Notas Didáticas da Disciplina "Especificação Formal de Software"

Draylson Micael de Souza
Maria Adelina Silva Brito
Vinícius Pereira

Sumário

Capítulo

1

Métodos Formais

1.1 Considerações Iniciais

Métodos formais de desenvolvimento de software referem-se a técnicas e ferramentas matematicamente rigorosas para a especificação, projeto e verificação de sistemas de hardware e software (Butler, 2001). A expressão "matematicamente rigorosas" significa que as especificações utilizadas em métodos formais são declarações bem formadas descritas em lógica matemática e que verificações formais são deduções rigorosas sobre esta lógica. Neste capítulo são discutidas as vantagens da utilização de métodos formais e alguns mitos associados à essa prática.

1.2 Fundamento de Métodos Formais

Um método formal deve ser não ambíguo e preciso em sua representação. Uma frase em português (que é uma linguagem natural) a depender de como foi formada pode ser não ambígua, mas ainda não há como provar sua precisão e nem mesmo sua corretude. Se fosse realizada uma pesquisa por exemplo, entre todas as pessoas que falam português, poderia haver indícios de que uma determinada frase não é ambígua, caso todos cheguem a mesma conclusão, porém não terá sido provado nada sobre sua precisão ou correção.

Procurando resolver esses problemas utiliza-se a matemática como fundamento de métodos formais, pois esta é uma linguagem que pode ser representada com precisão

e sem ambiguidades. Além disso, a matemática é uma linguagem universal, em que qualquer pessoa pode compreender sua representação. Ao contrário de uma linguagem natural, a matemática não é apenas rigorosa em sua representação, mas é formal. Pode-se provar a correção de suas expressões. Basicamente, são abordados dois grandes tópicos matemáticos na área de métodos formais: *Teoria dos conjuntos* e *Lógica*.

Considerando a matemática como fundamento para a especificação formal, como ela pode ser utilizada no desenvolvimento de software? Numa especificação estão descritos o software e seus requisitos. Assim, quando pretende-se especificar um novo software, são colocados os detalhes do mesmo juntamente com os detalhes do mundo real onde ele será aplicado. A especificação serve de contrato entre o cliente e o fornecedor do software, servindo para registrar "o que" e "como" será desenvolvido. Este documento é utilizado nas demais etapas (projeto, codificação, testes, etc).

A especificação também pode ser utilizada na modelagem. O mundo real pode ser modelado considerando o escopo do software, formando assim uma representação do mundo real. Tal representação pode ser útil para verificar propriedades ou ser utilizada para mostrar que o modelo do mundo real é igual ao modelo especificado, ou seja, atingem o mesmo objetivo. Métodos formais lidam com estas duas utilidades: 1) empregado no teste formal de software, que consiste na fase de validação e; 2) empregado na verificação.

1.3 Sete Mitos de Métodos Formais

O estudo "*Seven myths of formal methods*"(Hall, 1990), apresenta os sete mitos com relação a métodos formias. O primeiro mito diz que métodos formais podem garantir que o software está perfeito, correto. Este é apenas um mito, não podendo-se garantir que o software está especificado corretamente, pois nem tudo pode ser provado. Mas estes métodos oferecem mais garantias e chegam mais próximo do ideal.

O segundo mito é provar que o programa está correto. Métodos formais não são úteis apenas para isto, mas também podem ser utilizados para especificar, de forma que todos compreendam a mesma coisa. O uso de fórmulas assegura o mesmo entendimento entre as pessoas sobre uma propriedade, mas as fórmulas devem ser mescladas com texto, de modo que o texto descreva intuitivamente as fórmulas (como se o texto fosse o comentário de um código-fonte). Estas formalizam a ideia com precisão e sem ambiguidades.

O terceiro mito afirma que métodos formais são úteis apenas para sistemas críticos (que envolvem grandes somas em dinheiro e risco de vida). Mas isto não procede, pois existem outros casos em que é necessário evitar erros durante a execução do software. Deve-se analisar o custo/benefício de possuir estes erros e analisar se é necessário aplicar métodos formais.

O quarto mito considera que apenas matemáticos treinados podem usar métodos formais, o que não é verdade, pois os conceitos matemáticos utilizados são um subconjunto da matemática que é bem conhecido dos profissionais da área de computação. Além disso, existem métodos formais "leves" que abstraem a parte pesada da matemática.

O quinto mito diz que usar métodos formais aumenta o custo do desenvolvimento. Isso pode ser verdade em alguns casos. Deve-se avaliar o custo/benefício para sua aplicação. Não se pode provar que o custo de um projeto aumenta ou diminui apenas por utilizar métodos formais.

O sexto mito retrata que métodos formais não são aceitáveis para o usuário final. É importante que o usuário leia, entenda e aceite a especificação do software, e conceitos matemáticos podem não ser compreensíveis aos usuários e clientes. Porém, por ser utilizada notação matemática, é possível animar a especificação. Deste modo, simulando a execução do software o cliente avalia o que foi especificado. Isto também pode ser considerado uma forma de prototipação. Alguns métodos como Redes de Petri e Máquinas de Estados podem ser compreendidos pelo usuário, por serem mais intuitivos e mais familiares a outras representações.

O sétimo e último mito diz que métodos formais são usados apenas para sistemas pequenos. Realmente no ensino são utilizados pequenos exemplos de sistemas. Mas o maior benefício de métodos formais ocorre em grandes sistemas. Isso acontece pois a memória humana é limitada, podendo-se escrever uma grande especificação com partes contraditórias. Usando-se uma notação matemática, tais partes podem ser percebidas e anuladas, pois tem-se a capacidade de confrontar partes distintas do software e ver se estão em conflito.

1.4 Considerações Finais

Neste capítulo foi apresentada uma visão geral de métodos formais. No próximo capítulo é dada ênfase na fundamentação matemática que apoia a prática de métodos formais, apresentando um breve resumo de teoria dos conjuntos e lógicas.

Capítulo

2

Teoria dos Conjuntos e Lógicas

2.1 Considerações Iniciais

Um método formal deve ser não ambíguo e preciso em sua representação. Para isto, métodos formais utilizam como fundamentos a matemática, em especial, a teoria de conjuntos e lógicas. Neste sentido, neste capítulo é apresentado um resumo de teoria de conjuntos e lógicas.

2.2 Teoria dos Conjuntos

A teoria dos conjuntos é uma vertente da matemática considerada uma das mais intuitivas, sendo introduzida desde os níveis mais elementares do aprendizado. Conjuntos são coleções de elementos e o desenvolvimento da teoria visa representar e analisar a pertinência de tais conjuntos.

Algumas formas são usualmente adotadas para a descrição de conjuntos, como o uso de diagramas e representações visuais, descrições textuais ou declarações formais sobre elementos e propriedades. Como exemplo, considere A o conjunto dos números inteiros maiores que zero e menores que oito. Também pode-se descrever tal conjunto como $A = \{1, 2, 3, 4, 5, 6, 7\}$, ou ainda: $A = \{x | x \in \mathbb{Z}\ and\ 0 < x < 8\}$. A Figura 2.1 contém uma possível representação visual em diagrama de Venn para o conjunto A e um outro conjunto B, além da conter uma operação de intersecção para os dois conjuntos.

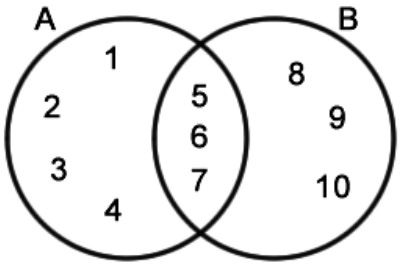

Figura 2.1: Representação Visual de Conjuntos

Para expressar o relacionamento entre elementos e conjuntos, algumas operações são utilizadas:

- **Pertence e Não Pertence ($\in$ e $\notin$)**: são operações básicas, denominadas "pertence" e "não pertence". Diz-se que um elemento x pertence a um conjunto S com a declaração $x \in S$. A negação desta declaração, seria $x \notin S$. De acordo com a Figura 2.1, pode-se dizer que $1 \in A$, $9 \notin A$, $2 \notin B$, $5 \in A$ e $5 \in B$. Todas as demais operações sobre conjuntos são derivadas a partir desta operação básica.

- **União ($\cup$)**: Serve para unir elementos de dois ou mais conjuntos, ou seja, $R \cup S = \{x | x \in R \ or \ x \in S\}$. Por definição, estão inclusos os elementos que pertencem simultaneamente a ambos os conjuntos envolvidos. A união é simétrica: $R \cup S = S \cup R$. Para os conjuntos A e B representados na Figura 2.1, $A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

- **Intersecção ($\cap$)**: para dois conjuntos conterá todos os elementos que pertencem simultaneamente a R e a S, ou seja, $R \cap S = \{x | x \in R \ and \ x \in S\}$. A intersecção é simétrica: $R \cap S = S \cap R$. Para os conjuntos A e B representados na Figura 2.1, $A \cap B = \{5, 6, 7\}$.

- **Diferença ($\backslash$)**: Para dois conjuntos R e S conterá os elementos que pertencem R e não pertencem a S, ou seja, $R \backslash S = \{x | x \in R \ and \ x \notin S\}$. A diferença não é simétrica: $R \backslash S \neq S \backslash R$. Para os conjuntos A e B representados na Figura 2.1, $A \backslash B = \{1, 2, 3, 4\}$.

- **Diferença Simétrica ($\ominus$)**: É um conjunto que contém os elementos que pertencem a R ou a S, mas não pertencem a ambos. Pode ser definida na forma $R \ominus S = (R \cup S) \backslash (R \cap S)$. Como o nome sugere, a simetria é respeitada: $R \ominus S = S \ominus R$. Para os conjuntos A e B representados na Figura 2.1, $A \ominus B = \{1, 2, 3, 4, 8, 9, 10\}$.

2.2.1 Paradoxo de Russell

Uma descrição de conjunto na forma $P = \{x | x \notin P\}$, perfeitamente válida de acordo com a teoria dos conjuntos, implica em um paradoxo. Tal situação foi questionada por Bertrand Russell em 1901, ficando conhecida como Paradoxo de Russell.

Nesta aula foram citados problemas desta natureza já em textos gregos, como o seguinte exemplo: seja uma cidade onde todos devem estar barbeados e, para tal, há um barbeiro que faz a barba de todos que não podem se barbear. A falha surge quando se considera que o barbeiro deve fazer sua própria barba: ele estaria fazendo a barba de alguém que pode se barbear. Se ele pode se barbear, ele (o barbeiro) não deveria estar fazendo a própria barba.

Para evitar paradoxo na descrição de conjuntos, deve-se tomar cuidado quando há necessidade de incluir o próprio conjunto em sua definição, implicando em recursividade. Uma separação entre conjuntos e categorias pode ser considerada, impedindo o uso de conjuntos na definição de outros conjuntos, permitindo apenas o uso de categorias para este fim.

2.2.2 Representação em Diagramas

O **Diagrama de Venn** é uma ferramenta útil para a representação gráfica de conjuntos e suas relações. Porém seu uso é limitado a uma quantidade pequena de conjuntos, pois o espaço necessário cresce muito rapidamente: para n conjuntos são necessárias 2^n zonas, correspondentes a todas as combinações de exclusão e inclusão para elementos em cada um dos conjuntos.

Para exemplificar, na Figura 2.2 estão representados diagramas com relações entre dois (a), três (b) e quatro (c) conjuntos, tendo este último exigido grande atenção para elaboração e interpretação se comparado com os anteriores.

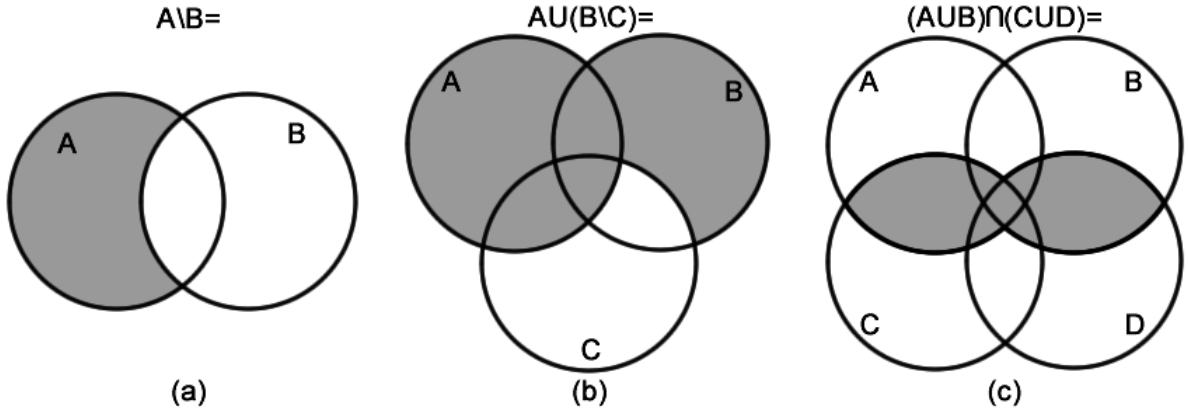

Figura 2.2: Exemplos de Diagrama de Venn

Há ainda o recurso da utilização de mais dimensões para viabilizar a representação de relações entre um número maior de conjuntos. No entanto, o aumento de complexidade inerente a essa solução limita seu uso para situações onde pode-se contar com ferramentas de representação, sem resolver ainda as dificuldades de interpretação. Em discussão, concluiu-se que um máximo de quatro conjuntos ainda gera representações bem compreensíveis em duas dimensões.

2.3 Lógica

O propósito do uso da lógica em Ciência da Computação é desenvolver linguagens para modelar situações de maneira que seja possível racionar sobre eles formalmente (Huth e Ryan, 2000). Lógica matemática permite expressar condições e declarações de grande complexidade, sem desconsiderar o rigor exigido por métodos formais. Tanto a lógica matemática fundamental como algumas de suas extensões são demasiadamente úteis para a especificação formal de software. Como fundamento para os métodos discutidos nesta nota didática, uma revisão de lógica de predicados, lógica modal e CTL são apresentados a seguir.

2.3.1 Lógica de Predicados

Na lógica proposicional não é possível expressar todos os tipos de argumentos formalmente, como exemplo, *Sócrates é homem. Todo homem é mortal. Logo, Sócrates é mortal.* Aparentemente esse argumento é válido, porém usando lógica proposicional a formalização desse argumento resulta em $p, q \models re$ não é possível mostrar que r é uma consequência lógica das premissas p e q. Uma razão para isso é a palavra *todo* que não pode ser expressa em lógica proposicional. Para tratar assuntos desse tipo é usada a lógica de predicados (Huth e Ryan, 2000).

A lógica de predicados é considerada a mais simples e fundamental. Baseia-se na definição de predicados, que são basicamente afirmações. Os predicados devem ser considerados verdadeiros ou falsos.

Para representar a validade de um predicado são utilizados os valores lógicos: 1 para verdadeiro, e 0 para falso. Sobre os predicados, pode-se aplicar os conectores/operadores lógicos. Sejam os predicados p e q, definem-se os seguintes operadores lógicos:

- $\neg$: *not*, negação
- $\wedge$: *and*, e – relacionado a intersecção
- $\vee$: *or*, ou – relacionado a união
- $\otimes$: *exclusive or*, ou exclusivo – relacionado a diferença simétrica

- $\Rightarrow$: *if*, implicação
- $\Leftrightarrow$: *iff*, bi-implicação

Os comportamentos desses operadores são exemplificados na Tabela 2.1. Em lógica, são utilizadas tabelas deste tipo para verificar a validade de predicados. Uma representação tabular com a finalidade descrita é chamada de **Tabela Verdade**.

Tabela 2.1: Tabela verdade para os operadores lógicos

		not	not	and/e	or/ou	xor	if	iff
p	q	$\neg p$	$\neg q$	$p \wedge q$	$p \vee q$	$p \otimes q$	$p \Rightarrow q$	$p \Leftrightarrow q$
0	0	1	1	0	0	0	1	1
0	1	1	0	0	1	1	1	0
1	0	0	1	0	1	1	0	0
1	1	0	0	1	1	0	1	1

Quando se lida com predicados relacionados por meio do operador de implicação ($\Rightarrow$), não é incomum que a abstração envolvida acabe por confundir a intuição de quem os interpreta. Como exemplo para essa situação temos: "Se o mordomo for o assassino, a mão dele estará suja de sangue" ($p \Rightarrow q$). É comum a quem interpreta aceitar que seja suficiente verificar a validade de q :"a mão está suja de sangue" para concluir que p:"mordomo é o assassino" seja verdadeiro. No entanto, a definição do operador de implicação garante que q seja verdadeiro ("a mão está suja de sangue") apenas se p("mordomo é o assassino") for verdadeiro. O fato de q ser verdadeiro não permite inferir qualquer conclusão sobre p, ao contrário do que sugere a intuição.

Para evitar os enganos que possam ser cometidos pela aplicação da intuição, é conveniente o uso da tabela verdade como uma ferramenta sistemática para analisar os predicados e suas implicações. A construção de tal tabela para o exemplo é apresentada na Tabela 2.2, considerando p=consoante e q=par.

Tabela 2.2: Tabela verdade para as cartas

p	q	$\neg p$	$\neg q$	$p \Rightarrow q$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

A lógica de predicados também inclui algumas regras de inferência, úteis para aplicar transformações e gerar conclusões e provas sobre predicados. Exemplos de algumas regras:

- $p, p \Rightarrow q \vdash q$ (*modus ponens*)
- $\neg q, p \Rightarrow q \vdash \neg p$ (*modus tollens*)
- $p \Rightarrow q, q \Rightarrow r \vdash p \Rightarrow r$ (*silogismo hipotético*)
- $\neg p, p \vee q \vdash q$ (*silogismo disjuntivo*)
- $p \vdash p \vee q$ (*simplificação*)
- $p \wedge q \vdash p$ (*simplificação*)
- $p \Rightarrow q \vdash \neg p \Rightarrow \neg q$

Para exemplificar a aplicação das regras de inferência, supanha que um indivíduo tenha perdido os seus óculos e as únicas informações que ele têm sobre os óculos são:

- (a)"se os óculos estão na cozinha, então eu os vi no café da manhã";
- (b)"eu estava lendo o jornal na sala de estar, ou eu estava lendo o jornal na cozinha";
- (c)"se eu estava lendo o jornal na sala de estar, então os óculos estão na mesa do café";
- (d)"eu não vi meus óculos no café da manhã";
- (e)"se eu estava lendo um livro na cama, os meus óculos estão no criado mudo";
- (f)"se eu estava lendo o jornal na cozinha, então os óculos estão na mesa da cozinha".

Tais informações podem ser tratadas como predicados e organizadas da seguinte maneira:

- p = óculos na mesa da cozinha
- q = óculos vistos no café da manhã
- (a) $p \Rightarrow q$ ("se os óculos estão na cozinha, então eu os vi no café da manhã")
- r = leitura do jornal na sala de estar
- s = leitura do jornal na cozinha
- (b) $r \vee s$ ("eu estava lendo o jornal na sala de estar, ou eu estava lendo o jornal na cozinha")
- (c) $r \Rightarrow t$ ("se eu estava lendo o jornal na sala de estar, então os óculos estão na mesa do café")
- t = óculos na mesa do café
- (d) $\neg q$ ("eu não vi meus óculos no café da manhã")
- u = leitura de livro na cama
- v = óculos no criado mudo
- (e) $u \Rightarrow v$ ("se eu estava lendo um livro na cama, os meus óculos estão no criado mudo")

- (f) $s \Rightarrow p$ ("se eu estava lendo o jornal na cozinha, então os óculos estão na mesa da cozinha")

Aplicando as regras de inferência sobre os predicados dados, têm-se que:

- (g) *modus tollens (a,d)*: $\neg q, p \Rightarrow q \vdash \neg p$
- (h) *modus tollens (f,g)*: $\neg p, s \Rightarrow p \vdash \neg s$
- (i) *silogismo disjuntivo (h,b)*: $\neg s, r \vee s \vdash r$
- (j) *modus ponens (i,c)*: $r, r \Rightarrow t \vdash t$

A última transformação, aplicada em (j), prova que o predicado t é verdadeiro, ou seja, os óculos estão na mesa do café.

2.3.2 Lógica de Primeira Ordem

A lógica de primeira ordem prevê a representação de características em elementos de conjuntos. Para esta finalidade, os predicados são apresentados como funções cujos domínios são os conjuntos que se pretende analisar. Como exemplo, para o conjunto S dos alunos matriculados na disciplina, pode-se representar um predicado P, sendo $P(x)$ verdadeiro pra todo aluno $x \in S$ que seja do sexo masculino, falso caso contrário.

O poder de expressão é complementado com o uso de **quantificadores**, representados por dois símbolos:

- $\forall$: universal – $\forall x \in S$, para todo elemento x do conjunto S
- $\exists$: existencial – $\exists y \in S$, existe ao menos um elemento y do conjunto S

Para ilustrar o uso dos quantificadores, juntamente com operadores lógicos, utilizou-se um silogismo da lógica aristotélica na seguinte forma: $\forall x \in A, P(x) \Rightarrow Q(x), y \in A, P(y) \vdash Q(y)$', descrito como o argumento clássico "todo homem é mortal, Sócrates é um homem, logo ele é mortal".

Tal como para a lógica de predicado, também há regras para a transformação de expressões em lógica de primeira ordem. A seguir estão relacionadas as regras apresentadas:

- $\forall x, P(x) \vdash \neg\exists\neg P(x)$
- $\exists x, P(x) \vdash \neg\forall\neg P(x)$
- $\forall x, P(x) \vdash P(y)$
- $P(y) \vdash \exists x, P(x)$
- $\forall x, P(x) \vdash \exists x, P(x)$

É importante ressaltar uma característica pouco intuitiva relacionada ao uso do operador lógico de implicação. Considere, por exemplo, duas expressões lógicas:

- $(a) \forall x \exists y, P(x, y) \Rightarrow \exists y \forall x, P(x, y)$ – falso
- $(b) \exists y \forall x, P(x, y) \Rightarrow \forall x \exists y, P(x, y)$ – verdadeiro

Embora pareça que ambas as expressões sejam semelhantes, essa impressão é incorreta. Apenas a segunda expressão é válida. Visando esclarecer essa diferença, ambas foram reescritas na forma de afirmações textuais equivalentes, sendo a expressão (a) representada como: "para todo bairro, existe um ônibus que passa por este bairro, isso implica que existe um ônibus que passa em todos os bairros". Quanto a expressão (b), escrita na forma: "há um ônibus que passa em todos os bairros, isso implica que em todos os bairros passa um ônibus", que soa adequado com a opinião das pessoas.

Além da lógica de primeira ordem, há também lógicas de segunda ordem, assim como suas extensões de ordens superiores. Nessas extensões o poder de representação é aumentado, juntamente com a complexidade das expressões, utilizando-se de recursos como a quantificação de predicados. Em razão do escopo desta nota didática tais lógicas não serão abordado em mais detalhes.

2.3.3 Lógica Modal

A lógica modal estende a lógica de predicados e a lógica de primeira ordem por meio de operadores que expressam modalidade. O poder de expressão introduzido por esses operadores é de grande utilidade quando se considera as necessidades envolvidas na formalização de especificações.

Em lógica de predicados, não é possível construir expressões que especulam sobre a validade de um predicado além da situação presente. A extensão modal disponibiliza esse recurso considerando as possibilidades em um estado futuro. Isso viabiliza a formalização e a avaliação de afirmações sobre condições ainda não estabelecidas.

A validade de predicados pode ser distinta em mundos distintos, sendo definidas transições possíveis entre tais mundos. A Figura 2.3 contém um diagrama com transições entre quatro mundos, nos quais os predicados p, q e r são ou não válidos. Os operadores modais são apresentados a seguir:

- $\Box$: indica necessidade – predicado deve valer todos os próximos mundos
- $\Diamond$: indica possibilidade – predicado deve valer em algum dos próximos mundos

As expressões são construídas na forma: $w \vdash \Box p$, sendo w o(s) mundo(s) a partir do(s) qual(is) o predicado p sempre será válido. Considerando as transições e predicados relacionados na Figura 2.3, alguns exemplos são apresentados:

- $1, 2 \vdash q$: q é verdadeiro nos mundos 1 e 2

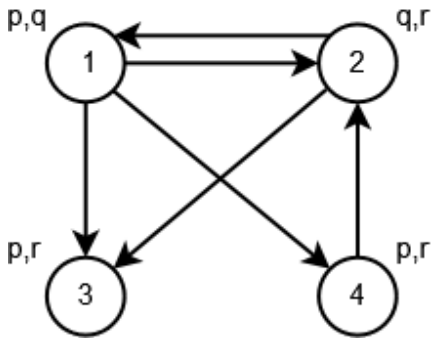

Figura 2.3: Transições em Lógica Modal

- $2 \nvdash p$: p não é verdadeiro no mundo 2
- $2 \vdash \Diamond p$: em algum mundo a partir de 2, p é verdadeiro
- $4 \nvdash \Diamond p$: em nenhum mundo a partir de 4, p é verdadeiro
- $3 \vdash \Box p$: em todos os mundos a partir de 3, p é verdadeiro
- $1 \vdash \Box r$: em todos os mundos a partir de 1, r é verdadeiro
- $1 \nvdash \Box p$: em algum mundo a partir de 1, p é verdadeiro
- $1 \nvdash \Box(\Diamond p)$: no futuro de algum mundo, dentre todos mundos que partem de 1, p não é verdadeiro
- $1 \vdash \Diamond(\Box p)$: em todos os futuros de algum mundo, dentre todos os mundos que partem de 1, p é verdadeiro

2.3.4 Lógica CTL

A lógica modal introduziu a possibilidade de avaliar predicados em situações imediatamente posteriores ao presente, ou seja, analisa-se um número limitado de passos até que as condições sejam ou não validadas. A *Computation Tree Logic* (CTL) amplia essa análise, incluindo no conjunto de ferramentas lógicas os operadores temporais. Nesta modalidade, condições podem ser avaliadas em qualquer tempo futuro, contemplando ainda a sequência em que as condições são satisfeitas.

