Safety Verification with Frama-C / WP

Frama-C

A platform for analysis of ANSI C code.

- Plug-in architecture
- command-line / GUI
- Multiple plug-ins implement many different analyses
- A specification language is defined to be used by the different plug-ins: the ANSI-C Specification Language (ACSL)

We will be looking at the following plug-ins:

- **RTE** ("RunTime Errors"): generates annotations in the code to guard against runtime errors
- **WP** ("Weakest Precondition"): deductive verification
- **E-ACSL**: runtime/dynamic verification

RTE Plug-in

Automatically annotates a C program with assertions preventing runtime errors:

- *numeric operations / overflow.* These are *silent errors* (program does not abort immediately but produces incorrect results)

- *memory accesses:* invalid pointer dereferencing / array indexes out-of-bounds. These cause immediate abrupt termination ("segmentation fault")

Although these are different categories of errors, with different observable effects, both correspond to *undefined behaviors.*

Example

Partition function (quicksort algorithm)

```
// file partition_init.c
int partition (int A[], int p, int r) {
  int x = A[r];
  int tmp, j, i = p-1;

  for(j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
    }
  tmp = A[i+1];
  A[i+1] = A[r];
  A[r] = tmp;
  return i+1;
}
```

Inserting Annotations Automatically

Runtime safety assertions can be inserted by running:

```
frama-c -rte partition_init.c -print
```

or, to output to a file:

```
frama-c -rte partition_init.c -ocode partition_rte.c
```

The result contains annotations written as special comments — invisible to C compilers but not to Frama-C.

All the annotations use the `@assert` keyword, which takes as argument a condition that must be true whenever the execution goes through that point of the program.

Note-se ainda que o código anotado foi "desugared", em particular todos os acessos a *arrays* foram convertidos em operações envolvendo *dereferencing* de apontadores.

```
/* Generated by Frama-C */
int partition(int *A, int p, int r)
{
  int __retres;
  int tmp;
  int j;
  /*@ assert rte: mem_access: \valid_read(A + r); */
  int x = *(A + r);
  /*@ assert rte: signed_overflow: -2147483648 ≤ p - 1; */
  int i = p - 1;
  j = p;
  while (j < r) {
    /*@ assert rte: mem_access: \valid_read(A + j); */
    if (*(A + j) <= x) {
      /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
      i ++;
      /*@ assert rte: mem_access: \valid_read(A + i); */
      tmp = *(A + i);
      /*@ assert rte: mem_access: \valid(A + i); */
      /*@ assert rte: mem_access: \valid_read(A + j); */
      *(A + i) = *(A + j);
      /*@ assert rte: mem_access: \valid(A + j); */
      *(A + j) = tmp;
    }
    /*@ assert rte: signed_overflow: j + 1 ≤ 2147483647; */
    j ++;
  }
  /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
  /*@ assert rte: mem_access: \valid_read(A + (int)(i + 1));
*/
  tmp = *(A + (i + 1));
  /*@ assert rte: mem_access: \valid(A + (int)(i + 1)); */
  /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
  /*@ assert rte: mem_access: \valid_read(A + r); */
  *(A + (i + 1)) = *(A + r);
  /*@ assert rte: mem_access: \valid(A + r); */
  *(A + r) = tmp;
  /*@ assert rte: signed_overflow: i + 1 ≤ 2147483647; */
  __retres = i + 1;
  return __retres;
}
```

The conditions include arithmetic formulas like `i + 1 ≤ 2147483647`, but also make use of the following ACSL operators, requiring proper allocation of a memory block:

- `\valid_read`
- `\valid`

The conditions also include a label, for instance in
`@ assert rte: mem_access: \valid_read(A + j)`

`rte: mem_access` means that this condition pertains to memory accesses and was automatically generated by the RTE plug_in.

