Towards the automatic translation from C program to Horn clauses [*]

Paul Raynaud
VERIMAG
Grenoble, France
paul.raynaud66@hotmail.fr

Supervised by: Michaël Périn.

I understand what plagiarism entails and I declare that this report is my own, original work.
Raynaud Paul, 24/08/2018 : [illegible]

Abstract

Automatically checking if a program realizes its specification allows us to prove something stronger than the mere absence of certain errors ; we are outright capable of proving the correction of the program. Therefore we are trying to prove, based on a source program, whether it's capable of satisfying the properties that are attached to it or not with an entirely automatic procedure.

We will rely on already known hypothesis such as Hoare Logic, and control flow graph study to generate Horn clauses. Once those logical formulas are generated, we resort to a SMT (Satisfiability modulo theories) solver to resolve the so-called formulas.

1 Related work

1.1 The formal verification and CompCert

Tools capable of realizing programs verification have been developed, like SeaHorn. However, as will be discussed later, those tools are not compatible with the "recent" compiler CompCert C. With the unique semantic of CompCert, being capable of automatically verifying a program would be an interesting feature to be added. Thereafter, on a code, the specification of which has been proven, we could have a proved program during the compilation without integrating errors by a compiler of a "hazardous" semantic.

1.2 SeaHorn

A tool like SeaHorn ([Gurfinkel *et al.*, 2015]) is practically based on the same hypothesis that are used in this internship, with the only difference that SeaHorn takes C and C++ programs and uses the bitecode generated by Clang [Fan, 2010]. Clang compiler is different than ComCert C. Therefore it doesn't have the same semantic. As a result, the generated code is different. It is for this exact reason that we cannot use SeaHorn.

The general reason why we cannot re-use the programs verification tools that already exist is not only that those tools were not developed with the same semantic as C but also that the in dependant development of a tool attached to CompCert C is preferable if there are modifications to be made.

1.3 CompCert

Yang and al [Yang *et al.*, 2011] realized a study with the purpose of showing the miscompilation of the different C compilers. All the compilers tested within their study generated incorrect codes for certain tests (including CompCert). Let us note that CompCert was not entirely developed in 2011, but it was already reducing bugs in comparison with the other compilers :

> "The striking thing about our CompCert results is that the middle- end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying : we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users".

CompCert is a project initiated in 2005 that aims to create the first proven C compiler [Blazy and Leroy, 2005]. It is a project with a complex structure that has been entirely proven. It is mainly developed in Coq. Using the Coq code, it is possible to extract Ocaml code. It is precisely on this Ocaml code that we are working.

2 The concept of Translation

2.1 Expected result

The global goal that is expected from our tool is specified in the Figure 1. We will go through the detailed translation, step by step.

[*]. These match the formatting instructions of IJCAI-07. The support of IJCAI, Inc. is acknowledged.

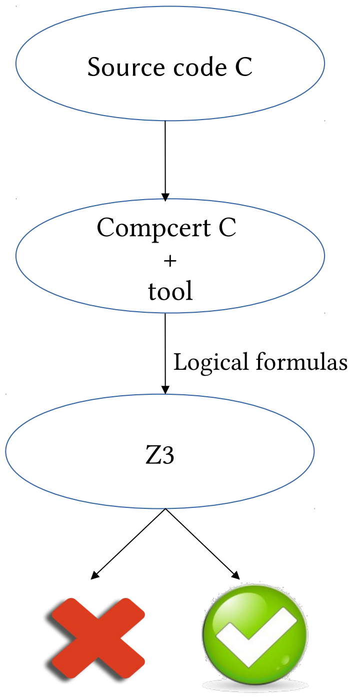


FIGURE 1 – The goal of our tool

2.2 The description of our example

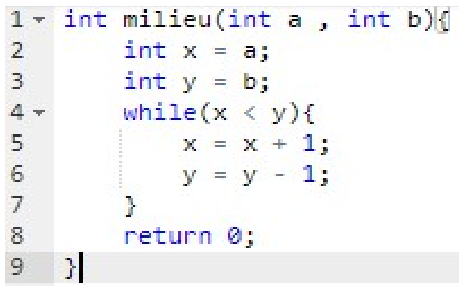
To illustrate our statements all along this paper, we will use an example of C code ”middle.c”.

