UNIVERSITÉ DE LORRAINE

TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

Cours Modélisation et Vérification des Systèmes Informatiques (MVSI)

Le langage de spécification pour Frama-C ACSL

Dominique Méry
Telecom Nancy, Université de Lorraine

Année universitaire 2024-2025

Plan

Sommaire

Current Summary

Exemple sans annotation

Listing 1 – slide0a.c

```
#include <limits.h>
void slide1(int a, int b, int c) {
    int x = a;
  int y = b;
    int z = c;
  //@ assert x + y== 4  && z  == x +2;
    z = x+ y + 2*z;
  //@ assert x + y== 4  && z+y == x+z;
     }
```

```
frama-c  slide0a.c -wp  -report
```

Exemple avec annotation

Listing 2 – slide0b.c

```
#include <limits.h>
/*@ requires a+b==4 && b+c==6 && c== 4 &&  a >= 0 && b >= 0 && b <= INT_MAX && a <= INT_MAX;
*/
void slide1(int a, int b, int c) {
    int x = a;
  int y = b;
    int z = c;
  //@ assert x + y== 4  && z  == x +2;
  z = x+ y + 2*z;
  //@ assert x + y== 4  && z+y == x+z;
     }
```

```
frama-c  slide0b.c -wp  -report
```

Exemple d'annotation

```
l1:x+y=3 /\ z=x+2;
z:=x+y+2*z;
l2:x+y=3 /\ z+y = x+z;
```

Exemple d'annotation

```
l1:x+y=3 /\ z=x+2;
z:=x+y+2*z;
l2:x+y=3 /\ z+y = x+z;
```

Listing 4 – slide1.c

```
#include <limits.h>
/*@
  requires  a >= 0 && b >= 0 && c>= 0 && a+b == 3 && c == a + 2;
  assigns \nothing;
*/
void slide1(int a, int b, int c) {
    int x = a;
  int y = b;
    int z = c;
  //@ assert x + y== 3  && z  == x +2;
  z = x+ y + 2*z;
  //@ assert x + y== 3  && z+y == x+z;
     }
```

Current Summary

Propriétés mathématiques des programmes

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post
- Propriétés de sûreté :

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post
- Propriétés de sûreté :
  - accès à une adresse valide

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post
- Propriétés de sûreté :
  - accès à une adresse valide
  - pas d'overflow arithmétque

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post
- Propriétés de sûreté :
  - accès à une adresse valide
  - pas d'overflow arithmétque
  - pas de division par zéro.
  - ...

Propriétés mathématiques des programmes

- Propriétés fonctionnelles : pré et post
- Propriétés de sûreté :
  - accès à une adresse valide
  - pas d'overflow arithmétque
  - pas de division par zéro.
  - ...
- Terminaison

Current Subsection Summary

Exemple 0 : compilation simple

Listing 5 – exemple0.c

```
// returns the absolute value of x
int abs ( int x ) {
if ( x >=0 ) return x ;
return -x ; }

// frama-c -wp file.c
```

Exemple 1 : définition du contrat

Listing 6 – exemple1.c

```
// returns the absolute value of x
/*@ ensures (x >= 0 ==> \result == x);
    ensures (x < 0 ==> \result == -x);
*/

int abs ( int x ) {
if ( x >=0 ) return x ;
return -x ; }
```

Exemple 1 : vérification du contrat

Listing 7 – exemple 1

```
macdome:lectures-malg mery$ frama-c  -wp exemple1.c
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no p
[kernel] Parsing exemple1.c (with preprocessing)
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] Proved goals:    2 / 2
     Qed:             2
macdome:lectures-malg mery$
```

Exemple 1 : vérification di contrat avec débordement

Listing 8 – exemple 1

```
macdome:lectures-malg mery$ frama-c  -wp -rte exemple1.c
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no p
[kernel] Parsing exemple1.c (with preprocessing)
[rte] annotating function abs
[wp] 3 goals scheduled
[wp] [Alt-Ergo] Goal typed_abs_assert_rte_signed_overflow : Unknown (58ms
[wp] Proved goals:    2 / 3
     Qed:             2
     Alt-Ergo:        0  (unknown: 1)
macdome:lectures-malg mery$
```

