Capítulo 3: “Satisfiability Model Theories” (1ª Parte)

Introdução

Neste capítulo procura-se alargar as metodologias SAT à Lógica de 1ª Ordem (FOL). Dado que a FOL tem um poder expressivo muito superior à Lógica Proposicional, ou qualquer outra lógica de ordem 0, será possível descrever sistemas mais complexos e formalizar problemas mais próximas de situações reais.

O obstáculo está, obviamente, no facto de a FOL, assim como a aritmética “standard”, serem lógicas indecidíveis. Por isso, em princípio, não existe qualquer algoritmo que possa certificar a validade de uma fórmula genérica de FOL. Por isso, também, não existe nenhum algoritmo universal de SAT que aceite uma fórmula arbitrária de FOL e não será possível construir uma ferramenta computacional universal para FOL e o problema SAT.

Isso não impede que em certos fragmentos da FOL e ou da aritmética não existam tais algoritmos de SAT. Os fragmentos da FOL e da aritmética onde a estrutura das fórmulas e a sua semântica, podem ser exploradas para construir um algoritmo de SAT correto e completo designam-se por SMT’s (“satisfiability modulo theories”).

A referência SMT-LIB contém uma classificação das principais SMT’s usadas nas aplicações mais usuais. Na terminologia aqui usadas estas SMT’s designam-se também como **teorias base** (“background theories”) . Existem teorias base para uma grande variedade de tipos de dados atómicos ou estruturados. Por exemplo: booleanos, inteiros, strings, vetores, conjuntos.

> Ver a instalação dos “solvers” Z3 e MathSAT como descrito na secção 7 no documento +Ferramentas Computacionais . Qualquer um destes “solvers" inclui várias teorias contidas na SMT-LIB.

Essencialmente uma teoria base suporta um fragmento decidível da aritmética conjuntamente com algumas operações que são transformáveis em operações nos inteiros.

O capítulo 26 do livro "Handbook of Satisfiability" tem uma introdução bastante completa à prática de construir algoritmos SAT específicos às várias teorias base.

Um exemplo: problema do centróide.

Este exemplo é semelhante ao problema dos k-centros que vimos no +Capítulo 2: Programação com Restrições .

A flexibilidade da aplicações dos SMT's é bem ilustrada na gama de problemas que pode resolver. Um exemplo, algo extremo, é a sua aplicações a problemas da Teoria da Aprendizagem.

Genericamente, como vimos anteriormente, em tais problemas é dado um conjunto finito de **experiências**. Neste exemplo, o "dataset" é formado por N palavras de bits de tamanho n

$$W \equiv \{ w_i \in \{0,1\}^n \}_{i=1\cdots N}$$

A partir destes dados, pretende-se "aprender" algo sobre o processo que os gerou.

O exemplo mais simples consiste no cálculo do **centroide** de W: isto é, o vetor $x \in \{0,1\}^n$ que minimiza a excentricidade de x ao conjunto W.

Nestes problemas, onde a domínio é um vetor de bits de tamanho fixo, as distâncias entre pontos ou entre pontos e conjuntos, baseia-se nas noções genéricas de **distância e peso de Hamming.**

Seja D um qualquer domínio com a estrutura algébrica de um grupo aditivo e em que a igualdade $=$ é uma relação decidível. Isso inclui os grupos finitos e os sub-grupos aditivos dos inteiros Z ou dos racionais Q ; mas não inclui, por exemplo, os reais R.
Então

1. Para vetores $x, y \in D^n$, a sua **distância de Hamming** é o número de posições onde os vetores diferem
$$\mathbf{d}(x,y) \equiv |\{ i \mid x_i \neq y_i \}|$$
2. O **peso de Hammig** de $x \in D^n$ define-se como a sua distância em relação ao vetor 0^n .

