Capítulo 6: Verificação Formal de Software

Na análise das propriedades de programas imperativos, propriedades como **correção** ou **terminação**, a verificação formal é uma alternativa à metodologia dos testes.
É frequente afirmar-se que "*testes detectam a presença de erros mas não são capazes de garantir a sua ausência*".

Em vez de uma política de repetição e execução sistemática de testes, a verificação formal descreve as propriedades dos programas em proposições lógicas cuja validade pode ser confirmada por um solver adequado.

Desta forma a verificação de propriedades em "software" é, de entre as aplicações dos SMT's e dos seus "solvers", uma das áreas mais importantes. Essa importância advém da capacidade que os SMT's têm para descrever, simultaneamente e com rigor, a sequência de controlo de um programa (o seu **fluxo**) e os **dados** que o programa usa.

Nesta disciplina o "software" é exclusivamente formado por **programas imperativos**; isto é, programas assentes na noção de **estado**, de **comando** transformador de estados e de **fluxo** de comandos representando o controlo para realizar essas transformações.

As propriedades desses programas podem envolver o tempo, e nesse caso dizem-se **dinâmicas**, ou não, caso em que se dizem **estáticas**. Um exemplo de uma propriedade dinâmica é a **terminação,** enquanto que a **correção** é a principal propriedade estática analisada neste curso. Este capítulo estuda essencialmente essa propriedade de correção.

Em termos muito simples, a correção estabelece uma relação entre um programa P uma especificação $\langle \phi , \varphi \rangle$.

O programa P é descrito numa linguagem imperativa apropriada. Em seguida veremos um exemplo de uma tal linguagem, designada LPA, que é vocacionada para o estudo da correção.

A especificação $\langle \phi , \varphi \rangle$ é definida por dois predicados. O predicado ϕ designa-se por **pré-condição** ou **antecedente**. O predicado φ designa-se por **pós-condição** ou **consequente**.

O triplo $\langle P , \phi , \varphi \rangle$ é habitualmente representado por

$$\{\phi\}\, P\, \{\varphi\}$$

e designa-se por **triplo de Floyd-Hoare** .

Quando o programa asseguradamente termina, diz-se que o programa P é **correcto** para a especificação $\langle \phi, \varphi \rangle$ quando, assumindo que ϕ é válido no estado imediatamente anterior ao

programa se iniciar (**estado inicial)** , então φ será válido no estado imediatamente a seguir ao programa terminar (**estado final**).
A definição de correção exige que o programa termine; isto porque só faz sentido quando os estados inicial e final estão bem determinados.

Linguagem de programas anotados.

A linguagem dos programas anotados (LPA) é uma abstração de uma linguagem de programação imperativa especialmente vocacionada para o estudo das propriedades de correção.

Numa linguagem de programação imperativa o estado de um programa é definido pelos valores das suas **variáveis.** Genericamente, **expressões** são as entidades sintáticas que descrevem os valores computáveis a partir do estado.

Comandos

As alterações elementares do estado são realizados pelas **atribuições.** As atribuições são definidas por pares (variáveis, expressões) e escritas como

$$vars \leftarrow exps$$

Uma variante da atribuição é o comando

$$\text{havoc } vars$$

interpretado como a atribuição de valores arbitrários às variáveis $vars$.

Anotações como comandos

As **anotações** comportam-se como comandos que não modificam o estado mas, ao invés, interagem com o estado e, como resultado, afetam o fluxo do programa.

Nem todas as linguagens de programação imperativa suportam anotações; as que o fazem designam-se, obviamente, por "linguagens anotadas". Neste curso vamos considerar linguagens com dois tipos de anotações ambas determinadas por um predicado booleano φ .

- $\text{assert}\,\alpha$
  Quando este comando ocorre em qualquer fluxo e no estado onde ocorre, o predicado α é Falso , então o programa termina em erro.
  Se α tiver o valor True , então o comando é equivalente ao fluxo skip.
  "Terminar em erro" implica a terminação de todos os fluxos ativos no programa, não só o fluxo onde se integra esta anotação.
- $\text{assume}\,\alpha$
  Quando este comando ocorre em qualquer fluxo e no estado onde ocorre, o predicado α é Falso , então o programa termina o actual fluxo mas pode continuar num fluxo alternativo.
  Se α tiver o valor True , então o comando é equivalente ao fluxo skip.

Resumidamente temos 4 tipos de comandos

$$\text{Comando} \;::=\; var \leftarrow exp \;|\; \text{havoc}\, var \;|\; \text{assert}\, \alpha \;|\; \text{assume}\, \alpha$$

Fluxos

A noção fundamental para caracterizar programas imperativos é a de **fluxo de controlo** ou, simplesmente, **fluxo**.
Sintaticamente um fluxo é

a. vazio, representado por skip, ou
b. é a **composição** de um comando com um fluxo, através de um operador $;$
c. é a **escolha** não deterministica entre dois fluxos, representada pelo operador $\|$.

Sintaticamente um fluxo é definido por uma das seguintes formas

$$\text{Fluxo} \;::=\; \text{skip} \;|\; \text{Comando} \,;\, \text{Fluxo} \;|\; \text{Fluxo} \;\|\; \text{Fluxo}$$

ou, em alternativa,

$$\text{Fluxo} \;::=\; \text{skip} \;|\; \text{Fluxo} \,;\, \text{Comando} \;|\; \text{Fluxo} \;\|\; \text{Fluxo}$$

A primeira forma sintática é aqui referida como *"orientada às pré-condições"* e a segunda é classificada como *"orientada às pós-condições".*

Um comando, por si só, não é um fluxo. Só se transforma num fluxo quando é composto com outro fluxo existente.

O único fluxo atómico é skip equivalente à composição de uma sequência vazia de comandos. Para transformar um comando C num fluxo temos de usar uma das composições $C \,;\, \text{skip}$ ou $\text{skip} \,;\, C$

A adição de um novo comando a um fluxo existente estabelece diferenças semânticas entre as duas formas sintáticas:

- Na construção $\text{Comando} \,;\, \text{Fluxo}$ pretende-se determinar os objetivos que um comando deve estabelecer (para garantir correção) em função dos objetivos do fluxo que continua o comando. Estes objetivos são designados por **pré-condições**.
- Na construção $\text{Fluxo} \,;\, \text{Comando}$ pretende-se fixar os resultados do comando em função dos resultados do fluxo que o antecede.

