# Combining two formal methods of the static analyses
# Master of Science Thesis

Jorge Luis Honorat Poblette

Universitat Politcnica de Catalunya (UPC)
Institut de Recherche en Informatique de Toulouse
&
ONERA The French Aeroespace Lab

Advisors
Xavier Thirioux (IRIT)
Pierre-Loc Garoche (Onera)

Barcelona, 2012

**Abstract**

With the widespread use of embedded systems, containing critical software parts, validation of such software now plays an important role. Static analysis allows to validate these systems by considering exhaustively all their behaviors. Among the different formal methods dedicated to the analysis of programs, we focus here on two methods: abstract interpretation [3] and the use of weakest precondition Floyd-Hoare [4]. The first method relies on a user-provided abstract representation of the data to compute, from which, in a fully automatic manner, properties are deduced with respect to the analyzed program. The second method needs the first one to enrich the program with the specification of the properties to be validated. Automatic provers, such as SMT-solvers, or proof assistants, such as Coq, could then evaluate these proof obligations and warrant their satisfiability.

The Frama-C platform [2] Framework for Modular analysis of C allows to federate multiple approaches and multiple tools on the same common language of properties: ACSL (ANSI/ISO C Specifica- tion Language) [1]. The context of this Master thesis is to study the possible links between properties expressed and computed within the abstract domains of abstract interpretation and the computation of weakest preconditions. The main objective is to help discharging complex proof obligations by combining the aforementioned methods.

# Contents

# 1 INTRODUCTION

According to Moore law which explains that the number of transistors is doubling every two years, we can ensure that more powerful hardware will need more powerful software. This is the case of embedded systems whose main feature is to handle a particular task. Embedded systems have been evolving in the last years, optimizing the size as well as their performance. More complex programs are developed and used in many critical embedded systems over the years. Those systems execute essential tasks that are not allowed to fail. They are mainly used in important areas such as aeronautics, space investigation, weather forecast, nuclear power plants, medical area among others. Nowadays the necessity of analyze the software these systems use is very important to have accurate information. That is to say, validating a program using formal methods could be very helpful in order to detect possible coding errors.

Today some tools as static program analysis, model checking, deductive methods and abstract interpretation are used to ensure that the execution of a program is correct. These last two tools, deductive methods and abstract interpretation are used independently of each other. The purpose of this document is to prove both of them can be used simultaneously, getting as a result a more efficient software analysis tool.

## 1.1 THESIS OBJECTIVE

Given the background in software and hardware evolution over the years as well as the demand of accurate information in different industrial areas is how this Master thesis born. Having as main objective helping to discharging complex proof obligations by combining the aforementioned methods to improve the precision in the software analysis.

This thesis is a continuation of the investigations already begun by the professors Xavier Thirioux form the IRIT and Pierre-Loc Garoche from Onera labs in Toulouse, France.

Motivated to create a more reliable tool in the static analysis this thesis arises. It should be recalled that the static analysis is used today for the validation of critical software parts, contained in embedded systems in areas as aeronautics, nuclear industry or even medical. It is for this reason that the validation of the information is so important.

As the title of the thesis suggests the goal is to combine two formal methods used in the static analysis, the first of these called *Abstract interpretation* and the second *Weakest precondition*. At the time of the realization of this thesis both methods are used independently. The thesis is developed through the help of a software which implements these formal methods.

# 2 METHODS BRIEF INTRODUCTION

Abstract interpretation is a theory of the approximation of semantics of (programming or specification) languages. It formalizes the idea that the semantics can be more or less precise according to the considered level of observation [3]. If the approximation is coarse enough, the abstraction of a semantics yields a less precise but computable version. Because of the corresponding loss of information, not all questions can be answered, but all answers given by the effective computation of the approximate semantics are always correct.

The abstract interpretation is a constructive theory of the approximation of fixed points of monotone operators in a lattice (partially ordered set). Figure one shows the representation of the construction of an abstraction between two lattices. Between the components it is possible to find $\alpha$ which represents the abstraction function, $\gamma$ represents the concrete function, $\top$ represents the top element in each lattice, $\perp$ represents the bottom element and fp refers to the fixed points (which refers to the points that are mapped to themselves by the function) in each lattice.

Every abstract interpretation requires a defined collection of abstract values. An abstract value is just a set of concrete values. The representation shows how it is possible to move through the concrete function to the abstraction function and inverse sense as well. The way in which the two sets are related is by defining total functions that match elements from one to the other. The concrete set can be differentiated from the abstract set by the # symbol.

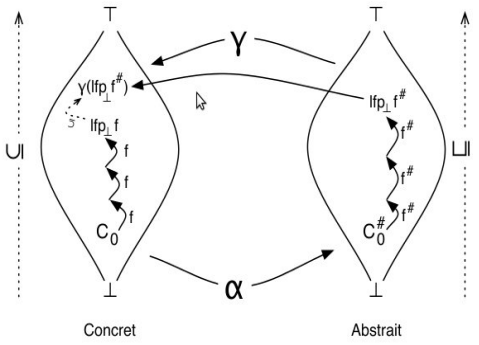


Figure 1: *Shows the construction of a correct abstraction between two lattices.*

## 2.1 WEAKEST PRECONDITION

The goal of Hoare logic is to provide a formal system for reasoning about program correctness. Hoare logic is based on the idea of comparing a specification as a contract between the implementation of a function and its clients. The specification is made up of a pre-condition and a post-condition. The pre-condition is a predicate describing the condition in which the function relies on for a correct operation; the client must fulfill this condition. The post-condition is a predicate describing the condition in which the function establishes after correctly running; the client can rely on this condition being true after the call to the function.