Para esse objetivo, as transições entre os estados são consideradas caminhos, incluindo a possibilidade de laços. A partir destes caminhos, podem ser formuladas expressões que verificam a validade de propriedades em diversos aspectos, com o uso das seguintes ferramentas:

- Quantificadores:
  - $E\phi$: Existe $(\exists, \Diamond)$ pelo menos um caminho onde ϕ vale
  - $A\phi$: ϕ vale em todos (All) os caminhos $(\forall, \Box)$
- Operadores Temporais:

- $X\phi$: ϕ vale no próximo ($neXt$) estado
- $G\phi$: ϕ vale em todo o caminho subsequente ($Globally$)
- $F\phi$: ϕ eventualmente ($Future$) tem que valer
- $\phi U\psi$: ϕ deve valer até que ($Until$) ψ seja válido

Considerando o predicado p = "tem aula amanhã", seguem exemplos de algumas propriedades em CTL:

- $AX(p)$ – "com certeza vai ter aula amanhã"
- $EX(p)$ – "pode ter aula amanhã"
- $AX(\neg p)$ – "com certeza não vai ter aula amanhã"
- $\neg AX(p)$ – "não é verdade que é certeza que amanhã terá aula"
- $EX(\neg p)$ – "pode não ter aula amanhã"
- $AF(p)$ – "certamente um dia haverá aula"
- $\neg EX(AF(\neg p))$ – "não é possível que certamente nunca terá aula"

2.4 Considerações Finais

Neste capítulo apresentamos um resumo de teoria dos conjuntos e algumas lógicas relevantes no estudo de métodos formais. Dentre elas, encontra-se a técnica temporal CTL. No próximo capítulo será apresentada a utilização de expressões CTL na aplicação de um método formal – o *model checking*.

Capítulo

3

Model Checking

3.1 Considerações Iniciais

Model Checking é um método formal usado para verificar modelos de estados finitos em que as expressões sobre os sistemas são descritas usando fórmulas em lógica temporal. Para percorrer o modelo definido para o sistema e checar se a especificação é válida ou não são usados algoritmos. Essa técnica tem sido muito utilizada para especificar sistemas críticos que precisam de mais rigor em seu desenvolvimento. Neste capítulo são apresentados os algoritmos de *model cheking* e ferramentas que facilitam sua utilização.

3.2 Algoritmos para Model Checking

Considerando os conectivos temporais (AX, EX, AG, EG, AU, EU, AF e EF) e outros operadores lógicos ($\bot$, $\top$, $\neg$, $\vee$, $\wedge$, $\longrightarrow$) pode-se resumir esse conjunto a cinco operadores mais básicos:

- $\neg$
- $\vee$
- EX
- EG

- *EU*

Para simplificar a solução de fórmulas CTL por meio de algoritmos de *model cheking*, os conectivos e operadores que não são os cinco apresentados são convertidos por fórmulas equivalentes. Duas fórmulas CTL podem ser consideradas equivalentes quando cada fórmula sendo aplicada a um modelo é possivel satisfazê-la, obtendo o mesmo resultado final para ambas. Assim, tem-se as sequintes equivalências para converter fórmulas CTL nos 5 operadores básicos acima relacionados:

1. EF p = E [True $\bigcup$ p]
2. AF p = ¬EG¬p
3. AG p = ¬EF¬p = ¬E [True $\bigcup$ ¬p]
4. AX p = ¬EX¬p
5. A[p1 $\bigcup$ p2] = ¬E [¬p2 $\bigcup$ (¬p1 $\wedge$ ¬p2)] $\wedge$ ¬EG(¬p2)
   = ¬(E [¬p2 $\bigcup$ ¬ (p1 $\vee$ p2)] $\vee$ EG (¬p2))
6. p $\longrightarrow$ q = ¬p $\vee$ ¬q

Utilizando o modelo descrito na Figura 3.1 algumas fórmulas CTL podem ser avaliadas, seguem-se alguns exemplos:

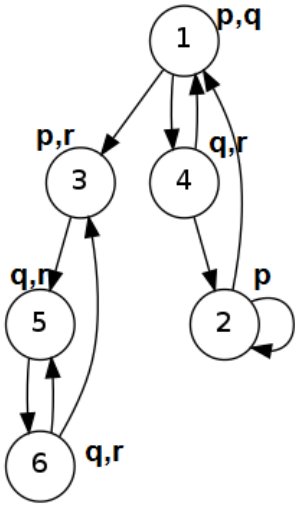

Figura 3.1: Modelo 1

Exemplo 1: EX(r)

O primeiro passo para resolver essa propriedade é escrevê-la na forma mais simples usando as equivalências descritas anteriormente. Essa conversão é utilizada para que a análise de qualquer fórmula seja feita utilizando apenas os algoritmos para cada um dos operadores anteriormente relacionados. Para a fórmula do Exemplo 1 não há necessidade desse passo, pois ela já está na forma mais simples. Pode-se então continuar a resolução da fórmula para o modelo sugerido na Figura 3.1 com o auxílio de uma árvore formada a partir da fórmula CTL, a qual está ilustrada na Figura 3.2.

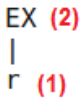

Figura 3.2: Decomposição da fórmula CTL 1

Para a construção dessa árvore, a fórmula CTL é desconstruída utilizando principios de árvore binária até a menor unidade que a forma. Em seguida os nós da árvore são rotulados, considerando uma ordem crescente, das folhas para a raíz. Para o Exemplo 1 considere agora essa árvore (Figura 3.2) e o modelo especificado na Figura 3.1, seguindo-se os passos:

1. Todos os estados do modelo que possuem a letra **r** são rotulados com (1) seguindo a numeração assumida na árvore descrita para a expressão CTL;

2. Todos estados que possuem caminhos para um próximo estado rotulado com (1) são rotulados com (2).

Após esse passo, obtém-se o modelo da Figura 3.3 que apresenta como solução para o exemplo 1 os estados **1, 3, 5, 6**.

Exemplo 2: EG(p)

Para solução do Exemplo 2 também não há necessidade de substituição dos operadores, pois estes já estão na forma mais simples. A Figura 3.4 apresenta a árvore CTL para essa fórmula.

Considerando então a árvore da Figura 3.4, o modelo da Figura 3.1 é rotulado considerando os seguintes passos:

1. Todos os estados que possuem a letra **p** são rotulados com (1);

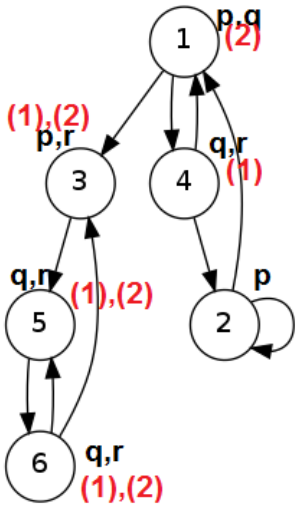

Figura 3.3: Modelo solução para o exemplo 1

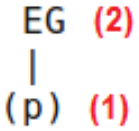

Figura 3.4: Decomposição da fórmula CTL 2

2. Todos os estados dentro de um laço, em que todos os estados têm rótulo (1), são rotulados com (2).

Dessa forma, após a rotulação do modelo sugerido, obtém-se o modelo especificado na Figura 3.5, o qual apresenta como resposta ao Exemplo 2 o estado **2**.

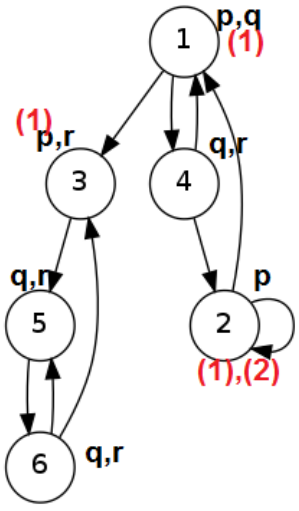

Figura 3.5: Modelo solução para o exemplo 2

Exemplo 3: E[q ⋃ r]

Para este exemplo também não há necessidade de substituição de operadores. A Figura 3.6 apresenta a árvore para a fórmula do Exemplo 2.

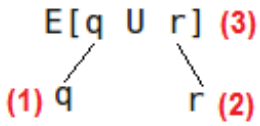

Figura 3.6: Decomposição da fórmula CTL 3

Considerando agora a árvore da Figura 3.6 e o modelo especificado na Figura 3.1 são necessários os seguintes passos para solução da propriedade:

1. Os estados no modelo que possuem a letra **q** são rotulados com (1);
2. Os estados que possuem a letra **r** são rotulados com (2);
3. Todos os estados que estão rotulados com (2) recebem o rótulo (3);
4. Todos os estados que tem o rótulo (1) e possuem caminhos para estados com rótulo (3) também são rotulados com (3).

Após esses passos o modelo da Figura 3.7 é obtido, apresentando como solução para o Exemplo 3 os estados: **1,3,4,5,6**.

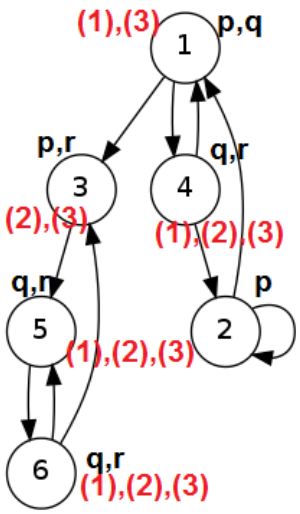

Figura 3.7: Modelo solução para o exemplo 3

Exemplo 4: EX(EGr)

Como nos exemplos anteriores, o primeiro passo para encontrar a solução para a fórmula é verificar se esta é constituída apenas pelos 5 operadores mais básicos, neste caso sim. O segundo passo é construir a árvore para a propriedade, a qual pode ser visualizada na Figura 3.8.

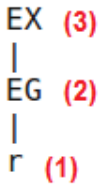

Figura 3.8: Árvore para o exemplo 4

Após construção da árvore, o modelo da Figura 3.1 é alterado considerando os rótulos inseridos na árvore para a fórmula especificada na Figura 3.8, da seguinte forma:

1. Todos os estados no modelo que possuem a letra **r** são rotulados com (1);
2. Todos os estados dentro de um laço, em que todos os estados têm rótulo (1), são rotulados com (2);
3. Todos os estados que possuem caminhos para estados com rótulo (2) são rotulados com (3).

Após essa operação obtém-se o modelo da Figura 3.9 com a resposta para a fórmula do Exemplo 4, que são os estados: **1,3,5,6**.

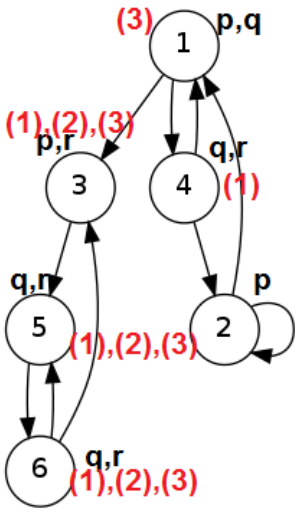

Figura 3.9: Modelo solução para o exemplo 4

Exemplo 5: EF(AG(q))

Esse exemplo possui o operador EF que precisará ser substituído. Usando a regra de equivalência, em que EF(p) = E[True ⋃ p], tem-se:

EF(AG(q)) = E[True ⋃ AG(q)], agora o AG precisa ser substituído, usando $AG(p) = \neg EF(\neg p)$. Assim a fórmula é equivalente a **E[True ⋃ ¬E[True ⋃ ¬q]]**

Neste passo, a fórmula está na forma mais simples e é construída a árvore CTL ilustrada na Figura 3.10.

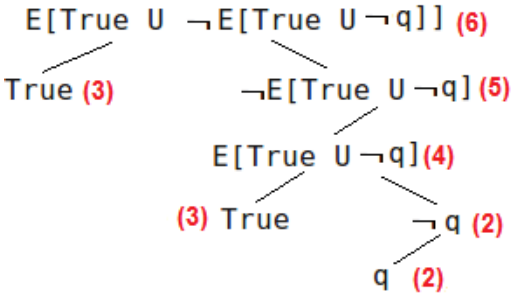

Figura 3.10: Decomposição da fórmula do exemplo 5

Com a árvore, ao modelo da Figura 3.1 são adicionados os rótulos correspondentes a cada estado, com os seguintes passos:

1. Todos os estados que possuem o q são rotulados com (1);
2. Todos estados que não tem rótulos (1) são rotulados com (2);
3. Todos os estados são rotulados com (3);
4. Todos os estados que possuem o rótulo (2) são rotulados com (4);
5. Todos os estados que tem o rótulo (3) e possuem caminhos para estados com rótulo (4) também são rotulados com (4);
6. Todos os estados que não tem o rótulo (4) são rotulados com (5);
7. Todos os estados que possuem o rótulo (5) são rotulados com (6);
8. Todos os estados que tem o rótulo (3) e possuem caminhos para estados rotulados com (6) também são rotulados com (6).

Com esses passos é obtido o modelo da Figura 3.11, o qual apresenta como resposta para a fórmula do Exemplo 5 os estados: **2,3,4,5,6**.

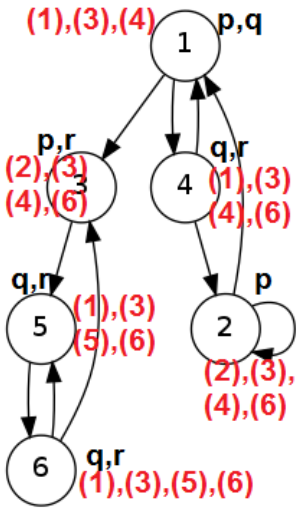

Figura 3.11: Modelo solução para o exemplo 5

Exemplo 6: E(p ⋃ AG(r))

O primeiro passo para a solução é verificar se a fórmula está na forma mais simples, nesse caso ela não está e o operador AG será substituído da seguinte forma:

$E(p \bigcup AG(r)) = E(p \bigcup \neg E[True \bigcup \neg r])$

O segundo passo é construir a árvore para a fórmula, nesse caso esta pode ser visualizada na Figura 3.14.

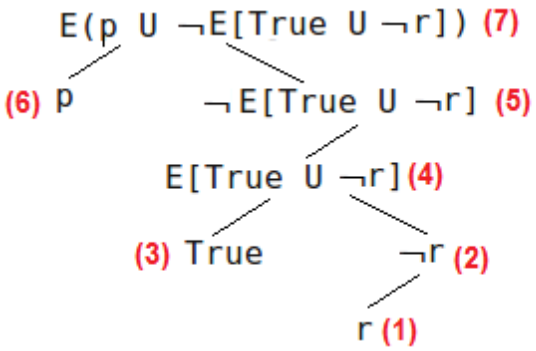

Figura 3.12: Decomposição da fórmula CTL do exemplo 6

Para chegar à solução do Exemplo 6 o modelo da Figura 3.1 é alterado considerando os rótulos da árvore da Figura 3.14 e os passos:

1. Todos os estados que possuem o r são rotulados com (1);
2. Todos os estados que não possuem o rótulo (1) são rotulados com (2);
3. Todos os estados são rotulados com (3);

4. Todos os estados rotulados com (2) recebem o rótulo (4);
5. Todos os estados que possuem o rótulo (3) e possuem caminho para estados com rótulos (4) recebem também o rótulo (4);
6. Todos os estados que não possuem o rótulo (4) recebem o rótulo (5);
7. Todos os estados rotulados com (5) recebem o rótulo (7);
8. Todos os estados rotulados com (6) e que possuem caminhos para estados com o rótulo (7) também são rotulados com (7).

Após esses passos obtém-se o modelo apresentado na Figura 3.15, o qual apresenta como solução para o Exemplo 6 os estados: **1,2,3,5,6**.

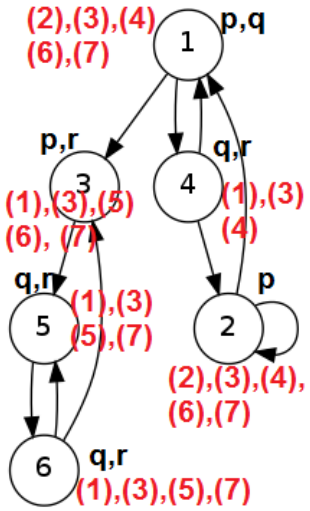

Figura 3.13: Modelo solução para o exemplo 6

Exemplo 7: A[p ⋃ r]

Para solução dessa fórmula, inicialmente, deve-se fazer as substituições necessárias considerando as regras de equivalência e os operadores mais simples especificados no início da aula:

A [p ⋃ r] = ¬[E [¬r ⋃ ¬(p ∨ r)] ∨ EG(¬r)]

Com a fórmula na forma mais simples, deve-se construir a árvore para a propriedade, a qual pode ser visualizada na Figura 3.14.

Após a construção da árvore, cada estado do grafo ilustrado na Figura 3.1 será rotulado com base na árvore CTL da Figura 3.14 da seguinte forma:

1. Todos os estados que possuem r são rotulados com (1);

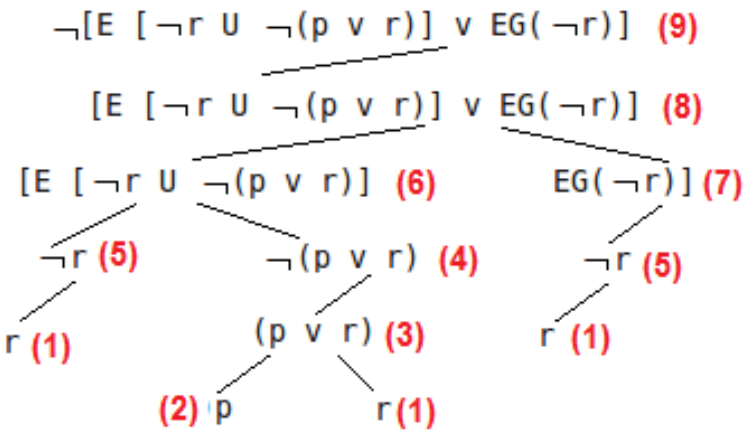

Figura 3.14: Decomposição da fórmula CTL do exemplo 7

2. Todos os estados que possuem p são rotulados com (2);

3. Todos os estados que possuem rótulos (1) ou (2) são rotulados com (3);

4. Todos os estados que não possuem rótulo (3) são rotulados com (4);

5. Todos os estados que não possuem rótulo (1) são rotulados com (5);

6. Todos os estados rotulados com (4) são rotulados com (6);

7. Todos os estados rótulados com (5) e possuem caminhos que levam a estados rotulados com (6) também são rotulados com (6);

8. Todos os estados dentro de um laço, em que todos os estados têm rótulo (5), são rotulados com (7);

9. Todos estados que possuem o rótulo (7) são rotulados com (8);

10. Todos os estados que possuem o rótulo (6) e que possuem caminhos para estados com rótulo (8) também são rotulados com (8);

11. Todos os estados que não possuem o rótulo (8) são rotulados com (9).

Seguindo-se esses passos obtém-se o modelo ilustrado na Figura 3.15 que apresenta como solução à propriedade do Exemplo 7 o estado rotulado com (9), ou seja os estados **1,3,4,5 e 6**.

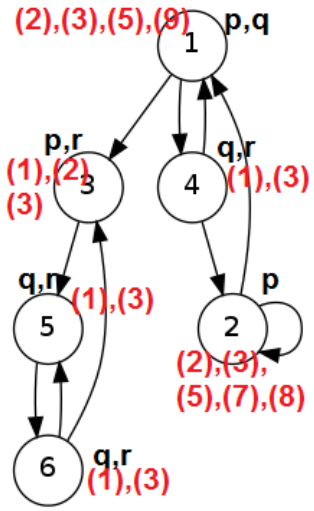

Figura 3.15: Modelo solução para o exemplo 7

Exemplo 8: E[q ⋃ (EG(p) ∨ AG(r))]

Primeiro passo para a solução do Exemplo 8 é a substituição dos operadores para a forma mais básica:

E[q ⋃ (EG(p) ∨ AG(r))] ≡ E [q ⋃ (EG(p) ∨ ¬E[True ⋃ ¬r])]

Construindo a árvore para a propriedade CTL, obtém-se a Figura 3.16. Para a solução dessa propriedade, os estados do modelo apresentados na Figura 3.1 são rotulados considerando a árvore da Figura 3.16, da seguinte forma:

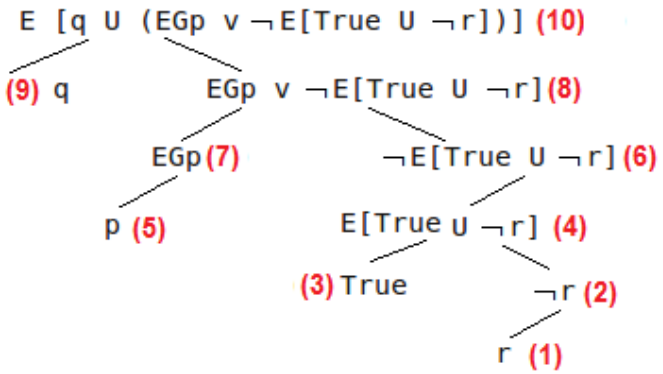

Figura 3.16: Árvore para a fórmula do exemplo 8

1. Todos os estados do modelo que possuem r são rotulados com (1);
2. Todos os estados que não possuem r são rotulados com (2);
3. Todos os estados são rotulados com (3);
4. Todos os estados que possuem rótulo (2) são rotulados com (4);

5. Todos os estados que possuem rótulo (3) e que possuem caminhos para estados os rotulados com (4) também são rotulados com (4);

6. Todos os estados que possuem p são rotulados com (5);

7. Todos os estados que não possuem rótulo (4) são rotulados com (6);

8. Todos os estados dentro de um laço, em que todos os estados têm rótulo (5), são rotulados com (7);

9. Todos estados que possuem rótulo (6) ou (7) são rotulados com (8);

10. Todos estados que possuem q são rotulados com (9);

11. Todos estados que possuem rótulo (8) são rotulados com (10);

12. Todos estados que possuem rótulo (9) e possuem caminhos para estados com rótulo (10) também são rotulados com (10).

Com base nesses passos, o modelo ilustrado na Figura 3.17 é obtido, apresentando como solução para a propriedade os estados **1,2,3,4,5,6**.

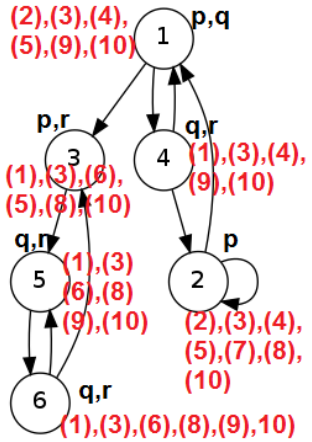

Figura 3.17: Modelo solução para o exemplo 8

Exemplo 9: AG(p $\longrightarrow$ AF(p))

Para solucionar esse exemplo inicia-se com a substituição dos operadores para a forma mais básica, em que se tem:

AG(p $\longrightarrow$ AF(p)) $\equiv \neg$E [True $\bigcup \neg$p (p $\longrightarrow$ AF(p))]
$\equiv \neg$E [True $\bigcup \neg$(p $\longrightarrow \neg$EG$\neg$p)]
$\equiv \neg$E [True $\bigcup \neg(\neg p \vee \neg$EG$\neg$p)]

Com a fórmula composta apenas pelos operadores mais básicos, a árvore CTL é construída e pode ser visualizada na Figura 3.18.