The algorithm shown in Figure 2 calculates the middle value of an interval simply by increasing x and decreasing y while y is greater than x.

At the end of the algorithm we have the following properties

y $\geq (a+b)/2 \geq x \wedge 0 \leq y-x \leq 1$

that frame the middle value.



```
int milieu(int a , int b){
    int x = a;
    int y = b;
    while(x < y){
        x = x + 1;
        y = y - 1;
    }
    return 0;
}
```

FIGURE 2 – Our example program ”middle.c”

However, it is not trivial to ”manually” prove this algorithm correction.

This algorithm is interesting because if any small change is made to the loop

$$< by \leq$$

the properties stated previously are no longer valid since y-x cannot be worth 2.

Therefore, making a small change in our logical formulas is enough to judge whether our tool works correctly or not.

2.3 From a Control Flow Graph to Logical Formulas

Let’s take a simple example, the C code and RTL translation of which are defined in Figure (...).

We will first recall the formulas of Hoare’s logic :

A triplet of Hoare is :

$$\{\rho\}program\{\gamma\}$$

where ρ represents the pre-condition, and γ the post-condition of the program.

There are 2 methods for generating the logical formulas of a program :

- Strongest Postcondition, that generates the formulas forward.

- Weakest Precondition, that generates the formulas backward. We will be using the Weakest Precondtion.

We will now explain the rules concerning Weakest Precondition : A program to be verified must satisfy :

$$WP : \frac{\rho \Longrightarrow WP(instr, \gamma)}{\{\rho\}\ instr\ \{\gamma\}}$$

$\gamma\ is\ the\ preconditionand\rho\ the\ postcondition.$

The sequence :

$$\frac{\{\rho\}\ S1\ \{\theta\}\ \{\theta\}\ S2\ \{\gamma\}}{\{\rho\}\ S1;\ S2\ \{\gamma\}}$$

where $\theta\ a\ descending\ property\ from\ S1\ and\ S2$

The affection :

$$\frac{}{\{\rho\}\ x\ :=\ e\ \{\gamma[x/e]\}}$$

The condition :

$$\frac{\{\rho \wedge C\}S1\{\gamma\} \wedge \{\rho \wedge \neg C\}S2\{\gamma\}}{\{\rho\}if\ C\ then\ S1\ else\ S2\{\gamma\}}$$

The implication :

$$\frac{\{\rho\} \Rightarrow \{\rho'\} \wedge \{\gamma'\} \Rightarrow \{\gamma\} \wedge \{\rho'\}i\{\gamma'\}}{\{\rho\}i\{\gamma\}}$$

We must now apply these formulas to our graph from the RTL representation Figure.

3 Contribution

Theoretical principle

We were integrated in CompCert as soon as it created the first CFG. We were able to use the principles mentioned previously, along with an already defined Hoare Logic. To apply them in CompCert, we will need to re-write the rules and adapt them following the intermediate language instructions that we are tapping : RTL (Register Transfer Language). Afterwards, we will pass the obtained logical formulas to a SMT solver to solve them Z3 [de Moura and Bjørner, 2007].

3.1 The work provided

CompCert C data recovery

First of all, we have to relate to CompCert, an industrial compiler. We have to fully understand the global architecture of the software of CompCert and the way the different files are linked before working on it. Despite the fact that we have had a relatively easy access to the needed data, it was not enough for a smooth manipulation. In fact, to modify something in the purpose of testing it, we had to recompile CompCert. For this reason, we used a particular Caml library (ppx_deriving.show) to retrieve the data structure handled by CompCert.

