Chương 14

Trình kiểm tra mô hình TLC

TLC là chương trình tìm lỗi trong thông số kỹ thuật TLA+. Nó được thiết kế và được thực hiện bởi Yuan Yu, với sự giúp đỡ của Leslie Lamport, Mark Hayden, và Mark Tuttle. Nó có sẵn thông qua trang web TLA. Chương này mô tả TLC Phiên bản 2. Tại thời điểm tôi viết bài này, Phiên bản 2 vẫn đang được được triển khai và chỉ có Phiên bản 1. Tham khảo tài liệu đó đi kèm với phần mềm để tìm hiểu xem nó là phiên bản nào và khác biệt như thế nào phiên bản được mô tả ở đây.

14.1 Giới thiệu về TLC

TLC xử lý các thông số kỹ thuật có dạng chuẩn

(14.1) Ban đầu [Tiếp theo]vars Tạm thời

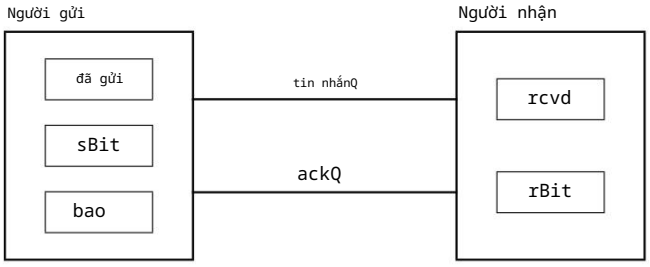
trong đó Init là vị từ ban đầu, Next là hành động trạng thái tiếp theo, vars là bộ dữ liệu của tất cả các biến và Temporal là một công thức thời gian thường chỉ định một điều kiện sống động. Các công thức thời gian và tính sống động được giải thích trong Chương 8. Nếu đặc tả của bạn không chứa công thức Thời gian, thì nó có dạng Bắt đầu [Next]vars , sau đó bạn có thể bỏ qua phần thảo luận về việc kiểm tra thời gian. TLC không xử lý toán tử ẩn (định lượng tồn tại theo thời gian). Bạn có thể kiểm tra thông số kỹ thuật có các biến ẩn bằng cách kiểm tra thông số kỹ thuật bên trong, trong đó các biến đó hiển thị.

Cách hiệu quả nhất để tìm ra lỗi trong đặc tả là cố gắng xác minh rằng nó thỏa mãn các tính chất cần có. TLC có thể kiểm tra thông số kỹ thuật thỏa mãn (ngụ ý) một lớp lớn các công thức TLA—một lớp có hạn chế chính là công thức không được chứa . Bạn cũng có thể chạy TLC mà không cần có nó kiểm tra bất kỳ thuộc tính nào, trong trường hợp đó nó sẽ chỉ tìm hai loại lỗi:

- Lỗi "ngớ ngẩn". Như đã giải thích trong Phần 6.2, một biểu thức ngớ ngẩn là một biểu thức như 3 + 1, 2, mà ý nghĩa của nó không được xác định bởi ngữ nghĩa của TLA+. Một đặc tả không chính xác nếu một số hành vi cụ thể có thỏa mãn hay không phụ thuộc vào ý nghĩa của một biểu thức ngớ ngẩn.

- Bế tắc. Việc không có bế tắc là một đặc tính cụ thể mà chúng ta thường muốn một đặc tả thỏa mãn; nó được biểu thị bằng thuộc tính bất biến (bật Tiếp theo). Một ví dụ phản biện cho thuộc tính này là một hành vi gây ra bế tắc— tức là đạt đến trạng thái trong đó Tiếp theo không được bật, do đó không thể thực hiện thêm bước nào (không lặp lại). TLC thường kiểm tra bế tắc, nhưng việc kiểm tra này có thể bị vô hiệu hóa vì đối với một số hệ thống, khóa chết có thể chỉ cho biết việc chấm dứt thành công.

Việc sử dụng TLC sẽ được minh họa bằng một ví dụ đơn giản - đặc điểm kỹ thuật của giao thức bit xen kẽ để gửi dữ liệu qua đường truyền FIFO bị tổn hao. Người thiết kế thuật toán có thể mô tả giao thức như một hệ thống trông như thế này:



Người gửi có thể gửi một giá trị khi giá trị một bit sBit và sAck bằng nhau. Nó đặt các biến được gửi theo giá trị mà nó đang gửi và bổ sung cho sBit. Giá trị này cuối cùng được gửi đến người nhận bằng cách đặt biến RCvd và bổ sung giá trị một bit rBit. Một thời gian sau, giá trị sAck của người gửi được bổ sung, cho phép giá trị tiếp theo được gửi. Giao thức sử dụng hai đường truyền FIFO bị mất: người gửi gửi dữ liệu và thông tin điều khiển trên msgQ và người nhận gửi xác nhận trên ackQ.

Đặc tả giao thức hoàn chỉnh xuất hiện trong mô-đun AlternatingBit trong Hình 14.1 trên hai trang sau. Nó khá đơn giản, ngoại trừ điều kiện sống động. Vì tin nhắn có thể bị mất nhiều lần khỏi hàng đợi nên cần phải có sự công bằng mạnh mẽ trong các hành động nhận tin nhắn để đảm bảo rằng cuối cùng, tin nhắn liên tục bị gửi lại sẽ được nhận. Tuy nhiên, đừng lo lắng về các chi tiết của thông số kỹ thuật. Hiện tại, tất cả những gì bạn cần biết là các khai báo

```
Dữ liệu không đổi Tập hợp các giá trị dữ liệu có thể được gửi.
```

```
biến msgQ, ackQ, sBit, sAck, rBit, đã gửi, rcvd
```

và các loại biến:

mô-đun xen kẽBit

Thông số kỹ thuật này mô tả một giao thức sử dụng các đường truyền FIFO bị tổn hao để truyền một chuỗi các giá trị từ người gửi đến người nhận. Người gửi gửi giá trị dữ liệu d bằng cách gửi một chuỗi các tin nhắn b, d trên msgQ, trong đó b là bit điều khiển. Nó biết rằng tin nhắn đã được nhận khi nó nhận được ack b từ người nhận trên ackQ. Nó gửi giá trị tiếp theo với một bit điều khiển khác. Người nhận biết rằng tin nhắn trên msgQ chứa một giá trị mới khi bit điều khiển khác với giá trị cuối cùng mà nó nhận được. Người nhận tiếp tục gửi bit điều khiển cuối cùng mà nó nhận được trên ackQ.

mở rộng Naturals, Sequences

hằng số Dữ liệu Tập hợp các giá trị dữ liệu có thể được gửi.

biến msgQ, Chuỗi bit điều khiển, các thông báo giá trị dữ liệu được truyền đến bộ thu.
ackQ, Chuỗi xác nhận một bit được chuyển đến người gửi. sBit, Bit điều
khiển cuối cùng được gửi bởi người gửi; nó được bổ sung khi gửi một giá trị dữ liệu mới.
sAck, Bit xác nhận cuối cùng mà người gửi nhận được. rBit,
Bit điều khiển cuối cùng mà người nhận nhận được.
đã gửi, Giá trị cuối cùng được gửi bởi
người gửi. rcvd Giá trị cuối cùng mà người nhận nhận được.

```
ABInit   =    msgQ =
               ackQ =
               sBit    {0, 1}
               sAck = sBit
               rBit = sBit
               đã gửi    Dữ liệu
                  rcvd    Dữ liệu
```

Điều kiện ban đầu:
Cả hai hàng đợi tin nhắn đều trống.
Tất cả các bit bằng 0 hoặc 1
và bình đẳng với nhau.

Các giá trị ban đầu của đã gửi và RCvd
là các giá trị dữ liệu tùy ý.

```
ABTypeInv   =    msgQ    Seq({0, 1} × Dữ liệu)
                   ackQ    Seq({0, 1})
                   sBit    {0, 1}
                   sAck    {0, 1}
                   rBit    {0, 1 }
                      đã gửi    Dữ
                   liệu    rcvd    Dữ liệu
```

Tính đúng kiểu bất biến.

```
SndNewValue(d)   =
  sAck = sBit
  đã gửi = d
     sBit = 1    sBit
     msgQ = Nối (msgQ, sBit    không    ,d )
  thay đổi ackQ, sAck, rBit, rcvd
```

Hành động trong đó người gửi gửi một giá trị dữ liệu mới d.

Đã bật nếu sAck bằng sBit.
Đặt gửi đến d.
Bổ sung bit điều khiển sBit Gửi giá
trị trên msgQ bằng bit điều khiển mới.

Hình 14.1a: Giao thức bit xen kẽ (bắt đầu).

```
ReSndMsg   =          Người gửi gửi lại tin nhắn cuối cùng đã gửi trên msgQ.
     sAck = sBit
                                                  Đã bật nếu sAck không bằng sBit.
     msgQ = Nối thêm(msgQ, sBit, đã gửi)          Gửi lại giá trị cuối cùng trong gửi.
  ackQ, sBit, sAck, rBit, đã gửi, rcvd không thay đổi

RcvMsg   =            Người nhận nhận được tin nhắn ở phần đầu của msgQ.
     msgQ =                                       Đã bật nếu thông báo Q không trống.
  msgQ = Tail(msgQ)    rBit                      Xóa tin nhắn khỏi phần đầu của msgQ.
  = Head(msgQ)[1]    rcvd =                      Đặt rBit thành bit điều khiển của tin nhắn.
  Head(msgQ)[2]    không thay                    Đặt RCvd thành giá trị dữ liệu của tin nhắn.
  đổi ackQ, sBit, sAck, đã gửi SndAck   =

ackQ = Nối thêm( ackQ, rBit) Người nhận gửi rBit trên ackQ bất kỳ  lúc nào.    không thay
             đổi msgQ, sBit, sAck, rBit, đã gửi, rcvd

RcvAck   =    ackQ =                             Người gửi nhận được một xác nhận về ackQ.
              ackQ = Tail(ackQ)                  Nó loại bỏ ack và đặt sAck về giá trị của nó.
              sAck = Head(ackQ)
              msgQ, sBit, rBit, đã gửi, rcvd không thay đổi

Thua(q)   =           Hành động làm mất tin nhắn từ hàng đợi q.
     q =                                          Đã bật nếu q không trống.
       i    1 . .                                Đối với một số tôi,
       Len(q): q = [j    1 . . (Len(q)    1)    if j < i   hãy xóa chữ i   tin nhắn từ q.
                                                    then q[j] else
  q[j + 1]]    không thay đổi sBit, sAck, rBit, đã gửi, rcvd   Giữ nguyên mọi biến ngoại trừ msgQ và ackQ.

LoseMsg   = Lose(msgQ)    không thay đổi ackQ Mất một tin nhắn từ msgQ.

LoseAck   = Lose(ackQ)    không thay đổi msgQ Mất một tin nhắn từ ackQ.

ABNext   =      d    Dữ liệu : SndNewValue(d)    Hành động ở trạng thái tiếp theo.
                  ReSndMsg    RcvMsg    SndAck    RcvAck
                LoseMsg    LoseAck
─────────────────────────────────────────────────────────────────────────────
abvars   = msgQ, ackQ, sBit, sAck, rBit, đã gửi, rcvd Bộ dữ liệu của tất cả các biến.

ABFairness   =    WFabvar (ReSndMsg)    WFabvar (SndAck)    SFabvar    Điều kiện sống động.
                    (RcvMsg)    SFabvar (RcvAck)
─────────────────────────────────────────────────────────────────────────────
ABSpec   = ABInit     [ABNext]abvars    ABFairness Thông số kỹ thuật hoàn chỉnh.
─────────────────────────────────────────────────────────────────────────────
định lý ABSpec     ABTypeInv
```

Hình 14.1b: Giao thức bit xen kẽ (cuối).

- msgQ là dãy các phần tử trong {0, 1} × Data.
- ackQ là dãy các phần tử trong {0, 1}.
- sBit, sAck và rBit là các phần tử của {0, 1}.
- đã gửi và rcvd là các phần tử của Dữ liệu.

Đầu vào của TLC bao gồm mô-đun TLA+ và tệp cấu hình. TLC giả định thông số kỹ thuật có dạng công thức (14.1) trên trang 221. Tệp cấu hình cho TLC biết tên của thông số kỹ thuật và các thuộc tính cần kiểm tra. Ví dụ: tệp cấu hình cho giao thức bit xen kẽ sẽ chứa phần khai báo

```
THÔNG SỐ KỸ THUẬT
```

yêu cầu TLC lấy ABSpec làm thông số kỹ thuật. Nếu đặc tả của bạn có dạng Init [Next]vars , không có điều kiện tồn tại thì thay vì sử dụng câu lệnh SPECIFICATION, bạn có thể khai báo vị từ ban đầu và hành động trạng thái tiếp theo bằng cách đặt hai câu lệnh sau vào tệp cấu hình:

```
INIT Ban đầu
TIẾP THEO Tiếp theo
```

Thuộc tính hoặc các thuộc tính cần kiểm tra được chỉ định bằng câu lệnh PROPERTY. Ví dụ: để kiểm tra xem ABTypeInv có thực sự là bất biến hay không, chúng ta có thể nhờ TLC kiểm tra xem thông số kỹ thuật có ngụ ý ABTypeInv hay không bằng cách thêm định nghĩa

```
Thuộc tính Inv = ABTypeInv
```

tới mô-đun AlternatingBit và đưa ra câu lệnh

```
BẤT ĐỘNG SẢN InvProperty
```

trong tập tin cấu hình. Việc kiểm tra bất biến phổ biến đến mức TLC cho phép bạn đặt câu lệnh sau vào tệp cấu hình:

```
ABLOST ABTypeInv
```

Câu lệnh INVARIANT phải chỉ định một vị từ trạng thái. Để kiểm tra tính bất biến bằng câu lệnh THUỘC TÍNH, thuộc tính được chỉ định phải có dạng P.
Việc chỉ định một vị từ trạng thái P trong câu lệnh THUỘC TÍNH sẽ yêu cầu TLC kiểm tra xem thông số kỹ thuật có ngụ ý P hay không, nghĩa là P đúng ở trạng thái ban đầu của mọi hành vi đáp ứng thông số kỹ thuật.

TLC hoạt động bằng cách tạo ra các hành vi đáp ứng đặc tả. Để làm được điều này, nó phải được cung cấp cái mà chúng tôi gọi là mô hình đặc tả. Để xác định một mô hình, chúng ta phải gán giá trị cho các tham số không đổi của đặc tả. Tham số không đổi duy nhất của đặc tả giao thức bit xen kẽ là tập hợp

Dữ liệu của các giá trị dữ liệu. Chúng ta có thể yêu cầu TLC để Data bằng tập hợp chứa hai phần tử tùy ý, được đặt tên là d1 và d2, bằng cách đặt khai báo sau vào tệp cấu hình:

Các từ khóa hằng số và CONSTANTS là tương đương, cũng như BẤT BÌNH và BẤT BÌNH.

```
Dữ liệu CONSTANT = {d1, d2}
```

(Chúng ta có thể sử dụng bất kỳ chuỗi chữ cái và chữ số nào chứa ít nhất một chữ cái làm tên của một phần tử.)

Có hai cách để sử dụng TLC. Phương pháp mặc định là kiểm tra mô hình, trong đó nó cố gắng tìm tất cả các trạng thái có thể truy cập—nghĩa là tất cả các trạng thái1 có thể xảy ra trong các hành vi thỏa mãn công thức Init [Next]vars . Bạn cũng có thể chạy TLC ở chế độ mô phỏng, trong đó nó tạo ra các hành vi một cách ngẫu nhiên mà không cần cố gắng kiểm tra tất cả các trạng thái có thể truy cập. Bây giờ chúng ta xem xét việc kiểm tra mô hình; chế độ mô phỏng được mô tả trong Phần 14.3.2 trên trang 243.

Việc kiểm tra toàn diện tất cả các trạng thái có thể truy cập là không thể đối với giao thức bit xen kẽ vì chuỗi tin nhắn có thể dài tùy ý, do đó có vô số trạng thái có thể truy cập. Chúng ta phải hạn chế hơn nữa mô hình để làm cho nó hữu hạn - nghĩa là, nó chỉ cho phép một số lượng hữu hạn các trạng thái có thể có. Chúng ta thực hiện điều này bằng cách định nghĩa một vị từ trạng thái được gọi là ràng buộc xác nhận các giới hạn Phần 14.3 dựa trên độ dài của các chuỗi. Ví dụ: ràng buộc sau đây xác nhận rằng msgQ và ackQ có độ dài tối đa là 2:

low mô tả cách các hành động cũng như các vị từ trạng thái có thể được sử dụng làm các ràng buộc.

```
  Len(msgQ) 2
Len(ackQ) 2
```

Thay vì chỉ định giới hạn về độ dài của chuỗi theo cách này, tôi thích đặt tham số cho chúng và gán giá trị cho chúng trong tệp cấu hình. Chúng tôi không muốn đưa vào đặc tả các tuyên bố và định nghĩa chỉ vì lợi ích của TLC. Vì vậy, chúng tôi viết một mô-đun mới, được gọi là MCAlternatingBit, mở rộng mô-đun AlternatingBit và có thể được sử dụng làm đầu vào cho TLC. Mô-đun này xuất hiện trong Hình 14.2 ở trang tiếp theo. Tệp cấu hình có thể có cho mô-đun xuất hiện trong Hình 14.3 trên trang tiếp theo. Lưu ý rằng tệp cấu hình phải chỉ định các giá trị cho tất cả tham số không đổi của thông số kỹ thuật—trong trường hợp này là tham số Dữ liệu từ mô-đun AlternatingBit và hai tham số được khai báo trong chính mô-đun MCAlternatingBit. Bạn có thể đưa nhận xét vào tệp cấu hình bằng cách sử dụng cú pháp nhận xét TLA+ được mô tả trong Phần 3.5 (trang 32).