$$|x| \equiv \mathbf{d}(x, 0^n) = |\{i \mid x_i \neq 0\}|$$

Uma vez escolhidos o domínio D e uma distância qualquer $\mathbf{d}$ em D^n, não necessariamente a distância de Hamming, então sejam $x \in D^n$ e $W, X \subseteq D^n$ quaisquer elemento e subconjuntos finitos de D^n

1. Define-se a distância entre um vetor $x \in D^n$ e um conjunto $W \subseteq D^n$ como a menor das distâncias de x aos diversos elementos de W: isto é, a distância de x ao elemento "mais próximo" em W.
$$\mathbf{d}(x, W) \equiv \min \{ \mathbf{d}(x, y) \mid y \in W \}$$

2. Define-se a **excentricidade** de x em relação a W, como a distância de x e o elemento "mais longínquo" em W
$$\mathbf{ex}(x, W) \equiv \max \{ \mathbf{d}(x, y) \mid y \in W \}$$

Problema do centróide

> *Dado um "data set" W determinar um vetor x que minimiza a excentricidade de x em relação a W.*

$$\hat{W} \equiv \min \{ \mathbf{ex}(x, W) \mid x \in \{0, 1\}^n \}$$

```
from z3 import *
import random as rn

n = 16

def hamm(x,y):
     return Sum([(x[i]+y[i])%2 for i in range(n)])
def expl():
    return rn.choices([0,1],k=n)

x  = IntVector('x',n)
e  = Int('e')
```

```
# data  set

W = {}
m = 32
for j in range(m):
    W[j] = expl()

# constraints

s = Solver()
for i in range(n):
    s.add(x[i] >= 0,  x[i] <= 1)

for j in range(m):
    s.add( hamm(x,W[j]) <= e)
```

O código Z3 acima representa a formalização da definição da excentricidade de um "data-set" W. Os vetores exemplo são codificados como `IntVec` de n componentes.
A distância de Hamming entre dois vetores é codificada calculando a paridade da soma, componente a componente, e somando estes valores.
Para obter o máximo valor das diversas distâncias, introduz-se uma variável `e` que representa esses máximo e introduz-se restrições (uma por cada elemento do "data set") que força a distância da incógnita `x` a esse elemento seja limitada por `e`.

O objetivo seria agora minimizar o valor de `e`. Como as restrições são não-lineares o Z3 não admite um critério de optimização direto. É necessário fazer uma pseudo-optimização em que se percorre todos os possíveis valores `t` que `e` pode assumir, de forma crescente.
Para cada um desses valores introduz-se a restrição `e == t` e tenta-se encontrar uma solução para o problema; o algoritmo para quando a solução é encontrada.

```
# Pseudo-Optimization
for t in range(1,n):
```

```
    s.push()
    s.add(e == t)
    if  s.check() == sat:
        print(t,"\n",s.model())
        break
    s.pop()
```

Usando precisamente a mesma abordagem pode-se tentar modelar os vetores exemplo por vetores de bits; no Z3 e no PySMT tais vetores estão na classe `BitVec`.

Vamos aqui usar uma implementação em PySMT par se poder explorar a possibilidade de comparar a eficiência do "solver" Z3 com a do "solver" MatSAT.

A implementação usada para a implementação do peso de Hamming está explicada no site `StackOverflow`.

```
from pysmt.shortcuts import *
import pysmt.typing as types
import random as rn
0
n = 16      # dimensão do espaço de amostras
m = 64      # numero de amostras

# Escolha de "solver"
name = "msat"
#name = "z3"

# Funções auxiliares para BitVec's
def bv_rn():        # gera pseudo-aleatoriamente um BitVec de t
amanho "n"
    a = rn.getrandbits(n)
    return BV(a,n)
```

```
def bv_sel(z,i):                    # seleciona o i-ésimo bit
do BitVec "z"
    return BVZExt(BVExtract(z,start=i,end=i),n-1)

def bv_hamm(z,y=None):              # distância e peso de Hamm
ing de BitVec
    if y != None:
        return bv_hamm(BVXor(z,y))
    return sum([bv_sel(z,i) for i in range(n)])

# Dataset
W = [bv_rn() for _ in range(m)]

# Variáveis e Problema
x = Symbol('x',types.BVType(n))
e = Symbol('e',types.BVType(n))
problem = And([(bv_hamm(w,x) <= e)  for w in W])

# Solução e Pseudo Optimização
with Solver(name=name) as solver:
    solver.add_assertion(problem)
    for t in range(1,n):
        solver.push()
        solver.add_assertion(e <= t)
        if solver.solve():
            print(solver.get_model())
            break
        solver.pop()
```