The principle of the library is to automatically generate printers for all the types : sum, record or other. In order to ensure this, we needed to modify not only the files where the types that we wanted to display had been defined but also the ones where we have defined all the basic types.

In Figure 3 we have the classic display of RTL by CompCert C. Below we detail an instruction and show the structure of a RTL instruction.

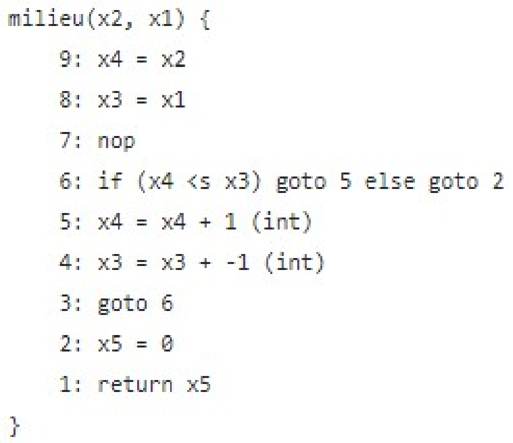
9 : x4 = x2

Here is the 9th line in our program. This is the way CompCert displays the following instruction, which is the real structure that we get from the library ppx$_deriving.show$.

(RTL.Iop (Op.Omove, [(BinNums.Coq_xO BinNums.Coq_xH)], (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH)), (BinNums.Coq_xO (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH)))))

Now let's see in details what it means :

(RTL.Iop $\longrightarrow$ indicates that this is an operation
(Op.Omove, $\longrightarrow$ same as a move in assembler code
(BinNums.Coq_xO BinNums.Coq_xH)
$\longrightarrow$BinNums defined binear 1 and 0, here we have $10_2 = 2_{10}$, for x2



```
milieu(x2, x1) {
    9: x4 = x2
    8: x3 = x1
    7: nop
    6: if (x4 <s x3) goto 5 else goto 2
    5: x4 = x4 + 1 (int)
    4: x3 = x3 + -1 (int)
    3: goto 6
    2: x5 = 0
    1: return x5
}
```

FIGURE 3 – Display of the representation RTL of "middle.c" during its compilation by CompCert C

(BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH))$\longrightarrow$ we have $100_2 = 4_10$, for x4
(BinNums.Coq_xO (BinNums.Coq_xO (BinNums.Coq_xO BinNums.Coq_xH))))) $\longrightarrow$ we have $1000_2 = 8_10$, 8 for the next instruction

Implementation of a first structure of CFG

The compilation was a bit complicated but once the structure of the compiled code by CompCert was extracted, we obtain a list of the Control Flow Graphs, each one represents a function/procedure.

By sorting and modifying the tree instructions we have implemented a first new structure that is better suited to the representation of a CFG, a structure by triplet.

Once our starting tree transformed into a CFG as a triplet structure, we have developed a printer in the format .dot allowing the display of a graph using the graphviz software [Ellson *et al.*, 2003].

Generation of Logical Formulas

Once the CFG recovered, we can start to generate the logical formulas, each one independently, by following Hoare rules. The formulas obtained for our program are the following :

$$\forall x_1, x_2, x_3, x_4, x_5,$$

precond(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P9(x_1, x_2, x_3, x_2, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$
P9(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P8(x_1, x_2, x_3, x_2, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$
P8(x$_1, x_2, x_3, x_4, x_5) \Rightarrow P7(x_1, x_2, x_1, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

