Cours MALG & MOVEX

Le langage de spécification ANSI/ISO C Specification Language (ACSL)

Dominique Méry
Telecom Nancy, Université de Lorraine

Année universitaire 2023-2024

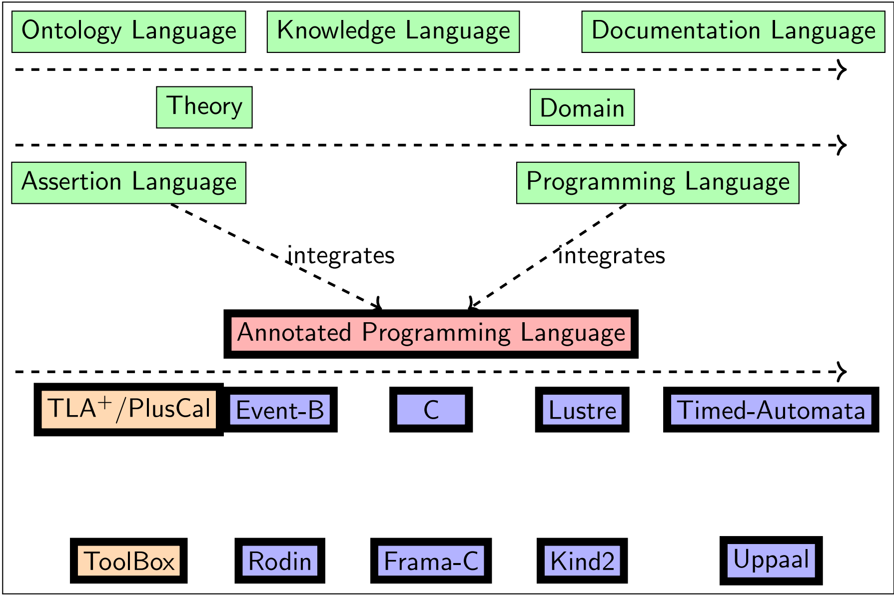
Plan

Sommaire

Sommaire du cours

- Mise en œuvre de la notion de contrat et de la vérification des contrats pour le langage C avec Frama-c
- Apprentissage d'un langage d'annotations pour les programmes C.
- Utilisation de vérificateurs comme les solveurs SMT Z3, AlterErgo ou encore Why3

Summry of concepts



Current Summary

Transformation

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow [P]\mathsf{post}(x_0, x_f)$

Transformation

- $\forall x, x_0.\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x, x_0.\mathsf{pre}(x_0) \Rightarrow x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow \forall x.x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$
- $\forall x_0.\mathsf{pre}(x_0) \Rightarrow [P]\mathsf{post}(x_0, x_f)$
- $[x := e]P(x) = P[x \mapsto e]$
- $[\mathsf{if}\ b(x)\ \mathsf{then}\ S1\ \mathsf{else}\ S2\ ]P(x) = b(x) \wedge [S1]P(x) \vee\ \mathsf{not}\ b(x)\ [S2]P(x)$

Transformation

- $[\text{while } b(x) \text{ do } S \text{ end}]P(x) =$

Transformation

- [while $b(x)$ do S end]$P(x) =$
- [if $b(x)$ then $S; w$ else $skip$]$P(x) =$

Transformation

- [while $b(x)$ do S end]$P(x) =$
- [if $b(x)$ then $S; w$ else $skip$]$P(x) =$
- $b(x) \wedge [S; w]P(x) \vee$ not $b(x)$ $P(x) =$

Transformation

- $[\text{while } b(x) \text{ do } S \text{ end}]P(x) =$
- $[\text{if } b(x) \text{ then } S; w \text{ else } skip \;]P(x) =$
- $b(x) \wedge [S; w]P(x) \vee \text{ not } b(x) \; P(x) =$
- $b(x) \wedge [S]w(P(x)) \vee \text{ not } b(x) \; P(x) = w(P(x))$

Current Summary

Current Subsection Summary

Listing 1 – difference de deux nombres

```
#include <limits.h>

/*@ requires a-b >= INT_MIN && a-b <= INT_MAX;
    assigns \nothing;
    ensures \result == (a - b);
*/
static int difference(int a, int b) {
  return a-b;
}
```

Exemple de ACSL/Frama-c

Listing 2 – incrément de nombre

```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0+2;
  @*/

int  exemple(int x0) {
  int x=x0;
  //@ assert x == x0;
  x = x + 2;
//@ assert  x== x0+2;
return x;
}
```

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
  - STATES est l'ensemble de tous les états de P : $STATES = \mathsf{X} \rightarrow \mathbb{Z}$ où X désigne les variables de P.
  - s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f.
  - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :

Intuition

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
  - STATES est l'ensemble de tous les états de P : $STATES = \mathsf{X} \rightarrow \mathbb{Z}$ où X désigne les variables de P.
  - s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f.
  - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :
$$s_0(X) = x_0,\ s_f(X) = x_f,\ s'(X) = x'$$

Intuition

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
  - STATES est l'ensemble de tous les états de P : $STATES = \mathsf{X} \rightarrow \mathbb{Z}$ où X désigne les variables de P.
  - s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f.
  - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :
    $$s_0(X) = x_0,\ s_f(X) = x_f,\ s'(X) = x'$$
  - $\mathcal{D}(P)(s_0) = s_f$ définit la relation suivante sur l'ensemble des valeurs :
    $$x_0 \xrightarrow{\mathsf{P}} x_f$$

Intuition

- Un programme P *produit* des résultats à partir de données en accord avec une sémantique :
  - STATES est l'ensemble de tous les états de P : $STATES = \mathsf{X} \rightarrow \mathbb{Z}$ où X désigne les variables de P.
  - s_0 et s_f deux états de STATES : $\mathcal{D}(P)(s_0) = s_f$ signifie que P est exécuté à partir d'un état s_0 et produit un état s_f.
  - Pour un état s de P courant, on notera $s(X) = x$ pour distinguer la valeur de la variable X et sa valeur courante en s :
    $$s_0(X) = x_0,\ s_f(X) = x_f,\ s'(X) = x'$$
  - $\mathcal{D}(P)(s_0) = s_f$ définit la relation suivante sur l'ensemble des valeurs :
    $$x_0 \xrightarrow{\mathsf{P}} x_f$$
- Un programme P *remplit* un contrat (pre,post) :
  - P transforme une variable x à partir d'une valeur initiale x_0 et produisant une valeur finale x_f : $x_0 \xrightarrow{\mathsf{P}} x_f$
  - x_0 satisfait pre : $\mathsf{pre}(x_0)$
  - x_f satisfait post : $\mathsf{post}(x_0, x_f)$
  - $\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathsf{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

Vérification du contrat

Un programme P *remplit* un contrat (pre,post) :

- P transforme une variable x à partir d'une valeur initiale x_0 et produisant une valeur finale x_f : $x_0 \xrightarrow{\text{P}} x_f$
- x_0 satisfait pre : pre(x_0)
- x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow$ post(x_0, x_f)

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

$$\left[\begin{array}{l} \text{begin} \\ 0 : P_0(x_0, x) \\ \texttt{instruction}_0 \\ \ldots \\ i : P_i(x_0, x) \\ \ldots \\ \texttt{instruction}_{f-1} \\ f : P_f(x_0, x) \\ \text{end} \end{array}\right.$$