A composição de dois fluxos $S \;;\; S'$ define-se, a partir das duas construções anteriores, por um mecanismo de indução.

- $\text{skip} \,;\, S \;\equiv\; S \,;\, \text{skip} \;\equiv\; S$
- $\{S \,;\, C\} \;;\; S' \;\equiv\; S \;;\; \{C \,;\, S'\}$

A construção $\text{Fluxo} \parallel \text{Fluxo}$ introduz a **escolha** não-determinística entre dois fluxos.

- Em termos de pré-condições (ou objetivos), o objetico de $A \parallel B$ é a **conjunção** dos objetivos de A e de B. De facto, como não sabemos qual desse fluxos será escolhido, para garantir que o objetivo de $A \parallel B$, independente da escolha, é necessário assegurar ambos os objetivos de A e B
- Em termos de pós-condições (ou resultados), o resultado de $A \parallel B$ é a **disjunção** dos resultados de A e de B: dado que não se sabe à priori qual o fluxo escolhido, o resultado da escolha será o resultado de um dos fluxos.

Colapso de Anotações

É muito importante reforçar um aspecto das anotações: <u>não modificam o estado.</u> Em particular não modificam o "program conter" se ele existir.
A composição de duas anotações A, A' (que podem ambas ser assume's ou assert's)

$$A \,;\, A'$$

<u>não significa</u> que A "executa" primeiro e depois "executa" A': a noção de "execução" não se aplica a anotações Por isso, na composição de anotações, a ordem entre elas é irrelevante.

O **colapso de anotações** é o mecanismo que permite transformar um par de anotações numa única anotação que lhe seja semanticamente equivalente. Para esse efeito aplicam-se as seguintes regras:

- $\text{assert}\,\phi \;;\; \text{assert}\,\varphi \;\equiv\; \text{assert}\,\phi \wedge \varphi$
- $\text{assume}\,\phi \;;\; \text{assert}\,\varphi \;\equiv\; \text{assert}\,\varphi \;;\; \text{assume}\,\phi \;\equiv\; \text{assert}\,(\phi \rightarrow \varphi)$
- $\text{assume}\,\phi \;;\; \text{assume}\,\varphi \;\equiv\; \text{assume}\,\phi \wedge \varphi$

Outras construções sintáticas

Neste conjunto de construções de comandos e fluxos reconhecemos a falta de algumas construções que habitualmente estão associadas a programas imperativos. Nomeadamente faltam os **comandos condicionais**, cujo exemplo paradigmático tem a forma

$$\text{if } b \text{ then } A \text{ else } B$$

como faltam os comandos iterativos ; por exemplo os ciclos while ou for

$$\text{while } b \text{ do } A \qquad \text{e} \qquad \text{for } x \in D \text{ do } A(x)$$

O comando condicional $\text{if } b \text{ then } A \text{ else } B$ reescreve-se facilmente na linguagem de programas anotados como

$$(\,\text{assume}\, b \;;\; A\,) \;\parallel\; (\,\text{assume}\, \neg b \;;\; B\,)$$

A situação é completamente distinta nos comandos iterativos. Os ciclos têm uma semântica bastante complexa e é necessário recorrer a várias aproximações à sua semântica de forma a se

poder construir uma representação lógica que possa ser verificada. Uma das próximas secções será inteiramente dedicada a este tema.

Correção de programas imperativos

Uma vez obtida a descrição sintática dos programas segue-se a formalização do seu significado. A semântica de um programa imperativo segue uma das seguintes abordagens:

Semântica Operacional
Nesta abordagem um programa é descrito por um sistema de transições. Existe um estado, definido pelos valores das variáveis do programa, e existe uma relação de transição que reflete a forma como os fluxos interagem com esse estado.

A formalização semântica assume a forma de um FOTS e usa-se uma lógica temporal para definir propriedades do programa. Recorre-se depois a técnicas como o BMC ("bounded model checking") ou as provas por indução para verificar a validade dessas propriedades.
O exemplo paradigmático de propriedade verificada nesta abordagem é a **terminação** de ciclos.

Semântica Denotacional
Esta abordagem denota (associa) a cada programa uma fórmula da lógica de 1ª ordem. Os "bons" programas, isto é os programas **corretos**. são os que são denotados por fórmulas sempre válidas; isto é, por tautologias.

Como as tautologias P são sempre testadas verificando se a sua negação $\neg P$ é **unsat** , os eventuais modelos de $\neg P$ indicam as situações onde a validade de P falha. Essas serão as razões porque o programa não estará correcta: aquilo que na terminologia vulgar se designa por "bugs".

A correção é apenas uma das propriedades estáticas que se podem verificar com esta metodologia. Outras propriedades podem igualmente ser usadas para denotar os programas que as verificam, e a sua validade atesta a satisfação dessa propriedade no programa em questão.

Metodologia de Floyd-Hoare (FH)

A abordagem de Floyd-Hoare à correção de programas imperativos é uma metodologia denotacional clássica que está na origem da maioria das metodologias das recentes para verificação da correção.

A metodologia define uma lógica cujas fórmulas se escrevem

$$\{\phi\}\, S\, \{\varphi\}$$

em que

i. S é um programa na linguagem imperativa que está a ser analisada

ii. ϕ e φ são predicados num fragmento decidível da LPO cujas variáveis livres são variáveis do programa S.

O triplo de entidades $\langle \phi , S , \varphi \rangle$ é designado por **triplo de Hoare**.

A semântica desta lógica traduz a semântica da linguagem de programas e vai depender obviamente da linguagem. Em qualquer caso, quando a fórmula é uma tautologia diz-se que

> *o programa S está parcialmente correto para a pré-condição ϕ e pós-condição φ*

A metodologia FH e as suas variantes que estudaremos adiante, só fazem sentido quando se assume à priori que o programa S termina. A prova de terminação deve ser efectuada antes da prova de correção parcial e de forma independente desta. Quando existem ambas as provas, para a terminação e para a correção parcial, então o programa diz-se **correto**.