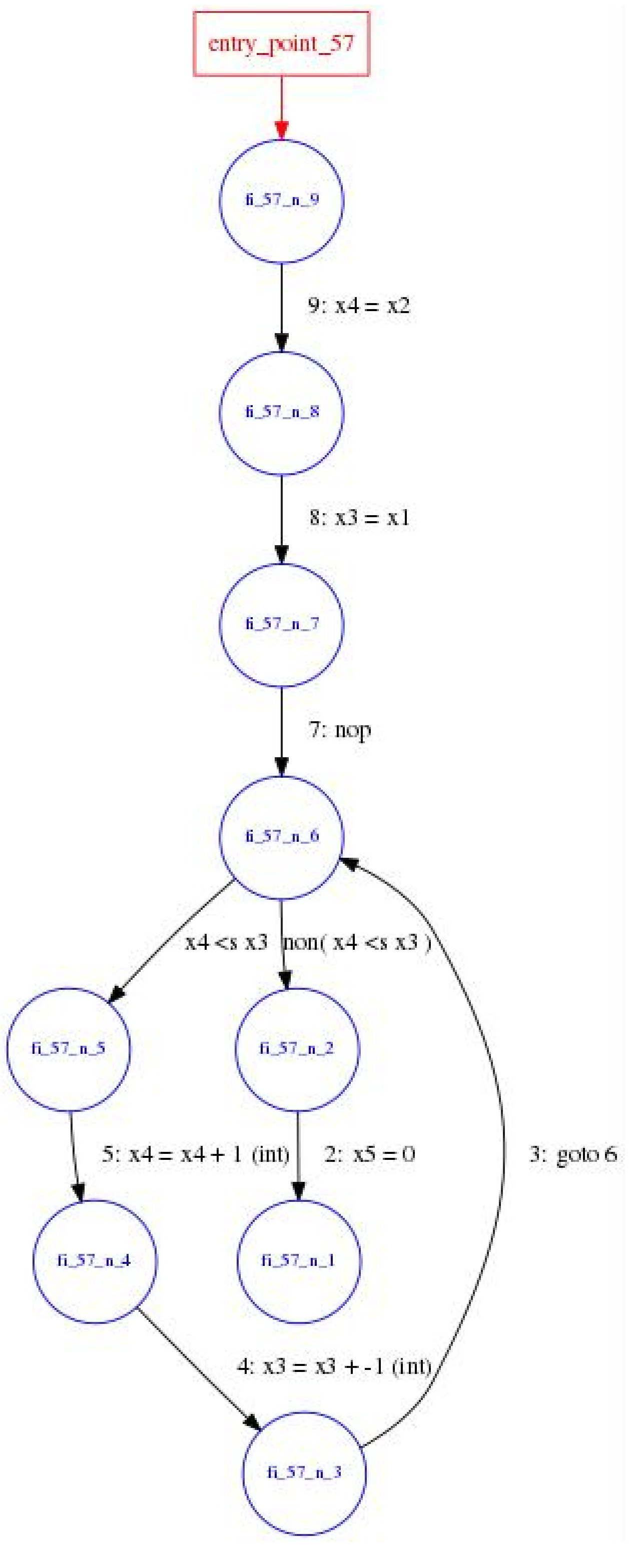


FIGURE 4 – Control Flow Graph of the program "middle.c"

$\mathrm{P7}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow P6(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P6}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \wedge (x_4 < x_7) \Rightarrow P5(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P6}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \wedge \neg (x_4 < x_7) \Rightarrow P2(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P5}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow P4(x_1, x_2, x_3, x_4 + 1, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P4}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow P3(x_1, x_2, x_3 - 1, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P3}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow P6(x_1, x_2, x_3, x_4, x_5)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P2}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow P1(x_1, x_2, x_3, x_4, 0)$
$\forall x_1, x_2, x_3, x_4, x_5,$

$\mathrm{P1}(\mathrm{x}_1, x_2, x_3, x_4, x_5) \Rightarrow postcond(x_1, x_2, x_3, x_2, x_5)$

These are the formulas (simplified) resulting of the CFG ref. Figure 4 Pi $\Rightarrow$ Pj indicates that the node i is pointing at the node j and the transformation is done in the parameters of Pj.