Khi một ràng buộc Constr được chỉ định, TLC sẽ kiểm tra mọi trạng thái xuất hiện trong hành vi thỏa mãn Init [Next]vars Constr . Trong phần còn lại của chương này, những trạng thái này sẽ được gọi là trạng thái có thể truy cập được.

?

1Như đã giải thích trong Phần 2.3 (trang 18), trạng thái là sự gán giá trị cho tất cả các biến có thể có. Tuy nhiên, khi thảo luận về một đặc tả cụ thể, chúng ta thường coi một trạng thái là sự gán giá trị cho các biến của đặc tả đó. Đó là những gì tôi đang làm trong chương này.

mô-đun MCAlternatingBit

mở rộng hằng số AlternatingBit

msgQLen, ackQLen

SeqConstraint = Len(msgQ) ≤ msgQLen Len(ackQ) ≤ ackQLen

Ràng buộc về độ dài của trình tự để TLC sử dụng.

Hình 14.2: Mô-đun MCAlternatingBit.

Có TLC kiểm tra kiểu bất biến sẽ mắc nhiều lỗi đơn giản. Khi chúng tôi đã sửa tất cả các lỗi có thể tìm thấy theo cách đó, thì chúng tôi muốn tìm những lỗi ít rõ ràng hơn. Một lỗi phổ biến là một hành động không được kích hoạt khi cần thiết, khiến không thể đạt được một số trạng thái. Bạn có thể phát hiện xem một hành động có bao giờ được bật hay không bằng cách sử dụng tùy chọn bao phủ, được mô tả ở trang 252. Để phát hiện xem một hành động đôi khi có bị vô hiệu hóa do nhầm lẫn hay không, hãy thử kiểm tra các thuộc tính hoạt động. Một đặc tính sống động rõ ràng của giao thức bit xen kẽ là mọi tin nhắn được gửi cuối cùng cũng được gửi đi. Một tin nhắn d đã được gửi khi send = d và sBit = sAck. Vì vậy, một cách ngây thơ để phát biểu tính chất này là

Toán tử tạm thời được xác định ở trang 91.

SentLeadsToRcvd = d

Dữ liệu : (sent = d) (sBit = sAck) (rcvd = d)

Công thức SentLeadsToRcvd xác nhận rằng, đối với mọi giá trị dữ liệu d, nếu được gửi bằng d khi sBit không bằng sAck thì rcvd cuối cùng phải bằng d. Điều này không khẳng định rằng mọi tin nhắn được gửi cuối cùng đều được gửi. Ví dụ: nó được thỏa mãn bởi hành vi trong đó một giá trị cụ thể d được gửi hai lần nhưng chỉ được nhận một lần.
Tuy nhiên, công thức này đủ tốt cho mục đích của chúng tôi vì giao thức không phụ thuộc vào giá trị thực tế được gửi. Nếu có thể gửi cùng một giá trị hai lần nhưng chỉ nhận được một lần thì có thể có hai giá trị khác nhau được gửi và chỉ nhận được một giá trị, vi phạm SentLeadsToRcvd. Do đó, chúng tôi thêm định nghĩa SentLeadsToRcvd vào mô-đun MCAlternatingBit và thêm câu lệnh sau vào tệp cấu hình:

SỞ HỮU Đã gửi Khách hàng tiềm năng tớiRcvd

hằng số Dữ liệu = {d1, d2} (* Cái này đã đủ lớn chưa? *) msgQLen = 2
ackQLen = 2 *
Hãy thử 3 lần tiếp theo.
THÔNG SỐ KỸ THUẬT ABSpec BẤT
BIẾN ABTypeInv SeqConstraint
HẠN CHẾ

Hình 14.3: Tệp cấu hình cho mô-đun MCAlternatingBit.

Việc kiểm tra các thuộc tính độ sống chậm hơn rất nhiều so với các loại kiểm tra khác, vì vậy bạn chỉ nên thực hiện việc đó sau khi đã tìm thấy tất cả các lỗi có thể bằng cách kiểm tra các thuộc tính bất biến.

Kiểm tra tính chính xác của loại và thuộc tính SentLeadsToRcvd là một cách hay để bắt đầu tìm kiếm lỗi. Nhưng cuối cùng, chúng tôi muốn xem liệu giao thức có đáp ứng được đặc điểm kỹ thuật của nó hay không. Tuy nhiên, chúng tôi không có đặc điểm kỹ thuật của nó. Trên thực tế, thông thường trong thực tế là chúng ta được yêu cầu kiểm tra tính đúng đắn của một thiết kế hệ thống mà không có bất kỳ đặc tả chính thức nào về những gì hệ thống phải làm. Trong trường hợp đó, chúng ta có thể viết một đặc tả thực tế sau đó. Tính chính xác của mô-đun ABC trong Hình 14.4 ở trang tiếp theo là đặc điểm kỹ thuật về tính chính xác của giao thức bit xen kẽ. Nó thực sự là một phiên bản đơn giản hóa của đặc tả giao thức, trong đó, thay vì được đọc từ các thông báo, các biến rcvd, rBit và sAck được lấy trực tiếp từ các biến của quy trình khác.

Chúng tôi muốn kiểm tra xem thông số kỹ thuật ABSpec của mô-đun AlternatingBit có ngụ ý công thức ABCSpec của mô-đun ABCchính xác hay không. Để làm điều này, chúng tôi sửa đổi mô-đun MCAlternatingBit bằng cách thêm câu lệnh

```
ví dụ ABC tính chính xác
```

và chúng tôi sửa đổi câu lệnh PROPERTY của tệp cấu hình thành

```
THUỘC TÍNH ABCSpec Đã gửi Khách hàng tiềm năngToRcvd
```

Các từ khóa TÀI SẢN và TÍNH CHẤT là tương đương.

Ví dụ này không điển hình vì đặc tả tính chính xác của ABCSpec không liên quan đến việc ẩn biến (định lượng tồn tại theo thời gian). Bây giờ, giả sử mô-đun ABCorchính xác đã khai báo một biến h khác xuất hiện trong ABCSpec và điều kiện đúng cho giao thức bit xen kẽ là ABCSpec với h ẩn. Điều kiện đúng đắn khi đó sẽ được thể hiện chính thức trong TLA+ như sau:

Việc sử dụng ví dụ này được giải thích trong Phần 4.3 (trang 41).

```
AB(h)   = thể hiện Định lý ABC đúng ABSpec
       h : AB(h)!ABCSpec
```

TLC không thể kiểm tra định lý này một cách trực tiếp vì nó không thể xử lý bộ định lượng tồn tại theo thời gian . Chúng ta sẽ kiểm tra định lý này bằng TLC giống như cách chúng ta cố gắng chứng minh nó—cụ thể là bằng cách sử dụng ánh xạ sàng lọc. Như đã giải thích trong Phần 5.8 trên trang 62, chúng ta sẽ định nghĩa một hàm trạng thái oh theo các biến của mô-đun AlternatingBit và chúng ta sẽ chứng minh

?

(14.2) ABSpec AB(oh)!ABCSpec

Để TLC kiểm tra định lý này, chúng ta sẽ thêm định nghĩa

```
ABCSpecBar   = AB(oh)!ABCSpec
```

và yêu cầu TLC kiểm tra thuộc tính ABCSpecBar .

```
──────────────────────── mô-đun ABC tính chính xác ────────────────────────
mở rộng Naturals

hằng số dữ liệu

biến sBit, sAck, rBit, đã gửi, rcvd
──────────────────────────────────────────────────────────────────────────
ABCInit   =    sBit    {0, 1}
                  sAck = sBit
                  rBit = sBit
                  đã gửi    Dữ
               liệu    rcvd    Dữ liệu

CSndNewValue(d)   =    sAck = sBit
                            đã gửi =
                         d    sBit = 1
                         sBit    không thay đổi sAck, rBit, rcvd

CRcvMsg   =    rBit = sBit
                  rBit = sBit
                  rcvd = đã
                gửi    không thay đổi sBit, sAck, đã gửi

CRcvAck   =    rBit = sAck
                sAck = rBit
                không thay đổi sBit, rBit, đã gửi, rcvd

ABCNext   =      d    Dữ liệu : CSndNewValue(d)
                   CRcvMsg    CRcvAck   =
──────────────────────────────────────────────────────────────────────────
cvars   sBit, sAck, rBit, đã gửi, rcvd

ABCCông bằng   = WFcvar (CRcvMsg)    WFcvar (CRcvAck)

ABCSpec   = ABCInit     [ABCNext]cvars    ABCFairness
──────────────────────────────────────────────────────────────────────────
```

Hình 14.4: Đặc tả tính chính xác của giao thức bit xen kẽ.

Khi TLC kiểm tra một thuộc tính, nó không thực sự xác minh rằng thông số kỹ thuật có hàm ý thuộc tính đó hay không. Thay vào đó, nó kiểm tra xem (i) phần an toàn của đặc tả hàm ý phần an toàn của tài sản và (ii) đặc tả ngụ ý phần sống động của tài sản. Ví dụ: giả sử đặc tả Spec và thuộc tính Prop là

```
Thông số   = Ban đầu     [Tiếp theo]vars    Tạm thời

Prop   = ImpliedInit      [Hành động ngụ ý]pvars    ImpliedTemporal
```

trong đó Temporal và ImpliedTemporal là thuộc tính sống động. Trong trường hợp này, TLC kiểm tra hai công thức

```
Ban đầu     [Tiếp theo]vars    ImpliedInit     [ImpliedAction]pvars
Thông số kỹ thuật    Tạm thời ngụ ý
```

Điều này có nghĩa là bạn không thể sử dụng TLC để kiểm tra xem thông số kỹ thuật không đóng máy có đáp ứng đặc tính an toàn hay không. (Việc đóng máy được thảo luận trong Phần 8.9.2 trên trang 111.) Phần 14.3 bên dưới mô tả chính xác hơn cách TLC kiểm tra các thuộc tính.

14.2 TLC có thể giải quyết những vấn đề gì

Không có trình kiểm tra mô hình nào có thể xử lý tất cả các thông số kỹ thuật mà chúng tôi có thể viết bằng ngôn ngữ có tính biểu cảm cao như TLA+. Tuy nhiên, TLC dường như có thể xử lý hầu hết các thông số kỹ thuật TLA+ mà mọi người thực sự viết. Để TLC xử lý một đặc tả có thể cần một chút thủ thuật, nhưng nó thường có thể được thực hiện mà không cần phải thực hiện bất kỳ thay đổi nào đối với đặc tả đó.

Phần này giải thích những gì TLC có thể và không thể giải quyết, đồng thời đưa ra một số cách để giải quyết. Cách tốt nhất để hiểu những hạn chế của TLC là hiểu cách thức hoạt động của nó. Vì vậy, phần này mô tả cách TLC "thực thi" một đặc tả.

14.2.1 Giá trị TLC

Trạng thái là sự gán giá trị cho các biến. TLA+ cho phép bạn mô tả nhiều giá trị khác nhau—ví dụ: tập hợp tất cả các chuỗi số nguyên tố.
TLC chỉ có thể tính toán một lớp giá trị bị hạn chế, được gọi là giá trị TLC. Các giá trị đó được xây dựng từ bốn loại giá trị nguyên thủy sau:

Boolean	Các giá trị đúng và sai.
số nguyên	Các giá trị như 3 và 1.
Dây	Các giá trị như "ab3".
Giá trị mẫu	Đây là các giá trị được giới thiệu trong câu lệnh CONSTANT của tệp cấu hình. Ví dụ: tệp cấu hình được hiển thị trong Hình 14.3 trên trang 227 giới thiệu các giá trị mô hình d1 và d2. Các giá trị mô hình có tên khác nhau được coi là khác nhau.

Giá trị TLC được xác định theo cách quy nạp là

1. một giá trị nguyên thủy, hoặc

2. một tập hợp hữu hạn các giá trị TLC có thể so sánh được (có thể so sánh được xác định bên dưới) hoặc

3. hàm f có miền xác định là giá trị TLC sao cho f [x] là giá trị TLC, với mọi x trong miền f .

Ví dụ, hai quy tắc đầu tiên ngụ ý rằng

(14.3) {{“a”, “b”}, {“b”, “c”}, {“c”, “d”}}

là giá trị TLC vì quy tắc 1 và 2 ngụ ý rằng {“a”, “b”}, {“b”, “c”} và {“c”, “d”} là các giá trị TLC và sau đó quy tắc thứ hai ngụ ý rằng (14.3) là giá trị TLC. Vì các bộ và bản ghi là các hàm nên quy tắc 3 ngụ ý rằng một bản ghi hoặc bộ có các thành phần là giá trị TLC là giá trị TLC. Ví dụ: 1, “a”, 2, “b” là giá trị TLC.

Để hoàn thành định nghĩa giá trị TLC là gì, tôi phải giải thích ý nghĩa của việc so sánh trong quy tắc 2. Ý tưởng cơ bản là hai giá trị phải có thể so sánh được nếu ngữ nghĩa của TLA+ xác định xem chúng có bằng nhau hay không.
Ví dụ: các chuỗi và số không thể so sánh được vì ngữ nghĩa của TLA+ không cho chúng ta biết liệu “abc” có bằng 42 hay không. Do đó, tập {“abc”, 42} không phải là giá trị TLC; quy tắc 2 không áp dụng vì “abc” và 42 không thể so sánh được. Mặt khác, {“abc”} và {4, 2} có thể so sánh được vì các tập hợp có số phần tử khác nhau phải không bằng nhau. Do đó, tập hợp hai phần tử {{“abc”}, {4, 2}} là giá trị TLC. TLC coi một giá trị mô hình có thể so sánh được và không bằng bất kỳ giá trị nào khác. Các quy tắc chính xác để so sánh được đưa ra trong Phần 14.7.2.

14.2.2 Cách TLC đánh giá biểu thức

Kiểm tra một đặc tả yêu cầu đánh giá các biểu thức. Ví dụ: TLC thực hiện kiểm tra tính bất biến bằng cách đánh giá tính bất biến ở mỗi trạng thái có thể tiếp cận—nghĩa là tính toán giá trị TLC của nó, giá trị này phải đúng. Để hiểu TLC có thể và không thể làm gì, bạn phải biết cách nó đánh giá các biểu thức.

TLC đánh giá các biểu thức một cách đơn giản, thường đánh giá biểu thức con “từ trái sang phải”. Đặc biệt:

- Nó đánh giá p q bằng cách đánh giá p đầu tiên và nếu nó bằng true thì đánh giá-ăn q.
- Nó đánh giá p q bằng cách đánh giá p đầu tiên và nếu nó bằng sai thì đánh giá-ăn q. Nó đánh giá p q là Âp q.
- Nó đánh giá nếu p thì e1 khác e2 bằng cách đánh giá p trước, sau đó đánh giá e1 hoặc e2 .

Để hiểu ý nghĩa của những quy tắc này, chúng ta hãy xem xét một ví dụ đơn giản. TLC không thể đánh giá biểu thức x [1] nếu x bằng , vì [1] là ngớ ngẩn. (Các

dãy trống là một hàm có miền xác định là tập rỗng và do đó
không chứa 1.) Quy tắc đầu tiên ngụ ý rằng, nếu x bằng , thì TLC có thể đánh giá
công thức

```
(x = )    (x [1] = 0)
```

nhưng không phải là công thức (tương đương về mặt logic)

```
(x [1] = 0)    (x = )
```

(Khi đánh giá công thức sau, trước tiên TLC cố gắng tính [1] = 0,
báo lỗi vì không thể.) May mắn thay, chúng ta viết câu đầu tiên một cách tự nhiên
công thức hơn là công thức thứ hai vì nó dễ hiểu hơn. Mọi người hiểu một công thức
bằng cách "đánh giá trong đầu" nó từ trái sang phải, giống như cách
TLC thì có.

TLC đánh giá x S : p bằng cách liệt kê các phần tử s1, . . . , sn của S trong
một số thứ tự và sau đó đánh giá p bằng si thay thế cho x , lần lượt cho
tôi = 1, . . . , N. Nó liệt kê các phần tử của tập hợp S một cách rất đơn giản
theo cách nào đó, nó bỏ cuộc và báo lỗi nếu tập hợp rõ ràng không hữu hạn.
Ví dụ: rõ ràng nó có thể liệt kê các phần tử của {0, 1, 2, 3} và 0 . . 3.
Nó liệt kê một tập hợp có dạng {x S : p} bằng cách liệt kê S đầu tiên, do đó nó có thể
liệt kê {i 0 . . 5 : i < 4} nhưng không phải {i Nat : i < 4}.

