần mềm được gọi là **Statemate** ([Harel et al., 1990](#)).

Công việc của Harel đã kích hoạt một loạt hoạt động, dẫn đến nhiều biến thể của mô hình ([Beeck, 1994](#)). Cú pháp trực quan của Harel sau đó đã được thông qua để trở thành một phần của **UML, ngôn ngữ mô hình hóa thống nhất** ([Booch et al., 1998](#)). Một phiên bản đặc biệt thanh lịch là **SynchCharts** ([Andre, 1996](#)), cung cấp cú pháp trực quan cho ngôn ngữ đồng bộ Esterel ([Berry and Gonthier, 1992](#)).

Một trong những tính chất chính của thành phần đồng bộ của các máy trạng thái là có thể mô hình hóa một thành phần của các thành phần như chính nó là một máy trạng thái. Một cơ chế đơn giản để thực hiện điều này dẫn đến một máy trạng thái có không gian trạng thái là tích chéo của các máy riêng lẻ. Các cơ chế phức tạp hơn đã được phát triển, chẳng hạn như máy tự động hóa giao diện ([de Alfaro and Henzinger, 2001](#)).

Hybrid systems cũng có thể được xem như các Mô hình Bộ nhớ, trong đó mô hình đồng thời là mô hình thời gian liên tục ([Maler và cộng sự, 1992; Henzinger, 2000; Lynch và cộng sự, 1996](#)). Theo công thức thông thường, các hệ thống lai kết hợp các FSM với các phương trình vi phân thông thường (ODEs), trong đó mỗi trạng thái của các FSM được liên kết với một cấu hình cụ thể của các ODE. Nhiều công cụ phần mềm đã được phát triển để xác định, mô phỏng và phân tích các hệ thống lai ([Carloni et al., 2006](#)).

Girault et al. (1999) đã chỉ ra rằng, trên thực tế, các FSM có thể được kết hợp theo thứ bậc với nhiều mô hình tính toán đồng thời phong phú. Họ gọi các thành phần như vậy là ***charts** hoặc **starCharts**, trong đó ngôi sao biểu trưng cho một ký tự đại diện. Một số dự án nghiên cứu hiện tại tiếp tục khám phá các biến thể đa dạng của các máy trạng thái. BIP (Basu et al., 2006), ví dụ, tổng hợp các máy trạng thái bằng cách sử dụng các tương tác điểm hẹn. Alur et al. (1999) đưa ra một nghiên cứu rất hay về các câu hỏi ngữ nghĩa xung quanh các FSM, bao gồm các câu hỏi phức tạp khác nhau. Prochnow và von Hanxleden (2007) mô tả các kỹ thuật phức tạp để chỉnh sửa trực quan các máy trạng thái đồng thời.

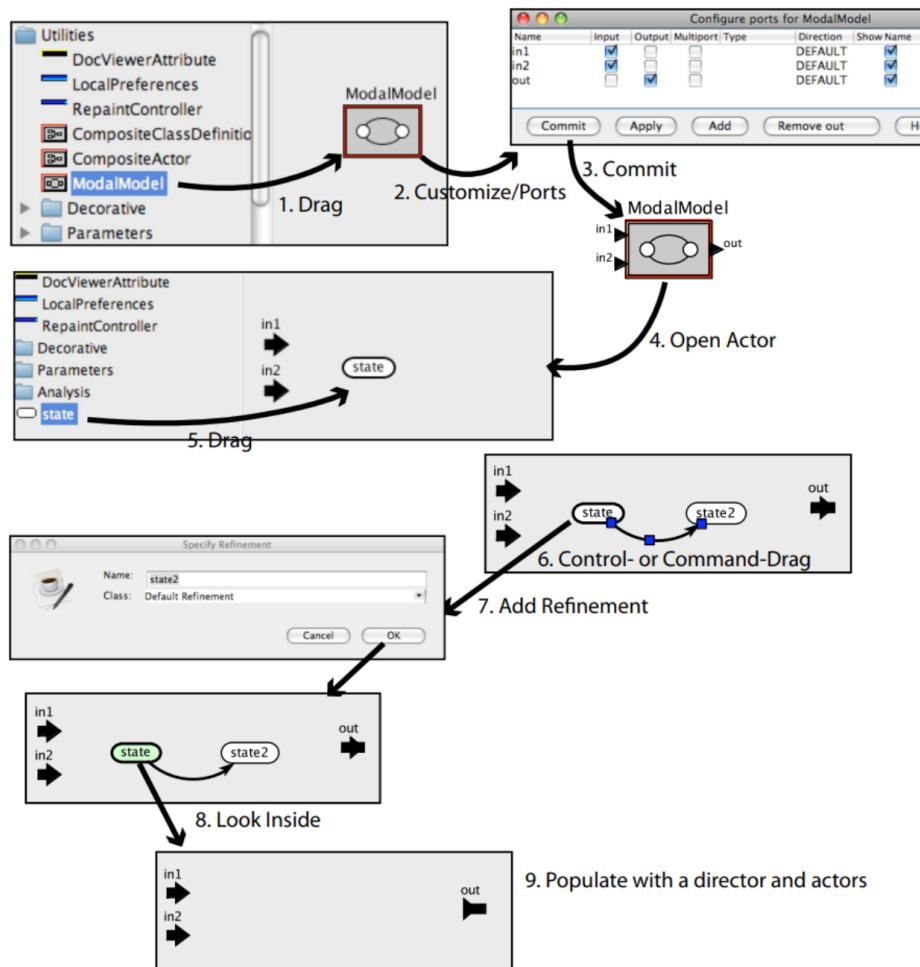


Figure 18: How to create modal models.

Mechanics: Creating Modal Models

Việc tạo ra một Mô hình Bộ nhớ được minh họa trong Hình 18. Quá trình này bắt đầu bằng cách kéo vào một *ModalModel* từ thư viện *Utilities* và điền vào đó các cổng. Sau đó, mở Mô hình Bộ nhớ và điền vào đó một hoặc nhiều trạng thái và chuyển tiếp. Để tạo hiệu ứng chuyển tiếp, hãy giữ phím Control (hoặc phím Command trên máy Mac) và nhấp và kéo từ trạng thái này sang trạng thái khác. Để thêm tinh chỉnh, nhấp chuột phải vào trạng thái và chọn [*Add Refinement*]. Bạn có thể chọn [*Default Refinement*] hoặc [*State Machine Refinement*]. Cái trước sẽ yêu cầu một chỉ dẫn và các actor xử lý dữ liệu đầu vào để tạo ra kết quả đầu ra. Cái sau sẽ cho phép tạo ra một hệ thống phân cấp FSM.

Probing Further: Internal Structure of a Modal Model

Trong Ptolemy II, mọi đối tượng (actor, trạng thái, quá trình chuyển đổi, cổng, tham số, v.v.) có thể có nhiều nhất một vùng chứa. Trong một Mô hình Bộ nhớ, hai trạng thái có thể chia sẻ cùng một tinh chỉnh. Điều này dường như vi phạm nguyên tắc “tối đa một container.”

Tuy nhiên, việc thực hiện trong Ptolemy II không vi phạm nguyên tắc này. Một actor *ModalModel* thực sự là một actor tổng hợp chuyên biệt có chứa một thể hiện *FSMDirector*, một *FSMActor* và bất kỳ số lượng tác nhân tổng hợp nào. Mỗi actor tổng hợp là một ứng cử viên để tinh chỉnh cho bất kỳ trạng thái nào của *FSMActor*. *FSMActor* là bộ điều khiển, trong đó nó xác định chế độ nào đang hoạt động bất cứ lúc nào. *FSMDirector* đảm bảo rằng dữ liệu đầu vào được gửi đến *FSMActor* và tất cả các chế độ đang hoạt động.