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $P_f(x_0, x) \Rightarrow post(x_0, x)$
- conditions de vérification pour toutes les paires $\ell \longrightarrow \ell'$
  $\forall v, v' \in \textsc{Memory}$
  $$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(v_0, v) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$

Vérification du contrat avec le solveur Z3

requires $x0 \geq 0$;
ensures $x_f = x0{+}2$;
variables X

begin
$intX = x0;$
$0 : x = x0$
$X = X{+}2;$
$1 : x = x0{+}2$
end

- $x0 \geq 0 \wedge x = x_0 \Rightarrow x = x0$
- $x = x0{+}2 \Rightarrow x = x0{+}2$
- conditions de vérification $0 \longrightarrow 1$:
  $x = x0 \wedge x' = x{+}2 \Rightarrow x' = x0{+}2$
- $(x0 >= 0, x == x0, x! = x0)$
- $(x == x0{+}2, x! = x0{+}2)$
- $(x == x0, xp == x{+}2, xp! = x0{+}2)$

Listing 3 – z3 en Python

```
from numbers import Real
from z3 import *
x = Real('x')
xp = Real('xp')
x0 = Real('x0')
s = Solver()
s.add(x0 >= 0, x == x0, x != x0)
print(s.check())
s.add(x ==  x0+2, x != x0+2)
print(s.check())
s.add(x == x0,   xp == x + 2,  xp != x0+2)
print(s.check())
```

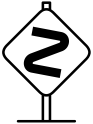
requires $x0 \geq 0$;
ensures $x_f = x0{+}2$;
variables X

$$\left[\begin{array}{l} \text{begin} \\ intX = x0; \\ 0 : x = x0 \\ X = X{+}2; \\ 1 : x = x0{+}2 \\ \text{end} \end{array}\right.$$

Conditions de vérification $0 \longrightarrow 1$:

- $x = x0 \wedge x' = x{+}2 \Rightarrow x' = x0{+}2$
- $x = x0 \Rightarrow (x' = x{+}2 \Rightarrow x' = x0{+}2)$
- $x = x0 \Rightarrow (x{+}2 = x0{+}2)$
- $wp(X := X{+}2)(x = x0{+}2) = (x{+}2 = x0{+}2)$
- $x = x0 \Rightarrow wp(X := X{+}2)(x = x0{+}2)$

- $x0 \geq 0 \wedge x = x_0 \Rightarrow x = x0$
- $x = x0{+}2 \Rightarrow x = x0{+}2$
- $x = x0 \Rightarrow wp(X := X{+}2)(x = x0{+}2)$



calcul de $wp(X := X{+}2)(x = x0{+}2)$

Listing 4 – incrément de nombre

```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0+1;
  @*/

int  exemple(int x0) {
  int x=x0;
  //@ assert x == x0;
  x = x + 2;
//@ assert  x== x0+2;
return x;
//@ assert  \result== x0+2;
}
```

Exemple avec wp de ACSL/Frama-c après wp

Listing 5 – incrément de nombre

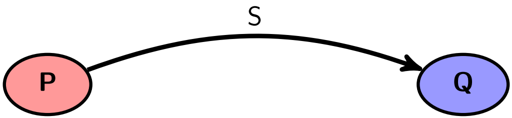
```
/*@ requires x0 >= 0;
    assigns \nothing;
    ensures \result == x0;
  @*/

int exemple(int x0) {
  int x=x0;
//@ assert x == x0+1;
  x = x + 2;
//@ assert x== x0+2;
return x;

}
```

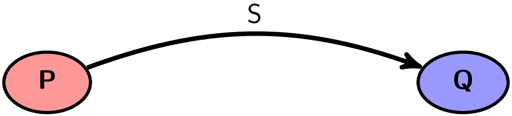
Asserted Program $\{P\}$ S $\{Q\}$

$$\{P\}\ S\ \{Q\}$$



Asserted Program {P} S {Q}

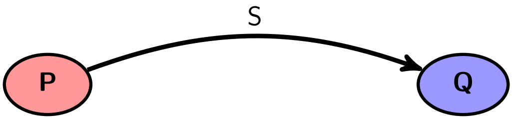
{P} S {Q}



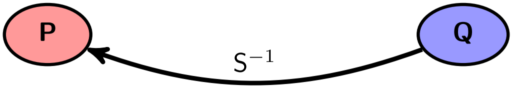
$$SP(S)(P) \Rightarrow \mathsf{Q}$$

Asserted Program {P} S {Q}

{P} S {Q}

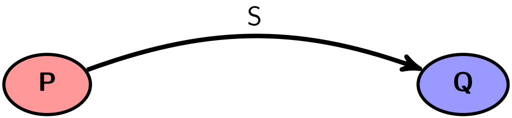


$SP(S)(P) \Rightarrow$ Q

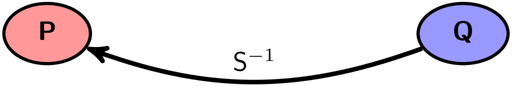


Asserted Program {P} S {Q}

{P} S {Q}



$$SP(S)(P) \Rightarrow \mathrm{Q}$$



$$P \Rightarrow WP(S)(\mathrm{Q})$$

Current Subsection Summary

Opérateur WP

Soit STATES l'ensemble des états sur l'ensemble X des variables. Soit S une instruction de programme sur X. Soit A une partie de STATES. $s \in WP(S)(A)$, si la condition suivante est vérifiée :

$$\left(\begin{array}{l} \forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow t \in A \\ \wedge \\ \exists t \in STATES : \mathcal{D}(S)(s) = t \end{array}\right)$$

- $WP(X := X{+}1)(A) = \{s \in STATES | s[X \mapsto s(X) \oplus 1] \in A\}$
- $WP(X := Y{+}1)(A) = \{s \in STATES | s[X \mapsto s(Y) \oplus 1] \in A\}$
- $WP(while\ X > 0\ do\ X := X{-}1\ od)(A) = \{s \in STATES | (s(X) \leq 0) \vee (s(X) \in A \wedge s(X) < 0)\}$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(A) = \{s \in STATES | (s(X) \in A \wedge s(X) \leq 0)\}$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(\varnothing) = \varnothing$
- $WP(while\ x > 0\ do\ x := x{+}1\ od)(STATES) = \{s \in STATES | s(X) \leq 0\}$

Propriétés

- WP est une fonction monotone pour l'inclusion d'ensembles de STATES.
- $WP(S)(\varnothing) = \varnothing$
- $WP(S)(A \cap B) = WP(S)(A) \cap WP(S)(B)$
- $WP(S)(A) \cup WP(S)(B) \subseteq WP(S)(A \cup B)$
- Si S est déterministe, $WP(S)(A \cup B) = WP(S)(A) \cup WP(S)(B)$

- WP est un opérateur avec le profil suivant

  pour toute instruction S du langage de programmation,
  $$WP(S) \in \mathcal{P}(STATES) \nrightarrow \mathcal{P}(STATES)$$
- $(\mathcal{P}(STATES), \subseteq)$ est un treillis complet.
- $(Pred, \Rightarrow)$ est une structure où
  - (1) $Pred$ est une *extension* du langage d'expressions booléennes
  - (2) $Pred$ est une *intension* introduite comme un langage d'assertions
  - $\Rightarrow$ est l'implication
  - $s \in A$ correspond une assertion P vraie en s notée $P(s)$.