The choice of the SMT Solver

To resolve the formulas that we have generated, we can choose between many SMT solvers to do that task. However, since we are working on a project similar to Gurfinkel's that uses Z3 and works just fine, we naturally chose to simply use the same SMT solver.

Adaption of the logical formulas to Z3

Once the formulas generated, we need to transform them so that it suits Z3. A function definition (predicate) in Z3 is defined as follows :

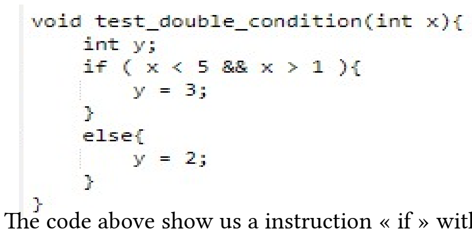
(declare-fun((type_var1) (type_var2)... (type_varN)) Bool)

Here we have a predicate declaration (Bool) with N variables.

To declare Z3 predicates, we have to get the types of all the variables of the program.

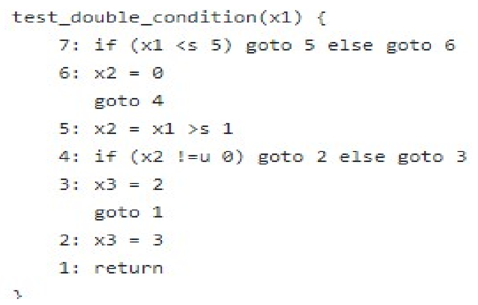
However, the language RTL is close to the Assembler language and manipulates pseudo-registers. Therefore the variables (or pseudo-registers) are not typed.

To find the types of the variables in RTL, we have to rebuild the types using the operations in which each variable participated.



```
void test_double_condition(int x){
     int y;
     if ( x < 5 && x > 1 ){
         y = 3;
     }
     else{
         y = 2;
     }
}
```

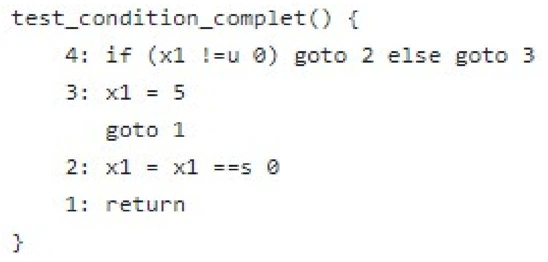
The code above show us a instruction « if » with two conditions.
Below we can see the RTL traduction



```
test_double_condition(x1) {
    7: if (x1 <s 5) goto 5 else goto 6
    6: x2 = 0
       goto 4
    5: x2 = x1 >s 1
    4: if (x2 !=u 0) goto 2 else goto 3
    3: x3 = 2
       goto 1
    2: x3 = 3
    1: return
}
```

A new variable has been introduce: x2. It is the variable making the link between the two conditions. x2 is set to 0 (false) if the first condition is'nt verify or set at the value of the second condition. This is a boolean.

FIGURE 5 –



```
test_condition_complet() {
    4: if (x1 !=u 0) goto 2 else goto 3
    3: x1 = 5
       goto 1
    2: x1 = x1 ==s 0
    1: return
}
```

In this case :
- at the line 4 we could say than x1 is a boolean
- at the line 2 we can say the x1 is a boolean
- at the line 3 we can say x1 is an integer

FIGURE 6 –

x3 = x3 + -1 (int)
from this operation we can deduce
that x3 is an integer variable

This is possible thanks to the multitude of possible operations defined in the RTL representation of CompCert which make it possible to find the types.

However to have a better use of Z3 we could decide to use Booleans instead of the variables that only realize conditions. Particularly useful on intermediate registers added by the compiler making the link between 2 conditions (as defined in the figure 5.

