Code Analysis using abstract interpretation
Milestone 1

Carlos Brito
Tiago Loureiro

Dept. Informtica,
Universidade do Minho
Braga, Portugal

2015

Introduction

- Our main objective for this project will be assisting ASML on improving the quality of some C code currently in use.
- In particular, we will attempt to establish **function contracts**, and through **deductive verification** ensure that such contracts hold for all executions.
- We will also employ **code slicing** techniques in order to facilitate code refactoring.
- Our main tool will be **Frama-C**.

What is Frama-C?

- Frama-C is a tool for **static analysis** of C programs.
- Frama-C is organized with a **plug-in architecture**, meaning different plug-ins implement different analysis methods.
- In this presentation, we will focus on the "WP" plug-in (weakest precondition).

The WP plug-in

- The WP plug-in aims at specifying functional/behavioral properties of C source code.
- It requires programmers to formulate, for each function in the C code, a set of **annotations** in the formal language ACSL, describing in **Hoare logic** its requirements and expected outcomes.
- The WP plug-in operates by inferring the set of **proof obligations** from the specified code annotations and the code using **weakest precondition calculus**.
- These proof obligations are then submitted to an external Theorem Prover in order to check their validity.

Weakest Precondition Calculus

- An annotated C statement can be interpreted as a Hoare Triplet of the form:

  $\{P\}$ stmt $\{Q\}$

- This reads "Whenever `P` holds, then after running `stmt`, `Q` will hold", where `Q` is the programmer-written assertion.

Weakest Precondition Calculus

- Through WPC, it is possible to infer the weakest pre-condition taking into account both stmt and Q, so the triplet becomes:

  {wp(stmt, Q)} stmt {Q}

Example

- Consider the following example:

```
x = x + 1;
//@ assert x > 0;
...
```

- The Hoare triplet equivalent would be:

$$\{P\}\ x = x + 1;\ \{x > 0\}$$

Example

- In this instance the result is intuitive as `P` is calculated using the substitution rule:

  $\{x + 1 > 0\}\ x = x + 1;\ \{x > 0\}$

- More generally, for any statement and any property, it is possible to define such weakest pre-condition.

Verification

- Consider now a function f, annotated with pre-condition `P` and post-condition `Q`. This could be formalized as:

$$\{P\}\ f\ \{Q\}$$

- By introducing the discussed wp function and assigning $W = wp(f, Q)$ the triplet becomes

$$\{W\}\ f\ \{Q\}$$

Verification

- If now we are able to prove that W is implied by the pre-condition P we can then conclude that the `P` also implies the post-condition `Q`, as described below.

$$\frac{(P \Longrightarrow W) \qquad \{W\}\, f\, \{Q\}}{\{P\}\, f\, \{Q\}}$$

- This is the main idea behind proving a property by weakest pre-condition computation.

Verification

- Generally speaking, the process consists of considering every annotation `Q`, and computing its weakest precondition `W` across all the statements from `Q` up to the beginning of the function.
- Finally the property $P \implies W$ is submitted to a theorem prover, where `P` are the specified preconditions of the function.
- This is a proof obligation, and if it is discharged then it is possible to conclude that the annotation `Q` holds.

Our work so far

- Our first assignment:

  Create a simple interface and implementation for a sensor.

- We were given the following function prototypes:

```
RESPONSE SENSOR_A_request_x (const SENSOR_request_scan_x_t *params_p, int *id_p);
RESPONSE SENSOR_A_get_result_scan_x (int id, SENSOR_A_get_result_x_t *params_p);
RESPONSE SENSOR_A_request_y (const SENSOR_request_scan_y_t *params_p, int *id_p);
RESPONSE SENSOR_A_get_result_scan_y (int id, SENSOR_A_get_result_y_t *params_p);
```

- And the definition of each of the data structures.

Our work so far

- Coupled with the following requirements:
  1. *Sensor A shall only handle 200 queued requests maximum.*
  2. *It shall only be possible to get results from valid logic ids.*
  3. *It should not be possible to get a scan result from something that wansn't queued before.*

The implementation

- Since the required sensor stores heterogeneous data, either a `result_x` or a `result_y`, a tagged union is used to describe each stored element.
- This wrapper struct also contains the id assigned to the result.

```
#define REQUEST_TYPE int
#define REQUEST_X 0
#define REQUEST_Y 1

typedef struct
{
    int id;
    REQUEST_TYPE request_type;
    union
    {
        SENSOR_A_get_result_x_t *result_x;
        SENSOR_A_get_result_y_t *result_y;
    };
} scan_result_t;
```

The container data structure

- Since remove operations are expected to be both frequent and unordered the first attempt of implementation was made using a simple linked list.
- However, we later determined that using either WPs or Jessies memory model this implementation would be very difficult to verify.
- This is due to runtime memory allocation. Linked list verification deals with issues regarding separation of memory, which is an advanced topic and currently not very well supported.

