Capítulo 4: Verificação de Modelos baseada em SAT (2ª parte)

Transformadores de Predicados

Vamos considerar um SFOTS

$$\Sigma \equiv \langle T, X, \text{next}, I, T, E \rangle$$

e vamos representar por P_T (aka P_X ou simplesmente P) o conjunto dos predicados unários que podem ser definidos em T com variáveis X.

Por **transformador de predicados (t.p.)** designamos qualquer operador $\text{op} : P_X \rightarrow P_X$ que é possível definir aumentando T com o operador "substituição de variáveis por termos".

A substituição de X por um "clone" arbitrário (distinto de X) não é um transformador de predicados porque exige um predicado que já tem variáveis que não estão em X.

O sistema Σ cria dois transformadores de predicados que representam a relação T:

1. O transformador de predicados wp, designado por **weakest pre-condition** mapeia um predicado genérico Q no predicado $\text{wp}(Q)$ que verifica, para todo $P \in P$,

   $P \rightarrow \text{wp}(Q)$ é tautologia se e só se $P \wedge T \rightarrow Q'$ é tautologia

2. O transformador de predicados sp, designado por **strongest post-condition** mapeia um predicado genérico P no predicado $\text{sp}(P)$ que verifica, para todo $Q \in P$,

   $\text{sp}(P) \rightarrow Q$ é tautologia se e só se $P \wedge T \rightarrow Q'$ é tautologia

> *A vantagem da descrição do comportamento de Σ por transformadores de predicados reside no facto de evitar a introdução de novas variáveis distintas das de X. Note-se que a relação T já usa as variáveis X e um seu clone X'; assim a verificação de que $P \wedge T \rightarrow Q'$ é tautologia exige, pelo menos, estes dois conjuntos de variáveis.*
>
> *Também qualquer das metodologias apresentadas na 1ª parte deste capítulo (BMC, indução, SAT-based, etc..) usa não só estes dois conjuntos de variáveis mas também vários outros clones: um clone para cada estado que seja representado explicitamente. Como consequência o típico problema SAT, que formaliza esta verificação, tem uma dimensão do espaço global de procura que é enorme.*
>
> *Em alternativa a fórmulas $P \rightarrow \text{wp}(Q)$ e $\text{sp}(P) \rightarrow Q$ só usam as variáveis X o que torna muito mais simples a sua verificação com SAT.*

Pode-se definir em P uma ordem parcial

$$A \geq B \quad \text{sse} \quad B \rightarrow A \text{ é tautologia.}$$

Então, para todos $P, Q \in P$, tem-se

$$\text{wp}(Q) \geq P \quad \text{sse} \quad Q \geq \text{sp}(P)$$

em que ambas as asserções acima são equivalentes a "$\mathrm{P} \wedge \mathrm{T} \rightarrow \mathrm{Q}'$ é tautologia".

Este é um exemplo de um conceito muito importante em Ciências da Computação designado por *dualidade de Galois* ("Galois connection").

Alguns exemplos de transformadores de predicados

O aspecto negativo do uso de transformadores de predicados wp e sp em vez da relação de transição T é a dificuldade em calcular, a partir de T, esses transformadores de predicados.

Vendo predicados P, Q com conjuntos de estados genéricos, então um transformador de predicados (t.p.) é uma função F que actua sobre tais conjuntos e dá origem a relações da forma

$$Q \equiv F(P)$$

Se os conjuntos P, Q forem descritos por fórmulas lógicas, também representadas por P e Q,

$$P(x) \equiv x \in P \quad , \quad Q(x) \equiv x \in Q$$

então idealmente o t.p. deve ser descrito por uma função ϕ_F que actua sobre fórmulas lógicas para produzir outras formas lógicas de tal forma que se verifique

$$Q \equiv F(P) \quad \text{se e só se, como fórmulas,} \quad Q \equiv \phi_F(P)$$

Da definição de $\mathrm{sp}(P)$ e de $\mathrm{wp}(Q)$ é simples verificar que

$$\mathrm{sp}_\mathrm{T}(P) \equiv \bigcup_{x \in P} \mathrm{T}(x) \equiv \{ y \mid \exists\, x \in P \cdot \mathrm{T}(x,y) \}$$

Isto é, $\mathrm{sp}_\mathrm{T}(P)$ é o conjunto de todos os estados acessíveis num passo a partir de um $x \in P$. Em princípio esta definição fornece um modo de definir $\mathrm{sp}_\mathrm{T}(P)$ como predicado

$$\mathrm{sp}_\mathrm{T}(P)(y) \equiv \exists\, x \cdot P(x) \wedge \mathrm{T}(x,y)$$

Análogamente se prova que

$$\mathrm{wp}_\mathrm{T}(Q) \equiv \bigcup_{y \in Q} \mathrm{T}^{-1}(y) \equiv \{ x \mid \exists\, y \in Q \cdot \mathrm{T}(x,y) \}$$

Portanto, $\mathrm{wp(Q)}$ é o conjunto de todos os estados a partir dos quais é possível alcançar um elemento de Q em uma transição. Como predicado

$$\mathrm{wp}_\mathrm{T}(Q)(x) \equiv \exists\, y \cdot Q(y) \wedge \mathrm{T}(x,y)$$

No entanto estes predicados $\mathrm{wp}_\mathrm{T}(Q)$ e $\mathrm{sp}_\mathrm{T}(P)$ são fórmulas dependentes das fórmulas Q, P mas que têm a forma de fórmulas quantificadas. Isto não é útil já que, no caso geral, não é possível inferir delas um transformador de predicados.

Para uma relação de transição genérica T não existe um caminho que permite obter esse transformador de fórmulas/predicados. Porém muitos são os casos de t.p.'s onde a transformação de fórmulas pode ser descrita.

Atribuição

A forma mais simples de transição de estados é a que, numa linguagem imperativa, deriva do comando *atribuição*

$$\mathrm{X} \leftarrow exp(\mathrm{X})$$

A relação de transição que representa esta transição de estado é

$$\mathrm{T}(\mathrm{X},\mathrm{X}') \equiv (\mathrm{X}' = exp(\mathrm{X}))$$

Portanto tem-se,

$$P \rightarrow \mathrm{wp}(Q) \equiv P(\mathrm{X}) \wedge \mathrm{T}(\mathrm{X},\mathrm{X}') \rightarrow Q(\mathrm{X}') \equiv$$
$$P(\mathrm{X}) \wedge (\mathrm{X}' = exp(\mathrm{X})) \rightarrow Q\{\mathrm{X}/\mathrm{X}'\} \equiv P(\mathrm{X}) \rightarrow Q\{\mathrm{X}/exp(\mathrm{X})\}$$

No último passo elimina-se a variável X' ficando a relação completamente expressa só com a variável X. Concluimos que

$$\mathrm{wp}(Q) \equiv Q\{\mathrm{X}/exp(\mathrm{X})\}$$

A transformação $P \mapsto \mathrm{sp}(P)$ é mais complexa. Partimos da relação $\mathrm{sp}(P) \rightarrow Q$ se e só se $P \wedge \mathrm{T} \rightarrow Q'$.

1. Introduz-se uma variável "fresca" Z e aplica-se a transformação $\{\mathrm{X}/\mathrm{Z}, \mathrm{X}'/\mathrm{X}\}$ à fórmula
$$P(\mathrm{X}) \wedge (\mathrm{X}' = exp(\mathrm{X})) \rightarrow Q\{\mathrm{X}/\mathrm{X}'\}$$

2. Obtém-se a implicação
$$P\{\mathrm{X}/\mathrm{Z}\} \wedge (\mathrm{X} = exp(\mathrm{Z})) \rightarrow Q(\mathrm{X})$$

   O que significa que se deve ter
$$\mathrm{sp}(P) \equiv P\{\mathrm{X}/\mathrm{Z}\} \wedge (\mathrm{X} = exp(\mathrm{Z}))$$
   sendo Z uma variável "fresca".

Esta condição não impede que se tenha de introduzir novas variáveis nos transformadores de predicados como ocorre nos sistemas de transições.

Havoc

A transição $\mathrm{havoc}(X)$ é uma **atribuição não-determinística**. Essencialmente é equivalente a

$$\exists\, a \cdot X \leftarrow a$$

A transição será $\bigvee a \cdot (X' = a)$. A pré-condição mais fraca é

$$\mathrm{wp}(Q) \equiv \bigwedge a \cdot Q\{X/a\}$$

e a pós-condição mais forte é

$$\mathrm{sp}(P) \equiv \bigvee a \cdot P\{\mathrm{X}/a\} \wedge (X = a)$$

Estas fórmulas são quantificadas: a primeira com o quantificador universal $\bigwedge$ e a segunda com o quantificador existencial $\bigvee$. O uso de fórmulas quantificadas em SMT's não é possível no caso geral. Porém em certas circunstâncias é possível, como sabemos, determinar o seu valor lógico usando os testes de satisfabilidade.

Guarda

Em uma transição da forma

$$\mathrm{T}(X,X') \equiv G(X) \wedge S(X,X')$$

o predicado G é uma "guarda" de uma outra transição S.

Tem-se $P \wedge \mathrm{T} \rightarrow Q' \equiv P \rightarrow (G \wedge S \rightarrow Q')$; portanto

$$\mathrm{wp}_{\mathrm{T}}(Q) \equiv G \rightarrow \mathrm{wp}_S(Q)$$

Da mesma forma

$$\mathrm{sp}_{\mathrm{T}}(P) \equiv \mathrm{sp}_S(G \wedge P)$$

Escolha

É frequente a relação de transição de um FOTS ser composta pela disjunção de várias outras relações de transição; vamos aqui considerar apenas duas mas obviamente esta análise estende-se para qualquer número. Temos por hipótese

$$\mathrm{T}(X, X') \equiv \mathrm{T}_0(X, X') \vee \mathrm{T}_1(X, X')$$

É vulgar designar T como a **escolha** entre T_0 e T_1 e representá-la por $\mathrm{T}_0 \parallel \mathrm{T}_1$.

