FRAMA-C: INTRODUCTION

Dr.S.Padmavathi
CSE, Amrita School of Engineering
Coimbatore

Frama-C

Frama-C = Framework for Modular Analysis of C Programs

It is a collection of plug-ins that perform a variety of static program analyses, e.g., program slicing, impact analysis, ...

One such plug-in is WP, for weakest preconditions. In order to enhance automated verification, Frama-C can make use of a variety of theorem provers:

Alt-Ergo, QED, Coq, Why3 ...

Easily installs under Linux: sudo apt-get install frama-c

Frama-C uses ACSL

ACSL = ANSI/ISO C Specification Language specifies:

(i) function contracts (requires/ensures)
(ii) function behaviors (assumes/ensures)
(iii) axioms for user-defined functions
(iv) loop invariants for partial correctness
(v) loop variants for termination
(vi) assertions at points in program
(vii) validity of pointers
(viii) changes, if any, to global memory

...

operators

>=	$\geq$
<=	$\leq$
>	$>$
<	$<$
!=	$\not\equiv$
==	$\equiv$
==>	$\Longrightarrow$
<==>	$\Longleftrightarrow$
&&	$\wedge$
||	$\vee$
^^	$\oplus$
!	$\neg$
\forall	$\forall$
\exists	$\exists$

- The at sign (@) is a blank character, thus equivalent to a space character.
- Identifiers may start with the backslash character (\).

Let us give a quick reminder about all truth tables of usual logic connectors in first order logic ($\neg$ = `!`, $\wedge$ = `&&`, $\vee$ = `||`) :

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

ACSL syntax	Name	Reading
`!P`	negation	P is not true
`P && Q`	conjunction	P is true and Q is true
`P || Q`	disjunction	P is true or Q is true
`P ==> Q`	implication	if P is true, then Q is true
`P <==> Q`	equivalence	if, and only if, P is true, then Q is true
`x < y == z`	relation chain	x is less than y and y is equal to z
`\forall int x; P(x)`	universal quantifier	P(x) is true for every **int** value of x
`\exists int x; P(x)`	existential quantifier	P(x) is true for some **int** value of x

Types

- “mathematical” types:
  - integer for unbounded, mathematical integers,
  - real for real numbers,
  - boolean for booleans (with values written \true and \false);
- There are implicit coercions for numeric types:
  - C integral types char, short, int and long, signed or unsigned, are all subtypes of type integer;
  - integer is itself a subtype of type real;
  - C types float and double are subtypes of type real.

Casts and overflows

- (unsigned char)1000 is 1000 mod 256 i.e. 232.
- //@ **logic** int f(int x) = x+1 ;
- is not allowed because x+1, which is a mathematical integer, must be casted to int. One should write
- either
- //@ **logic** integer f(int x) = x+1 ;
- or
- //@ **logic** int f(int x) = (int)(x+1) ;

Assignment rule

- //@ **assert** y+1 > 0 && a[2*(y+1)] == 0;
- x = y+1;
- //@ **assert** x > 0 && a[2*x] == 0;

- //@ **assert** y > 0;
- x = y+1;
- //@ **assert** y > 0 && x == y+1;

Wrong specification

- //@ **assert true**;
- x = x+1;
- //@ **assert** x == x+1;

Wrong specification

Choice rule

- //@ **assert** 0 <= i < n; // given precondition
- **if** (i < n-1) {
- //@ **assert** 0 <= i < n - 1; // using that i < n-1 holds in this branch
- //@ **assert** 1 <= i+1 < n; // by the implication rule
- i = i+1;
- //@ **assert** 1 <= i < n; // by the assignment rule
- //@ **assert** 0 <= i < n; // weakened by the implication rule
- } **else** {
- //@ **assert** 0 <= i == n-1 < n; // using that !(i < n-1) holds in else part
- //@ **assert** 0 == 0 && 0 < n; // weakened by the implication rule
- i = 0;
- //@ **assert** i == 0 && 0 < n; // by the assignment rule
- //@ **assert** 0 <= i < n; // weakened by the implication rule
- }
- //@ **assert** 0 <= i < n; // by the choice rule from both branches

Function contracts

- Built-in constructs **\old** and **\result**
- \old(e) denotes the value of e in the pre-state of the function.
- \result denotes the returned value of the function.
- \old(e) and \result can be used only in assigns clauses and ensures clauses, since requires and assumes clauses refer to the pre-state.