The validity of the results of a program (or part of a program) depends on the values taken by the variables before the program is initiated. The connection between a precondition (P), a program (Q) and the result of its execution (R) is written as:

$$P \{Q\} R$$

The interpretation of the expression about it is as follows: if the assertion P is true before the initiation of the program Q, then the assertion R will be true on its completion. If there are no preconditions composed, then we have:

$$\text{true} \{Q\} R$$

## 2.2 DIJKSTRA WEAKEST PRECONDITION

The way in which predicates are used (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by Hoare. The main difference has been that things have been tightened up a bit: while Hoare introduces sufficient pre-conditions such that the mechanism will not produce the wrong result (but may fail to terminate), it shall be introduced the terms necessary and sufficient, in other words so-called weakest pre-conditions such that the mechanisms are guaranteed

to produce the correct result [5]. Dijkstra uses the notation WP (S, R), in which S refers to a statement list, R is the condition on the state of the system, WP in this case refers to the predicate transformer.

The next properties were defined:

For any S, we have for all states:

WP(S,F)=F

For any S and any two post-conditions:

WP(S,P) $\Rightarrow$ WP(S,Q)

For any S and any two post-conditions P and Q

( WP(S,P) and WP(S,Q) ) = WP(S,P and Q)

For any deterministic S and any post-conditions P and Q

WP(S,P) or WP(S,Q)= WP(S,Q or P)

# 3 TOOLS INTRODUCTION

Now that a brief explanation of the methods that are intended to be used in this thesis has been made, in this section a brief explanation of the tools used for the project is going to be found. It is important to have an idea of how the tools work in order to understand how the formal methods are applied on them and how the thesis is being developed.

## 3.1 FRAMA-C

Frama-C is the platform where all the test have been made. The principal reason why this platform has been used is because Frama-C is a suite of tools dedicated to the analysis of the source code of software written in C. Frama-C

gathers several static analysis techniques in a single collaborative framework [6]. The collaborative approach of Frama-C allows static analyzers to build upon the results already computed by other analyzers in the framework. Thanks to this approach, Frama-C provides sophisticated tools, such as a slicer and dependency analysis.

## 3.2 IMPORTANCE OF VALUE PLUG-IN 20100401 BORON

The Value plug-in is important for this thesis given that from its execution it is possible to get computed values which will serve as preconditions later. In other words this plug-in is the implementation of the abstract interpretation. This plug-in does the partial execution of a program in order to extract the information of its execution without having to perform all calculations. The Value analysis plug-in automatically computes sets of possible values for the variables of the program. Synthetic information about each analyzed function can then be computed automatically from the values provided by the value analysis [6]. The value plug-in can be used in two modes terminal mode shown in figure 2, and graphic mode shown in figure 3. For this thesis purposes it was used the command line mode.

## 3.3 VALUE PLUG-IN COMMAND LINE OPTIONS

The value analysis plug-in automatically computes sets of possible values for the variables of the program. Synthetic information about each analyzed function can then be computed automatically from the values provided by the value analysis [6].

Some of the commands that can be execute from the terminal valid for the value plug-in are shown below:

```
-slevel : Indicates that the analyzer is allowed to separate,
          in each point of the analyzed code, up to n states
          from different execution paths before starting to
          compute the unions of said states.
```

-context-depth : It may also be observed for non-recursive types if they are deep enough. The limit can be changed.

-context-widht : A pointer type is assumed to point at the beginning of an array of two elements. This number can be changed.

-context-valid-pointers : Causes the pointers to be assumed to be valid (and NULL therefore to be omitted from the possible values) at depths that are less than the context width.

-val : The value analysis plug-in is able to guarantee that at each passage through the return statement of function f.

-warn-unespecified-order : If set, warns for side effects occuring in unspecified order.

-mem-exec f : The analyzer consequently analyzes the function f a single time in a context created to be as general as possible, and the obtained results will later be reused each time a call to f is encountered in the actual analysis of the application. This option has not been intensively tested, and should be considered as experimental at this point.

-klr : Keep only last run of value analysis.

-propagate-top : Do not stop value analysis even if state is degenerating.

-wlevel : Set n number of iterations before widening. Default to 3.

-plevel : Use n as the precision level for arrays accesses. Array accesses are precise as long as the interval for the index contains less than n values.

As the purpose of the thesis is aimed to the validation of programs written in C code, examples coded in C language are going to be used. In the next figures it is possible to see the execution of the Frama-C program calling the Value analysis plug-in for the first code example. It is pretended to extract all the possible computed values of the *unimportantfunction* called inside the main. Figure three is showing the graphic mode of the execution of the Value analysis plug-in.

In the figure two it is possible to see the results after executing the Value analysis, including a group of elements for the global variables $t$ and $i$ defined in the line one of the C code shown below. It can be clearly seen how after running the value analysis which uses the abstract interpretation technique, the result gives us all the possible values for $t$ between 0 and 199 and for $i$ all the elements from 0 to 200.

Code of *function.c* file:

```
1  int t[200], i;
2  void unimportant_function(void)
3  {
4          for(i=0;i<200;i++)
5            t[i]=i;
6  }
7
8  int main()
9  {
10         unimportant_function();
11
12         return t[143];
13 }
```

After executing the value analysis plug-in we obtain the next result.