Interessa-nos aplicar a metodologia de Floyd-Hoare à linguagem de programas anotados que temos vindo a apresentar. Nessa linguagem existem algumas particularidades que é importante referir.

1. Nesta linguagem não existem ciclos e, por isso, a questão da terminação não se coloca: todos os fluxos ou conjuntos finitos de fluxos terminam.
   Portanto aqui as provas de correção parcial garantem também a correção do programa.

2. Nesta linguagem um triplo de Hoare $\langle \phi , S , \varphi \rangle$ pode ser reescrito absorvendo num só programa todas as suas componentes.

   $$\text{assume}\,\phi \ ; \ S \ ; \ \text{assert}\,\varphi$$

   Com esta estratégia a denotação lógica de um programa representa completamente a sua correção sem necessidade de introduzir pré ou pós condições adicionais.

3. O comando $\text{assume}\,\phi \ ; \ S \ ; \ \text{assert}\,\varphi$ é fechado por colapso das anotações; isto é, não contém qualquer composição de duas anotações.

Com esta abordagem vamos introduzir dois mecanismos para definir denotações: a denotação **weakest pre-condition (WPC)** e a denotação **strongest pos-condition (SPC)**.
Para cada uma dessas denotações vamos definir também uma **linguagem anotada base,** que coincide com a nossa linguagem anotada apropriada a essa denotação.

Denotação WPC e sua linguagem anotada base.

A linguagem base é definida pela sintaxe

$$\begin{aligned} \text{Comando} &::= \mathit{var} \leftarrow \mathit{exp} \mid \text{havoc}\,\mathit{var} \mid \text{assert}\,\varphi \mid \text{assume}\,\phi \\ \text{Fluxo} &::= \text{skip} \mid \text{Comando} \,;\, \text{Fluxo} \mid \text{Fluxo} \parallel \text{Fluxo} \end{aligned}$$

A denotação associa a cada fluxo S um predicado $[S]$, num fragmento decidível da LPO. Esta denotação está orientada às pré-condições e estas denotam **objetivos a cumprir.**

Adicionalmente a denotação só se aplica a fluxos *invariantes sobre o colapso de anotações*. Isto é, o fluxo não contém composições de duas anotações.

Isso justifica as seguintes regras:

1. As variáveis do programa S são convertidas em variáveis livres da denotação $[S]$.

2. A denotação do fluxo skip é o predicado True. Adicionalmente

   a. A anotação do comando isolado $C \neq \mathrm{skip}$ é a anotação do fluxo $C\,;\,\mathrm{skip}$.
   b. Qualquer fluxo S cuja denotação seja True é denotacionalmente equivalente ao fluxo skip

3. A denotação da escolha não-determinística $S_0 \parallel S_1$ é a conjunção das denotações dos S_i
   $$[S_0 \parallel S_1] \equiv [S_0] \wedge [S_1]$$

4. A denotação de um fluxo $C\,;\,S$ verifica as regras seguintes

   - $[var \leftarrow exp\,;\,S] \quad \equiv \quad [S][var/exp]$
   - $[\mathrm{havoc}\,var\,;\,S] \quad \equiv \quad \bigwedge_a [S][var/a]$
   - $[\mathrm{assert}\,\varphi\,;\,S] \quad \equiv \quad \varphi \wedge [S]$
   - $[\mathrm{assume}\,\phi\,;\,S] \quad \equiv \quad \phi \rightarrow [S]$

As duas primeiras regras referem-se à atribuição e exigem alguma explicação.

Em primeiro lugar a denotação do fluxo $(var \leftarrow exp\,;\,S)$ obtém-se substituindo, na denotação de S, a variável var pela expressão exp. Esta é principal regra para alterações no estado: uma atribuição no programa é denotada por uma substituição na denotação do fluxo seguinte.

A regra do havoc é um tanto especial. Vimos que esse comando é uma espécie de atribuição para um valor arbitrário. Por isso o que se faz é uma quantificação: consideram-se todos os valores possíveis a que formam o domínio da variável var; substituindo a variável por qualquer um desses valores, a denotação de S deve ser válida.

Exemplo

Considere-se o problema de verificar a correção da seguinte fórmula FH

$$\{x > 0\}\ x \leftarrow x - 1\ \{x \geq 0\}$$

assumindo que x é uma variável inteira.

Na linguagem dos programas anotados este triplo pode ser codificado no fluxo

$$S \equiv \text{assume}(x > 0)\,;\ x \leftarrow x - 1\,;\ \text{assert}(x \geq 0)\,;\ \text{skip}$$

A aplicação das regras WPC da esquerda para a direita, produz

$$[S] = (x > 0) \rightarrow [x \leftarrow x - 1\,;\ \text{assert}(x \geq 0)\,;\ \text{skip}] =$$
$$= (x > 0) \rightarrow [\text{assert}(x \geq 0)\,;\ \text{skip}][x/x - 1] = (x > 0) \rightarrow ((x \geq 0) \wedge \text{True})[x/x - 1] =$$
$$= (x > 0) \rightarrow (x - 1 \geq 0)$$

Esta fórmula é uma tautologia na LIA ou equivalentemente a sua negação é *unsat*.

$(x > 0) \wedge (x < 1)$ é *unsat*

Denotação SPC ("strongest post-condition")

Na denotação SPC a linguagem base é orientada às pós-condições e é definida pela sintaxe

$$\text{Comando} ::= var \leftarrow exp \mid \text{havoc}\, var \mid \text{assert}\, \varphi \mid \text{assume}\, \phi$$
$$\text{Fluxo} ::= \text{skip} \mid \text{Fluxo}\ ;\ \text{Comando} \mid \text{Fluxo} \parallel \text{Fluxo}$$

Esta forma de denotação está orientada para as pós-condições e estas denotam **resultados a alcançar**. Por isso as regras são distintas das da WPC.