Tuy nhiên, giao diện người dùng Vergil ẩn cấu trúc này. Khi bạn thực hiện một lệnh [*Open Actor*] trên một *ModalModel*, giao diện người dùng thực sự bỏ qua một mức của hệ thống phân cấp và đưa bạn trực tiếp đến bộ điều khiển *FSMActor*. Nó không hiển thị lớp cấu trúc phân cấp chứa *FSMActor*, *FSMDirector* và các tinh chỉnh. Hơn nữa, khi bạn [*Look Inside*] một trạng thái, giao diện người dùng sẽ tăng lên một cấp của hệ thống phân cấp và mở ra *tất cả* các tinh chỉnh của trạng thái đã chọn. Kiến trúc hơi phức tạp này cân bằng tính đa dạng với sự tiện lợi của người dùng.

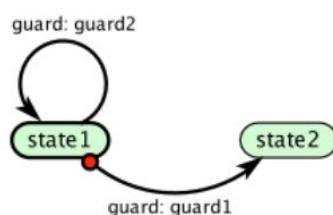


Figure 19: A preemptive transition, if chosen, results in the refinement not being executed. It is indicated by the red circle at the start of the transition.

3.4 Reset Transitions

Quá trình chuyển đổi đặt lại, nếu được chọn, dẫn đến việc tinh chỉnh trạng thái *đích* được đặt lại về điều kiện ban đầu. Nó được biểu thị bằng đầu mũi tên mở ở cuối quá trình chuyển đổi, như thể hiện trong Hình 20. Cụ thể, sau khi quá trình chuyển đổi được thực hiện, phương thức *initialize()* của các tinh chỉnh của trạng thái đích sẽ được gọi. Ví dụ, nếu trạng thái đích có FSM tinh chỉnh, thì trạng thái của FSM tinh chỉnh đó sẽ được đặt thành trạng thái ban đầu.

Quá trình chuyển đổi đặt lại có thể được sử dụng để khởi động lại FSM sau khi nó đạt đến trạng thái cuối cùng, như được minh họa trong ví dụ sau.

Example 10: Chúng ta có thể sử dụng trạng thái cuối cùng để tạm thời dừng thực thi mô hình con và chuyển đổi đặt lại để khởi động lại nó. Trong Hình 21, *ModalModel* chỉ có một trạng thái duy nhất và quá trình đặt lại được thực hiện bất cứ khi nào đầu vào là bội số của 10. Khi quá trình chuyển đổi này được thực hiện, quá trình tinh chỉnh sẽ được khởi tạo. Tinh chỉnh là một FSM mở rộng có 5 đầu vào, mỗi lần sao chép chúng không đổi sang đầu ra. Khi 5 đầu vào đã đến, tinh chỉnh chuyển sang trạng thái cuối cùng và dừng thực thi. Các lần lặp lại tiếp theo sẽ không tạo ra đầu ra nào (đầu ra của *ModalModel* sẽ không có). Khi quá trình tinh chỉnh được khởi tạo lại, thì nó sẽ bắt đầu lại, sao chép năm đầu vào tiếp theo sang đầu ra rồi dừng lại. Đối với mô hình này, giả sử actor *Ramp* tạo chuỗi 1, 2, 3,..., thì đầu ra sẽ là 1, 2, 3, 4, 5, tiếp theo là 5 lần null, tiếp theo là 11, 12, 13, 14, 15, tiếp theo là 5 lần null khác, v.v. Chỉ dẫn SR được sử dụng ở đây để làm rõ các lần lặp và giá trị null.

Việc sử dụng các trạng thái cuối cùng này đôi khi được gọi trong tài liệu là **sự kết thúc bình thường**. Mô hình con ngừng thực thi khi nó đi vào trạng thái cuối cùng và có thể được khởi động lại bằng cách đặt lại.



Figure 20: A reset transition, if chosen, results in the refinement of the destination state being reset to its initial condition. It is indicated by the open arrowhead at the end of the transition.

3.5 Transition Refinements

Một quá trình chuyển đổi cũng có thể có những tinh chỉnh. Để tạo một **tinh chỉnh chuyển tiếp**, hãy nhấp chuột phải vào vùng chuyển tiếp và chọn [Add Refinement]. Cú pháp được hiển thị trong Hình 22. Cụ thể, quá trình chuyển đổi có phần tinh chỉnh được hiển thị bằng đường kẻ đậm hơn so với quá trình chuyển đổi không có phần tinh chỉnh. Tinh chỉnh quá trình chuyển đổi kích hoạt khi quá trình chuyển đổi được chọn và kích hoạt sau khi quá trình chuyển đổi được thực hiện. Khi nó kích hoạt, nó có thể cung cấp đầu ra.

Thật thú vị, một tinh chỉnh chuyển tiếp cũng có thể lấy làm đầu vào các đầu ra của sự tinh chỉnh trạng thái mà từ đó quá trình chuyển đổi bắt đầu. Cách thức hoạt động này là một cổng đầu ra (được đặt tên, là *out*, như trong hình) có một cổng chị em với “*_in*” được thêm vào tên (*out_in* trong hình). Cổng chị em đó cung cấp bất kỳ dữ liệu nào mà các tinh chỉnh trạng thái được tạo ra trên cổng đầu ra đó trước khi chuyển đổi được chọn.

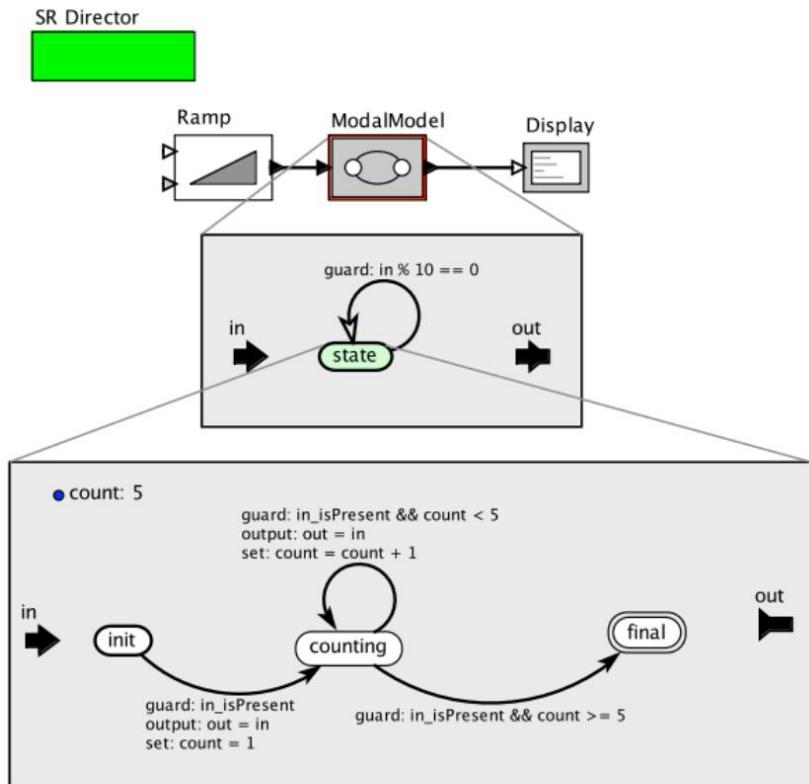


Figure 21: A reset transition may be used to restart an FSM after it has reached a final state.

3.6 Execution Policy for Modal Models

Việc thực thi một *ModalModel* tương tự như việc thực hiện một *FSMActor* được mô tả trong Phần 2.2. Tóm lại, trong phương thức *fire()*, actor *ModalModel*

1. đọc đầu vào;
2. đánh giá những bộ bảo vệ chuyển đổi ưu tiên ra khỏi trạng thái hiện tại;
3. nếu không có chuyển đổi ưu tiên nào được bật, actor
 1. kích hoạt các tinh chỉnh của trạng thái hiện tại (nếu có); và
 2. đánh giá các bảo vệ đối với các chuyển đổi không ưu tiên ra khỏi trạng thái hiện tại;
3. chọn một quá trình chuyển đổi mà bộ bảo vệ đánh giá là đúng, ưu tiên cho các chuyển đổi ưu tiên;
4. thực hiện các hành động đầu ra của quá trình chuyển đổi đã chọn; và
5. kích hoạt các tinh chỉnh chuyển tiếp của quá trình chuyển đổi đã chọn.

