Lab
Fun with tools!

Preliminaries

Preparation A `.tgz` archive of this lab is available on the course's website:

`http://laure.gonnord.org/pro/teaching/verifM2.html`

ENS Machines: paths In your bashrc:

```
OPAMDIR=/soft/enseignants/Laure.Gonnord/opam
export OPAMROOT=$OPAMDIR/root
PATH=$OPAMDIR/bin:$PATH
eval $(opam config env)
```

We installed Z3 (smt solver) and Frama-C for you in this OPAM directory. This should work out of the box.

It is possible to use Z3 using SMT-LIB2 files, from a C/C++ API, from an OCaml API and from Python. In this lab class we explain how to deal with SMT-LIB2 files.

Those interested in trying the Python API can use (one line command):

```
export
PYTHONPATH=/soft/enseignants/Laure.Gonnord/z3/spacer_2016-10-01_027d303
 /lib/python2.7/dist-packages/:$PYTHONPATH
```

On your own machines

```
opam repo add termite 'https://github.com/termite-analyser/opam-termite.git'
opam install z3 frama-c lablgtk ocamlgraph altgr-ergo coq coqide why3
```

Unfortunately, installing Z3 through opam does not install the Python interface. You'll have to install Z3 yourself. Go to `https://github.com/Z3Prover`

1- Exercises with Z3 and Spacer

`https://github.com/Z3Prover/z3`

As SMT-solver

This exercise comes from real-world concerns about finding the worst-case execution time of the body of critical control loops in embedded software. If you wish to know more about this, see Henry et al., *How to Compute Worst-Case Execution Time by Optimization Modulo Theory and a Clever Encoding of Program Semantics*, `https://arxiv.org/abs/1405.7962`

We assume some pre-analysis gives us an upper bound on the number of clock cycles taken to execute each program block. A naive method to compute the worst-case execution time of a loop-free program would be to compute the path of maximal weight from beginning to end. This can however grossly overestimate the execution time, depending on how the program operates. For instance, some control software have constructs such as these (here is a very simplified and obvious version):

```
if (clockCycle % 3 == 0) {
  do something expensive, 200000 cycles
}
...
```

```
if (clockCycle % 6 == 1) {
  do something expensive, 100000 cycles
}
```

The path of maximal weight goes through both "if", counted as 300000 cycles. Yet this path is impossible, because the clock cycle cannot be both divisible by 3 and congruent to 1 modulo 6.

Because of this, it is tempting to isolate the path of maximal weight that is *feasible* according to program semantics. That is, we encode the loop-free program into SMT so that each execution trace corresponds to a solution of the SMT formula, giving the values of the variables and the locations crossed during execution. The last step of the process is to check that there is no path longer than the longest path found. We'll now see how this approach works on a very simple example. Consider the program:

boolean $b_1, \ldots, b_n$;

```
if (b_1) { spend ≤2 clock cycles }
   else { spend ≤3 clock cycles}
if (b_1) { spend ≤3 clock cycles }
   else { spend ≤2 clock cycles}
⋮
if (b_n) { spend ≤2 clock cycles }
   else { spend ≤3 clock cycles}
if (b_n) { spend ≤3 clock cycles }
   else { spend ≤2 clock cycles}
```

Obviously, the total execution time takes $\leq 5n$ cycles. We'll now see how to encode the above problem into SMT-LIB2 and prove it automatically.

In the `Diamonds` directory of the archive, you will find:

- Two scripts `gen_diamond.py` and `gen_diamond2.py` to generate two different families of unsat "diamond" formula. For instance, `python gen_diamond.py 1` generates the following formula:

$$(y_0 \leq x_0 + 2) \wedge (z_0 \leq x_0 + 3) \wedge \big(x_1 \leq y_0 \vee x_1 \leq z_0\big) \wedge (x_1 > 3) \wedge (x_0 = 0)$$

- A script `gen_diamond3.py`that generates a family of SMT formulas in an SMT-LIB2 file.
- And also `gen_horn_diamond.py` `gen_horn_diamond_ungrouped.py`.

To check for sat/unsat, for instance:

```
$ python gen_diamond.py 1 > diam1.smt
$ z3 -smt2 diam1.smt
unsat
```

With Python/Gnuplot/whatever, try to caracterise the experimental complexity of Z3's algorithm on the first three classes of formula ($time = f(n)$ with an adequate n). You can use the `-st` option of Z3 to get some useful stats.

As Horn clause solver

Z3 and Spacer, in addition to deciding existential first-order formulas, can also solve second-order formulas — that is, formulas where some of the unknowns are predicates, in other words, functions from the program variables to the Booleans. This makes it possible to directly ask them to solve fixed point problems.