Cada uma das transições T_i , com $i \in \{0, 1\}$, pode ser vista como descrevendo um FOTS "local" que representa uma componente do sistema global. Cada uma destas componentes é definida por um par de transformadores de predicados $\langle \mathrm{wp}_{\mathrm{T}_i}(\cdot), \mathrm{sp}_{\mathrm{T}_i}(\cdot) \rangle$. Então em seguida prova-se que

$$\mathrm{wp}_{\mathrm{T}} \equiv \mathrm{wp}_{\mathrm{T}_0} \wedge \mathrm{wp}_{\mathrm{T}_1} \quad \text{e que} \quad \mathrm{sp}_{\mathrm{T}} \equiv \mathrm{sp}_{\mathrm{T}_0} \vee \mathrm{sp}_{\mathrm{T}_1}$$

Justificação
Para todo P, Q , $P \wedge \neg \mathrm{wp}_i(Q)$ é unsat $\Leftrightarrow$ $\mathrm{sp}_i(P) \wedge \neg Q$ é unsat $\Leftrightarrow$ $P \wedge \mathrm{T}_i \wedge \neg Q'$ é unsat.
Por outro lado $P \wedge \mathrm{T} \wedge \neg Q' \equiv (P \wedge \mathrm{T}_0 \wedge Q') \vee (P \wedge \mathrm{T}_1 \wedge Q')$ e, por isso ,
$P \wedge \mathrm{T} \wedge \neg Q'$ é unsat $\Leftrightarrow$ $P \wedge \mathrm{T}_0 \wedge \neg Q'$ é unsat **&** $P \wedge \mathrm{T}_1 \wedge \neg Q'$ é unsat
Equivalentemente

$$P \leq \mathrm{wp}_{\mathrm{T}}(Q) \Leftrightarrow P \leq \mathrm{wp}_{\mathrm{T}_0}(Q) \;\&\; P \leq \mathrm{wp}_{\mathrm{T}_1}(Q)$$

e portanto $\mathrm{wp}_{\mathrm{T}} = \mathrm{wp}_{\mathrm{T}_0} \wedge \mathrm{wp}_{\mathrm{T}_1}$.
Também

$$\mathrm{sp}_{\mathrm{T}}(P) \leq Q \Leftrightarrow \mathrm{sp}_{\mathrm{T}_0}(P) \leq Q \;\&\; \mathrm{sp}_{\mathrm{T}_1}(P) \leq Q$$

o que implica $\mathrm{sp}_{\mathrm{T}} = \mathrm{sp}_{\mathrm{T}_0} \vee \mathrm{sp}_{\mathrm{T}_1}$.

Composição

Uma segunda construção para combinar duas transições T_0 e T_1 usa a noção de **composição** ou (**sequenciação**) que captura a acção de transitar T_1 imediatamente a seguir à transição T_0.
Pode-se representar por $\mathrm{T} \equiv \mathrm{T}_1 ; \mathrm{T}_0$ e a sua definição é

$$\mathrm{T}(X, X') \equiv \exists\, Y \cdot \mathrm{T}_1(X, Y) \wedge \mathrm{T}_0(Y, X')$$

Os transformadores de predicados da composição vão ser construídos por composição dos operadores

$$\mathrm{sp}_{\mathrm{T}}(P) \equiv \mathrm{sp}_{\mathrm{T}_0}(\mathrm{sp}_{\mathrm{T}_1}(P)) \quad \text{e} \quad \mathrm{wp}_{\mathrm{T}}(Q) \equiv \mathrm{wp}_{\mathrm{T}_1}(\mathrm{wp}_{\mathrm{T}_0}(Q))$$

Justificação
A primeira idntidade resulta de

$$\mathrm{sp}_{\mathrm{T}}(P) \rightarrow Q \Leftrightarrow P \wedge \mathrm{T}_0 \wedge \mathrm{T}_1' \rightarrow Q'' \Leftrightarrow P \wedge T_0 \rightarrow (\mathrm{T}_1 \rightarrow Q'') \Leftrightarrow \mathrm{sp}_{\mathrm{T}_0}(P) \wedge \mathrm{T}_1 \rightarrow Q' \Leftrightarrow \mathrm{sp}_{\mathrm{T}_1}(\mathrm{sp}_{\mathrm{T}_0(P)})$$

A prova da 2ª identidade segue essencialmente os mesmos passos.

Invariantes

Num FOTS com uma relação de transição T o predicado P é invariante quando $P \wedge \mathrm{T} \rightarrow P'$ é tautologia. Em termos de transformadores de predicados esta asserção é equivalente à afirmação de que são tautologias ambas as asserções

$$\mathrm{sp}_{\mathrm{T}}(P) \rightarrow P \qquad \text{e} \qquad P \rightarrow \mathrm{wp}_{\mathrm{T}}(P)$$

Acessibilidade e Segurança

Os dois operadores wp, sp podem ser aplicados múltiplas vezes. Por exemplo

- $\mathrm{sp}^n(I)$ é a pós-condição mais forte após a aplicação n vezes da transição T a partir de um estado inicial $s \in \mathrm{I}$. Formalmente
- $\mathrm{sp}^0(I) \equiv I \quad , \quad \mathrm{sp}^n(I) \equiv \mathrm{sp}(\mathrm{sp}^{n-1}(I))$ para $n > 0$

- $\mathrm{wp}^m(E)$ é a pré-condição mais fraca que assegura um estado de erro $e \in \mathrm{E}$ após m aplicações da transição T. Igualmente

  $\mathrm{wp}^0(E) \equiv E \quad , \quad \mathrm{wp}^m(E) \equiv \mathrm{wp}(\mathrm{wp}^{m-1}(E))$ para $m > 0$

Fazendo a disjunção para todo n ou m definimos

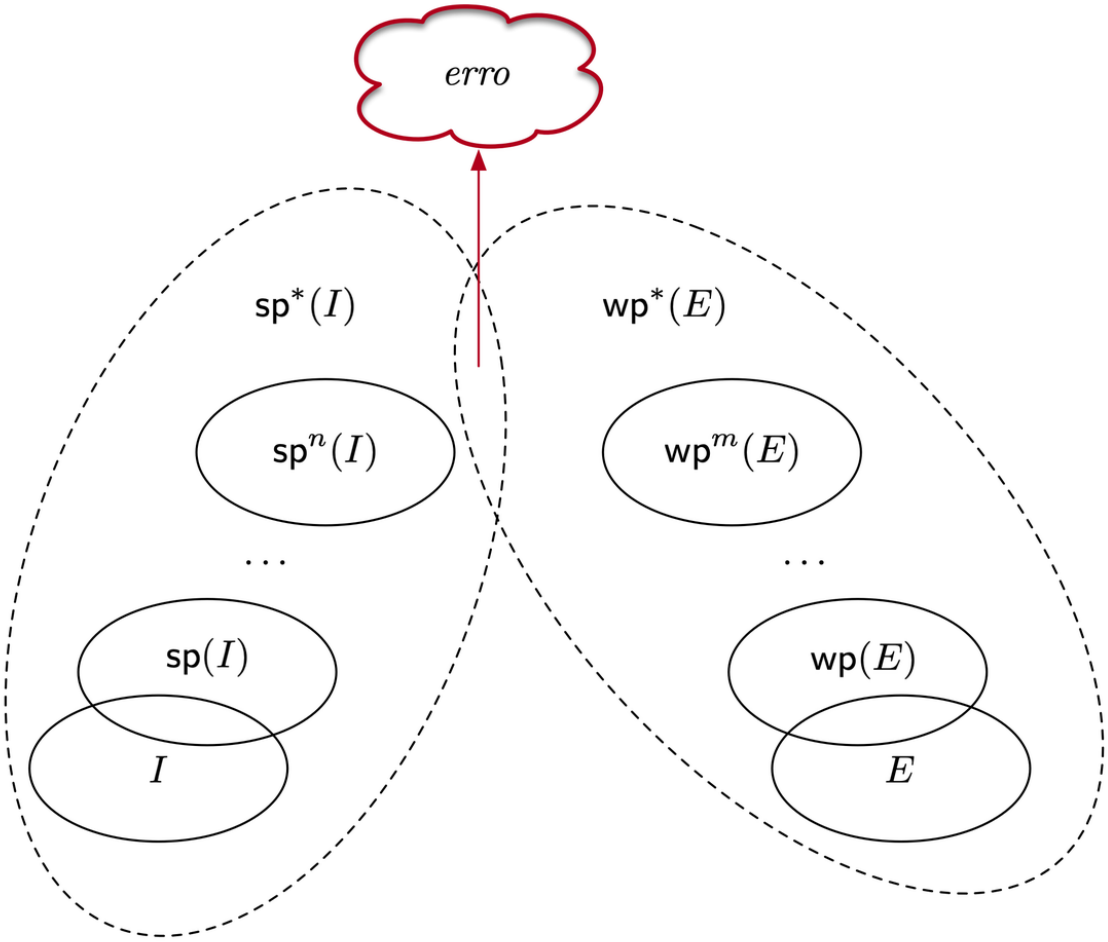
$$\mathrm{sp}^*(I) \equiv \bigvee_{n=0}^{\infty} \mathrm{sp}^n(I) \qquad \text{e} \qquad \mathrm{wp}^*(E) \equiv \bigvee_{m=0}^{\infty} \mathrm{wp}^m(E)$$

Desta forma $\mathrm{sp}^*(I)$ denota o conjunto dos estados que são acessíveis a partir de um estado inicial $s \in I$ com um número apropriado de transições.
Da mesma forma $\mathrm{wp}^*(E)$ representa o conjunto dos estados que asseguram um estado de erro $e \in E$ ao fim de um número apropriado de transições.