Trong *postfire()*, actor *ModalModel*

1. kích hoạt lại các tinh chỉnh của trạng thái hiện tại nếu chúng đã kích hoạt;
2. thực hiện các hành động đã đặt của quá trình chuyển đổi đã chọn;
3. kích hoạt sau các tinh chỉnh chuyển đổi của quá trình chuyển đổi đã chọn;
4. thay đổi trạng thái hiện tại thành đích của quá trình chuyển đổi đã chọn; và
5. khởi tạo các tinh chỉnh của trạng thái đích nếu quá trình chuyển đổi là quá trình đặt lại.

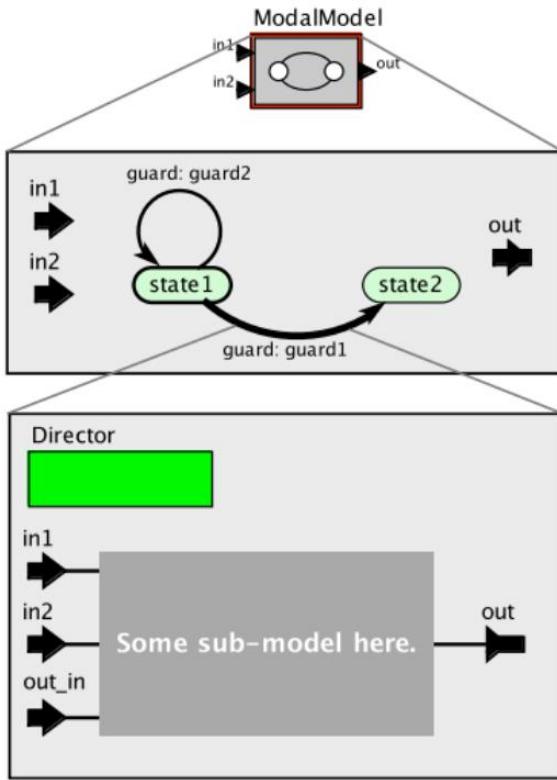


Figure 22: A transition refinement is a sub-model that fires when the transition is chosen and postfires when the transition is actually committed in the `postfire()` method of the container.

Actor *ModalModel* không thực hiện các thay đổi trạng thái liên tục trong phương thức `fire()` của nó, do đó, miễn là điều tương tự cũng đúng với các chỉ dẫn và actor tinh chỉnh, một Mô hình Bộ nhớ có thể được sử dụng trong bất kỳ miền nào. Tuy nhiên, cách nó hoạt động trong mỗi miền có thể hơi tinh tế, đặc biệt với các miền có ngữ nghĩa điểm cố định và khi sử dụng các chuyển tiếp không xác định. Hành vi này chính xác như được mô tả ở trên đối với *FSMActor*, với ngoại lệ duy nhất là các tinh chỉnh trạng thái và quá trình chuyển đổi được kích hoạt và kích hoạt sau vào những thời điểm thích hợp trong quá trình thực thi. Một điều tinh tế là nếu một quá trình chuyển đổi ưu tiên được kích hoạt, thì những bộ bảo vệ các quá trình chuyển đổi không ưu tiên sẽ không được đánh giá. Do đó, tính không xác định không bao giờ xuất phát từ việc những bộ bảo vệ chuyển đổi ưu tiên và không ưu tiên đều trở thành sự thật trong một lần lặp.

Lưu ý rằng các tinh chỉnh trạng thái được kích hoạt trước khi đánh giá bất kỳ biện pháp bảo vệ không phòng ngừa nào. Một hậu quả của điều này là các kết quả đầu ra từ việc kích hoạt tinh chỉnh có thể được tham chiếu trong các bộ bảo vệ của các chuyển đổi không được ưu tiên! Do đó, việc quá trình chuyển đổi có được thực hiện hay không có thể phụ thuộc vào cách phản ứng của quá trình tinh chỉnh hiện tại với các yếu tố đầu vào. Trên thực tế, người đọc tinh ý có thể đã nhận thấy trong các hình ở đây rằng khi bộ điều khiển *FSMActor* của *ModalModel* được hiển thị, biểu tượng cho các cổng đầu ra trông không giống một cổng đầu ra bình thường (xem ví dụ cổng nhiệt ở giữa sơ đồ hình 17). Trên thực tế, biểu tượng này là biểu tượng cho một cổng vừa là đầu vào vừa là đầu ra. Trên thực tế, nó phục vụ cả hai vai trò này cho bộ điều khiển mô hình phương thức, vì nó có thể ảnh hưởng đến việc đánh giá các bộ bảo vệ và nó có thể cung cấp đầu ra cho môi trường.

Trong một lần kích hoạt, có thể việc tinh chỉnh trạng thái hiện tại tạo ra một đầu ra và quá trình chuyển đổi được thực hiện cũng tạo ra một đầu ra trên cùng một cổng. Trong trường hợp này, chỉ đầu ra thứ hai trong số này sẽ xuất hiện trên đầu ra của *ModalModel*. Tuy nhiên, giá trị đầu ra đầu tiên trong số này, giá trị do tinh chỉnh tạo ra, có thể ảnh hưởng đến việc quá trình chuyển đổi có được thực hiện hay không. Đó là, nó có thể ảnh hưởng đến bộ bảo vệ. Ngoài ra, nếu một quá trình tinh chỉnh chuyển tiếp ghi vào đầu ra, thì giá trị đó sẽ được tạo ra, ghi đè giá trị được tạo ra bởi quá trình tinh chỉnh trạng thái hoặc hành động đầu ra trên quá trình chuyển đổi

3.7 Time and Modal Models

Nhiều chỉ dẫn Ptolemy II thực hiện một mô hình tính toán theo thời gian. Bản thân *ModalModel* và *FSMActor* không có thời gian, nhưng chúng có một số tính năng nhất định để hỗ trợ việc sử dụng chúng trong các miền có thời gian.

Các FSM mà chúng tôi đã mô tả cho đến nay là **phản ứng**, nghĩa là chúng chỉ tạo ra đầu ra tương ứng với đầu vào. Trong một miền thời gian, đầu vào có dấu thời gian. Đối với một FSM phản ứng, dấu thời gian của đầu ra giống như dấu thời gian của đầu vào. Do đó, FSM dường như đang phản ứng trong thời gian không. Đó là tức thời, từ góc nhìn của miền thời gian.

Tuy nhiên, trong miền thời gian, cũng có thể xác định các FSM tự phát. **FSM tự phát** hoặc **mô hình phương thức tự phát** là mô hình tạo ra đầu ra ngay cả khi không có đầu vào.

Example 11: Mô hình trong Hình 23 chuyển đổi giữa hai chế độ cứ sau 2,5 đơn vị thời gian. Ở chế độ thông thường, nó tạo tín hiệu đồng hồ có khoảng cách đều đặn với chu kỳ 1.0 (và với giá trị 1, giá trị đầu ra mặc định cho *DiscreteClock*). Ở chế độ bất thường, nó tạo ra các sự kiện có khoảng cách ngẫu nhiên bằng cách sử dụng tác nhân *PoissonClock* với thời gian trung bình giữa các sự kiện được đặt thành 1 và giá trị được đặt thành 2. Kết quả của một lần chạy điển hình được vẽ trong Hình 24, với nền bóng mờ hiển thị thời gian trên đó nó ở trong hai chế độ. Các sự kiện đầu ra từ *ModalModel* là tự phát ở chỗ chúng không nhất thiết được tạo ra để phản ứng với các sự kiện đầu vào.