Aplicações das SMT's

As principais áreas de aplicação das SMT's são:

- A verificação das propriedades lógicas/temporais de **sistemas dinâmicos;** isto é, sistemas que evoluem com o tempo e cujas propriedades lógicas envolvem a grandeza "tempo".
- A verificação da correção e terminação de **programas imperativos**.

As propriedades dos sistemas dinâmicos são, de uma forma geral, o objetivo deste capítulo e do capítulo seguintes. A análise de programas imperativos será estudada no último capítulo.

Neste capítulo vamos apresentar as estruturas e metodologias básicas na
análise do comportamento de sistemas dinâmicos.
Para isso vamos descrever dois tipos de modelos de sistemas dinâmicos

- "Finite State Machines" (FSM's), e
- "First Order Transition Systems" (FOTS).

Sistemas dinâmicos concretos, derivados dos modelos básicos, serão estudados no próximo capítulo.

Máquinas de Estados Finitas (FSM's)

Sucintamente uma máquina de estados finita é um autómato onde, como o nome sugere, o espaço de estados tem cardinalidade finita.

Formalmente uma FSM $\Sigma \equiv \langle Q, I, \delta \rangle$ é um triplo de conjuntos em que

- Q é um conjunto finito de elementos designados por **estados**.
- $I \subseteq Q$ é um conjunto de estados designados por **iniciais.**
- $\delta \subseteq Q \times Q$ é uma relação binária designada por **relação de transição;** a relação também se representa como

$$s \rightarrow s' \quad \text{sse} \quad (s, s') \in \delta$$

Terminologia e notação relativos à FSM $\Sigma \equiv \langle Q, I, \delta \rangle$

- A relação δ^* é o **fecho transitivo** de δ define-se como a menor relação que verifica

$$(x,y) \in \delta^* \quad \text{sse} \quad (x = y) \vee \exists\, z \in Q \cdot (x,z) \in \delta \wedge (z,y) \in \delta^*$$

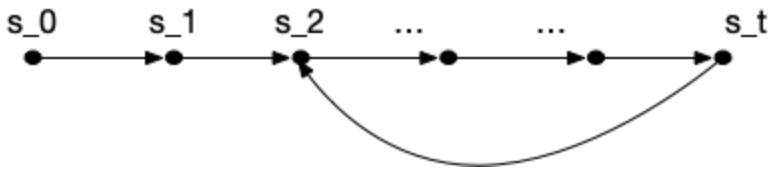
- Um **traço** é uma sequência

$$\alpha \equiv \langle s_0, s_1, \cdots, s_k, \cdots \rangle$$

finita ($\alpha \in Q^*$) ou infinita ($\alpha \in Q^\infty$)de estados tais que $s_i \rightarrow s_{i+1}$ para todo $i \geq 0$.

Se $s_0 \in I$ o traço diz-se "**rooted**".

- Um traço é **limitado** ("bounded") se o conjunto de todos os seus estados distintos é finito. É evidente que todo o traço finito é limitado; no entanto existem também traços infinitos que são limitados. Um traço infinito e limitado contém necessariamente um "**loop**"



Este exemplo ilustra um traço infinito com um número de estados distintos menor ou igual a $t+1$.
A figura ilustra um traço em que existe uma transição $s_t \rightarrow s_\ell$ com $\ell < t$.

É evidente que numa FSM, dado que o espaço de estados é sempre finito, todos os traços são limitados. No entanto mantemos esta designação porque a terminologia dos traços se estende a outro tipo de autómatos onde o espaço de estados Q é contável mas é infinito.