Exemple 2 : modification de la précondition

- Si x=INT_MIN, alors -x n'est pas codable.
- Ajouter une condition sur x.
- Ajouter la bibliothèque $\#\mathbf{include} < \text{limits.h} >$

Listing 9 – exemple 2

```
#include <limits.h>
/*@ requires x >INT_MIN;
    requires x <=INT_MAX;
    ensures (x >= 0 ==> \result == x);
    ensures (x < 0 ==> \result == -x);
*/

int abs ( int x ) {
if ( x >=0 ) return x ;
return -x ; }
```

Exemple 2

- Si x=INT_MIN, alors -x n'est pas codable.
- Ajouter une condition sur x.

Listing 10 – exemple 2

```
macdome:lectures-malg mery$ frama-c
-wp -rte exemple2.c
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_nor
[kernel] Parsing exemple2.c (with preprocessing)
[rte] annotating function abs
[wp] 3 goals scheduled
[wp] Proved goals:    3 / 3
     Qed:             3
macdome:lectures-malg mery$
```

Current Subsection Summary

Second exemple

Listing 11 – exemple 2

```
/*@ requires x >= 0 ;
  @*/
int f(int x){
  return x+1;
}
```

Second exemple : précondition à modifier pour le débordement

Listing 12 – exemple 2

```
frama-c -wp -wp-rte  -report -wp-print  frama9.c
```

```
------------------------------------------------------------------------
  Function f
------------------------------------------------------------------------

Goal Assertion 'rte,signed_overflow' (file frama9.c, line 4):
Assume { Type: is_sint32(x). (* Pre-condition *) Have: 0 <= x. }
Prove: x <= 2147483646.
Prover Alt-Ergo returns Unknown (59ms)

------------------------------------------------------------------------
```

Second exemple

Listing 13 – exemple 2

```
#include <limits.h>
/*@  requires x >= 0 ;
  @ requires x < INT_MAX ;
  @*/
int f(int x){
  return x+1;
}
```

Second exemple

Listing 14 – exemple 2

```
------------------------------------------------------------------------
--- Status Report Summary
------------------------------------------------------------------------

     1 Completely validated
     1 Total
------------------------------------------------------------------------
[rte] annotating function f
[wp] 1 goal scheduled
[wp] Proved goals:    1 / 1
     Qed:             1
------------------------------------------------------------------------
  Function f
------------------------------------------------------------------------

Goal Assertion 'rte,signed_overflow' (file frama9bis.c, line 6):
Prove: true.
Prover Qed returns Valid

------------------------------------------------------------------------
```

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

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- Pour toute paire d'étiquettes ℓ, ℓ' telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $x, x' \in \text{MEMORY}$

$$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$

- Pour toute paire d'étiquettes m, n telle que $m \longrightarrow n$, on vérifie que, $\forall x, x' \in \text{MEMORY} : pre(x_0) \wedge P_m(x_0, x) \Rightarrow \textbf{DOM}(m, n)(x)$

Conditions à vérifier

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

Exemple **DOM**(m,n)(x)

$DOM(\ell_0, \ell_1)(u) = u \in minint..maxint \wedge 5 \in minint..maxint \wedge u{+}5 \in minint..maxint$ où $\ell_0 : P_{\ell_0}(u) U := U{+}5; \ell_1 : P_{\ell_0}(u)$

Current Subsection Summary

Observations

Observations

- Vérifier les énoncés de la forme $\Gamma \vdash P$ (séquents)

Observations

- Vérifier les énoncés de la forme $\Gamma \vdash P$ (séquents)
- Enoncer ou calculer les invariants d'un modèle : $\text{REACHABLE}(M)$.