Ví dụ này minh họa một số điểm tinh tế về thời gian. Xem xét biểu đồ trong Hình 24, chúng ta thấy rằng một sự kiện có giá trị 1 và một sự kiện khác có giá trị 2 được tạo ra tại thời điểm 0. Tại sao? Trạng thái ban đầu là *bình thường* và chính sách thực thi được mô tả trong phần 3.6 giải thích rằng việc tinh chỉnh trạng thái ban đầu đó được kích hoạt trước khi các bộ bảo vệ được đánh giá. Việc kích hoạt đó tạo ra đầu ra đầu tiên của *DiscreteClock*. Thay vào đó, nếu chúng ta sử dụng chuyển đổi ưu tiên, như trong Hình 25, thì sự kiện đầu ra đầu tiên đó sẽ không xuất hiện

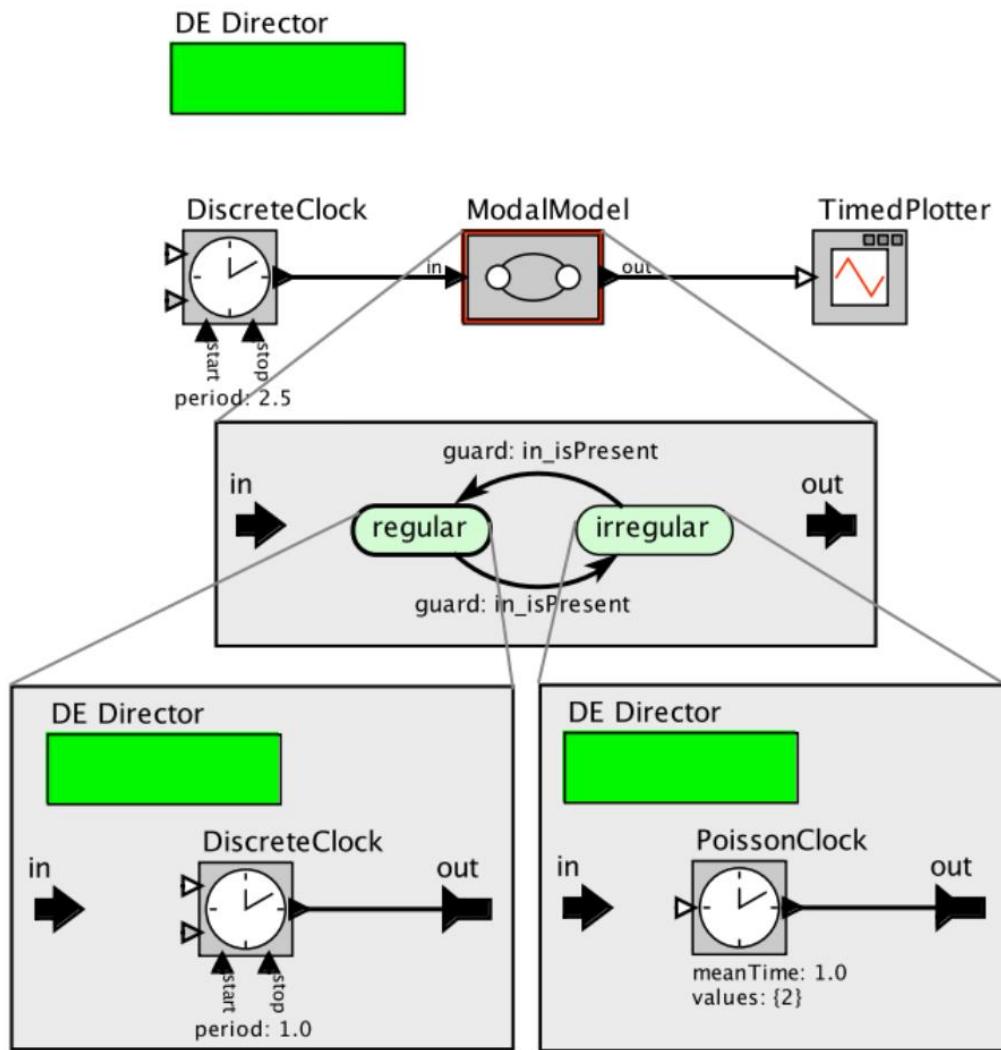


Figure 23: A spontaneous modal model, which produces output events that are not triggered by input events.

Sự kiện thứ hai trong Hình 24 (với giá trị 2) tại thời điểm 0 được tạo ra bởi vì *PoissonClock*, theo mặc định, tạo ra một sự kiện tại thời điểm bắt đầu thực thi, thời điểm 0. Sự kiện này được tạo ra trong lần lặp lại thứ hai của *ModalModel*, sau khi chuyển sang trạng thái *bất thường*. Mặc dù sự kiện có cùng mốc thời gian với sự kiện đầu tiên (cả hai đều xảy ra tại thời điểm 0), nhưng chúng có thứ tự được xác định rõ ràng. Sự kiện có giá trị 1 xuất hiện trước sự kiện có giá trị 2. Trong Ptolemy II, giá trị của thời gian thực sự được biểu thị bằng một cặp số được gọi là thẻ, $(t, n) \in \mathbb{R} \times \mathbb{N}$, chứ không phải là một số đơn lẻ. Số đầu tiên

trong số này, t , được gọi là dấu thời gian. Nó xấp xỉ một số thực (nó là một số thực được lượng tử hóa với độ chính xác được chỉ định). Chúng tôi giải thích dấu thời gian t để biểu thị số giây (hoặc bất kỳ đơn vị thời gian nào khác) kể từ khi bắt đầu thực hiện mô hình. Số thứ hai trong số này, n , được gọi là chỉ số thời gian hoặc vi bước và nó đại diện cho một số thứ tự cho các sự kiện xảy ra cùng một mốc thời gian. Trong ví dụ của chúng tôi, sự kiện đầu tiên (có giá trị 1) có thẻ $(0; 0)$ và sự kiện thứ hai (có giá trị 2) có thẻ $(0; 1)$. Nếu chúng ta đã đặt tham số *fireAtStart* của tác nhân *PoissonClock* thành *false*, thì sự kiện thứ hai sẽ không xảy ra.

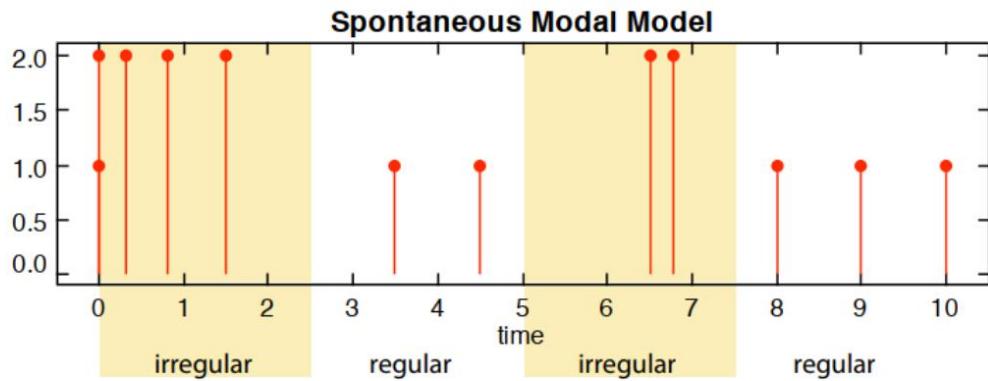


Figure 24: A plot of the output from one run of the model in Figure 23.

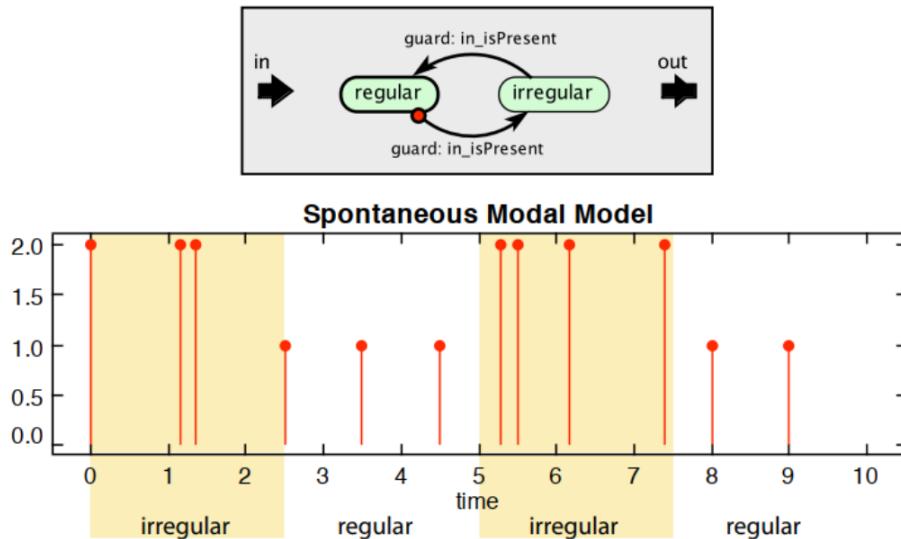


Figure 25: A variant of Figure 23 where a preemptive transition prevents the initial firing of the DiscreteClock.