- O conjunto dos traços "rooted" designa-se por **comportamento** de Σ e é também representado por Σ.
Dado um sub-conjunto de estados iniciais $J \subset I$ representamos por Σ_J o conjunto dos traços em que o estado inicial s_0 pertence a J.

- Dado um traço $\alpha \equiv \langle s_0, s_1, \cdots, s_i, s_{i+1}, \cdots \rangle \in \Sigma$ representamos por $\alpha^{(i)}$ o sufixo de ordem i de α; isto é a sequência $\alpha^{(i)} = \langle s_i\ ,\ s_{i+1}\ ,\ \cdots \rangle$.

- Em particular, representamos por α' o sufixo $\alpha^{(1)} \equiv \langle s_1\ ,\ s_2\ ,\ \cdots \rangle$ que se obtém retirando de α o seu primeiro elemento.

  Em geral, se $A \subseteq \Sigma$ é um sub-comportamente de Σ determinado por algum conjunto de restrições, representamos por A' o conjunto de todos os traços que se obtêm retirando o primeiro elemento de todos os traços em A.

$$A' \equiv \{\, \alpha' \mid \alpha \in A \,\}$$

Veremos adiante que o operador $A \mapsto A'$ é uma das construções fundamentais em qualquer descrição dos modelos em lógicas que lidam explicitamente com a dimensão tempo (aka *lógicas temporais*).

- Escreve-se $s \rightarrow_\alpha r$ quando $s \rightarrow r$ estão em transição direta ao longo do traço $\alpha \in \Sigma$. Esta notação generaliza-se para $s \rightarrow_A r$ quando $A \subseteq \Sigma$ é um sub-comportamento.

"Labelled Finite State Machines" (LFSM)

O principal objetivo, na modelação de um sistema dinâmico, é a verificação das suas propriedades lógicas.

A forma mais simples de descrever essas propriedades, e o seu significado lógico, passa por aumentar a descrição de uma FSM $\Sigma \equiv \langle Q\ ,\ I\ ,\ \delta \rangle$ com um conjunto de proposições atómicas L, que aqui se designam por **labels**, e por uma **relação de "labeling"** $\ell \subseteq Q \times \mathrm{L}$ que liga estados $s \in Q$ com as "labels" $p \in \mathrm{L}$ que são consideradas válidas em s.

$(s, p) \in \ell$ é interpretado como *"a proposição p é válida no estado s".*

A entidades resultante

$$\Sigma_{\mathrm{L}} \equiv \langle Q, I, \delta, \mathrm{L}, \ell \rangle$$

designa-se por **"labelled finite state machine" (LFSM).**

"Error Transition Systems" (ETS)