> *O sistema Σ é **seguro** se estes dois conjuntos nunca se intersectam.*

Isto é o sistema será seguro se e só se

$$\mathrm{SAT}(\mathrm{sp}^*(I) \wedge \mathrm{wp}^*(E)) = \varnothing$$



Genericamente nem $\mathrm{sp}^*(I)$ nem $\mathrm{wp}^*(E)$ pode ser computado diretamente porque a sua definição envolve um número infinito de disjunções. Por isso um algoritmo de verificação baseado nesta abordagem tem de recorrer a alguma forma de "over-approximation" usando uma forma adequada de indução

Uma forma simples de computar "over-approximations" de $\mathrm{sp}^*(I)$ ou de $\mathrm{wp}^*(E)$ recorre à noção de invariante; nomeadamente tentar encontrar predicados R ou U tais que

$$\mathrm{sp(R)} \rightarrow \mathrm{R} \qquad \text{ou} \qquad \mathrm{U} \rightarrow \mathrm{wp(U)}$$

sejam tautologias.

Control Flow Automata (CFA)

Genericamente CFA's são modelos de sistemas dinâmicos usados, em alternativa aos FOTS, em circunstâncias em que estes últimos definem modelos demasiadamente restritivos ou de difícil aplicação. Sendo um CFA um modelo mais genérico que um FOTS, são frequentes os CFA's que não podem ser descritos fielmente por um sistema de transições.

De qualquer forma, no contexto deste capítulo dedicado ao "SAT based model-checking", há pelo menos algo que é comum a ambas as classes de modelos: ambos têm uma **SMT de base** onde existe um algoritmo de SAT. Por isso, tal como nos FOTS, a especificação de um CFA tem como ponto de partida uma SMT.

Um FOTS é um **modelo global** de comportamento: através de uma relação de transição fica completamente determinado o comportamento global do sistema.
Ao invés um CFA é um **modelo local** do comportamento: o sistema é formado por um conjunto de **locais** (também designados por **modos** de funcionamento) e cada um desses locais tem um comportamento próprio que, em grande medida, é independente do comportamento dos restantes locais.

Nomeadamente, o modelo de comportamento usado num local pode ter uma natureza completamente distinta do modelo usado noutro local. Por exemplo um local pode usar um sistema de transições e, no mesmo sistema, outro local usar transformadores de predicados.

O comportamento de um local só não é completamente independente do comportamento nos restantes locais porque, dentro de um mesmo sistema, os locais precisam de trocar informação. Para tal sistema define um conjunto de **variáveis do sistema** que podem ser lidas e escritas por cada um dos seus locais.

A noção central num CFA, como o nome indica, é a noção de **controlo**. O controlo num CFA é um atributo de local que, em cada instante, privilegia um e só um local do sistema; isto é,

a. em cada instante um local tem (ou não) controlo do sistema
b. em cada instante um e só local tem controlo do sistema.

O **fluxo de controlo** é o conjunto de regras que definem em que circunstâncias um local transfere o controlo para outro local. Cada um destas regras designa-se por **switch** ou **guarda.**

A descrição do fluxo de controlo assume a forma de um grafo orientado tal que:

a. os nodos do grafo identificam-se com o locais do sistema,
b. os ramos do grafo representam a relação "transferência de controlo": um ramo
$$\ell \xrightarrow{s} \ell'$$
representa a transferência de controlo do local ℓ para o local ℓ' que ocorre quando a condição s é válida.

Resumidamente a descrição de um CFA genérico compreende:

a. A SMT T usada na verificação de propriedades temporais, na definição dos "switchs", e na definição de comportamento dos comportamentos locais,
b. O conjunto finito L dos locais do sistema,
c. O conjunto V das variáveis do sistema,
d. O fluxo de controlo descrito pelo grafo referido atrás; nomeadamente faz parte da descrição do fluxo de controlo a definição das "guardas" associadas a cada ramo.
e. O modelo de comportamento M_ℓ associado a cada local $\ell \in \mathrm{L}$.

Como foi dito atrás, os diversos modelos de comportamento M_ℓ dependem do tipo de CFA e mesmo dentro do mesmo CPA, a natureza do modelo depende do local ℓ. Exemplos típicos são sistemas de transições, transformadores de predicados e, para o caso em que o CFA descreve um *autómato híbrido*, sistemas de equações diferenciais.

Sobre este modelo genérico várias outras componentes podem ser adicionadas formando-se assim subclasses dentro da classe de modelos CFA. Nomeadamente,

- A subclasse **Timed Control Flow Automata** (TCFA) acrescenta ao modelo CFA uma representação explícita do **tempo**. Essa representação é feita através de uma variável de sistema t, real ou inteira, que pode ser manipulada como qualquer outra variável do sistema desde que tenha sempre valores positivos e nunca pode ser decrementada.
  A variável t permite medir a duração do intervalo de tempo em que cada local tem controlo e, com essa informação, proceder (ou não) a uma transferência desse controlo.

- A subclasse **Sync Control Flow Automata** (SyncTFA) acrescenta ao modelo CFA a noção de **evento** e a **sincronização** de eventos entre dois ou mais CFA's.
  Essencialmente eventos são marcas que podem ser associadas aos ramos do grafo "fluxo de controlo". Cada ramo do grafo pode ter zero ou mais eventos associados; adicionalmente o mesmo evento pode estar associados a vários ramos.
  Quando dois (ou mais) CFA's que partilham os mesmos eventos estão agrupados, o fluxo de controlo dentro de cada um é condicionado por **regras de sincronismo**; em particular uma determinada transferência de controlo dentro de um dos CFA's, marcada como um evento e, só ocorre se também ocorrer, em cada um dos restantes CFA's do grupo, transferências de controlo marcadas com o mesmo evento e.

Exemplos

A . Multiplicação de inteiros com erros

Nesse exemplo quase todos os locais podem ser identificados com valores específicos de um "program counter" pc num programa imperativo que implemente este algoritmo.

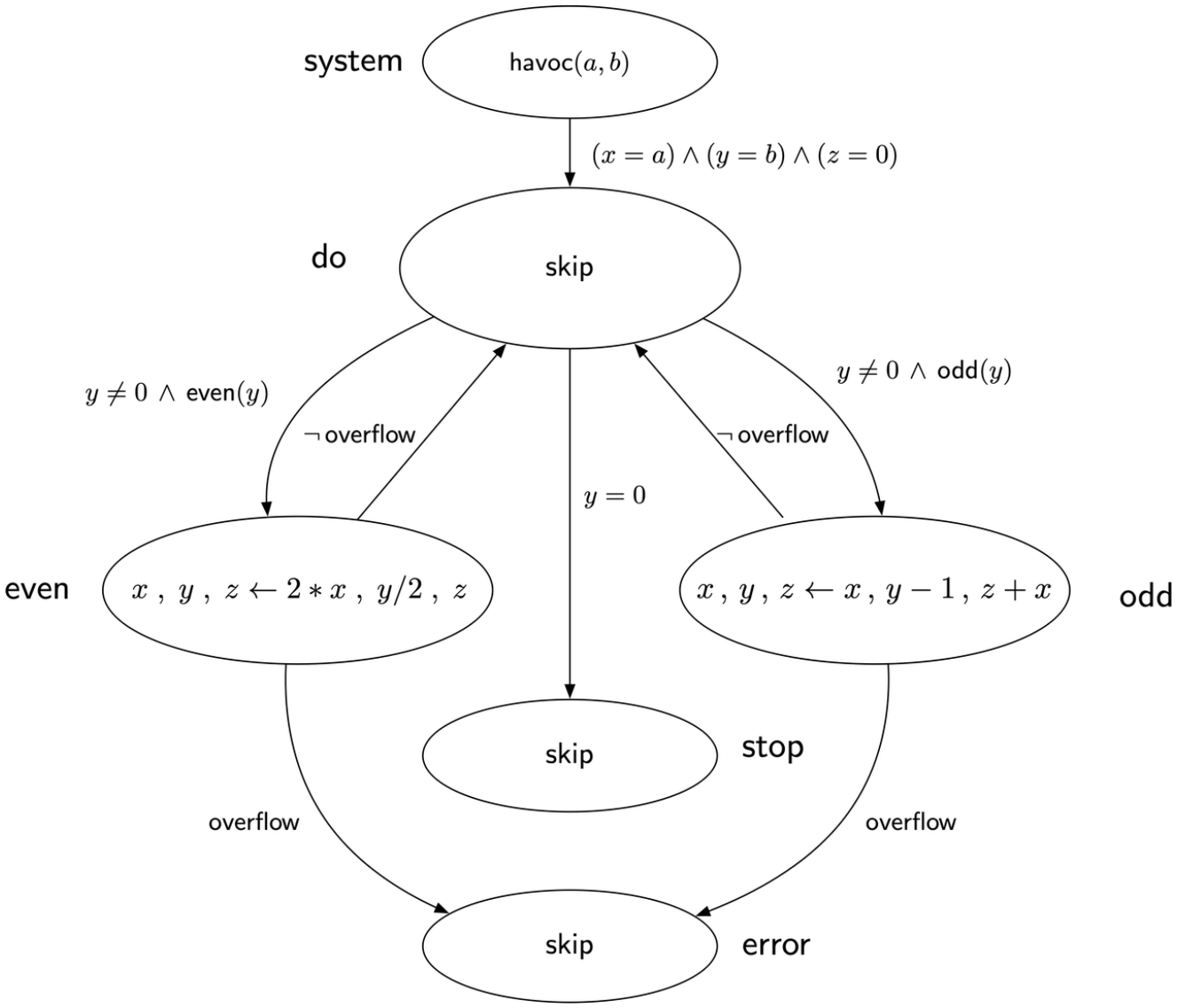
A única excepção é o local error que não pode ser identificado com nenhuma posição do pc porque o erro de "overflow" pode surgir de várias formas. Isso é por si uma indicação de que um simples FOTS não consegue capturar completamente o comportamento desse sistema.