Lưu ý thêm rằng actor *DiscreteClock* trong tinh chỉnh chế độ *thông thường* có chu kỳ 1.0, nhưng tạo ra các sự kiện tại các thời điểm 0.0, 3.5 và 4.5, 8.0, 9.0, v.v.. Đây không phải là bội số của 1.0 từ thời điểm bắt đầu thực thi. Tại sao?

Phương thức bắt đầu ở chế độ *thông thường*, nhưng không dành thời gian ở đó. Nó ngay lập tức chuyển sang chế độ *bắt thường*. Do đó, tại thời điểm 0,0, chế độ *thông thường* sẽ không hoạt động. Trong khi nó không hoạt động, khái niệm thời gian cục bộ của nó không tiến lên. Nó hoạt động trở lại vào giờ toàn cầu 2,5, nhưng khái niệm thời gian địa phương của nó vẫn là 0,0. Do đó, nó phải đợi thêm một đơn vị thời gian nữa, cho đến thời điểm 3,5, để tạo đầu ra tiếp theo.

Khái niệm về giờ địa phương này rất quan trọng để hiểu các tính thời gian Mô hình Bộ nhớ. Rất đơn giản, giờ địa phương đứng yên trong khi chế độ không hoạt động. Các actor đẽ cập đến thời gian, chẳng hạn như *TimedPlotter* và *CurrentTime*, có một tham số *useLocalTime*, mặc định là *sai*. Do đó, theo mặc định, một máy vẽ sẽ luôn vẽ các sự kiện trên dòng thời gian toàn cầu. Tuy nhiên, nếu không có actor nào truy cập thời gian toàn cầu, thì một chế độ tinh chỉnh sẽ hoàn toàn không biết rằng nó đã từng bị tạm dừng. Nó không xuất hiện như thể thời gian đã trôi qua.

Một đặc tính thú vị khác của đầu ra của mô hình này là không có sự kiện nào được tạo ra tại thời điểm 5,0, khi chế độ *bắt thường* hoạt động trở lại. Điều này xuất phát từ cùng một nguyên tắc. Chế độ *bắt thường* trở nên không hoạt động tại thời điểm 2,5 và do đó, từ thời điểm 2,5 đến 5,0, khái niệm thời gian cục bộ của nó không tăng lên. Khi nó hoạt động trở lại vào thời điểm 5,0, nó sẽ tiếp tục chờ đúng thời điểm (giờ địa phương) để tạo đầu ra tiếp theo từ actor *PoissonClock*.

Nếu một sự kiện được mong muốn tại thời điểm 5,0, thì có thể sử dụng quá trình đặt lại, như thể hiện trong Hình 26. Phương thức *initialize()* của *PoissonClock* khiến một sự kiện đầu ra được tạo ra tại *thời điểm khởi tạo*.

Thú vị là, do thuộc tính không có bộ nhớ của quy trình Poisson, thời gian đến sự kiện tiếp theo sau khi hoạt động giống hệt về mặt thống kê với thời gian giữa các sự kiện của quy trình Poisson. Nhưng thực tế này ít liên quan đến ngữ nghĩa của các Mô hình Bộ nhớ.

3.8 Time Delays in Modal Models

Độ trễ thời gian tương tác với các Mô hình Bộ nhớ theo những cách thú vị, như được minh họa trong ví dụ tiếp theo.

Example 12: Hình 27 cho thấy một mô hình tạo ra một chuỗi đếm các sự kiện cách nhau một đơn vị thời gian, sau đó lần lượt trì hoãn các sự kiện theo một đơn vị thời gian và không trì hoãn chúng. Trong chế độ *delay*, actor *TimeDelay* áp đặt thời gian trễ của một đơn vị thời gian. Trong chế độ *noDelay*, đầu vào được gửi trực tiếp đến đầu ra mà không bị trễ. Kết quả của việc thực hiện mô hình này được hiển thị trong Hình 28. Lưu ý rằng giá trị 0 xuất hiện tại thời điểm 2. Tại sao?

Mô hình bắt đầu ở chế độ *delay*, do đó, chế độ đó nhận đầu vào đầu tiên có giá trị 0. Tuy nhiên, Mô hình Bộ nhớ chuyển ngay lập tức ra khỏi chế độ đó sang chế độ *noDelay*. Vì vậy, không có thời gian trôi qua. Chế độ *delay* sẽ hoạt động trở lại vào thời điểm 1, nhưng tại thời điểm đó, giờ địa phương của nó chưa tăng. Giờ cục bộ vẫn là 0. Do đó, nó vẫn phải trì hoãn đầu vào có giá trị 0 một đơn vị thời gian, do đó, nó tạo ra đầu ra đó tại thời điểm 2, ngay trước khi chuyển ra ngoài một lần nữa sang chế độ *noDelay*.

3.9 Time in Transition Refinements

Một tinh chỉnh chuyển tiếp cũng có thể tham chiếu đến thời gian hiện tại, nhưng sẽ không có ý nghĩa gì khi có các tinh chỉnh chuyển tiếp không phản ứng. Giờ địa phương trong tinh chỉnh chuyển tiếp sẽ luôn giống với giờ địa phương trong môi trường mà Mô hình Bộ nhớ chứa quá trình chuyển đổi đang thực thi. Do đó, bất kỳ quyền truy cập nào vào thời gian cục bộ thông qua, chẳng hạn như actor *CurrentTime* sẽ mang lại thời gian chuyển đổi được thực hiện.

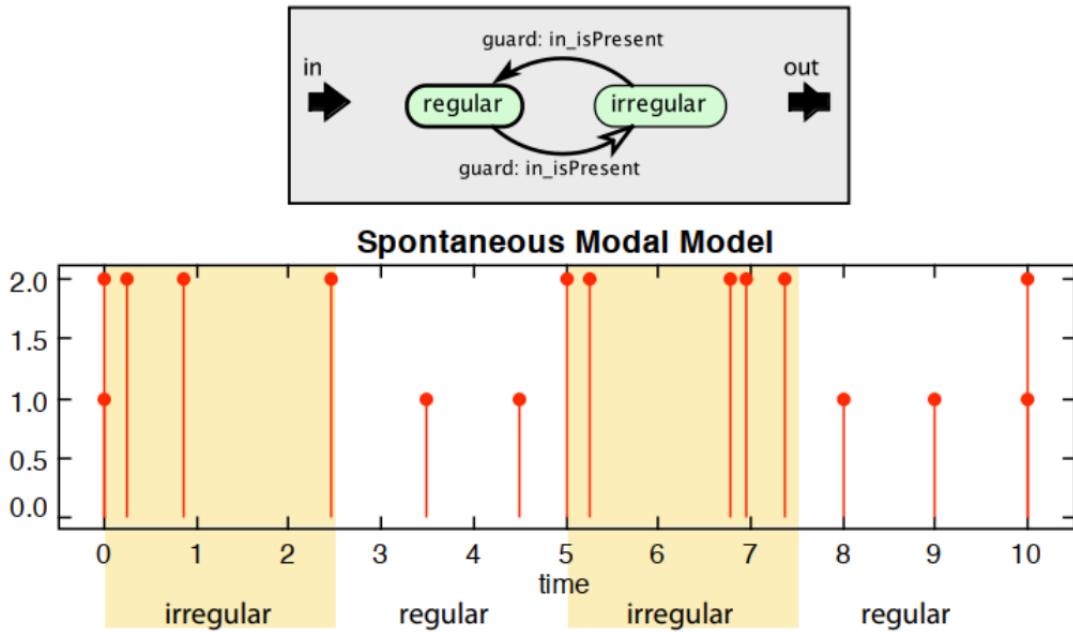


Figure 26: A variant of Figure 23 where a reset transition causes the PoissonClock to produce events when the `irregular` mode is reactivated.