??? TLC đánh giá các biểu thức x S : p và chọn x S : p trước
liệt kê các phần tử của S, giống như cách đánh giá x S: p.
Ngữ nghĩa của các trạng thái TLA+ chọn x S : p là một giá trị tùy ý nếu
không có x nào trong S mà p đúng. Tuy nhiên trường hợp này hầu như luôn phát sinh
do nhầm lẫn nên TLC coi đó là lỗi. Lưu ý rằng việc đánh giá
sự biểu lộ

```
nếu n > 5 thì chọn i    1 . . n : i > 5 khác 42
```

sẽ không tạo ra lỗi vì TLC sẽ không đánh giá biểu thức chọn nếu
n 5. (TLC sẽ báo lỗi nếu nó cố đánh giá biểu thức chọn
khi n 5.)

TLC không thể đánh giá các bộ định lượng "không giới hạn" hoặc chọn các biểu thức—điều đó
là, biểu thức có một trong các dạng

```
x :p          x :p        chọn x :p
```

TLC không thể đánh giá bất kỳ biểu thức nào có giá trị không phải là giá trị TLC, như được xác định
trong Mục 14.2.1 ở trên. Đặc biệt, TLC có thể đánh giá biểu thức có giá trị được đặt
chỉ khi biểu thức đó bằng một tập hợp hữu hạn và nó có thể đánh giá một giá trị hàm
chỉ biểu thức nếu biểu thức đó bằng một hàm có miền xác định là tập hữu hạn.
TLC sẽ chỉ đánh giá các biểu thức của các dạng sau nếu nó có thể liệt kê
bộ:

```
   x    S :          x    S :        chọn x    S : p
p {x    S :        p {e : x          [x    S    e]
p} tập con S       S} hợp S
```

TLC thường có thể đánh giá một biểu thức ngay cả khi nó không thể đánh giá tất cả các biểu thức con. Ví dụ, nó có thể đánh giá

```
[n    Nat    n    (n + 1)][3]
```

bằng giá trị TLC 12, mặc dù nó không thể đánh giá

```
[n    Nat    n    (n + 1)]
```

bằng với một hàm có miền xác định là tập Nat. (Một hàm chỉ có thể là giá trị TLC nếu miền của nó là tập hữu hạn.)

TLC đánh giá các hàm được xác định đệ quy bằng quy trình đệ quy đơn giản. Nếu f được xác định bởi f [x S] = e, thì TLC đánh giá f [c] bằng cách đánh giá e bằng c thay thế cho x . Điều này có nghĩa là nó không thể xử lý một số định nghĩa chức năng pháp lý. Ví dụ: hãy xem xét định nghĩa này từ trang 68:

```
mr [n    Nat]   =
      [f    if n = 0 then 17 else mr [n    1].f    mr [n].g , g    if n
      = 0 then 42 else mr [n    1]. f + ông [n    1].g ]
```

Để đánh giá mr [3], TLC thay thế 3 cho n và bắt đầu đánh giá vế phải. Nhưng vì mr [n] xuất hiện ở vế phải, TLC phải đánh giá biểu thức con mr [3], nó thực hiện bằng cách thay thế 3 cho n và bắt đầu đánh giá vế phải. Và như thế. TLC cuối cùng phát hiện ra rằng nó đang ở trong một vòng lặp vô hạn và báo lỗi.

Các định nghĩa đệ quy hợp pháp khiến TLC lặp lại như thế này rất hiếm và chúng có thể được viết lại để TLC có thể xử lý chúng. Hãy nhớ lại rằng chúng ta đã định nghĩa mr để biểu diễn đệ quy tương hỗ:

```
f [n] = nếu n = 0 thì 17 trường hợp khác f [n    1]
g[n] g[n] = nếu n = 0 thì 42 trường hợp khác f [n    1] + g[n    1]
```

Biểu thức con mr [n] xuất hiện trong biểu thức xác định mr [n] vì f [n] phụ thuộc vào g[n]. Để loại bỏ nó, chúng ta phải viết lại đệ quy tương hỗ sao cho f [n] chỉ phụ thuộc vào f [n 1] và g[n 1]. Chúng ta thực hiện điều này bằng cách mở rộng định nghĩa của g[n] trong biểu thức của f [n]. Vì mệnh đề else chỉ áp dụng cho trường hợp n = 0 nên chúng ta có thể viết lại biểu thức cho f [n] dưới dạng

```
f [n] = nếu n = 0 thì 17 trường hợp khác f [n    1]    (f [n    1] + g[n    1])
```

Điều này dẫn đến định nghĩa tương đương sau đây của mr :

```
ông [n    Nat]   =
      [f    nếu n = 0 thì 17
                        else mr [n    1].f    (mr [n    1].f + mr [n    1].g), g
       if n = 0 then 42 else mr [n    1].f + mr [ n    1].g ]
```

Với định nghĩa này, TLC không gặp khó khăn gì trong việc đánh giá ông [3].

Việc đánh giá các vị từ được kích hoạt và toán tử thành phần hành động "·" được mô tả ở trang 240 trong Mục 14.2.6. Phần 14.3 giải thích cách TLC đánh giá các công thức logic thời gian để kiểm tra thời gian.

Nếu bạn không chắc liệu TLC có thể đánh giá một biểu thức hay không, hãy thử và xem. Nhưng đừng đợi cho đến khi TLC đạt được biểu thức ở giữa quá trình kiểm tra toàn bộ thông số kỹ thuật. Thay vào đó, hãy tạo một ví dụ nhỏ trong đó TLC chỉ đánh giá biểu thức đó. Xem phần giải thích ở trang 14.5.3 về cách sử dụng TLC làm máy tính TLA+.

14.2.3 Chuyển nhượng và thay thế

Như chúng ta đã thấy trong ví dụ về bit xen kẽ, tệp cấu hình phải xác định giá trị của từng tham số không đổi. Để gán giá trị TLC v cho tham số c không đổi của thông số kỹ thuật, chúng ta viết c = v trong câu lệnh CONSTANT của tệp cấu hình. Giá trị v có thể là giá trị TLC nguyên thủy hoặc tập hợp hữu hạn các giá trị TLC nguyên thủy được viết dưới dạng {v1, . . . , vn}—ví dụ: {1, -3, 2}. Trong v, bất kỳ chuỗi ký tự nào như a1 hoặc foo không phải là số, chuỗi trích dẫn hoặc TRUE hoặc FALSE đều được coi là giá trị mô hình.

Trong phép gán c = v, ký hiệu c không nhất thiết phải là tham số không đổi; nó cũng có thể là một biểu tượng được xác định. Phép gán này làm cho TLC bỏ qua định nghĩa thực sự của c và lấy v làm giá trị của nó. Phép gán như vậy thường được sử dụng khi TLC không thể tính giá trị của c từ định nghĩa của nó. Đặc biệt, TLC không thể tính giá trị của NotAnS từ định nghĩa

```
KhôngAnS = chọn n : n ∈ S
```

bởi vì nó không thể đánh giá biểu thức chọn không giới hạn. Bạn có thể ghi đè định nghĩa này bằng cách gán giá trị NotAnS trong câu lệnh CONSTANT của tệp cấu hình. Chẳng hạn, nhiệm vụ

```
NotAnS = NS
```

khiến TLC gán cho NotAnS giá trị mô hình NS. TLC bỏ qua định nghĩa thực tế của NotAnS. Nếu bạn đã sử dụng tên NotAnS trong đặc tả, có thể bạn muốn thông báo lỗi của TLC gọi nó là NotAnS thay vì NS. Vì vậy, có lẽ bạn sẽ sử dụng bài tập

```
NotAnS = KhôngAnS
```

gán cho ký hiệu NotAnS giá trị mô hình NotAnS. Hãy nhớ rằng, trong phép gán c = v, ký hiệu c phải được xác định hoặc khai báo trong mô-đun TLA+ và v phải là giá trị TLC nguyên thủy hoặc tập hợp hữu hạn các giá trị đó.

Câu lệnh CONSTANT của tệp cấu hình cũng có thể chứa các phần thay thế có dạng c <- d, trong đó c và d là các ký hiệu được xác định trong TLA+

Lưu ý rằng d là ký hiệu được xác định trong thay thế c <- d, trong khi v là giá trị TLC trong thay thế c = v.

mô-đun. Điều này khiến TLC thay thế c bằng d khi thực hiện các phép tính của nó.
Một cách sử dụng thay thế là đưa ra một giá trị cho tham số toán tử. Ví dụ: giả sử chúng tôi muốn sử dụng TLC để kiểm tra đặc tả bộ nhớ đệm ghi của Mục 5.6 (trang 54). Mô-đun WriteThroughCache mở rộng mô-đun MemoryInterface, chứa phần khai báo

hằng số Gửi(, — — , — , —), Hồi đáp(—, — , — , —), . . .

Chúng ta phải cho TLC biết cách đánh giá các toán tử Gửi và Trả lời. Trước tiên, chúng tôi thực hiện điều này bằng cách viết mô-đun MCWriteThroughCache mở rộng mô-đun WriteThroughCache và xác định hai toán tử

```
MCSend(p, d, cũ, mới)  =  . . .
MCReply(p, d, cũ, mới)  =  . . .
```

Sau đó, chúng tôi thêm vào câu lệnh CONSTANT của tệp cấu hình các phần thay thế

```
Gửi <- MCGửi
Trả lời <- MCReply
```

Việc thay thế cũng có thể thay thế một ký hiệu được xác định bằng một ký hiệu khác. Trong bản đặc tả, chúng ta thường viết những định nghĩa đơn giản nhất có thể. Một định nghĩa đơn giản không phải lúc nào cũng là định nghĩa dễ sử dụng nhất đối với TLC. Ví dụ: giả sử đặc tả của chúng tôi yêu cầu toán tử Sắp xếp sao cho Sắp xếp(S) là một dãy chứa các phần tử của S theo thứ tự tăng dần, nếu S là một tập hữu hạn các số. Đặc tả của chúng tôi trong mô-đun SpecMod có thể sử dụng định nghĩa đơn giản

```
Sắp xếp(S)  =  chọn s  [1 . . Số lượng(S)  S] :
                    i, j  miền s : (i < j)  (s[i] < s[j])
```

Để đánh giá Sắp xếp(S) cho tập hợp S chứa n phần tử, TLC phải liệt kê n n phần tử trong tập hợp [1 · · n S] của hàm. Điều này có thể chậm đến mức không thể chấp nhận được. Chúng ta có thể viết một mô-đun MCSpecMod mở rộng SpecMod và xác định FastSort để nó bằng Sắp xếp khi áp dụng cho các tập hợp số hữu hạn, nhưng có thể được TLC đánh giá hiệu quả hơn. Sau đó chúng ta có thể chạy TLC với tệp cấu hình chứa phần thay thế

```
Sắp xếp <- Sắp xếp nhanh
```

Một định nghĩa có thể có về FastSort được đưa ra trong Phần 14.4, trên trang 250.

14.2.4 Đánh giá các công thức tạm thời

Phần 14.2.2 (trang 231) giải thích loại biểu thức thông thường mà TLC có thể đánh giá. Thông số kỹ thuật và các thuộc tính mà TLC kiểm tra là các công thức tạm thời; phần này mô tả lớp công thức thời gian mà nó có thể xử lý.

TLC có thể đánh giá một công thức tạm thời TLA nếu (i) công thức đó hay—một thuật ngữ được định nghĩa trong đoạn tiếp theo—và (ii) TLC có thể đánh giá tất cả các biểu thức thông thường chứa công thức đó. Ví dụ: một công thức có dạng P Q là tốt, vì vậy TLC có thể đánh giá nó nếu nó có thể đánh giá P và Q.

(Phần 14.3 bên dưới giải thích những trạng thái và cặp trạng thái nào TLC đánh giá các biểu thức thành phần của một công thức thời gian.)

Một công thức thời gian sẽ tốt nếu nó là sự kết hợp của các công thức thuộc về thuộc một trong bốn lớp sau:

Vị ngữ tiểu bang

Công thức bất biến Một công thức có dạng P, trong đó P là một vị từ trạng thái.

Công thức hành động hộp Một công thức có dạng [A]v , trong đó A là một hành động và v là một hàm trạng thái.

Công thức thời gian đơn giản Để định nghĩa lớp này, trước tiên chúng ta thực hiện như sau các định nghĩa:

Thuật ngữ được sử dụng ở đây không phải là tiêu chuẩn.

- Các toán tử Boolean đơn giản bao gồm các toán tử

  и ≡ đúng sai

  của logic mệnh đề cùng với việc định lượng trên các tập hữu hạn, hằng số.

- Công thức trạng thái thời gian là công thức thu được từ các vị từ trạng thái bằng cách áp dụng các toán tử Boolean đơn giản và các toán tử thời gian , và . Ví dụ: nếu N là hằng số thì i 1 . . N : ((x = i) j 1 . . i : (y = j)) là một công thức trạng thái thời gian.

- Công thức hành động đơn giản là một trong những công thức sau đây, trong đó A là một hành động và hàm trạng thái va:

  WFv (A) SFv (A) Av [A]v Các biểu thức thành phần của WFv (A) và SFv (A) là Av và được bật Av . (Việc đánh giá các công thức được kích hoạt được mô tả ở trang 240.)

Khi đó, một công thức thời gian đơn giản được định nghĩa là một công thức được xây dựng từ các công thức trạng thái thời gian và các công thức hành động đơn giản bằng cách áp dụng các toán tử Boolean đơn giản.

Để thuận tiện, chúng tôi loại trừ các công thức bất biến khỏi lớp công thức thời gian đơn giản, do đó bốn lớp công thức thời gian đẹp này không liên kết với nhau.

Do đó TLC có thể đánh giá công thức thời gian

tôi 1 . . N : (y = i) WFy ((y = y + 1) (y ≥ i))

nếu N là một hằng số, bởi vì đây là một công thức thời gian đơn giản (và do đó rất hay) và TLC có thể đánh giá tất cả các biểu thức thành phần của nó. TLC không thể đánh giá x = 1x WFx , vì đây không phải là một công thức hay. Nó không thể đánh giá công thức (x [1] = 0) nếu nó phải đánh giá hành động x [1] = 0x trên một bước s t trong đó x = ở trạng thái t.

Câu lệnh THUỘC TÍ NH có thể chỉ định bất kỳ công thức nào mà TLC có thể đánh giá. Công thức của câu lệnh SPECIFICATION phải chứa chính xác một liên từ là công thức box-action. Liên từ đó chỉ định hành động ở trạng thái tiếp theo.

14.2.5 Ghi đè các mô-đun

TLC không thể tính 2 + 2 từ định nghĩa + có trong mô-đun Naturals tiêu chuẩn. Ngay cả khi chúng ta sử dụng định nghĩa + để từ đó TLC có thể tính tổng, thì việc tính tổng cũng sẽ không diễn ra nhanh chóng. Các toán tử số học như + được triển khai trực tiếp trong Java, ngôn ngữ viết TLC. Điều này đạt được nhờ cơ chế chung của TLC cho phép một mô-đun bị ghi đè bởi một lớp Java triển khai các toán tử được xác định trong mô-đun. Khi TLC gặp câu lệnh mở rộng Naturals, nó sẽ tải lớp Java ghi đè mô-đun Naturals thay vì đọc chính mô-đun đó. Có các lớp Java để ghi đè các mô-đun tiêu chuẩn sau: Naturals, Integers, Sequences, FiniteSets và Bags. (Mô-đun TLC được mô tả bên dưới trong Phần 14.4 cũng bị ghi đè bởi một lớp Java.) Những lập trình viên Java dũng cảm sẽ thấy rằng việc viết một lớp Java để ghi đè một mô-đun không quá khó.

14.2.6 Cách TLC tính toán các trạng thái

Khi TLC đánh giá một bất biến, nó sẽ tính giá trị của bất biến đó, giá trị này là đúng hoặc sai. Khi TLC đánh giá vị từ ban đầu hoặc hành động ở trạng thái tiếp theo, nó sẽ tính toán một tập hợp các trạng thái—đối với vị từ ban đầu, tập hợp tất cả các trạng thái ban đầu và đối với hành động ở trạng thái tiếp theo, tập hợp các trạng thái kế tiếp có thể có (trạng thái bắt đầu).) đạt được từ trạng thái ban đầu (không có mồi) nhất định. Tôi sẽ mô tả cách TLC thực hiện điều này cho hành động ở trạng thái tiếp theo; việc đánh giá vị ngữ ban đầu là tương tự.

Hãy nhớ lại rằng trạng thái là sự gán giá trị cho các biến. TLC tính toán các trạng thái kế tiếp của một trạng thái nhất định s bằng cách gán cho tất cả các biến không được xác định giá trị của chúng ở trạng thái s, không gán giá trị nào cho các biến được xác định trước và sau đó đánh giá hành động ở trạng thái tiếp theo. TLC đánh giá hành động ở trạng thái tiếp theo như được mô tả trong Phần 14.2.2 (trang 231), ngoại trừ hai điểm khác biệt mà bây giờ tôi mô tả. Mô tả này giả định rằng TLC đã thực hiện tất cả các phép gán và thay thế được chỉ định bởi câu lệnh CONSTANT của tệp cấu hình và đã mở rộng tất cả các định nghĩa. Do đó, hành động ở trạng thái tiếp theo là một công thức chỉ chứa các biến, biến số nguyên, giá trị mô hình, toán tử và hằng số TLA+ tích hợp.