Por isso se optou por uma descrição do comportamento através de predicados e transformadores de predicados.

Nota
Obviamente, para este caso particular onde o "overflow" só ocorre em duas situações distintas, seria eventualmente possível escrever um sistema de transições que descreve-se essas situações.
Não seria simples porque requer inventar um valor fictício do pc que represente o estado de erro.

Para além do facto, já de si negativo, de destruir o significado próprio do pc atribuindo-lhe valores fictícios (fonte provável de "bugs" futuros), este esforço é, em grande medida, inútil porque não se tem a certeza que essas são as duas únicas situações onde aparece o "overflow".

Vamos considerar o exemplo da multiplicação de dois inteiros positivos. O algoritmo foi modelado por um CFA ("control flow automaton") sumariamente descrito no seguinte diagrama.



O diagrama introduz alguma notação adicional em relação à apresentada anteriormente:

1. Todos os locais estão identificados com um nome
2. A inicialização é descrita por uma transição havoc , que atribui às variáveis a, b valores arbitrários, seguida de uma guarda $(x = a) \wedge (y = b) \wedge (z = 0)$ que deixa passar apenas valores apropriados para o nosso problema.

Nestas circunstâncias pode-se assumir que o predicado dos estados iniciais é

$$\mathrm{I} \equiv \mathrm{True}$$

e que o 1º estado de acessibilidade $\mathrm{sp(I)}$ é determinado por este comando havoc seguido pela guarda referida. Por isso

$$\mathrm{sp(I)} = \bigvee a, b \cdot (x = a) \wedge (y = b) \wedge (z = 0)$$

Todos os restantes locais vão ser representados por pré-condições mais fracas. Nomeadamente o local do é representado por $\mathrm{wp(do)}$. O sistema será seguro quando é tautologia

$$\mathrm{sp(I)} \rightarrow \mathrm{wp(do)}$$

ou, equivalentemente, é insatisfazível

$$\bigwedge a, b \cdot (x = a) \wedge (y = b) \wedge (z = 0) \wedge \neg \mathrm{wp(do)}$$

Podemos associar um predicado a cada um dos locais do sistema de acordo com os seguintes princípios.

- Ao local system associamos a condição de segurança atrá referida
- Aos restantes locais com excepção de stop e error associamos a pré-condição mais fraca.
- Aos locais stop e error associamos predicados constantes que vão depender do tipo de problema que vamos tentar modelar.

Ignorando por momentos a semântica pode-se escrever num sistema de equações

---

$$\mathrm{system} = \bigwedge a, b \cdot (x = a) \wedge (y = b) \wedge (z = 0) \rightarrow \mathrm{do}$$
$$\mathrm{do} = (y = 0 \rightarrow \mathrm{stop}) \wedge (y \neq 0 \wedge \mathit{even}(y) \rightarrow \mathrm{even}) \wedge (y \neq 0 \wedge \mathit{odd}(y) \rightarrow \mathrm{odd})$$
$$\mathrm{even} = \mathrm{WP}_{\mathrm{even}}((\mathrm{overflow} \rightarrow \mathrm{error}) \wedge (\neg \mathrm{overflow} \rightarrow \mathrm{do}))$$
$$\mathrm{odd} = \mathrm{WP}_{\mathrm{odd}}((\mathrm{overflow} \rightarrow \mathrm{error}) \wedge (\neg \mathrm{overflow} \rightarrow \mathrm{do}))$$

Os predicados $\mathrm{overflow}, \mathrm{stop}, \mathrm{error}$ não estão especificamente definidos nesta especificação; devem ser concretizados de acordo com as particularidades de cada sistema.

Se for possível encontrar uma solução para este sistema de equações então pode-se resolver vários tipos problemas. Por exemplo

- Para determinar se existe algum estado acessível que seja um estado inseguro faz-se $\mathrm{error} \equiv \mathrm{True}$ e $\mathrm{stop} \equiv \mathrm{False}$ e verifica-se se system é insatisfazível.
- Para determinar se existe algum estado acessível que conduza a uma paragem no estado stop , faz-se $\mathrm{stop} \equiv \mathrm{True}$ e $\mathrm{error} \equiv \mathrm{False}$ e verifica-se se system é insatisfazível.
- Para determinar se p programa está correto, (i.e. se calcula em r o valor $a * b$ e não termina em "overflow") basta introduzir $\mathrm{stop} \equiv (r = a * b)$ e $\mathrm{error} \equiv \mathrm{False}$ e verificar se system é insatisfazível.

O principal problema é encontrar uma solução para o sistema de equações. Uma tal solução pode ser obtida de forma iterativa através das definições recursivas

---

$$\mathrm{system}_n \equiv \bigwedge a, b \cdot (x = a) \wedge (y = b) \wedge (z = 0) \wedge \neg \mathrm{do}_n$$
$$\mathrm{do}_n \equiv (y = 0 \rightarrow \mathrm{stop}) \wedge (y \neq 0 \wedge \mathit{even}(y) \rightarrow \mathrm{even}_n) \wedge (y \neq 0 \wedge \mathit{odd}(y) \rightarrow \mathrm{odd}_n)$$
$$\mathrm{even}_n \equiv \mathrm{WP}_{\mathrm{even}}((\mathrm{overflow} \rightarrow \mathrm{error}) \wedge (\neg \mathrm{overflow} \rightarrow \mathrm{do}_{n-1}))$$
$$\mathrm{odd}_n \equiv \mathrm{WP}_{\mathrm{odd}}((\mathrm{overflow} \rightarrow \mathrm{error}) \wedge (\neg \mathrm{overflow} \rightarrow \mathrm{do}_{n-1}))$$

É importante notar que, aqui, temos um sistema de definições enquanto anteriormente tinhamos um sistema de equações. O sistema é inicializado com

$$\mathrm{do}_0 \equiv \mathrm{False}$$

e iterativamente usa as definições acima para produzir os sucessivos valores do_n ; o algoritmo termina com sucesso se for possível construir um invariante: isto é se existir algum n tal que é tautologia $\mathrm{do}_n \Leftrightarrow \mathrm{do}_{n-1}$.

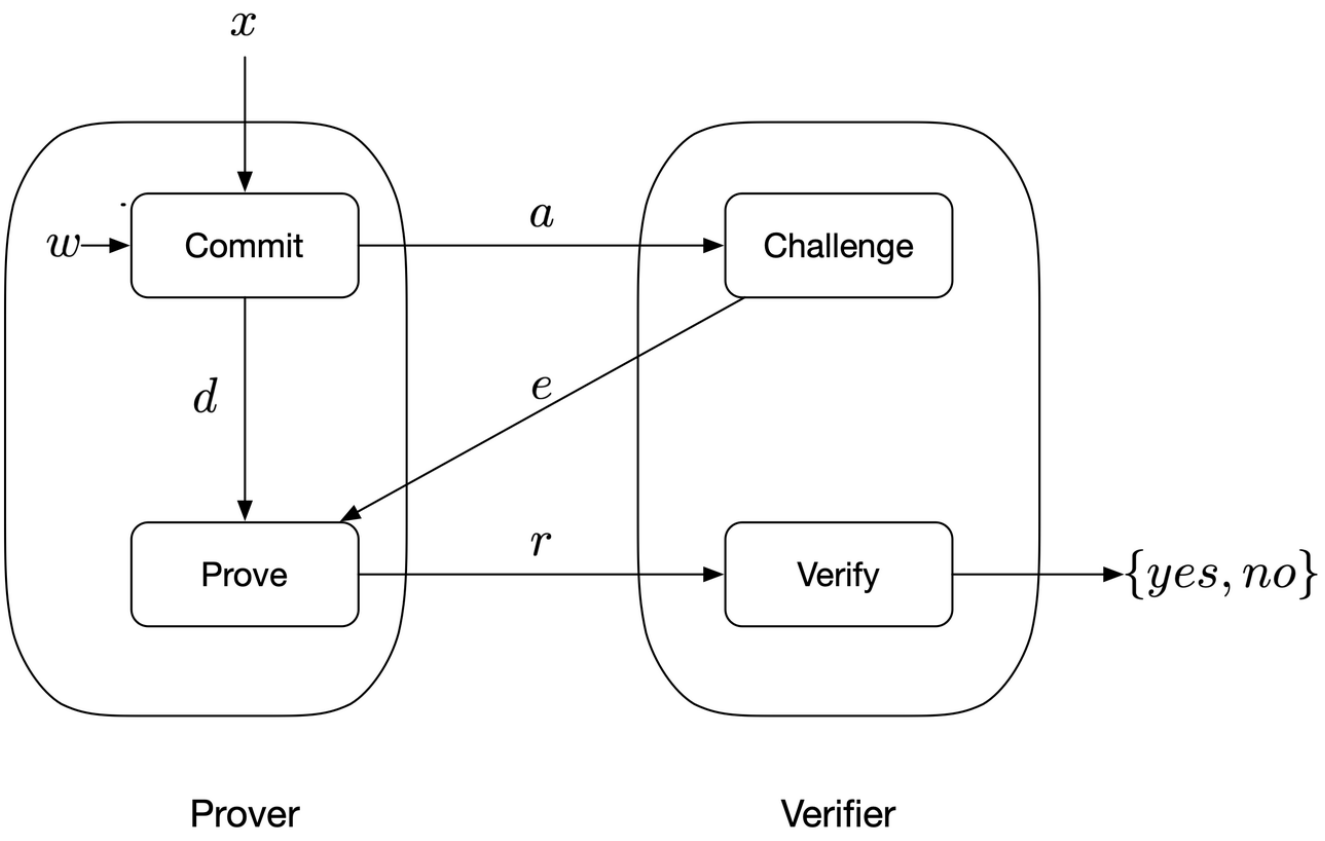
É evidente que se pode sempre assumir uma abordagem análoga à do BMC, colocando um limite ao número de iteração efectuadas. Com isto o algoritmo termina sempre; porém um resultado que seja obtido com um

pequeno valor do limite pode não produzir uma decisão fiável.