Về mặt khái niệm, một tinh chỉnh chuyển đổi luôn hoạt động trong thời gian chính xác bằng không. Nó không mất thời gian để có một quá trình chuyển đổi. Kết quả là, sẽ không có ý nghĩa gì nhiều nếu đưa vào các bộ tạo sự kiện tự phát tinh chỉnh quá trình chuyển đổi như *DiscreteClock* hoặc *PoissonClock*. Việc bao gồm các actor trì hoãn thời gian cũng không có ý nghĩa gì.

3.10 Time and FSMs

Cũng có thể tạo một FSM đơn giản (không phải Mô hình Bộ nhớ) theo một nghĩa nào đó là tự phát, mặc dù nó sẽ không thể tạo ra kết quả đầu ra vào những thời điểm tùy ý. Nó có thể tạo đầu ra tại thời điểm khởi tạo và có thể tạo đầu ra với cùng dấu thời gian nhưng chỉ số cao hơn các sự kiện đầu vào.

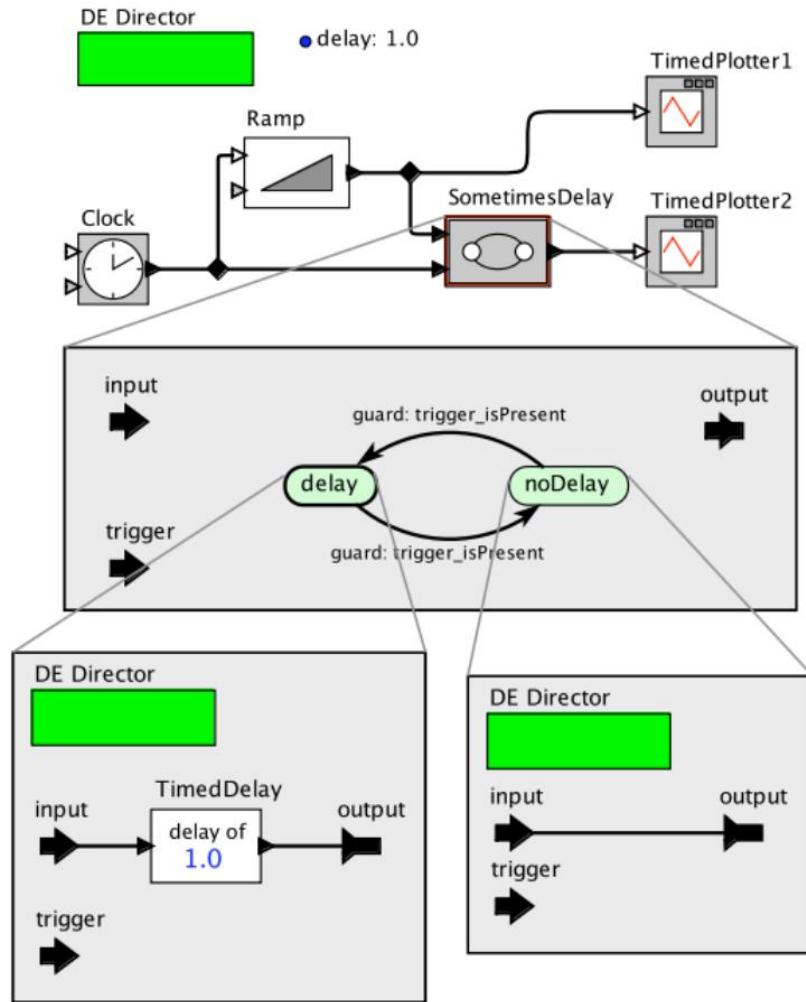


Figure 27: A modal model that switches between delaying the input by one time unit and not delaying it.

Example 13: Mô hình được hiển thị trong Hình 29 bao gồm một *FSMActor* vừa phản ứng vừa tự phát. Nó tạo ra đầu ra tại thời điểm 0 (với giá trị 0) mặc dù thực tế là nó không nhận được đầu vào nào tại thời điểm 0 (tham số *fireAtStart* của tác nhân *PoissonClock* là sai). Đầu ra này được tạo ra do quá trình chuyển đổi ra khỏi trạng thái *ban đầu* có bảo vệ *đúng* và do đó được thực hiện ngay khi khởi tạo máy trạng thái. Quá trình chuyển đổi đó có một hành động đầu ra, *out = 0*.

Sau khi khởi tạo, khi sự kiện đầu vào đầu tiên đến (gần thời gian 0,7, với giá trị 1), đầu vào cho phép chuyển từ trạng thái *chờ* sang trạng thái *trùng*

lặp. Quá trình chuyển đổi đó sao chép giá trị của đầu vào sang đầu ra và cũng ghi lại giá trị của đầu vào trong một biến cục bộ đã *recordedInput*. Khi máy trạng thái đã chuyển sang trạng thái *trùng lặp*, quá trình chuyển đổi trở lại trạng thái *chờ* sẽ được bật ngay lập tức, do đó, quá trình này được thực hiện trong bước vi mô tiếp theo. Do đó, máy trạng thái tạo ra đầu ra thứ hai cùng lúc (khoảng 0,7), như thể hiện trong biểu đồ. Đầu ra thứ hai có giá trị gấp đôi đầu ra thứ nhất, làm cho cả hai đầu ra hiển thị trong biểu đồ.

Trạng thái trong đó máy trạng thái không dành thời gian được gọi là **trạng thái nhất thời**. Trong ví dụ trước, cả trạng thái *ban đầu* và trạng thái *trùng lặp* đều tạm thời. Lưu ý rằng bất kỳ trạng thái nào có quá trình chuyển đổi mặc định (không có bộ bảo vệ hoặc có bộ bảo vệ đánh giá là đúng ngay lập tức) đều là trạng thái nhất thời, vì việc thoát khỏi trạng thái luôn được kích hoạt ngay lập tức sau khi vào trạng thái.

3.11 Modal Model Principles

Các mô hình phương thức rõ ràng là tinh tế và biểu cảm, đặc biệt đối với các mô hình tính thời gian. Thật hữu ích khi lùi lại và suy ngẫm về các nguyên tắc chi phối các lựa chọn thiết kế trong việc triển khai các Mô hình Bộ nhớ của Ptolemy II. Ý tưởng chính đằng sau một chế độ là nó chỉ định một phần của hệ thống chỉ hoạt động trong một phần thời gian. Khi nó không hoạt động, nó có ngừng tồn tại không? Thời gian có trôi không? Trạng thái của nó có thể tiến hóa không? Đây không phải là những câu hỏi dễ vì hành vi mong muốn phụ thuộc rất nhiều vào ứng dụng.

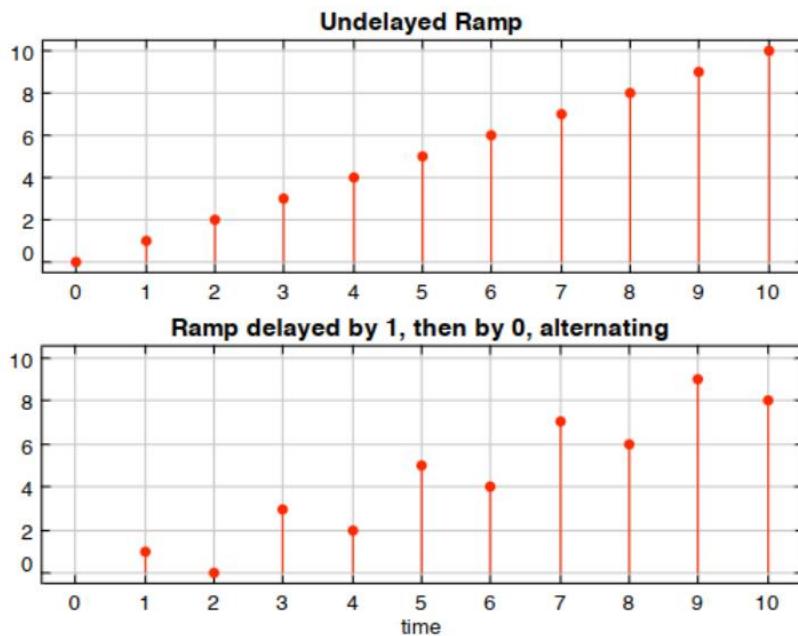


Figure 28: The result of executing the model in Figure 27.