Définition structurelle des transformateurs de prédicats

- S est une instruction de STATS.
- T est le type ou les types des variables et D est la constante ou les constantes Définie(s).
- P est un prédicat du langage Pred
- X est une variable de programme
- $E(X,D)$ (resp. $B(X,D)$) est une expression arithmétique (resp. booléenne) dépendant de X et de D.
- x est la valeur de X (X contient la valeur x).
- $e(x,d)$ (resp. $b(x,d)$) est l'expression arithmétique (resp. booléenne) du langage Pred associée à l'expression $E(X,D)$ (resp. $B(X,D)$) du langage des expressions arithmétiques (resp. booléennes) du langage de programmation Prog
- $b(x,d)$ est l'expression arithmétique du langage Pred associée à l'expression $E(X,D)$ du langage des expressions arithmétiques du langage de programmation Prog

Définition structurelle des transformateurs de prédicats

S	wp(S)(P)
X :=E(X,D)	$P[e(x,d)/x]$
SKIP	P
$\mathrm{S}_1;\mathrm{S}_2$	$wp(\mathrm{S}_1)(wp(\mathrm{S}_2)(P))$
IF B S_1 ELSE S_2 FI	$(B \Rightarrow wp(S_1)(P)) \wedge (\neg B \Rightarrow wp(S_2)(P))$
WHILE B DO S OD	$\mu.(\lambda X.(B \Rightarrow wp(S)(X)) \wedge (\neg B \Rightarrow P))$

- $wp(X := X{+}5)(x \geq 8) \stackrel{def}{=} x{+}5 \geq 8 \stackrel{\sim}{=} x \geq 3$
- $wp(\text{WHILE } x > 1 \text{ DO } X := X{+}1 \text{ OD})(x = 4) = FALSE$
- $wp(\text{WHILE } x > 1 \text{ DO } X := X{+}1 \text{ OD})(x = 0) = x = 0$

Current Subsection Summary

. . . .

☒ Definition(Axiomes et règles d'inférence)

- Axiome d'affectation : $\{P(e/x)\}$**X :=E(X)**$\{P\}$.
- Axiome du saut : $\{P\}$**skip**$\{P\}$.
- Règle de composition : Si $\{P\}$**S**$_1\{R\}$ et $\{R\}$**S**$_2\{Q\}$, alors $\{P\}$**S**$_1$ **;S**$_2\{Q\}$.
- Si $\{P \wedge B\}$**S**$_1\{Q\}$ et $\{P \wedge \neg B\}$**S**$_2\{Q\}$, alors $\{P\}$**if B then S**$_1$ **then S**$_2$ **fi**$\{Q\}$.
- Si $\{P \wedge B\}$**S**$\{P\}$, alors $\{P\}$**while B do S od**$\{P \wedge \neg B\}$.
- Règle de renforcement/affaiblissement : Si $P' \Rightarrow P$, $\{P\}$**S**$\{Q\}$, $Q \Rightarrow Q'$, alors $\{P'\}$**S**$\{Q'\}$.

. . . .

Exemple de preuve $\{x=1\}$**Z :=X ;X :=Y ;Y :=Z**$\{y=1\}$

- (1) $x=1 \Rightarrow (z=1)[x/z]$ (propriété logique)
- (2) $\{(z=1)[x/z]\}$**Z :=X**$\{z=1\}$ (axiome d'affectation)
- (3) $\{x=1\}$**Z :=X**$\{z=1\}$ (Règle de renforcement/affaiblissement avec (1) et (2))
- (4) $z=1 \Rightarrow (z=1)[y/x]$ (propriété logique)
- (5) $\{(z=1)[y/x]\}$**X :=Y**$\{z=1\}$ (axiome d'affectation)
- (6) $\{z=1\}$**X :=Y**$\{z=1\}$ (Règle de renforcement/affaiblissement avec (4) et (5))
- (7) $z=1 \Rightarrow (y=1)[z/y]$ (propriété logique)
- (8) $\{(z=1)[x/z]\}$**Y :=Z**$\{y=1\}$ (axiome d'affectation)
- (9) $\{z=1\}$**Y :=Z**$\{y=1\}$ (Règle de renforcement/affaiblissement avec (7) et (8))
- (10) $\{x=1\}$**Z :=X ;X :=Y ;**$\{z=1\}$ (Règle de composition avec 3 et 6)
- (11) $\{x=1\}$**Z :=X ;X :=Y ;Y :=Z**$\{y=1\}$ (Règle de composition avec 11 et 9)

Sémantique des triplets de Hoare

. . . .

☒ Definition

$\{P\}\mathbf{S}\{Q\}$ est défini par $\forall s,t \in STATES : P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t)$

. . . .

. . . .

☺ PropertyCorrection du système axiomatique des programmes commentés

- S'il existe une preuve construite avec les règles précédentes de $\{P\}\mathbf{S}\{Q\}$, alors $\{P\}\mathbf{S}\{Q\}$ est valide.
- Si $\{P'\}\mathbf{S}\{Q'\}$ est valide et si le langage d'assertions est suffisamment expressif, alors il existe une preuve construite avec les règles précédentes de $\{P\}\mathbf{S}\{Q\}$.

. . . .

☒ Definition

Un langage d'assertions est la donnée d'un ensemble de prédicats et d'opérateurs de composition comme la disjonction et la conjonction ; il est muni d'une relation d'ordre partielle appelée implication. On le notera $(\textsc{Pred}, \Rightarrow, \mathbf{false}, \mathbf{true}, \wedge, \vee)$: $(\textsc{Pred}, \Rightarrow, \mathbf{false}, \mathbf{true}, \wedge, \vee)$ est un treillis complet.

- $\{P\}\mathbf{S}\{Q\}$
- $\forall s,t \in STATES : P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t)$
- $\forall s \in STATES : P(s) \Rightarrow (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$

....

Définition de wlp

$$wlp(S)(Q) \stackrel{def}{=} (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$$

....

$$wlp(S)(Q) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t \wedge \overline{Q}(t))}$$

....

....

Lien entre wp et wlp

- $loop(S) \equiv \overline{(\exists t \in STATES : \mathcal{D}(S)(s) = t}$ (ensemble des états qui ne permettent pas à S de terminer)
- $wp(S)(Q) \equiv wlp(S)(Q) \wedge \overline{loop(S)}$

....

Définition de wlp

☒ Definition

$WLP(S)(P) = \nu\lambda X.((B \wedge wlp(BS)(X)) \vee (\neg B \wedge P))$

☺ Property

- Si $P \Rightarrow Q$, then $wlp(S)(P) \Rightarrow wlp(S)(Q)$.