Tal como no caso da WPC as regras só se aplicam a fluxos que são invariantes em relação ao colapso de anotações.

1. As variáveis do programa S são convertidas em variáveis livres da denotação $[S]$.

2. A denotação do fluxo skip é o predicado True. Adicionalmente

   a. A anotação do comando isolado $C \neq \text{skip}$ é a anotação do fluxo $\text{skip}\,;\ C$.
   b. Qualquer fluxo S cuja denotação seja True é denotacionalmente equivalente ao fluxo skip.

3. A denotação da escolha não-determinística $S_0 \parallel S_1$ é a disjunção das denotações dos S_i

   $$[S_0 \parallel S_1] \equiv [S_0] \vee [S_1]$$

4. A denotação de um fluxo $S\,;\ C$ verifica as regras

   - $[S\,;\ v \leftarrow e] \equiv \exists\ a \cdot (v = e[v/a]) \wedge [S][v/a]$

> No predicado $[S; v \leftarrow e]$ as variáveis têm valores depois a atribuição. Por isso, o valor de a deve ser interpretado como o valor da variável v antes da atribuição e depois da execução de S.

- $[S\,;\,\text{havoc}\,v] \quad \equiv \quad \exists\, a \cdot [S][v/a]$

- $[S\,;\,\text{assert}\,\varphi] \quad \equiv \quad [S] \rightarrow \varphi$

- $[S\,;\,\text{assume}\,\phi] \quad \equiv \quad \phi \wedge [S]$

Composição e Escolha

Um ponto que falta discutir é a existência de eventuais propriedades de distributividade entre os operadores composição `;` e escolha `||`. Pretende-se averiguar se existem relações entre

$$C\,;\,\{S \parallel S'\} \quad \text{e} \quad \{C\,;\,S\} \parallel \{C\,;\,S'\}$$

e entre

$$\{S \parallel S'\}\,;\,C \quad \text{e} \quad \{S\,;\,C\} \parallel \{S'\,;\,C\}$$

sendo C um comando arbitrário e sendo S e S' fluxos arbitrários.

Note-se que no primeiro par os fluxos são construídos na linguagem orientada às pré-condições enquanto que no segundo par os fluxos são construídos na linguagem orientada às pós-condições.

1º Resultado: Se C *é um comando arbitrário e* S e S' *são fluxos arbitrários, então tem-se*

$$C\,;\,\{S \parallel S'\} \equiv \{C\,;\,S\} \parallel \{C\,;\,S'\}$$

no sentido em que são ambos denotados pelo mesmo predicado

Justificação
Para provar este esultado vamos calcular a denotação WPC de ambos os programas fazendo com que C seja cada uma das 4 formas de comandos: atribuição, havoc, assume e assert.

1. $[v \leftarrow e;\,\{S \parallel S'\}] \equiv [S \parallel S'][v/e] \equiv ([S] \wedge [S'])[v/e] \equiv$
   $[S][v/e] \wedge [S'][v/e] \equiv [\{v \leftarrow e;\,S\} \parallel \{v \leftarrow e;\,S'\}]$
2. Para os 3 restantes comandos procede-se do mesmo modo

2º Resultado: *Sejam* S e S' *fluxos arbitrários e* C *um comando*

a. $\{S \parallel S'\}\,;\,C \quad \Rightarrow \quad \{S\,;\,C\} \parallel \{S'\,;\,C\}$ *quando* C *é da forma* $\text{assert}\,\varphi$.
b. $\{S \parallel S'\}\,;\,C \quad \equiv \quad \{S\,;\,C\} \parallel \{S'\,;\,C\}$ *quando* C *não é da forma* $\text{assert}\,\varphi$.

Justificação
Vamos usar denotações SPC e percorrer também as 4 formas de comandos começando pelo $\text{assert}\,\varphi$.

1. $[\,\{S \parallel S'\}\,;\,\text{assert}\,\varphi\,] \equiv ([S] \vee [S']) \rightarrow \varphi \equiv ([S] \rightarrow \varphi) \wedge ([S'] \rightarrow \varphi) \equiv$
   $[\,S\,;\,\text{assert}\,\varphi\,] \wedge [\,S'\,;\,\text{assert}\,\varphi\,]$
   Por outro lado $[\,\{S\,;\,\text{assert}\,\varphi\} \parallel \{S'\,;\,\text{assert}\,\varphi\}\,] \equiv [\,S\,;\,\text{assert}\,\varphi\,] \vee [\,S'\,;\,\text{assert}\,\varphi\,]$

   Portanto verifica-se a implicação $[\,\{S \parallel S'\}\,;\,\text{assert}\,\varphi\,] \Rightarrow [\,\{S\,;\,\text{assert}\,\varphi\} \parallel \{S'\,;\,\text{assert}\,\varphi\}]$ mas a implicação em sentido inverso não se verifica no caso geral.

2. Para as formas restantes de comandos pode-se repetir este processo com a regras SPC respetivas. Em cada uma delas as denotações coincidem e por isso os programas são equivalentes.

Estes resultados podem-se estender a programas $S\,;\,\{S_0 \parallel S_1\}$ ou $\{S_0 \parallel S_1\}\,;\,S$ em que $S, S_0 \text{ e } S_1$ são fluxos. Usando os dois resultados anteriores é simples provar por indução na sequência S.

3º Resultado: *Seja S um fluxo formado por uma sequência de comandos ligados por composição (i.e. em S não intervém o operador escolha). Sejam S_0, S_1 quaisquer fluxos. Então*

a. $S\,;\,\{S_0 \parallel S_1\}$ *e* $\{S\,;\,S_0\} \parallel \{S\,;\,S_1\}$ *são equivalentes.*
b. $\{S_0 \parallel S_1\}\,;\,S$ *e* $\{S_0\,;\,S\} \parallel \{S_1\,;\,S\}$ *são equivalentes quando S não contém anotações* $\text{assert}\,\varphi$.

Se S contiver anotações assert*, então* $[\,\{S_0 \parallel S_1\}\,;\,S\,] \Rightarrow [\,\{S_0\,;\,S\} \parallel \{S_1\,;\,S\}\,]$.