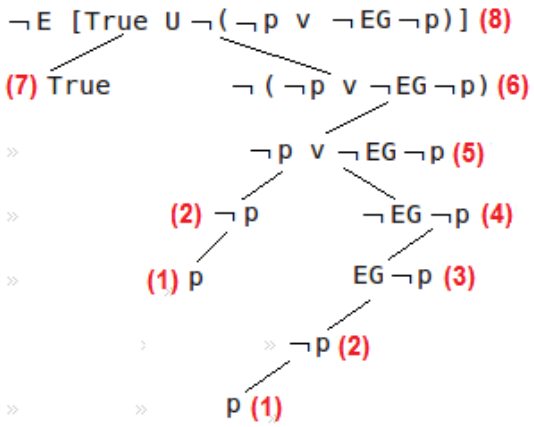

Figura 3.18: Árvore para a fórmula do exemplo 9

O modelo da Figura 3.1 será alterado considerando a a árvore da Figura 3.18 da seguinte forma:

1. Todos os estados no modelo que possuem p são rotulados com (1);
2. Todos estados que não possuem o rótulo (1) são rotulados com (2);
3. Todos os estados dentro de um laço, em que todos os estados têm rótulo (2), são rotulados com (3);
4. Todos estados que não possuem o rótulo (3) são rotulados com (4);
5. Todos estados que possuem o rótulo (2) ou o rótulo (4) são rotulados com (5);
6. Todos estados que não possuem o rótulo (5) são rotulados com (6);
7. Todos estados que possuem o rótulo (6) são rotulados com (8);
8. Todos estados que possuem o rótulo (7) e possuem caminhos para estados com rótulo (8) também são rotulados com (8);
9. Todos estados que não possuem o rótulo (8) são rotulados com (9).

A Figura 3.19 apresenta a solução para o exemplo, que é o conjunto de estados: **1,2,3,4,5 e 6**.

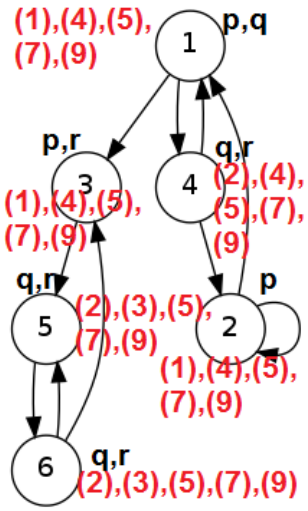

Figura 3.19: Modelo solução para o exemplo 9

3.2.1 Complexidade dos Algoritmos

Após os exemplos da Seção 5.2 conclui-se que são necessários basicamente 5 algoritmos para analisar uma propriedade qualquer. Considerando um determinado modelo de estados, inicialmente a propriedade deve ser escrita usando os 5 operadores básicos vistos na Seção 5.2, em seguida deve-se percorrer o modelo de estados e verificar se a propriedade é válida. Mas quais as complexidades desses algoritmos?

Para analisar a complexidade, tem-se:

$$\neg = O(n)$$

$$\vee = O(n)$$

$$EX = O(n_K)$$

em que k é a média de números de próximos nós.

Se m é o número de arestas e n é o número de nós, tem-se:

$$k = m/n$$

$$EG = O(mn)$$

$$EU = O(mn)$$

Assim, dada uma fórmula é possível decompô-la num número limitado de nós, exceto pelo AU que é linear na quantidade de símbolos possíveis de serem utilizados.

Curiosidade: Um pesquisador conseguiu enumerar explicitamente um grafo com 10^{18} estados, porém implicitamente foi conseguido enumerar um grafo com 10^{120} estados.

3.3 Model Checkers

Model checkers são ferramentas que implementam os algoritmos apresentados na seção anterior, podendo automaticamente verificar se uma especificação satisfaz determinadas propriedades.

Um dos *model checkers* muito utilizados atualmente é o UPPAAL[1]. O nome do *model checker* é uma combinação das siglas das duas universidades que desenvolveram a ferramenta – *Uppsala Universitet* e *Aalborg University*. A Figura 3.20 apresenta a interface do UPPAAL. Em linhas gerais, por meio da interface da ferramenta é possível representar máquinas de estados, cujas transições são rotuladas conforme a notação proposta. Em seguida, é possível expressar propriedades em CTL e verificar se tais expressões são válidas ou não.

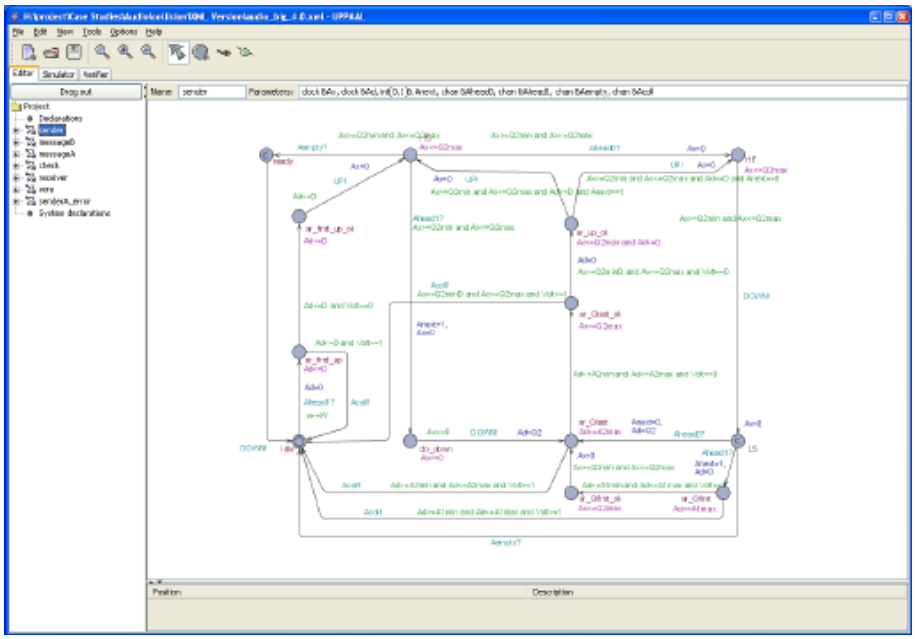

Figura 3.20: Interface do UPPAAL

Uma ferramenta interessante que possui aplicação na indústria é a UPPAAL TRON (Figura 3.21)[2]. Baseada na UPPAAL, ela apoia o teste de conformidade em sistemas de tempo real. Os testes são realizados *online*, ou seja, durante a execução do software, sem um projeto prévio de casos de teste.

Outro *model checker* é o SPIN[3]. Uma das suas principais características é o fato dele utilizar para especificar as máquinas de estado, a linguagem PROMELA, que por sua vez é muito próxima da linguagem C.

[1] disponível em `http://www.uppaal.org/`
[2] disponível em `http://people.cs.aau.dk/~marius/tron/`
[3] disponível em `http://spinroot.com/spin/whatispin.html`

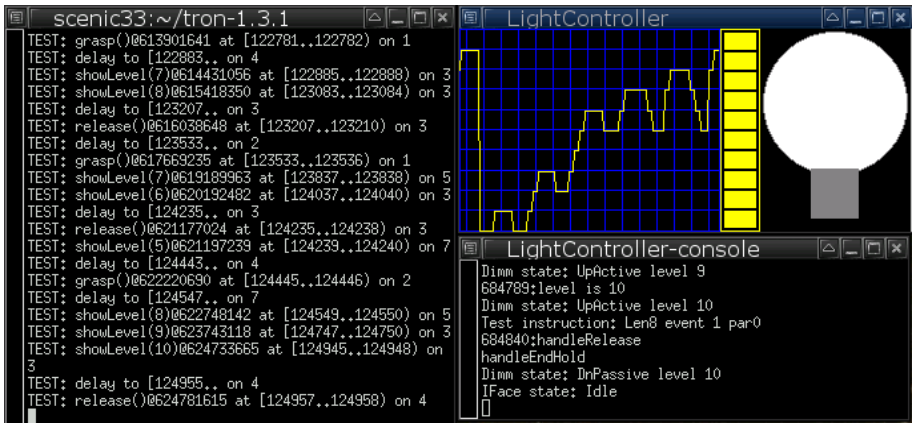

Figura 3.21: Interface do UPPAAL TRON

Embora a linguagem PROMELA seja parecida com C, existem algumas diferenças. Como exemplo, em um `if` da linguagem C, as opções são mutuamente exclusivas, ou seja, ou consideram-se os comandos dentro do bloco do `if` ou os comandos dentro do bloco do `else`. A Figura 3.22 mostra como escrever um `if` em PROMELA. No exemplo: (1) se somente `A` for verdade, a opção `option1` será considerada; (2) se somente `B` for verdade, a opção `option2` será considerada; e (3) se `A` e `B` forem verdade, ambas opções `option1` e `option2` são consideradas, ou seja, `option1` e `option2` não são mutuamente exclusivas.

```
if
:: (A == true) -> option1;
:: (B == true) -> option2;
fi
```

Figura 3.22: Exemplo de Código em PROMELA

Nesse sentido, a vantagem em utilizar o SPIN é a possibilidade de traduzir um código C ou JAVA em PROMELA para posteriormente aplicar o *Model Checker*. Uma das principais dificuldades é o fato de que os tipos de dados em PROMELA são mais limitados do que os de uma linguagem de alto nível. Outra dificuldade é que não é possível expressar uma propriedade diretamente utilizando CTL, uma vez que as propriedades também devem ser especificadas com um processo PROMELA.

O terceiro *model checker* a ser considerado é o NUSMV. O NUSMV é a reimplementação e extensão do SMV – o primeiro *model checker* baseado em Diagramas de Decisão Binária (*Binary Decision Diagrams* – BDDs). A ferramenta tem sido projetada com uma arquitetura aberta e visa a verificação confiável de projetos industriamente grandes, para uso como *backend* de outras ferramentas de verificação ou como ferramenta de pesquisa para técnicas de verificação formal.

O NuSMV tem sido desenvolvido como um projeto conjunto entre ITC-IRST (*Istituto Trentino di Cultura* emn Trento, Itália), *Carnegie Mellon University*, *University of Genoa* e *University of Trento*.

O UPPAAL, o Spin e o NuSMV são os três principais *model checkers* existentes e eles representam três paradigmas diferentes de especificação de uma máquina de estados. Dos três *model checkers*, o NuSMV é considerado o mais direto e intuitivo. Por esse motivo exemplos de uso do NuSMV serão descritos em mais detalhes a seguir.

3.3.1 Especificação de uma Máquina de Estados no NuSMV

A Figura 3.23 apresenta a máquina de estados considerada no primeiro exemplo de especificação no NuSMV.

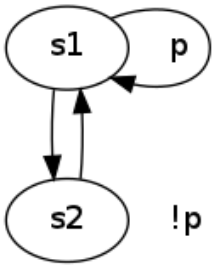

Figura 3.23: Exemplo 1: Máquina de Estados

O código da Figura 3.24 mostra a especificação da máquina na linguagem do NuSMV. Na linha 3 são definidos os dois estados da máquina (`s1` e `s2`). Na linha 5, o estado `s1` é definido como o estado inicial. Nas linhas 6 à 9 são definidas as transições entre os estados:

1. caso o estado atual for igual a `s1` (`st = s1`), o estado seguinte (`next(st)`) pode ser tanto `s1` como `s2` (`{s1,s2}`);

2. caso o estado atual for igual a `s2` (`st = s2`), o estado seguinte (`next(st)`) será `s1`.

Por fim, na linha 11 é definida a propriedade `p` que só é verdade no estado `s1` (ou seja, quando "`st = s1`").

O comando da Figura 3.25 deve ser considerado para executar o NuSMV com a máquina especificada.

O resultado deve ser semelhante ao exemplificado na Figura 3.26.

Para carregar a máquina, deve-se executar o comando da Figura 3.27. Durante a execução do comando, o NuSMV verifica a análise sintática da especificação. Se nenhum erro ocorrer, significa que a especificação está sintaticamente correta.

```
MODULE main
VAR
    st : {s1, s2};
ASSIGN
    init(st) := s1;
    next(st) := case
        st = s1 : {s1, s2};
        st = s2 : s1;
    esac;
DEFINE
    p := st = s1;
```

Figura 3.24: Exemplo 1: Especificação da Maquina de Estados

```
NuSMV -int <nome do arquivo>
```

Figura 3.25: NuSMV: Comando para executar o NuSMV (Modo Iterativo)

```
*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 13:47:30 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

NuSMV >
```

Figura 3.26: NuSMV: Saída Inicial do NuSMV

```
NuSMV > go
```

Figura 3.27: NuSMV: Comando para Iniciar a Execução da Máquina de Estados

```
NuSMV > pick_state -i
```

Figura 3.28: NuSMV: Comando para Escolher um Estado Inicial

Em seguida, é necessário escolher um estado inicial. Para isso, deve-se executar o comando descrito na Figura 3.28.

O resultado deve ser semelhante ao mostrado na Figura 3.29. Como no exemplo só foi definido o estado `s1` como estado inicial, somente ele estará disponível e será automaticamente escolhido quando pressionado `<Enter>`.

```
********** AVAILABLE STATES **********

========== State ==========
0) -------------------------
p = TRUE
st = s1


There's only one available state. Press Return to Proceed.

Chosen state is: 0
```

Figura 3.29: NuSMV: Estados Iniciais Disponíveis

Após definir o estado inicial, pode-se simular iterações na máquina de estados por meio do comando da Figura 3.30.

```
NuSMV > simulate -i 1
```

Figura 3.30: NuSMV: Comando para Simular uma Iteração na Máquina de Estados

O NuSMV deve apresentar os possíveis estados seguintes ao estado atual, conforme ilustrado na Figura 3.31. No exemplo, os possíveis estados seguintes são: (1) o próprio estado `s1`, onde `p` é verdade; e (2) o estado `s2`, em que `p` não é verdade. Ao digitar o identificador de um dos estados e pressionar `<Enter>`, realiza-se a transição do estado atual para o estado seguinte escolhido. No exemplo, realiza-se a transição do estado `s1` para o estado `s2`.

Pode-se repetir o passo anterior várias vezes para realizar diversas transições. Além disso, a qualquer momento é possível verificar quais estados são passados ao longo das transições por meio do comando descrito na Figura 3.32.

A Figura 3.33 mostra as transições realizadas para este exemplo. Inicialmente, o estado atual era o `s1`, em que `p` é verdade. Em seguida, realiza-se uma transição para o estado `s2`, em que `p` não é verdade.

Por fim, pode-se verificar a validade de uma propriedade em `CLT` por meio do comando da Figura 3.34. No exemplo, a propriedade "`EG(p)`" é verificada.

O resultado deve ser semelhante ao ilustrado na Figura 3.35.

Para sair do NuSMV, deve-se digitar o comando descrito na Figura 3.36.

```
******** Simulation Starting From State 2.1 ********

************** AVAILABLE STATES *************

-------------------- State --------------------
0) ---------------------------
p = FALSE
st = s2


-------------------- State --------------------
1) ---------------------------
p = TRUE
st = s1


Choose a state from the above (0-1): 0

Chosen state is: 0
```

Figura 3.31: NuSMV: Estados Seguintes Disponíveis

```
NuSMV > show_traces
```

Figura 3.32: NuSMV: Comando para Verificar as Transições Realizadas

```
<!-- ######################## Trace number: 2 ######################## -->
Trace Description: Simulation Trace
Trace Type: Simulation
-> State: 2.1 <-
    st = s1
    p = TRUE
-> State: 2.2 <-
    st = s2
    p = FALSE
```

Figura 3.33: NuSMV: Transições Realizadas

```
NuSMV > check_ctlspec -p EG(p)
```

Figura 3.34: NuSMV: Verificação de uma Propriedade em CTL

```
-- specification EG p  is true
```

Figura 3.35: NuSMV: Resultado da Validação de uma Propriedade em CTL

```
NuSMV > quit
```

Figura 3.36: NuSMV: Comando para encerrar a execução da ferramenta.

3.3.2 Definição de Propriedades na Especificação da Máquina

Uma forma mais fácil de validar propriedades CTL no NuSVM é definindo tais propriedades na própria especificação da máquina de estados. A Figura 3.37 ilustra a especificação da máquina definida na Seção 3.29. Na linha 12 é especificada a propriedade "`EG(p)`", verificada durante a execução da máquina no NuSMV. Na linha 13 é especificada a propriedade "`AG(!p -> AX(p))`".

```
1  MODULE main
2  VAR
3      st : {s1,s2};
4  ASSIGN
5      init(st) := s1;
6      next(st) := case
7          st = s1 : {s1,s2};
8          st = s2 : s1;
9      esac;
10 DEFINE
11     p := st = s1;
12 SPEC EG(p);
13 SPEC AG(!p -> AX(p));
```

Figura 3.37: Exemplo 2: Definição de Propriedades na Especificação da Máquina

Para verificar a validade dessas propriedades o NuSMV não deve ser executado no modo iterativo. Para isso, executa-se o NuSMV conforme o comando da Figura 3.38.

```
NuSMV <nome do arquivo>
```

Figura 3.38: NuSMV: Comando para executar o NuSMV

O resultado deve ser semelhante ao ilustrado na Figura 3.39. As linhas 16 e 17 exibem o resultado da validação das prorpiedades definidas. No exemplo, ambas são verdade.

3.3.3 Exercício de Fixação

Nesta seção é apresentado um exercício a fim de fixar os conteúdos vistos até o momento. O exercício consiste em representar a máquina de estados da Figura 3.40 como um modelo SMV e, em seguida, verificar se as seguintes propriedades são verdadeiras:

```
*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 13:47:30 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification EG p  is true
-- specification AG (!p -> AX p)  is true
```

Figura 3.39: NuSMV: Saída Inicial com Validação das Propriedades

- `AG( Start => AF Heat )`
- `A[ !Heat U Close ]`
- `AG( Error => AF !Error )`

A Figura 3.41 mostra um possível modelo SMV para a máquina, incluindo a especificação das propriedades a serem verificadas.

A Figura 3.42 mostra o resultado fornecido pelo NuSMV para a expressão `AG( Start => AF Heat )`. Tal expressão é `falsa`. Ao chegar no estado `s2`, `Start` se torna verdade. Segundo a expressão, `Start` sendo verdade implica que para todos os caminhos seguintes, em algum momento no futuro, `Heat` será verdade. No entanto, segundo o contra-exemplo fornecido pelo `NuSMV`, a máquina pode ficar infinitamente em *loop* entre os estados `s2` e `s5`, onde `Heat` é falso.

O resultado para a expressão `A[ !Heat U Close ]` fornecido pelo NuSMV encontra-se na Figura 3.43. Como é possível observar, a expressão é verdadeira.

Por fim, na Figura 3.44 pode-se visualizar o resultado fornecido pelo NuSMV para a expressão `AG( Error => AF !Error )`. Tal expressão é `falsa`. De maneira análoga para a expressão `AG( Start => AF Heat )`, quando a execução chega no estado `s2`, `Error` ela é verdadeira. Segundo a expressão, `Error` sendo verdade implica que para todos os caminhos seguintes, em algum momento no futuro, `Error` será falso. No entanto, segundo o contra-exemplo fornecido pelo `NuSMV`, a máquina pode ficar infinitamente em *loop* entre os estados `s2` e `s5`, em que `Error` é sempre verdade.

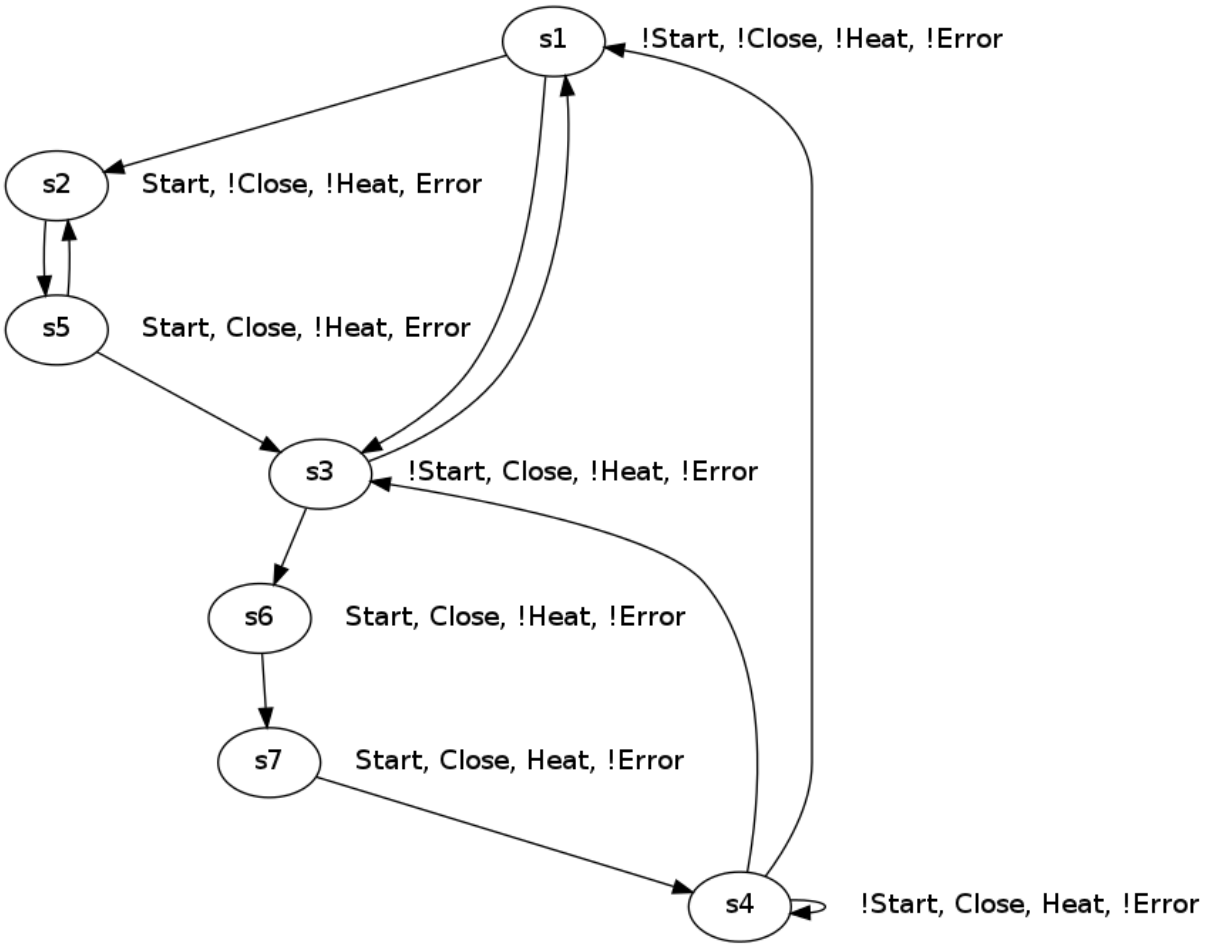

Figura 3.40: Exercício de Fixação: Máquina de Estados

3.3.4 Especificação de Máquinas com Muitos Estados

Uma estratégia que pode ser utilizada para representar os estados de uma máquina no NuSMV seria enumerar todos eles, conforme visto nas seções anteriores. No entanto, geralmente, máquinas com muitos estados possuem uma estrutura que pode ser explorada a fim de facilitar a especificação no modelo do NuSMV ou de outro *model checker*.

Considere a Figura 3.45 que ilustra um exemplo de máquina de estados utilizando outra forma de representação de máquinas. O objetivo foi reduzir a quantidade de estados necessários para representar uma máquina, facilitando a representação de máquinas grandes e complexas.

A linguagem propõe a definição e uso de variáveis nos estados e transições. No exemplo, a variável x é inicializada com o valor 1 na transição do estado `0` para o estado `1`. A máquina permanecerá no estado `1` enquanto a expressão $x < 5$ for verdadeira. A cada transição do estado `1` para ele mesmo, o valor de x é incrementado. Quando a expressão $x >= 5$ for verdadeira, ocorrerá a transição do estado `1` para o estado `2` e a variável x é redefinida com o valor 0.

```
MODULE main
VAR
        st : {s1,s2,s3,s4,s5,s6,s7};
ASSIGN
        init(st) := s1;
        next(st) := case
                st = s1 : {s2,s3};
                st = s2 : s5;
                st = s3 : {s1,s6};
                st = s4 : {s1,s3,s4};
                st = s5 : {s2,s3};
                st = s6 : s7;
                st = s7 : s4;
        esac;
DEFINE
        start := st in {s2,s5,s6,s7};
        close := st in {s3,s4,s5,s6,s7};
        heat := st in {s4,s7};
        error := st in {s2,s5};
SPEC AG(start -> AF(heat));
SPEC A[!heat U close];
SPEC AG(error -> AF(!error));
```

Figura 3.41: Exercício de Fixação: Modelo SMV

```
-- specification AG (start -> AF heat)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
 st = s1
 error = FALSE
 heat = FALSE
 close = FALSE
 start = FALSE
-- Loop starts here
-> State: 1.2 <-
 st = s2
 error = TRUE
 start = TRUE
-> State: 1.3 <-
 st = s5
 close = TRUE
-> State: 1.4 <-
 st = s2
 close = FALSE
```

Figura 3.42: Exercício de Fixação: Resultado para `AG( Start => AF Heat )`

Uma das principais características dessa linguagem é a possibilidade de representar o tempo por meio de variáveis contadoras. No exemplo, a máquina permanece no estado

```
-- specification A [ !heat U close ]    is true
```

Figura 3.43: Exercício de Fixação: Resultado para `A[ !Heat U Close ]`

```
-- specification AG (error -> AF !error)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  st = s1
  error = FALSE
  heat = FALSE
  close = FALSE
  start = FALSE
-- Loop starts here
-> State: 2.2 <-
  st = s2
  error = TRUE
  start = TRUE
-> State: 2.3 <-
  st = s5
  close = TRUE
-> State: 2.4 <-
  st = s2
  close = FALSE
```

Figura 3.44: Exercício de Fixação: Resultado para `AG( Error => AF !Error )`

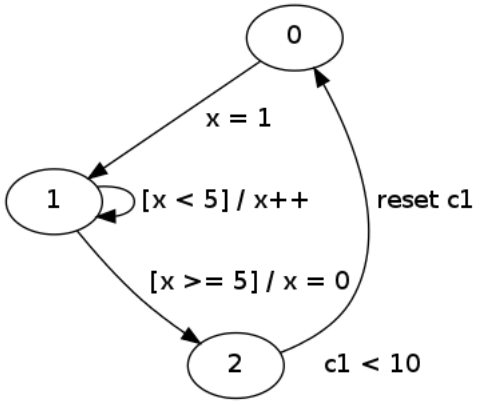

Figura 3.45: Exemplo de Representação de uma Máquina de Estados

2 enquanto a variável contadora $c1$ for menor que 10. Em seguida, ocorre a transição do estado 2 para o estado 0 e a variável contadora $c1$ é resetada.