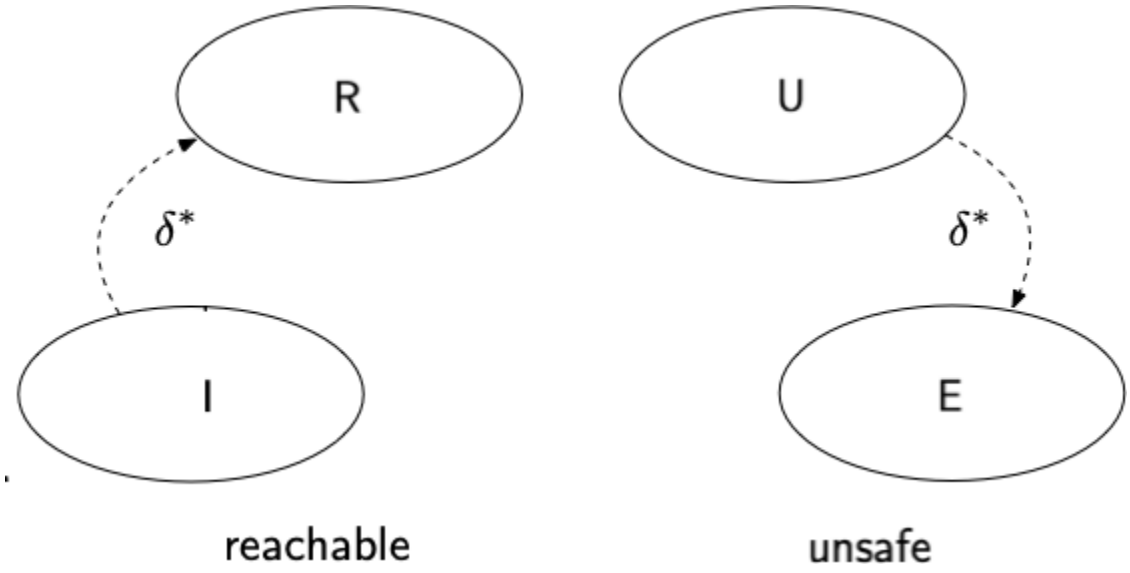
Os ETS são uma outra forma de estender o modelo básico $\Sigma \equiv \langle Q, I, \delta \rangle$. A este modelo acrescenta-se um segundo sub-conjunto $E \subseteq Q$ cujos elementos $e \in E$ se designam por **estados de erro** ou simplesmente **erros.**

No modelo $\Sigma_E \equiv \langle Q, I, \delta, E \rangle$ define-se os seguintes conceitos

- Um estado $r \in Q$ diz-se **acessível** ("reachable") se $r \in I$ ou então existe uma transição $(s, r) \in \delta$ em que s é acessível.
  O conjunto R dos estados acessíveis pode ser definido recursivamente como o menor conjunto que verifica
  $$r \in R \equiv \quad r \in I \ \vee \ \exists\, (s, r) \in \delta \cdot s \in R$$

- Um estado $e \in Q$ diz-se **inseguro** ("unsafe") se se tem $e \in E$ ou então existe uma transição $(e, s) \in \delta$ em que s é inseguro.
  O conjunto U dos estados inseguros pode ser definido recursivamente como o menor conjunto que verifica
  $$e \in U \equiv \quad e \in E \ \vee \ \exists\, (e, s) \in \delta \cdot s \in U$$

- O ETS Σ_E é **inseguro** quando existe algum estado s que seja simultâneamente acessível e inseguro. Equivalentemente o sistema é inseguro se e só se
  $$R \cap U \neq \emptyset$$



Bounded Temporal Logic (BTL)

Genericamente uma lógica diz-se **temporal** quando a validade das suas fórmulas está relacionada como alguma noção de *tempo*. Nos sistemas dinâmicos descritos neste curso a noção de "tempo" é descrita essencialmente pelo comportamento de um sistema dinâmico através dos seus estados e dos seus traços. Por isso todo o sistema dinâmico deve estabelecer relações semânticas envolvendo estados s, traços α e fórmulas ϕ.

De fato qualquer lógica temporal deve definir uma ou mais das seguintes *relações semânticas*

- $s \models \phi$ *a fórmula* ϕ *é válida no estado* s
- $\alpha \models \phi$ *a fórmula* ϕ *é válida no traço* α
- $s, \alpha \models \phi$ *a fórmula* ϕ *é válida no estado* s do *traço* α

Para descrever as propriedades do comportamento de uma LFSM pode-se usar uma lógica temporal muito simples, designada por "Bounded Temporal Logic" (BTL).
Uma fórmula genérica $\phi \in \mathrm{BTL}$ tem uma das seguintes formas

$$\phi ::= p \mid \psi \rightarrow \varphi \mid \neg\psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \varphi$$

com $p \in \mathrm{L}$ e com $\psi, \varphi \in \mathrm{BTL}$.

As três primeiras alternativas descrevem, simplesmente, uma lógica proposicional com o espaço de símbolos L. Só as duas últimas introduzam comportamento temporal; essencialmente

- $\mathrm{X}\psi$ ("next ψ") é válida no estado presente se e só se ψ é válida no "próximo estado".
- $\psi \,\mathrm{U}\, \varphi$ ("ψ until φ") é válida no estado presente sse φ é válido em algum estado futuro e ψ é válido no presente e em todos os estados até φ ser válido.