B. Σ-Protocol

Um autómato não tem "inputs" ou "outputs" que não sejam expressos pelos valores iniciais ou finais dos estados. Adicionalmente num autómato o controlo está sempre num local e é transferido de local para local através da validade lógica de determinadas guardas. Por isso a descrição de qualquer comunicação entre agentes, através de um CPA, requer alguma análise.

O seguinte diagrama representa um tipo de comunicação conhecida por Σ-protocol e descreve um mecanismo criptográfico de autenticação de agentes conhecido como "prova de conhecimento zero" (ZKP). Muitos dos esquemas criptográficos associados a processos de autenticação de identidade derivam deste tipo de protocolos.



O mecanismo ("protocolo") representa uma troca de mensagens entre dois agentes. É completamente definido pelos quatro algoritmo, representados no diagrama acima, que seguem as indicações seguintes.

1. Um agente, designado por "prover", detêm um segredo e pretende demonstrar que conhece realmente esse segredo sem o publicitar; o segundo agente, designado por "verifier", não conhece o segredo mas,

ainda assim, pretende certificar-se de que o "prover" conhece o segredo. Essa certificação tem a forma de uma mensagem "yes" ou "no" num mecanismo dito de *verificação.*

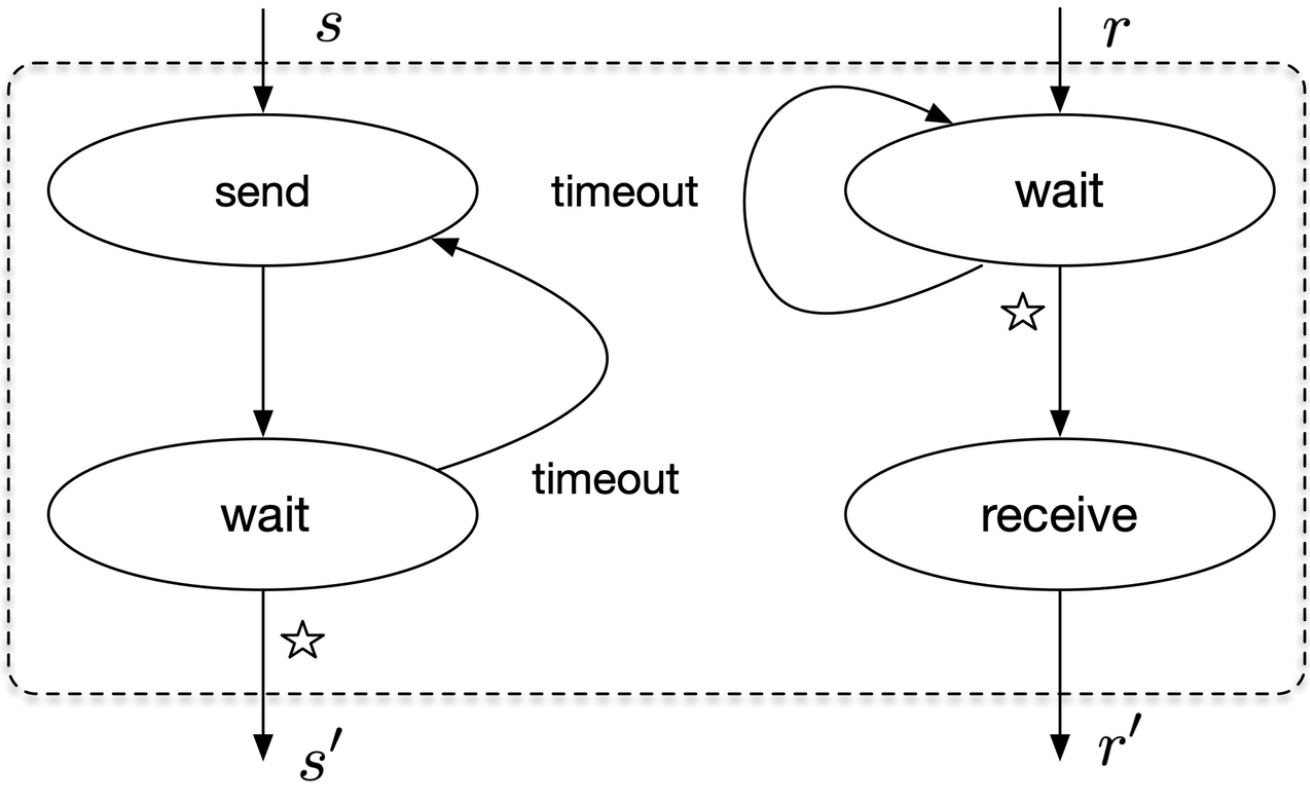
2. O "prover" recebe como "input" uma versão pública x da identidade que pretende provar: a *identidade (ou chave) pública* . Para além disso conhece o segredo w que está únicamente associado a essa identidade; esse segredo w é designado por *identidade (ou chave) privada* .
3. O "prover" começa por gerar probabilisticamente um par
$$(a, d) \leftarrow \text{comm}(x, w)$$
Desse par, a é designado por "commit" e é tornado público ; d é a "descrição" do "commit" e é mantido secreto; no "prover" , d é só é usado na 2ª parte da prova após o que é destruído.
3. O agente "verifier" recebe o "comit" a e com esta informação gera aleatoriamente um desafio e e torna-o público.
$$e \leftarrow \text{chall}(a)$$
4. O "prover" recebe o desafio e e, com a descrição d do segredo consegue construir uma resposta r que prova conhecer o segredo. Calcula e torna pública
$$r \leftarrow \text{prove}(d, e)$$
5. O triplo $\pi = (a, e, r)$, formado só por mensagens públicas, constitui a **prova honesta** . Qualquer triplo alternativo (a', e', r') construído por um outro qualquer mecanismo que não use o segredo w mas recorra eventualmente a provas honestas anteriores , forma uma **prova desonesta** ou **fraude**.
6. O agente "verifier" determina uma resposta
$$\text{verify}(x, \pi)$$
tal que:
   i. a resposta de "verify" é sempre "yes" quando a prova π é honesta.
   ii. se a prova π não for honesta (for uma fraude), a resposta de "verify" tem baixa probabilidade de ser "yes"
   iii. a partir de uma prova π , honesta ou não, e da resposta de "verify" não é possivel extrair qualquer conhecimento sobre a chave privada w que não fosse possível obter sem o conhecimento de π ; em particular não é possível extrair de uma prova honesta π uma outra prova honesta distinta de π .

   É esta última propriedade que dá ao protocolo a qualidade de ser *conhecimento zero*.

A *segurança física* do protocolo é determinada pelo conjunto de dispositivos e procedimentos que mantém secreta a informação privada (a chave privada w e as descrições d) ao longo de todo o tempo de vida do protocolo. A *segurança criptográfica* do protocolo é determinada pelo conjunto de técnicas que garantem (iii) e a qualidade de *conhecimento zero*. A *correção do protocolo* é determinada pelas técnicas formais que garantem a verificação das propriedades (i) e (ii) acima.