Um outro exemplo, é a máquina da Figura 3.46. É possível notar que existe uma certa repetição nas transições dos estados. Os estados *s1*, *s5* e *s9* são os estados iniciais. Dependendo do estado inicial tomado, a máquina avança por um grupo diferente de quatro estados e depois retorna a algum dos estados iniciais.

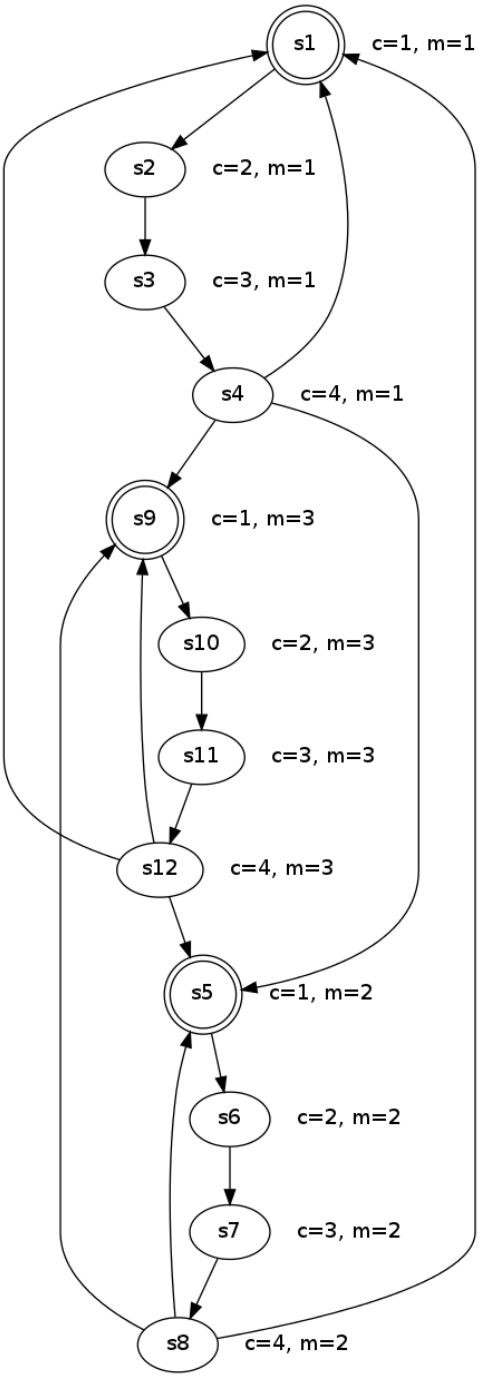

Figura 3.46: Exemplo 2: Máquina de Estados

Os estados da máquina apresentada no exemplo podem ser representados por meio de duas variáveis: (1) a variável contadora `c` que contabiliza quantos estados a máquina atingiu a partir do estado inicial; e (2) a variável `m` que indica em qual grupo de quatro estados a máquina está. A Figura 3.47 mostra a especificação da máquina no modelo `NuSMV`. Ao invés de enumerar cada um dos 12 estados da máquina, os estados são especificados por meio das variáveis `c` e `m`. Na linha 3 é especificado que a variável `c` pode

assumir valores inteiros em um intervalo de 1 a 4 e na linha 4 é especificado que a variável `m` pode assumir valores inteiros em um intervalo de 1 a 3.

```
1  MODULE main
2  VAR
3      c : 1..4;
4     m : 1..3;
5  ASSIGN
6      init(c) := 1;
7      init(m) := 1;
8      next(c) := case
9          c <= 3 : c + 1;
10         c = 4 : 1;
11     esac;
12     next(m) := case
13         c = 4 : {1, 2, 3};
14         c <= 3 : m;
15     esac;
```

Figura 3.47: Exemplo 2: Especificação no NuSMV

Por sua vez, a transição entre os estados é especificada definindo o que ocorre com as variáveis durante as transições. Nas linhas 8 a 11 a variável `c` é considerada: 1) se `c` for menor que 3, `c` será incrementado em 1 e; 2) se `c` é igual a 4, `c` receberá o valor 1.

De forma análoga, nas linhas 12 a 15 a variável `m` é considerada: 1) se `c` for igual a 4, no estado seguinte, `m` pode ter os valores 1, 2 ou 3 e; 2) se `c` é menor ou igual à 3, `m` permanecerá com o mesmo valor no estado seguinte.

3.3.5 Outras Possibilidades Providas pelo Modelo do NuSMV

A especificação por meio de variáveis possibilita representar de forma viável máquinas com muitos estados. A Figura 3.48 mostra uma extensão da especificação considerada na Seção 3.3.4. Na extensão são exploradas outras possibilidades providas pelo modelo do NuSMV que potencializam a especificação implícita de máquinas por meio de variáveis.

Uma dessas possibilidades é o uso de diferentes tipos de dados, além do tipo inteiro. Na linha 5, por exemplo, a variável `x` é declarada com tipo `boolean`. A linha 13 exemplifica como inicializar essa variável, a qual é inicializada com `FALSE`. É possível também declarar variáveis como *arrays*. Na linha 6, por exemplo, a variável `a` é declarada como um `array of boolean` com quatro posições. Ainda, na linha 7, a variável `b` é declarada como uma matriz de inteiros (`array of array`) com tamanho variável entre $0x0$ a $4x5$, sendo que os valores inteiros devem ser entre 1 a 3.

Uma outra possibilidade é a declaração de variáveis como módulos. No exemplo, o módulo `prc(in)` é especificado entre as linhas 24 e 31. Por sua vez, as variáveis `p1` e `p2`,

```
1  MODULE main
2  VAR
3      c : 1..4;
4      m : 1..3;
5      x : boolean;
6      a : array 0..4 of boolean;
7      b : array 0..4 of array 0..5 of 1..3;
8      p1 : prc(c);
9      p2 : prc(c);
10 ASSIGN
11     init(c) := 1;
12     init(m) := 1;
13     init(x) := FALSE;
14     next(c) := case
15         c <= 3 : c + 1;
16         c = 4 : 1;
17     esac;
18     next(m) := case
19         p1.st = s1 : {s1, s2, s3};
20         c = 4 : {1, 2, 3};
21         c <= 3 : m;
22     esac;
23
24 MODULE prc(in)
25 VAR
26     st : {s1, s2, s3};
27 ASSIGN
28     init(st) := s1;
29 FAIRNESS
30     running;
```

Figura 3.48: Exemplo 3: Especificação no NuSMV

respectivamente, nas linhas 8 e 9, são declaradas como duas instâncias diferentes desse módulo. O comportamento das variáveis de cada instância do módulo também podem ser especificados dentro do módulo `main`, conforme exemplificado na linha 19.

Por fim, outra possibilidade é a especificação de condições de *fairness*. Definir uma propriedade como uma condição de *fairness* significa que esta será válida em todos os caminhos infinitos. A condição de *fairness* mais comum é a propriedade *running*. Se, por exemplo, dois módulos tiverem a propriedade *running* definida como uma condição de *fairness*, significa que esses dois módulos deverão ficar alternando a execução. Na linha 30 do exemplo, a propriedade *running* é definida como uma condição de *fairness* do módulo `prc(in)`.

3.3.6 Exemplo Completo

A especificação da Figura 3.49 representa um esquema de mutex, ou seja, dois processos disputando uma região crítica. Como é possível observar, no módulo `main` tem-se

duas instâncias do processo `user` e uma variável booleana que representa um semáforo, indicando quando um processo está região crítica.

```
1  MODULE main
2  VAR
3     sem : boolean;
4     proc1 : process user(sem);
5     proc2 : process user(sem);
6  ASSIGN
7     init(sem) := FALSE;
8
9  MODULE user(sem)
10 VAR
11    st : {idle, entering, critical, exiting};
12 ASSIGN
13    init(st) := idle;
14    next(st) := case
15          st = idle : {idle, entering};
16          st = entering & !sem : critical;
17          st = critical : {critical, exiting};
18          st = exiting : idle;
19          TRUE : st;
20    esac
21    next(sem) := case
22          st = entering : TRUE;
23          st = exiting : FALSE;
24          TRUE : sem;
25    esac;
```

Figura 3.49: Exemplo 4: Especificação no NuSMV

Os processos `user`, por sua vez, podem estar em quatro estados diferentes: 1) `idle`, quando não estiver processando nenhuma informação; 2) `entering`, quando estiver entrando na regiao crítica; 3) `critical`, quando estiver na região crítica; e 4) `exiting`, quando estiver saindo da região crítica. Nas linhas 14 a 20 são especificadas as transições de estados dos processos `user` e nas linhas 21 a 25 são especificadas as mudanças no semáforo conforme um processo entra ou sai da região crítica. Embora intuitivamente o modelo parece estar adequado, algumas propriedades são interessantes de serem validadas: 1) dois processos não podem estar na região crítica ao mesmo tempo (Figura 3.50); 2) nenhum processo pode ficar infinitamente na região crítica (3.51); 3) se um processo quiser entrar na região crítica, ele eventualmente entrará na região crítica (Figura 3.52); 4) um processo pode ficar infinitamente em `idle` (Figura 3.53) e; 5) um processo pode ficar infinitamente em `entering` (Figura 3.54).

Ao executar o modelo no NuSMV tem-se que as propriedades 2 e 3 são falsas. Isto ocorre porque na linha 17 da especificação da máquina foi definidos que quando um processo está na região crítica ele pode continuar lá permitindo que um processo fique

```
SPEC AG(!(proc1.st = critical & proc2.st = critical));
```

Figura 3.50: Exemplo 4: Propriedade 1

```
SPEC AG(proc1.st = critical -> AG(proc1.st = exiting));
```

Figura 3.51: Exemplo 4: Propriedade 2

```
SPEC AG(proc1.st = entering -> AG(proc1.st = critical));
```

Figura 3.52: Exemplo 4: Propriedade 3

```
SPEC EF(EG(proc1.st = idle));
```

Figura 3.53: Exemplo 4: Propriedade 4

```
SPEC EF(EG(proc1.st = entering));
```

Figura 3.54: Exemplo 4: Propriedade 5

infinitamente na região crítica e outro nunca consiga entrar. No entanto, se for considerado justo que um processo não possa ficar infinitamente na região crítica pode-se alterar o modelo adicionando uma condição de *fairness*, conforme ilustrado na Figura 3.55. A condição diz que ambos processos devem entrar na região crítica infinitas vezes. Dessa forma, para que um processo possa entrar na região crítica infinitas vezes, se outro processo estiver na região crítica, eventualmente ele deve sair.

Executando novamente o modelo no NuSMV, tem-se que as propriedade 2 e 3 agora são verdadeiras e as propriedades 4 e 5 falsas. Como foi definido que é justo que os processos entrem na região crítica infinitas vezes, os processos não poderão ficar infinitamente em `idle` e `entering`.

3.4 Considerações Finais

Neste capítulo foi apresentado o método de *model cheking*, sendo dada ênfase a exemplos de uso da ferramenta NuSMV. No próximo capítulo serão abordados provadores de teoremas.

```
MODULE user(sem)
VAR
  st : {idle, entering, critical, exiting};
ASSIGN
  init(st) := idle;
  next(st) := case
      st = idle : {idle, entering};
      st = entering & !sem : critical;
      st = critical : {critical, exiting};
      st = exiting : idle;
      TRUE : st;
  esac
  next(sem) := case
      st = entering : TRUE;
      st = exiting : FALSE;
      TRUE : sem;
  esac;
FAIRNESS
  critical;
```

Figura 3.55: Exemplo 4: Modulo `user` com Condição de `Fairness`

Capítulo

4

Provas Formais

4.1 Considerações Iniciais

Nesta capítulo serão abordados aspectos relacionados a provas formais, bem como ferramentas que apoiam esse método.

4.2 Métodos de Prova

Para alguns, métodos formais envolve o desenvolvimento de programas os quais são possíveis de serem provados como corretos. É possível realizar isso com *model checker*, uma vez que a propriedade a ser testada esteja escrita em CTL e tenha uma máquina de estados para verificá-la. Entretanto, isso não é trivial e em alguns casos não é possível (tanto ter a proprietade em CTL quanto a máquina de estados). Para contornar esse problema, existem outras soluções, entre elas as provas formais. Uma prova envolve demonstrar se uma determinada assertiva é correta. Algumas premissas são apresentadas:

$p, p \rightarrow q \vdash q$

Não é possível afirmar que o conteúdo à esquerda de $\vdash$ é verdadeiro. Porém, se for verdadeiro, então o que está à direita também é verdadeiro. A prova é meramente formal, não importando o que significa p ou q. Um exemplo disso é "Existem unicórnios vermelhos (p). Se existem unicórnios vermelhos, então o Batman mora na minha casa ($p \rightarrow q$). Isso

conclui ($\vdash$) que o Batman mora na minha casa (q)". A veracidade desse exemplo não é o que está em análise. O que é importante é que dada as premissas (p e $p \rightarrow q$), é possível chegar a uma conclusão (q). Esse tipo de premissa é considerado um lema. Na sequência, outro exemplo é apresentado:

$\vdash p \rightarrow q \vee p$

Como não existe premissa antes de $\vdash$, então é ela é considerada uma verdade absoluta e o que está à direita é sempre válido. Ou seja, é uma tautologia.

Além de provar a validade de uma propriedade, também é possível provar se a mesma é satisfeita ou não. Para demonstrar isso, o seguinte exemplo é utilizado:

$\vdash (p \vee q) \rightarrow p$

Essa propriedade não é válida. Não há nada que garanta que se temos p ou q, então temos p. Apesar disso, é possível satisfazer essa propriedade? Uma propriedade satisfazível significa que existem valores booleanos para p ou q de forma que ela se torne verdade. Para descobrir se essa propriedade é satisfazível, é necessário atribuir valores (por exemplo, por meio de uma tabela verdade):

p	q	$\vdash (p \vee q) \rightarrow p$
T	T	T
T	F	T
F	T	F
F	F	T

Tabela 4.1: Exemplo de propriedade que pode ser satisfeira

Como pode ser visto na Tabela 4.1, existem 3 valores que satisfazem a propriedade. Porém ela não é válida, pois existe 1 valor falso (linha 3 da tabela). Portanto, a propriedade não é verdadeira para qualquer atribuição de p ou q (não é uma propriedade válida).

Existem diferentes métodos de prova, entre eles Força Bruta, Dedução e Método de Tableaux, além de uma ferramenta. O Força Bruta funciona, mas é indicado para um número pequeno de predicados em razão da dificuldade de sua execução. Por exemplo, 2 predicados geram 4 combinações e 3 predicados geram 8 combinações. A primeira propriedade (com 2 predicados) pode ser utilizada para demonstrar as diferentes técnicas de prova:

$(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$ (Expressão 1)

Uma forma de ler a expressão acima é: "Se chover (p) a calçada estará molhada (q), isso implica que se a calçada não está molhada ($\neg q$) então não choveu ($\neg p$)". O primeiro

método apresentado é o **Força Bruta**. Inicialmente é criada uma tabela verdade com os seguintes termos: p, q, $p \rightarrow q$, $\neg p$, $\neg q$, $\neg q \rightarrow \neg p$. A Tabela 4.2 mostra a tabela verdade dessa propriedade com esses termos.

p	q	$p \rightarrow q$	$\neg p$	$\neg q$	$\neg q \rightarrow \neg p$	Expressão 1	$(p \rightarrow q) \rightarrow (\neg p \rightarrow \neg q)$
T	T	T	F	F	T	T	T
T	F	F	F	T	F	T	T
F	T	T	T	F	T	T	F
F	F	T	T	T	T	T	T

Tabela 4.2: Força Bruta

Como pode ser visto na tabela, a Expressão 1 é válida (também pode ser chamada de tautológica), pois todas as suas combinações são verdadeiras. Em compensação, a expressão da última coluna ($(p \rightarrow q) \rightarrow (\neg p \rightarrow \neg q)$) não é uma taltologia, apesar parecer: "Se chover (p) a calçada estará molhada (q), isso implica que se não chover ($\neg p$) então a calçada não está molhada ($\neg q$)". Essa expressão não é válida porque uma das suas combinações é falsa.

Outra técnica de prova (**Dedução**) envolve o uso de regras de equivalência (por exemplo, De Morgan), apresentadas em aulas anteriores. No caso da Expressão 1, o predicado $p \rightarrow q$ é equivalente a $\neg p \vee q$. O mesmo se aplica para o outro predicado, $\neg q \rightarrow \neg p$ que é equivalente a $\neg\neg q \vee \neg p$. Portanto, a Expressão 1 ficou da seguinte forma:

$(\neg p \vee q) \rightarrow (\neg\neg q \vee \neg p)$

A implicação entre os dois predicados pode ser retirada da mesma forma que as outras duas implicações anteriores. Além disso, os símbolos $\neg\neg q$ podem ser simplificados, deixando apenas q. Agora, a expressão é essa:

$\neg(\neg p \vee q) \vee (q \vee \neg p)$

Note que as expressões $(\neg p \vee q)$ e $(q \vee \neg p)$ são equivalentes. Se for inserido um novo termo para representá-las (r), então a expressão fica nessa forma:

$\neg r \vee r$

Essa expressão é sempre verdadeira. Portanto, a propriedade é válida. A dificuldade dessa técnica é usar as regras de equivalência corretamente. Um erro ao utilizar uma determinada regra compromete o resultado final.

A terceira técnica é mais algorítmica, conhecida como **Método de Tableaux**. Não é um método garantidamente eficiente, porque o próprio problema de ser satisfazível (ou não) é NP-Completo, mas é um método que permite a resolução de problemas mais complexos, como os que serão mostrados a seguir, de uma forma relativamente sistemática. Considerando a mesma propriedade utilizada nas outras duas técnicas ($(\neg p \vee q) \rightarrow (\neg\neg q \vee \neg p)$), o que será provada é a sua validade. Relembrando, provar a validade de uma propri-

edade (se ela é válida) é mostrar que para todas as possíveis atribuições que o resultado é *TRUE*. Então, usando essa técnica, é inicia-se por analisar a contradição da propriedade (se ela é falsa). Se for garantido que a propriedade não pode ser falsa, então ela é válida (verdadeira). Porém, se ao tentar garantir que a propriedade é possível de ser satisfeita, analisa-se se a propriedade não pode ser verdadeira. Usando o mesmo exemplo, pretende-se verificar se a expressão é válida:

$(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$

Para construir a árvore envolvida nesta técnica, são utilizados os seguintes símbolos: ! no lugar do $\neg$; OR no lugar de $\vee$; e AND no lugar de $\wedge$.

```
(p --> q) --> (!q --> !p) = F    [1]
           |
           | [1]
           |
    (p --> q) = T    [2]
   (!q --> !p) = F   [3]
           |
           | [3]
           |
        !q = T   [4]
        !p = F   [5]
           |
           | [4]
           |
         q = F    {closed 1}
           |
           | [5]
           |
         p = T    {closed 2}
          /\                          (portanto a propriedade
   [2]   /  \   [2]                    não pode ser falsa e
        /    \                         a expressão é válida)
   p = F    q = T    {closed 1}
{closed 2}
```

Imagine que a expressão $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$ é falsa (1). O operador de mais alto nível é o de implicação ($\rightarrow$). Para que essa expressão seja falsa, a única possibilidade é $T \rightarrow F$. Portanto, "quebra-se" a expressão em duas partes: $(p \rightarrow q) = T$ (2) e $(\neg q \rightarrow \neg p) = F$ (3). Usando a segunda expressão (3), pode-se utilizar a mesma regra ($T \rightarrow F = F$), obtendo $\neg q = T$ (4) e $\neg p = F$ (5). Isso significa que $q = F$ e $p = T$. Usando a expressão que faltou (2), para que a implicação seja verdadeira tem-se duas possibilidades: $p = F$ ou $q = T$ (Lembrar da tabela verdade - $T \rightarrow T = T$, $F \rightarrow T = T$ e

$F \rightarrow F = T$). Como estão dentro do mesmo ramo da árvore as expressões $q = F$ e $q = T$; e $p = T$ e $p = F$, cada um desses pares formam um fecho (*closed*). Isso significa que os elementos dos pares são contraditórios dentro da mesma sequência de derivação. Como não existem mais possibilidades de derivação dentro da árvore e todos os ramos (um, nesse caso) estão fechados, então não é possível provar a condição inicial: a propriedade ser falsa (é falso a propriedade ser falsa). Portanto a propriedade é verdadeira (válida).

DICA: Derivar as expressões mais "simples" antes. Por exemplo, a expressão 3 do exemplo acima gera uma possibilidade. Por outro lado, a expressão 2 gera duas possibilidades. Portanto, é mais indicado manipular a expressão 3 antes da 2.

A próxima propriedade possui 3 símbolos, portanto é um pouco mais complexa:

$((p \vee q) \rightarrow r) \rightarrow (q \vee r)$

Ela é válida? É verdadeira para todas as possíveis combinações de p, q e r?

```
((p OR q) —> r) —> (q OR r) = F   [1]
                    |
                    | [1]
                    |
          (p OR q) —> r = T [2]
            (q OR r) = F   [3]
                    |
                    | [3]
                    |
                  q = F   {??}
                  r = F   {closed}
                   / \
              [2] /   \ [2]
                 /     \
 [4] (p OR q) = F     r = T   {closed}
              |
              | [4]             (portanto a propriedade
              |                  pode ser falsa e não
           p = F   {??}          é válida, porque não fechou
           q = F   {??}          em todos os caminhos)
```

Essa propriedade não é válida, pois existe uma possibilidade dela ser falsa. De acordo com a árvore da técnica de Tableaux, essa possibilidade é o caminho com $p = F$, $q = F$ e $r = F$ (ramo da esquerda). Substituindo esses valores na expressão, tem-se:

$((F \vee F) \rightarrow F) \rightarrow (F \vee F)$

$(F \rightarrow F) \rightarrow F$

$T \rightarrow F = F$

Exatamente a possibilidade que impede a expressão de ser válida. Apesar de não ser de difícil implementação, o Tableaux não é eficiente. Na maior parte dos casos, ele chega rapidamente em uma solução, mas existem exemplos em que o tempo que demora para resolver um problema é exponencial. Exemplo disso é uma propriedade que leva a derivação de muitos caminhos (2^n). Não tem como provar, na teoria, que o Tableaux é eficiente. Porém, na prática ele é eficiente para a maior parte dos casos que foram analisados. Atualmente, existem implementações mais sofisticadas do Tableaux, com o uso de indexações mais sofisticadas (por exemplo).

4.2.1 Exemplos

Nesta seção são apresentados alguns exemplos de prova por meio do método Tableaux.

Exemplo 1:

$\neg((p \rightarrow q) \wedge ((p \wedge q) \rightarrow r) \rightarrow (p \rightarrow r))$

```
1 !((p --> q) AND ((p AND q) --> r) --> (p --> r)) = T   [1]
2                                 |
3                                 | [1]
4                                 |
5  (p --> q) AND ((p AND q) --> r) --> (p --> r) = F   [2]
6                                 |
7                                 | [2]
8                                 |
9       (p --> q) AND ((p AND q) --> r) = T   [3]
10                 (p --> r) = F                [4]
11                                |
12                                | [3]
13                                |
14                        p --> q = T       [5]
15                 (p AND q) --> r = T   [6]
16                                |
17                                | [4]
18                                |
19          {closed 1}        p = T
20                            r = F    {closed 2}
21                            /   \
22                      [5] /       \ [5]
23                        /           \
24       {closed 1}   p = F          q = T   {closed 3}
25                                    /   \
```

```
                                [6] /     \ [6]
                                  /       \
                [7] p AND q = F       r = T   {closed 2}
                                |
                              [7] |
                                |          (portanto a expressão
                              p = F         não é satisfazível)
        {closed 3}            q = F
```

Como todos os ramos estão fechados, então não é verdade que a propriedade é satisfazível (igual a *TRUE*). Como ela não é satisfazível, então a inversa dela é válida. Um ramo fechado (closed) não sofre mais derivações. A derivação continua no ramo ainda não fechado. Se todos os ramos estão fechados, então não existe mais derivações. No caso do fecho 3 (closed 3), como um dos elementos dele foi fechado (q = F), não é necessário fechar o outro elemento (p = F).