A semântica desta lógica define-se de forma diferente consoante o modelo de estados é finito ou não. Assumindo traços limitados (com um número finito de estados distintos), tem-se

1. Com $p \in \mathrm{L}$ verifica-se $\alpha \models p$ sse p é válido no estado inicial do traço α.
$$\alpha \models p \quad \text{sse} \quad (\alpha_0, p) \in \ell$$

2. As fórmulas $\psi \rightarrow \varphi$ e $\neg\psi$ interpretam-se como na lógica proposicional.

3. $\alpha \models \mathrm{X}\psi$ sse $\alpha' \models \psi$

4. <u>No modelo de estado finitos</u> a semântica de $\phi \,\mathrm{U}\, \varphi$ é definida recursivamente; sendo α um traço limitado define-se

$$\alpha \models \psi \,\mathrm{U}\, \varphi \quad \text{sse} \quad \alpha \models \varphi \quad \text{ou então} \quad \alpha \models \psi \quad \& \quad \alpha' \models \psi \,\mathrm{U}\, \varphi$$

Isto é

$$\psi \,\mathrm{U}\, \varphi \;\equiv\; \varphi \;\vee\; (\psi \;\wedge\; \mathrm{X}(\psi \,\mathrm{U}\, \varphi))$$

Outros operadores temporais podem definidos à custa dos dois operadores X e U

- A fórmula $\mathrm{F}\phi$ (*"no **F**uturo"* ϕ) deve ser interpretada como:
"$\mathrm{F}\,\phi$ é válida agora sse existe um estado futuro onde ϕ é válido".
equivalentemente
$$\mathrm{F}\,\phi \;\equiv\; \phi \;\vee\; \mathrm{X}(\mathrm{F}\,\phi)$$

- A fórmula $\mathrm{G}\phi$ ("***G**lobalmente"* ϕ) deve ser interpretada como:

> $\mathrm{G}\,\phi$ *é válida agora sse* ϕ *é válida agora e em todos os estados futuros*

equivalentemente

$$\mathrm{G}\,\phi \equiv \phi \wedge \mathrm{X}(\mathrm{G}\,\phi)$$

Ambos operadores podem ser definidos à custa dos operadores temporais básicos X e U

- $\mathrm{F}\,\varphi \equiv \mathrm{True}\,\mathrm{U}\,\varphi$
- $\mathrm{G}\,\phi \equiv \phi\,\mathrm{U}\,\mathrm{False}$

O facto de esta semântica ser definidas apenas para traços limitados (i.e. o número de estados distintos em α é finito) faz com que a definição recursiva de $\alpha \models \mathrm{U}$ tenha sempre um valor lógico bem determinado. Portanto F e G também têm uma semântica bem determinada.

Quando o sistema dinâmico é modelado em lógica de 1ª ordem, como nos FOTS que veremos na secção seguinte, estas semânticas recursivas não definem corretamente os operadores $\mathrm{F}, \mathrm{G}\,\mathrm{e}\,\mathrm{U}$ e outra semântica é necessária. Analisaremos essa semântica quando apresentarmos os FOTS.

Exemplo (A)

Como exemplo de uma FSM podemos considerar a porção de código imperativo seguinte:

```
assert (z >= 0)
0: while z > 0:
1:    z = z - 1
2: stop
```

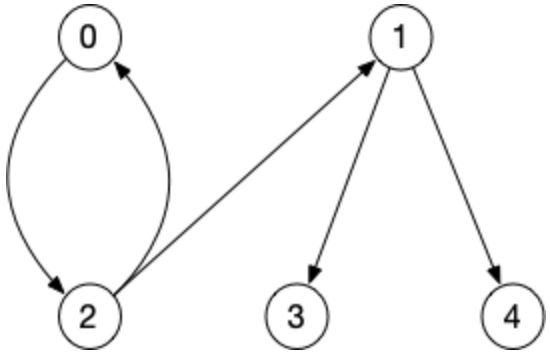
Neste programa assumimos que a variável $\mathbf{z}$ toma valores inteiros não-negativos. Em todo o programa imperativo está implícita uma variável, designada por "program counter", que identifica o comando que, em cada momento, está disponível para executar. Vamos aqui usar $\mathbf{c}$ para designar tal variável.