Observations

- Vérifier les énoncés de la forme $\Gamma \vdash P$ (séquents)
- Enoncer ou calculer les invariants d'un modèle : $\text{REACHABLE}(M)$.
- TLA$^+$ versus Event-B
  - Plate-formes : TLA$^+$ avec TLAPS et Toolbox, Event-B avec Rodin
  - Langage de la théorie des ensembles avec quelques différences
  - Fonctionnalités des outils
    - Editeurs de modèles : TLA$^+$ et Event-B
    - Model-Checking : TLA$^+$ et Event-B
    - Assistant de preuve : Event-B
    - Animateur et Model-Checker ProB

Observations

- Vérifier les énoncés de la forme $\Gamma \vdash P$ (séquents)
- Enoncer ou calculer les invariants d'un modèle : $\text{REACHABLE}(M)$.
- TLA$^+$ versus Event-B
  - Plate-formes : TLA$^+$ avec TLAPS et Toolbox, Event-B avec Rodin
  - Langage de la théorie des ensembles avec quelques différences
  - Fonctionnalités des outils
    - Editeurs de modèles : TLA$^+$ et Event-B
    - Model-Checking : TLA$^+$ et Event-B
    - Assistant de preuve : Event-B
    - Animateur et Model-Checker ProB
- Développement d'outils symboliques comme les solveurs SMT ou des procédures de décision

Revue des outils

- TLA$^+$ et TLA Toolbox : logique temporelle, théorie des ensembles, calcul des prédicats, model-checker
- Event-B et Rodin : théorie des ensembles, assistant de preuve, model-checker, animateur
- B et Event-B et ProB : théorie des ensembles, model-checker, animateur, validation
- Promela et SPIN : logique temporelle, model-checking
- C et Frama-C : analyse sémantique des programmes, assistants de preuve, solveurs SMT.
- Spec# et Rise4fun : pre/post, contrats
- PAT : cadre générique pour créer son propre model-checker (classique, temps réel, probabiliste, stochastique)
- C et `cppcheck` : analyse statique de programmes C ou C++

Point d'étape

- $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x)$
- $\forall x \in \textsc{Vals}. (\exists x_0. x_0 \in \textsc{Vals} \wedge Init(x_0) \wedge \textsc{Next}^\star(x_0, x)) \Rightarrow A(x).$
- $\textsc{reachable}(M) = \{u | u \in \textsc{Vals} \wedge (\exists x_0. x_0 \in \textsc{Vals} \wedge Init(x_0) \wedge \textsc{Next}^\star(x_0, u))\}$ est l'ensemble des états accessibles à partir des états initiaux.
- Model Checking : on doit montrer l'inclusion $\textsc{reachable}(M) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$.
- Preuves : définir un invariant $\mathrm{I}(\ell, v) \equiv \bigvee_{\ell \in \textsc{Locations}} \left(\bigvee_{v \in \textsc{Memory}} P_\ell(v) \right)$ avec la famille d'annotations $\{P_\ell(v) : \ell \in \textsc{Locations}\}$ et démontrer les conditions de vérification.
- Analyse automatique :
  - Mécaniser la vérification des conditions de vérification
  - Calculer $\textsc{reachable}(M)$
  - Calculer une valeur approchée de $\textsc{reachable}(M)$

  ¡ $(\mathcal{P}(\textsc{Vals}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D, \sqsubseteq)$
  $\alpha(\textsc{reachable}(M)) \sqsubseteq A$ ssi $\textsc{reachable}(M) \sqsubseteq \gamma(A)$
  Si $\gamma(A) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$, alors $\textsc{reachable}(M) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$

Analyse automatique

- Mécaniser la vérification des conditions de vérification
- Calculer $\textsc{reachable}(M)$ comme un point-fixe.
- Calculer une valeur approchée de $\textsc{reachable}(M)$

$$(\mathcal{P}(\textsc{Vals}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D, \sqsubseteq)$$
$$\alpha(\textsc{reachable}(M)) \sqsubseteq A \text{ ssi } \textsc{reachable}(M) \sqsubseteq \gamma(A)$$

Si A vérifie $\gamma(A) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$, alors
$\textsc{reachable}(M) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$

Environnement de développement Frama-C

- le langage de programmation est C.
- extension du langage de programmation par des annotations et la programmation par contrat.
- prise en charge du langage et de son annotation par des outils de preuve automatiques.