Contracts

- Goal: specification of imperative functions
- Approach: give assertions (i.e. properties) about the functions
  - Precondition is supposed to be true on entry (ensured by callers of the function)
  - Post condition must be true on exit (ensured by the function if it terminates)
- Nothing is guaranteed when the precondition is not satisfied
- Termination may or may not be guaranteed (total or partial correctness)
- Primary role of contracts
  - Main input of the verification process
  - Must reflect the informal specification
  - Should not be modified just to suit the verification tasks

simple function contract

- A simple function contract, having no named behavior, has the following form:
- /*@ requires P1; requires P2; ...
- @ assigns L1; assigns L2; ...
- @ ensures E1; ensures E2; ...
- @*/

```
/*@ requires P1 && P2 && ...;
@ assigns L1;L2; ...;
@ ensures E1 && E2 && ...;
@*/
```

- The semantics of such a contract is as follows:
  - The caller of the function must guarantee that it is called in a state where the property P1&&P2&& ... holds.
  - The called function returns a state where the property E1&&E2&& ...holds.
  - All memory locations of the pre-state that do not belong to the set L1 U L2 U ....remain allocated and are left unchanged in the post-state. The set L1 U L2 U ... is interpreted in the post-state.

Example: specification sqrt

- If no clause requires is given, it defaults to \true, and similarly for ensures clause.
- Giving no assigns clause means that locations assigned by the function are not specified, so the caller has no information at all on this function's side effects.

- The following function is given a simple contract for computation of the integer square root.
- /*@ **requires** x ≥0;
- @ **ensures \result** ≥ 0;
- @ **ensures \result** * **\result** ≤ x;
- @ **ensures** x < (**\result** + 1) * (**\result** + 1);
- @*/
- int isqrt(int x);
- The contract means that the function must be called with a nonnegative argument, and returns a value satisfying the conjunction of the three ensures clauses.

Example

Specify and prove the following program:

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

Example

Specify and prove the following program:

// returns the absolute value of x

- /*@
- ensures \result >= 0;
- */

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

Specification

Explain the proof failure for the following program:

```
/*@ ensures (x >= 0 ==> \result == x) &&
(x < 0 ==> \result == -x);
*/
int abs ( int x ) {
        if ( x >=0 )
                return x ;
        return -x ;
}
```

For x=INT_MIN, -x cannot be represented by an int and overflows
Example: on 32-bit, INT_MIN= -2^{31} while INT_MAX= $2^{31} - 1$

Safety warnings: arithmetic overflows

- Absence of arithmetic overflows can be important to check
- A sad example: crash of Ariane 5 in 1996
- automatically generates VCs to check absence of overflows
- They ensure that arithmetic operations do not overflow
- If not proved, an overflow may occur. Is it intended?

Corrected Specification

This is the completely specified program:

```
# include < limits .h>
/*@ requires x > INT_MIN ;
        ensures (x >= 0 ==> \result == x) &&
        (x < 0 ==> \result == -x);
        assigns \nothing ;
*/
int abs ( int x ) {
        if ( x >=0 )
                return x ;
        return -x ;
}
```

Motivation for RTE Guard

```
File  Edit  View  Search  Terminal  Help
// Version 1: Basic (but incomplete) specification

#include <limits.h>

/*@
  @ ensures x >= 0 ==> \result == x;
  @ ensures x < 0 ==> \result == -x;
*/

int abs(int x) {
  if (x < 0)
        return -x;
  else
        return x;
}

int main() {
abs(10);
abs(-20);
abs(INT_MIN);
}
~
```

Question: What is the potential probem with this simple program for returning the absolute value of a number?