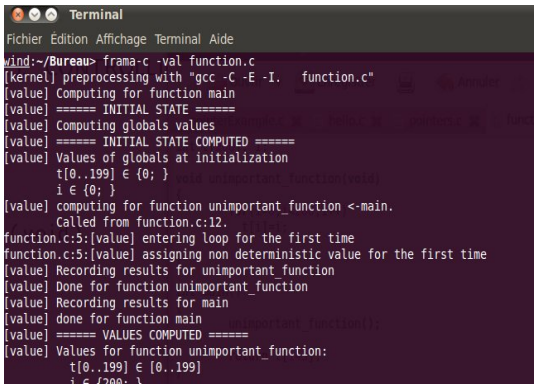


Figure 2: *Value analysis plug-in execution in terminal using a C file named function.c.*

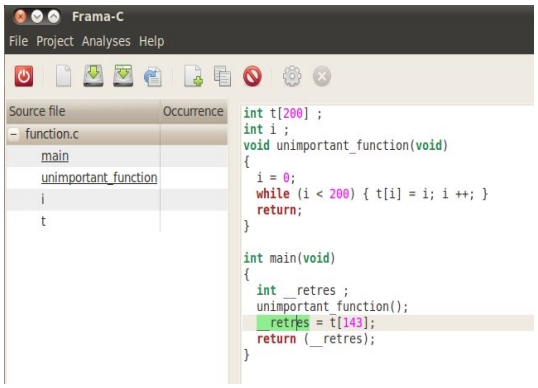


Figure 3: *Value analysis plug-in execution in graphic mode using a C file named function.c.*

## 3.4 IMPORTANCE OF JESSIE PLUG-IN

The Jessie plug-in is important for the thesis given that it is the implementation of the weakest precondition. The preconditions are going to be injected using ACSL (explained in next section). In other words the Jessie plug-in allows to perform deductive verification of C programs inside Frama-C. The C file possibly annotated in ACSL is first checked for syntax errors by Frama-C core, before it is translated to various intermediate languages inside the Why Platform embedded in Frama-C, and finally verification conditions (VC) are generated and a prove is called on these [7]. In other words, Jessie plug-in proves that C functions satisfy their specifications as expressed in ACSL. These proofs are modular: the

specifications of the called functions are used to establish the proof without looking at their code.

## 3.5 JESSIE PLUG-IN COMMAND LINE OPTIONS

Some of the Jessie plug-in commands that can be used from the terminal are shown below:

```
-jessie activates the plugin, to perform C to Jessie
 translation
-jessie-project-name=<s> specify project name for Jessie
 analysis
-jessie-atp=<s> do not launch the GUI but run specified
 automated theorem prover in batch (among alt-ergo, cvc3,
 simplify,yices, z3),or just generate the verification
 conditions (goals)
-jessie-cpu-limit=<i> set the time limit in sec. for proving
 each VC. Only works when -jessie-atp is set.
-jessie-behavior=<s> restrict verification to the given
 behavior(safety, default or a user-defined behavior)
-jessie-std-stubs (obsolete) use annotated standard headers
-jessie-hint-level=<i> level of hints, i.e. assertions to
 help the proof (e.g. for string usage)
-jessie-infer-annot=<s> infer function annotations
 (inv, pre, spre, wpre)
-jessie-abstract-domain=<s> use specified abstract domain
 (box, oct or poly)
-jessie-jc-opt=<s> give an option to the jessie tool
 (e.g., -trust-ai)
-jessie-why-opt=<s> give an option to Why (e.g., -fast-wp)
```

Using the same C code example as for the Value plug-in, figure four shows the execution of the Jessie plug-in in graphic mode. It is possible to see how the Alt-ergo (an automatic theorem prover dedicated to program verification) is enabled in order to validate the proofs marked in green and the ones that are not valid marked with a red x, also some unknown proofs are marked with the ? Symbol.

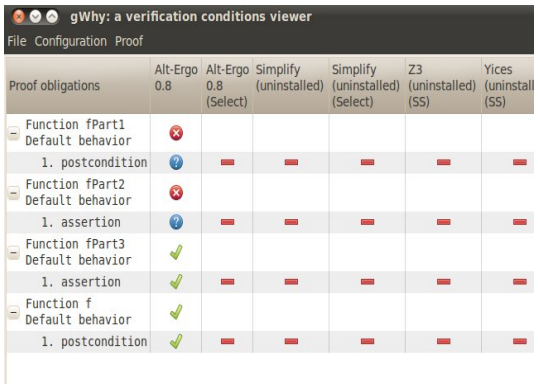


Figure 4: *Jessie analysis execution in graphic mode.*

## 3.6 ACSL: ANSI/ISO C SPECIFICATION LANGUAGE

ACSL is a formal language. This means that the specifications written in ACSL can be automatically manipulated by helper programs, in the same way that a programming language is a formal language manipulated by a compiler, and by opposition to informally written comments that can only be useful to humans [1]. The annotations can be separated in two main groups. Both global annotations and statement annotations are described below.

Global annotations:

function contract : such an annotation is inserted just before the declaration or the definition of a function.
global invariant : this is allowed at the level of global declarations.
type invariant : this allows to declare both structure or union invariants, and invariants on type names introduced by typedef.
logic specifications : definitions of logic functions or predicates, lemmas, axiomatizations by declaration of new logic types, logic functions, predicates with axioms they satisfy. Such an annotation is placed at the level of global declarations.

Statement annotations:

assert clause : allowed just before a block closing brace.

loop annotation (invariant, variant, assign clauses): is allowed immediately before a loop statement: for , while , do . . . while .
statement contract : very similar to a function contract, and placed before a statement or a block. Semantical conditions must be checked (no goto going inside, no goto going outside).
ghost code : regular C code, only visible from the specifications, that is only allowed to modify ghost variables. This includes ghost braces for enclosing blocks.