Current Subsection Summary

Le plugin WP

- Le plugin WP met en œuvre le calcul WP pour la correction partielle et la correction totale.
- Révision de la logique de Hoare avec le plugin qui suit les règles de composition du calcul WP et aussi des règles de Hoare.

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
$S_1;S_2$	$wp(S_1)(wp(S_2)(P))$
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	$\mu.(\lambda X.(B \Rightarrow wp(S)(X)) \wedge (\neg B \Rightarrow P))$

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

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 : $x_0 \xrightarrow{P} x_f$
- x_0 satisfait pre : $pre(x_0)$
- x_f satisfait post : $post(x_0, x_f)$
- $pre(x_0) \wedge x_0 \xrightarrow{P} x_f \Rightarrow post(x_0, x_f)$

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

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

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $P_f(x_0, x) \Rightarrow post(x_0, x)$
- conditions de vérification pour toutes les paires $\ell \longrightarrow \ell'$

Conditions à vérifier

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- Pour toute paire d'étiquettes ℓ, ℓ' telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $x, x' \in \text{MEMORY}$
$$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$
- Pour toute paire d'étiquettes m, n telle que $m \longrightarrow n$, on vérifie que, $\forall x, x' \in \text{MEMORY} : pre(x_0) \wedge P_m(x_0, x) \Rightarrow \mathbf{DOM}(m, n)(x)$
- Pour toute instruction S et pour toute assertion Q, $\{WP(S)(Q)\}S\{Q\}$ est vérifiée par définition de WP(S)(Q).
- Pour toute instruction S et pour toutes assertions P et Q, $\{P\}S\{Q\}$ est équivalent à $P \Rightarrow WP(S)(Q)$ vérifiée par définition de la sémantique de S.
- $\text{pre}(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f)$ est équivalent à $\text{pre}(x_0) \wedge x = x_0 \Rightarrow WP(P)(\text{post}(x_0, x))$
- $\{\text{pre}(x_0) \wedge x = x_0\}P\{\text{post}(x_0, x)\}$

Conditions de vérification

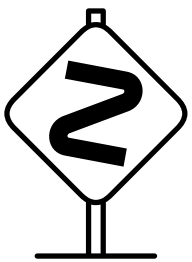
- Pour toute instruction S et pour toute assertion Q, $\{WP(S)(Q)\}S\{Q\}$ est vérifiée par définition de WP(S)(Q).
- Pour toute instruction S et pour toutes assertions P et Q, $\{P\}S\{Q\}$ est équivalent à $P \Rightarrow WP(S)(Q)$ vérifiée par définition de la sémantique de S.
- $\text{pre}(x_0) \wedge x_0 \xrightarrow{\text{P}} x_f \Rightarrow \text{post}(x_0, x_f)$
  est équivalent à
  $\text{pre}(x_0) \wedge x = x_0 \Rightarrow WP(P)(\text{post}(x_0, x))$
- $\{\text{pre}(x_0) \wedge x = x_0\}P\{\text{post}(x_0, x)\}$

Listing 15 – slide5.c

```
// #include <limits.h>

int slide5(int a) {
    int i = a;
  //@ assert i == a;
      i++;
  //@ assert i == a+1;
      return i;
}
```

1. $i == a \Rightarrow (i == a)$ (initialisation)
2. $i == a \Rightarrow (i == a+1)[i \mapsto i+1]$ (assignment)



% frama-c slide5.c -wp -report

Listing 16 – slide4.c

```
#include <limits.h>
/*@
   requires  a >= 0;
   requires  (a+1)*(a+1) <= INT_MAX;
   assigns \nothing;
   ensures \result == (a+1)*(a+1);
*/
int slide4(int a) {
     int i = a;
  //@ assert i == a;
     i++;
  //@ assert i == a+1;
     i=i+a;
  //@ assert i == 2*a+1;
     i=i+a*a;
  //@ assert i ==  (a+1)*(a+1);
     return i;
}
```