Verifying the Annotated Program

It suffices to invoke the WP plug-in on the program generated by the RTE plug-in:

```
frama-c  -wp partition_rte.c
```

This produces the following output:

```
[wp] 16 goals scheduled
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_signed_overflow_2 : Timeout (Qed:4ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access_2 : Timeout (Qed:4ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_signed_overflow : Timeout (Qed:2ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access : Timeout (Qed:2ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access_4 : Timeout (Qed:9ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access_3 : Timeout (Qed:9ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access_7 : Timeout (Qed:5ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_signed_overflow_4 : Timeout (Qed:4ms) (10s)
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_access_8 : Timeout (Qed:5ms) (10s)
```

```
[wp] [Alt-Ergo 2.4.0] Goal typed_partition_assert_rte_mem_acce
ss_9 : Timeout (Qed:5ms) (10s)
[wp] [Cache] updated:13
[wp] Proved goals:    6 / 16
  Qed:               3  (2ms-7ms-18ms)
  Alt-Ergo 2.4.0:    3  (17ms-20ms) (75) (interrupted: 10)
```

16 verification conditions were generated (as many as the @assert annotations), of which only 6 were successfully proved (3 by Qed and 3 by Alt-Ergo). For the remaining VCs, listed above, the provers timed-out.

Notes:

- Qed is the internal simplifier, which is able to prove simple verification conditions
- Alt-Ergo is the default prover (SMT solver) included with the Frama-C distribution
- It is possible to install many other external proof tools through the Why3 platform

It is possible to run WP directly on the initial file, invoking simultaneously the RTE plug-in, without having to save the annotated program (in principle it is not even necessary to observe it):

```
frama-c  -wp -rte  partition_init.c
```

Adding Preconditions

Array allocation cannot be guaranteed — it must be assumed when the function is called. The adequate way to handle this is to include a precondition, written in ACSL just before the function, as follows.

```
/*@ requires 0 <= p <= r && \valid(A+(p..r));
  @*/
int partition (int A[], int p, int r)
```

```
{
    ...
}
```

The ACSL syntax is based on C expression syntax. The above conditions place restrictions on the values of indices `p` and `r` , in addition to requiring the array to be allocated between those indices.

The precondition allows for two more VCs to be proved:

```
frama-c  -wp -rte  partition_init.c
(...)
[wp] Proved goals:    8 / 16
  Qed:               3  (2ms-6ms-20ms)
  Alt-Ergo 2.4.0:    5  (17ms-20ms-27ms) (84) (interrupted: 8)
```

By observing the program it is obvious that, given the precondition, no errors may occur:

- no overflows occur when incrementing the variables `i j` , since their values are both bounded by `r`; and
- all array accesses are within the valid range.

However, some VCs could still not be proved, because no assumptions about the values taken by the variables `i j` can be made automatically inside the loop: the tool needs help from the user, in the form of a loop invariant:

```
/*@ loop invariant p <= j <= r && p-1 <= i < j;
  @*/
  for(j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
    }
```

Rerunning the verifier now results in the following:

```
[wp] 18 goals scheduled
[wp] Proved goals:   12 / 18
  Qed:               4   (2ms-8ms-26ms)
  Alt-Ergo 2.4.0:    8   (17ms-20ms-29ms) (89) (interrupted: 6)
(cached: 2)
```

Note that the total number of VCs increased from 16 to 18, because when an invariant is included as an annotation, it is necessary to prove that it is properly **initialized** (i.e. true when the loop is reached for the first time) and **preserved** by arbitrary iterations. By an inductive argument, the invariant can then be placed in the context to prove subsequent conditions (in this case, the postcondition).

Still, some VCs could not be proved. The reason for this is that it cannot be assumed that the array remains valid (i.e. properly allocated) between positions `p` and `r` throughout execution of the loop (see for instance the instruction commented out below). So we add a second invariant as follows:

```
/*@ loop invariant p <= j <= r && p-1 <= i < j;
  @ loop invariant \valid(A+(p..r));
  @*/
  for(j=p; j<r; j++)
    /* A++; */
    if (A[j] <= x) {
      i++;
      tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
    }
```

Frame Conditions

Lists of variables and memory positions that are assigned by the execution of a function —
known as frame conditions — are supported in ACSL contracts.

These conditions may be of great importance when verifying code dealing with mutable data structures.

```
/*@ requires 0 <= p <= r && \valid(A+(p..r));
  @ assigns A[p..r];
  @*/
int partition (int A[], int p, int r) { ...
```

The `assigns` clause is a frame condition stating that the only externally visible mutations affect the positions `p..r` of the array.