For the purpose of the thesis, ACSL is going to be used by the Jessie plug-in in order to generate the proofs we want to validate. The annotations used until this point in the code programs are the next ones: *assumes*, *asserts* and *loopinvariants*. It is possible to see the examples of how these annotations were used in the approaches sections.

# 4 PROJECT ORGANIZATION

Now that it has been shown in sections two and three a brief explanation of the tools and previously a brief explanation of the formal methods. In this section it will be explained how it is intended to link the abstract interpretation and the weakest precondition in order to improve the software analysis.

A graphical interpretation of the objective can be seen in figure 5. It can be seen that it is pretended to extract data from a C code execution using the abstract interpretation technique, which is applied by the Value-analysis. Then, it is pretended to inject this data which would serve as preconditions into the Jessie tool which applies the weakest precondition technique using ACSL notations. The Jessie plug-in will translate the code into an intermediate language inside the *Why* platform. These files generated in why are the ones going to be manipulated later in order to check that the precondition is met.

The arrow in the figure shown below represents the execution of a program and the action of stopping the execution at one point, in order to inject the data previously extracted by the Value analysis at this point into the Jessie-why analysis (being more specific in the *.why* files generated by the embedded platform inside the Jessie plug-in). This arrow in the middle shows this process is pretended to be done also in the inverse sense.

If this is achieved, we can generate properties to be validated getting as a result a more reliable code analysis and as a consequence help discharging complex proof obligations by combining the aforementioned methods.

The whole process idea is to go from a pre-condition to a post-condition. The A = WP(C2,POST) formula represents Dijkstra notation mentioned in section 2.2, in which C2 refers to the statement list, in this case the data being injected from the value analysis into the code using ACSL notations. A' represent the value true that is got after executing the weakest pre-condition which is using the ACSL notations as preconditions, this A' represents a valid statement that then can be used in the other sense.

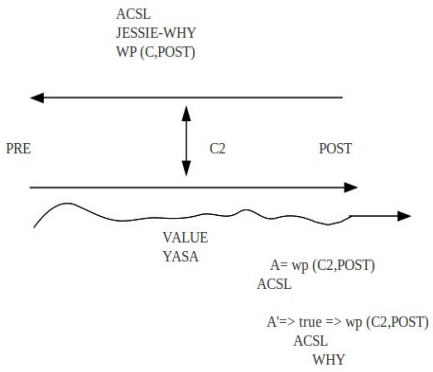


Figure 5: *Graphic Representation of the goal (link both methods) showing iteration between Value Analysis and Jessie-Why.*

## 4.1 APPROACHES

### FIRST APPROACH

## 4.2 INJECT DATA IN THE JESSIE-WHY DECLARING PREDICATES WITH ACSL

This first goal approach is to be sure if it is going to be possible to inject the data in the Jessie-why files, in order to create the connection between the Value analysis and the Jessie plug-in. The files generated by the Jessie-why can be found inside a folder that is created after executing the Jessie tool. These files are generated with a *.why* extension. It is important to be sure that it is possible to modify the *.why* files even at this point manually since these files contain the objectives wanted to be proved.

The C code example shown below which contains a function called $functionf$, this function can be seen between line 1 and line 6 of the code. As the goal is to apply this methodology at any point of a C code, the first thing to do is to separate the function in different places where we want to apply the analysis.

With this example it is possible to see how the function f () is separated at three points (line 3, line 4 and line 5 of the code) in order to generate what is called the part 1 (line 8), part 2 (line 12) and part 3 (line 17) as shown below:

```
1  int f() {
2    int x, y;
3    x = 1;
4    y = x;
5    return y;
6  }
7
8  int f_part1( int y) {
9    return y;
10 }
11
12 int f_part2() {
13   int x, y;
14   y = x;
15 }
16
17 int f part3() {
18
19   int x, y;
20   x = 1;
21 }
```

The preconditions need to be added with the help of the ACSL annotations. For example between lines 4 and 9 of the code below it is possible to see how the annotations in ACSL (preconditions) are added manually. This was done after running the value analysis which gives us back the computed values from each variable at the given point. Doing this it is intended to generate the predicate definition that is going to be proved by the Jessie-why plug-in later:

```
1  int glob;
2  int glob2;
3
4  /*@ predicate my_pred (int x) = (x==1); */
5
6  // Main function
7  /*@ requires \true;
8    @ ensures \result == 1;
9  */
10 int f() {
11   int x, y;
12   x = 1;
13   y = x;
```

```
14    return y;
15  }
16
17  // 1st suffix
18  //@ ensures \result == 1;
19  int f_part1( int y) {
20    return y;
21  }
22
23
24  //@ predicate suffix1 (int x) = (x == 1);
25  // 2nd suffix
26  /*@ requires \true;
27   */
28  int f_part2() {
29    int x, y;
30    y = x;
31   //@ assert suffix1(y);
32  }
33
34
35  //@ predicate suffix2 (int x, int y) = (x == 1);
36  // Last suffix
37  /*@ requires \true;
38  */
39  int f_part3() {
40    int x, y;
41    x = 1;
42   //@ assert suffix2(x, y);
43  }
```

The predicates on lines 4, 24 and 35 are used to declare the local variables of each function and its value using ACSL annotations. Asserts are used on lines 31 and 42 to prove the predicates declared on them. As the *.why* files contain too much information in a particular syntax one dummy predicate was also defined at line 4, in order to locate in which point it is needed to start modifying the file.