Sự khác biệt đầu tiên trong việc đánh giá hành động ở trạng thái tiếp theo là TLC không đánh giá các phân tách từ trái sang phải. Thay vào đó, khi nó đánh giá một công thức con A1 . . . An, nó chia phép tính thành n đánh giá riêng biệt, mỗi đánh giá lấy công thức con là một trong Ai . Tương tự, khi đánh giá x S : p, nó chia phép tính thành các đánh giá riêng biệt cho từng phần tử của S. Hàm ý P Q được coi là phân tách (-P) Q. Ví dụ: TLC phân tách đánh giá của

(A B) (C (i S : D(i)) E)

thành những đánh giá riêng biệt về ba điểm khác nhau -A, B và

C (i S : D(i)) E

Để đánh giá phần tách rời sau, trước tiên nó đánh giá C . Nếu nó nhận được giá trị đúng thì nó sẽ chia đánh giá này thành các đánh giá riêng biệt của D(i) E, cho mỗi i trong S. Nó đánh giá D(i) E bằng cách đánh giá D(i) trước tiên và, nếu nó đạt được giá trị đúng, sau đó đánh giá E.

Sự khác biệt thứ hai trong cách TLC đánh giá hành động ở trạng thái tiếp theo là nếu đối với bất kỳ ,nó đánh giá một biểu thức có dạng x = e khi x không có biến x nào chưa được gán một giá trị thì việc đánh giá mang lại giá trị đúng và TLC gán cho x giá trị thu được bằng cách đánh giá biểu thức e. TLC đánh giá một biểu thức có dạng x S như thể nó là v S : x = v. Nó đánh giá x không đổi khi x = x đối với bất kỳ biến x nào , và e1 không thay đổi, . . . , và như

(không thay đổi e1) . . . (không thay đổi en)

cho bất kỳ biểu thức ei . Do đó, TLC đánh giá x ,y, z không thay đổi như thể nó đã từng

(x = x) (y = y) (z = z)

Ngoại trừ khi đánh giá biểu thức có dạng x = e, TLC sẽ báo lỗi nếu gặp một biến nguyên tố chưa được gán giá trị. Việc đánh giá dừng lại, không tìm thấy trạng thái nào nếu liên từ được đánh giá là sai. Một đánh giá hoàn thành và thu được giá trị true sẽ tìm thấy trạng thái được xác định bởi các giá trị được gán cho các biến được xác định trước. Trong trường hợp sau, TLC báo lỗi nếu một số biến nguyên tố chưa được gán giá trị.

Để minh họa cách thức hoạt động của điều này, hãy xem xét cách TLC đánh giá hành động ở trạng thái tiếp theo

```
(14.4)      x   1 . . Len(y)
             y = Nối(Đuôi(y), x )
       x = x + 1   y =
          Nối(y, x )
```

Trước tiên, chúng tôi xem xét trạng thái bắt đầu với x = 1 và y = 2, 3. TLC chia phép tính thành việc đánh giá hai điểm phân biệt riêng biệt. Nó bắt đầu đánh giá

giao đầu tiên của (14.4) bằng cách đánh giá liên tiếp đầu tiên của nó, được coi là i
1 . . Len(y): x = i. Vì Len(y) = 2 nên việc đánh giá được chia thành các đánh giá riêng
biệt về

(14,5) x = 1 x = 2
y = Nối(Đuôi(y), x) y = Nối(Đuôi(y), x)

TLC đánh giá hành động đầu tiên như sau. Nó đánh giá kết hợp đầu tiên, lấy giá trị true và gán cho x giá trị 1; sau đó nó đánh giá liên kết thứ hai, lấy giá trị true và gán cho y giá trị Append(Tail(2, 3), 1). Vì vậy, đánh giá tác động đầu tiên của (14.5) sẽ tìm thấy trạng thái kế tiếp với x = 1 và y = 3, 1. Tương tự, đánh giá tác động thứ hai của (14.5) tìm thấy trạng thái kế tiếp với x = 2 và y = 3, 2. Theo cách tương tự, TLC đánh giá phân cách thứ hai của (14.4) để tìm trạng thái kế tiếp với x = 2 và y = 2, 3, 2. Do đó, việc đánh giá (14.4) tìm thấy ba trạng thái kế tiếp

Những trạng thái.

Tiếp theo, hãy xem xét cách TLC đánh giá hành động trạng thái tiếp theo (14.4) ở trạng thái có x = 1 và y bằng chuỗi trống. Vì Len(y) = 0 và 1 . . 0 là tập rỗng { }, TLC đánh giá phần rời rạc đầu tiên là

i { } : x =
i y = Nối(Đuôi(y), x)

Việc đánh giá liên kết đầu tiên cho kết quả sai, do đó việc đánh giá liên kết đầu tiên của (14.4) dừng lại, không tìm thấy trạng thái kế tiếp nào. Việc đánh giá điểm phân biệt thứ hai mang lại trạng thái kế tiếp với x = 2 và y = 2.

Vì TLC đánh giá các liên từ từ trái sang phải nên thứ tự của chúng có thể ảnh hưởng đến việc TLC có thể đánh giá hành động ở trạng thái tiếp theo hay không. Ví dụ, giả sử hai liên từ trong liên tiếp đầu tiên của (14.4) bị đảo ngược, như sau:

y = Nối(Đuôi(y), x) x
1 . . Len(y)

Khi TLC đánh giá kết hợp đầu tiên của hành động này, nó gặp biểu thức Append(Tail(y), x) trước khi gán giá trị cho x nên nó báo lỗi. Hơn nữa, ngay cả khi chúng ta thay đổi x đó thành x TLC vẫn không thể đánh giá hành động bắt đầu ở trạng thái với y = vì nó sẽ gặp biểu thức ngớ ngẩn Tail( ) khi đánh giá liên kết đầu tiên.

Mô tả ở trên về cách TLC đánh giá một hành động tùy ý ở trạng thái tiếp theo đủ tốt để giải thích cách thức hoạt động của nó trong hầu hết các trường hợp phát sinh trong thực tế. Tuy nhiên, nó không hoàn toàn chính xác. Ví dụ, nếu hiểu theo nghĩa đen, nó sẽ ngụ ý rằng TLC có thể xử lý hai hành động ở trạng thái tiếp theo sau, cả hai đều tương đương về mặt logic với (x = true) (y = 1):

(14.6) (x = (y = 1)) (x = true) nếu x = đúng thì y = 1 câu khác sai

?????

Trên thực tế, TLC sẽ tạo ra các thông báo lỗi khi được hiển thị với một trong những hành động trạng thái tiếp theo kỳ lạ này.

Hãy nhớ rằng TLC tính toán các trạng thái ban đầu bằng cách sử dụng quy trình tương tự để đánh giá vị từ ban đầu. Thay vì bắt đầu từ các giá trị đã cho của các biến không có số nguyên và gán giá trị cho các biến có số nguyên, nó sẽ gán giá trị cho các biến không có số nguyên.

TLC đánh giá các công thức được kích hoạt về cơ bản giống như cách nó đánh giá hành động ở trạng thái tiếp theo. Chính xác hơn, để đánh giá một công thức đã kích hoạt A, TLC tính toán các trạng thái kế tiếp như thể A là hành động ở trạng thái tiếp theo. Công thức đánh giá là đúng nếu tồn tại trạng thái kế tiếp. Để kiểm tra xem bước s t có thỏa mãn thành phần A·B của hành động A và B hay không, trước tiên TLC tính toán tất cả các trạng thái u sao cho thành phần Hành động s u là một bước A và sau đó kiểm tra xem u t có phải là một B bước cho một số bạn như vậy. được thảo luận ở trang 77.

TLC cũng có thể phải đánh giá một hành động khi kiểm tra tài sản. Trong trường hợp đó, nó đánh giá hành động giống như bất kỳ biểu thức nào và nó không gặp khó khăn gì khi đánh giá ngay cả những hành động kỳ lạ (14.6).

14.3 Cách TLC kiểm tra thuộc tính

Phần 14.2 ở trên giải thích cách TLC đánh giá các biểu thức và tính toán các trạng thái ban đầu và trạng thái tiếp theo. Phần này mô tả cách TLC sử dụng đánh giá để kiểm tra các thuộc tính—đầu tiên là cho chế độ kiểm tra mô hình (mặc định của nó), sau đó là cho chế độ mô phỏng.

Trước tiên, hãy xác định một số công thức thu được từ tệp cấu hình. Trong các định nghĩa này, liên từ đặc tả là liên từ của công thức được đặt tên theo câu lệnh SPECIFICATION (nếu có), liên từ thuộc tính là liên từ của một công thức được đặt tên bởi câu lệnh PROPERTY và liên từ của một tập hợp công thức trống là được xác định là đúng. Các định nghĩa sử dụng bốn loại công thức thời gian đẹp được xác định ở trên trong Phần 14.2.4 trên trang 235.

Init Vị từ trạng thái ban đầu của đặc tả. Nó được chỉ định bởi câu lệnh INIT hoặc SPECIFICATION. Trong trường hợp sau, nó là sự kết hợp của tất cả các liên kết đặc tả là các vị từ trạng thái.

Tiếp theo Hành động ở trạng thái tiếp theo của đặc tả. Nó được chỉ định bởi câu lệnh NEXT hoặc câu lệnh SPECIFICATION. Trong trường hợp sau, chính hành động N sao cho có một liên kết đặc tả có dạng [N]v . Không được có nhiều hơn một liên từ như vậy.

Thời gian Sự kết hợp của mọi liên từ đặc tả không phải là vị từ trạng thái hay công thức hành động hộp. Nó thường là điều kiện sống động của đặc điểm kỹ thuật.

Bất biến Sự kết hợp của mọi vị từ trạng thái I được đặt tên bởi một câu lệnh BẤT ĐỔI hoặc liên kết thuộc tính nào đó bằng I.

ImpliedInit Sự kết hợp của mọi liên kết thuộc tính là một vị từ trạng thái.

ImpliedAction Sự kết hợp của mọi hành động [A]v sao cho một số thuộc tính
liên từ bằng [A]v .

ImpliedTemporal Sự kết hợp của mọi liên kết thuộc tính là một công thức thời gian đơn giản
nhưng không có dạng I, trong đó I là một vị từ trạng thái.

Ràng buộc Sự kết hợp của tất cả các vị từ trạng thái được đặt tên theo trạng thái CONSTRAINT-
ment.

ActionConstraint Sự kết hợp của tất cả các hành động được đặt tên bởi câu lệnh ACTION-
CONSTRAINT. Một ràng buộc hành động tương tự như một ràng buộc thông thường, ngoại trừ
việc nó loại bỏ các chuyển đổi có thể xảy ra thay vì các trạng thái. Ràng buộc thông
thường P tương đương với ràng buộc hành động P .

14.3.1 Chế độ kiểm tra mô hình

TLC giữ hai cấu trúc dữ liệu: đồ thị có hướng G có các nút là trạng thái và hàng đợi (chuỗi)
U gồm các trạng thái. Trạng thái trong G có nghĩa là trạng thái là nút của biểu đồ G. Biểu đồ
G là một phần của biểu đồ khả năng tiếp cận trạng thái mà TLC đã tìm thấy cho đến nay và U
chứa tất cả các trạng thái trong G mà các trạng thái kế tiếp mà TLC chưa tính toán. Tính toán
của TLC duy trì các bất biến sau:

- Các trạng thái của G thỏa mãn vị từ ràng buộc.
- Với mọi trạng thái s thuộc G, cạnh từ s tới s đều thuộc G.
- Nếu có một cạnh trong G từ trạng thái s đến trạng thái t khác, thì t là trạng thái kế
tiếp của s thỏa mãn ràng buộc hành động. Nói cách khác, bước s t thỏa mãn Next
ActionConstraint.
- Mỗi trạng thái s của G có thể truy cập được từ một trạng thái ban đầu (một trạng thái thỏa mãn
vị từ Init) bằng một đường dẫn trong G.
- U là một dãy các trạng thái riêng biệt là các nút trong G.
- Với mọi trạng thái s trong G mà không nằm trong U, và với mọi trạng thái t thỏa mãn Ràng
buộc sao cho bước s t thỏa mãn Next ActionConstraint, trạng thái t và cạnh từ s
tới t đều thuộc G.

TLC thực hiện thuật toán sau, bắt đầu bằng G và U trống:

1. Kiểm tra xem mọi giả định trong đặc tả có được thỏa mãn bởi các giá trị được gán cho
các tham số không đổi hay không.
2. Tính toán tập hợp các trạng thái ban đầu bằng cách đánh giá vị từ ban đầu Init, như
được mô tả ở trên trong Phần 14.2.6. Đối với mỗi trạng thái ban đầu s được tìm thấy:

(a) Đánh giá các vị từ Invariant và ImpliedInit ở trạng thái s; báo cáo lỗi và dừng lại nếu sai.

(b) Nếu ràng buộc vị ngữ là đúng ở trạng thái s thì thêm s vào hàng đợi U và thêm nút s và cạnh s s vào đồ thị G.

3. Trong khi U khác trống, hãy làm như sau:

(a) Loại bỏ trạng thái đầu tiên khỏi U và gọi s là trạng thái

đó. (b) Tìm tập T của tất cả các trạng thái kế tiếp của s bằng cách đánh giá hành động trạng thái tiếp theo bắt đầu từ s, như được mô tả ở trên trong

Phần 14.2.6. (c) Nếu T trống và tùy chọn khóa chết không được chọn thì báo cáo lỗi bế tắc và dừng lại. (d)

Với mỗi trạng thái t trong T, thực hiện như sau:

Tôi. Nếu Bất biến ở trạng thái t là sai hoặc Hành động ngụ ý là sai ở bước s t thì hãy báo lỗi và dừng. ii. Nếu Ràng

buộc vị ngữ là đúng ở trạng thái t và bước s t thỏa mãn ActionConstraint thì A. Nếu t không thuộc G thì thêm

nó vào đuôi của U và thêm nút t và cạnh t t vào G.

B. Cộng cạnh s t vào G.

TLC có thể sử dụng nhiều luồng và các bước 3(b)–(d) có thể được thực hiện đồng thời bởi các luồng khác nhau cho các trạng thái khác nhau. Xem mô tả về tùy chọn công nhân ở trang 253 bên dưới.

Nếu công thức ImpliedTemporal không bằng true thì bất cứ khi nào nó thêm cạnh s t trong quy trình trên, TLC sẽ đánh giá tất cả các vị từ và hành động xuất hiện trong các công thức Temporal và ImpliedTemporal cho bước s t. (Nó thực hiện điều này khi thêm bất kỳ cạnh nào, bao gồm các vòng lặp tự s s và t t ở bước 2(b) và 3(d)ii.A.)

Định kỳ trong quá trình tính toán G và khi tính toán xong G, TLC sẽ kiểm tra thuộc tính ImpliedTemporal như sau. Đặt T là tập hợp bao gồm mọi hành vi τ là chuỗi các trạng thái trên một đường dẫn vô hạn trong G bắt đầu từ trạng thái ban đầu. (Ví dụ: T chứa đường dẫn s s s . . . cho mọi trạng thái ban đầu s trong G.) Lưu ý rằng mọi hành vi trong T đều thỏa mãn Init [Next]vars . TLC kiểm tra xem mọi hành vi trong T cũng thỏa mãn công thức Temporal ImpliedTemporal. (Về mặt khái niệm, đây là những gì xảy ra; TLC không thực sự kiểm tra từng hành vi riêng biệt.) Xem Phần 14.3.5 trên trang 247 bên dưới để thảo luận về lý do tại sao việc kiểm tra thuộc tính ImpliedTemporal của TLC có thể không mang lại kết quả như bạn mong đợi.

Việc tính toán G chỉ kết thúc nếu tập hợp các trạng thái có thể truy cập là hữu hạn. Xem trang 226 để biết định nghĩa về trạng thái có thể truy cập. Nếu không, TLC sẽ chạy mãi mãi—tức là cho đến khi hết tài nguyên hoặc bị dừng.

TLC không phải lúc nào cũng thực hiện cả ba bước được mô tả ở trên. Nó chỉ thực hiện bước 2 đối với mô-đun không cố định, trong trường hợp đó, tệp cấu hình phải chỉ định công thức init. TLC chỉ thực hiện bước 3 nếu tệp cấu hình chỉ định công thức Tiếp theo. TLC phải thực hiện bước này nếu chỉ định công thức Bất biến, ImpliedAction hoặc ImpliedTemporal.

14.3.2 Chế độ mô phỏng

Trong chế độ mô phỏng, TLC liên tục xây dựng và kiểm tra các hành vi riêng lẻ có độ dài tối đa cố định. Độ dài tối đa có thể được chỉ định bằng tùy chọn độ sâu, như được mô tả ở trang 251 bên dưới. (Giá trị mặc định của nó là 100 trạng thái.)
Ở chế độ mô phỏng, TLC chạy cho đến khi bạn dừng nó.

Để tạo và kiểm tra hành vi, TLC sử dụng quy trình được mô tả ở trên để xây dựng biểu đồ G— ngoại trừ điểm khác biệt sau. Sau khi tính toán tập hợp các trạng thái ban đầu và sau khi tính toán tập T tiếp theo cho trạng thái s, TLC chọn ngẫu nhiên một phần tử của tập hợp đó. Nếu phần tử không thỏa mãn ràng buộc thì việc tính toán G sẽ dừng lại. Mặt khác, TLC chỉ đặt trạng thái đó trong G và U, đồng thời kiểm tra công thức Bất biến và ImpliedInit hoặc ImpliedAction cho nó. (Hàng đợi U thực sự không được duy trì, vì nó sẽ không bao giờ chứa nhiều hơn một phần tử.) Việc xây dựng G dừng lại và công thức Temporal ImpliedTemporal được chọn khi số lượng trạng thái tối đa được chỉ định đã được tạo ra. TLC sau đó lặp lại quy trình, bắt đầu với G và U trống.