Listing 17 – slide2.c

```
/*@
   requires  a >= 0  && a < 30;
   ensures \result == 30;
*/
int slide2(int a) {
     int i = a;
   //@ assert i == a;
     /*@ loop invariant 0<= i && i <=30;
         loop assigns i;
       */

     while (i<30) {
   //@ assert 0 <= i < 30;
       i++;
   //@ assert 0 <= i <= 30;
     };
   //@ assert i == 30;
      return i;
}
```

Current Subsection Summary

Axiomatisation de la Logique de Hoare

⊠ 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** $\mathbf{S}_1$ **then** $\mathbf{S}_2$ **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 $\{P'\}\mathbf{S}\{Q'\}$.

Axiomatisation : la Logique de Hoare

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

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

Introduction de wlp

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

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 $wlp(S)(P) \Rightarrow wlp(S)(Q)$.

Axiomatisation de la Logique de Hoare

⊠ 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 $\{P\}$**if B then S$_1$ then S$_2$ fi**$\{Q\}$.
- 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.

Logique de Hoare Correction Totale

⊠ 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 $[P]$**if B then S$_1$ then S$_2$ fi**$[Q]$.
- 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 $[P]\mathbf{S}[Q]$ 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 $[P]\mathbf{S}[Q]$ construites avec les règles ci-dessus,

- $P \Rightarrow wp(X := E(X))(Q) : P \Rightarrow Q(e/x)$ et $[Q(e/x)]\mathbf{X := E(X)}[Q]$ 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.

- Revoir les wp
- Faire le calcul pour l'affectation et la conditionnelle

Current Summary

Current Subsection Summary

Exemple de ACSL/Frama-c

Listing 18 – swap de deux contenus

```
/*@
     requires \valid(a) && \valid(b);
     assigns *a, *b;
     ensures *a == \old(*b);
     ensures *b == \old(*a);
*/
static void swap(int* a, int* b) {
     int temp;
     temp = (*a);
     (*a) = (*b);
     (*b) = temp;
}
```

Exemple de ACSL/Frama-c

Listing 19 – difference de deux nombres

```
/*@
     assigns \result;
     ensures \result == (a - b);
*/
static int difference(int a, int b) {
     return a-b;
}
```

Exemple de ACSL/Frama-c

Listing 20 – prepostframa3.cl

```
int main(int argc, char* argv[]) {
    int a = 21;
    int b = 42;

    swap(&a, &b);
    //@ assert a == 42 && b == 21;

    int c = difference(b, a);
    //@ assert c == 22;

    return 0;
}
```

Current Subsection Summary

Assertions

- Assertions à un point du programme :

  ```
  //*@  assert pred; */
  ```

- Assertions à un point du programme selon les comportements.

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

Exemple d'assertions

Listing 21 – 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;
  }
```

Listing 22 – 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;
  }
```

Invariant de boucle

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

Listing 23 – 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

Listing 24 – 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.

Listing 25 – 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.
```

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`)

Listing 26 – 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 :
  $\forall x_1, x_2.b(x_1) \wedge x_1 \xrightarrow{S} x_2 \Rightarrow \text{expvariant}(x_1) > \text{expvariant}(x_2)$

Listing 27 – variant1.c

```
/*@ requires 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

Listing 28 – 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

Listing 29 – variant4.c

```
/*@ 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

Listing 30 – 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);
 }
```

Listing 31 – wp3.c

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

Listing 32 – 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);
}
```

Listing 33 – 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);
}
```

Listing 34 – 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);
}
```

Listing 35 – 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;
}
```

Listing 36 – wp8.c

```
int main()
{
  int x = -1;
  int *p;
//@assert  x == -1;
p = &x ;
x = x+1;
//@assert  x >= 0 && *p >= 0  ;
  return(0);
}
```