Transformadores de Predicados

Em alternativa à transformação de triplos $\{\phi\}\,P\,\{\varphi\}$ num fluxo $\text{assume}\,\phi\,;\,P\,;\,\text{assert}\,\varphi$ pode-se usar os transformadores de predicados

$$\varphi \mapsto \text{wp}_P(\varphi) \quad \text{e} \quad \phi \mapsto \text{sp}_P(\phi)$$

que definimos no Capítulo 4.

Neste caso a correção assume uma das formas

$$\phi \rightarrow \text{wp}_P(\varphi) \quad \text{é tautologia}$$

ou

$$\text{sp}_P(\phi) \rightarrow \varphi \quad \text{é tautologia}$$

Os operadores wp e sp constroem-se a partir das duas formas de denotação que vimos atrás

Para a pré-condição mais fraca

- Escolha
  $$\text{wp}_{S \parallel R}(\varphi) \equiv \text{wp}_S(\varphi) \wedge \text{wp}_R(\varphi)$$
- Skip

$$\mathrm{wp}_{\mathrm{skip}}(\varphi) \quad \equiv \quad \varphi$$

- Composição

$$\mathrm{wp}_{C;S} : \varphi] \quad \equiv \quad \mathrm{wp}[C](\mathrm{wp}_S(\varphi))$$

onde

- $\mathrm{wp}[var \leftarrow exp] \quad \equiv \quad \varphi \mapsto \varphi[var/exp]$
- $\mathrm{wp}[\mathrm{havoc}\, var)] \quad \equiv \quad \varphi \mapsto \bigwedge_a \varphi[var/a]$
- $\mathrm{wp}[\mathrm{assert}\, \alpha] \quad \equiv \quad \varphi \mapsto (\alpha \wedge \varphi)$
- $\mathrm{wp}[\mathrm{assume}\, \alpha] \quad \equiv \quad \varphi \mapsto (\alpha \rightarrow \varphi)$

Para a pós-condição mais forte

- Escolha

$$\mathrm{sp}_{S \parallel R}(\varphi) \quad \equiv \quad \mathrm{sp}_S(\varphi) \vee \mathrm{sp}_R(\varphi)$$

- Skip

$$\mathrm{sp}_{\mathrm{skip}}(\varphi) \quad \equiv \quad \varphi$$

- Composição

$$\mathrm{sp}_{S;C}(\varphi) \quad \equiv \quad \mathrm{sp}[C](\mathrm{sp}_S(\varphi))$$

onde

- $\mathrm{sp}[var \leftarrow exp] \quad \equiv \quad \varphi \mapsto \exists\, a \cdot (v = e[v/a]) \wedge \varphi[v/a]$
- $\mathrm{sp}[\mathrm{havoc}\, var] \quad \equiv \quad \varphi \mapsto \exists\, a \cdot \varphi[var/a]$
- $\mathrm{sp}[\mathrm{assert}\, \alpha] \quad \equiv \quad \varphi \mapsto (\varphi \rightarrow \alpha)$
- $\mathrm{sp}[\mathrm{assume}\, \alpha] \quad \equiv \quad \varphi \mapsto (\alpha \wedge \varphi)$

Ciclos

Os ciclos não fazem parte na linguagem base dos programas anotados; no entanto é possível garantir, em programas anotados, a correção parcial de um qualquer triplo FW da forma

$$\{\phi\} \quad W \quad \{\varphi\}$$

em que

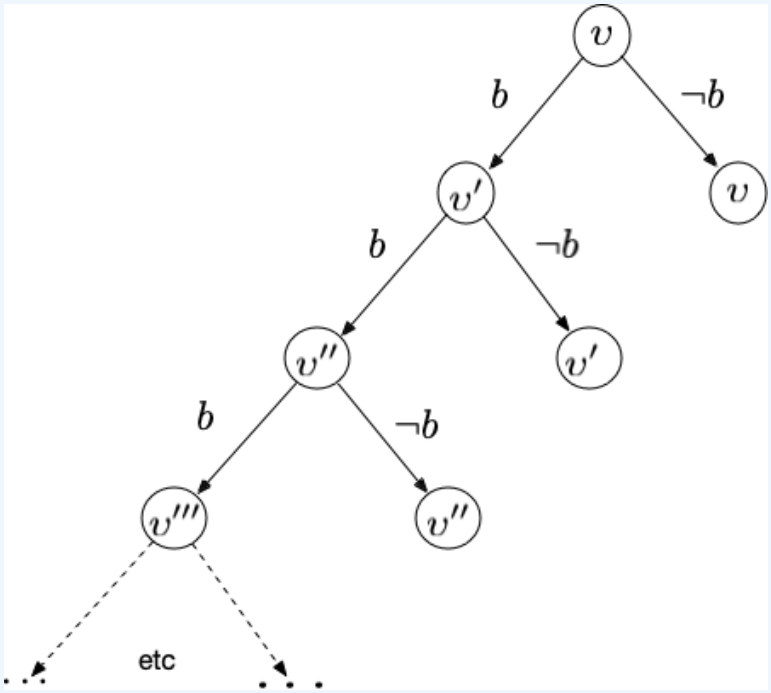
$$W \equiv \mathrm{while}\, b\, \mathrm{do}\, S$$

e em que b é uma expressão boolena e S é um fluxo na linguagem de programas anotados.

Este ciclo não pode ser construído na linguagem base dos programas anotados; no entanto pode ser definido recursivamente nessa linguagem: basta escrever

$$W \quad \equiv \quad \{\,\text{assume}\, b\ ;\ S\ ;\ W\,\} \quad \| \quad \{\,\text{assume}\, \neg b\ ;\ \text{skip}\,\}$$

Justificação
O programa W , como sistema de transições de estado, pode ser representado pela árvore



1. Se, por hipótese, se aceita que o programa termina, então isto é equivalente a afirmar-se que a árvore é finita.
2. As transições marcadas com b implementam a mudança de estado provocada pela execução do corpo do ciclo S . Na linguagem de programas anotados isso é equivalente a um fluxo
$$\{\,\text{assume}\, b\ ;\ S\,\}$$
3. As transições marcadas com $\neg b$ não modificam os valores das variáveis v . Isso é descrito por
$$\{\,\text{assume}\, \neg b\,\}$$
4. Após a transição $\text{assume}\neg b$ o fluxo termina e o estado das variáveis v é o estado no final do ciclo.
Porém, após a transição $\{\,\text{assume}\, b\ ;\ S\,\}$, o ciclo não termina; o valor das variáveis após esta transição pode ser ou não o valor no final do ciclo. Se não terminar volta repetir-se W .