After running the Jessie plug-in which proves that C functions satisfy their specifications as expressed in ACSL. This *.why* files generated contain the goals of the three separations generated in the C code function shown before. The Jessie plug-in generates one file for each section.

It was important to detect inside the files generated if the values of the local variables generated by the Jessie plug-in were computed values given it uses

a particular syntax. In case of being true, it is possible to insert data coming from value analysis further on. The code below shows the syntax generated by the plug-in as well as the content of the first section (goal) separated before in the C code example. Some modifications need to be done in order to define the predicate that is going to be applied later.

Before modifying:

```
1 goal f_part1_ensures_default_po_1:
2   forall y:int32.
3   ("JC_12": true) ->
4   forall return:int32.
5   (return = y) ->
6   ("JC_13": (integer_of_int32(return) = 1))
```

After modifying:

```
1 predicate suffix1 (y : int32) =
2   forall y_0:int32.
3   ("JC_12": y = y_0) ->
4   forall return:int32.
5   (return = y_0) ->
6   ("JC_13": (integer_of_int32(return) = 1))
```

If we look at the code *before modifying* the line 6 shows the plug-in generates computed values and also that the Jessie-why tool generates different variables from those originally defined (different names). Given these facts the next thing to be done is to try to match the variables (defined originally in the predicates in the C code using ACSL) with the ones generated by the tool.

Looking at the code after modifying in order to match the variables generated by the plug-in with the ones defined in the ACSL notations, the following modifications were done. On the line 1 of the code after being modified it was defined the predicate with its parameters. Later on at lines 2, 3 and 5 the variables defined by Jessie plug-in were joined with the ones originally declared.

It is possible to see how it is declared parameter $y$ as an integer for the suffix1 on line 1. It is matched the $y$ with $y\_0$ on line 3 as well as it substitutes it on line 2 and 5. The parameter $y\_0$ was the name defined by the Jessie analysis and which computed value was 1.

After linking the variables in the goals, it is possible then to concatenate all the *.why* files (files generated by each separate section in the function) into one

single file in order to run the Jessie-why analysis and see if it could generate a valid proof.

Before running the analysis, it was necessary to remove the dummy predicate defined on line 4 of the C code example. Given it was used just to see the exact place where the *.why* file can be modified.

Finally, it is possible to run the Jessie-why analysis over the last file generated containing the three files created and modified as explained before. A valid result will prove it is possible to modify the Jessie-why files inserting preconditions defined by us. During this process a shell script could be created in order to make all the process mentioned above faster and to be able to test it with different C codes. Below it is possible to see the result obtained after executing the Jessie-why analysis.

```
total    :   1
valid    :   1 (100%)
invalid  :   0 (  0%)
unknown  :   0 (  0%)
timeout  :   0 (  0%)
failure  :   0 (  0%)
total wallclock time : 0.26 sec
total CPU time       : 0.03 sec
valid VCs:
    average CPU time : 0.03
    max CPU time     : 0.03
invalid VCs:
    average CPU time : -nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : -nan
    max CPU time     : 0.00
```

After running the Jessie-why analysis for this approach a valid result was obtained, this means the predicate which was generated after using ACSL notations (which data comes from the value analysis) in our example function and modified later in order to run the validation of this precondition works. In other words, we are combining the abstract interpretation and the weakest precondition manually.

With this approach it is possible to see that the *.why* files can be manipulated. This let us be sure it was possible to achieve one of the important steps

of the thesis that is to inject data that will come from value analysis which applies abstract interpretation technique in the Jessie analysis which applies the weakest precondition technique.

## 4.3 PROBLEMS FACED WITH THE FIRST APPROACH

Automating the process in a shell script is not so simple. Given that when it is tried to get the number of parameters in each function and the type of each parameter, the process requires using complex *grep* and *sed* instructions.

Another problem faced was that this example shows a simple C code function without many variables, this means linking the variables in a bigger function could be complicated. Sometimes the variables generated by the Jessie analysis are not easy to follow and match in all the file. This means with more complex C codes, the Jessie plug-in generates for a parameter $y$ variables as $y\_0$, $y\_0\_0$ and so on in different parts of the *.why* file, causing it is easy to miss and to match one variable in the predicate definition. As consequence we could easily make a mistake and not get a valid proof when it is applied in other goal.

**SECOND APPROACH**

## 4.4 INJECT DATA IN USING ACSL ASSERT ANNOTATIONS TO AVOID REPLACE VARIABLES MANUALLY IN THE JESSIE-WHY ANALYSIS

In this approach, instead of matching variables in the *.why* files generated by the Jessie tool, it is intended to inject the data directly in the C code using the ACSL assert annotations. The objective of doing this is to avoid doing mistakes at the moment of matching variables in the *.why* file. This approach is explained below.

The first thing done is to obtain the data intended to inject in the *.why* files. This is obtained from our original C code after executing the *valuewrapper* tool. The value wrapper is just a plug-in created that works in parallel with the value analysis plug-in. This tool helps us obtain the data of each variable at each line of the C code using the abstract interpretation technique. This was done with the aim to get the values computed at each line of our C code so we can easily split our code in the line wanted.