Các lựa chọn của TLC không hoàn toàn ngẫu nhiên mà được tạo bằng cách sử dụng trình tạo số giả ngẫu nhiên từ hạt giống được chọn ngẫu nhiên. Hạt giống và một giá trị khác gọi là lớp vỏ ngoài được in nếu TLC tìm thấy lỗi. Như được mô tả trong Phần 14.5.1 bên dưới, bằng cách sử dụng các tùy chọn khóa và aril, bạn có thể yêu cầu TLC tạo ra hành vi hiển thị lỗi.

14.3.3 Lượt xem và dấu vân tay

Trong phần mô tả ở trên về cách TLC kiểm tra các thuộc tính, tôi đã viết rằng các nút của đồ thị G là các trạng thái. Điều đó không hoàn toàn chính xác. Các nút của G là các giá trị của hàm trạng thái được gọi là khung nhìn. Chế độ xem mặc định của TLC là bộ tất cả các biến được khai báo, giá trị của chúng xác định trạng thái. Tuy nhiên, bạn có thể chỉ định rằng dạng xem phải là một hàm trạng thái khác myview bằng cách đặt thuật ngữ câu lệnh trạng thái không chính thức

```
XEM cái nhìn của tôi
```

trong tệp cấu hình, trong đó myview là mã định danh được xác định hoặc được khai báo là một biến.

Hãy nhớ rằng chúng ta đang sử dụng có nghĩa là gán các giá trị cho các biến được khai báo, thay vì cho tất cả các biến.

Khi TLC tính toán các trạng thái ban đầu, nó đặt các khung nhìn của chúng thay vì chính các trạng thái đó trong G. (Khung nhìn của trạng thái s là giá trị của hàm trạng thái VIEW trong

trạng thái s.) Nếu có nhiều trạng thái ban đầu có cùng một khung nhìn, thì chỉ một trong số chúng được đặt trong hàng đợi U. Thay vì chèn một cạnh từ trạng thái s vào trạng thái t, TLC sẽ chèn cạnh từ khung nhìn của s vào quan điểm của t. Ở bước 3(d)ii.A trong thuật toán trên, TLC kiểm tra xem khung nhìn của t có nằm trong G hay không.

Khi sử dụng chế độ xem khác với chế độ xem mặc định, TLC có thể dừng trước khi tìm thấy tất cả các trạng thái có thể truy cập. Đối với các trạng thái được tìm thấy, nó thực hiện kiểm tra an toàn một cách chính xác—tức là kiểm tra Bất biến, ImpliedInit và ImpliedAction. Hơn nữa, nó in ra một ví dụ mẫu chính xác (một chuỗi hữu hạn các trạng thái) nếu nó tìm thấy lỗi ở một trong các thuộc tính đó. Tuy nhiên, nó có thể kiểm tra sai thuộc tính ImpliedTemporal. Bởi vì biểu đồ G mà TLC đang xây dựng không phải là biểu đồ khả năng tiếp cận thực tế, nên nó có thể báo cáo lỗi trong thuộc tính ImpliedTemporal khi không tồn tại, in ra một ví dụ mẫu không có thật.

Việc chỉ định chế độ xem không chuẩn có thể khiến TLC không kiểm tra nhiều trạng thái. Bạn nên làm điều đó khi không có nhu cầu kiểm tra các trạng thái khác nhau có cùng góc nhìn. Chế độ xem thay thế có khả năng xảy ra nhất là một bộ dữ liệu bao gồm một số, nhưng không phải tất cả, các biến được khai báo. Ví dụ: bạn có thể đã thêm một hoặc nhiều biến để giúp gỡ lỗi đặc tả. Việc sử dụng bộ biến ban đầu làm dạng xem cho phép bạn thêm các biến gỡ lỗi mà không làm tăng số lượng trạng thái mà TLC phải khám phá. Nếu các thuộc tính đang được kiểm tra không đề cập đến các biến gỡ lỗi thì TLC sẽ tìm tất cả các trạng thái có thể truy cập được của đặc tả gốc và sẽ kiểm tra chính xác tất cả các thuộc tính.

Trong triển khai thực tế, các nút của đồ thị G không phải là các khung nhìn của các trạng thái mà là dấu vân tay của các khung nhìn đó. Dấu vân tay TLC là số 64 bit được tạo bởi chức năng “băm”. Lý tưởng nhất là xác suất để hai chế độ xem khác nhau có cùng dấu vân tay là 2 64, đây là một con số rất nhỏ. Tuy nhiên, có thể xảy ra va chạm, nghĩa là TLC lầm tưởng rằng hai góc nhìn khác nhau là giống nhau vì chúng có cùng một dấu vân tay. Nếu điều này xảy ra, TLC sẽ không khám phá tất cả các trạng thái cần thiết. Đặc biệt, với chế độ xem mặc định, TLC sẽ báo cáo rằng nó đã kiểm tra tất cả các trạng thái có thể truy cập khi chưa có.

Khi nó kết thúc, TLC in ra hai ước tính về xác suất xảy ra xung đột dấu vân tay. Cách đầu tiên dựa trên giả định rằng xác suất của hai chế độ xem khác nhau có cùng dấu vân tay là 2 64. (Theo giả định này, nếu TLC tạo ra n lượt xem với m dấu vân tay riêng biệt thì xác suất xảy ra xung đột là khoảng m (n m) 2 64.) Tuy nhiên, quá trình tạo ra các trạng thái rất không ngẫu nhiên và chưa được biết đến. Sơ đồ lấy dấu vân tay có thể đảm bảo rằng xác suất của hai trạng thái riêng biệt bất kỳ do TLC tạo ra có cùng dấu vân tay thực sự là 2 64. Vì vậy, TLC cũng in ra ước tính thực nghiệm về xác suất xảy ra va chạm. Dựa trên quan sát rằng, nếu đã xảy ra va chạm thì rất có thể đã xảy ra trường hợp "cận va chạm". Ước tính là giá trị tối đa của 1/|f 1 f 2| trên tất cả các cặp f 1, f 2 của dấu vân tay riêng biệt được tạo bởi TLC. Trong thực tế, xác suất va chạm hóa ra là rất nhỏ trừ khi TLC tạo ra hàng tỷ trạng thái riêng biệt.

Chế độ xem và dấu vân tay chỉ áp dụng cho chế độ kiểm tra mô hình. Trong chế độ mô phỏng, TLC bỏ qua mọi câu lệnh VIEW.

14.3.4 Tận dụng tính đối xứng

Các đặc tả bộ nhớ của Chương 5 có tính đối xứng trong tập Proc của các bộ xử lý. Về mặt trực quan, điều này có nghĩa là việc hoán vị các bộ xử lý không làm thay đổi hành vi có đáp ứng đặc tả hay không. Để định nghĩa tính đối xứng chính xác hơn, trước tiên chúng ta cần một số định nghĩa.

Hoán vị của tập hữu hạn S là một hàm có miền xác định và phạm vi bằng S. Nói cách khác, π là một hoán vị của S iff

(S = miền π) (w S : v S : π[v] = w)

Hoán vị là một hàm là hoán vị của miền (hữu hạn) của nó. Nếu π là hoán vị của một tập hợp S các giá trị và s là một trạng thái, hãy đặt s π là trạng thái thu được từ s bằng cách thay thế từng giá trị v trong S bằng π[v]. Để xem ý nghĩa của s, hãy lấy ví dụ về hoán vị π của {“a”, “b”, “c”} sao cho π[“a”] = “b”, π[“b”] = “c ”, và π[“c”] = “a”. Giả sử rằng, ở trạng thái s, giá trị của các biến x và y là

x = “b”, “c”, “d” y =
[i {“a”, “b”} nếu i = “a” thì 7 khác 42]

Khi đó ở trạng thái s^{π}, giá trị của các biến x và y là

x = “c”, “a”, “d” y =
[i {“b”, “c”} nếu i = “b” thì 7 khác 42]

Ví dụ này sẽ cho bạn ý tưởng trực quan về ý nghĩa của s; Tôi sẽ không cố gắng định nghĩa nó một cách chặt chẽ. Nếu σ là hành vi s1, s2, . . . , gọi σ^{π} là hành vi $s_1^{\pi}, s_2^{\pi}, \ldots$.

Bây giờ chúng ta có thể định nghĩa tính đối xứng có nghĩa là gì. Đặc tả Spec đối xứng với một hoán vị π nếu điều kiện sau thỏa mãn: đối với mọi hành vi σ, công thức Spec được thỏa mãn bởi σ nếu nó được thỏa mãn bởi σ^{π}. Các đặc tả bộ nhớ của Chương 5 là đối xứng với bất kỳ hoán vị nào của Proc. Điều này có nghĩa là TLC không cần kiểm tra hành vi σ nếu nó đã kiểm tra hành vi σ^{π} đối với một số hoán vị π của Proc. (Bất kỳ lỗi nào được tiết lộ bởi σ cũng sẽ được tiết lộ bởi σ^{π}.) Chúng ta có thể yêu cầu TLC tận dụng tính đối xứng này bằng cách đặt câu lệnh sau vào tệp cấu hình:

Perms đối xứng

trong đó Perms được xác định trong mô-đun bằng Permutations(Proc), tập hợp tất cả các hoán vị của Proc. (Toán tử hoán vị được định nghĩa trong TLC

mô-đun, được mô tả trong Phần 14.4 bên dưới.) Câu lệnh ĐỐI XỨC này khiến TLC sửa đổi thuật toán được mô tả trên trang 241–242 để nó không bao giờ thêm trạng thái s vào hàng đợi U gồm các trạng thái chưa được kiểm tra và vào biểu đồ trạng thái G của nó nếu G đã chứa trạng thái π s cho một số hoán vị π của Proc. Nếu có n quy trình, điều này sẽ làm giảm số lượng trạng thái mà TLC kiểm tra theo hệ số n !.

Các thông số kỹ thuật bộ nhớ của Chương 5 cũng đối xứng với bất kỳ hoán vị nào của tập hợp địa chỉ bộ nhớ. Để tận dụng tính đối xứng này cũng như tính đối xứng liên quan đến hoán vị của bộ xử lý, chúng tôi xác định tập đối xứng (tập được chỉ định bởi câu lệnh SYMMETRY) bằng nhau

Hoán vị(Proc) Hoán vị(Adr)

Nói chung, câu lệnh SYMMETRY có thể chỉ định một tập đối xứng tùy ý Π, mỗi phần tử của nó là một hoán vị của một tập hợp các giá trị mô hình. Chính xác hơn, mỗi phần tử π trong Π phải là một hoán vị sao cho tất cả các phần tử của miền π được gán các giá trị mô hình theo câu lệnh CONSTANT của tệp cấu hình. (Nếu cấu hình không có câu lệnh SYMMETRY, chúng ta lấy tập đối xứng Π làm tập trống.)

Để giải thích TLC làm gì khi cho một tập đối xứng tùy ý Π, tôi cần thêm một vài định nghĩa. Nếu τ là dãy π1, . . . , πn của các hoán vị trong Π, let). . .) πn . $^{S}\tau$ bằng (. . .((s $^{\pi 1}$ $^{\pi 2}$ (Nếu τ là dãy trống thì s τ được định nghĩa là) bằng s.) Xác định lớp tương đương s của trạng thái s là tập hợp các trạng thái s τ cho tất cả các chuỗi τ hoán vị trong Π. Đối với bất kỳ trạng thái s nào, TLC chỉ giữ lại một phần tử duy nhất của s trong U và G. Điều này được thực hiện bằng những sửa đổi sau đối với thuật toán ở trang 241–242. Ở bước 2(b), TLC chỉ thêm trạng thái s vào U và G nếu U và G chưa chứa trạng thái trong s. Bước 3(d)ii được đổi thành

A. Nếu không có phần tử nào của t thuộc G thì thêm t vào đuôi của U và thêm nút t và cạnh t t tới G.

B. Thêm cạnh s tt vào G, trong đó tt là phần tử duy nhất của t tức là (bây giờ) ở G

Khi một câu lệnh VIEW xuất hiện trong tệp cấu hình, những thay đổi này sẽ được sửa đổi như được mô tả trong Phần 14.3.3 ở trên để các khung nhìn thay vì trạng thái được đặt trong G.

Nếu thực tế, thông số kỹ thuật và các thuộc tính đang được kiểm tra đối xứng với tất cả các hoán vị trong tập hợp đối xứng thì việc kiểm tra Bất biến, ImpliedInit và ImpliedAction của TLC sẽ tìm và báo cáo chính xác bất kỳ lỗi nào mà chúng có thể tìm thấy nếu có câu lệnh SYMMETRY bị bỏ qua.
Tuy nhiên, TLC có thể thực hiện kiểm tra ImpliedTemporal không chính xác—nó có thể bỏ sót lỗi, báo cáo lỗi không tồn tại hoặc báo cáo lỗi thực sự với ví dụ mẫu không chính xác. Vì vậy, bạn chỉ nên thực hiện kiểm tra ImpliedTemporal khi sử dụng câu lệnh SYMMETRY nếu bạn hiểu chính xác TLC đang làm gì.

Nếu thông số kỹ thuật và các thuộc tính không đối xứng đối với tất cả các hoán vị trong tập hợp đối xứng thì TLC có thể không in được dấu vết lỗi nếu nó tìm thấy lỗi. Trong trường hợp đó, nó sẽ in thông báo lỗi

Không thể khôi phục trạng thái từ dấu vân tay của nó.

Bộ đối xứng chỉ được sử dụng trong chế độ kiểm tra mô hình. TLC bỏ qua nó trong chế độ mô phỏng.

14.3.5 Hạn chế của việc kiểm tra tính sống động

Nếu một thông số kỹ thuật vi phạm thuộc tính an toàn thì sẽ có một hành vi hữu hạn mà thuộc hiển thị vi phạm. Hành vi đó có thể được tạo ra bằng một mô hình hữu hạn. Do đó, về nguyên tắc, có thể phát hiện ra hành vi vi phạm TLC. Có thể không thể phát hiện ra sự vi phạm thuộc tính sống động với bất kỳ mô hình hữu hạn nào.

tính An toàn được xác định ở trang 87.

Để biết lý do tại sao, hãy xem xét đặc tả đơn giản EvenSpec sau đây bắt đầu bằng x bằng 0 và tăng liên tục lên 2:

Thông số chẵn = (x = 0) [x = x + 2]x WFx (x = x + 2)

Rõ ràng, x không bao giờ bằng 1 trong bất kỳ hành vi nào thỏa mãn EvenSpec. Vì vậy, EvenSpec không thỏa mãn tính chất sống động (x = 1). Giả sử chúng ta yêu cầu TLC kiểm tra xem EvenSpec có ngụ ý (x = 1) hay không. Để khiến TLC chấm dứt, chúng tôi phải cung cấp một ràng buộc giới hạn nó tạo ra một số lượng hữu hạn các trạng thái có thể truy cập. Tất cả các hành vi vô hạn thỏa mãn (x = 0) [x = x + 2]x mà TLC tạo ra sau đó sẽ kết thúc với vô số bước nói lắp. Trong bất kỳ hành vi nào như vậy, hành động x = x + 2 luôn được bật, nhưng chỉ có một số hữu hạn x = x + 2 bước xảy ra, do đó WFx (x = x + 2) là sai. Do đó TLC sẽ không báo lỗi vì công thức

WFx (x = x + 2) (x = 1)

được thỏa mãn bởi tất cả các hành vi vô hạn mà nó tạo ra.

Khi thực hiện kiểm tra theo thời gian, hãy đảm bảo rằng mô hình của bạn sẽ cho phép các hành vi vô hạn thỏa mãn điều kiện tồn tại của đặc tả. Ví dụ, hãy xem xét mô hình hữu hạn của đặc tả giao thức bit xen kẽ được xác định bởi tệp cấu hình của Hình 14.3 trên trang 227. Bạn nên tự thuyết phục mình rằng nó cho phép các hành vi vô hạn thỏa mãn công thức ABFairness.

Bạn nên xác minh rằng TLC đang thực hiện việc kiểm tra mức độ hoạt động mà bạn mong đợi. Yêu cầu nó kiểm tra thuộc tính sống động mà thông số kỹ thuật không đáp ứng và đảm bảo rằng nó báo lỗi.

```
mô-đun TLC

phiên bản cục bộ Naturals Từ khóa cục bộ có nghĩa là các định nghĩa từ mô-đun đã khởi tạo
không được mô-đun mở rộng TLC lấy được. trình tự cá
thể cục bộ

CÁC VẬN HÀNH ĐỂ GỠ LỖI

Print(out, val)   = val Làm cho TLC in các giá trị ra và val.

Khẳng định(val, out)   =  nếu val = true thì đúng
                          nếu không thì chọn v : true
                                                    Làm cho TLC báo lỗi và in ra nếu
                                                    giá trị không đúng.

Thời gian Java   =  chọn n : n    Nat Khiến TLC in thời gian hiện tại, tính bằng mili giây đã trôi qua
                                  kể từ 00:00 ngày 1 tháng 1 năm 1970 UT, modulo 231 .

Toán tử biểu diễn hàm và tập hợp hoán vị

đ :> đ      = [x    {d}    e]
                                                    Hàm f có miền {d1, . . . , dn } sao cho f
f @@ g      = [x    (miền f )    (miền g)    if x   [di ] = ei n có thể    , với i = 1, . . . ,
                    miền f thì f [x ] khác g[x ]]   bằng văn bản
                                                      d1 :> e1 @@ . . . @@ dn :> vi

Hoán vị(S)   = {f    [S    S] :    w    S :    v    S : f [v] = w} Tập hợp các hoán vị của S.

MỘT NGƯỜI ĐIỀU HÀNH ĐỂ SẮP XẾP

SortSeq(s,  —)  = đặt      Kết quả sắp xếp dãy s theo thứ tự  .
   Perm   =          chọn p    Hoán vị(1 . . Len(s)) :    i, j
                          1 . . Len(s) : (i < j)    (s[p[i]]    s[p[j]])    (s[p[i]] = s[p[j]])
   trong [i    1 . . Len(s)    s[Perm[i]]]
```

Hình 14.5: Mô-đun TLC tiêu chuẩn.

