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

Current Subsection Summary

Exemple de ACSL/Frama-c

Listing 1 – 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 2 – 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 3 – 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; */

/@  assert pred;
```

- Assertions à un point du programme selon les comportements.

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

Exemple d'assertions

Listing 4 – 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 5 – 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 6 – 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 7 – 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 8 – 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 9 – 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 10 – 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 11 – 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 12 – 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 13 – 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 14 – wp3.c

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

Listing 15 – 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 16 – 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 17 – 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 18 – 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 19 – wp8.c

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

- 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 20 – 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 21 – schema de contrat

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

Listing 22 – 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 23 – 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 24 – contrat increment

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

Listing 25 – 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 26 – 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 27 – 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 28 – contrat2.c

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

Programmation par contrat f

Listing 29 – 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 30 – 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 31 – 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 32 – 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 33 – 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 34 – 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`