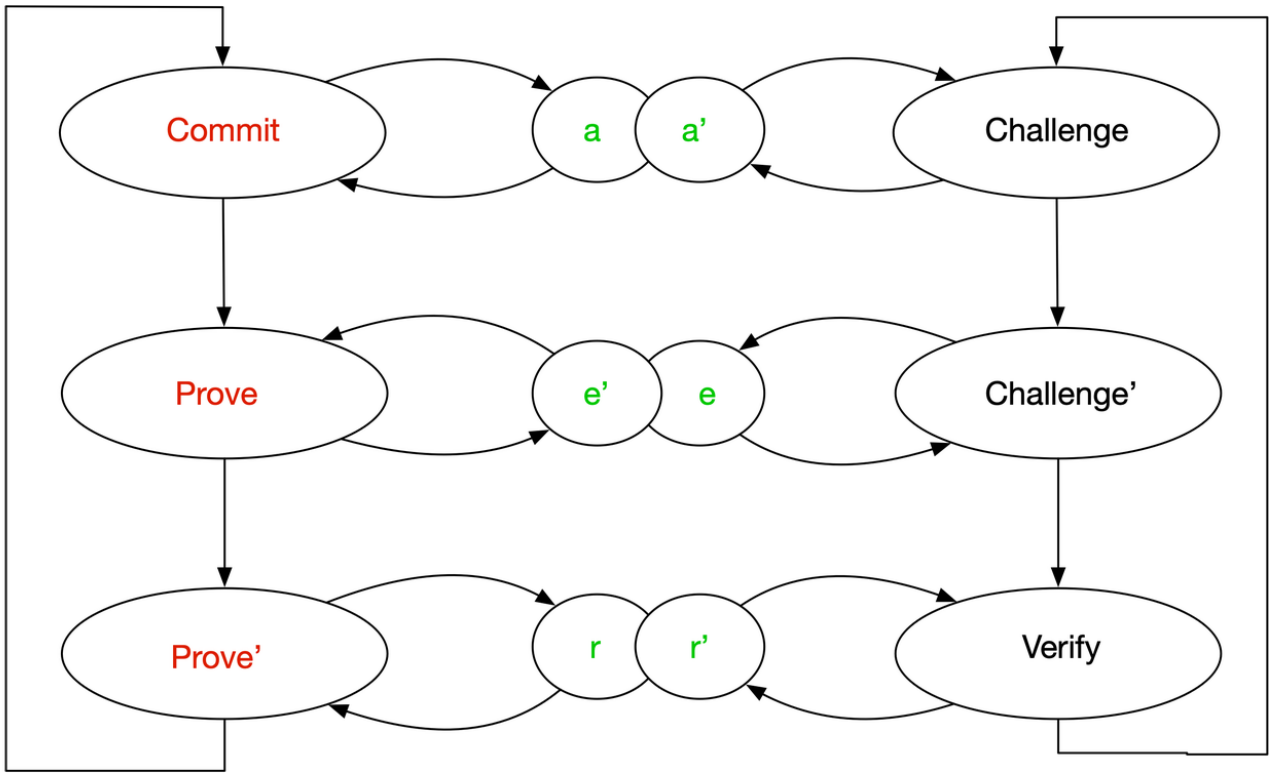
Ignorando as propriedades criptográficas que fazem com que este protocolo seja correto e seguro, ignorando também erros de "overflow" nas computações executadas, vamos tentar representar num grupo de CFA's este protocolo assumindo que alguma das comunicações pode falhar.

Antes de representarmos todo o protocolo vamos começar por representar uma comunicação simples por dois CFA's que partilham um evento.



Neste grupo temos dois CFA's: "send" e "receive". O grupo tem 4 "portas" correspondendo a outras tantas transferências de controlo: duas portas s, s' relativas à execução do "send" e duas portas relativas à execução do "receive".
O CFA do lado esquerdo ("send") envia um item de informação colocando esse item numa variável. Transfere o controlo para um local de espera durante um certo período de tempo. Se um "timeout" ocorre o item volta a ser enviado. A alternativa é sincronizar com um evento (aqui marcado com uma estrela). O CFA do lado direito ("receive") espera até poder sincronizar com o evento; se não sincronizar e o "timeout" ocorrer volta a iniciar o processo espera. Se o sincronismo ocorrer recebe o item de informação.

Este grupo vai ser usado para comunicar cada uma das mensagens a, e, r. O modelo vai conter dois CFA's com dois distintos conjuntos de variáveis descrevendo o comportamentos do "prover" e do "verifier" .



CFA's e Transformadores de Predicados

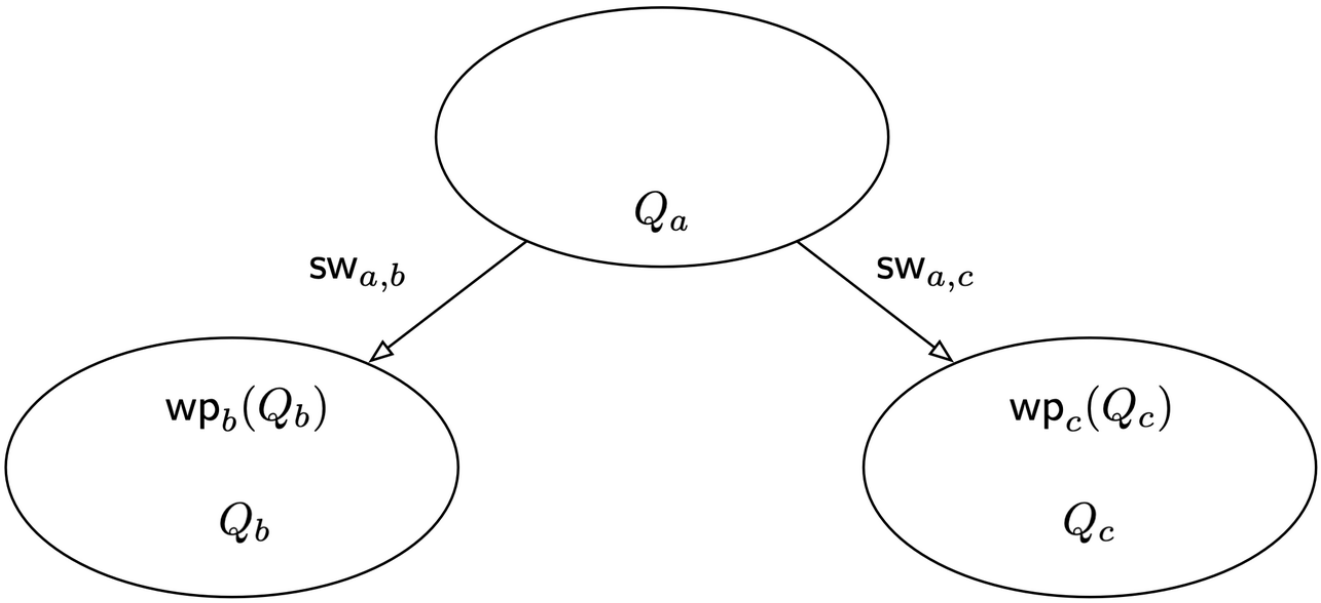
Essencialmente um CFA é descrito por um grafo orientado $\langle \mathrm{V}, \mathrm{E} \rangle$ em que os vértices/nodos $a \in \mathrm{V}$ representam os locais e os ramos $e \in \mathrm{E}$ representam "switchs".
Existe um nodo marcado com system que é o único local que não tem antecessores. Existem nodos que são "sinks" (i.e. não têm sucessores) que são marcados com os identificadores stop ou error. A verificação da correção é estabelecida por um predicado φ designado por *pós-condição* que tem de ser válida nos locais stop.

O comportamento do sistema é gerado a partir de interpretações semânticas dos locais segundo uma de duas abordagens: orientada às pré-condições e orientada às pós-condições.

Representação orientada às pré-condições

1. Cada nodo $a \in \mathrm{V}$ tem associado um transformador de predicados wp_a e cada ramo $(a, b) \in \mathrm{E}$ tem associado um predicado $\mathrm{sw}_{a,b}$.
2. O comportamento global do sistema é determinado por um conjunto de predicados $\{Q_a\}_{a \in \mathrm{V}}$, um para cada local a, que representam o conjunto de estados acessíveis **após** a boa terminação do fluxo de informação associado a a.
3. As relações que determinam o comportamento descrevem a relação entre cada local a e os seus sucessores (locais que são destino de ramos com origem em a). Para todo a são tautologias

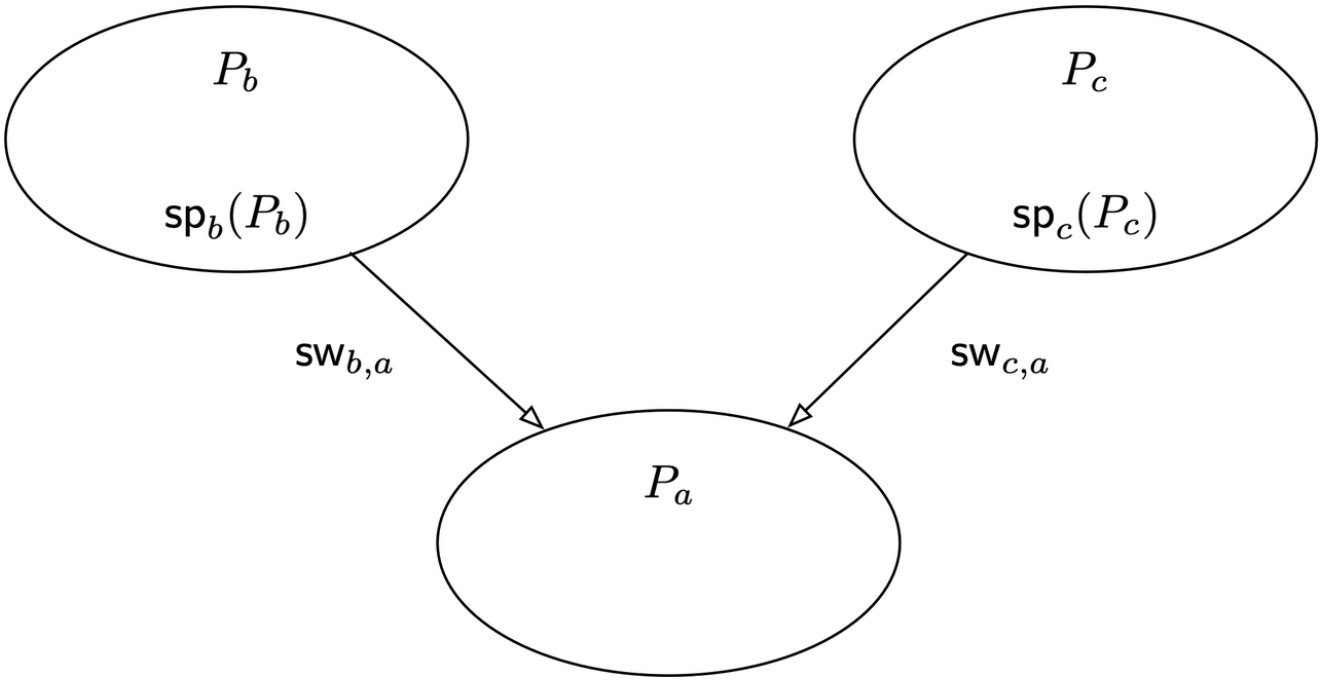
$$Q_a \equiv \bigwedge_{(a,b) \in \mathrm{succ}(a)} \mathrm{sw}_{(a,b)} \rightarrow \mathrm{wp}_b(Q_b)$$