14.4 Mô-đun TLC

Mô-đun TLC tiêu chuẩn, trong Hình 14.5 trên trang này, xác định các toán tử hữu ích khi sử dụng TLC. Mô-đun mà bạn chạy TLC trên đó thường nằm ngoài mô-đun TLC, mô-đun này bị ghi đè khi triển khai Java.

Ghi đè mô-đun được giải thích ở trên trong Phần 14.2.5.

Mô-đun TLC bắt đầu bằng câu lệnh

```
phiên bản địa phương Naturals
```

Như đã giải thích ở trang 171, đây giống như một câu lệnh mở rộng, ngoại trừ việc các định nghĩa có trong mô-đun Naturals không được lấy bởi bất kỳ mô-đun nào khác mở rộng hoặc khởi tạo mô-đun TLC . Tương tự, câu lệnh tiếp theo khởi tạo cục bộ mô-đun Sequences.

Mô-đun TLC tiếp theo xác định ba toán tử Print, Assert và JavaTime.
Chúng không có tác dụng gì ngoại trừ việc chạy TLC, khi chúng có thể giúp bạn theo dõi các vấn đề.

Toán tử Print được xác định sao cho Print(out, val) bằng val. Nhưng khi TLC đánh giá biểu thức này, nó sẽ in ra các giá trị của out và val. Bạn có thể thêm biểu thức In vào đặc tả để giúp xác định lỗi. Ví dụ: nếu thông số kỹ thuật của bạn chứa

```
Print(“a”, true)
P
Print(“b”, true)
```

và TLC in "a" chứ không phải "b" trước khi báo lỗi, thì lỗi xảy ra trong khi TLC đang đánh giá P. Nếu bạn biết lỗi ở đâu nhưng không biết tại sao nó lại xảy ra, bạn có thể thêm biểu thức Print vào cung cấp cho bạn thêm thông tin về những giá trị TLC đã tính toán.

Để hiểu những gì được in khi nào, bạn phải biết cách TLC đánh giá các biểu thức, được giải thích ở trên trong Phần 14.2 và 14.3. TLC thường đánh giá một biểu thức nhiều lần, do đó, việc chèn biểu thức In vào thông số kỹ thuật có thể tạo ra nhiều kết quả. Một cách để giới hạn số lượng đầu ra là đặt biểu thức In bên trong biểu thức if/then để nó chỉ được thực thi trong những trường hợp thú vị.

Mô-đun TLC tiếp theo xác định toán tử Assert sao cho Assert(val, out) bằng true nếu val bằng true. Nếu val không bằng true, việc đánh giá Assert(val, out) sẽ khiến TLC in giá trị out và dừng lại. (Trong trường hợp này, giá trị của Assert(val, out) không liên quan.)

Tiếp theo, toán tử JavaTime được định nghĩa bằng một số tự nhiên tùy ý.
Tuy nhiên, TLC không tuân theo định nghĩa của JavaTime khi đánh giá nó.
Thay vào đó, việc đánh giá JavaTime mang lại thời gian diễn ra đánh giá, được đo bằng mili giây kể từ 00:00 Giờ quốc tế ngày 1 tháng 1 năm 1970, modulo 231. Nếu TLC tạo trạng thái chậm, việc sử dụng toán tử JavaTime kết hợp với biểu thức Print có thể hữu ích bạn hiểu tại sao. Nếu TLC dành quá nhiều thời gian để đánh giá một toán tử, bạn có thể thay thế định nghĩa của toán tử đó bằng một định nghĩa tương đương mà TLC có thể đánh giá hiệu quả hơn. (Xem Phần 14.2.3 trang 234.)

Mô-đun TLC tiếp theo xác định các toán tử :> và @@ sao cho biểu thức

```
d 1 :> e1 @@ . . . @@ dn :> vi
```

là hàm f với miền {d 1, . . . , dn} sao cho f [di] = ei Ví dụ: dãy $_{, \text{với } i = 1, \ldots, N}$.
“ab”, “cd”, là hàm có miền {1, 2}, có thể được viết là

```
1 :> "ab" @@ 2 :> "cd"
```

TLC sử dụng các toán tử này để biểu thị các giá trị hàm mà nó in ra khi đánh giá biểu thức In hoặc báo cáo lỗi. Tuy nhiên, nó thường in các giá trị theo cách chúng xuất hiện trong đặc tả, do đó, nó thường in một chuỗi dưới dạng một chuỗi, không phải theo các toán tử :> và @@.

Tiếp theo là định nghĩa Hoán vị(S) là tập hợp tất cả các hoán vị của S, nếu S là tập hợp hữu hạn. Toán tử Hoán vị có thể được sử dụng để chỉ định một tập hợp các hoán vị cho câu lệnh ĐỐI XỨC được mô tả trong Phần 14.3.4 ở trên. Những sự đối xứng phức tạp hơn có thể được biểu diễn bằng cách định nghĩa một tập hợp {π1, . . . , πn} của các hoán vị, trong đó mỗi πi được viết dưới dạng một hàm rõ ràng bằng cách sử dụng các toán tử :> và @@. Ví dụ, hãy xem xét đặc tả của một hệ thống bộ nhớ trong đó mỗi địa chỉ theo một cách nào đó được liên kết với một bộ xử lý. Thông số kỹ thuật sẽ đối xứng theo hai loại hoán vị: một loại hoán vị các địa chỉ được liên kết với cùng một bộ xử lý và một loại hoán vị các bộ xử lý cùng với các địa chỉ được liên kết của chúng. Giả sử chúng ta yêu cầu TLC sử dụng hai bộ xử lý và bốn địa chỉ, trong đó địa chỉ a11 và a12 được liên kết với bộ xử lý p1 và địa chỉ a21 và a22 được liên kết với bộ xử lý p2. Chúng ta có thể khiến TLC tận dụng được tính đối xứng bằng cách cung cấp cho nó bộ hoán vị sau đây làm tập đối xứng:

```
Hoán vị({a11, a12})    {p1 :> p2 @@ p2 :> p1 @@ a11 :>
                                    a21 @@ a21 :> a11 @@ a12 :> a22
                                    @@ a22 :> a12}
```

Hoán vị p1 :> p2 @@ . . . @@ a22 :> a12 trao đổi bộ xử lý và địa chỉ liên quan của chúng. Hoán vị chỉ hoán đổi a21 và a22 không cần phải được chỉ định rõ ràng vì nó có được bằng cách hoán đổi các bộ xử lý, hoán đổi a11 và a12 và hoán đổi lại các bộ xử lý.

Mô-đun TLC kết thúc bằng cách xác định toán tử SortSeq, có thể được sử dụng để thay thế các định nghĩa toán tử bằng các định nghĩa mà TLC có thể đánh giá hiệu quả hơn. Nếu s là một chuỗi hữu hạn và là mối quan hệ thứ tự tổng thể trên các phần tử của nó thì SortSeq(s,) là chuỗi thu được từ s bằng cách sắp xếp các phần tử của nó theo . Ví dụ: SortSeq(3, 1, 3, 8, >) bằng 8, 3, 3, 1. Việc triển khai SortSeq trên Java cho phép TLC đánh giá nó hiệu quả hơn toán tử sắp xếp do người dùng xác định. Ví dụ: đây là cách chúng ta có thể sử dụng SortSeq để xác định toán tử FastSort nhằm thay thế toán tử Sắp xếp được xác định ở trang 235.

```
FastSort(S)  =
  đặt MakeSeq[SS   tập con S]  = nếu
         SS = {} thì
                          nếu không hãy để ss       = chọn ss   SS : true trong
                                  Append(MakeSeq[SS \ {ss}], ss)
  trong SortSeq(MakeSeq[S], <)
```

14.5 Cách sử dụng TLC

14.5.1 Chạy TLC

Cách bạn chạy TLC chính xác phụ thuộc vào hệ điều hành bạn đang sử dụng và cách cấu hình nó. Có thể bạn sẽ gõ lệnh có dạng

tập tin đặc tả tùy chọn tên chương trình

Ở đâu

tên chương trình dành riêng cho hệ thống của bạn. Nó có thể là java tlatk.TLC.

tệp spec là tên của tệp chứa thông số kỹ thuật TLA+. Mỗi mô-đun TLA+ có tên M xuất hiện trong thông số kỹ thuật phải nằm trong một tệp riêng biệt có tên M .tla. Phần mở rộng .tla có thể bị bỏ qua trong tệp spec.

tùy chọn là một chuỗi bao gồm 0 hoặc nhiều tùy chọn sau:

-bế tắc

Yêu cầu TLC không kiểm tra bế tắc. Trừ khi tùy chọn này được chỉ định, TLC sẽ dừng nếu tìm thấy bế tắc—tức là trạng thái có thể truy cập không có trạng thái kế tiếp.

-mô phỏng

Yêu cầu TLC chạy ở chế độ mô phỏng, tạo ra các hành vi được chọn ngẫu nhiên, thay vì tạo ra tất cả các trạng thái có thể tiếp cận. (Xem Phần 14.3.2 ở trên.)

-số độ sâu

Tùy chọn này khiến TLC tạo ra các hành vi có độ dài tối đa là num trong chế độ mô phỏng. Nếu không có tùy chọn này, TLC sẽ tạo ra các chuỗi có độ dài tối đa là 100. Tùy chọn này chỉ có ý nghĩa khi sử dụng tùy chọn mô phỏng.

-số hạt

Trong chế độ mô phỏng, các hành vi do TLC tạo ra được xác định bởi hạt giống ban đầu được cung cấp cho trình tạo số giả ngẫu nhiên. Thông thường, hạt giống được tạo ngẫu nhiên. Tùy chọn này làm cho TLC đạt tới 263 1. đặt hạt giống là num, phải là số 2 Chạy TLC hai lần trong chế độ mô phỏng với cùng hạt giống và hạt 63 nguyên từ giống (xem tùy chọn hạt giống bên dưới) sẽ tạo ra kết quả giống hệt nhau. Tùy chọn này chỉ có ý nghĩa khi sử dụng tùy chọn mô phỏng.

-aril số

Tùy chọn này khiến TLC sử dụng num làm aril trong chế độ mô phỏng.
Lớp màng ngoài là chất bổ sung của hạt giống ban đầu. Khi TLC tìm thấy lỗi trong chế độ mô phỏng, nó sẽ in ra cả hạt giống ban đầu và lớp hạt.

con số. Việc sử dụng hạt giống và màng hạt ban đầu này sẽ khiến dấu vết đầu tiên được tạo ra trở thành dấu vết lỗi đó. Việc thêm biểu thức In thường sẽ không thay đổi thứ tự TLC tạo dấu vết. Vì vậy, nếu dấu vết không cho bạn biết điều gì đã xảy ra, bạn có thể thử chạy lại TLC trên dấu vết đó để in ra thông tin bổ sung.

-số bảo hiểm

Tùy chọn này khiến TLC in thông tin "phạm vi bảo hiểm" cứ sau vài phút và khi kết thúc quá trình thực thi. Đối với mỗi liên kết hành động "gán một giá trị" cho một biến, TLC sẽ in số lần liên kết đó thực sự được sử dụng để xây dựng một trạng thái mới. Các giá trị mà nó in ra có thể không chính xác nhưng độ lớn của chúng có thể cung cấp thông tin hữu ích. Cụ thể, giá trị 0 biểu thị một phần của hành động ở trạng thái tiếp theo chưa bao giờ được "thực thi". Điều này có thể chỉ ra lỗi trong đặc tả hoặc có thể có nghĩa là TLC mô hình đang kiểm tra quá nhỏ để thực hiện phần hành động đó.

-khôi phục id chạy

Tùy chọn này khiến TLC bắt đầu thực thi thông số kỹ thuật không phải ngay từ đầu mà từ nơi nó dừng lại ở điểm kiểm tra cuối cùng. Khi TLC lấy điểm kiểm tra, nó sẽ in mã định danh lần chạy. (Mã định danh đó giống nhau trong suốt quá trình thực thi TLC.) Giá trị của id lượt chạy phải là mã định danh lượt chạy đó. –

-cleanup

TLC tạo một số tệp khi chạy. Khi nó hoàn thành, nó sẽ xóa tất cả chúng. Nếu TLC phát hiện ra lỗi hoặc nếu bạn dừng lỗi trước khi quá trình này kết thúc, TLC có thể để lại một số tệp lớn. Tùy chọn dọn dẹp khiến TLC xóa tất cả các tệp được tạo bởi các lần chạy trước. Không sử dụng tùy chọn này nếu bạn hiện đang chạy một bản sao TLC khác trong cùng thư mục; nếu bạn làm vậy, nó có thể khiến bản sao kia bị lỗi.

-số khác biệt

Khi TLC tìm thấy lỗi, nó sẽ in dấu vết lỗi. Thông thường, dấu vết đó được in dưới dạng một chuỗi các trạng thái hoàn chỉnh, trong đó trạng thái liệt kê các giá trị của tất cả các biến được khai báo. Tùy chọn difftrace khiến TLC in phiên bản rút gọn của từng trạng thái, chỉ liệt kê các biến có giá trị khác với trạng thái trước đó. Điều này giúp dễ dàng xem những gì đang xảy ra trong mỗi bước nhưng khó tìm được trạng thái hoàn chỉnh hơn.

-ngắn gọn

Thông thường, TLC mở rộng hoàn toàn các giá trị xuất hiện trong thông báo lỗi hoặc trong đầu ra từ việc đánh giá biểu thức In. Tùy chọn ngắn gọn khiến TLC thay vào đó in các phiên bản ngắn hơn, được đánh giá một phần của các giá trị này.

-số công nhân

Các bước 3(b)-(d) của thuật toán thực thi TLC được mô tả trên trang 241–242 có thể được tăng tốc trên máy tính đa bộ xử lý bằng cách sử dụng nhiều luồng. Tùy chọn này khiến TLC sử dụng số luồng khi tìm trạng thái có thể truy cập. Không có lý do gì để sử dụng nhiều luồng hơn số bộ xử lý thực tế trên máy tính của bạn. Nếu tùy chọn này bị bỏ qua, TLC sẽ sử dụng một luồng duy nhất. -config config file Chỉ định rằng tệp cấu hình được đặt tên là

tệp cấu hình, tệp này

phải là tệp có phần mở rộng .cfg. Phần mở rộng .cfg có thể bị bỏ qua trong tệp cấu hình. Nếu tùy chọn này bị bỏ qua, tệp cấu hình được coi là có cùng tên với tệp spec, ngoại trừ phần mở rộng .cfg.

-nowarning Có

các biểu thức TLA+ hợp pháp nhưng không chắc rằng sự hiện diện của chúng có thể chỉ ra lỗi. Ví dụ: biểu thức [f ngoại trừ ![v] = e] có thể không chính xác nếu v không phải là một phần tử của tập xác định f . (Trong trường hợp này, biểu thức chỉ bằng f .)

TLC thường đưa ra cảnh báo khi gặp biểu hiện khó xảy ra như vậy; tùy chọn này ngăn chặn những cảnh báo này.

14.5.2 Gỡ lỗi một đặc tả

Khi bạn viết một đặc tả, nó thường có lỗi. Mục đích của việc chạy TLC là tìm ra càng nhiều lỗi càng tốt. Chúng tôi hy vọng lỗi trong đặc tả sẽ khiến TLC báo lỗi. Thử thách của việc gỡ lỗi là tìm ra lỗi trong đặc tả gây ra lỗi mà TLC báo cáo.

Trước khi giải quyết thách thức này, trước tiên chúng ta hãy kiểm tra kết quả đầu ra của TLC khi nó tìm thấy không có lỗi.

Đầu ra bình thường của TLC

Khi bạn chạy TLC, thứ đầu tiên nó in ra là số phiên bản và ngày tạo:

Thông báo của TLC có thể khác về format so với thông báo được mô tả ở đây.