⊠ Definitiontriplets de Hoare

$$\{P\}\mathbf{S}\{Q\} \stackrel{def}{=} P \Rightarrow wlp(S)(Q)$$

Axiomatisation de la Logique de Hoare

☒ Definitiontriplets de Hoare

$\{P\}\mathbf{S}\{Q\} \stackrel{def}{=} P \Rightarrow wlp(S)(Q)$

☒ Definition(Axiomes et règles d'inférence)

- Axiome d'affectation : $\{P(e/x)\}$**X :=E(X)**$\{P\}$.
- Axiome du saut : $\{P\}$**skip**$\{P\}$.
- Règle de composition : Si $\{P\}\mathbf{S}_1\{R\}$ et $\{R\}\mathbf{S}_2\{Q\}$, alors $\{P\}\mathbf{S}_1\,;\mathbf{S}_2\{Q\}$.
- Si $\{P \wedge B\}\mathbf{S}_1\{Q\}$ et $\{P \wedge \neg B\}\mathbf{S}_2\{Q\}$, alors $\{P\}$**if B then S**$_1$ **then S**$_2$ **fi**$\{Q\}$.
- Si $\{P \wedge B\}\mathbf{S}\{P\}$, alors $\{P\}$**while B do S od**$\{P \wedge \neg B\}$.
- Règle de renforcement/affaiblissement : Si $P' \Rightarrow P$, $\{P\}\mathbf{S}\{Q\}$, $Q \Rightarrow Q'$, alors $\{P'\}\mathbf{S}\{Q'\}$.

Vérification

- $\{P\}\mathbf{S}\{Q\}$
- $\forall s \in STATES.P(s) \Rightarrow wlp(S)(Q)(s)$
- $\forall s \in STATES.P(s) \Rightarrow (\forall t \in STATES : \mathcal{D}(S)(s) = t \Rightarrow Q(t))$
- $\forall s, t \in STATES.P(s) \wedge \mathcal{D}(S)(s) = t \Rightarrow Q(t))$
- Correction : Si on a construit une preuve de $\{P\}\mathbf{S}\{Q\}$ avec les règles de la logique de Hoare, alors $P \Rightarrow wlp(S)(Q)$
- Complétude sémantique : Si $P \Rightarrow wlp(S)(Q)$, alors on peut construire une preuve de $\{P\}\mathbf{S}\{Q\}$ avec les règles de la logique de Hoare si on peut exprimer $wlp(S)(P)$ dans le langae d'assertions.

⊠ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

. . . .

☒ Definitiontriplets de Hoare Correction Totale

$[P]\mathbf{S}[Q] \stackrel{def}{=} P \Rightarrow wp(S)(Q)$

. . . .

. . . .

☒ Definition(Axiomes et règles d'inférence)

- Axiome d'affectation : $[P(e/x)]$**X :=E(X)**$[P]$.
- Axiome du saut : $[P]$**skip**$[P]$.
- Règle de composition : Si $[P]\mathbf{S}_1[R]$ et $[R]\mathbf{S}_2[Q]$, alors $[P]\mathbf{S}_1\ \mathbf{;}\mathbf{S}_2[Q]$.
- Si $[P \wedge B]\mathbf{S}_1[Q]$ et $[P \wedge \neg B]\mathbf{S}_2[Q]$, alors $[P]$**if B then S**$_1$ **then S**$_2$ **fi**$[Q]$.
- Si $[P(n+1)]\mathbf{S}[P(n)]$, $P(n+1) \Rightarrow b$, $P(0) \Rightarrow \neg b$, alors $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$.
- Règle de renforcement/affaiblissement : Si $P' \Rightarrow P$, $[P]\mathbf{S}[Q]$, $Q \Rightarrow Q'$, alors $[P']\mathbf{S}[Q']$.

. . . .

Correction

:
Si $[P]$**S**$[Q]$ est dérivé selon les règles ci-dessus, alors $P\wp(S)5Q)$.

- $[P(e/x)]$**X :=E(X)**$[P]$ est valide : $wp(X := E)(P)/x = P(e/x)$.
- $[\exists n \in \mathbb{N}.P(n)]$**while B do S od**$[P(0)]$: si s est un état de P(n) alors au bout de n boucles on atteint un état s_f tel que P(0) est vrai en s_f.

Complétude

:
Si $P \Rightarrow wp(S)(Q)$, alors il existe une preuve de $[P]$**S**$[Q]$ construites avec les règles ci-dessus,

- $P \Rightarrow wp(X := E(X))(Q)$: $P \Rightarrow Q(e/x)$ et $[Q(e/x)]$**X :=E(X)**$[Q]$ constituent une preuve.
- $P \Rightarrow wp(while)(Q)$:
  - On construit la suite de $P(n)$ en définissant $P(n) = W_n$.
  - On vérifie que cela vérifie la règle du while.

Transformations de programmes annotés (1)

$//@$ assert $P(v0,v)$:
$S1; S2$
$//@$ assert $Q(v0,v)$:

- Application de la propriété : $wp(S1;S2)(A) = wp(S1)(wp(S2)(A))$
-

$//@$ assert $P(v0,v)$:
$S1;$
$//@$ assert $wp(S2)(Q(v0,v))$:
$S2;$
$//@$ assert $Q(v0,v)$:

$//@$ assert $P(v0,v)$:
$//@$ assert $xp(S1)(wp(S2)(Q(v0,v)))$:
$S1;$
$//@$ assert $wp(S2)(Q(v0,v))$:
$S2;$
$//@$ assert $Q(v0,v)$:

Transformations de programmes annotés (2)

```
//@ assert P(v0,v) :
IF B THEN
   S1
ELSE
   S2
FI
//@ assert Q(v0,v) :
```

- Application de la propriété : $wp(if(B,S1,S2)(A) = b \wedge wp(S1)(A) \vee \neg B \wedge wp(S2)(A)$.
-

```
//@ assert P(v0,v) :
IF B THEN
   S1
ELSE
   S2
FI
//@ assert Q(v0,v) :
```

```
//@ assert P(v0,v) :
IF B THEN
   S1
//@ assert Q(v0,v) :
ELSE
   S2
//@ assert Q(v0,v) :
FI
//@ assert Q(v0,v) :
```

Transformations de programmes annotés (2)

```
//@ assert P(v0, v) :
IF B THEN
  S1
//@ assert Q(v0, v) :
ELSE
  S2
//@ assert Q(v0, v) :
FI
//@ assert Q(v0, v) :
```

Transformations de programmes annotés (2)

```
//@ assert P(v0, v) :
IF B THEN
   S1
//@ assert Q(v0, v) :
ELSE
   S2
//@ assert Q(v0, v) :
FI
//@ assert Q(v0, v) :
```

```
//@ assert P(v0, v) :
IF B THEN
//@ assert B ∧ wp(S2)(Q(v0, v)) :
   S1
//@ assert Q(v0, v) :
ELSE
//@ assert ¬B ∧ wp(S2)(Q(v0, v)) :
   S2
//@ assert Q(v0, v) :
FI
```

Transformations de programmes annotés (2)

//@ assert $P(v0, v)$:
IF B THEN
  $S1$
//@ assert $Q(v0, v)$:
ELSE
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $B \wedge wp(S2)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg B \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:
FI

//@ assert $P(v0, v)$:
IF B THEN
//@ assert $b \wedge wp(S1)(Q(v0, v))$:
  $S1$
//@ assert $Q(v0, v)$:
ELSE
//@ assert $\neg b \wedge wp(S2)(Q(v0, v))$:
  $S2$
//@ assert $Q(v0, v)$:
FI
//@ assert $Q(v0, v)$:

Transformations de programmes annotés (2)

```
//@ assert P(v0,v) :
IF B THEN
  S1
//@ assert Q(v0,v) :
ELSE
  S2
//@ assert Q(v0,v) :
FI
//@ assert Q(v0,v) :
```

```
//@ assert P(v0,v) :
IF B THEN
//@ assert B ∧ wp(S2)(Q(v0,v)) :
  S1
//@ assert Q(v0,v) :
ELSE
//@ assert ¬B ∧ wp(S2)(Q(v0,v)) :
  S2
//@ assert Q(v0,v) :
FI
```

```
//@ assert P(v0,v) :
IF B THEN
//@ assert b ∧ wp(S1)(Q(v0,v)) :
  S1
//@ assert Q(v0,v) :
ELSE
//@ assert ¬b ∧ wp(S2)(Q(v0,v)) :
  S2
//@ assert Q(v0,v) :
FI
//@ assert Q(v0,v) :
```

- $b \wedge P(v0, v) \Rightarrow$
  $b \wedge wp(S1)(Q(v0, v))$
- $\neg b \wedge P(v0, v) \Rightarrow$
  $\neg b \wedge wp(S2)(Q(v0, v))$

Transformations de programmes annotés (3)

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
WHILE B THEN
   S
OD
//@ assert Q(v0, v) :
```

- Application de la règle de nla logique de Hoare pour l'itération

Transformations de programmes annotés (3)

//@ assert $P(v0, v)$:
//@ loop invariant $I(v0, v)$:
WHILE B THEN
  S
OD
//@ assert $Q(v0, v)$:

- Application de la règle de nla logique de Hoare pour l'itération

//@ assert $P(v0, v)$:
//@ loop invariant $I(v0, v)$:
//@ assert $I(v0, v)$:
WHILE B THEN
//@ assert $b \wedge I(v0, v)$:
  S
//@ assert $I(v0, v)$:
OD
//@ assert $Q(v0, v)$:

Transformations de programmes annotés (3)

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
WHILE B THEN
   S
OD
//@ assert Q(v0, v) :
```

- Application de la règle de nla logique de Hoare pour l'itération

```
//@ assert P(v0, v) :
//@ loop invariant I(v0, v) :
//@ assert I(v0, v) :
WHILE B THEN
//@ assert b ∧ I(v0, v) :
   S
//@ assert I(v0, v) :
OD
//@ assert Q(v0, v) :
```

- $b \wedge I(v0, v) \Rightarrow wp(S)(I(v0, v))$
- $P(v0, v) \Rightarrow I(v0, v))$
- $\neg b \wedge I(v0, v) \Rightarrow Q(v0, v)$

Sommaire sur les transformations

- Etablir que l'invariant est préservé.
- Appliquer les wps sur les assertions selon les instructions.

Current Subsection Summary

Assertions

- Assertions à un point du programme :

```
/*@  assert pred; */

//@  assert pred;
```

- Assertions à un point du programme selon les comportements.

```
/*@  for id1,id2, ..., idn: assert pred; */
```

Vérification avec wp

(Incrément d'un nombre)

Listing 6 – project-divers/compwp0.c

```
#define x0 5
/*@ assigns \nothing;*/
int  exemple() {
  int x=x0;
  //@ assert x == x0;
  x = x + 1;
//@ assert  x== x0+1;
return x;
}
```

Vérification avec wp

(Incrément d'un nombre)

Listing 7 – project-divers/compwp00.c

```
#define x0 5
/*@ assigns \nothing;
*/
int  exemple() {
  int x=x0;
  //@ assert x == x0;
//@ assert  x+1== x0+1;
  x = x + 1;
//@ assert  x== x0+1;
return x;
}
```

(Incrément d'un nombre)

Listing 8 – project-divers/compwp000.c

```
#define x0 5
/*@ assigns \nothing;
*/
int  exemple() {
  //@ assert x0 == x0;
//@ assert  x0+1== x0+1;
  int x=x0;
  //@ assert x == x0;
//@ assert  x+1== x0+1;
  x = x + 1;
//@ assert  x== x0+1;

return x;
}
```

Vérification avec wp (skip)

(Condition de vérification)

Listing 9 – project-divers/compwp0000.c

```
...
//@ assert x0 == x0;
//@ assert  x0+1== x0+1;
...
```

- condition de vérification : $x0 == x0 \Rightarrow x0{+}1 == x0{+}1$
- Le simplificateur QED produit le prédicat TRUE et avlide le

Contrat valide et contrat invalide

(Contrat invalide)

Listing 10 – project-divers/anno0.c

```
int main(void){
  signed long int x,y,z;
  x = 1;
  /*@ assert x == 1; */
  y = 2;
  /*@  assert x == 1 && y == 2; */
  z = x * y;
  /*@  assert x == 1 && y == 1 && z==2; */
  return 0;
  }
```

(Contrat valide)

Listing 11 – project-divers/anno00.c

```
int main(void){
  signed long int x,y,z; //     int x,y,z;
  x = 1;
  /*@  assert x == 1; */
  y = 2;
  /*@ assert x == 1 && y == 2; */
  z = x *y;
  /*@  assert  x == 1 && y == 2 && z == 2; */
  return 0;
  }
```

```
...
/*@ loop invariant I;
  @ loop assigns L;
*/
...
```

(Invariant de boucle)

Listing 12 – project-divers/anno5.c

```
/*@ requires a >= 0 && b >= 0;
  ensures 0 <= \result;
  ensures \result < b;
  ensures \exists integer k; a == k * b + \result;
*/
int rem(int a, int b) {
  int r = a;
  /*@
    loop invariant
    (\exists integer i; a == i * b + r) &&
    r >= 0;
    loop assigns r;
   */
  while (r >= b) { r = r - b;};
  return r;
}
```

Exemple d'invariant de boucle

(Invariant de boucle)

Listing 13 – project-divers/anno6.c

```
/*@ requires a >= 0 && b >= 0;
  ensures 0 <= \result;
  ensures \result < b;
  ensures \exists integer k; a == k * b + \result;
*/
int rem(int a, int b) {
  int r = a;
  /*@
    loop invariant
    (\exists integer i; a == i * b + r) &&
    r >= 0;
    loop assigns r;
   */
  while (r >= b) { r = r - b;};
  return r;
}
```

Résultat de l'analyse

Echec de la preuve

L'invariant est insuffisamment informatif pour être prouvé et il faut ajouter une information sur y.

```
frama-c -wp anno6.c
[kernel] Parsing anno6.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] anno6.c:8: Warning: Missing assigns clause (assigns 'everything' i
[wp] 2 goals scheduled
[wp] [Alt-Ergo 2.3.3] Goal typed_f_loop_invariant_preserved : Timeout (
[wp] [Cache] found:1
[wp] Proved goals:    1 / 2
  Qed:               1  (0.57ms)
  Alt-Ergo 2.3.3:    0  (interrupted: 1) (cached: 1)
[wp:pedantic-assigns] anno6.c:1: Warning:
  No 'assigns' specification for function 'f'.
  Callers assumptions might be imprecise.
```

Exemple d'invariant de boucle

Analyse avec succès

L'invariant est plus précis et donne des conditions liant x et y.