Neste curso vai-se usar três abordagens à verificação da correção parcial do triplo $\{\phi\}\ \ W\ \ \{\varphi\}$.

Em primeiro lugar vamos usar a atribuição universal $\mathbf{havoc}\, v$ para aproximar o ciclo por um um programa não cíclico cuja correção força a correção parcial do ciclo. Essa garantia resulta de no comando não-cíclico se considerar todos os valores possíveis das variáveis do ciclo e provar que, para cada um desses valores, o programa está correto.

Em segundo lugar vamos usar **invariantes** ϑ e provar por indução a correção de W a partir da sua definição recursiva. Invariantes são predicados que mantêm o seu valor lógico com a execução do corpo do ciclo S: na semântica operacional um invariante representa um conjunto de estados para a qual S, como transição de estados, é um "loop".

Em último lugar vamos usar técnicas de **unfold** de ciclos. Isto é vamos aproximar W por um conjunto de fluxos lineares (sem usar escolhas $\|$) finitos. Cada um desses fluxos é analisado como um programa anotado e sem ciclos.

Solução iterativa de definição recursiva

Seja

$$H \equiv \text{assume}\, \phi\, ;\, W\, ;\, \text{assert}\, \varphi$$

com

$$W \equiv \text{while}\, b\, \text{do}\, S \quad \equiv \quad \{\, \text{assume}\, b\, ;\, S\, ;\, W\, \} \;\|\; \{\, \text{assume}\, \neg b\, ;\, \text{skip}\, \}$$

A forma mais directa de determinar a denotação do programa H consiste na determinação iterativa de programas H_k que serão corretos se e só se o ciclo W executa exatamente k vezes e com a precondição ϕ e a pós-condição φ.

Representando a ação de S através de um transformador de predicados wp_S associado a esse programa , pode-se construir a sequência das denotações $[H_k]$ da seguinte forma

$$\begin{array}{lcll} [H_0] & \equiv & \phi \rightarrow \neg b \;\wedge\; \varphi & \\ [H_k] & \equiv & \phi \rightarrow b \;\wedge\; \text{wp}_S([H_{k-1}]) & \text{para } k > 0 \end{array}$$

Formalmente, como por hipótese o ciclo termina, a denotação de H será

$$[H] \quad \equiv \quad \bigvee_{k=0}^{\infty} [H_k]$$

Isto sugere um algoritmo iterativo em que se calcula um acumulador $A_k \equiv \bigvee_{\ell=0}^{k} [H_k]$ e procura-se um limite superior para este acumulador; isto é, procura-se

$$[H] \equiv \lim_{k\rightarrow\infty} A_k$$

1. Se $k = 0$ calcula-se $[H_0]$ como está indicado acima e inicializa-se acumulador $A_0 \equiv [H_0]$
2. Para $k > 0$ calcula-se $[H_k]$ e testa-se se $A_{k-1} \rightarrow [H_k]$ é tautologia. Se for tautologia, então o ciclo termina com o limite calculado como $[H] \equiv A_{k-1}$.

3. Se $A_{k-1} \rightarrow [H_k]$ não for tautologia, então faz-se $A_k \equiv A_{k-1} \vee [H_k]$, incrementa-se k uma unidade e volta-se a executar o ponto 2.
4. Determinado o limite $[H]$ verifica-se se é tautologia; isto porque o programa H é correto se e só se $[H]$ é tautologia. Para isso verifica-se se a sua negação $\neg[H]$ é unsat.

Uso do comando havoc

> *Seja* v *o conjunto de todas as variáveis que ocorrem no copo do ciclo.*
> *Seja* W^* *o seguinte programa anotado.*
> $$W^* \equiv \text{havoc}\, v \,;\, \{\, \{\, \text{assume}\, b \,;\, S \,;\, \text{assume False}\,\} \;\|\; \{\, \text{assume}\, \neg b\,\}\,\}$$
>
> *Então, para todos os predicados* ϕ e φ *, verifica-se na lógica de Floyd-Hoare*
> $$\{\phi\}\ W^*\ \{\varphi\} \implies \{\phi\}\ W\ \{\varphi\}$$

Assumindo que W termina, o programa W^* aproxima W substituindo uma árvore finita mas com um número desconhecido de níveis por uma outra árvore com um número fixo de níveis.

Para isso impõe duas modificações na definição recursiva:

i. A invocação recursiva de W, no primeiro fluxo, é substituída por um comando que força a validade de qualquer pós-condição.
   O comando assume False verifica essa condição. Para qualquer pós-condição $\text{assert}\,\varphi$ tem-se
   $$[\,\text{assume False}\,;\ \text{assert}\,\varphi\,] \equiv \text{False} \rightarrow \varphi \equiv \text{True}$$
   Por isso o fluxo $\{\,\text{assume False}\,;\ \text{assert}\,\varphi\,\}$ pode ser substituído por skip sempre que ocorra num programa.
ii. O programa que resulta da 1ª substituição
   $$P \equiv \text{assume}\,\phi \,;\, \{\, \{\, \text{assume}\, b \,;\, S\,\} \;\|\; \{\, \text{assume}\, \neg b \,;\, \text{assert}\,\varphi\}\,\}$$
   não é suficiente para assegurar a correção de $\{\phi\}\ W\ \{\varphi\}$. Isto porque:

   Note-se que P só usa um nível da árvore. Logo, mesmo sabendo que P é correto, não é possível conhecer o estado no último nível da árvore (onde φ é calculado), a partir do estado no primeiro nível (onde ϕ é calculado).
   A menos, obviamente, que se percorra toda a árvore. Ou então se assegurarmos que isso ocorre qualquer que seja o estado no 1º nível. Essa é a função do comando $\text{havoc}\, v$ colocado no 1º nível da árvore antes da escolha não-determinística.