The next C code is going to be used in order to explain how it is pretended to inject the data directly in the code using the ACSL annotations instead of linking them manually. As the approach explained before, we split a function of the C code at any point. In the code shown below it is possible to see on line 27 part 2 and on line 33 part 1. These are the new functions defined. This original function g () is split at line 10.

```
1  #define assumes(P) if (P) {} else miracle()
2
3  void g() {
4    int x,y,z,w;
5    x = 1;
6    y = 2;
7    z = 0;
8    w = 5;
9    z = y + y;
10   w = z + w;
11   z = x + y + w;
12 }
13
14 // miracle function
15 // @ requires \true;
16 // @ ensures \false;
17 void miracle() {
18  while (1);
19  return;
20 }
21
22 int x,y,z,w;
23
24 //@ predicate suffix1 = x && y && z && w;
25
26 // 1st suffix
27 int g_part2() {
28   //@ assert suffix1;
29     z = x + y + w;
30   //@ assert (z == 12);
31 }
32
33 int g_part1() {
34   x = 1;
35   y = 2;
36   z = 0;
37   w = 5;
38   z = y + y;
39   w = z + w;
40 assumes (w == 9);
41 //@ assert suffix1;
42 }
43
44 void main(){
45     g();
46 }
```

We can see on line 24 the declaration of the predicate, but in this case inside the predicate the parameters $x$, $y$, $z$ and $w$ are already declared directly. The assert annotation is used as shown on line 30 to inject directly the final value of $z = 12$ obtained from the value analysis. On line 41 an assert annotation is used to prove the suffix 1 predicate and on line 40 an assume annotation is used in order to inject the data of the $w$ variable.

Again the execution of the Jessie analysis is made to generate the *.why* files. It is possible to see in the code below on line 7 that the parameters were already injected into the *.why* file generated by the Jessie analysis. On line 16 the final value of $z$ is already computed so now it is easier to declare the definition of the predicate. Given this it is not necessary to define them manually and match them any more in the *.why* file as done in the first approach. In the next section more examples are going to be shown.

```
1  goal g_part2_ensures_default_po_2:
2    forall w:int32.
3    forall x:int32.
4    forall y:int32.
5    forall z:int32.
6    ("JC_30": true) ->
7    ("JC_39": suffix1(w, z, y, x)) ->
8    forall result:int32.
9    (integer_of_int32(result) = (integer_of_int32(x)
10         + integer_of_int32(y))) ->
11   forall result0:int32.
12   (integer_of_int32(result0) = (integer_of_int32(result)
13         + integer_of_int32(w))) ->
14   forall z0:int32.
15   (z0 = result0) ->
16   ("JC_40": ("JC_40": (integer_of_int32(z0) = 12)))
```

## 4.5 PROBLEMS FACED WITH THE SECOND APPROACH

The problems faced with this approach appear at the moment of generating the Jessie-tool analysis, it is possible to see that the parameters and the suffix were already declared by the tool, but it is still needed to make some modifications on the *.why* file in order the predicate can be applied and analyzed by the Jessie-why analysis. It is shown below a *.why* file before and after it is modified manually.

The code below shows the content of the *.why* file of the goal before being modified. In this case, the suffix definition appears on line 15:

Before modifying:

```
1  goal loopFunction_part2_ensures_default_po_2:
2    forall i:int32.
3    forall int_P_a_1_alloc_table:int_P alloc_table.
4    forall int_P_b_2_alloc_table:int_P alloc_table.
5    forall int_P_c_3_alloc_table:int_P alloc_table.
6    forall int_P_int_M_a_1:(int_P, int32) memory.
7    forall int_P_int_M_b_2:(int_P, int32) memory.
8    forall int_P_int_M_c_3:(int_P,
9    int32) memory.
10   forall y:int32.
11   ("JC_131":
12   (("JC_128": valid_a(int_P_a_1_alloc_table)) and
13    (("JC_129": valid_b(int_P_b_2_alloc_table)) and
14     ("JC_130": valid_c(int_P_c_3_alloc_table))))) ->
15   ("JC_150": suffix1(int_P_int_M_c_3, int_P_int_M_b_2,
16         int_P_int_M_a_1, y,i)) ->
17   forall result:int32.
18   (result = select(int_P_int_M_a_1,
19         shift(a, integer_of_int32(i)))) ->
20   forall result0:int32.
21   (integer_of_int32(result0) =
22         (integer_of_int32(result) - 1)) ->
23   forall int_P_int_M_a_1_0:(int_P,
24   int32) memory.
25   (int_P_int_M_a_1_0 = store(int_P_int_M_a_1,
26         shift(a, integer_of_int32(i)),
27   result0)) ->
28   ("JC_153": ("JC_151": ("JC_151": (integer_of_int32(i) = 7))))
```

It is possible to see below that line 1 has been modified from line 15 in the code shown before. The important thing done was to detect the suffix definition in the *.why* file and change the code that appears before the suffix to predicate. Then, it is neccesary to define all the statements before line 15 inside the new declaration which is done below between lines 3 and 10. After modifying the *.why* file goal, it looks like this:

After modifying:

```
1  predicate suffix1(int_P_int_M_c_3, int_P_int_M_b_2,
2            int_P_int_M_a_1, y, i)=
3
4    forall int_P_a_1_alloc_table:int_P alloc_table.
5    forall int_P_b_2_alloc_table:int_P alloc_table.
6    forall int_P_c_3_alloc_table:int_P alloc_table.
7   ("JC_131":
8   (("JC_128": valid_a(int_P_a_1_alloc_table)) and
9    (("JC_129": valid_b(int_P_b_2_alloc_table)) and
10    ("JC_130": valid_c(int_P_c_3_alloc_table)))))->
11
12   forall result:int32.
13   (result = select(int_P_int_M_a_1, shift(a,
14         integer_of_int32(i)))) ->
15   forall result0:int32.
16   (integer_of_int32(result0) =
17         (integer_of_int32(result) - 1)) ->
18   forall int_P_int_M_a_1_0:(int_P,
19   int32) memory.
20   (int_P_int_M_a_1_0 = store(int_P_int_M_a_1, shift(a,
21          integer_of_int32(i)),result0)) ->
22   ("JC_153": ("JC_151": ("JC_151": (integer_of_int32(i) = 7))))
```

## 4.6 SECOND APPROACH PROCEDURE EXAMPLES

In this section it is possible to find some examples in order to prove if the second approach works. The next example shows the declaration of the global parameters in the C code and how the data obtained from value analysis (value wrapper) is injected into the C code directly instead of doing it from the *.why* file as the first approach.