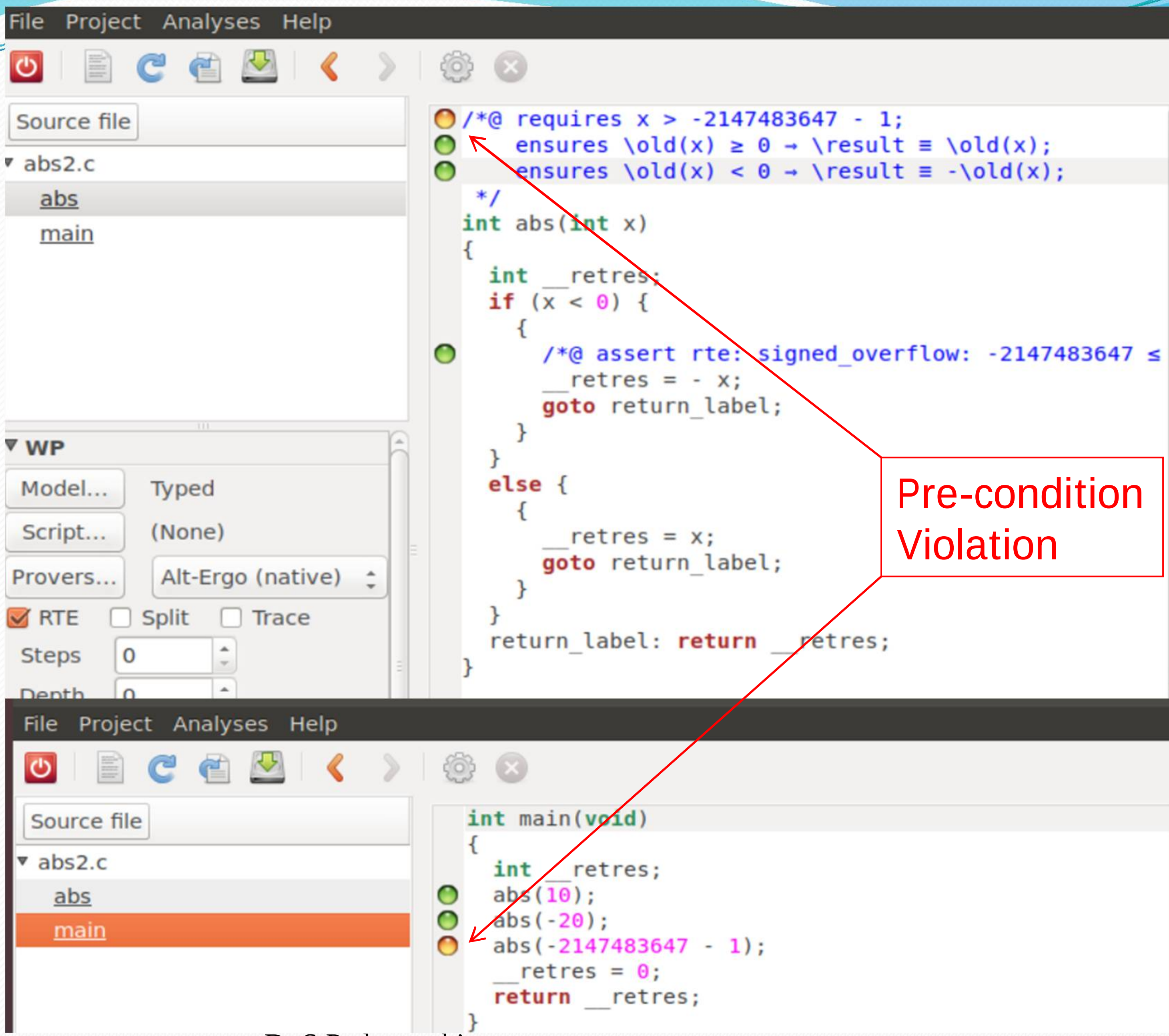
HINT

Let's Demo with Frama-C-GUI

Better Specification

```
#include <limits.h>

/*@ requires x > INT_MIN;
  @ ensures x >= 0 ==> \result == x;
  @ ensures x < 0 ==> \result == -x;
*/

int abs(int x) {
  if (x < 0)
        return -x;
  else
        return x;
}

int main() {

abs(10);
abs(-20);
abs(INT_MIN);

}
```

This call will get flagged as an error.


File Project Analyses Help
Source file
abs2.c
abs
main
/*@ requires x > -2147483647 - 1;
ensures \old(x) ≥ 0 → \result ≡ \old(x);
ensures \old(x) < 0 → \result ≡ -\old(x);
*/
int abs(int x)
{
int __retres;
if (x < 0) {
{
/*@ assert rte: signed_overflow: -2147483647 ≤
__retres = - x;
goto return_label;
}
}
else {
{
__retres = x;
goto return_label;
}
}
return_label: return __retres;
}
WP
Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
RTE Split Trace
Steps 0
Pre-condition Violation
File Project Analyses Help
Source file
abs2.c
abs
main
int main(void)
{
int __retres;
abs(10);
abs(-20);
abs(-2147483647 - 1);
__retres = 0;
return __retres;
}