TLC Phiên bản 2.12 ngày 26 tháng 5 năm 2003

Luôn bao gồm thông tin này khi báo cáo bất kỳ vấn đề nào với TLC. Tiếp theo, TLC mô tả chế độ nó đang được chạy. Các khả năng là

Kiểm tra mô hình

trong đó nó đang kiểm tra toàn diện tất cả các trạng thái có thể truy cập được, hoặc

```
    Chạy mô phỏng ngẫu nhiên với hạt giống 19018030140888511111.
```

trong đó nó đang chạy ở chế độ mô phỏng, sử dụng hạt giống được chỉ định. (Hạt giống được mô tả ở trang 251–252.) Giả sử nó đang chạy ở chế độ kiểm tra mô hình. Nếu bạn yêu cầu TLC thực hiện kiểm tra độ sống, thì giờ đây nó sẽ in nội dung như

```
    Kiểm tra ngụ ý theo thời gian--độ phức tạp tương đối = 8.
```

Thời gian TLC dành cho việc kiểm tra độ sống gần như tỷ lệ thuận với độ phức tạp tương đối. Ngay cả với độ phức tạp tương đối là 1, việc kiểm tra mức độ hoạt động sẽ mất nhiều thời gian hơn so với việc kiểm tra mức độ an toàn. Vì vậy, nếu độ phức tạp tương đối không nhỏ thì TLC có thể sẽ mất rất nhiều thời gian để hoàn thành, trừ khi mô hình rất nhỏ. Trong chế độ mô phỏng, độ phức tạp lớn có nghĩa là TLC sẽ không thể mô phỏng nhiều hành vi. Độ phức tạp tương đối phụ thuộc vào số lượng thuật ngữ và kích thước của các tập hợp được định lượng trong các công thức thời gian.

TLC tiếp theo in một thông báo như

```
    Tính toán xong các trạng thái ban đầu: 4
    trạng thái được tạo, trong đó có 2 trạng thái riêng biệt.
```

Điều này chỉ ra rằng, khi đánh giá vị từ ban đầu, TLC tạo ra 4 trạng thái, trong đó có 2 trạng thái riêng biệt. TLC sau đó in một hoặc nhiều tin nhắn như

```
    Tiến trình (9): 2846 trạng thái được tạo, tìm thấy 984 trạng thái riêng
    biệt. 856 tiểu bang còn lại trong hàng đợi.
```

Thông báo này chỉ ra rằng TLC cho đến nay đã xây dựng một biểu đồ trạng thái G của G và U bằng đường kính[2] 9, nó đã tạo và kiểm tra 2846 trạng thái, tìm ra 984 trạng thái riêng biệt và hàng đợi U của các trạng thái chưa được khám phá chứa 856 trạng thái. Sau khi chạy một thời gian, TLC tạo các báo cáo tiến độ này khoảng 5 phút một lần. Đối với hầu hết các thông số kỹ thuật, số lượng trạng thái trên hàng đợi tăng đều đặn khi bắt đầu thực hiện và giảm dần khi kết thúc. Do đó, báo cáo tiến độ cung cấp hướng dẫn hữu ích về thời gian thực hiện có thể kéo dài.

được ghi ở Mục 14.3.1 trang 241.

Khi TLC hoàn tất thành công, nó sẽ in

```
    Kiểm tra mô hình đã hoàn tất. Không có lỗi nào được tìm thấy.
```

Sau đó nó in một cái gì đó như

2 Đường kính của G là số d nhỏ nhất sao cho mọi trạng thái trong G có thể đạt được từ trạng thái ban đầu bằng một đường đi chứa nhiều nhất d trạng thái. Đó là độ sâu mà TLC đã đạt được trong lần khám phá đầu tiên về chiều rộng của tập hợp các trạng thái. Khi sử dụng nhiều ren (được chỉ định bằng tùy chọn công nhân), báo cáo TLC đường kính có thể không hoàn toàn chính xác.

```
Ước tính xác suất TLC không kiểm tra tất cả các trạng thái có thể truy cập vì
hai trạng thái riêng biệt có cùng
dấu vân tay: được
   tính toán (lạc quan): .000003 dựa trên dấu vân
   tay thực tế: .00007
```

Như đã giải thích ở trang 244, đây là hai ước tính của TLC về xác suất xảy ra xung đột dấu vân tay. Cuối cùng, TLC in một thông báo như

```
2846 trạng thái được tạo, 984 trạng thái riêng biệt được tìm thấy, 0
trạng thái còn lại trong hàng đợi.
Đồ thị trạng thái có đường kính 15.
```

với tổng số trạng thái và đường kính của đồ thị trạng thái.

Trong khi TLC đang chạy, nó cũng có thể in một thông báo như

```
-- Trạng thái chạy điểm kiểm tra/99-05-20-15-47-55 đã hoàn tất
```

Điều này cho thấy rằng nó đã viết một điểm kiểm tra mà bạn có thể sử dụng để khởi động lại TLC trong trường hợp máy tính bị lỗi. (Như đã giải thích trong Phần 14.5.3 trên trang 260, điểm kiểm tra cũng có những cách sử dụng khác.) Mã định danh lần chạy

```
tiểu bang/99-05-20-15-47-55
```

được sử dụng với tùy chọn khôi phục để khởi động lại TLC từ nơi đã thực hiện điểm kiểm tra. Nếu chỉ một phần của thông báo này được in—ví dụ: do máy tính của bạn gặp sự cố trong khi TLC đang sử dụng điểm kiểm tra—có khả năng nhỏ là tất cả các điểm kiểm tra đều bị hỏng và bạn phải khởi động lại TLC lại từ đầu.

Báo cáo lỗi

Vấn đề đầu tiên bạn tìm thấy trong đặc tả của mình có thể là lỗi cú pháp.
TLC báo cáo chúng với

```
ParseException trong ParseSpec:
```

theo sau là thông báo lỗi do Trình phân tích cú pháp tạo ra. Chương 12 giải thích cách diễn giải các thông báo lỗi của máy phân tích. Chạy thông số kỹ thuật của bạn thông qua máy phân tích khi bạn viết nó sẽ nhanh chóng phát hiện được rất nhiều lỗi đơn giản.

Như đã giải thích trong Phần 14.3.1 ở trên, TLC thực hiện ba giai đoạn cơ bản. Trong giai đoạn đầu tiên, nó kiểm tra các giả định; trong phần thứ hai, nó tính toán các trạng thái ban đầu; và ở phần thứ ba, nó tạo ra các trạng thái kế tiếp của các trạng thái trên hàng đợi U của các trạng thái chưa được khám phá. Bạn có thể biết liệu nó đã bước vào giai đoạn thứ ba hay chưa bằng cách nó có in thông báo “các trạng thái ban đầu được tính toán” hay không.

Báo cáo lỗi đơn giản nhất của TLC xảy ra khi phát hiện thấy một trong các thuộc tính mà nó đang kiểm tra không giữ được. Giả sử chúng ta đưa ra một lỗi trong đặc tả bit xen kẽ của mình (Hình 14.1 trên trang 223 và 224) bằng cách thay thế liên kết đầu tiên của ABTypeInv bất biến bằng

```
msgQ  Seq(Dữ liệu)
```

TLC nhanh chóng tìm ra lỗi và in

```
Bất biến ABTypeInv bị vi phạm
```

Tiếp theo, nó in ra một hành vi có độ dài tối thiểu[3] dẫn đến trạng thái không thỏa mãn bất biến:

Lưu ý rằng TLC cho biết phần nào của hành động ở trạng thái tiếp theo cho phép bước tạo ra từng trạng thái.

```
Hành vi cho đến thời điểm này là:
TRẠNG THÁI 1: <Vị ngữ ban đầu> /\ rBit
= 0 /\ sBit =
0 /\ ackQ = <<
>> /\ rcvd = d1 /\ đã
gửi = d1 /\ sAck
= 0 /\ msgQ = <<
>>


TRẠNG THÁI 2: <Hành động ở dòng 66 trong AlternatingBit> /\
rBit = 0 /\
sBit = 1 /\
ackQ = << >> /\ rcvd
= d1 /\ đã gửi
= d1 /\ sAck =
0 /\ msgQ = <<
<< 1, d1 >> >>
```

TLC in từng trạng thái dưới dạng vị từ TLA+ xác định trạng thái. Khi in trạng thái, TLC mô tả các hàm bằng cách sử dụng các toán tử :> và @@ được xác định trong mô-đun TLC. (Xem Phần 14.4 trang 248.)

Các lỗi khó xác định nhất thường là những lỗi được phát hiện khi TLC buộc phải đánh giá một biểu thức mà nó không thể xử lý hoặc một lỗi "ngớ ngẩn" vì giá trị của nó không được chỉ định bởi ngữ nghĩa của TLA+. Ví dụ: hãy đưa một lỗi "từng lỗi một" điển hình vào giao thức bit xen kẽ bằng cách thay thế liên từ thứ hai trong định nghĩa Lose bằng

```
tôi  1 . . Len(q):
q = [j  1 . . (Len(q)  1)  if j < i then q[j  1] else
                                        q[j]]
```

3Khi sử dụng nhiều luồng, có thể, mặc dù không chắc chắn, sẽ có một luồng ngắn hơn hành vi đó cũng vi phạm bất biến.

Nếu q có độ dài lớn hơn 1 thì điều này xác định Lose(q)[1] bằng q[0], đây là một giá trị vô nghĩa nếu q là một chuỗi. (Miền của dãy q là tập 1. . . Len(q), không chứa 0.) Chạy TLC sẽ tạo ra lỗi
tin nhắn

```
Lỗi: Áp dụng bộ << << 1, d1
>>, << 1, d1 >> >> cho số nguyên 0 nằm
ngoài miền.
```

Sau đó nó in ra một hành vi dẫn đến lỗi. TLC tìm thấy lỗi khi đánh giá hành động ở trạng thái tiếp theo để tính toán các trạng thái kế tiếp cho một số trạng thái s và s là trạng thái cuối cùng trong hành vi đó. Nếu xảy ra lỗi khi đánh giá bất biến hoặc hành động ngụ ý, TLC sẽ đánh giá lỗi đó ở trạng thái hoặc bước cuối cùng của hành vi.

Cuối cùng, TLC in vị trí lỗi:

```
Đã xảy ra lỗi khi TLC đang đánh giá các biểu thức lồng nhau ở các vị trí
sau: 0. Dòng 57, cột 7 đến dòng 59, cột 60 trong
AlternatingBit 1. Dòng 58, cột 55 đến dòng 58, cột 60 trong AlternatingBit
```

Vị trí đầu tiên xác định liên kết thứ hai của định nghĩa Thua; cái thứ hai xác định biểu thức q[j 1]. Điều này cho bạn biết rằng lỗi đã xảy ra khi TLC đang đánh giá q[j 1], lỗi này được thực hiện như một phần của việc đánh giá liên từ thứ hai của định nghĩa Lose. Bạn phải suy ra từ dấu vết được in ra rằng nó đang đánh giá định nghĩa Lose trong khi đánh giá hành động LoseMsg. Nói chung, TLC in một cây biểu thức lồng nhau—trước tiên là những biểu thức cấp cao hơn. Nó hiếm khi xác định được lỗi chính xác như bạn mong muốn; thường thì nó chỉ thu hẹp nó lại thành một liên từ hoặc rời rạc của một công thức. Bạn có thể cần phải chèn biểu thức In để xác định vấn đề. Xem phần thảo luận ở trang 259 để được tư vấn thêm về việc xác định lỗi.

14.5.3 Gợi ý sử dụng TLC hiệu quả

Khởi đầu nhỏ

Ràng buộc và gán giá trị cho các tham số không đổi xác định mô hình đặc tả. TLC mất bao lâu để kiểm tra một thông số kỹ thuật tùy thuộc vào thông số kỹ thuật và kích thước của mô hình. Chạy trên trạm làm việc 600 MHz, TLC tìm thấy khoảng 700 trạng thái có thể truy cập riêng biệt mỗi giây đối với đặc tả giao thức bit xen kẽ. Đối với một số thông số kỹ thuật, thời gian TLC cần để tạo trạng thái tăng theo kích thước của mô hình; nó cũng có thể tăng lên khi các trạng thái được tạo ra trở nên phức tạp hơn. Đối với một số thông số kỹ thuật chạy trên các kiểu máy lớn hơn, TLC tìm thấy ít hơn một trạng thái có thể truy cập mỗi giây.

Bạn phải luôn bắt đầu thử nghiệm thông số kỹ thuật bằng một mô hình nhỏ mà TLC có thể kiểm tra nhanh chóng. Đặt các tập hợp quy trình và giá trị dữ liệu chỉ có một phần tử; hãy để hàng đợi có độ dài một. Một thông số kỹ thuật chưa được kiểm tra có thể có rất nhiều lỗi. Một mô hình nhỏ sẽ nhanh chóng nắm bắt được hầu hết những mô hình đơn giản. Khi một mô hình rất nhỏ không còn lỗi nữa, bạn có thể chạy TLC với các mô hình lớn hơn để cố gắng phát hiện các lỗi tinh vi hơn.

Một cách để tìm hiểu mức độ lớn mà TLC mô hình có thể xử lý là ước tính số lượng trạng thái có thể tiếp cận gần đúng dưới dạng hàm của các tham số.
Tuy nhiên, điều này có thể khó khăn. Nếu bạn không thể làm được, hãy tăng kích thước mô hình lên dần dần. Số lượng trạng thái có thể truy cập thường là hàm số mũ của các tham số của mô hình; và giá trị của a tăng rất nhanh khi giá trị của b tăng dần.

Nhiều hệ thống có lỗi chỉ hiển thị trên các mẫu quá lớn để TLC có thể kiểm tra toàn diện. Sau khi yêu cầu mô hình TLC kiểm tra thông số kỹ thuật của bạn trên mô hình lớn nhất mà sự kiên nhẫn của bạn cho phép, bạn có thể chạy mô hình đó ở chế độ mô phỏng trên các mô hình lớn hơn. Mô phỏng ngẫu nhiên không phải là cách hiệu quả để phát hiện các lỗi tinh vi, nhưng nó đáng để thử; bạn có thể gặp may mắn.

Hãy nghi ngờ về sự thành công

Mục 14.3.5 trên trang 247 giải thích lý do tại sao bạn nên nghi ngờ nếu TLC không phát hiện thấy hành vi vi phạm tài sản sống; mô hình hữu hạn có thể che giấu lỗi.
Bạn cũng nên nghi ngờ nếu TLC không tìm thấy lỗi khi kiểm tra các thuộc tính an toàn. Rất dễ dàng để đáp ứng một đặc tính an toàn bằng cách không làm gì cả. Ví dụ: giả sử chúng ta quên đưa hành động SndNewValue vào hành động trạng thái tiếp theo của đặc tả giao thức bit thay thế. Người gửi sau đó sẽ không bao giờ cố gắng gửi bất kỳ giá trị nào. Nhưng đặc tả thu được vẫn đáp ứng điều kiện về tính đúng đắn của giao thức, công thức ABCSpec của mô-đun ABCchính xác.
(Thông số kỹ thuật không yêu cầu phải gửi các giá trị đó.)

Tùy chọn bảo hiểm được mô tả ở trang 252 cung cấp một cách để nắm bắt những vấn đề như vậy. Một cách khác là đảm bảo rằng TLC tìm thấy lỗi trong các thuộc tính cần vi phạm. Ví dụ: nếu giao thức bit xen kẽ đang gửi tin nhắn thì giá trị đã gửi sẽ thay đổi. Bạn có thể xác minh rằng nó có thay đổi bằng cách kiểm tra xem TLC có báo cáo vi phạm thuộc tính không

```
d    Dữ liệu : (đã gửi = d)     (đã gửi = d)
```

Kiểm tra độ chính xác tốt là xác minh rằng TLC tìm thấy các trạng thái chỉ đạt được bằng cách thực hiện một số thao tác. Ví dụ: đặc tả bộ nhớ đệm của Phần 5.6 phải có các trạng thái có thể truy cập được trong đó một bộ xử lý cụ thể có cả thao tác đọc và hai thao tác ghi trong hàng đợi memQ. Để đạt được trạng thái như vậy, bộ xử lý phải thực hiện hai lần ghi, sau đó là đọc vào một địa chỉ không được lưu trong bộ nhớ đệm. Chúng tôi có thể xác minh rằng trạng thái như vậy có thể truy cập được bằng cách yêu cầu TLC tìm thấy sự vi phạm tuyên bố bất biến rằng không có một lần đọc và hai lần ghi cho

cùng một bộ xử lý trong memQ. (Tất nhiên, điều này đòi hỏi một mô hình trong đó memQ có thể đủ lớn.) Một cách khác để kiểm tra xem đã đạt đến trạng thái nhất định hay chưa là sử dụng toán tử Print bên trong biểu thức if/then trong một bất biến để in thông báo khi có trạng thái phù hợp. đạt.

Hãy để TLC giúp bạn tìm ra điều gì đã xảy ra

Khi TLC báo cáo rằng một bất biến bị vi phạm, có thể không rõ phần nào của bất biến đó là sai. Nếu bạn đặt tên riêng cho các liên từ của bất biến và liệt kê chúng riêng biệt trong câu lệnh INVARIANT của tệp cấu hình, TLC sẽ cho bạn biết liên từ nào là sai. Tuy nhiên, có thể khó hiểu tại sao ngay cả một liên từ riêng lẻ cũng sai. Thay vì dành nhiều thời gian cố gắng tự mình tìm ra, việc thêm biểu thức In sẽ dễ dàng hơn và để TLC cho bạn biết điều gì đang xảy ra.

