Journée thématique QeR

Assurance Qualité Logiciel

18 avril 2019 à Paris

Qualité Logiciel dans un projet de Nanosatellite

Colin González

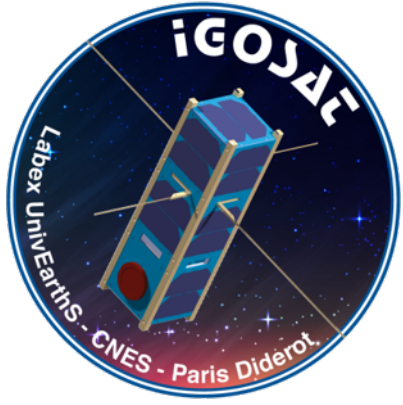
IGOSat - AstroParticules et Cosmologie

18 avril 2019

IGOSat Mission

Ionospheric and Gamma-ray Observation satellite

- Student Driven Nanosatellite Mission
- Over 250 students involved as of year 2018
- Dimensions are $10x10x30$cm
- Expected to be ready for launch by the end of year 2020 and fly at nearly 600km of altitude with a polar orbit
- IPGP payload : a dual-frequency GPS receiver to measure the total electron content (TEC) of the ionosphere
- APC payload : a scintillator to measure the γ-ray spectrum and electrons in the Auroral Zones and South Atlantic Anomaly



- Safety properties : the computer will never reach an undefined state. For instance, attempt to divide by zero, invalid memory access, memory overflow (Nothing bad will ever happen).
- Liveness properties : the computer will carry out meaningful computations and not stay in infinite loops resulting into loosing control of the platform (Something good will eventually happen).
- Modularity and readability : functions should be generic for all *correct* use cases. This makes it easy to use the same functions for several purposes and breaking computations into smaller intuitive functions, which helps readability.

Formal Methods, Testing and Proving

What are formal methods, anyway ?

- *Formal method* : any systematic technique based on a set of strict rules in order to achieve a result
- Often the set of strict rules is related to mathematics or logic (perceived as trustworthy)
- The word *formal* should be understood as a standardized structure, therefore providing a trust base
- In programming formal methods are almost exclusively based upon mathematics and logic
- Formal methods in programming involve *formal specifications* and *formal proofs*. The former being the requirements and the later being compliance with such requirements.
- Provers are a kind of software that carry out *formal proofs* while constantly checking that subjects proof do not perform illegal actions, making sure that the rules are respected
- *Formal specifications* are easily used as input in fully automatic provers. Correlated with source files it is possible to automatically compute a proof of correction and verify the compliance of code with a formal specification

We have removed a step in the development cycle. The code verification happens during the implementation phase (no code is compiled until the code is proved correct).

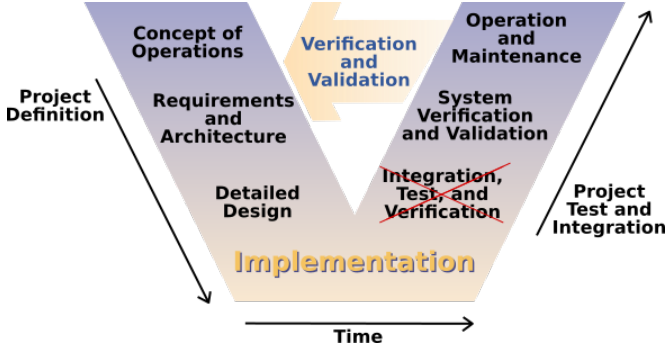


Figure – Impact on the development cycle

boolean_t equals_one(unsigned int x)

```
boolean_t equals_one (unsigned int x) {
  if (x==1) {
    return TRUE;
  }
  else {
    return FALSE;
  }
}
```

Figure – A trivial function

The function returns *TRUE* if $x = 1$ and *FALSE* otherwise.

Testing the function

```
boolean_t test_equals_one () {

  for(i=0;i<MAX;i++){
    if (i==1){
      if(equals_one(i) == FALSE)
        return FALSE;
    }else{
      if(equals_one(i) == TRUE)
        return FALSE;
    }
  }

  last = equals_one(MAX);
  return (last == FALSE);
}
```

Figure – A test function in C

Let's make a program that *tries* all the possible values and compare the real result against to the expected result.

Proving the function

```
boolan_t proof_equals_one() {
  x=0,y=1,z=1;
  assert(equals_one(x)==FALSE);
  x = y;
  assert(equals_one(x)==TRUE);
  x = y + 1;
  assert(equals_one(x)==FALSE);
  x = y + z;
  assert(x > 1);
  assert(equals_one(x)==FALSE);
  return TRUE;
}
```

Figure – A proof of correction in C

- If for all $x > 1$ then $equals\_one(x) \rightarrow FALSE$.
- For $z \geq 1$
- If $x = y + z = y + 1 + (z - 1)$
- And $y + 1 + (z - 1) > 1 \rightarrow y + z > 1$
- $equals\_one(y + z) \rightarrow FALSE$
- For all x greater to 1 then $equals\_one(x) = FALSE$