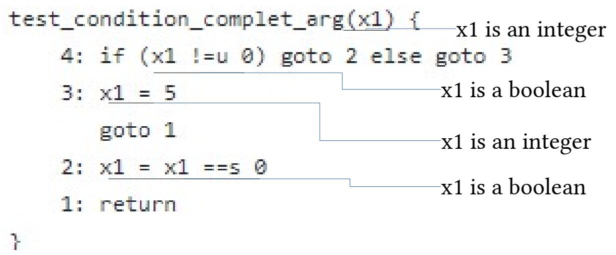
It is nevertheless possible for an integer to be used as a boolean temporarily (as defined in Figure 6)

We have done a hierarchy of types $Min < Bool < Int < Float$. to proceed afterwards to a homogenization of the types by creating a link between all the instructions of the program. After that, the types of the arguments given to the function must be compared. It's crucial that everything remains consistent. (as defined in Figure 7).

Thereby, we can find the types of all the program's variables.

In our example, it is simple because as it turns out there are integers only. (voir Figure 8).

Give an example of 2 functions definitions and the logical formula that makes the link between them. In our case, we will write the formula that draws the link between the lines 9 and 8 :

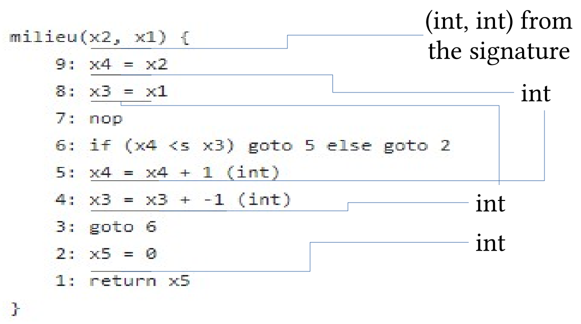


```
test_condition_complet_arg(x1) {        x1 is an integer
    4: if (x1 !=u 0) goto 2 else goto 3
    3: x1 = 5                            x1 is a boolean
       goto 1                            x1 is an integer
    2: x1 = x1 ==s 0                     x1 is a boolean
    1: return
}
```

In this case we have 4 indications for the type of x1, and to deduce the type we are looking for the max of thos 4 indications.

Max(int, bool, int, bool) = int
Our variable x1 is an integer.

FIGURE 7 –



```
milieu(x2, x1) {
    9: x4 = x2
    8: x3 = x1
    7: nop
    6: if (x4 <s x3) goto 5 else goto 2
    5: x4 = x4 + 1 (int)
    4: x3 = x3 + -1 (int)
    3: goto 6
    2: x5 = 0
    1: return x5
}
```

Therefore x1,x2,x3,x4,x5 are integer variables

FIGURE 8 –

$$\forall\, x_1, x_2, x_3, x_4, x_5,$$

P9(x_1, x_2, x_3, x_4, x_5) $\Rightarrow$ $P8(x_1, x_2, x_3, x_2, x_5)$

par

```
(declare-fun P_9 (Int Int Int Int Int ) Bool)
(declare-fun P_8 (Int Int Int Int Int ) Bool)

(forall ( (x1 Int ) (x2 Int ) (x3 Int ) (x4 Int ) (x5 Int ) )
  (⇒
    (P_9 x1 x2 x3 x4 x5)
    (P_8 x1 x2 x3 x2 x5) ; from x4 = x2
  )
)
```

Once the types are determined, we write the predicates as well as the logical formulas associated to the program while transcribing solely our formulas into the format Z3.

Adding of the pre/post condition

Whatever concerns the C code formulas is therefore written automatically.

We need to define thereafter the true properties at the beginning of the program (pre-condition) and the ones that we wish to prove at the end of the program (post-condition).

The writing of the pre-condition and the post-condition is left to the developer because the properties that he wishes to show are the ones that should be written. The links between the pre-condition od the program and the post-condition are written automatically, only the bodies of the pre-condition and the post-condition should be written.

Interpretation of Z3 results

When everything is written, it only remains to launch Z3 and wait for the result. With the way the formulas have been written, if Z3 responds with sat then the program is proven. It's not proven if it responds with unsat.