**EXAMPLE 1**

Code before being modified.

```
1  void operation(){
2  int x,y,z,aux;
3  int *ptr1,*ptr2;
4  x=30;
5  y=0;
6  z=0;
7  ptr1=&x;
8  printf("ptr1 = %p\n", ptr1);
9  ptr2=&y;
10 printf("ptr2 = %p\n", ptr2);
11 x=x * x;
12 printf("x = %d\n", x);
13 y=y+100;
14 printf("y = %d\n", y);
15 z=x-y;
16 printf("z = %d\n", z);
17 aux=*ptr1 + *ptr2 - 6;
18 printf("aux = %d\n", aux);
19 }
20 void main(){
21 operation();
22 }
```

The value wrapper analysis shows the computed data at each line in order to get the data that it is intended to be injected into the Jessie-why analysis using ACSL annotations. This is the code with data already injected and split:

```
#define assumes(P) if (P) {} else miracle()

// miracle function
// @ requires \true;
// @ ensures \false;

void miracle() {
 while (1) ;
 return;
}

int x;
int y;
int z;
int aux;

int *ptr1;
int *ptr2;

int glob;

// 1st suffix
int operation_part2() {

  aux=*ptr1 + *ptr2 - 6;
  //@ assert (aux == 994);
}

//@ predicate suffix1 = glob;
int operation_part1() {
x=30;
y=0;
z=0;
ptr1=&x;
ptr2=&y;
x=x * x;
y=y+100;
  z=x-y;
assumes (z=800);
//@ assert suffix1;
}
```

```
42
43 void main(){
44 operation();
45 }
```

The *miraclefunction* is supposed to do a nop that will not be simplified by Jessie. From the value analysis using value-wrapper tool we know that the *aux* variable is going to have the value 994. This is defined using an assert annotation on line 26. Also $z$ will have the value 800 defined on line 39, it is declared using an *assume* given that it is declared before proving the suffix 1 predicate.

These values are added directly in the code using ACSL annotations. For this example the code was separated in two parts and the variables of the function split were declared as globals as shown between lines 12 and 15. Part 1 can be founded on line 30 and part 2 on line 23.

After running the Jessie-Why Analysis the next result was obtained:

```
total    :   2
valid    :   1 ( 50%)
invalid  :   0 (  0%)
unknown  :   0 (  0%)
timeout  :   0 (  0%)
failure  :   1 ( 50%)
total wallclock time : 0.07 sec
total CPU time       : 0.05 sec
valid VCs:
    average CPU time : 0.02
    max CPU time     : 0.02
invalid VCs:
    average CPU time : -nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : -nan
    max CPU time     : 0.00
```

With this example it is possible to show a first attempt to see if the second approach works in order to inject data in the Jessie-why files. In this case, after running the Jessie-why analysis we obtain 1 valid result and 1 failure shown as inconsistent assumption.

After looking in the source code and in the predicate definition file, it was

detected that the $x$ and $y$ were declared as pointers by the Jessie-why tool, these two locals are used by our predicate so the problem was generated there. In other words that is the reason why the invalid result was generated.

**EXAMPLE 2**

Code before being modified (same as above, but removing pointers):

Code with data already injected and split (same as above, but removing pointers):

```
1  #define assumes(P) if (P) {} else miracle()
2
3  void operation(){
4
5  int x,y,z,aux;
6
7  x=30;
8  y=0;
9  z=0;
10 x=x * x;
11 y=y+100;
12 z=x-y;
13 aux= z - 6;
14 }
15
16
17 // miracle function
18 // @ requires \true;
19 // @ ensures \false;
20
21 void miracle() {
22  while (1) ;
23  return;
24 }
25
26 int x;
27 int y;
28 int z;
29 int aux;
30
31 int glob;
32
```

```
33 // 1st suffix
34 int operation_part2() {
35
36   aux=z - 6;
37   //@ assert (aux == 794);
38 }
39
40 //@ predicate suffix1 = glob;
41 int operation_part1() {
42 x=30;
43 y=0;
44 z=0;
45 x=x * x;
46 y=y+100;
47   z=x-y;
48 assumes (z=800);
49 //@ assert suffix1;
50 }
51
52 void main(){
53
54 operation();
55
56 }
```

Jessie-why Analysis

After running the Jessie-Why Analysis the next result was obtained.

```
total    :   2
valid    :   1 ( 50%)
invalid  :   0 (  0%)
unknown  :   1 ( 50%)
timeout  :   0 (  0%)
failure  :   0 (  0%)
total wallclock time : 0.06 sec
total CPU time       : 0.04 sec
valid VCs:
    average CPU time : 0.01
    max CPU time     : 0.01
invalid VCs:
    average CPU time : -nan
```

```
    max CPU time       : 0.00
unknown VCs:
    average CPU time : 0.03
    max CPU time       : 0.03
```

In this case the pointers were removed in the same code as in example 1, the parameters of the predicate were also declared as global variables. The final Jessie-why analysis launches 1 unknown result instead of a failure as it can be seen in the previous image. This was a more positive result. It is possible to think the pointers are still a problem for the Jessie-tool.