The container data structure

- Our next option was using a simple array with an auxiliary stack to keep track of free positions.
- However, for simplicity, we switched to a pure array based implementation, where the array must be scanned at each insertion in order to find a free position.
- Although inefficient, this option was easiest to verify.

The implementation

- This is the sensor implementation

```
typedef struct
{
    scan_result_t *scan_results[CAPACITY];
    unsigned int size;

} sensor_t;
```

Verification example

- The init function is fully verified and we will use it as an example:

```
void init_sensor (sensor_t *s)
{
    int i;

    for(i = 0; i < CAPACITY; i++)
        s->scan_results[i] = 0;

    s->size = 0;
}
```

Verification example

- Pre and post conditions

```
/*
    requires \valid(s);
    requires \valid(Storage(s)+(0.. Capacity1 ));

    requires Capacity > 0;

    assigns Storage(s)[0..( Capacity1 )];
    assigns s->size;

    ensures VALID: Valid(s);
    ensures EMPTY: Empty(s);

    ensures NULLIFIED: \forall integer k ;
            (0 <= k < Capacity) ==> (Storage(s)[k] == \null);
*/
```

Loop invariant

- Loop invariant

```
/*
   loop invariant RANGE: 0 <= i <= Capacity;
   loop invariant ZERO: \forall integer k; 0 <= k < i ==> (Storage(s)[k] == \null);
   loop assigns i, Storage(s)[0..( Ca
   loop variant Capacity - i;
   for(i = 0; i < CAPACITY; i++)
       s->scan_results[i] = 0;
*/
```

A slightly more complex example

```
int  get_result ( sensor_t  *s,  int  id ,   scan_result_t  ** results ,  REQUEST_TYPE rtype)
{
    for ( int  i  = 0;  i  <  CAPACITY;  i++)
    {
        if (  s->scan_results[i]   &&
        s->scan_results[i]->id == id &&
            s->scan_results[i]->request_type == rtype
          )
        {
            * results  = s->scan_results[i];
            s->scan_results[i]  = 0;
            s->size--;
            return  0;
        }
    }
    return  -1;
}
```

A slightly more complex example

- Pre and post conditions

```
/*
    requires \valid(s);
    requires \valid(Storage(s)[0..(  Capacity1  )]);
    requires s->size > 0;

    requires \valid( results );

    requires NoIdEqual(s);

behavior no_result :
    assumes \ forall integer k; (0 <= k < 200) ==> !(\valid(Storage(s)[k]) &&
                                                   Storage(s)[k]->id == id &&
                                                   (Storage(s)[k]->request_type == rtype));
       ensures NRES : \result == -1;
       ensures NoIdEqual(s);
       ensures Unchanged{Pre,Here}(s);

    behavior has_result :
       assumes \exists integer k; (0 <= k < Capacity) && \valid(Storage(s)[k]) &&
                                                   Storage(s)[k]->id == id &&
                                                   (Storage(s)[k]->request_type == rtype);
       ensures HRES : \result == 0;
       ensures NoIdEqual(s);
       ensures \exists integer k; (0 <= k < Capacity) && \valid(Storage(s)[k]) &&
                                                   Storage(s)[k]->id == id &&
                                                   (Storage(s)[k]->request_type == rtype) &&
                                                    popIndex{Pre,Here}(s, k);
complete behaviors;
 disjoint behaviors;
*/
```

Loop invariant

- Loop invariant

```
/*
    loop invariant RANGE: 0 <= i <= Capacity;
    loop invariant \forall integer k; (0 <= k < i) ==> !(\valid(Storage(s)[k]) &&
                                                     Storage(s)[k]->id == id &&
                                                     (Storage(s)[k]->request_type == rtype));
    loop assigns i;
    loop variant Capacity - i;
*/
for(int i = 0; i < CAPACITY; i++)
{
    if( s->scan_results[i]  &&
        s->scan_results[i]->id == id &&
            s->scan_results[i]->request_type == rtype
          )
        {
            *results = s->scan_results[i];
            s->scan_results[i] = 0;
            s->size--;
            return 0;
        }
    }
```

Code Analysis using abstract interpretation
Milestone 1

Carlos Brito
Tiago Loureiro

Dept. Informtica,
Universidade do Minho
Braga, Portugal

2015