These annotations also apply to loops. In fact loop frame conditions are required, in order to allow the function's frame condition to be proved. The `loop assigns` condition below states that the loop assigns variables `i j tmp`, in addition to the array. The variables are not included in the function's frame condition because they are local, and thus not externally visible.

```
/*@ loop invariant p <= j <= r && p-1 <= i < j;
  @ loop assigns i, j, tmp, A[p..r];
  @*/
  for(j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
    }
```

Note that the use of a frame condition for the loop dispenses the inclusion of the `\valid(A+(p..r))` invariant, because it implies that the variable `A` is not modified.

Termination: Loop Variants

Following Hoare's logic, termination can be proved using loop variants — integer expressions that have positive value each time the loop is executed, and whose value strictly decreases with each iteration.

The final version of our example function is the following:

```
/*@ requires 0 <= p <= r && \valid(A+(p..r));
  @ assigns A[p..r];
  @*/
int partition (int A[], int p, int r)
{
  int x = A[r];
  int tmp, j, i = p-1;

/*@ loop invariant p <= j <= r && p-1 <= i < j;
  @ loop assigns i, j, tmp, A[p..r];
  @ loop variant r-j;
  @*/
  for(j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      tmp = A[i];
      A[i] = A[j];
      A[j] = tmp;
    }
  tmp = A[i+1];
  A[i+1] = A[r];
  A[r] = tmp;
  return i+1;
}
```

IDE

Alternatively the Frama-C IDE can be used, allowing for RTE and WP to be invoked sequentially. The GUI allows for the list of VCs and their present state to be visualized in a convenient way.
[**DEMO**]

```
frama-c-gui partition_init.c
```

Exercises

1. Consider the following alternative version of `partition`, using an auxiliary function `swap`.

```

void swap(int t[], int i, int j, int start, int end);


/*@ requires 0 <= p <= r && \valid(A+(p..r));
  @ assigns A[p..r];
  @*/
```

```
int partition (int A[], int p, int r)
{
  int x = A[r];
  int j, i = p-1;

  for (j=p; j<r; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j, p, r);
    }
  swap(A,i+1,r,p,r);
  return i+1;
}
```

a. Verify `partition` again. To do so you will have to define and verify the function `swap` , for which you will have to equip it with an adequate contract. New verification conditions will be generated for `partition`, enforcing that the precondition of `swap` is satisfied when it is called by the latter function.

Note that `partition` can only be correct if `swap` is correct: for instance if a seg. fault occurs in `swap`, this will also be a fault in the execution of `partition`. In terms of verification, this means that successfully verifying `partition` implies proving all verification conditions of both functions.

b. Now delete / comment out the implementation of `swap`, leaving only its prototype and contract, and verify `partition` again. Observe that VCs will only be generated for the latter function, and they will all be successfully proved. What does this mean?

The implementation of `partition` is correct regardless of the implementation of `swap`: as long as the latter is correct (i.e. consistent with its contract), we know that `partition` will be correct, since it meets the precondition of `swap` .

This style of modular verification is know as contract-based.
The idea is that, when verifying a collection of mutually-recursive functions, we verify each of them with respect to its own contract, and use those contracts to reason about function calls (rather than expanding function definitions, which would lead to spaghetti-style verification 😀).

When function `f` calls `g`:

- a VC is generated for `f` based on the precondition of `g`, ensuring that when `f` calls `g` this precondition is always satisfied
- the postcondition of `g` can in turn be used to help prove subsequent verification conditions of `f` , for instance to prove loop invariants / frame conditions / postconditions (in the above example this will become evident when we consider functional properties)

The program, seen as a collection of functions, is correct if the VCs of all functions are valid. By itself, the correctness of each function is *conditional,* since it assumes that the functions invoked by it are all also correct.

2. Consider the following program consisting of an array maximum function and a main function that invokes it. Note that the program contains an error, since `maxarray` is called with an array that is longer than the global allocated array.

```
#define LENGTH 100
int vec[LENGTH];
int max;

int maxarray(int u[], int size) {
  int i = 1;
  max = 0;
  while (i < size) {
    if (u[i] > u[max]) { max = i; }
    i++;
  }
  return max;
}


void main() {
  maxarray(vec, 150);
}
```

a. Equip `maxarray` with a contract and prove that it does not produce runtime errors.

b. Now try to verify the whole program and see how the error is identified. Correct it by allocating a longer array and verify the program again.