Exemplo 2:

$(\neg((p \rightarrow q) \wedge (p \wedge q \rightarrow r) \rightarrow (\neg p \rightarrow r)))$

```
(!((p --> q) AND (p AND q --> r) --> (!p --> r))) = T   [1]
                                |
                                | [1]
                                |
 (p --> q) AND (p AND q --> r) --> (!p --> r) = F   [2]
                                |
                                | [2]
                                |
       (p --> q) AND (p AND q --> r) = T   [3]
                  (!p --> r) = F           [4]
                                |
                                | [4]
                                |
                             !p = T   [5]
                              r = F   {closed 1}
                                |
                                | [5]
                                |
                              p = F
                                |
                                | [3]
                                |
```

```
                        p --> q = T       [6]
                    p AND q --> r = T     [7]
                           /          \
                    [6]   /            \ [6]
                         /              \
                    p = F                q = T    {closed 2}
                    /   \                /   \
              [7]  /     \              /     \  [7]
                  /       \            /       \
[8] p AND q = F     r = T     p AND q = F     r = T   {closed 1}
        |                             |
        | [8]                         | [8]    (a propriedade não
        |                             |        (fechou em um dos
      p = F                         p = F       ramos, então ela
      q = F                         q = F       é satisfazível)
                                  {closed 2}
```

Repare que a expressão 7 é derivada nos dois ramos originários da expressão 6. O mesmo ocorre com a expressão 8 nos quatro ramos da expressão 7 (dois ramos para cada ramo da expressão 6). Perceba que, no final, o ramo da direita fechou, mas o ramo da esquerda não.

Se for considerado o ramo aberto ($p = F$, $q = F$ e $r = F$), encontra-se uma atribuição que faz a propriedade ser verdadeira (é o que era necessário provar), portanto a expressão é satisfazível.

Exemplo 3:

$(p \leftrightarrow q) \rightarrow (\neg q \leftrightarrow p)$

Ela é possível de ser satisfeita?

```
(p <--> q) --> (!q <--> p) = T   [1]
          /          \
         /            \ [1]
        /              \
[2] p <--> q = F     !q <--> p = T   [3]
      / \                  /   \
 [2] /   \                /     \  [3]
    /     \              /       \
 p = T   p = F       !q = T [4]  !q = F [5]
 q = F   q = T        p = T       p = F
                      |           |       (a propriedade
                      | [4]       | [5]    é satisfazível)
```

```
13                          |              |
14                        q = F          q = T
```

Como existe pelo menos um ramo aberto (no exemplo acima, todos estão abertos), então é possível satisfazer a propriedade. Neste caso, existe dois pares de atribuição que satisfazem a propriedade: ($p = T$ e $q = F$) e ($p = F$ e $q = T$).

Exemplo 4:

$(p \rightarrow r) \rightarrow (p \vee q \rightarrow r \vee q)$

É válida?

```
(p --> r) --> (p OR q --> r OR q) = F   [1]
2                    |
3                    |   [1]
4                    |
5             p --> r = T          [2]
6       p OR q --> r OR q = F     [3]
7                    |
8                    |   [3]
9                    |
10             p OR q = T    [4]
11             r OR q = F    [5]
12                   |
13                   |   [5]
14                   |
15               r = F     {closed 1}
16               q = F     {closed 3}
17                / \
18          [2]  /   \  [2]
19              /     \
20{closed 2} p = F     r = T   {closed 1}
21            / \
22      [4]  /   \  [4]                 (ramos fechados,
23          /     \                      então é válida)
24       p = T    q = T   {closed 3}
25    {closed 2}
```

Como todos os ramos estão fechados, então não existe uma atribuição que faz a propriedade ser falsa. Por isso, ela é válida (verdadeira).

Exemplo 5:

$(p \rightarrow q), (q \rightarrow r) \vdash (p \rightarrow r)$

Neste caso tem-se um lema e duas premissas que levam a uma conclusão. Para aplicar o Tableaux, é necessário converter esse lema para uma propriedade como as outras que foram analisadas até agora. Para isso, a vírgula que separa as duas premissas é alterada para um $\wedge$ e o conclusão ($\vdash$) torna-se uma implicação ($\rightarrow$). E como é um lema, então tem-se que provar a validade dele (analisar se existe alguma atribuição que o faz ser falso). Com isso, é possível aplicar o Tableaux.

```
((p --> q) AND (q --> r)) --> (p --> r) = F   [1]
                          |
                          | [1]
                          |
        (p --> q) AND (q --> r) = T   [2]
                    p --> r = F       [3]
                          |
                          | [3]
                          |
                      p = T   {closed 1}
                      r = F   {closed 3}
                          |
                          | [2]
                          |
                    p --> q = T   [4]
                    q --> r = T   [5]
                       /   \
                [4]   /     \   [4]
                     /       \
     {closed 1} p = F      q = T   {closed 2}
                              / \
                         [5] /   \ [5]
                            /     \          (ramos fechados,
          {closed 2}   q = F     r = T        então é válida)
                                 {closed 3}
```

O lema é válido, porque todos os ramos da propriedade estão fechados. Isso significa que não existe uma atribuição que faz com que o resultado seja falso.

Exemplo 6:

$(p \vee q) \wedge (p \vee \neg q) \equiv p$

Esta usa o símbolo de equivalência (≡). Neste caso, substitui-se o símbolo de equivalência por um se e somente se (↔). Em seguida, testa-se a equivalência (existe alguma atribuição que faz a propriedade ser falsa?).

```
(p OR q) AND (p OR !q) <--> p = F   [1]
                          /     \
                         /       \
                        /         \
(p OR q) AND (p OR !q) = T  [2]    (p OR q) AND (p OR !q) = F [3]
        p = F    {closed 1}                  p = T   {closed 3}
          |                                  /   \
          | [2]                             /     \ [3]
          |                                /       \
      p OR q = T [4]              [6] p OR q = F     p OR !q = F [7]
     p OR !q = T [5]                      |                 |
        / \                               | [6]             | [7]
  [4]  /   \ [4]                          |                 |
      /     \                           p = F             p = F
    p = T    q = T  {closed 2}          q = F            !q = F
{closed 1}    /\                    {closed 3}       {closed 3}
             /  \ [5]
            /    \
         p = T  !q = T [8]
     {closed 1}    |
                   | [8]     (todos os ramos fechados,
                   |          então é válida)
                 q = F
               {closed 2}
```

Como todos os ramos estão fechados, então não é possível atribuir um valor falso a expressão. Portanto ela é verdadeira sempre e a equivalência é válida.

DICA: como o ramo mais a direita foi fechado com o p = F, não é necessário derivar o !q = F.

4.2.2 Tableaux para Lógica de Primeira Ordem

É possível utilizar o Tableaux na Lógica de Primeira Ordem. Isso pode ser ilustrado com o seguinte exemplo:

$\forall x(p(x) \rightarrow q(x)), \forall y(p(y)) \vdash \forall z(q(z))$

Note que x, y e z estão em domínios diferentes. Isso significa que poderia ser tudo x. Deve-se provar se as duas premissas levam a essa conclusão. Para isso, troca-se a vírgula por um $\wedge$ e a conclusão ($\vdash$) por uma implicação ($\rightarrow$) e verifica-se se ela é válida. Como

o símbolo ∀ não é reconhecido no frame abaixo, troca-se por A. O mesmo se aplica ao símbolo ∃, que será trocado por E.

```
(Ax(p(x) —> q(x)) AND Ay(p(y))) —> Az(q(z)) = F [1]
                        |
                        | [1]
                        |
          Ax(p(x) —> q(x)) AND Ay(p(y)) = T [2]
                  Az(q(z)) = F              [3]
                        |
                        | [3]
                        |
                  Ez(!q(z)) = T
                        |
                   !q(c) = T     (c é uma constante)
                        |
                    q(c) = F  {closed 2}
                        |
                        | [2]
                        |
              Ax(p(x) —> q(x)) = T  [4]
                  Ay(p(y)) = T      [5]
                        |
                        | [5]
                        |
                  p(c) = T  {closed 1}
                        |
                        | [4}
                        |
                p(c) —> q(c) = T [6]
                     /    \              (ramos fechados,
                    /      \  [6]        então é válida)
                   /        \
             p(c) = F    q(c) = T {closed 2}
           {closed 1}
```

Foi utilizada a constante c, porque ela simboliza um valor escolhido no domínio $\forall x$ (para todo x). Como os ramos estão fechados, então é possível provar, usando Tableaux, que o argumento escrito na Lógica de Primeira Ordem é válido, pois não existe uma atribuição em que o resultado seja falso. Isso mostra que o Tableaux pode ser aplicado em qualquer tipo de lógica. Por exemplo, na CTL.

4.3 Ferramenta Z3Py

A ferramenta Z3Py[1] é um provador de teoremas, desenvolvida pela Microsoft e funciona em conjunto com a linguagem de programação Python. Essa ferramenta lida com os problemas de satisfatibilidade (*SAT Problems*), além de poder ser estendida para SMT (*Satisfiability Model Theory*). No SMT, dada uma teoria, tenta-se provar que algo é satisfazível.

No Z3Py, o comando 'prove' analisa a validade de um argumento e o comando 'solve' verifica a satisfatibilidade. Exemplos:

$((p \rightarrow q) \wedge p) \rightarrow q$

A Figura 4.1 mostra esse argumento escrito no Z3Py e o resultado.

Figura 4.1: Utilizando o Z3Py.

A variável p da esquerda é Python e a da direita (entre aspas) é a variável interna ao Z3Py. O Z3Py usa a notação "PREFIX". Como o resultado foi 'proved', então o Z3Py garante que a expressão é válida.

A seguir aparece um exemplo em que o Z3Py mostra que a expressão não é válida e apresenta um contra-exemplo.

$((p \rightarrow q) \wedge \neg p) \rightarrow \neg q$

Esse é um exemplo da famosa falácia da negação do antecedente. Em "palavras", temos: "Se o mordomo é o assassino (p), então ele estará com as mãos sujas de sangue

[1] http://rise4fun.com/Z3Py

(q). Ele não é o assassino (¬ p), então ele não está com as mãos sujas de sangue (¬ q)". Isso é uma falácia, porque o mordomo poderia sujar as mãos de sangue sem ter matado alguém. A Figura 4.2 mostra o resultado para esse argumento quando aplicado no Z3Py.

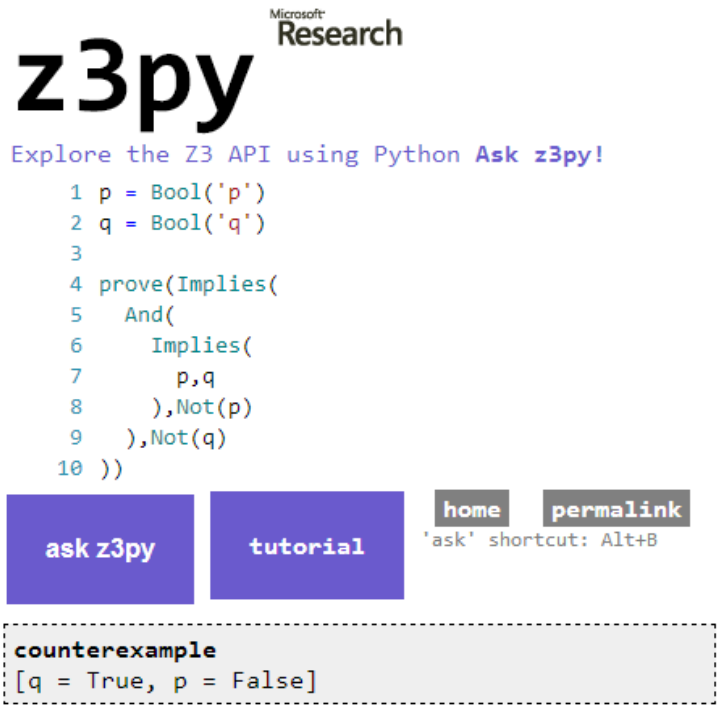

Figura 4.2: Segundo exemplo utilizando o Z3Py.

O Z3Py mostra que essa nova expressão é falsa. Para provar isso, basta utilizar o contra-exemplo apresentado por ele. Essa ferramenta da Microsoft usa o Tableaux (com otimizações) e consegue resolver fórmulas com várias equações. Além disso, utilizando o Tableaux, a ferramenta consegue aprender lemas (que se repetem) no processo de provar uma propriedade.

O próximo exemplo mostra o Z3Py utilizando o comando 'solve'. Neste caso, se houve uma atribuição que satisfaça a condição, é apresentada essa atribuição. A Figura 4.3 mostra isso.

Mais um exemplo utilizando o comando 'solve', considerando agora uma terceira variável e mais condições dentro do comando 'solve' (Figura 4.4).

No próximo exemplo, com a soma das três variáveis sendo divisível por 2 e a soma de x+y sendo divisível por 3. A Figura 4.5 apresenta a resposta apresentada pelo Z3Py.

Neste caso, o resultado envolve um número negativo ($y = -7$). Para evitar que isso ($x > 0$, $y > 0$ e $z > 0$) ocorra, pode-se incluir regras no comando 'solve'. Para isso pode-se escrever de outra forma, criando um objeto 'Solver' e adicionando regras a medida que for necessário. Essas regras podem ser checadas e o modelo criado pode ser impresso. A Figura 4.6 mostra o mesmo exemplo da Figura 4.5, com a utilização dessa composição.

Figura 4.3: Utilizando o comando 'solve' no Z3Py.

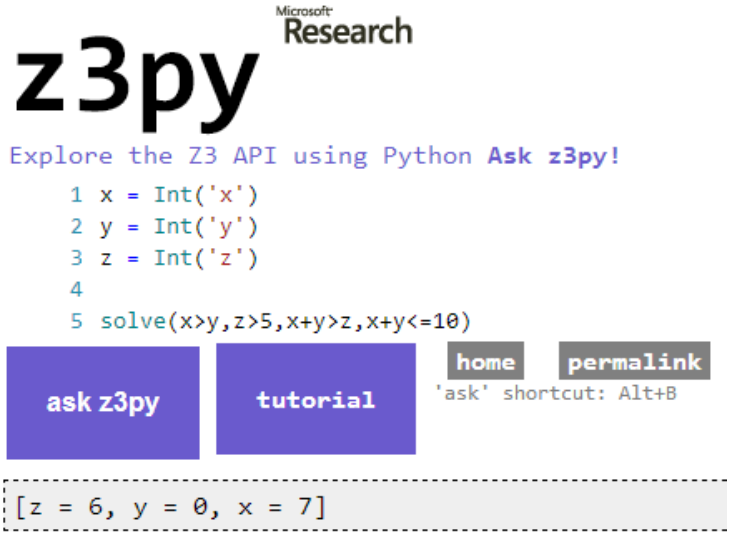

Figura 4.4: Segundo exemplo utilizando o comando 'solve' no Z3Py.

Para finalizar, este último exemplo utiliza uma função que retorna o maior divisor comum entre dois valores (função gcd):

```
gcd(x,y)
2 while(true) {
3   m = x % y
4   if m == 0
5     return y
6   x = y
7   y = m
8 }
```

Pretende-se que o Z3Py dê um valor para x e y que execute o laço (while) três vezes. Com isso, gera-se um caso de teste que cobre um caminho específico. Entretanto, o Z3Py

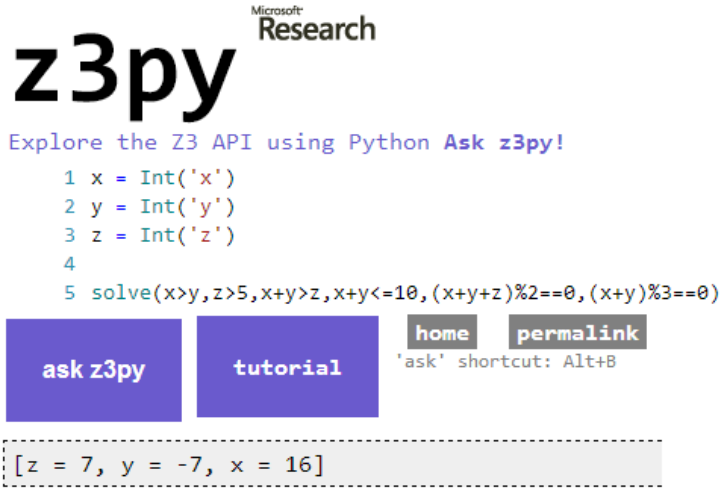

Figura 4.5: Segundo exemplo utilizando o comando 'solve' no Z3Py.

lida com matemática e $x = x + 1$ (por exemplo) é um absurdo, porque a simplificação disso é $0 = 1$. Apenas nas linguagens de programação tal comando existe, porque a máquina entende que existe um "x velho" e um "x novo". No caso do Z3Py isso tem que ser explícito. Como pretende-se executar três vezes é necessário o uso de três variáveis x, y e m. Além disso, para garantir que o resultado será apenas com números positivos, pode-se incluir as regras $x \geq 0$ e $y \geq 0$. O equivalente a isso no Z3Py é apresentado na Figura 4.7.

Pode-se aumentar os valores de x e y. Por exemplo, $x \geq 10$, $y \geq 10$, $y \geq 10^{10}$ e $y \geq 10^{100}$. Ainda assim, o Z3Py consegue apresentar resultados para provar que o modelo testado é satisfazível.

CURIOSIDADE: O inteiro do Z3Py é o inteiro matemático, diferente do inteiro de máquina. Como pode ser visto, se for testado com $y \geq 10^{10}$ e $y \geq 10^{100}$ (dois dos exemplos citados anteriormente). Mas se precisar, o Z3Py tem o inteiro de máquina.

4.3.1 Prova de Teoremas usando Z3Py

Conhecer métodos formais pode fornecer um poder de "persuasão" grande em razão das afirmações possíveis de serem feitas com as conclusões fornecidas pelo formalismo. A ferramenta Z3Py é um provador automático de teoremas. Dado um determinado programa, para garantir que este está correto, considera-se uma função $f(x, y)$, $\forall x, y$ se $f()$ retorna o maior de dois valores, ou seja, um z, nesse caso $(z \geq x; z \geq y; z == x; z == y)$. Com isso, obtém-se a noção de pré e pós condição. Caso a pré condição não seja satisfeita, podem acontecer 3 situações:

- Não executa a função $f()$;
- Executa $f()$, mas não garante a pós condição;

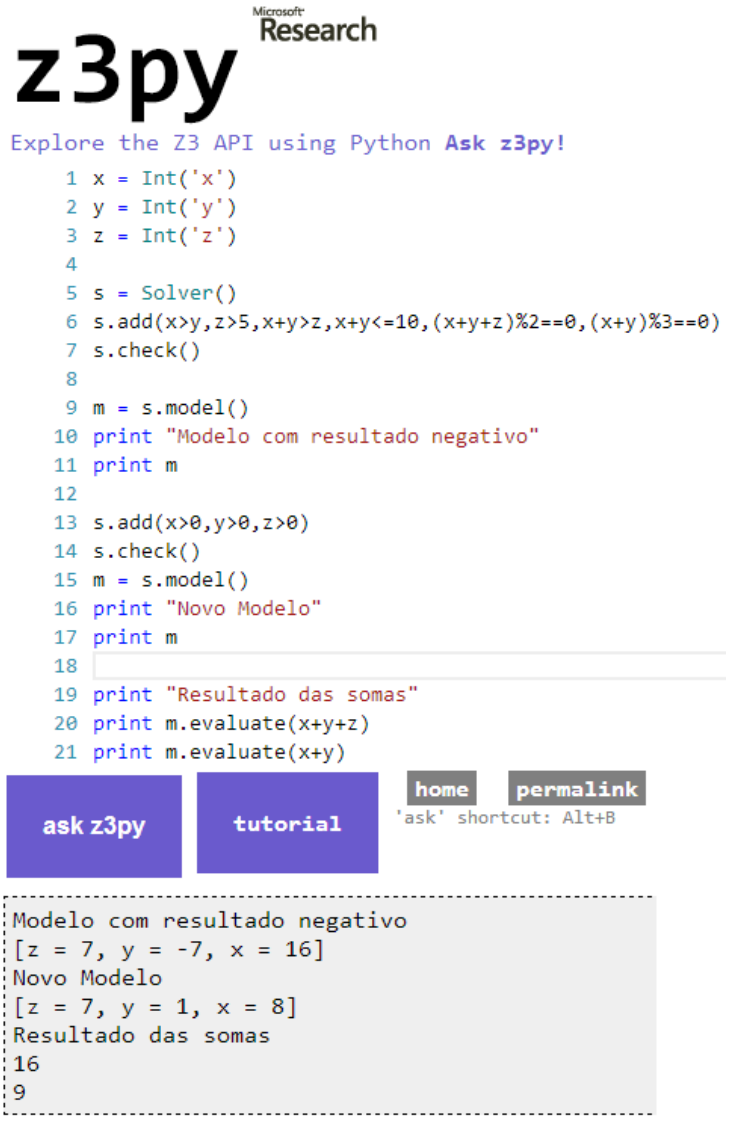

Figura 4.6: Exemplo utilizando um objeto Solver e adicionando as regras.

- Executa e aborta.

Em outro exemplo, imagine que:

- **Pré-Condição:** $x > 5$;
- **Expressão:** $y = x + 1$;
- **Pós-Condição:** $y > 5$.

Na prática isso quer dizer, $(x > 5 \wedge y == x + 1) \rightarrow (y > 5)$ (Expressão 1). Ou seja, a pré condição e a expressão tem que implicar na pós condição. A resolução dessa expressão no Z3Py é apresentada na Figura 4.8.

A resolução para uma situação diferente, em que $(x > 5 \wedge y == x + w) \rightarrow (y > 5)$ (Expressão 2) é apresentada na Figura 4.9.

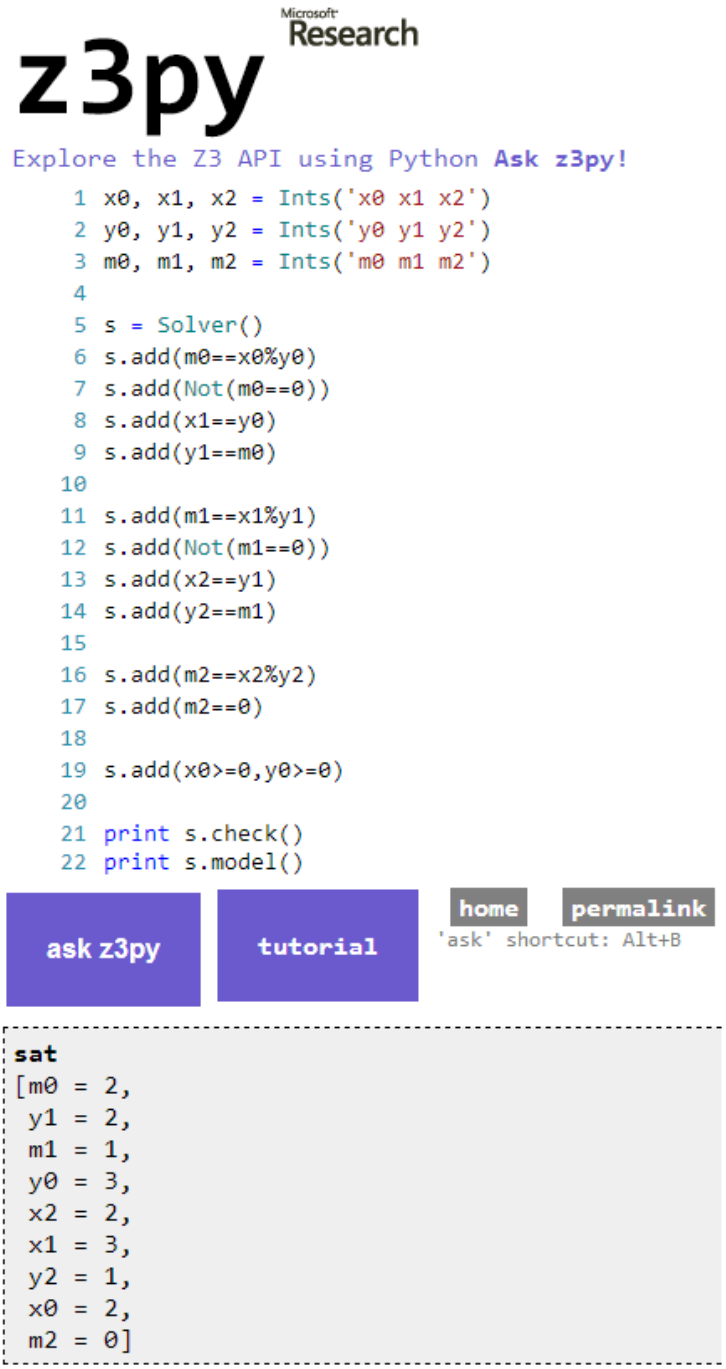

Figura 4.7: Função gcd() no Z3Py.