Vamos considerar uma FSM que vai servir de primeira aproximação à semântica do programa imperativo acima.

1. O par de variáveis $(\mathbf{z}, \mathbf{c})$ tem como domínio $\mathrm{N} \times \{0, 1, 2\}$ que vai servir de base ao espaço de estados. O espaço de estados $Q \equiv \{s_0, s_1, s_2, s_3, s_4\}$ é uma partição do domínio dos $(\mathbf{z}, \mathbf{c})$ dado por

$$\begin{array}{lcllcl} s_0 & \equiv & c = 0 \wedge z > 0 & , & s_1 & \equiv & c = 0 \wedge z \leq 0 \\ s_2 & \equiv & c = 1 & , & & & \\ s_3 & \equiv & c = 2 \wedge z = 0 & , & s_4 & \equiv & c = 2 \wedge z \neq 0 \end{array}$$

2. O conjunto de estados iniciais é $I \equiv \{s_0, s_1\}$.
3. A relação de transição δ é representada no seguinte grafo:



Na relação de transição δ, descrita por este grafo, verifica-se que

i. existe um "loop" infinito alternando entre os estados s_0 e s_2,
ii. o estado s_2 apresenta um não-determinismo com transições para os estados s_0 e s_1,
iii. de forma análogo o estado s_1 também apresenta um não-determinismo com transições para os estados s_3 e s_4.

Só estas observações fazem suspeitar que a FSM não é uma descrição fiel do programa imperativo acima. De facto a FSM tem muito mais traços que o programa; o comportamento da FSM inclui, mas não coincide com, o comportamento do programa.

Para clarificar este ponto pode-se introduzir propriedades lógicas na FSM e no programa e verificar que existem propriedades que não se verificam na FSM mas verificam-se no programa.

Informalmente a propriedade com que pretendemos estabelecer a distinção entre os dois modelos diz algo do tipo *"qualquer traço "rooted" do programa eventualmente para ao atingir* stop*".*

Parece intuitivo afirmar que a propriedade se verifica no programa mas, devido ao "loop" atrás mencionado, não se verifica na FSM.

Para introduzir semântica temporal na FSM vamos aumentá-la com "labels" L e um relação de "labelling" $\ell \subseteq Q \times \mathrm{L}$. O conjunto de "labels" é simplesmente

$$\mathrm{L} \equiv \{\mathrm{start}, \mathrm{stop}\}$$

A relação de "labelling" é

$$\ell \equiv \{(s, \mathrm{start}) \mid s \neq s_3\} \cup \{(s_3, \mathrm{stop})\}$$

A propriedade que queremos verificar, quer no programa quer na FSM, escreve-se

$$\mathrm{F}\ \mathrm{stop}$$

Mais precisamente queremos verificar que em qualquer traço α que seja "rooted" e pertença ao comportamento do sistema, se verifique a asserção $\alpha \models \mathrm{F}\ \mathrm{stop}$.

A semântica que temos para este programa imperativo diz-nos que, em qualquer traço, o programa termina. Portanto, no programa, a asserção é válida.

Porém a asserção não é válida na FSM. Para que a asserção se verificasse na FSM, qualquer traço iniciado no estado s_0 ou no estado s_1 teria de terminar no estado s_3 e isso não é verdade.

Exemplo (B)

Vimos, no exemplo anterior, um programa imperativo e uma FSM que o pretende descrever. Vimos que a FSM apresentada não reflete as propriedades temporais do programa.