We would have preferred having more information when Z3 responds with unsat, unfortunately we couldn't dig further in the returns of Z3. Get-proof returns a proof that there is no way to satisfy the formulas but it is illegible.

Z3 can also respond with unknown or timeout. And in those cases, we don't have further information.

We remain very dependent on the solver. If it can't solve our formulas, even if they are correct, we need another way to proceed.

Observation on Z3

The pre-condition has a significant impact on the response of Z3.

We fix the properties $0 \leq x \leq y \leq parameter$ We have to put the results into perspective since the tests weren't completely under control. Realized on Z3 online (https ://rise4fun.com/Z3) , based on the formulas defined earlier (written in Z3).

parameter	time
3	2
5	3
7	7
8	7.7
9	timeout (12)
10 and +	timeout

Given that the experiment was not quite normalised, I was not able to go further in an estimation in function on the size of the parameters, however it is obvious that the bigger the space of the pre-condition, the longer the response will be.

If Z3 give us a time-out, we can restrain our pre-condition space to do local verification. That can give us indication about the program completeness but that don't prove the completeness.

4 Conclusion

4.1 Prove a small program

It was possible to prove a small program. We have also proven that if we did a small modification $\leq$. the program was no longer valid. Since the proof is annoying to do by hand, we can consider that the internship mission is accomplished.

4.2 The representation RTL

The real goal of the internship was to check if the representation RTL had the necessary information to realize this verifier. During this internship, I haven't had time to get into the details of all the RTL language instructions, however, based on what i have done I estimate that there are enough information to implement the program verifier from this representation. It will not be inevitably immediate but the necessary information are there.

+ : . The CFG
. The functions have a lot of information
. Possibility of finding the unavailable information

- : . Not everything is straight forward, there is some work that has to be done.

5 Possible and likely extensions

5.1 Addition of a property to test in the program

The idea is to replace the functions calls assume() and assert() of the C source code by some properties to verify in the program, exactly where there are functions calls.

Try to recover information of SMT solver when it returns unsat to see which properties remain unverified.

Apply the ideas of other projects that are aiming to prove code. Some of them are in the bibliography [Temesghen kahsai, 2015].

5.2 Mid-term objective

It is necessary to eventually complete this tool to treat all the RTL language instructions. By doing so, my work will be completed and a verdict on the relevance of the intermediary RTL representation for the realization of a verifier will be reached.

If it is not possible to treat certain instructions from the RTL representation then the tool will not be usable. It may be necessary to transmit other information from the previous steps into the compilation. If all this is done, the tool will be the equivalent of Seahorn or Boogie, adapted to the CompCert C compiler.

5.3 To be synchronized with CompCert

To remain in the same logic as CompCert, we have to prove this software, the same way as CompCert and develop this checker (verifier) in Coq, which is, according to Mr. Périn, would be the work of a thesis.

If the project came to an end, then a program compiled by CompCert C (and our tool) would be able to have properties, and maintain them throughout the compilation.

Références

[Blazy and Leroy, 2005] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[de Moura and Bjørner, 2007] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. *21st International Conference on Automated Deduction, Bremen, Germany,*, volume 4603 of Lecture Notes in Computer Science :183–198, July 2007.

[Ellson *et al.*, 2003] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[Fan, 2010] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[Gurfinkel *et al.*, 2015] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. *27th International Conference on Computer Aided Verification (CAV 2015); 18-24 Jul. 2015; San Francisco, CA; United States*, pages 1–17, 2015.

[Temesghen kahsai, 2015] Dejan Jovanovic Martin Schäf Temesghen kahsai, Jorge A. Navas. finding inconsistencies in programs with loops. pages 1–15, 2015.

[Yang *et al.*, 2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *Proceedings of the 2011 ACM SIGPLAN Conference PLDI, San Jose*, pages 1–17, June 2011.