Além do conceito de pré e pós condição, existe o de invariante, o qual é sempre válido no contexto do programa. Ao desconsiderar o uso de invariantes, tem-se $(pre-condicao \wedge comando) \rightarrow pos-condicao$. Considerando a variante, tem-se $(pre-condicao \wedge condicao \wedge invariante) \rightarrow (pos - condicao \wedge invariante)$.

A parte mais difícil da prova é quando ela envolve o fluxo de controle dentro da prova. Considere o exemplo da Figura 4.10, com o uso de `if` e `else`. As possibilidades de fluxo seriam:

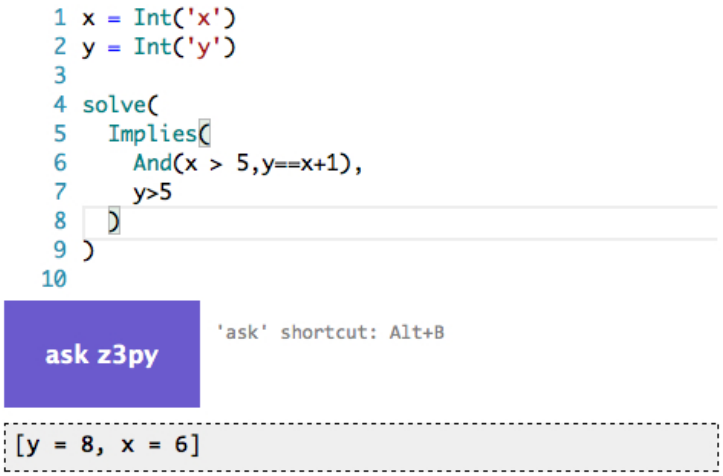

Figura 4.8: Expressão 1 escrita no Z3Py e seu resultado.

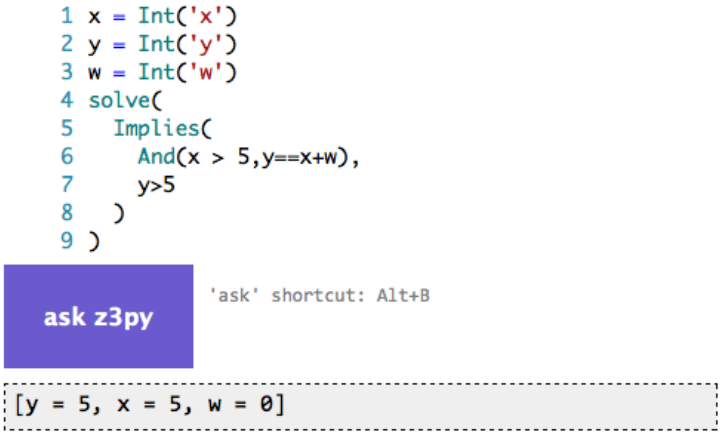

Figura 4.9: Expressão 2 escrita no *Z3Py e seu resultado.*

1. $((P \wedge E \wedge S1) \rightarrow Q) \wedge ((P \wedge \neg E \wedge S2) \rightarrow Q)$
2. $((P \wedge E \wedge S1)) \rightarrow Q) \wedge (P \wedge \neg E) \rightarrow Q)$

```
{P}
    if E then
        S1
    else
        S2
{Q}
```

Figura 4.10: Fluxo de Controle na Prova – `if` e `else`

No entanto, quando ao utilizar um comando `while` (Figura 4.11), a prova torna-se mais complexa. Neste caso, tem-se:

1. $P \wedge \neg E) \rightarrow Q$

2. $(P \wedge E \wedge S1 \wedge \neg E') \rightarrow Q$

3. $(P \wedge E \wedge S1 \wedge \neg E \wedge S1 \wedge \neg E' \wedge S1' \wedge lnotE'' \wedge S1''...) \rightarrow Q$

```
{P}
    while(E){
        S1
    }
{Q}
```

Figura 4.11: Fluxo de Controle na Prova – `while`

Neste ponto, ocorre o clássico "problema da parada", em que o provador não consegue assumir um limite para interromper o `while`. É necessário definir manualmente um limite de parada, por exemplo se o loop for executado 3x, o valor máximo de E é E''' para que o provador termine o while com esse número de loops e execute Q.

Observação: A notação E' indica que o valor de E foi executado uma segunda vez. Assim, E''' é o valor de E executado pela quarta vez.

4.3.2 Utilizando o Z3Py

Para declarar variáveis no Z3Py e resolver expressões, pode-se usar a seguinte sintaxe:

```
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

Neste exemplo, a função $Int('x')$ cria uma variável. O 'Solver' resolve o conjunto de restrições declaradas: $x > 2$, $y < 10$ e $x + 2 * y == 7$. Além dessas funções, o Z3Py também possui um simplificador de restrições, por exemplo:

```
x = Int('x')
y = Int('y')
print simplify(x + y + 2*x + 3)
print simplify(x < y + x + 2)
print simplify(And(x + 1 >= 3, x**2 + x**2 + y**2 + 2 >= 5))
```

Neste caso, para a primeira expressão, o simplificador retorna $3 + 3x + y$. Para a segunda, tem-se $\neg(y \leq 2)$. Finalmente, para a última expressão, a simplificação resulta em $(x \geq 2) \wedge (2x * *2 + y * *2 \geq 3)$ (Expressão 3).

```
1 x = Int('x')
2 y = Int('y')
3 s = Solver()
4 print s
5 s.add(x > 10, y == x + 2)
6 print s
7 print "Solving constraints in the solver s ..."
8 print s.check()
9 print "Create a new scope..."
10 s.push()
11 s.add(y < 11)
12 print s
13 print "Solving updated set of constraints..."
14 print s.check()
15 print "Restoring state..."
16 s.pop()
17 print s
18 print "Solving restored set of constraints..."
19 print s.check()
```

```
[]
[x > 10, y = x + 2]
Solving constraints in the solver s ...
sat
Create a new scope...
[x > 10, y = x + 2, y < 11]
Solving updated set of constraints...
unsat
Restoring state...
[x > 10, y = x + 2]
Solving restored set of constraints...
sat
```

Figura 4.12: Utilizando o Z3Py para resolver a Expressão 3.

Visando um melhor suporte para a escrita das restrições, o Z3Py oferece a opção de acrescentar e remover restrições, utilizando as funções 'push()', 'pop()' e 'add()' dentro de um 'Solver'. A Figura 4.12 apresenta um exemplo para este caso.

Neste exemplo, na Linha 4 é instanciado um objeto 'Solver' em s. O primeiro comando 'print' mostra que s está vazio, como visto na primeira linha da saída. É possível adicionar novas restrições através dos métodos *add*() do 'Solver', como pode ser visto na linha 5. O método *check*() resolve as restrições já adicionadas. Com o método *push*() cria-se um ponto de restauração, em que as próximas utilizações do comando *add*() podem ser removidas através do método *pop*(). Mais recursos do Z3Py podem ser acessados em: `http://rise4fun.com/Z3Py/tutorial`.

4.3.3 Resolução de Problema: Sudoku

Este problema envolve uma matriz $9x9$ previamente preenchida com alguns números entre 1 e 9. Para resolvê-la, basta completar cada linha, coluna e quadrado $3x3$ com números. Porém, deve-se seguir as seguintes restrições: não pode ter números repetidos nas linhas

horizontais e verticais, como também nos quadrados grandes de $3x3$. A Figura 4.13 apresenta um exemplo de Sudoku.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 4.13: Apresentação de um possível Sudoku.

Para resolver este problema utilizando a ferramenta Z3Py[2], cada posição da matriz deve ser uma variável. Também deve-se levar em conta que algumas das posições da matriz já estão preenchidas. Assim:

- x[0][0] = x_0_0 -> cada posição deve ser uma variável
- x[0][0] >= 1 e x[0][0] <= 9 -> os números devem estar no intervalo de 1 a 9
- x[0][0] != x[0][1], x[0][0] != x[0][2], x[0][1] != x[0][2] -> para cada submatriz $3x3$ não pode haver um número repetido

No Z3Py, a diretiva 'Distinct' faz a restrição acima para as submatrizes. O código da Figuras 4.14 mostram uma possível resolução do problema para matriz de tamanho 9x9.

Outros provadores de teoremas também podem ser utilizados, como o Yices, mas a ferramenta Z3Py é considerada uma das melhores.

4.4 Considerações Finais

Neste capítulo foram apresentados métodos de provas de teorema. Além disso, foram apresentados algumas funções do provador automático de teoremas Z3Py. Foi visto

[2]http://rise4fun.com/Z3Py/tutorial/guide

```
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(4) ] for i in range(4) ]

s = Solver()

for i in range(4):
  for j in range(4):
    s.add(X[i][j] >=1, X[i][j] <=4)

for i in range(4):
  s.add(Distinct(X[0][i],X[1][i], X[2][i], X[3][i] ) )

for j in range(4):
  s.add(Distinct(X[j][0],X[j][1], X[j][2], X[j][3]) )

for i in [0,2]:
  for j in [0,2]:
    s.add(Distinct(X[i][j], X[i][j+1], X[i+1][j], X[i+1][j+1] ) )