(Invariant de boucle)

Listing 14 – project-divers/anno7.c

```
int f() {
int x = 0;
int y = 10;
/*@
    loop invariant
    0 <= x < 11 && x+y == 10;
   */
while (y > 0) {
  x++;
  y--;
}
 return 0;
}
```

Résultat de l'analyse

```
frama-c -wp anno7.c
[kernel] Parsing anno7.c (with preprocessing)
[wp] Warning: Missing RTE guards
[wp] anno7.c:8: Warning: Missing assigns clause (assigns 'everything' i
[wp] 2 goals scheduled
[wp] [Cache] found:1
[wp] Proved goals:    2 / 2
  Qed:               1  (0.32ms-3ms)
  Alt-Ergo 2.3.3:    1  (6ms) (8) (cached: 1)
[wp:pedantic-assigns] anno7.c:1: Warning:
  No 'assigns' specification for function 'f'.
  Callers assumptions might be imprecise.
```

Current Summary

Terminaison

- Un variant est une quantité qui décroît au cours de la boucle.
- Deux possibilités d'analyse sont possibles :
  - Terminaison d'une boucle (`variant`)
  - Terminaison de l'appel d'une fonction récursive (`decreawse`)

(Variant)

Listing 15 – project-divers/variant2.c

```
//@ loop variant e;

//@ decreases e;
```

Terminaison de boucle

- La terminaison est assurée en montrant que chaque boucle termine.
- Une boucle est caractérisée par une expression `expvariant(x)` appelée `variant` qui doit décroître à chaque exécution du corps de la boucle S où x_1 et x_2 sont les valeurs de X respectiveùent au début de la boucle S et à la fin de S :
  $\forall x_1, x_2 . b(x_1) \wedge x_1 \xrightarrow{\mathsf{S}} x_2 \Rightarrow \mathsf{expvariant}(x_1) > \mathsf{expvariant}(x_2)$

(Variant)

Listing 16 – project-divers/variant1.c

```
/*@ requires n > 0;
  terminates n > 0;

  ensures \result == 0;
*/
int code(int n) {
  int x = n;
  /*@ loop invariant x >= 0 &&  x <= n;
    loop assigns x;
    loop variant x;
   */
  while (x != 0) {
    x = x - 1;
  };
  return x;
}
```

Exemple de variant de boucle

(Variant)

Listing 17 – project-divers/variant3.c

```
int f() {
int x = 0;
int y = 10;
/*@
    loop invariant
    0 <= x < 11 && x+y == 10;
    loop variant y;
  */
while (y > 0) {
  x++;
  y--;
}
 return 0;
}
```

Exemple d’appel récursif

(Variant)

Listing 18 – project-divers/variant4.c

```
g/*@ requires n <= 12;
  @ decreases n;
  @*/
int fact(int n){
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

Modèle de mémoire HOARE

- Pas de gestion de la mémoire comme les pointeurs
- Affectation à chaque variable une variable logique
- x++ avec x de type int et la C-variable est affectée à deux L-variables $x2 = x1 + 1$.

Exemples d'annotation

(Variant)

Listing 19 – project-divers/wp2.c

```
/*@CONSOLE
#include <LIMITS.h>
int q1() {
   int x=10,y=30,z=20;
//@ assert x== 10 && y == z+x && z==2*x;
 y= z+x;
  //@ assert  x== 10 && y == x+2*10;
 x = x+1;
//@ assert  x-1== 10 && y == x-1+2*10;
 return(0);
 }
```

(Variant)

Listing 20 – project-divers/wp3.c

```
int q1() {
  int c = 2 ;
  /*@ assert c == 2;  */
  int x;
  /*@ assert c == 2;  */
  x = 3 * c ;
  /*@ assert x == 6;  */
  return(0);
}
```

(Variant)

Listing 21 – project-divers/wp4.c

```
int main()
{
  int a = 42; int b = 37;
  int c = a+b; // i:1
//@assert b == 37 ;
  a -= c; // i:2
  b += a; // i:3
//@assert b == 0 && c == 79;
  return(0);
}
```

(Variant)

Listing 22 – project-divers/wp5.c

```
int main()
{
  int z; // instruction 8
  int a = 4; // instruction 7
//@assert a == 4 ;
  int b = 3; // instyruction 6
//@assert b == 3 && a == 4;
  int c = a+b; // instruction 4
/*@ assert b == 3 && c == 7 && a == 4 ; */
  a += c; // instruction 3
  b += a; // instruction 2
//@ assert a == 11 && b == 14 && c == 7 ;
//@ assert a +b == 25 ;
  z = a*b; // instruction 1
//@assert a == 11 && b == 14 && c == 7 && z == 154;
  return(0);
}
```

(Variant)

Listing 23 – project-divers/wp6.c

```
int main()
{
  int a = 4;
  int b = 3;
  int c = a+b; // i:1
  a += c; // i:2
  b += a; // i:3
//@assert a == 11 && b == 14 && c == 7 ;
  return(0);
}
```

(Variant)

Listing 24 – project-divers/wp7.c

```
/*@ ensures x == a;
  ensures y == b;
 */
void swap1(int a, int b) {
  int x = a;
  int y = b;
  //@ assert x == a && y == b;
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
  //@ assert x == a && y == a;
}


void swap2(int a, int b) {
  int x = a;
  int y = b;
  //@ assert x == a && y == b;
  x = x + y;
  y = x - y;
  x = x - y;
  //@ assert x == b && y == a;
}

/*@ requires \valid(a);
  requires \valid(b);
  ensures *a == \old(*b);
  ensures *b == \old(*a);
 */
void swap3(int *a, int *b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Current Summary

Current Subsection Summary

Méthode de vérification pour la correction partielle et RTE

Un programme P *remplit* un contrat (pre,post) :

- P transforme une variable x à partir d'une valeur initiale x_0 et produisant une valeur finale x_f : $x_0 \xrightarrow{P} x_f$
- x_0 satisfait pre : pre(x_0) and x_f satisfait post : post(x_0, x_f)
- pre$(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow$ post(x_0, x_f)
- $\mathbb{D}$ est le domaine RTE de X

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X

begin
$0 : P_0(x_0, x)$
`instruction`$_0$
...
$i : P_i(x_0, x)$
...
`instruction`$_{f-1}$
$f : P_f(x_0, x)$
end

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $P_f(x_0, x) \Rightarrow post(x_0, x)$
- Pour toute paire d'étiquettes ℓ, ℓ' telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $x, x' \in \textsc{Memory}$ $$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$
- Pour toute paire d'étiquettes m, n telle que $m \longrightarrow n$, on vérifie que, $\forall x, x' \in \textsc{Memory} : pre(x_0) \wedge P_m(x_0, x) \Rightarrow$ **DOM**$(m, n)(x)$

Définition du contrat et des axiomes (spécification)

- Définir la fonction mathématique à calculer.
- Définition inductive sous forme d'axiomes.