   Com esta inclusão temos um programa $P \equiv \{\phi\}\, W^*\, \{\varphi\}$ que será

   $$P \equiv \text{assume}\,\phi \,;\, \text{havoc}\, v \,;\, \{\, \{\, \text{assume}\, b \,;\, S\,\} \;\|\; \{\, \text{assume}\, \neg b \,;\, \text{assert}\,\varphi\}\,\}$$

   cuja correção implica a correção de

$$\{\phi\}\ \ \text{while}\, b\ \text{do}\ S\ \text{od}\ \ \{\varphi\}$$

Em resumo

> *A correção do triplo* $\{\phi\}\ W^*\ \{\varphi\}$ *é equivalente à correção do programa anotado*
> $$\mathrm{P}\ \equiv\ \text{assume}\,\phi\ ;\ \text{havoc}\,v\ ;\ \{\ \{\,\text{assume}\,b\ ;\ S\ \}\ \ \|\ \ \{\,\text{assume}\,\neg b\ ;\ \text{assert}\,\varphi\ \}\ \}$$

Usando as regras da denotação WPC, a denotação deste programa calcula-se

$$[\,\mathrm{P}\,]\ \equiv\ \phi\ \rightarrow\ \bigwedge a\ \cdot\ (b[v/a]\ \rightarrow\ [\,S\,][v/a])\ \wedge\ (\neg b[v/a]\ \rightarrow\ \varphi[v/a])$$

Exemplo

Considere-se o triplo FH

$$\{x>0\}\ \ \text{while}\, x\neq 0\ \text{do}\ x\leftarrow x-1\ \text{od}\ \ \{x=0\}$$

Pela regra acima, a sua denotação será

$$(x>0)\ \rightarrow\ \bigwedge a\ \cdot\ (a\neq 0\ \rightarrow\ \text{True})\ \wedge\ (a=0\ \rightarrow\ a=0)$$

que é uma tautologia.

Invariantes

A análise de ciclos usando **havoc** produz apenas uma aproximação à correção de ciclos. Se a aproximação W^* produzir um triplo $\{\phi\}\ W^*\ \{\varphi\}$ correto para, então o triplo $\{\phi\}W\{\varphi\}$ será correto. Porém o inverso não é válido: se aproximação não produzir um triplo correto, não se pode concluir que $\{\phi\}W\{\phi\}$ não está correto.
É necessário, nesse caso, investigar outras técnicas e a procura de **invariantes de ciclo** é frequentemente usada. Vamos assumir sempre que previamente se provou a terminação de W.

Considere-se de novo a definição recursiva de W

$$W\ =\ \{\,\text{assume}\,b\ ;\ S\ ;\ W\ \}\ \ \|\ \ \{\,\text{assume}\,\neg b\ \}\qquad (1)$$

e a definição de um triplo de Hoare através do comando

$$H\ \equiv\ \{\text{assume}\,\phi\ ;\ W\ ;\ \text{assert}\,\varphi\ \}\qquad (2)$$

Substituindo em (2) a definição (1) e aplicando as regras do colapso das anotações e as regras de composição com escolha obtém-se uma aproximação $H^*\ \Rightarrow H$ definida pela equação

$$H^*\ =\ \{\,I\,;\,H^*\,\}\ \ \|\ \ \{\,T\,\}\qquad (3)$$

em que I, T são os seguintes programas anotados

- $I\ \equiv\ \text{assume}\,(b\ \wedge\ \eta)\,;\ S\ ;\ \text{assert}\,\eta$
- $T\ \equiv\ \text{assume}\,(\neg b\ \wedge\ \eta)\,;\ \text{assert}\ \varphi$
- com η uma qualquer proposição tal que $\phi\ \rightarrow\ \eta$ é tautologia

Note-se que tando I como T são programas descritos na LPA e, por isso, a sua correção pode ser feita usando qualquer umas das técnicas descritas na secção anterior: resumidamente calculando a respetiva denotação e verificando se essa denotação é uma tautologia. Então

> *Se I e T forem ambos programas corretos então qualquer solução H^* de (3) é correcta e consequentemente H é correto.*

Portanto

> *A correção parcial do triplo $\{\phi\}W\{\varphi\}$ é verificada procurando encontrar uma proposição η, designada por **invariante,** tal que são tautologias*
>
> - $(b \wedge \eta) \rightarrow \mathrm{wp}_S(\eta)$
> - $(\neg b \wedge \eta) \rightarrow \varphi$
> - $\phi \rightarrow \eta$

Justificação

Caso I e T forem corretos então as respetivas denotação são ambas equivalentes a True o que significa que ambos são equivalentes ao fluxo skip. Portanto a equação (3) reescreve-se, neste caso, como

$$H^* \;=\; \{\mathrm{skip}\,;\, H^*\} \;\|\; \{\mathrm{skip}\}$$

que tem como solução $H^* = \mathrm{skip}$. Portanto H^* é correto e porque se tem $H^* \Rightarrow H$, também H é correto.

Um predicado ϕ que verifique as hipóteses deste resultado (i.e. I e T são corretos) designa-se um **invariante do ciclo**. A metodologia dos invariantes para provar a correção de ciclos que terminam consiste na procura de um invariante ϕ e na verificação de que esse predicado é realmente um invariante.