k = s.check()
print k

if k == sat:
  m = s.model()
  for i in range(4):
    print [ m.evaluate(X[i][j]) for j in range(4) ]
```

Figura 4.14: Solução do Sudoku no Z3Py

também que apesar do Z3Py conseguir provar inúmeros teoremas, ele não é capaz de tratar o "problema da parada" de maneira automática. Apesar disso, ele pode ser utilizado para outras aplicações, como resolver o problema do Sudoku. No próximo capítulo, serão apresentados métodos de teste formais.

Capítulo

5

Teste Formal

5.1 Considerações Iniciais

Uma especificação é um modelo do mundo real que define características do sistema e seu comportamento. Esta deve ser formal, de modo a evitar ambiguidades e possibilitar a análise algorítmica. Até agora foi visto que podem ser provadas propriedades que estão na especificação. O próximo passo é implementar o software. Com o código pronto, no teste tradicional é testado apenas o programa, sem base formal. Entretanto, é possível testar o software formalmente, utilizando a especificação formal. O teste caixa preta é o mais simples de ser formalizado, pois são consideradas apenas as entradas e saídas do programa. Neste capítulo é aboradado a aplicação de métodos formais em teste de software, com a construção de métodos para sequências de verificação para máquinas de estados.

5.2 Geração de Testes para Máquinas com Sequência de Distinção

O teste formal permite construir um modelo provido de embasamento matemático capaz de fornecer garantias mais fortes no teste. Esses modelos mais simples possibilitam afirmações mais poderosas. Considere a seguinte Máquina de Estados determinística ilustrada na Figura 5.1.

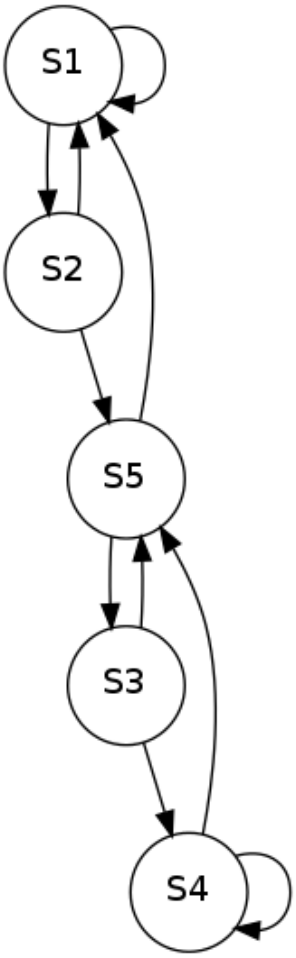

Figura 5.1: Máquina de Estados deterministica 1

Esta máquina de estados é determinística pois, para cada entrada fornecida ao modelo há apenas uma saída possível. Esse autômato acima define uma linguagem regular. Neste exemplo há dois conjuntos diferentes: $I = a, beO = 0, 1$ e L(m) $\in$ (IO)* e observa-se que:

1. A entrada abb com saída a0b1b1 (lê-se a com saida 0, b com saída 1 e b com saída 1) $\in$ L(m);

2. A entrada ab com saída a1b0 (lê-se a com saída 1 e b com saída 0) $\notin$ L(m);

3. A entrada aab com saída a0a0b0 (lê-se a com saída 0, a com saída 0 e b com saída 0) $\notin$ L(m).

Outra notação pode ser importante no teste. No teste funcional há entradas que são fornecidas e saídas que são observadas. No exemplo fornecido na Figura 5.1 o conjunto I é o conjunto de entrada e O é o conjunto de saída. Ao considerar agora a entrada abb com saída 011, pode-se convertê-la usando:

λ(S1, abb) = 011 e pode-se escrever: $\lambda(S, \alpha) = \beta \in O^* \mid \alpha \bigotimes \beta \in$ L(m).

- λ(S1,aab)=001;
- λ(S1,aab)=000.

Se um caminho for adicionado de S4 para S3 com entrada a e saída 1, tem-se: λ(S4,aab) = $\{000, 110, 010\}$. Nesse caso a máquina seria não determinística. Se ela fosse determinística teria essa propriedade: $\forall \in$ S, $\alpha \in I^* \mid \lambda\ (1,\alpha) \mid \leq 1$.

Outra situação seria retirar a transição de S4 para S5 e aí λ(S4, aab) = {}. Se a máquina fosse completa seria: $\forall$S $\in$ S, $\alpha \in I^* \mid \lambda\ (s, \alpha) \mid \geq 1$.

A máquina poderia ser não determinística (se para algumas entradas há mais de uma transição possível) e completa (não há transições não definidas).

Para gerar testes a partir de uma máquina pode-se pensar inicialmente em testar todos estados, como exemplo, testar a sequência 'baabab' que geraria como saída '100110'. Porém, é possível que um desafio seja formulado: escrever uma MEF determinística com no máximo 5 estados e que produz para essa mesma entrada uma saída diferente.

Após esse exercício, qual a chance de construir mais de uma MEF igual? Ou, quantas MEFs completas e determinísticas existem para um modelo com 5 estados, 2 entradas e 2 saídas? Para cada 10 transições há 10 possibilidades, o que fornece: $10^{10} = 10.000.000.000$ de máquinas.

Para criar uma sequência mais elaborada, após visitar todos estados pode-se pensar em visitar todas as transições. A sequência (com a entrada e a saída) agora poderia ser: 'b1a0a0b1a1b0a1a0b1a0a0b1b0b1a1a0'. O próximo desafio seria criar uma MEF que aceite esse caso de teste.

Nesse ponto o que pode ser feito? Aumentar a sequência e verificar se uma das 10 bilhões de MEFs existentes para esse modelo deixaria passar essa sequência? Existe um método que gera uma sequência que nenhuma das 10 milhões de MEFs aceitariam. Esse método gera uma sequência G(ω) = ω que é fornecida ao programa. Se a saída obtida com a sequência ω for incorreta é o que se espera como correto, se a saída for correta a sequência está correta. Portanto considera-se que ou o programa está correto ou ele não é equivalente às outras 10 bilhões de MEFs.

Para encontrar esse método imagine uma máquina com 20 estados e que para cada estado há 5 entradas e 5 saídas, haveria $(20x5)^{20x5}=100^{100}=10^{200}$ possibilidades, um número gigante. Pode-se pensar que a sequência gerada por um determinado método é muito pequena considerando a afirmação possível de ser feita com ela.

Para todo programa se $\forall$ S, S' $\in$ S, S $\neq$ S' $\rightarrow \lambda$(S, γ) $\neq \lambda$(S', γ). Se fosse feita a suposição γ = aba, nota-se que para cada estado do modelo há saídas diferentes para essa mesma entrada, observe:

Ao considerar 'aba' com saída '010', estando no estado S1 essa sequência leva ao estado S2 e se o ponto de partida é qualquer outro estado do modelo, não há caminho definido

para essa mesma entrada e saída. Ao considerar agora a mesma sequência 'aba' com saída '011' ela só está definida para o estado S2 que leva ao estado S4. Considerando agora a sequência 'aba' com saída '101' ela só está definida em S3 que leva ao estado S1. A mesma entrada com saída '001' só está definida para S4 e '110' para S5. Com isso, tem-se a sequência de distinção apresentada na Tabela 5.1.

Tabela 5.1: Seguência de distinção para γ = aba

Estado	Sequência
S1	010
S2	011
S3	101
S4	001
S5	110

Isso significa que imagine o 'aba' aparecendo 2 vezes numa sequência '010' (estado S1) e '001' (estado S4). Se eles estão em estados diferentes sabe-se que aplicando 'aba' é possível ir para o estado S1.

Pode-se fazer a geração da seguinte forma: γ = aba, se:

- '010' (sabe-se que este é o estado S1 na implementação que segue o modelo sugerido na Figura 5.1, isto pode ser conferido na Tabela 5.1;
- para o próximo estado usa-se 'aba' obtendo-se '011' e de acordo com o modelo ou com a Tabela 5.1 somente o estado S2 resulta nessa saída. Conclui-se então que este é o estado S2.
- como não se sabe qual o próximo estado usa-se 'aba' que é a informação disponível do estado S1, obtendo saída '001' e somente o estado S4 resultaria nessa saída, o que é suficiente para concluir que este é o estado S4.
- pode-se aplicar 'aba' novamente para saber o próximo estado, o que fornece '010' e a certeza de que este é o estado S1;
- pela informação anterior sabe-se que estando em S1 e aplicando novamente 'aba' chega-se em S2 e sabendo-se qual o estado atual testa-se as sequências;
- exemplo, testando agora em 'a' com saída '0' e aplicando em seguida 'aba' com saída '110', sabe-se que este é o estado S5;
- aplicando 'aba' novamente com saída '001', sabe-se que este é S4, pois ele vem após S2;
- em S4 pode-se testar outra transição: 'a' com saída '0';

- aplicando 'aba' e tendo como saída '001', sabe-se que o estado atual é o S4;
- testando a transição 'a' com saída '0' e aplicando 'aba' obtendo '011', sabe-se que o estado atual é o S2;
- testando a transição 'b' com saída '0' e aplicando 'aba' com saída 110, sabe-se que o atual é o estado S5;
- testando a transição 'b' com saída '1' e usando 'aba' obtendo '010', sabe-se que o atual é o estado S1;
- a transição 'a' com saída '0' pode ser testada e para isso usa-se agora apenas a transição 'a' com saída '1' que leva ao estado S5;
- agora testa-se o 'aba' com saída '010', o que indica que o atual é o estado S1;
- em seguida pode-se testar a transição 'a' com saída '0' usando a transição 'b' com saída '1', o que indica como estado atual o S5;
- ao testar a trasição 'aba' com saída '101' após a trasição 'b' com saída '1' sabe-se que o atual é o estado S3;
- em seguida pode-se testar a transição 'aba' com saída '010', o que indica S1 como estado atual;
- estando no estado S1 pode-se testar a transição 'b' com saída '1' que leva ao estado S2;
- aqui é possivel testar novamente a transição 'b' com saida '1' e este passo faz continuar no estado S1;
- agora se usar-se o conhecido 'aba' com saída 010 sabe-se que este é o estado S1;
- testando agora a transição 'a' com saida '0' sabe-se que este é o estado 2;
- testando agora 'b' som saída '1' sabe-se que este é o estado S5;
- pode-se testar agora 'b' com saída '0', o que indicará estar no estado S3;
- nesse passo pode-se usar o 'aba' com saída '110', o que indica que este é o estado S5;
- pode-se testar agora a transição 'a' com saida '0' e identificar que este é o estado S2;

- pode-se testar agora a transição 'b' com saída '1', o que indica que este é o estado S5;
- pode-se testar agora a transição 'a' com saída '1' e sabe-se que este é o estado S3;
- para fechar pode-se finalizar com a transição 'aba' com saída '001' que indica o estado S4 como atual.

Com essa sequência de 63 entradas são eliminadas as 10 bilhões de MEFs que existem para esse modelo e consegue-se criar um método de geração para sequências de verificação. Para essa máquina uma sequência com menos de 30 entradas é capaz de eliminar as 10 bilhões de possibilidades, porém não há prova analítica que o método gera essa sequência, é possível gerá-la apenas por força bruta.

É possível que sejam feitas otimizações na sequência gerada anteriormente. Exemplo: '01' indica estar no estado S1 ou S2, se for aplicado '10' sabe-se que necessariamente este é o estado S3, com '00' sabe-se que este é o estado S4 e com '11' sabe-se que este é o estado S5.

O princípio para essa otimização parte do pressuposto de que para 'aba' com a saída '010' sabe-se que essa sequência leva ao estado S1 e em seguida como já existe um 'a' no final dessa sequência não há necessidade de inseri-lo novamente no final para um segundo teste. Aplicando as otimizações tem-se:

1. com 'aba' e saída '010' sabe-se que este é o estado S1;
2. em seguida pode-se testar a transição 'b' com saída '1' o que permite continuar no estado S1;
3. testa-se agora a transição 'a' com saída '0', o que leva do estado S1 para o estado S2;
4. testa-se 'a' com saída '0' novamente o que leva a S5;
5. estando em S5 pode-se testar 'b' com saída '1' e acredita-se estar em S3;
6. testa-se a sequência 'a' com saída '1' e acredita-se estar em S4;
7. testa-se a sequência 'b' com saída '0' e acredita-se estar em S5;
8. testa-se a sequência 'a' com saída '1' e acredita-se estar em S1;
9. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
10. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;

11. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
12. testa-se a sequência 'a' com saída '1' e acredita-se estar em S1;
13. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
14. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
15. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
16. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
17. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
18. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
19. testa-se a sequência 'a' com saída '1' e acredita-se estar em S1;
20. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
21. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
22. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
23. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
24. testa-se a sequência 'b' com saída '1' e acredita-se estar em S3;
25. testa-se a sequência 'b' com saída '0' e acredita-se estar em S5;
26. testa-se a sequência 'a' com saída '1' e acredita-se estar em S1;
27. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
28. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
29. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
30. testa-se a sequência 'b' com saída '1' e acredita-se estar em S3;
31. testa-se a sequência 'a' com saída '1' e acredita-se estar em S4;
32. testa-se a sequência 'a' com saída '0' e acredita-se estar em S4;
33. testa-se a sequência 'b' com saída '0' e acredita-se estar em S5;
34. testa-se a sequência 'a' com saída '1' e acredita-se estar em S1;

35. testa-se a sequência 'b' com saída '1' e acredita-se estar em S1;
36. testa-se a sequência 'a' com saída '0' e acredita-se estar em S2;
37. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
38. testa-se a sequência 'b' com saída '1' e acredita-se estar em S3;
39. testa-se a sequência 'a' com saída '1' e acredita-se estar em S4;
40. testa-se a sequência 'a' com saída '0' e acredita-se estar em S4;
41. testa-se a sequência 'a' com saída '0' e acredita-se estar em S5;
42. testa-se a sequência 'b' com saída '0';

Com as otimizações de 63 entradas foi reduzida para 45 entradas e continua sendo capaz de eliminar 10 bilhões de possibilidade, utilizando força bruta.

5.3 Geração de Testes para Máquinas sem Sequência de Distinção

O método de geração de testes visto na aula anterior é bastante eficiente para máquinas com sequência de distinção. No entanto, nem todas as máquinas possuem tal sequência. Neste sentido, é necessário um método mais geral para a geração de testes, que não dependa da existência dessa sequência.

A máquina da Figura 5.2 pode ser utilizada como exemplo. Conforme ilustrado na Tabela 5.2, é possível distinguir o estado `S3` com uma entrada igual a 'a'. No entanto, tanto no estado `S1` como no `S2`, ao aplicar a entrada 'a', ocorre uma transição para o estado `S2`. Dessa forma, independente de quais sejam as próximas entradas, não é possível distinguir se os estados anteriores eram `S1` ou `S2`.

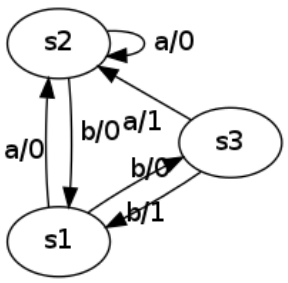