Listing 25 – mathfact

```
#ifndef _A_H
#define _A_H
/*@ axiomatic mathfact {
  @ logic integer mathfact(integer n);
  @ axiom mathfact_1: mathfact(1) == 1;
  @ axiom mathfact_rec: \forall integer n; n > 1
  ==> mathfact(n) == n * mathfact(n-1);
  @ } */

/*@ requires n > 0;
   decreases n;
   ensures \result == mathfact(n);
   assigns \nothing;
*/
int codefact(int n);
#endif
```

Définition du contrat et des axiomes (programmation)

- Définir le calcul codefact
- Définition de l'agorithme réalisant le calcul

Listing 26 – codefact

```
#include "factorial.h"

int codefact(int n) {
  int y = 1;
  int x = n;
  /*@ loop invariant x >= 1 && x <= n && mathfact(n) == y *
    loop assigns x, y;
    loop variant x;
   */
  while (x != 1) {
    y = y * x;
    x = x - 1;
  };
  return y;

}
```

Définition du contrat et des axiomes (démarche)

- La spécification d'une fonction mathfact à calculer nécessite de la définir mathématiquement.
- Cette définition axiomatique est fondée sur une définition inductive de la fonction mathfact qui sera utilisée dans les assertions pour le contrat définissant la fonction informatique de calcul.
- La relation entre la valeur calculée \result et mathfact(n) est établie dans la partie ensures : \result == mathfact(n).
- On peut aussi écrire codefact(n)==mathfact(n) : l'appel de la fonction codefact pour la valeur n renvoie une valeur égale à celle de mathfact(n).

Current Summary

Programmation par contrat

- La fonction appelante doit garantir que l'assertion (requires)) $P1 \wedge \ldots \wedge Pn$ est vraie au point d'appel.
- La fonction appelée renvoie un résultat satisfaisant (ensures) $E1 \wedge \ldots \wedge Em$
- Les variables qui ne figurent pas dans l'ensemble $L1 \cup \ldots \cup Lp$ ne sont pas modifiées.

Listing 27 – schema de contrat

```
/*@ requires P1;...;requires Pn;
  @ assigns L1;...;assigns Lm;
  @ ensures E1;...;ensures Ep;
  @*/
```

Listing 28 – division

```
#ifndef _A_H
#define _A_H
#include "structures.h"
/*@ requires a >= 0 && b >= 0;
@ behavior b :
  @ assumes b == 0;
  @ assigns \nothing;
  @ ensures \result.q == -1  && \result.r == -1 ;
@ behavior B2:
  @ assumes b != 0;
  @ assigns \nothing;
  @ ensures 0 <= \result.r;
  @ ensures \result.r < b;
  @ ensures  a ==  b * \result.q + \result.r;
*/
struct s  division(int a, int b);
#endif
```

Listing 29 – division

```
#include <stdio.h>
#include <stdlib.h>
#include "division.h"

struct s  division(int a, int b)
{   int rr = a;
    int qq = 0;
    struct s  silly = {-1,-1};
    struct s resu;
    if (b == 0) {
      return silly;
    }
    else
      {
  /*@
    loop invariant
    ( a ==  b*qq + rr) &&
    rr >= 0;
```

Current Summary

Listing 30 – schema de contrat

```
/*@ requires P1 && ... && Pn;
  @ assigns L1,..., Lm;
  @ ensures E1 && ... && Ep;
  @*/
```

- $\backslash old(x)$ fait référence à la valeur de x à l'appel ou dans le pré-état (x0)
- $\backslash result$ fait référence à la valeur du résultat de l'appel.
- Ces deux expressions ne peuvent être utilisées uniquement dans une clause ensure.
- $\backslash valid(x)$ signifie que x est une adresse valide et $\cdot x$ désigne le contenu de x et $\&x$ désigne l'adresse de x.

Listing 31 – change1.c

```
/*@ requires \valid(a) && *a >= 0;
  @ assigns *a;
  @ ensures   *a == \old(*a)+2 && \result == 0;
*/
int  change1(int *a)
{   int x = *a;
    x = x + 2;
    *a = x;
    return 0;
}
```

Current Subsection Summary

Programmation par contrat (Exemple)

- Pré-condition ou requires $x \geq 0$
- Postcondition ou ensures $result \cdot result \leq x < (result+1) \cdot (result+1)$

Listing 32 – contrat squareroot

```
/*@ requires x >= 0;
  @ ensures \result >= 0;
  @ ensures \result * \result <= x;
  @ ensures (\result+1) * (\result + 1) > x;
  @*/
int squareroot(int x);
```

Programmation par contrat (Exemple)

- Précondition $def(p)$
- Postcondition $\star p = \star p0{+}1$

Listing 33 – contrat increment

```
/*@ requires \valid(p);
  @ assigns *p;
  @ ensures *p == \old(*p) + 1;
  @*/
void increment(int *p);
```

Current Subsection Summary

Listing 34 – schema de contrat

```
/*@ requires P;
  @ behavior B1;
  @   assumes  A1;
  @   requires R1;
  @    assigns L1;
  @    ensures E1;
  @ behavior B2;
  @   assumes  A2;
  @   requires R2;
  @    assigns L2;
  @    ensures E2;
  @*/


/*@
 @ complete behaviors b1,...,bn;

@ complete behaviors;
```

- La fonction appelante doit garantir que $P \wedge (A1 \Rightarrow R1) \wedge (A2 \Rightarrow R2)$ est vraie à l'appel.
- La fonction appelante renvoie un état satisfaisant le prédicat $\backslash old(A1) \Rightarrow E1$ et $\backslash old(A2) \Rightarrow E2$
- Les variables qui ne figurent pas dans l'ensemble $L1 \cup \ldots \cup Lp$ ne sont pas modifiées.

Programmation par contrat (analyse des conditions)

(contrat3.c)

Listing 35 – project-divers/contrat3.c

```
/*@ behavior  change_p:
  @   assumes n > 0;
  @   requires \valid(p);
  @   assigns *p;
  @   ensures *p == n;
  @ behavior change_q:
  @   assumes n <= 0;
  @   requires \valid(q);
  @   assigns *q;
  @   ensures *q == n;
  @*/
void f(int n, int *p, int *q) {
  if (n > 0) *p = n; else *q = n;
}
```

Programmation par contrat (squareroot)

(quareroot)

Listing 36 – project-divers/contrat1.c

```
/*@ requires x >= 0;
@ ensures \result >= 0;
@ ensures \result * \result <= x;
@ ensures x < (\result + 1) * (\result + 1); @*/
int squareroot(int x);
```

Programmation par contrat (increment)

(contrat2)

Listing 37 – project-divers/contrat2.c

```
/*@ requires \valid(p);
  @ assigns *p;
  @ ensures *p == \old(*p) + 1;
  @*/
void increment(int *p);
```

Programmation par contrat (f)

(contrat3)

Listing 38 – project-divers/contrat3.c

```
/*@ behavior  change_p:
  @   assumes n > 0;
  @   requires \valid(p);
  @   assigns *p;
  @   ensures *p == n;
  @ behavior change_q:
  @   assumes n <= 0;
  @   requires \valid(q);
  @   assigns *q;
  @   ensures *q == n;
  @*/
void f(int n, int *p, int *q) {
  if (n > 0) *p = n; else *q = n;
}
```