4. O grafo tem uma única fonte (local sem antecessores) marcada como system e condições iniciais definitas por um predicado init. A seguinte relação é tautologia.
$$\mathrm{init} \rightarrow \mathrm{wp}_{\mathrm{system}}(Q_{\mathrm{system}})$$
5. A configuração pode ser usada para verificar a **correção** ou a **segurança** do sistema.
   a. A correção é definida por uma pós-condição φ que tem que ser válida nos locais stop. Adicionalmente nos locais error não deve existir estados acessíveis. Assim axiomática ente estabelecem-se as condições
   $$Q_{\mathrm{stop}} \equiv \varphi \quad , \quad Q_{\mathrm{error}} \equiv \mathrm{False}$$
   b. A insegurança é detectada pela verificação da condição de inicialização quando os locais terminais são
   $$Q_{\mathrm{stop}} \equiv \mathrm{False} \quad , \quad Q_{\mathrm{error}} \equiv \mathrm{True}$$

Representação orientada às pós-condições

1. Cada nodo $a \in \mathrm{V}$ tem associado um transformador de predicados sp_a e cada ramo $(a,b) \in \mathrm{E}$ tem associado um predicado $\mathrm{sw}_{a,b}$.
2. O comportamento global do sistema é determinado por um conjunto de predicados $\{P_a\}_{a \in \mathrm{V}}$, um para cada local a, que representam o conjunto de estados acessíveis imediatamente **antes** da execução do fluxo de informação associado a a.
3. As relações que determinam o comportamento descrevem a relação entre cada local a e os seus antecessores (locais que são origem de ramos que terminam em a). Para todo a são tautologias
$$P_a \equiv \bigvee_{(b,a) \in \mathrm{pred}(a)} \mathrm{sw}_{(b,a)} \wedge \mathrm{sp}_b(P_b)$$



4. O grafo tem uma única fonte (local sem antecessores) marcada como system e condições iniciais definitas por um predicado init. A seguinte relação é tautologia.

$$P_{\mathrm{system}} \equiv \mathrm{init}$$

5. A configuração pode ser usada para verificar a **correção** ou a **segurança** do sistema.
   a. A correção é definida por uma pós-condição φ que tem que ser válida nos locais stop. Adicionalmente nos locais error não deve existir estados acessíveis. Assim axiomática ente estabelecem-se as tautologias

$$P_{\mathrm{stop}} \rightarrow \varphi \quad , \quad P_{\mathrm{error}} \rightarrow \mathrm{False}$$

   b. A insegurança é detectada pela verificação da condição de inicialização quando os locais terminais são

$$P_{\mathrm{stop}} \rightarrow \mathrm{False} \quad , \quad P_{\mathrm{error}} \rightarrow \mathrm{True}$$

O problema computacional está na determinação do vetor de predicados $\mathrm{P} \equiv \{P_a\}_{a \in \mathrm{V}}$ ou $\mathrm{Q} \equiv \{Q_a\}_{a \in \mathrm{V}}$. Para tal vamos considerar as várias equivalência como transformadores de predicados vetoriais $\mathrm{P} \equiv \mathrm{WP(P)}$ (ou $\mathrm{Q} \equiv \mathrm{SP(Q)}$) e vamos tentar provar por construção iterativa que os transformadores têm um ponto fixo.

“Property Directed Reachability (PDR)”

O algoritmo PDR (“property directed reachability”) é uma algoritmo eficiente para verificar sistemas dinâmicos seguros. Foi introduzido inicialmente na verificação de “hardware” , em FSM, mas tem sido generalizado para outras formas de sistemas dinâmicos nomeadamente autómatos híbridos e programas imperativos ambos objeto dos próximos capítulos.

Neste capítulo vamos apresentar o PDR na sua forma original o que requer alguma adaptação da terminologia e notação que introduzimos anteriormente.

Em primeiro lugar o nosso sistema dinâmico é descrita por um SFOTS

$$\Sigma \equiv \langle X, \mathrm{I}, \mathrm{T}, \mathrm{P} \rangle$$

em que as variáveis têm sempre domínios finitos. Sem perda de generalidade vamos considerar que as variáveis $X = \langle x_1, \cdots, x_n \rangle$ são todas booleanas.
Note-se que neste modelo a propriedade P designa os "bons estados". A propriedade de erro é a negação $\neg \mathrm{P}$.

Assim

- o espaço de estados é o domínio $\mathrm{B}_n \equiv \{0,1\}^n$.
- os predicados I, P e quaisquer outros predicados unários, são descritos por fórmulas proposicionais nas variáveis X.
- a relação de transição T é descrita por uma fórmula proposicional nas variáveis X, X'.

Para representar estes estados, predicados e relações vamos recuperar a terminologia das formas normais (ver +Capítulo 1: Breve Revisão da Lógica) fazendo algumas alterações:

- Um *literal* é uma variável x_i ou a sua negação $\neg x_i$

- *Cláusulas conjuntivas* (conjunções de literais) são aqui designadas por **cubos,** enquanto que as cláusulas disjuntivas (disjunções de literais) são designadas simplesmente por **cláusulas**. Note-se que uma cláusula é sempre a negação de um cubo.

- *Conjuntos de cubos interpretados disjuntivamente* (DNF's) são designados por **frames.**

- *Conjuntos de cláusulas interpretados conjuntivamente* (CNF's) são designados como **propriedades**.

Notas sobre "cubos" como extensões de estados

1. Um **cubo determinado** é um conjunto de literais onde ocorrem todas as variáveis. Se falarem variáveis ao conjunto de literais este designa-se por **cubo indeterminado.**
   Cada estado é representado por um cubo determinado. Cubo indeterminados representam conjuntos de estados.

Exemplo

1. Considere-se $n = 8$ e o estado $s \equiv (0,1,1,0,1,1,1,0) \in \mathrm{B}_8$.

Este é o único estado que satisfaz o ("é modelo de") cubo

$$s \equiv \neg x_0 \wedge x_1 \wedge x_2 \wedge \neg x_3 \wedge x_4 \wedge x_5 \wedge x_6 \wedge \neg x_7$$

2. A fórmula $\alpha \equiv x_1 \wedge x_2 \wedge \neg x_3 \wedge x_5$ é um cubo indeterminado que representa o conjunto de 16 estados

$$\alpha \subset \mathrm{B}_8 \equiv \{(x_0,1,1,0,x_4,1,x_6,x_7) \mid x_0,x_4,x_6,x_7 \in \{0,1\}\}$$

2. O cubo $\alpha \equiv x_1 \wedge x_2 \wedge \neg x_3 \wedge x_5$ pode também ser ser representado por um único **vector** de n componentes pertencentes ao conjunto de três símbolos $\{0,1,\top\}$

$$(\top,1,1,0,\top,1,\top,\top)$$

Aqui, na chamada **lógica de três símbolos**, $\top$ é o símbolo escolhido para designar "valor indeterminado".

3. Genericamente cada cubo na lógica de dois símbolos $\{0,1\}$ é representado por um único vetor $c \in \{0,1,\top\}^n$. Nessa notação, um estado é o caso particular de um cubo em que nenhuma componente é o símbolo $\top$.
   Um cubo onde ocorrem $m \leq n$ variáveis descreve um um conjunto de estados com 2^{n-m} elementos.

4. A semântica da lógica de três valores aumenta a da lógica proposicional de dois valores com as seguintes regras pertinentes ao símbolo $\top$:

$$\top \wedge a = \begin{cases} 0 & \text{se } a = 0 \\ \top & \text{se } a \neq 0 \end{cases}, \qquad \top \vee a = \begin{cases} 1 & \text{se } a = 1 \\ \top & \text{se } a \neq 1 \end{cases}, \qquad \neg\top = \top$$

Problema da expansão de um cubo.

> *Um cubo $\widetilde{c}$ que se obtém do cubo c eliminando alguns literais diz-se uma **expansão** de c. Isto ocorre quando $c, \widetilde{c}$ são ambos cubos e $c \rightarrow \widetilde{c}$ é uma tautologia.*

O *problema da expansão* de um cubo c limitado a uma proposição f define-se como

> *Dados uma fórmula proposicional f e um cubo $c \subset f$ determinar a sua maior expansão $\widetilde{c}$ tal que $\widetilde{c} \wedge \neg f$ seja insatisfazível (equivalentemente $\widetilde{c} \subset f$).*

Um algoritmo trivial para resolver aproximadamente este problema consiste em percorrer todos os literais de c até encontrar um que, eliminado de c, produza uma expansão $\widetilde{c}$ tal que $\widetilde{c} \wedge \neg f$ é insatisfazível. Substitui-se c por $\widetilde{c}$ e o processo repete-se iterativamente, calculando novos valores de $\widetilde{c}$ até que $\widetilde{c} \wedge \neg f$ seja satisfazível.

Este não é o melhor algoritmo de expansão. Existem melhores algoritmos e a eficiência do PDR depende fortemente do algoritmo de expansão usado. No entanto detalhes desses algoritmos não são relevantes nesta fase introdutória .