Trong Ptolemy II, nguyên tắc chỉ đạo là khi một chế độ không hoạt động, thời gian địa phương đứng yên, nhưng thời gian toàn cầu trôi qua. Do đó, một chế độ không hoạt động ở trạng thái hoạt hình bị treo. Giờ địa phương trong một chế độ sẽ trễ thời gian trong môi trường của nó. Mức độ trễ bắt đầu từ 0 và tăng lên mỗi khi chế độ không hoạt động. Khi một sự kiện vượt qua ranh giới phân cấp vào hoặc ra khỏi chế độ, dấu thời gian của nó được điều chỉnh theo mức độ trễ. Do đó, trong chế độ, thời gian dường như không bị gián đoạn.

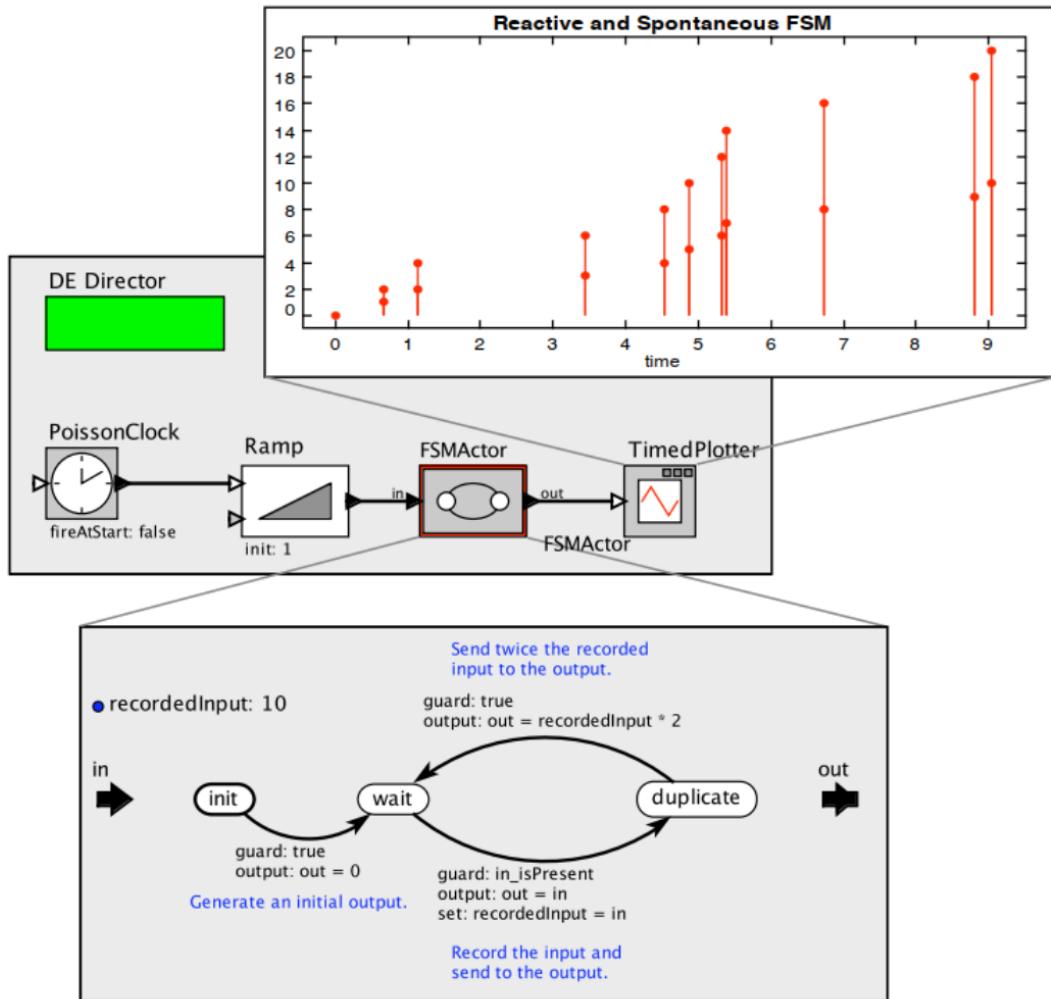


Figure 29: An FSM that spontaneously produces events at the start time and at the times of input events.

Điểm mấu chốt là không hoạt động không nhất thiết đồng nghĩa với việc không nhận được thông tin đầu vào và không bị quan sát. Điểm này được minh họa trong mô hình của Hình 30, thể hiện hai phiên bản của *DiscreteClock*, được gắn nhãn *DiscreteClock1* và *DiscreteClock2*, có cùng giá trị tham số. *DiscreteClock2* nằm trong một Mô hình Bộ nhớ có nhãn *ModalClock* và *DiscreteClock1* nằm ngoài bất kỳ Mô hình Bộ nhớ nào. Đầu ra của *DiscreteClock1* được lọc bởi một Mô hình Bộ nhớ có nhãn *ModalFilter* chuyển đầu vào đến đầu ra một cách có chọn lọc. Hai Mô hình Bộ nhớ được điều khiển bởi cùng một *ControlClock*, xác định thời điểm chúng chuyển đổi trạng

thái *hoạt động* và *không hoạt động*. Ba lô được hiển thị. Biểu đồ trên cùng chỉ đơn giản là đầu ra của *DiscreteClock1* không được sửa đổi theo bất kỳ cách nào. Biểu đồ ở giữa là kết quả của việc chuyển đổi giữa quan sát và không quan sát đầu ra của *DiscreteClock1*. Biểu đồ dưới cùng là kết quả của việc kích hoạt và hủy kích hoạt *DiscreteClock2*, nếu không thì nó giống hệt với *DiscreteClock1*.

Các actor *DiscreteClock* trong ví dụ này được thiết lập để tạo ra một chuỗi các giá trị, 1, 2, 3, 4, theo chu kỳ. Do đó, ngoài việc được tính thời gian, các actor này còn có trạng thái, vì chúng cần nhớ lại giá trị đầu ra cuối cùng để tạo ra giá trị đầu ra tiếp theo. Khi *DiscreteClock2* không hoạt động, trạng thái của nó không thay đổi và thời gian không tăng. Do đó, khi nó hoạt động trở lại, nó chỉ đơn giản là tiếp tục nơi nó đã dừng lại.

4 Kết luận

FSM và các Mô hình Bộ nhớ trong Ptolemy II cung cấp một cách rất rõ ràng để xây dựng các hành vi mô hình phức tạp. Do tính đa dạng này, cần phải thực hành để học cách sử dụng chúng tốt. Báo cáo này nhằm cung cấp một điểm khởi đầu hợp lý. Độc giả muốn tìm hiểu thêm được khuyến khích kiểm tra tài liệu về các lớp Java triển khai các cơ chế này. Nhiều trong số này có thể truy cập được khi chạy Vergil bằng cách nhấp chuột phải và chọn [*Documentation*]. Vui lòng gửi nhận xét tới eal@eecs.berkeley.edu.

Probing Further: Implementation of Transient States

Khi một quá trình chuyển đổi được thực hiện trong một FSM, *FSMActor* hoặc *ModalModel* gọi *fireAtcurrentTime()* trên chỉ dẫn kèm theo của nó. Phương pháp này yêu cầu một lần kích hoạt mới trong bước vi mô tiếp theo bất kể có sẵn bất kỳ đầu vào bổ sung nào hay không. Nếu chỉ dẫn tôn trọng yêu cầu này (thông thường các chỉ dẫn hẹn giờ sẽ làm như vậy), thì actor sẽ được kích hoạt lại vào thời điểm hiện tại, chậm hơn một bước. Điều này đảm bảo rằng nếu trạng thái đích có một quá trình chuyển đổi được kích hoạt ngay lập tức (trong bước vi mô tiếp theo), thì quá trình chuyển đổi đó sẽ được thực hiện ngay lập tức. Cũng lưu ý rằng trong một Mô hình Bộ nhớ, nếu trạng thái đích có một tinh chỉnh, thì tinh chỉnh đó sẽ được kích hoạt tại thời điểm hiện tại trong bước vi mô tiếp theo. Điều này đặc biệt hữu ích cho các mô hình thời gian liên tục, vì quá trình chuyển đổi có thể biểu thị sự gián đoạn trong các tín hiệu liên tục khác. Sự gián đoạn chuyển thành hai sự kiện riêng biệt có cùng dấu thời gian.