Exemplo

Considere-se de novo o triplo de Hoare apresentado no exemplo anterior

$$\{x \geq 0\} \;\; \mathrm{while}\, x \neq 0 \,\mathrm{do}\; x \leftarrow x - 1 \;\mathrm{od} \;\; \{x = 0\}$$

Vamos provar a correção deste triplo usando o invariante $\phi \;\equiv\; (x \geq 0)$. Temos os dois programas

- $I \;\equiv\; \mathrm{assume}\,(x \geq 0) \wedge (x \neq 0)\,;\, x \leftarrow x - 1\,;\, \mathrm{assert}\,(x \geq 0)$
- $T \;\equiv\; \mathrm{assert}\,(x \geq 0) \wedge (x = 0) \rightarrow (x = 0)$

Usando a denotação WPC , a correção destes programas é a verificação que são tautologias cada um dos predicados:

i. $(x > 0) \rightarrow (x \geq 0)[x/x - 1] \;\equiv\; (x > 0) \rightarrow (x - 1 \geq 0)$

ii. $(x \geq 0 \;\wedge\; x = 0) \rightarrow (x = 0)$

“Single Assignment Unfold” (SAU)

A terceira abordagem para analisar a correção de um triplo $\{\phi\}\ W\ \{\varphi\}$, sendo

$$W \equiv \text{ while } b \text{ do } S \text{ od}$$

passa por “esticar” (**unfold**) a execução do ciclo numa aproximação formada por uma sequência finita numero finito de execuções do “corpo do ciclo”.

A metodologia para provar a correção do triplo $\{\phi\}\ W\ \{\varphi\}$ consiste em calcular os diversos W_n sucessivamente, até que se encontre um para o qual o programa

$$\text{assume } \phi\,;\ W_n \text{ assert } \varphi$$

esteja correto.

Nesta abordagem vamos representar denotações W_n como um predicado que tem as variáveis livres v no programa. Teremos assim predicados $W_n(v)$ que vamos reconstruir iterativamente como vamos indicar em seguida.

A interpretação desejada para $W_n(v)$ é

> $W_n(v)$ *é válido quando num estado inicial definido pelo valor de* v *o ciclo termina em exatamente* n *passos e verifica a pós-condição.*

Adicionalmente

> *A denotação do corpo do ciclo* S *é uma função de transição* $S(v, v')$ *envolvendo os valores das variáveis antes e depois da execução de* S.

Desta forma, é simples determinar iterativamente os predicados $W_n(v)$ para $n \geq 0$.

- $W_0(v) \equiv \neg b(v) \wedge \varphi(v)$
  No estado v, o ciclo termina com correção e em 0 iterações se imediatamente $b(v)$ é falso e $\varphi(v)$ é verdadeiro.

- $W_n(v) \equiv b(v) \wedge (\bigwedge v' \cdot S(v, v') \rightarrow W_{n-1}(v'))$ par todo $n > 0$
  No estado v o ciclo termina corretamente em $n > 0$ iterações se o controlo $b(v)$ é verdadeiro e, para todo o estado v' que transite de v numa execução do corpo do ciclo, o ciclo termina corretamente em $n - 1$ passos.

O quantificado $\bigwedge v'$ é implementado declarando uma nova variável. Isto é equivalente a uma “atribuição única”: a variável v, antes de lhe ser atribuído um novo valor, preserva o antigo valor criando uma cópia e é a essa cópia que é atribuído o novo valor. Deste modo a n-ésima iteração vai ter $n + 1$ variáveis distintas.

O algoritmo SAU consiste em calcular sucessivamente os vários predicados $W_n(v)$ e testar

$$\phi(v) \rightarrow \bigvee_n W_n(v)$$

é uma tautologia. O algoritmo para com a mensagem *sucesso* logo que tal iteração seja encontrada.

Normalmente a implementação desta metodologia impõe um limite ao número de iterações que podem ser calculadas com os recursos computacionais disponíveis. Se o limite for ultrapassado sem que se verifique a condição de paragem, o algoritmo termina com a mensagem *falha* .

A seguinte implementação é uma versão não-optimizada deste algoritmo

```
from pysmt.shortcuts import *
from pysmt.typing import *

# Auxiliares
def prime(v):
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())
def fresh(v):
    return FreshSymbol(typename=v.symbol_type(),template=v.symbol_name()
+"_%d")

# A classe "Sigle Assignment Unfold"
class SAU(object):
    """Trivial representation of a while cycle and its unfolding."""
    def __init__(self, variables, pre , pos, control, trans, sname="z3"):

        self.variables = variables       # variables
        self.pre = pre                   # pre-condition as a predicate in
"variables"
        self.pos = pos                   # pos-condition as a predicate in
"variables"
        self.control = control           # cycle control as a predicate in
"variables"
        self.trans = trans               # cycle body as a binary transiti
on relation
                                         # in "variables" and "prime varia
bles"

        self.prime_variables = [prime(v) for v in self.variables]
        self.frames = [And([Not(control),pos])]
                 # inializa com uma só frame: a da terminação do ciclo
```

```
        self.solver = Solver(name=sname)

    def new_frame(self):
        freshs = [fresh(v) for v in self.variables]
        b = self.control
        S = self.trans.substitute(dict(zip(self.prime_variables,freshs)))
        W = self.frames[-1].substitute(dict(zip(self.variables,freshs)))

        self.frames.append(And([b , ForAll(freshs, Implies(S, W))]))

    def unfold(self,bound=0):
        n = 0
        while True:
            if n > bound:
                print("falha: número de tentativas ultrapassa o limite %d
"%bound)
                break

            f = Or(self.frames)
            if self.solver.solve([self.pre,Not(f)]):
                self.new_frame()
                n += 1
            else:
                print("sucesso na tentativa %d "%n)
                break
```

Exemplo

Modelar o ciclo

`while (x > 0) do x = x - 1 od`

com a pré-condição $\phi \equiv (0 \leq x) \wedge (x \leq 6)$ e a pós-condição $\varphi \equiv (x = 0)$

```
# constantes auxiliares
N = Int(6)
zero = Int(0)
um   = Int(1)
```

```
# O ciclo
x  = Symbol("x",INT)
variables = [x]

pre  =  And(LE(zero,x),LE(x,N))     # pré-condição
pos  =  Equals(x,zero)              # pós-condição
cond =  GT(x , zero)                # condição de controlo do ciclo
trans = Equals(prime(x),x - um)     # corpo do ciclo como uma relaçao de t
ransição

W = SAU(variables, pre, pos, cond, trans)

#Run
W.unfold(6)
```