Proving the function (details)

```
assert(x > 1);
assert( {if (x==1) {
      return TRUE; }
   else {
      return FALSE;}
} == FALSE);
```

Figure – Inlining the function call

- Call to *equals_one*() inlined
- *assert*$(x > 1)$ succeeds
- $x = 1$ will not hold and return *FALSE*
- $x \in \{0 - 2^{64} - 1\}$
- For $x = 0$, $x = 1$ and $x > 1$ the function is proved correct
- *equals_one* returns *TRUE* when $x = 1$ and *FALSE* otherwise

Program testing can be used to show the presence of bugs, but never to show their absence!
(E. Dijkstra, 1970)

Frama-C

FRAmework for Modular Analysis of C programs made by the CEA List and Inria Saclay - Ile-de-France. It is a tool capable of automatically proving C software source files annotated with formal specifications. This step happens just before compilation.

Evolved Value Analysis (EVA)

- Performs Abstract Interpretation
- Over Approximation of variable domains
- Raises warnings about possible run-time errors

Weakest Precondition (WP)

- Performs a deductive analysis
- Checks for preconditions, loop invariants and postconditions validity
- Requires to have absence of run-time errors to provide useful results

Case Study

This are some formal specification stating when a date and telecommands (a tc for short) is valid. That is a date within the bounds of the beginning of life (BOL) and the end of life (EOL) ; a telecommand is either, power on, power off, reset or deleted and has a valid date.

predicate ValidDate(min_bound $\in \mathbb{N}$, max_bound $\in \mathbb{N}$, date $\in \mathbb{N}$) =
    min_bound $\leq$ date $\leq$ max_bound ;

predicate ValidTC(tc_t t) =
        (t.cmd $\equiv$ power_on $\vee$ t.cmd $\equiv$ power_off $\vee$ t.cmd $\equiv$ reset $\vee$ t.cmd $\equiv$ deleted)
        $\wedge$ ValidDate(BOL,EOL,t.date) ;

A Function Specification to Compare Two Dates

```
requires ValidDate(BOL,EOL,x);
requires ValidDate(BOL,EOL,y);
ensures (BOL-EOL) ≤ result;
ensures result ≤ (EOL-BOL);

behavior EQUAL:
assumes x ≡ y;
ensures result ≡ 0;

behavior LGTR:
assumes x > y;
ensures result > 0;

behavior LLTR:
assumes x < y;
ensures result < 0;
```

- Require statements have to be *true* before the function call
- Ensures statements have to be *true* after the function call
- Assumes statements introduce hypothesis to specify a given case

A correct implementation for the previous specification

```
int compare_date(unsigned int x, unsigned int y) {
  if (x > y) {
    return x - y;
  }
  else if (y > x) {
    int a = y-x;
    return -a;
  }
  return 0;
}
```

Figure – An overflow safe function to compare two dates

- Deduce the correct order to perform the difference
- Apply the sign if needed
- This function is very simple yet avoiding the overflow can be easily forgotten !

Discovering errors

```
for (int i = 0; i < TC_SIZE; i++) {
   now = get_date ();
   if (now < BOL ∨ now > EOL){
      write_log("inconsistent date",error,EOL);
      return some;
   }
   else {
      some_tc = get_next_tc();
      else {
         diff = compare_date(some_tc.date, now);
         switch(tc_ready(some_tc,diff)) {
```

Figure – Code to process telecommands for execution

- Get the current date
- Get next telecommand from an array of volatile data
- Compares the current date with the due execution date
- Frama-C will fail when the dates are compared
- Because the values are stored in an *unsafe* memory (i.e. *volatile*) where the value may change regardless of the program

A possible fix

```
behavior VALID:
assumes ValidTC(BOL,EOL,t);
ensures result ≡TRUE;

behavior INVALID:
assumes ¬ValidTC(BOL,EOL,t);
ensures result ≡ FALSE;
```

```
bool_t check_tc(tc_t t) {
  if ((t.cmd == power_on ∨
   t.cmd == power_off ∨
   t.cmd == reset ∨
   t.cmd == deleted) ∧
   BOL ≤ t.date ∧
   t.date ≤ EOL) {
    return TRUE;
  }
  else{
    return FALSE;
  }
}
```

We use a helper function to ensure that $some\_tc.date$ is valid.
That is $BOL \leq some\_tc.date \leq EOL$.

Conclusion and Discussion

Proving something useful can be hard

- Example of a valid implementation of well known *malloc* which returns either a pointer or *NULL*

```
void * malloc(size_t size) {
  return NULL;
}
```

- Proving the correction of this is trivial but only tests can inform us that this implementation is not useful (however correct it may be)
- However writing a robust test base that will eventually cover all the possible bugs can be equally hard

A new element in the toolbox

- Depending on the program under study both a proof and a test can be extremely useful
- Proof of programs is but another tool in the static analysis field
- Proving properties can reveal behaviors that are unpredictable, that are not bugs and not discovered by tests

Proving programs is easier than you can think

- Mathematical proofs are infamous for their difficulty
- However new tools are making this easier, faster and more robust
- Formal methods do not prevent us from testing and provide different information
- The quality of the code we have written has improved (safer, more modular and more readable) and so have our programming techniques