Algoritmo PDR

Neste algoritmo PDR usa-se as seguintes representações

- os predicados I e P são descritos por propriedades (conjunções de cláusulas).

Consequentemente $\neg\mathrm{I}$ e $\neg\mathrm{P}$ são descritas por "frames" (disjunções de cubos).

- A relação de transição é descrita por uma "frame" nas variáveis X, X'.

- Nenhum estado inicial é inseguro; isto é, a proposição $\mathrm{I} \rightarrow \mathrm{P}$ é uma tautologia.

O algoritmo cria e gere uma sequência de conjuntos de cubos

$$F_0 \supseteq F_1 \cdots \supseteq F_k \supseteq \cdots$$

que devem verificar , para todo k , as seguintes condições:

a. Nenhum estado $s \in c$ de qualquer cubo $c \in F_k$ pode ser alcançado, a partir de um estado inicial, em k ou menos passos de transição. Tais estados dizem-se **bloqueados** pela "frame" F_k.

b. Todo o estado inseguro $s \in \neg \mathrm{P}$ pertence a algum cubo da "frame" F_k ; equivalentemente, a proposição $\neg \mathrm{P} \rightarrow F_k$ tem de ser uma tautologia.

c. São tautologias $F_0 \equiv \neg \mathrm{I}$ e $F_{k+1} \rightarrow F_k$.

d. A propriedade $R_k \equiv \neg F_k \equiv \bigwedge_{c \in F_k} \neg c$ é sobre-aproximação do conjunto de estados que podem ser alcançados em k passos a partir de um estado inicial.

algoritmo de bloqueio

A sequência de "frames" deve verificar a propriedade (a). Por isso uma componente essencial do algoritmo PDR é um **bloqueio** que, dado um cubo c e um número de "frame" k , decide se:

> *Para todo o estado acessível em k passos, $s \in R_k$ que não pertence ao cubo c, e para toda a transição $s \rightarrow s'$, o cubo c também não contém s'. Um tal cubo c diz-se **indutivo** para R_k.*

Para verificar se o cubo c é indutivo para propriedade R_k verifica-se se é tautologia

$$R_k \wedge \mathrm{T} \wedge \neg c \rightarrow \neg c'$$

Para isso usa-se o algoritmo de SAT com a "query"

$$\mathrm{SAT}\{\neg F_k \wedge \mathrm{T} \wedge \neg c \wedge c'\}$$

A eventual tautologia acima verifica-se se e só se a resposta é **unsat**.

O significado associado a um cubo C indutivo para R_k é o de estar **bloqueado** na "frame " k. Ou seja, para todo $s \in R_k$ (i.e alcançável em k transições), se o cubo c não contém s então, após a uma transição adicional $s \rightarrow s'$, o cubo c também não contém s' .

Um tal cubo c pode ser adicionado a F_k e a todos os "frames" F_i, com $0 < i < k$ e o processo de bloqueio termina em sucesso.

Se a resposta for **sat** , o cubo não for indutivo, então existe uma testemunha s que valida a fórmula submetida ao SAT. Suponhamos que $\widetilde{s}$ é a expansão do cubo s. Então, recursivamente, vai-se repetir este algoritmo de bloqueio com o cubo $\widetilde{s}$ e no "frame" $k-1$.
Se se atingir o "frame" F_0 com um cubo que não pertence a $\neg \mathrm{I}$ então o processo de bloqueio falha, porque não se pode bloquear um estado inicial.

get_bad_cube

Um "bad cube" no "frame" k é um cubo que contém um estado inseguro e não pertence a qualquer cubo de F_k. Para o detetar temos apenas se submeter ao SAT a "query"

$$\mathrm{SAT}\{\neg F_k \wedge \neg \mathrm{P}\}$$

Se a resposta for **unsat** não existe um "mau cubo". Se a resposta for **sat** recolhe-se uma testemunha s e expande-se esse estado para um cubo que se devolve como resultado.

Com estas componentes podemos construir o algoritmo PDR

PDR(X, I, T, p)

1. Inicializa a lista de frames com a lista singular $[\neg I]$; inicializar $k = 0$
2. **while** True
3. bad = **get_bad_cube**(p, k)
4. se bad não existe então
5. se $F_{k-1} \leftrightarrow F_k$ então o algoritmo termina com a mensagem **safe**
6. em caso contrário aumenta-se a sequência criando um novo "frame" $F_{k+1} = \text{False}$, faz-se $k = k + 1$ e o ciclo continua
7. se bad existe então corre-se o algoritmo **bloqueio**(bad, k)
8. se a resposta de **bloqueio**(bad,*k*) for **sucesso** o ciclo continua
9. em caso contrário o ciclo termina com a mensagem **unsafe**

O seguinte código usando a interface `pysmt` dá uma implementação muito simples deste algoritmo. O exemplo consta da lista de tutoriais na documentação do `pysmt`.

```
# This example shows a more advance use of pySMT.
#
# It provides a simple implementation of Bounded Model Checking
# with , and applies it on a simple transition system.
#
#
from pysmt.shortcuts import Symbol, Not, And, Or, EqualsOrIff, Implies
from pysmt.shortcuts import is_sat, is_unsat, Solver, TRUE
from pysmt.typing import BOOL
def next_var(v):
    """Returns the 'next' of the given variable"""
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())
def at_time(v, t):
    """Builds an SMT variable representing v at time t"""
    return Symbol("%s@%d" % (v.symbol_name(), t), v.symbol_type())

class TransitionSystem(object):
    """Trivial representation of a Transition System."""
    def __init__(self, variables, init, trans):
        self.variables = variables
        self.init = init
```

```
        self.trans = trans
# EOC TransitionSystem
```

```
class PDR(object):
    def __init__(self, system):
        self.system = system
        self.frames = [system.init]
        self.solver = Solver()
        self.prime_map = dict([(v, next_var(v)) for v in self.system.variables])

    def check_property(self, prop):
        """Property Directed Reachability approach without optimizations."""
        print("Checking property %s..." % prop)
        while True:
            cube = self.get_bad_state(prop)
            if cube is not None:
                # Blocking phase of a bad state
                if self.recursive_block(cube):
                    print("--> Bug found at step %d" % (len(self.frames)))
                    break
                else:
                    print("   [PDR] Cube blocked '%s'" % str(cube))
            else:
                # Checking if the last two frames are equivalent i.e., are induct
ive
                if self.inductive():
                    print("--> The system is safe!")
                    break
                else:
                    print("   [PDR] Adding frame %d..." % (len(self.frames)))
                    self.frames.append(TRUE())

    def get_bad_state(self, prop):
        """Extracts a reachable state that intersects the negation
        of the property and the last current frame"""
        return self.solve(And(self.frames[-1], Not(prop)))
```

```
    def solve(self, formula):
        """Provides a satisfiable assignment to the state variables that are cons
istent with the input formula"""
        if self.solver.solve([formula]):
            return And([EqualsOrIff(v, self.solver.get_value(v)) for v in self.sy
stem.variables])
        return None

    def get_bad_state(self, prop):
        """Extracts a reachable state that intersects the negation
        of the property and the last current frame"""
        return self.solve(And(self.frames[-1], Not(prop)))
    def solve(self, formula):
        """Provides a satisfiable assignment to the state variables that are cons
istent with the input formula"""
        if self.solver.solve([formula]):
            return And([EqualsOrIff(v, self.solver.get_value(v)) for v in self.sy
stem.variables])
        return None

    def recursive_block(self, cube):
        """Blocks the cube at each frame, if possible.
        Returns True if the cube cannot be blocked.
        """
        for i in range(len(self.frames)-1, 0, -1):
            cubeprime = cube.substitute(dict([(v, next_var(v)) for v in self.syst
em.variables]))
            cubepre = self.solve(And(self.frames[i-1], self.system.trans, Not(cub
e), cubeprime))
            if cubepre is None:
                for j in range(1, i+1):
                    self.frames[j] = And(self.frames[j], Not(cube))
                return False
            cube = cubepre
        return True

    def inductive(self):
        """Checks if last two frames are equivalent """
        if len(self.frames) > 1 and \
```

```
            self.solve(Not(EqualsOrIff(self.frames[-1], self.frames[-2]))) is Non
e:
                return True
            return False

    def __del__(self):
        self.solver.exit()
```

```
def counter(bit_count):
    """Counter example with n bits and reset signal."""
    # Example Counter System (SMV-like syntax)
    #
    # VAR bits: word[bit_count];
    #     reset: boolean;
    #
    # INIT: bits = 0 & reset = FALSE;
    #
    # TRANS: next(bits) = bits + 1
    # TRANS: next(bits = 0) <-> next(reset)
    from pysmt.typing import BVType
    bits = Symbol("bits", BVType(bit_count))
    nbits = next_var(bits)
    reset = Symbol("r", BOOL)
    nreset = next_var(reset)
    variables = [bits, reset]
    init = bits.Equals(0) & Not(reset)
    trans = nbits.Equals(bits + 1) &\
            (nbits.Equals(0)).Iff(nreset)
    # A true invariant property: (reset -> bits = 0)
    true_prop = reset.Implies(bits.Equals(0))
    # A false invariant property: (bits != 2**bit_count-1)
    false_prop = bits.NotEquals(2**bit_count -1)
    return (TransitionSystem(variables, init, trans), [true_prop, false_prop])

def main():
    example = counter(4)
    pdr = PDR(example[0])
```

```
    for prop in example[1]:
        pdr.check_property(prop)
        print("")
if __name__ == "__main__":
    main()
```