Consider for instance

```
int i=0, j=1;
while(i < 10) {
  i = i+1;
  j = j+2;
}
assert(j < 25);
```

An inductive loop invariant *loop* should satisfy:

$$loop(0,1) \tag{1.1}$$
$$\forall i, j \in \mathbb{Z}\ loop(i,j) \wedge i < 10 \implies loop(i+1, j+2) \tag{1.2}$$
$$\forall i, j \in \mathbb{Z}\ loop(i,j) \wedge i \geq 10 \implies j < 25 \tag{1.3}$$

This can be written in SMT-Lib format as:

```
(set-logic HORN)

(declare-fun loop (Int Int) Bool) ; i j

(assert (loop 0 1))

(assert (forall ((i Int) (j Int))
  (=> (and (loop i j) (< i 10)) (loop (+ i 1) (+ j 2)))))

(assert (forall ((i Int) (j Int))
  (=> (and (loop i j) (>= i 10)) (< j 25))))

(check-sat)
```

Calling the solver:

- Z3/PDR is called as `z3 file.smt2`.
- Spacer is called as `z3 fixedpoint.engine=spacer file.smt2` (beware, if forgetting `fixedpoint.engine=spacer`, one calls the default fixed point solver, that is, PDR).

Assignment:
1. Try this example.
2. Try this example with 25 replaced by 18.
3. Try this example with 10 and 25 replaced by suitable larger numbers. What happens? [teachable moment]
4. Try this example replacing 10 by a symbolic constant n and 25 by a suitable expression.
5. Try with Pagai.

2- Exercises with FramaC

The official webpage:

`http://frama-c.com/index.html`

Some help about the language used for assertions (ACSL) an be found here:

`https://frama-c.com/acsl_tutorial_index.html`

or in the official documentation:

`http://frama-c.com/download/frama-c-user-manual.pdf`

In the archive, the `FramaC` directory contains:

- A Readme to use FRAMA-C: `ModeEmploiFramaC.txt`.
- A set of `.c` files.
- A Makefile.

First, compile all C files with `-Wall`, in order to be at least confident in their syntax (just type `make all`).

Presentation Frama-C features several plugins, one of which (*Value*) performs static analysis by abstract interpretation using intervals and partitioning. Here we shall use the *WP* tool, which performs a kind of *weakest precondition* computation. The user may provide, for each function, what kind of arguments it *requires* and, assuming these requirements are satisfied, what kind of properties it *ensures* at the end of its execution The *assigns* annotation specifies what the function can or cannot modify, which may be difficult to describe due to the availability in C of global variables, pointers etc.

The tool then checks that the requirements of a function truly entail the properties that it is supposed to ensure; it does so by calling a SMT-solver (by default, Alt-Ergo) and asking it to find values for an execution starting in the requirements but ending outside of the property to ensure. The absence of such a solution shows the property to be true (green light), its presence shows it to be false (orange light).

When there are loops, the user has to provide *loop invariants*. Each loop invariant must be *established* by the program preceding the loop (that is, it must hold initially) and must be *preserved* by the guard and loop body. One may also add a *assigns* statement to specify what the loop body may modify. These checks proceed in the same way.

To prove termination (optionally), the user has to provide *loop variants*: an integer expression with a value initially nonnegative and that decreases along iterations.

The user may also provide extra *assertions*. The tool will use them to provide properties later in the program, and will of course try to prove them.

Frama-C can use several SMT-solvers (including Yices, Z3 etc.), and also use non-SMT provers (Spass, Coq etc.), but we shall stick to Alt-Ergo.

To the properties to prove provided by the user, *WP* may optionally add properties ensuring the absence of *runtime errors* (RTE), such as array access of out bounds, arithmetic overflow, access through incorrect pointers. If you select that option (`-wp-rte`) you may have to provide extra requirements and invariants ensuring the absence of runtime errors (this may be a bit tedious with respect to arithmetic overflow).

The ACSL specification language is very rich; you may see its definition at `https://frama-c.com/download/acsl_1.3.pdf`

Do it yourself! For each file [1], prove with FRAMA-C that the program has no RTE, and that all the assertions are true (some indications are given as comments inside the files). You'll have to find the necessary loop invariants.

You may wish to start without the checks for incorrect executions (`-wp-rte`) and insert it again later.

Prove them in the following order:

- `mult.c`: a simple simple case, prove it with:[2]

  ```
  frama-c-gui -wp -wp-rte -wp-split -wp-alt-ergo-opt="-backward-compat" mult.c &
  ```

  Find in the doc the meaning of all types of all decorations : `loop invariant`, `assigns`, `variant`, `assert`, and the `at(x,Pre)` in ACSL.

- `arith1.c`: a first simple invariant
- `div*.c`: invariants + precondition + postcondition + function calls.
- `linear_search.c`: linear search in an array.
- `min_sort.c`: selection sort with a bit of pointer manipulation (`swap`)

[1] except `any.c`, which is an auxiliary file defining a function returning nondeterministically chosen values

[2] In the ENSL installation, the version of Alt-Ergo uses by default an output format too modern for the version of Frama-C, thus the need for `-wp-alt-ergo-opt="-backward-compat"`. This may or may not be needed other installations.