Nếu bạn chạy lại TLC từ đầu với nhiều biểu thức Print, nó sẽ in đầu ra cho mọi trạng thái mà nó kiểm tra. Thay vào đó, bạn nên bắt đầu TLC từ trạng thái mà bất biến là sai. Xác định một vị từ, chẳng hạn như ErrorState, mô tả trạng thái này và sửa đổi tệp cấu hình để sử dụng ErrorState làm vị từ ban đầu. Viết định nghĩa của ErrorState thật dễ dàng—chỉ cần sao chép trạng thái cuối cùng trong dấu vết lỗi của TLC.4

Bạn có thể sử dụng thủ thuật tương tự nếu bất kỳ thuộc tính an toàn nào bị vi phạm hoặc nếu TLC báo lỗi khi đánh giá hành động ở trạng thái tiếp theo. Đối với lỗi trong thuộc tính có dạng [A]v , hãy chạy lại TLC bằng cách sử dụng trạng thái tiếp theo trong dấu vết lỗi làm vị từ ban đầu và sử dụng trạng thái cuối cùng trong dấu vết, với các tên biến được đặt trước, như hành động ở trạng thái tiếp theo. Để tìm lỗi xảy ra khi đánh giá hành động ở trạng thái tiếp theo, hãy sử dụng trạng thái cuối cùng trong dấu vết lỗi làm vị từ ban đầu. (Trong trường hợp này, TLC có thể tìm thấy một số trạng thái kế tiếp trước khi báo lỗi.)

Nếu bạn đã giới thiệu các giá trị mô hình trong tệp cấu hình, chắc chắn chúng sẽ xuất hiện ở các trạng thái được in bởi TLC. Vì vậy, nếu bạn muốn sao chép các trạng thái đó vào mô-đun, bạn sẽ phải khai báo các giá trị mô hình dưới dạng tham số không đổi và sau đó gán cho mỗi tham số này giá trị mô hình cùng tên. Ví dụ: tệp cấu hình mà chúng tôi sử dụng cho giao thức bit xen kẽ giới thiệu các giá trị mô hình d1 và d2. Vì vậy, chúng tôi sẽ thêm vào mô-đun MCAlternatingBit khai báo

```
        hằng số d1, d2
```

và thêm vào câu lệnh CONSTANT của tệp cấu hình các bài tập

```
        d1 = d1         d2 = d2
```

trong đó gán cho các tham số không đổi d1 và d2 các giá trị mô hình tương ứng là d1 và d2.

4Việc xác định ErrorState không dễ dàng như vậy nếu bạn sử dụng tùy chọn difftrace, đây là lý do khiến bạn không sử dụng tùy chọn đó.

Đừng bắt đầu lại sau mỗi lỗi

Sau khi bạn đã loại bỏ những lỗi dễ tìm, TLC có thể phải chạy rất lâu mới tìm ra lỗi. Rất thường xuyên, phải mất nhiều lần để sửa lỗi một cách chính xác. Nếu bạn khởi động TLC lại từ đầu sau khi sửa lỗi, nó có thể chạy trong một thời gian dài chỉ để báo cáo rằng bạn đã mắc một lỗi ngớ ngẩn khi sửa lỗi. Nếu lỗi được phát hiện khi thực hiện một bước từ trạng thái chính xác thì bạn nên kiểm tra sự điều chỉnh của mình bằng cách bắt đầu TLC từ trạng thái đó. Như đã giải thích ở trên, bạn thực hiện việc này bằng cách xác định một vị từ ban đầu mới bằng trạng thái được in bởi TLC.

Một cách khác để tránh bắt đầu lại từ đầu sau khi xảy ra lỗi là sử dụng các điểm kiểm tra. Điểm kiểm tra lưu biểu đồ trạng thái hiện tại G và hàng đợi U của các trạng thái chưa được khám phá. Nó không lưu bất kỳ thông tin nào khác về đặc điểm kỹ thuật. Bạn có thể khởi động lại TLC từ một điểm kiểm tra ngay cả khi bạn đã thay đổi thông số kỹ thuật, miễn là các biến của thông số kỹ thuật và các giá trị mà chúng có thể giả định không thay đổi. Chính xác hơn, bạn có thể khởi động lại từ một điểm kiểm tra nếu chế độ xem của bất kỳ Chế độ xem và trạng thái được tính toán trước điểm kiểm tra không thay đổi và tập hợp đối xứng được xác định trong Phần giống nhau. Khi bạn sửa thấy sau khi chạy phiên bản 14.3.3 và 14.3.4 trong một thời gian dài, bạn có thể muốn sử dụng (trang 252) để tiếp tục TLC tương ứng. từ điểm kiểm tra cuối cùng thay vì yêu cầu nó kiểm tra trạng thái mà nó đã kiểm tra.5

lỗi mà TLC tìm
tùy chọn khôi phục
lại tất cả các

Kiểm tra mọi thứ bạn có thể

Xác minh rằng thông số kỹ thuật của bạn đáp ứng tất cả các thuộc tính mà bạn cho là cần thiết. Ví dụ: bạn không nên hài lòng khi kiểm tra xem thông số kỹ thuật pro-tocol bit xen kẽ có đáp ứng thông số kỹ thuật cấp cao hơn ABCSpec của mô-đun ABCchính xác hay không. Bạn cũng nên kiểm tra các thuộc tính cấp thấp hơn mà bạn mong đợi nó đáp ứng. Một thuộc tính như vậy, được phát hiện qua việc nghiên cứu thuật toán, là không bao giờ có nhiều hơn hai thông báo khác nhau trong hàng đợi msgQ. Vì vậy, chúng ta có thể kiểm tra xem vị từ sau có phải là bất biến hay không:

```
Số lượng( {msgQ[i] : i    1 . . Len(msgQ) } ) 2
```

(Chúng ta phải thêm định nghĩa về Cardinality vào mô-đun MCAlternatingBit bằng cách thêm FiniteSets vào câu lệnh mở rộng của nó.)

Bạn nên kiểm tra càng nhiều thuộc tính bất biến càng tốt. Nếu bạn cho rằng một số vị từ trạng thái phải là bất biến, hãy để TLC kiểm tra nếu đúng như vậy.
Việc phát hiện ra rằng vị từ không phải là bất biến có thể không phát hiện ra lỗi nhưng nó có thể sẽ dạy cho bạn điều gì đó về đặc tả của bạn.

5Một số trạng thái trong biểu đồ G có thể không được điểm kiểm tra lưu lại; chúng sẽ được kiểm tra lại khi khởi động lại từ trạm kiểm soát.

Sáng tạo

Ngay cả khi một thông số kỹ thuật dường như nằm ngoài phạm vi những gì nó có thể xử lý, TLC vẫn có thể giúp kiểm tra nó. Ví dụ: giả sử hành động ở trạng thái tiếp theo của đặc tả có dạng n Nat : A(n). TLC không thể đánh giá việc định lượng trên một tập hợp vô hạn, vì vậy rõ ràng nó không thể giải quyết được thông số kỹ thuật này. Tuy nhiên, chúng ta có thể cho phép TLC đánh giá công thức định lượng bằng cách sử dụng câu lệnh CONSTANT của tệp cấu hình để thay thế Nat bằng tập hữu hạn 0 . . n, đối với một số n. Việc thay thế được giải thích trong Phần- Sự thay thế này làm thay đổi sâu sắc ý nghĩa của đặc tả. Tuy nhiên, nó có phiên bản 14.2.3. tuy nhiên có thể cho phép TLC phát hiện ra các lỗi trong đặc tả. Đừng bao giờ quên rằng mục tiêu của bạn khi sử dụng TLC không phải là để xác minh rằng thông số kỹ thuật có chính xác hay không; đó là để tìm ra lỗi.

Sử dụng TLC làm Công cụ tính TLA+

Hiểu sai một số khía cạnh của TLA+ có thể dẫn đến sai sót trong đặc tả của bạn.
Sử dụng TLC để kiểm tra hiểu biết của bạn về TLA+ bằng cách chạy nó trên các ví dụ nhỏ.
TLC kiểm tra các giả định, do đó bạn có thể biến nó thành một máy tính TLA+ bằng cách yêu cầu nó kiểm tra một mô-đun không có thông số kỹ thuật mà chỉ có các câu lệnh giả định. Ví dụ, nếu g bằng

```
[f ngoại trừ ![d] = e1, ![d] = e2]
```

giá trị của g[d] là bao nhiêu? Bạn có thể hỏi TLC bằng cách để nó kiểm tra mô-đun chứa

```
giả sử f = [i 1 . . 10 1] = [f
                ngoại trừ ![2] = 3, ![2] = 4] g in
          Print(g[2], true)
```

Bạn có thể yêu cầu nó xác minh rằng (F G) ≡ (€F G) là một phản phục bằng cách kiểm tra

```
giả sử F, G boolean : (F G) ≡ (€F G)
```

TLC thậm chí có thể tìm kiếm các phản ví dụ cho một phỏng đoán. Mọi tập hợp có thể được viết dưới dạng phân của hai tập hợp khác nhau không? Kiểm tra nó cho tất cả các tập hợp con của 1 . . 4 với

```
giả sử S tập con (1 . . 4) : if
            T, U tập con (1 . . 4) : (T = U ) (S = T U )
           thì đúng
           khác In(S, đúng)
```

Khi TLC được chạy chỉ để kiểm tra các giả định, nó có thể không cần thông tin từ tệp cấu hình. Nhưng bạn phải cung cấp tệp cấu hình, ngay cả khi tệp đó trống.

14.6 TLC không làm được gì

Chúng tôi muốn TLC tạo ra tất cả các hành vi đáp ứng thông số kỹ thuật. Nhưng không có chương trình nào có thể làm được điều này với một thông số kỹ thuật tùy ý. Tôi đã đề cập đến một số hạn chế của TLC. Có những hạn chế khác mà bạn có thể vấp phải.
Một trong số đó là các lớp Java ghi đè Naturals và Integers . . (231 - 1); TLC báo cáo trong lỗi khoảng 2 nếu bất kỳ phép tính nào tạo ra ^{31} một mô-đun chỉ xử lý các số giá trị nằm ngoài khoảng này.

TLC không thể tạo ra tất cả các hành vi đáp ứng một đặc tả tùy ý, nhưng nó có thể đạt được mục tiêu dễ dàng hơn là đảm bảo rằng mọi hành vi mà nó tạo ra đều đáp ứng đặc tả đó. Tuy nhiên, vì lý do hiệu quả nên không phải lúc nào TLC cũng đáp ứng được mục tiêu này. Nó khác với ngữ nghĩa của TLA+ theo hai cách.

Sai lệch đầu tiên là TLC không bảo toàn ngữ nghĩa chính xác của sự lựa chọn. Như đã giải thích trong Phần 16.1, nếu S bằng T thì chọn x S : P nên bằng chọn x T : P. Tuy nhiên, TLC chỉ đảm bảo điều này nếu S và T giống nhau về mặt cú pháp. Ví dụ: TLC có thể tính các giá trị khác nhau cho hai biểu thức

```
chọn x {1, 2, 3} : x < 3        chọn x {3, 2, 1} : x < 3
```

Sự vi phạm tương tự về ngữ nghĩa của TLA+ tồn tại với các biểu thức trường hợp, ngữ nghĩa của chúng được xác định (trong Phần 16.1.4) theo thuật ngữ lựa chọn.

Phần thứ hai trong ngữ nghĩa của TLA+ mà TLC không bảo toàn là cách biểu diễn các chuỗi. Trong TLA+, chuỗi “abc” là một chuỗi ba phần tử—tức là một hàm có miền {1, 2, 3}. TLC coi chuỗi là giá trị nguyên thủy, không phải là hàm. Do đó, nó coi biểu thức TLA+ hợp pháp “abc”[2] là một lỗi.

14.7 Bản in đẹp

Phần này mô tả chi tiết hai khía cạnh của TLC đã được phác thảo ở trên: ngữ pháp của tệp cấu hình và định nghĩa chính xác về các giá trị TLC.

14.7.1 Ngữ pháp của tệp cấu hình

Ngữ pháp của tệp cấu hình TLC được mô tả trong mô-đun TLA+ ConfigFileGrammar trong Hình 14.6 trên trang tiếp theo. Chính xác hơn, tập hợp các câu ConfigGrammar .File, trong đó ConfigGrammar được xác định trong mô-đun, mô tả tất cả các tệp cấu hình đúng về mặt cú pháp mà các nhận xét đã bị xóa. Mô-đun ConfigFileGrammar mở rộng mô-đun BNFGrammars, được giải thích ở trên trong Phần 11.1.4 (trang 179).

Dưới đây là một số hạn chế bổ sung đối với tệp cấu hình không được mô-đun ConfigFileGrammar chỉ định. Có thể có nhiều nhất một INIT và

```
mô-đun ConfigFileGrammar
mở rộng BNFGrammar

từ vựng
Chữ cái   = OneOf (“abcdefghijklmnopqrstuvwxyz_ABCDEFGHIJKLMNOPQRSTUVWXYZ”)
Số   = Một Trong (“0123456789”)
LetterOrNum   = Chữ cái    Num
                                                                    ''
[ ] ,            = LetterOrNum    OneOf (“    ! @ # \ $ % ^ &       + = |( ) { }  : ;   < > . ? / ”)
AnyChar SingularKW   = {“SPECIFICATION”, “INIT”, “TIẾP THEO”, “XEM”, “ ĐỐI DIỆN"}
PluralKW   =
   {“HÀNH ĐỘNG”, “HẠN CHẾ”, “HÀNH ĐỘNG-HÀNH ĐỘNG”, “HÀNH ĐỘNG-HÀNH ĐỘNG”, “BẤT BIẾN”, “BẤT BIẾN”,
    “TÀI SẢN”, “TÀI SẢN”}
Từ khóa   =Số ítKW    Số nhiềuKW    {“CONSTANT”, “CONSTANTS”}
AnyIdent   = LetterOrNum   & Letter & LetterOrNum    =
Nhận dạng       AnyIdent \ Từ khóa

ConfigGrammar   =          NGỮ PHÁP BNF
  let P(G)   =
            G.File ::= G.Statement+
            G.Statement ::= Tok(SingularKW ) & Tok(Ident )
                            | Tok(Số nhiềuKW ) & Tok(Nhận
                            dạng)    | Tok({“HÀNG”, “HÀNG”})
                              & (G.Replacement | G.Assignment)
          G.Replacement ::= Tok(Ident) & tok(“<  ”) & Tok(AnyIdent)
          G.Assignment ::= Tok(Ident) & tok(“ =”) & G.IdentValue
          G.IdentValue ::= Tok(AnyIdent) | G.Số | G.String tok(“{”)
                                |
                                  & (Nil | G.IdentValue & (tok(“,”) & G.IdentValue)    )
                                  & tok(“}”)
            G.Number ::= (Nil | tok(“  ”)) & Tok(Num+ )
          G.String ::= tok(“ " ") & Tok(AnyChar    ) & tok(“ " ”)
   trong ngữ pháp tối thiểu (P)
```

Hình 14.6: Ngữ pháp BNF của file cấu hình.

một tuyên bố TIẾP THEO. Có thể có một câu lệnh SPECIFICATION, nhưng chỉ khi không có câu lệnh INIT hoặc NEXT. (Xem trang 243 trong Phần 14.3.1 để biết các điều kiện khi các câu lệnh này phải xuất hiện.) Có thể có nhiều nhất một câu lệnh VIEW và nhiều nhất một câu lệnh SYMMETRY. Nhiều trường hợp của các câu lệnh khác được cho phép. Ví dụ, hai phát biểu

```
Inv1 BẤT BIẾN
BẤT BIẾN Inv2 Inv3
```

chỉ định rằng TLC sẽ kiểm tra ba bất biến Inv1, Inv2 và Inv3. Những câu lệnh này tương đương với câu lệnh đơn

```
BẤT BIẾN Inv1 Inv2 Inv3
```

14.7.2 Giá trị TLC có thể so sánh

Mục 14.2.1 (trang 230) mô tả các giá trị TLC. Mô tả đó không đầy đủ vì nó không xác định chính xác khi nào các giá trị có thể so sánh được. Định nghĩa chính xác là hai giá trị TLC có thể so sánh được nếu các quy tắc sau ngụ ý rằng chúng là:

1. Hai giá trị nguyên thủy có thể so sánh được nếu chúng có cùng loại giá trị.

   Quy tắc này ngụ ý rằng “abc” và “123” có thể so sánh được, nhưng “abc” và 123 thì không.

2. Giá trị mô hình có thể so sánh được với bất kỳ giá trị nào. (Nó chỉ bằng chính nó.)

3. Hai tập hợp có thể so sánh được nếu chúng có số phần tử khác nhau hoặc nếu chúng có cùng số phần tử và tất cả các phần tử trong một tập hợp đều có thể so sánh được với tất cả các phần tử trong tập hợp kia.

   Quy tắc này ngụ ý rằng {1} và {“a”, “b”} có thể so sánh được và {1, 2} và {2, 3} có thể so sánh được. Tuy nhiên, {1, 2} và {“a”, “b”} không thể so sánh được.

4. Hai hàm f và g có thể so sánh được nếu (i) miền xác định của chúng có thể so sánh được và (ii) nếu miền xác định của chúng bằng nhau thì f [x] và g[x] có thể so sánh được với mọi phần tử x trong miền xác định của chúng.

   Quy tắc này ngụ ý rằng 1, 2 và “a”, “b”, “c” có thể so sánh được và 1, “a” và 2, “bc” có thể so sánh được. Tuy nhiên, 1, 2 và “a”, “b” không thể so sánh được.