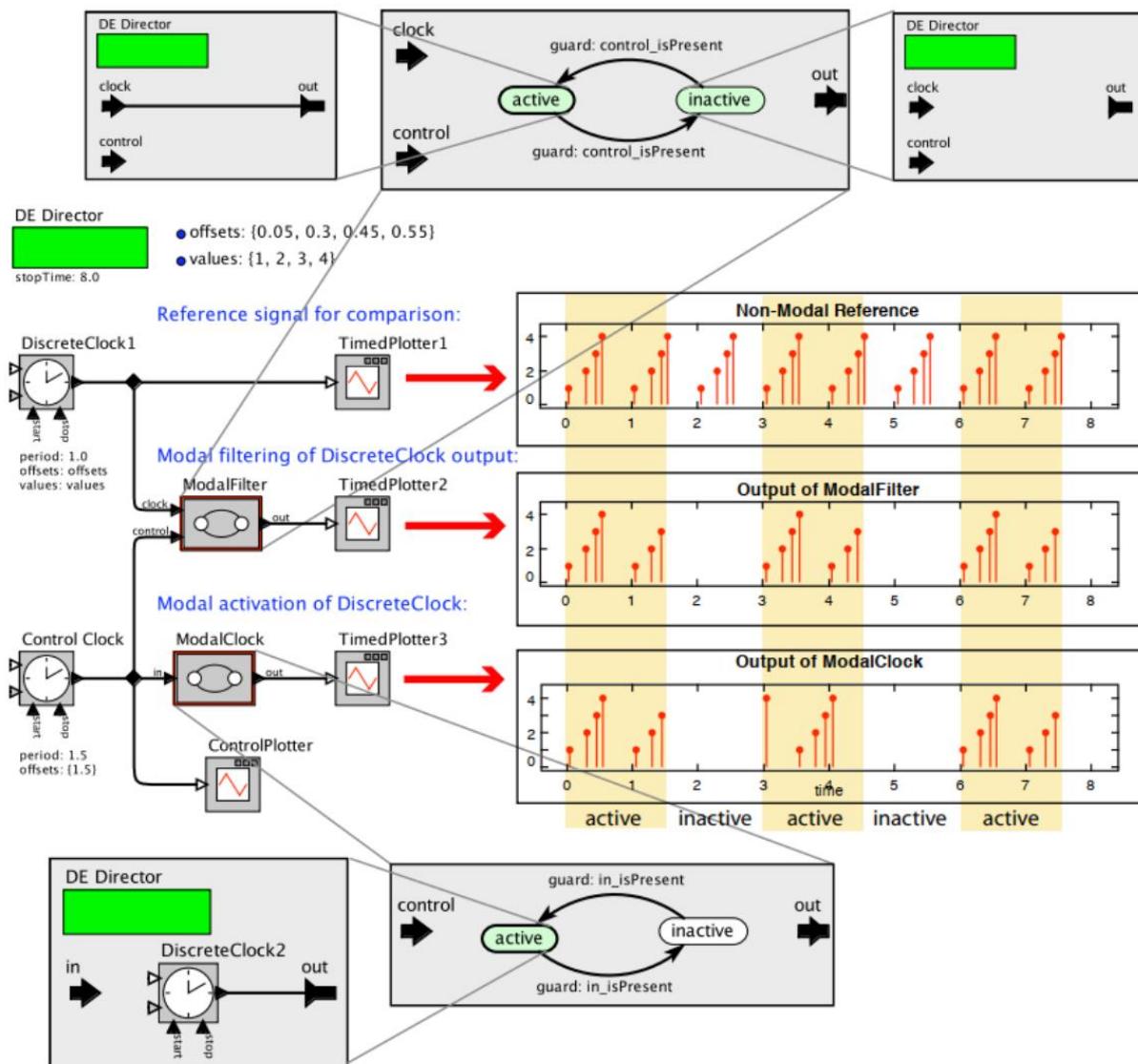


Figure 30: A model that illustrates that putting a stateful timed actor such as **DiscreteClock** inside a modal model is not the same switching between observing and not observing its output.

Ghi nhận

Nỗ lực của nhiều hơn một người đã góp phần giúp cho tác giả hoàn thành Máy trạng thái và Mô hình Bộ nhớ trong Ptolemy II. Phần Bộ nhớ được đảm nhận chủ yếu bởi Thomas Huining Feng, Xiaojun Liu, và Haiyang Zheng. Phần minh họa trong Vergil cho Máy trạng thái được tạo bởi Stephen Neuen-dorffer và Hideo John Reekie. Joern Janneck và Stavros Tripakis đóng góp phần ngữ nghĩa cho Máy định thời trạng thái. Christopher Brooks tạo bản online của mô hình, có thể truy cập từ hyperlink của tập tài liệu này, và cũng đóng góp rất nhiều cho nền móng Ptolemy II. Ngoài ra còn có sự đóng góp của David Hermann, Jie Liu, and Ye Zhou nữa.

Tham khảo

Alur, R., S. Kannan, and M. Yannakakis, 1999: Communicating hierarchical state machines. In 26th International Colloquium on Automata, Languages, and Programming, Springer, vol. LNCS 1644, pp. 169-178.

Andre, C., 1996: SyncCharts: A visual representation of reactive behaviors. Tech. Rep. RR 95-52, rev. RR (96-56), I3S.

Basu, A., M. Bozga, and J. Sifakis, 2006: Modeling heterogeneous real-time components in BIP. In International Conference on Software Engineering and Formal Methods (SEFM), Pune, pp. 3-12.

Beeck, M. v. d., 1994: A comparison of Statecharts variants. In Langmaack, H., W. P. de Roever, and J. Vytopil, eds., Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer-Verlag, Lubeck, Germany, vol. 863 of Lecture Notes in Computer Science, pp. 128-148.

Berry, G. and G. Gonthier, 1992: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2), 87-152.

Booch, G., I. Jacobson, and J. Rumbaugh, 1998: The Unified Modeling Language User Guide. Addison-Wesley.

Carloni, L. P., R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, 2006: Languages and tools for hybrid systems design. Foundations and Trends in Electronic Design Automation, 1(1/2).

de Alfaro, L. and T. Henzinger, 2001: Interface automata. In ESEC/FSE 01: the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, 2003: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE, 91(2), 127-144.

Girault, A., B. Lee, and E. A. Lee, 1999: Hierarchical finite state machines with multiple concurrency models. IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, 18(6), 742-760.

Harel, D., 1987: Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8, 231-274.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, 1990: STATEMATE: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering, 16(4).

Henzinger, T. A., 2000: The theory of hybrid automata. In Inan, M. and R. Kurshan, eds., Verification of Digital and Hybrid Systems, Springer-Verlag, vol. 170 of NATO ASI Series F: Computer and Systems Sciences, pp. 265-292.

Hopcroft, J. and J. Ullman, 1979: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA.

Lynch, N., R. Segala, F. Vaandrager, and H. Weinberg, 1996: Hybrid I/O automata. In Alur, R., T. Henzinger, and E. Sontag, eds., Hybrid Systems III, Springer-Verlag, vol. LNCS 1066, pp. 496-510.

Maler, O., Z. Manna, and A. Pnueli, 1992: From timed to hybrid systems. In Real-Time: Theory and Practice, REX Workshop, Springer-Verlag, pp. 447-484.

Prochnow, S. and R. von Hanxleden, 2007: Statechart development beyond WYSIWYG. In International Conference on Model Driven Engineering Languages and Systems (MoDELS), ACM/IEEE, Nashville, TN, USA.