Current Summary

- Revoir la notion de contrat
- Revoir la notion d'invariant de boucle.
- Gestion des labels
- notion de théorie

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

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- Pour toute paire d'étiquettes ℓ, ℓ' telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $x, x' \in \text{MEMORY}$

$$\left(\begin{array}{l} \left(\begin{array}{l} pre(x_0) \wedge P_\ell(x_0, x)) \\ \wedge cond_{\ell,\ell'}(x) \wedge x' = f_{\ell,\ell'}(x) \end{array} \right) \\ \Rightarrow P_{\ell'}(x_0, x') \end{array} \right),$$

- Pour toute paire d'étiquettes m, n telle que $m \longrightarrow n$, on vérifie que, $\forall x, x' \in \text{MEMORY} : pre(x_0) \wedge P_m(x_0, x) \Rightarrow \textbf{DOM}(m, n)(x)$

Définition du contrat et des axiomes

Listing 37 – factoriel

```
/*@ 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;
   ensures \result == mathfact(n);
*/
int codefact(int n) {
    //@ assert n >= 1 && n <= n &&   mathfact(n) ==  1* mathfact(n);
  int y = 1;
 //@ assert n >= 1 && n <= n &&   mathfact(n) == y * mathfact(n);
  int x = n;
  //@ assert x >= 1 && x <= n &&   mathfact(n) == y * mathfact(x);

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

Définition du contrat et des axiomes

```
/*@ 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);
  @ } */
```

- La spécification d'une fonction à calculer nécessite de la définir mathématiquement.
- Cette définition axiomatique est fondée sur une définition inductive de la fonction mfact qui sera utilisée dans les assertions pour le contrat définissant la fonction informatique de calcul.

```
/*@ requires n > 0;
  ensures \result == mathfact(n);
*/
int codefact(int n) {
  int y = 1;
  int x = n;
  /*@ loop invariant x >= 1 &&
                     mathfact(n) == y * mathfact(x);
    loop assigns x, y;
   */
  while (x != 1) {
    y = y * x;
    x = x - 1;
  };
  return y;
}
```

- La fonction appelante doit garantir que $P1 \wedge \ldots \wedge Pn$ est vraie à l'appel.
- La fonction appelante renvoie un résultat satisfaisant le prédicat $E1 \wedge \ldots \wedge Em$
- Les variables évaluées aux pré et post états qui ne figurent pas dans l'ensemble $L1 \cup \ldots \cup Lp$ ne sont pas modifiées.

Listing 38 – schema de contrat

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

Listing 39 – 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.
- $\backslash result$ fait référence à la valeur du résultat de l'appel.

Current Subsection Summary

- Précondition $x \geq 0$
- Postcondition $result \cdot result \leq x < (result+1) \cdot (result+1)$

Listing 40 – contrat squareroot

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

- Précondition $def(p)$
- Postcondition $\cdot p = \underline{\cdot p}+1$

Listing 41 – contrat increment

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

Listing 42 – fschema3.c

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

- 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.

Listing 43 – 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

Listing 44 – 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

Listing 45 – contrat2.c

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

Programmation par contrat f

Listing 46 – 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

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

Listing 47 – 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; }
```

Listing 48 – 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;
   */
  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 dans la postcondition *ensures*

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

Listing 49 – old1.c

```
/*@
  requires a >= 0 && b >= 0;
  ensures  a == \old(a)+2;
  ensures  b == \old(b)+\old(a)+2;
*/
int old(int a, int b) {
  a = a +1;
  a = a +1;
  b = b +a;
  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 $\backslash at(e, id)$

Listing 50 – prevaleur

```
/*@
   requires a >= 0 && b >= 0;
assigns \nothing;
   ensures  \result == \old(a)+2;
*/
int at(int a, int b) {
   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;
   return a ;
}
```

–

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

Listing 51 – prevaleur

```
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).
- `frama-c -wp -rte -wp-model Hoare -wp-out <dir> <file>.c`