Behaviors

- Specification by cases
  - Global precondition (requires) applies to all cases
  - Global post condition (ensures, assigns) applies to all cases
  - Behaviors define contracts (refine global contract) in particular cases
  - For each case (each behavior)
    - the subdomain is defined by assumes clause
    - the behavior's precondition is defined by requires clauses
      - it is supposed to be true whenever assumes condition is true
  - the behavior's post condition is defined by ensures, assigns clauses
    - it must be ensured whenever assumes condition is true
  - complete behaviors states that given behaviors cover all cases
  - disjoint behaviors states that given behaviors do not overlap

Example behavior

- Specify using behaviors and prove the function abs:
- // returns the absolute value of x
- int abs (int x) {
- if (x >=0)
  - return x ;
- return -x ;
- }

Explain the proof failure

- #include<limits . h>
- /*@ requires x > INT MIN ;
- assigns \nothing ;
- behavior pos :
  - assumes x > 0 ;
  - ensures \r e s u l t == x ;
- behavior neg :
  - assumes x < 0 ;
  - ensures \r e s u l t == -x ;
- complete behaviors ;
- disjoint behav ior s ;
- */
- int abs (int x) {
- i f (x >=0)
  - r e turn x ;
- r e turn -x ;
- }

The behaviors are not complete

The case x==0 is missing. A wrong value could be returned.

Explain the proof failure

- #include<limits . h>
- /*@ requires x > INT MIN ;
- assigns \nothing ;
- behavior pos :
  - assumes x >= 0 ;
  - ensures \r e s u l t == x ;
- behavior neg :
  - assumes x <= 0 ;
  - ensures \r e s u l t == -x ;
- complete behaviors ;
- disjoint behav ior s ;
- */
- int abs (int x) {
- i f (x >=0)
  - r e turn x ;
- r e turn -x ;
- }

The behaviors are not disjoint
The case x==0 is covered by both behaviors. Is it intended?

Correct specification

- #include<limits . h>
- /*@ requires x > INT MIN ;
- assigns \nothing ;
- behavior pos :
  - assumes x >= 0 ;
  - ensures \r e s u l t == x ;
- behavior neg :
  - assumes x < 0 ;
  - ensures \r e s u l t == -x ;
- complete behaviors ;
- disjoint behav ior s ;
- */
- int abs (int x) {
- i f (x >=0)
  - r e turn x ;
- r e turn -x ;
- }

Use of Behaviors

```
/*@ requires x > INT_MIN ;
    assigns \nothing ;

behavior non_negative:
        assumes x >= 0 ;
        ensures \result == x ;

behavior negative:
        assumes x < 0 ;
        ensures \result == -x ;

complete behaviors ;
disjoint behaviors ;

*/
```

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

Use of Behaviors enhances readability.

P ==> Q is given as:

assumes P ;

ensures Q ;

Need to specify complete/ disjoint behaviors.

Specification is thrice as long as Program! But this program is vulnerable to unusual run-time error, hence a bit more work.

Contracts with named behaviors

- /*@ requires P;
- @ behavior b1:
- @ assumes A1;
- @ requires R1;
- @ assigns L1;
- @ ensures E1;
- @ behavior b2:
- @ assumes A2;
- @ requires R2;
- @ assigns L2;
- @ ensures E2;
- @*/

- /*@ requires P&&(A1==>R1)&&(A2==>R2);
- @ behavior b1:
- @ assumes A1;
- @ assigns L1;
- @ ensures E1;
- @ behavior b2:
- @ assumes A2;
- @ assigns L2;
- @ ensures E2;
- @*/

The caller of the function must guarantee that the call is performed in a state where the property P&&(A1==>R1)&&(A2==>R2) holds.

The called function returns a state where the properties \old(Ai)==>Ei hold for each i.

For each i, if the function is called in a pre-state where Ai holds, then each memory location of that pre-state that does not belong to the set Li remains allocated and is left unchanged in the post-state.

simple contract

simple contract

- /*@ requires P;
- assigns L;
- ensures E; */

equivalent to a single named behavior

- /*@ requires P;
- @ behavior <any name>:
- @ assumes \true;
- @ assigns L;
- @ ensures E;
- @*/

Global assigns and ensures

global assigns and ensures

- global assigns and ensures clauses are equivalent to a single named behavior
- /*@ requires P;
- @ assigns L;
- @ ensures E;
- @ behavior b1: ...
- @ behavior b2: ...
- @
- ...
- @*/

is equivalent to

- /*@ requires P;
- @ behavior <any name>:
- @ assumes \true;
- @ assigns L;
- @ ensures E;
- @ behavior b1: ...
- @ behavior b2: ...
- @
- ...
- @*/