Current Summary

Current Subsection Summary

Sommaire des annotations et autres assertions

- requires
- assigns
- ensures
- decreases
- predicate
- logic
- lemma

()

Listing 39 – project-divers/predicate1.c

```
/*@ predicate is_positive(integer x) = x > 0; */

/*@ logic integer get_sign(real x) =  @ x>0.0?1:(x<0.0?-1:0);
*/
/*@ logic integer max(int x,int y) =   x>=y?x:y;
*/
```

()

Listing 40 – project-divers/lemma1.c

```
/*@ lemma div_mul_identity:
@ \forall real x, real y; y != 0.0 ==> y*(x/y) == x; @*/

/*@  lemma div_qr:
@ \forall int a, int b; a >= 0 && b >0 ==>
\exists  int q, int r; a == b*q +r && 0<=r && r <b; @*/
```

Predicates - Logic - Lemma (2)

(Définition de la fonction fibonacci)

Listing 41 – project-divers/predicate2.c

```
/*@ axiomatic mathfibonacci{
  @ logic integer mathfib(integer n);
  @ axiom mathfib0: mathfib(0) == 1;
  @ axiom mathfib1: mathfib(1) == 1;
  @ axiom mathfibrec: \forall integer n; n > 1
  ==> mathfib(n) ==  mathfib(n-1)+mathfib(n-2);
  @ } */
```

(Définition de la fonction factoriel)

Listing 42 – project-factorial/factorial.h

```
#ifndef _A_H
#define _A_H
/*@ axiomatic mathfact {
  @ logic integer mathfact(integer n);
  @ axiom mathfact_1: mathfact(1) == 1;
  @ axiom mathfact_rec: \forall integer n; n > 1
  ==> mathfact(n) == n * mathfact(n-1);
  @ } */

/*@ requires n > 0;
   decreases n;
   ensures \result == mathfact(n);
   assigns \nothing;
*/
int codefact(int n);
#endif
```

(Définition de la relation gc)

Listing 43 – project-divers/predicate3.c

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
@ case gcd_zero:
@ \forall integer n; is_gcd(n,0,n);
@ case gcd_succ:
@ \forall integer a,b,d; is_gcd(b, a % b, d) ==> is_gcd(a,b,d); @}
@*/
```

(Définition de la fonction pair/impair)

Listing 44 – project-divers/predicate4.c

```
//@ predicate pair(integer x) = (x/2)*2==x;
//@ predicate impair(integer x) = (x/2)*2!=x;
//@ lemma ex: \forall integer a,b; a < b ==> 2*a < 2*b;

/*@ inductive  is_gcd(integer a,integer b , integer c) {
  case zero:  \forall integer n; is_gcd(n,0,n);
   case un:  \forall integer u,v,w; u >= v ==>  is_gcd(u-v,v,w);
   case deux:  \forall integer u,v,w; u <  v ==>  is_gcd(u,v-u,w);
   }
*/
```

Current Subsection Summary

Variable de type ghost

- Une variable dite *ghost* permet de désigner de manière cachée ou masquée une valeur calculée et utile pour exprimer une propriété.
- Elle ne doit pas changer la sémantique des autres variables et on ne modifie pas le code dans les instructions ghost.

Erreur d'utilisation de ghost

(erreur)

Listing 45 – project-divers/ghost2.c

```
int f (int x, int y) {
  //@ghost int z=x+y;
switch (x) {
case 0: return y;
//@ ghost case 1: z=y;
// above statement is correct.
//@ ghost case 2: { z++; break; }
// invalid, would bypass the non-ghost default
default: y++; }
return y; }

int g(int x) { //@ ghost int z=x;
if (x>0){return x;}
//@ ghost else { z++; return x; }
// invalid, would bypass the non-ghost return
return x+1; }
```

(Variable ghost)

Listing 46 – project-divers/ghost1.c

```
/*@ requires a >= 0 && b >= 0;
  ensures 0 <= \result;
  ensures \result < b;
  ensures \exists integer k; a == k * b + \result; */
int rem(int a, int b) {
  int r = a;
 /*@ ghost   int q=0;    */
  /*@
    loop invariant
    a == q * b + r &&
    r >= 0 &&  r <= a;
    loop assigns r;
    loop assigns q;
// loop variant r;
*/
  while (r >= b) {
    r = r - b;
/*@ ghost     q = q+1;    */
  };
  return r;
}
```

Current Subsection Summary

Valeur initiale de x $\backslash old(x)$

- $\backslash old(x)$ désigne la valur de la variable x au moment de l'appel de la fonction.
- Cette expression est utilisable uniquement dans la postcondition *ensures*

(Valeur initiale x0)

Listing 47 – project-divers/old1.c

```
/*@ requires \valid(a) && \valid(b);
   @ assigns *a,*b;
    @ ensures   *a == \at(*a,Pre) +2;
    @ ensures  *b == \at(*b,Pre)+\at(*a,Pre)+2;

        @ ensures  \result == 0;
*/
int old(int *a, int *b) {
  int x,y;
  x = *a;
  y = *b;
  x=x+2;
   y = y +x;

  *a = x;
  *b = y;
  return 0 ;
}
```

Expression $\backslash at(e, id)$

- $\backslash at(e, id)$ désigne la valeur de e au point de contrôle id.
- id doit être rencontré avant $\backslash at(e, id)$
- id est une expression parmi Pre, Here, Old, Post, LoopEntry, LoopCurrent, Init
- $\backslash old(e)$ est équivalent à $\backslash at(e, Old)$

Exemple pour $\at(e, id)$

(label Pre)

Listing 48 – project-divers/at1.c

```
/*@
  requires \valid(a) && \valid(b);
  assigns *a,*b;
  ensures  *a == \old(*a)+2;
  ensures  *b == \old(*b)+\old(*a)+2;
*/
int at1(int *a, int *b) {
//@ assert *a == \at(*a,Pre);
  *a = *a +1;
//@ assert *a == \at(*a,Pre)+1;
  *a = *a +1;
//@ assert *a == \at(*a,Pre)+2;
  *b = *b +*a;
//@ assert *a == \at(*a,Pre)+2 && *b == \at(*b,Pre)+\at(*a,Pre)+2;
  return 0;
}
```

Exemple pour $\backslash at(e, id)$

(autre label)

Listing 49 – project-divers/at2.c

```
void f (int n) {
for (int i = 0; i < n; i++) {
/*@ assert \at(i,LoopEntry) == 0; */
int j=0;
while (j++ < i) {
/*@ assert \at(j,LoopEntry) == 0; */
/*@ assert \at(j,LoopCurrent) + 1 == j; */
} }
}
```

Current Summary

Commentaires

- Ce cours est une introduction et n'a pas vocation à être complet sur Frama-C et il est préférable de se reporter aux documents officiels sur le site `www.frama-c.org`.
- Frama-C permet d'énoncer les contrats (`requires`, `ensures`), d'annoter les codes séquentiels et de vérifier les annotations : programmation par contrat.
- La commande `frama-c` offre deux greffons `-wp` et `-rte` pour respectivement produire *les weakest-preconditions* et les conditions de débordement de mémoire.
- Les outils sont des procédures d'analyse de formules logiques de type SMT (Alt-Ergo) et des assistannt de preuve (Why3).