Figura 5.2: Máquina de Estados sem Sequência de Distinção

Tabela 5.2: Saída para Sequências com Início 'a' e 'b'

	'a' ...
S1	0
S2	0
S3	1

	'b' ...
S1	1
S2	0
S3	0

De forma análoga, é possível distinguir o estado S1 com uma entrada igual a 'b', mas independente de quais sejam as próximas entradas, não é possível distinguir se os estados eram S2 ou S3. No entanto, se aplicar as entradas 'a' e 'b' ao mesmo tempo, é possível distinguir em qual estado a máquina estava. Ao observar a saída produzida com a entrada 'a' pode-se identificar se o estado era S3. Se não fosse, por meio da saída produzida com a entrada 'b' pode-se identificar se o estado era S1. Caso não fosse, conclui-se, então, que o estado era S2.

Para conseguir aplicar as entradas 'a' e 'b' no mesmo estado, o método assume que existe uma sequência r que, tanto na especificação como na implementação, é capaz de resetar a máquina, ou seja, independente do estado em que a máquina está, ao aplicar a sequência r ela sempre volta ao estado inicial. Se r for confiável, isso significa que é possível utilizar a saída combinada de 'a' e 'b' para saber qual era o estado.

Neste sentido, a Tabela 5.3 ilustra como distinguir os estados utilizando a sequência r. Inicialmente, a máquina é resetada e aplica-se a entrada 'a'. Como a saída foi '0' conclui-se que o estado era S1 ou S2. Resetando novamente a máquina e aplicando a entrada 'b' a saída é igual a '1', o que leva a conclusão de que o estado era S1.

Tabela 5.3: Distinção de Estados Utilizando a Sequência r

r	'a'
'0'	
r	'b'
'1'	

Dessa forma, pode-se substituir a sequência de distinção por esse conjunto de sequências para avaliar qual era o estado da máquina. Esse conjunto de sequências é chamado **conjunto de caracterização (w)** e é definido por: $\forall s, s' \in S, s \neq s', \exists \propto \in w, \lambda(s, \propto) \neq \lambda(s', \propto)$. Para o exemplo, o conjunto de caracterização é $w = \{a, b\}$.

O principal aspecto do conjunto de caracterização é que praticamente toda máquina possui um. Se uma máquina não tiver um conjunto de caracterização, significa que a máquina não é minimal, ou seja, que existe uma outra máquina equivalente a essa com menos estados. Ao reduzir a máquina para outra equivalente e minimal, esta terá um conjunto de caracterização.

A Tabela 5.4 mostra as sequências que deve-se utilizar para testar a transição (S3, S2). As duas primeiras sequências são utilizadas para testar a origem da transição. Para

atigir o estado `S3` utiliza-se a entrada 'rb' e para garantir que o estado atingido era `S3` aplica-se a esta sequência o conjunto de caracterização $w = \{a, b\}$, gerando as duas primeiras sequências da Tabela 5.4.

Tabela 5.4: Conjunto de Sequências para Testar a Transição *(s3, s2)*

'rb|a'
'rb|b'
'rba'
'rba|a'
'rba|b'

A terceira sequência é utilizada para testar a transição. Inicialmente, aplica-se a sequência 'rb' para atingir o estado *S3* e depois aplica-se a entrada 'a', correspondente a transição que se quer testar. Por fim, as duas últimas sequências testam o destino da transição. Utilizando a sequência 'rba', realiza-se a transição *(S3, S2)* e em seguida aplica-se a esta sequência o conjunto de caracterização para garantir que o destino é o esperado, gerando as duas últimas sequências da tabela.

Como é possível observar, a terceira sequência pode ser desconsiderada, pois é igual a primeira. Ainda, a terceira sequência também pode ser desconsiderada, uma vez que é prefixo das duas últimas sequências. Ao executar qualquer uma das últimas sequências, já que a primeira sequência estaria sendo executada.

Por sua vez, a Tabela 5.5 mostra as sequências que devem ser utilizadas para testar a transição *(S3, S1)*. De forma análoga, as duas primeiras sequências são utilizadas para testar a origem, a terceira é utilizada para testar a transição e as duas últimas para testar o destino.

Tabela 5.5: Conjunto de Sequências para Testar a Transição *(s3, s1)*

'r|a'
'r|b'
'rb'
'rb|a'
'rb|b'

Como é possível observar, a segunda e a terceira sequência podem ser desconsideradas, pois são prefixos das duas últimas sequências. Ainda, as duas últimas sequências também podem ser desconsideradas, uma vez que já foram consideradas ao testar a transição `(S3, S2)`.

Dentro desse contexto, percebe-se que por meio de simplificações é possível obter um conjunto reduzido de sequências de entradas capazes de testar todas as transições. Um dos métodos existentes que permite de forma prática obter uma sequência de teste que executa todas as transições de uma máquina é o **Método W**.

Para obter um conjunto de sequências de teste por meio do método W é necessário gerar uma árvore de execução. Para construir esta árvore, o estado inicial deve ser considerado como a raíz. No exemplo, a raíz da árvore é o estado `S1` (Figura 5.3).

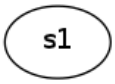

Figura 5.3: Construção da Árvore de Execução: Passo1

Em seguida, os filhos devem ser considerados como cada um dos estados que pode-se atingir a partir do estado inicial (Figura 5.4). No caso do estado `S1` são: 1) o estado `S2`, quando a entrada é igual a 'a'; e 2) o estado `S3`, quando a entrada é igual a 'b'.

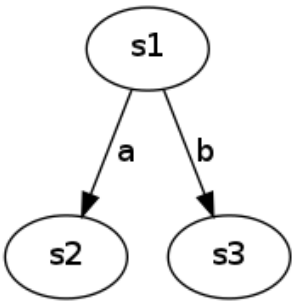

Figura 5.4: Construção da Árvore de Execução: Passo 2

O mesmo processo deve ser seguido para cada um dos nós folhas atuais até que se atinja um estado que já foi inserido na árvore (Figura 5.5). Ao construir o terceiro nível da ávore exemplificada, tem-se que os estados que podem ser atingidos tanto a partir de `S2` como a partir de `S3` são: 1) `S2` com entrada igual a 'a'; e 2) `S1`, com entrada 'b'. Sendo que ambos já foram inseridos anteriormente, o que encerra a construção da árvore.

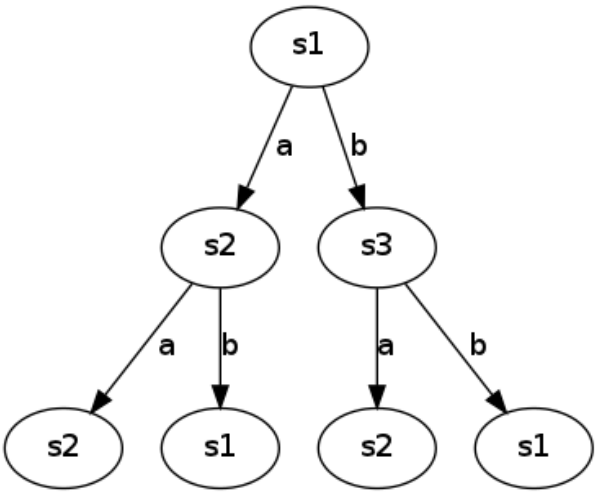

Figura 5.5: Construção da Árvore de Execução: Passo 3

Em seguida, deve-se aplicar o conjunto de caracterização w em cada um dos nós da árvore de execução, conforme ilustrado na Figura 5.6. No exemplo, a aplicação do conjunto de caracterização somente irá resultar na adição de dois nós filhos em cada um dos nós folhas da árvore de decisão. No entanto, isto é uma peculiaridade deste exemplo.

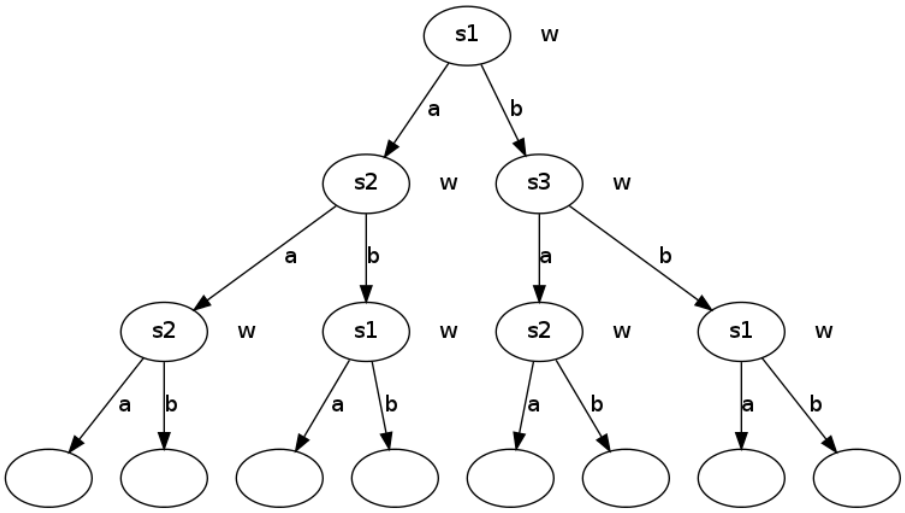

Figura 5.6: Aplicação do Conjunto de Caracterização

Por fim, para obter o conjunto de sequências necessárias para testar todas as transições da máquina, deve-se percorrer a árvore da raíz a cada um dos nós folhas, anotando o valor das arestas. Dessa forma, para o exemplo, são obtidas as sequências apresentadas na Tabela 5.6. Uma vez que o conjunto possui no total 32 caracteres, considera-se que o conjunto de sequências de teste tem tamanho 32.

Tabela 5.6: Conjunto de Sequências

'raaa'
'raab'
'raba'
'rabb'
'rbaa'
'rbab'
'rbba'
'rbbb'

Considere a máquina apresentada na Figura 5.7. O conjunto de caracterização para a máquina é $w = \{y, x, yy\}$. Como y é prefixo de yy, pode-se considerar como conjunto de caracterização apenas $w = \{x, yy\}$.

A partir do modelo fornecido, construi-se a árvore de execução da máquina, a qual é ilustrada na Figura 5.8. Aplicando o conjunto de caracterização em cada um dos nós da árvore de execução, obtem-se a árvore da Figura 5.9. Por fim, percorrendo a árvore da raíz a cada nó folha, obtem-se o conjunto de sequência de teste ilustrado na Tabela 5.7.

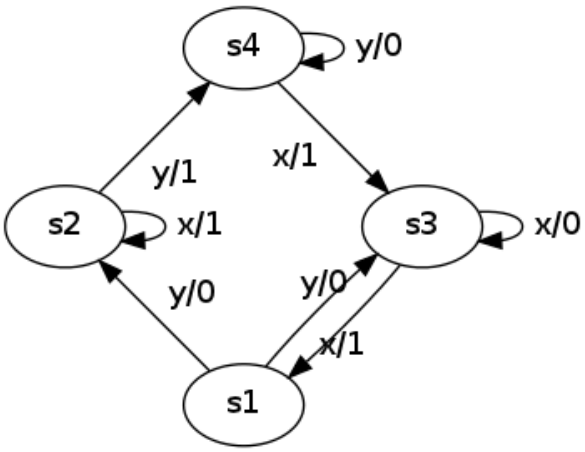

Figura 5.7: Máquina de Estados

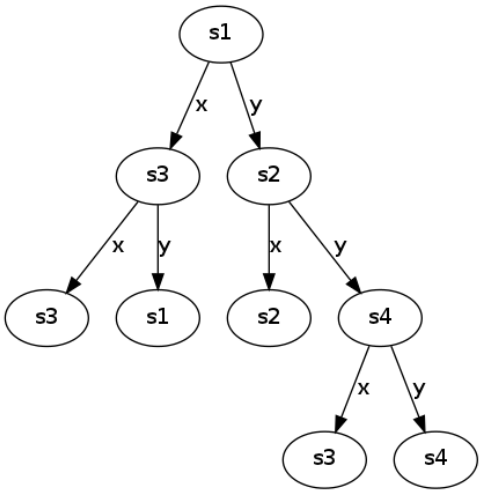

Figura 5.8: Árvore de Execução

Tabela 5.7: Sequências de Teste

'rxxx'
'rxxyy'
'rxyx'
'rxyyy'
'ryxx'
'ryxyy'
'ryyxx'
'ryyxyy'
'ryyyx'
'ryyyyy'

O conjunto de teste obtido possui tamanho 49. Na próxima seção, descreve-se um método visto em aula, o qual permite provar que a sequência de tamanho 12 é suficiente para testar todas as transições da máquina.

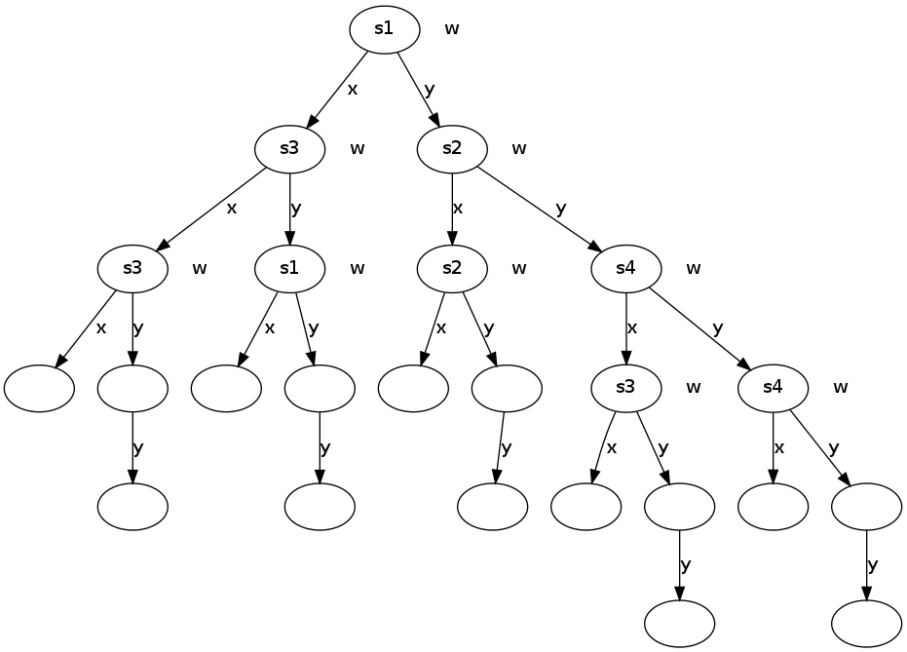

Figura 5.9: Aplicação de w

5.4 Cobertura de um Conjunto de Sequências

O conjunto de sequências de tamanho 12, apresentado na Tabela 5.8 é suficiente para testar todos os estados da máquina.

Tabela 5.8: Sequências de Teste

'rxyyxy'
'ryyyyyyxyyy'

Inicialmente, é necessário construir a árvore de execução a partir do conjunto de sequências, conforme ilustrado na Figura 5.10. Na árvore, cada estado é nomeado com letras de `A` a `P`, pois não se sabe se os estados e transições da implementação correspondem aos estados da especificação (máquina de estados). Além disso, por meio da especificação, pode-se obter as saídas esperadas ('0' ou '1') para cada entrada ('x' ou 'y') fornecida durante a execução. Tais entradas e saídas também são anotadas na árvore de execução.

Com a árvore de execução construída é possível descobrir que alguns estados devem ser distintos. Por exemplo, considerando o estado `A`, ao aplicarmos a sequência de entrada 'yy', a saída esperada é '01'. Por outro lado, ao aplicar a mesma sequência de entrada no estado `B`, a saída esperada é '00', diferente da saída esperada para o estado `A`. Conclui-se que os estados `A` e `B` devem ser distintos.

O mesmo raciocício não pode ser realizado para o estado `C`, pois não é possível, por meio da árvore, encontrar uma sequência que distingue os estados `A` e `C`. No entanto, para os estado `D`, pode-se considerar a sequência de entrada 'xy', cuja saída esperada é '10'

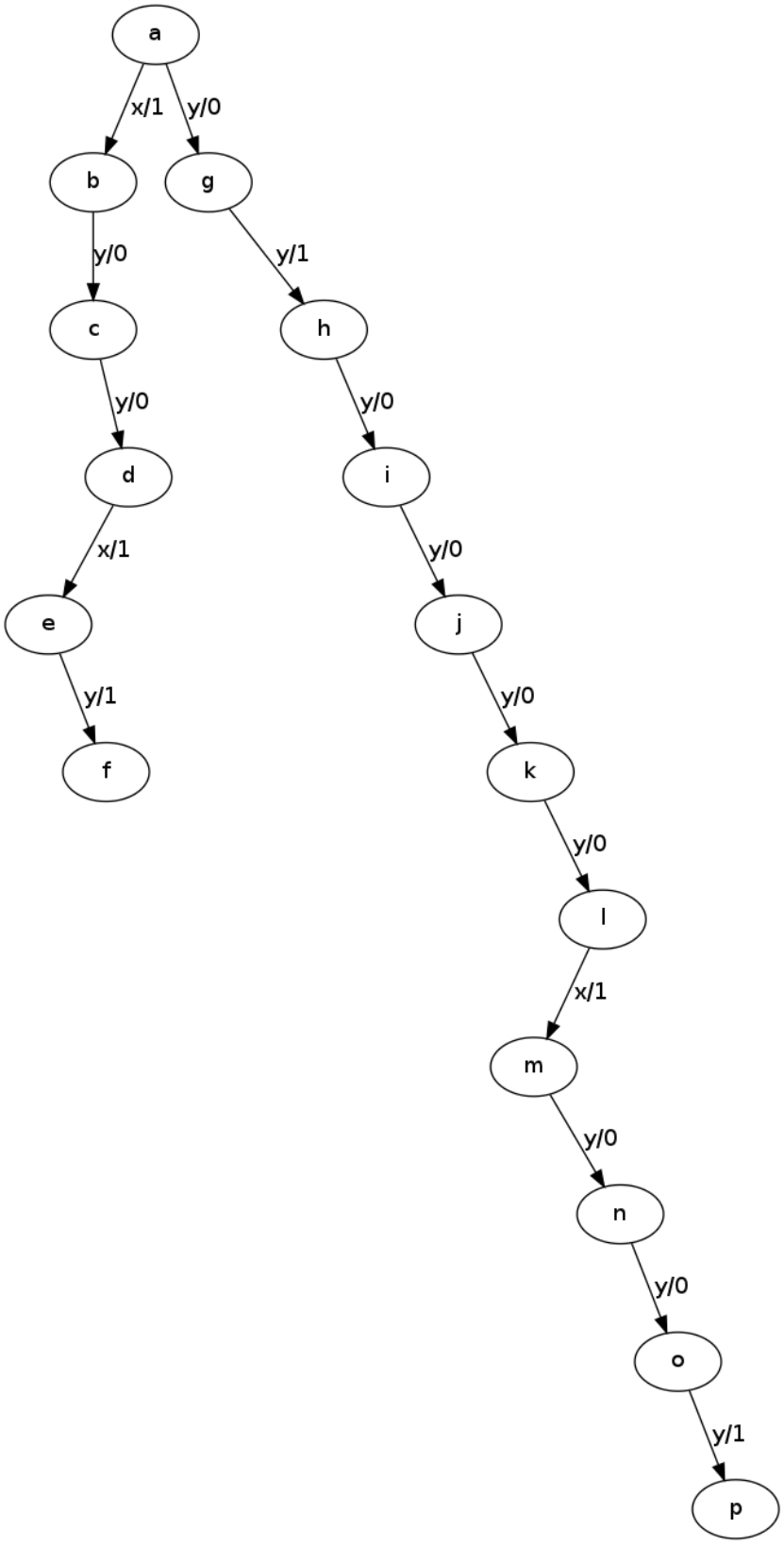

Figura 5.10: Árvore de Execução

para o estado `A` e '11' para o estado `D`. Assim, conclui-se que os estados `A` e `D` também devem ser distintos.

Aplicando esse raciocínio entre `A` e todos os demais estados, pode-se concluir que o estado `A` é distindo em relação aos estados `B`, `D`, `E`, `G`, `H`, `I`, `J`, `M` e `O`. Assim, representa-se a distinção entre esses estado por meio de um grafo de distinção, ilustrado na Figura 5.11.

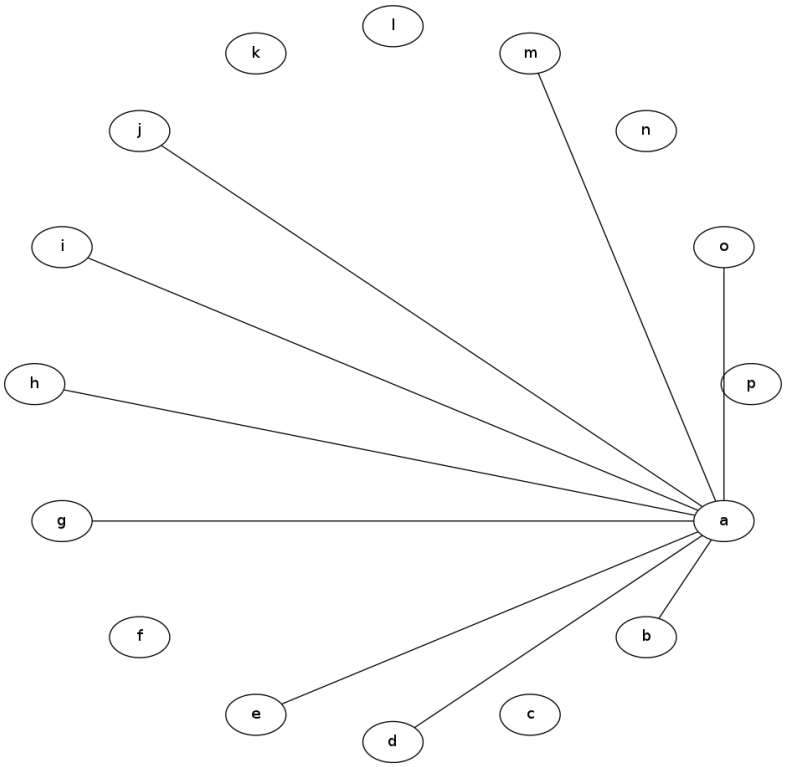

Figura 5.11: Grafo de Distinção para o Estado `A`

O mesmo processo pode ser aplicado para o estado `B`. Em seguida, pode ser aplicado para o estado `C`. E assim por diante até verificar as possíveis distinções entre estados para cada um dos estados da árvore. Após verificar todas as possíveis distinções de estados, obtem-se o grafo de distinção ilustrado na Figura 5.12.

Ao analisar o gráfico de distinção, identifica-se grupos de quatro estados distintos, como é o caso dos estados `A`, `G`, `H` e `M`. Sabendo que a máquina de estados em questão possui quatro estados, conclui-se que estes quatro estados correspondem aos quatros estados da máquina. Para registrar tal conclusão, rotula-se com quatro símbolos diferentes, como ilustrado na Figura 5.13.

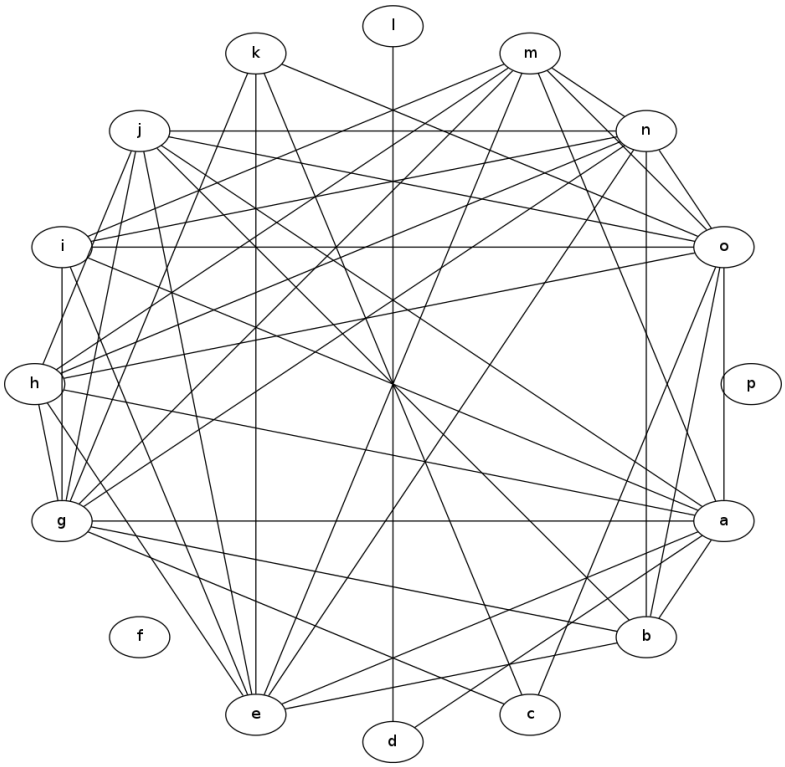

Figura 5.12: Grafo de Distinção

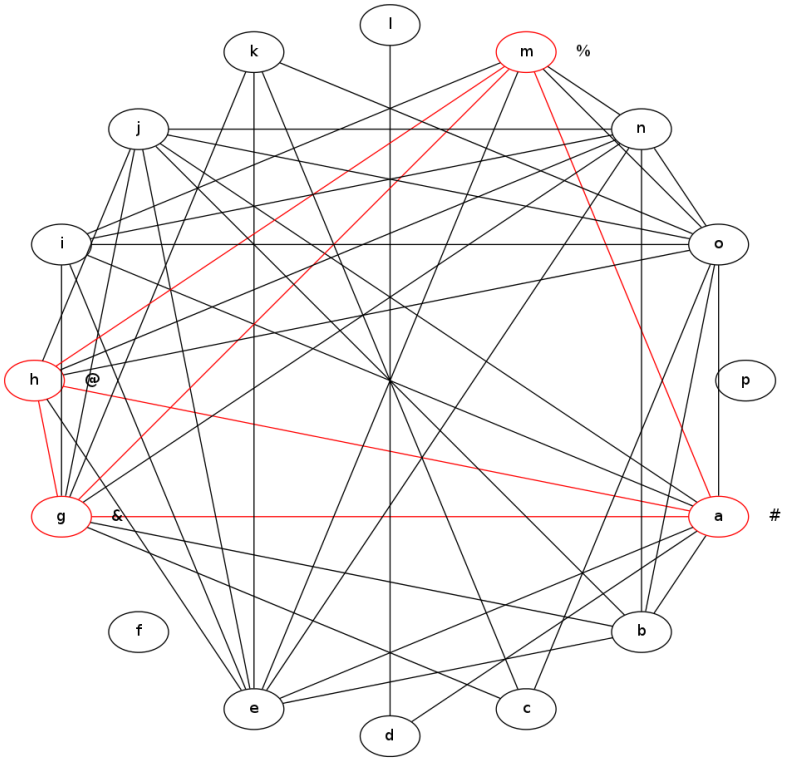

Figura 5.13: Grafo de Distinção: Relação entre os Estados `A`, `G`, `H` e `M`

Seguindo o mesmo raciocínio, observa-se que o estado `E` também é distinto em relação aos estados `A`, `H` e `M`. Como `A` é `#`, `H` é `@` e `M` é `%`, conclui-se que `E` é `&` (Figura 5.14). Continua-se com esse raciocínio até obter o grafo ilustrado na Figura 5.15.

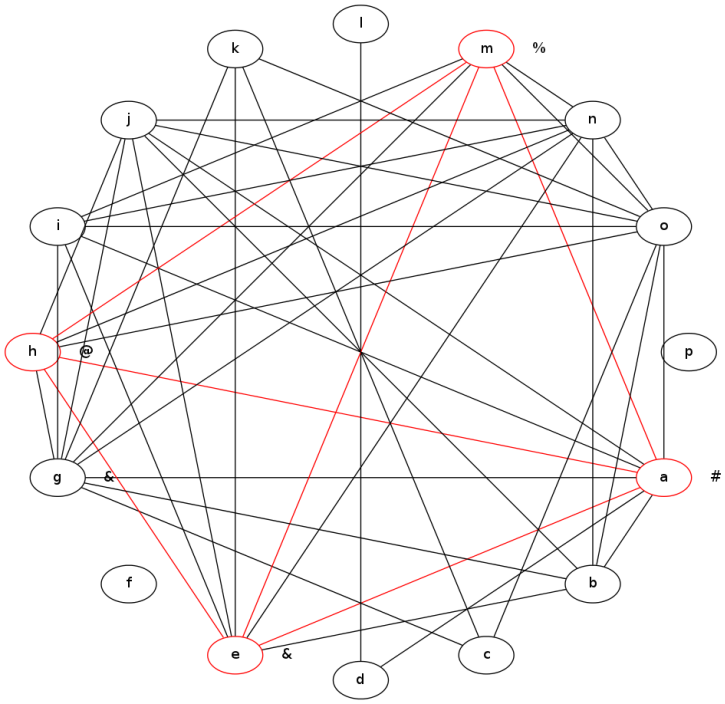

Figura 5.14: Grafo de Distinção: Rotulação do Estado `E`

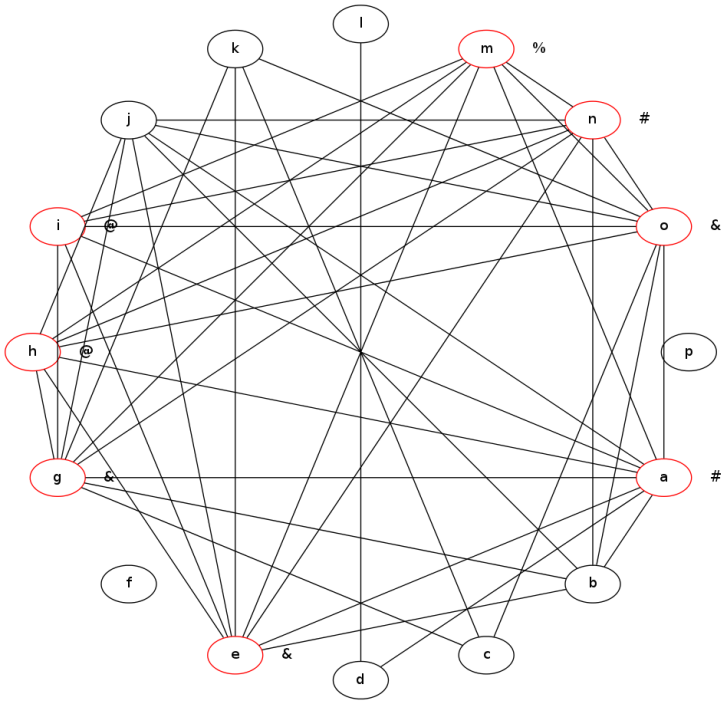

Figura 5.15: Grafo de Distinção: Possíveis Rotulações de Estados

Uma vez rotulados os estados do grafo de distinção, pode-se também rotular os estados na árvore de execução, conforme ilustrado da Figura 5.16.

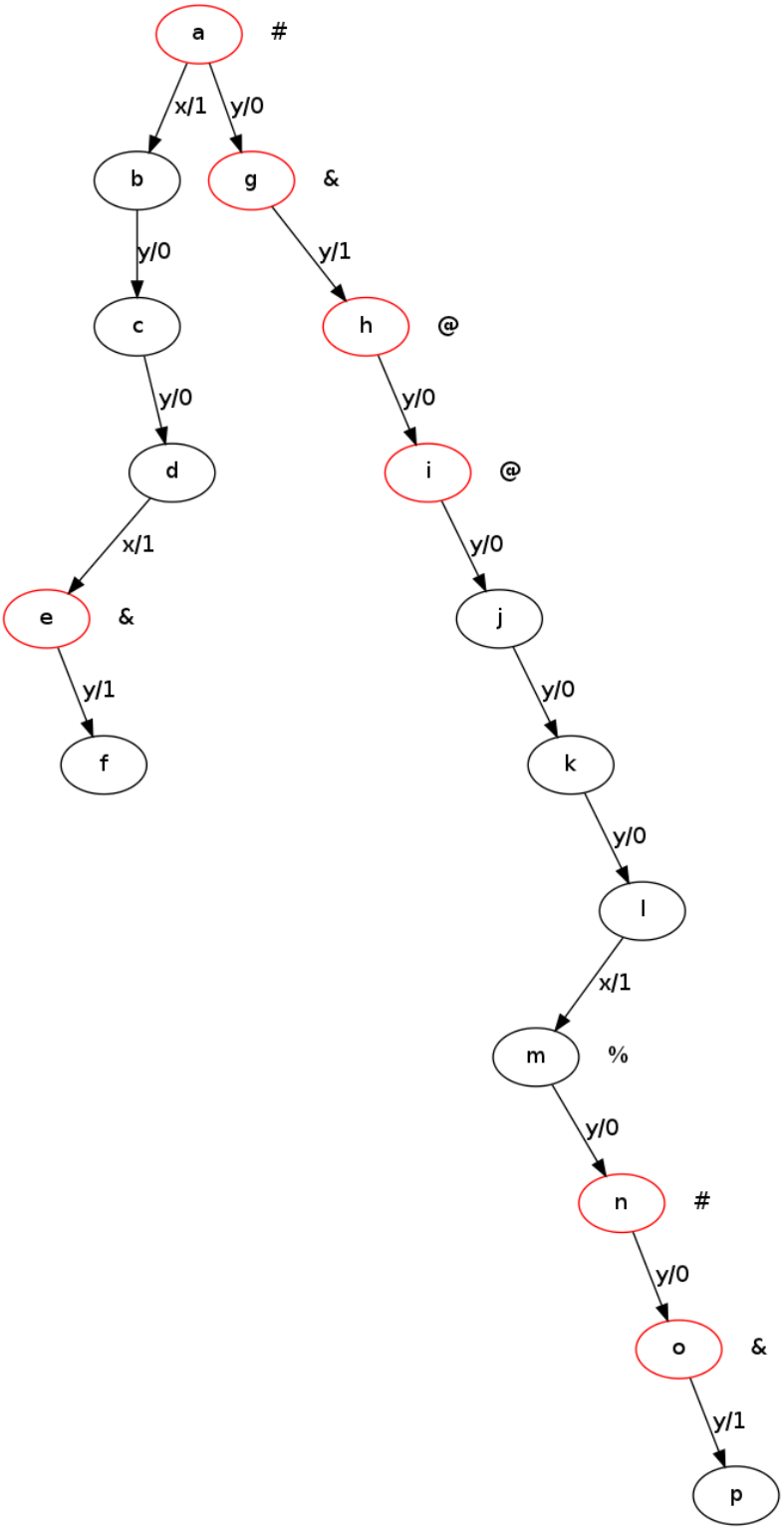

Figura 5.16: Árvore de Execução: Rotulação dos Estados

Após rotulados, verifica-se que os estados `H` e `I` são o mesmo estado `@`. Verifica-se que quando a máquina está no estado `@` e recebe uma entrada igual a 'y', ela deve continuar no estado `@`. Assim, conclui-se que os estados `J`, `K` e `L` também devem ser `@` (Figura 5.17).

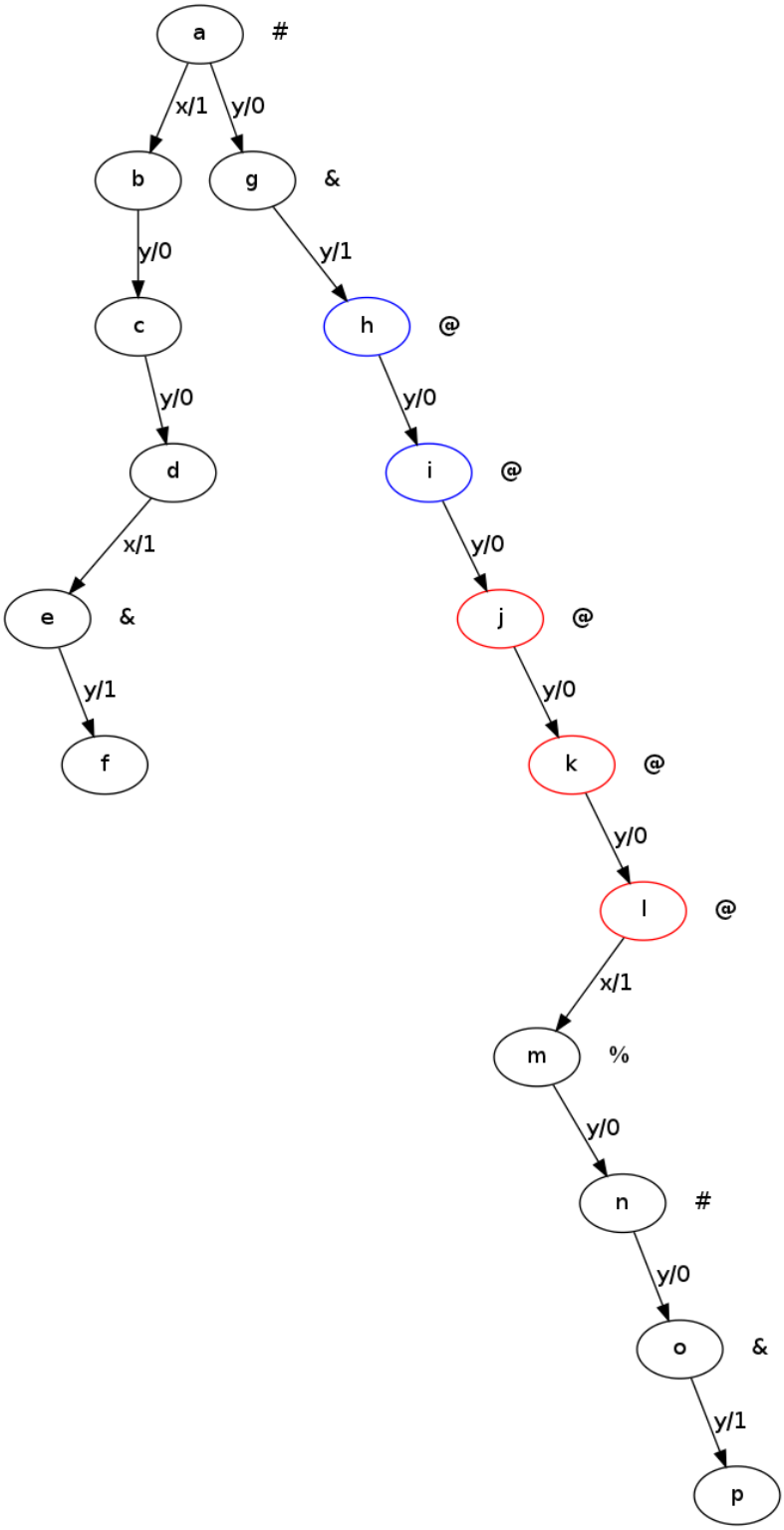

Figura 5.17: Árvore de Execução: Rotulação dos Estados `j`, `k` e `l`

De forma análoga, observando `G` e `H`, conclui-se que quando a máquina estiver no estado `&` e recebe entrada 'y', ela deve ir para o estado `@`. Como `E` e `O` também são `&`, conclui-se que `F` e `P` também devem ser `@` (Figura 5.18).

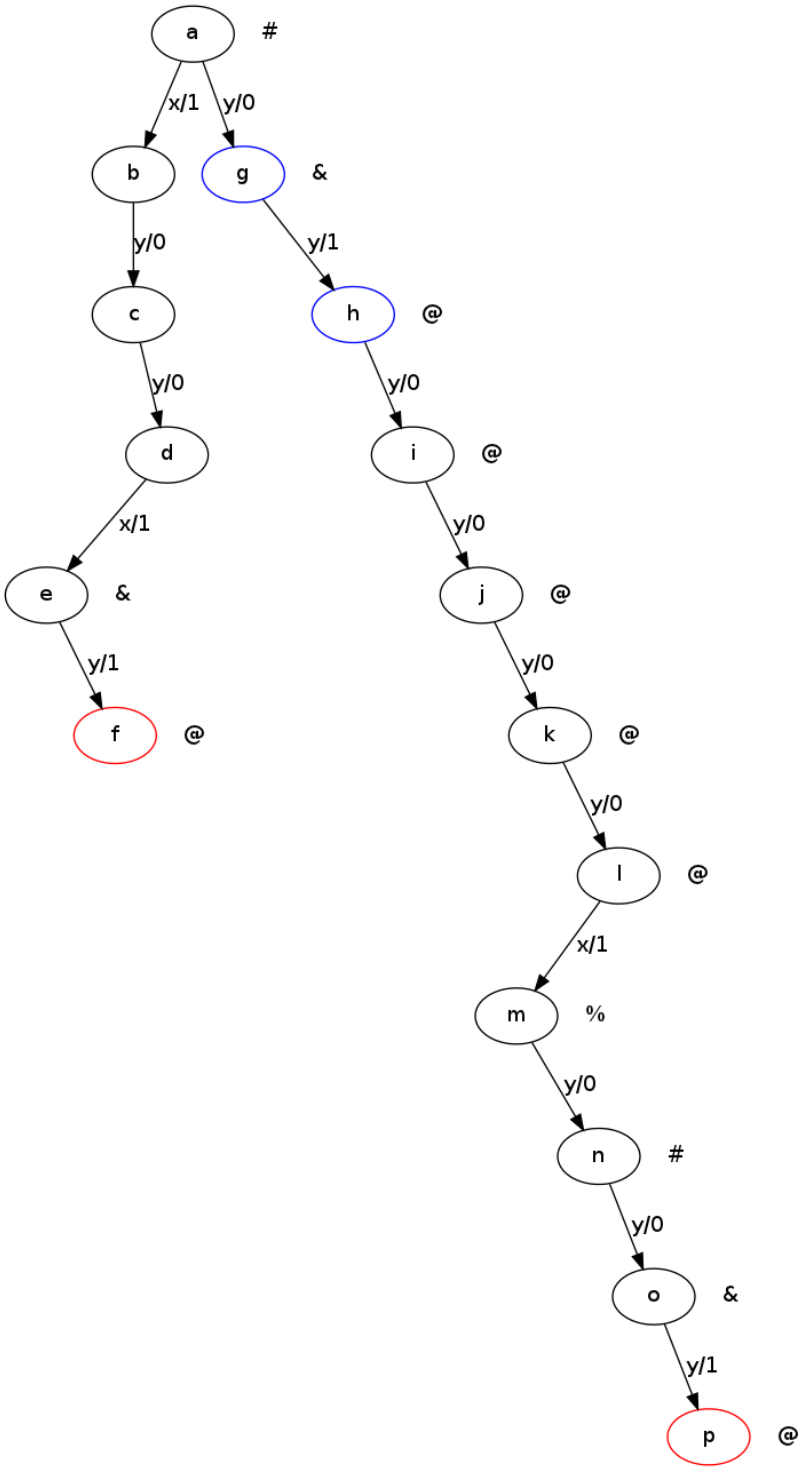

Figura 5.18: Árvore de Execução: Rotulação dos Estados `F` e `P`

Rotulando esses estados no grafo de distinção, pode-se concluir que `B` é `$` (Figura 5.19).

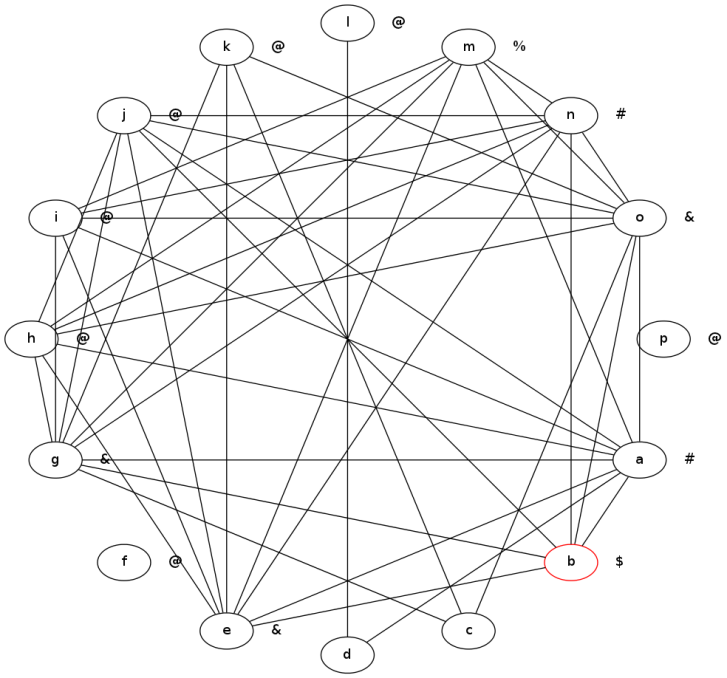

Figura 5.19: Grafo de Distinção: Rotulação do Estado `B`

Rotulando `B`, na árvore de execução, pode-se concluir que `C` é `#`. Se `C` é `#`, conclui-se que `D` é `&` (Figura 5.21).

Uma vez rotulado todos os estados, pode-se construir uma máquina de estados com os rótulos e transições identificadas na árvore de execução e no grafo de distinção. A máquina resultante é ilustrada na Figura 5.20.

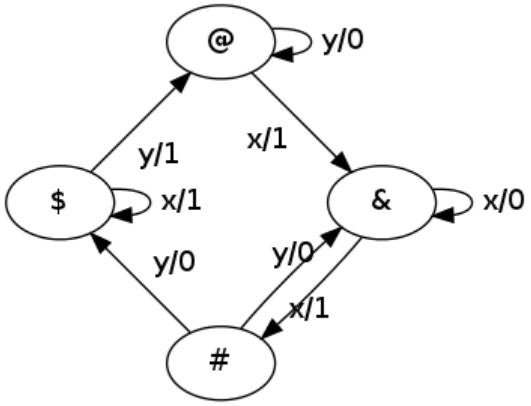

Figura 5.20: Máquina de Estados Construída por meio da Árvore de Execução

Como é possível perceber, ela corresponde exatamente a máquina da especificação. Isto mostra que para uma implementação satisfazer as sequências de teste propostas, ela deve se comportar exatamente como a máquina de estados da especificação. Em outras

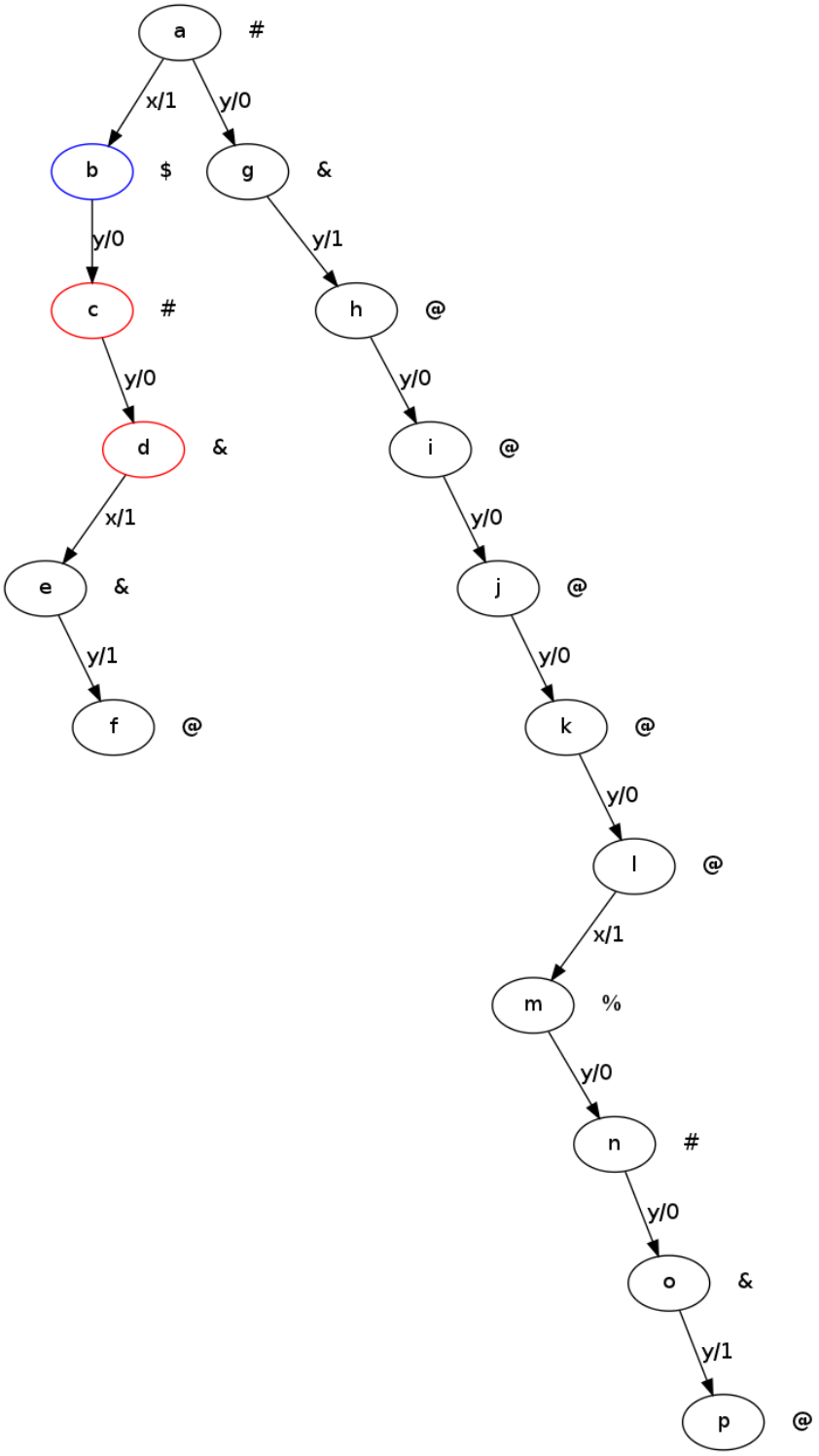

Figura 5.21: Árvore de Execução: Rotulação dos Estados `C` e `D`

palavras, esse conjunto de sequências de tamanho 12 é suficiente para exercitar todas as transições da máquina de estados.

5.5 Considerações Finais

Neste capítulo foram aboradados métodos de teste formal com a apresentação de métodos para a geração de sequência de verificação para máquinas de estado, encerrando o conteúdo dessa nota didática.

Referências

Butler, R. W. What is formal methods? NASA Langley Formal Methods Site, http://shemesh.larc.nasa.gov/fm/fm-what.html, 2001.

Hall, A. Seven myths of formal methods. *IEEE Softw.*, v. 7, n. 5, p. 11–19, 1990. Disponível em `http://dx.doi.org/10.1109/52.57887`

Huth, M. R. A.; Ryan, M. D. *Logic in computer science, modelling and reasoning about systems*. New York, NY, USA: Cambridge University Press, 2000.