Poder-se-ia tentar outra FSM mas, de facto, isso não resulta: nenhuma FSM consegue representar fielmente o programa.

Isso resulta de o programa usar uma varável $\mathbf{z}$ cujo domínio é todo N e, portanto, infinito. Só será possível descrever o programa imperativo por uma FSM se todos as variáveis estão estritas a domínios finitos.

De um modo geral todas as ferramentas, que validam propriedades temporais usando máquinas de estado finitas, exigem que as variáveis estejam limitadas a domínios finitos.

No problema apresentado no exemplo (A) a variável $\mathbf{c}$ (o "program counter") está limitada a um domínio finito; nomeadamente o domínio $Z_3 \equiv \{0,1,2\}$. Temos de limitar $\mathbf{z}$ de forma análoga: o domínio de valores distintos a variável será um domínio $Z_N \equiv \{0,1,\cdots,N-1\}$.

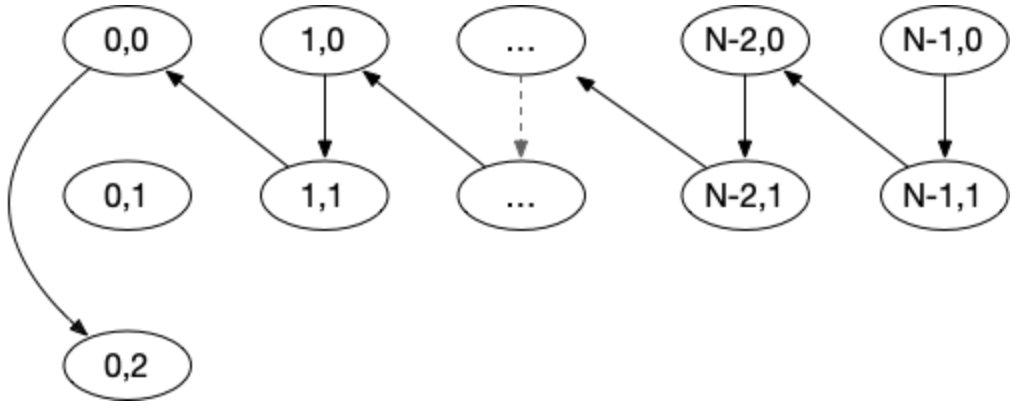
Para isso impõe-se que, no estado inicial, a variável $\mathbf{z}$ tenha um valor no domínio Z_N. Para isso o comando `assert` que ocorre no `Python` e em outras linguagens imperativas.

```
assert (0 <= z) and (z < N)
0: while z > 0:
1:     z = z - 1
2: stop
```

A nova FSM é definido por

1. Um espaço de estados $Q \equiv Z_N \times Z_3$ e os estados são pares $(\mathbf{z},\mathbf{c})$
2. Os estados iniciais são $I \equiv \{(n,0) \mid 0 \leq n < N\}$
3. A relação de transição δ é definida pelo seguinte conjunto de transições

$$\begin{cases} (0,0) \to (0,2) & , \\ (n,0) \to (n,1) & , \quad \text{se } N > n > 0 \\ (n,1) \to (n-1,0) & , \quad \text{se } N > n > 0 \end{cases}$$



O conjunto de "labels" mantém-se em $\mathrm{L} \equiv \{\mathrm{start}, \mathrm{stop}\}$ e a relação de "labelling" é análoga à anterior: $\ell(0,2) = \mathrm{stop}$ e $\ell(s) = \mathrm{start}$ para todo o estado $s \in \{(n,0) | n \in \mathrm{Z}_N\}$.

Pelo grafo da relação de transição vê-se que não existem ciclos e qualquer traço iniciado num dos estados iniciais termina sempre no estado $(0,2)$. Por isso, em qualquer traço "rooted" α desta FSM verifica-se

$$\alpha \models \mathrm{F}\ \mathrm{stop}$$

+Capítulo 3 : "Satisfiability Modulo Theories" (2ª Parte)