**EXAMPLE 3**

Code before being modified.

```
1  void g() {
2    int x,y,z,w;
3    x = 1;
4    y = 2;
5    z = 0;
6    w = 5;
7    z = y + y;
8    w = z + w;
9    z = x + y + w;
10 }
11
12 void main(){
13     g();
14 }
```

Code with data already injected and split:

```
1  #define assumes(P) if (P) {} else miracle()
2
3  void g() {
4    int x,y,z,w;
5    x = 1;
6    y = 2;
7    z = 0;
8    w = 5;
```

```
9     z = y + y;
10    w = z + w;
11    z = x + y + w;
12  }
13
14
15  // miracle function
16  // @ requires \true;
17  // @ ensures \false;
18  void miracle() {
19   while (1);
20   return;
21  }
22
23  int x,y,z,w;
24  int glob;
25
26  // 1st suffix
27  int g_part2() {
28       z = x + y + w;
29    //@ assert (z == 12);
30  }
31
32
33  //@ predicate suffix1 = glob;
34  int g_part1() {
35    x = 1;
36    y = 2;
37    z = 0;
38    w = 5;
39    z = y + y;
40    w = z + w;
41  assumes (w=9);
42  //@ assert suffix1;
43  }
44
45
46  void main(){
47
48       g();
49
50  }
```

The *miraclefunction* is supposed to do a nop that will not be simplified by Jessie. From the value wrapper analysis it is known that the $w$ variable is going to have the value 9, which is declared on line 49 and also that $z$ will have the value 12, which is declared on line 29. These values are added directly in the code using ACSL annotations. For this example the code was separated in two parts as shown below, also global parameters were declared as seen on line 23.

After running the Jessie-Why Analysis, the next result was obtained.

```
total    :   2
valid    :   1 ( 50%)
invalid  :   0 (  0%)
unknown  :   0 (  0%)
timeout  :   0 (  0%)
failure  :   1 ( 50%)
total wallclock time : 0.24 sec
total CPU time       : 0.04 sec
valid VCs:
    average CPU time : 0.01
    max CPU time     : 0.01
invalid VCs:
    average CPU time : -nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : -nan
    max CPU time     : 0.00
```

For this example it was avoided using pointers and it was tried to use a more simple code that includes just additions. An error was get this time called *assert_failure*, which seems to be generated because some global parameters get lost in the *.why* file when the Jessie-tool makes the translation.

**EXAMPLE 4**

The difference between this example and the previous ones is this one declares the parameters of the suffix directly in the predicate declaration.

Code before being modified:

```
1  void g() {
2    int x,y,z,w;
3    x = 1;
4    y = 2;
5    z = 0;
6    w = 5;
7    z = y + y;
8    w = z + w;
9    z = x + y + w;
10 }
11
12 void main(){
13      g();
14 }
```

Value Wrapper Analysis: Code with data already injected and split declares the parameters of the suffix directly in the predicate declaration on line 26:

```
1  #define assumes(P) if (P) {} else miracle()
2
3
4  void g() {
5    int x,y,z,w;
6    x = 1;
7    y = 2;
8    z = 0;
9    w = 5;
10   z = y + y;
11   w = z + w;
12   z = x + y + w;
13 }
14
15
16 // miracle function
```

```
17 // @ requires \true;
18 // @ ensures \false;
19 void miracle() {
20  while (1);
21  return;
22 }
23
24 int x,y,z,w;
25
26 //@ predicate suffix1 = x && y && z && w;
27
28 // 1st suffix
29 int g_part2() {
30   //@ assert suffix1;
31     z = x + y + w;
32   //@ assert (z == 12);
33 }
34
35 int g_part1() {
36   x = 1;
37   y = 2;
38   z = 0;
39   w = 5;
40   z = y + y;
41   w = z + w;
42 assumes (w == 9);
43 //@ assert suffix1;
44 }
45
46
47 void main(){
48
49      g();
50
51 }
```

The miracle function is supposed to do a nop that will not be simplified by Jessie. From the value wrapper analysis it is known that the $w$ variable is going to have the value 9 declared on line 42 and also that $z$ will have the value 12 declared on line 32. These values were added directly in the code using ACSL annotations. This example was separated in two parts as it is possible to be seen on lines 29 and 35.

After running the Jessie-Why Analysis, the next result was obtained.

```
total    :   5
valid    :   2 ( 40%)
invalid  :   0 (  0%)
unknown  :   3 ( 60%)
timeout  :   0 (  0%)
failure  :   0 (  0%)
total wallclock time : 0.13 sec
total CPU time       : 0.13 sec
valid VCs:
    average CPU time : 0.03
    max CPU time     : 0.03
invalid VCs:
    average CPU time : -nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : 0.02
    max CPU time     : 0.03
```

Procedure :

After executing the Jessie-why analysis it is possible to see that applying the declaration of the parameter inside the predicate in the C code works better than declaring it just as global. This way any failure was get and instead it was get 2 valid proofs and 3 unknown, without errors.

# 5 CONDITIONALS

Now that in the previous sections it was presented how to handle some approaches to inject the data in the Jessie-why, it is possible to present how to handle special cases. Given that the examples before just include sequential codes, conditional statements with two or more branches need to be analyzed in a different way compared to the sequential code. In the next section it is going to be explained the approach for handling them.

### FIRST APPROACH

## 5.1 SPLIT THE CONDITIONAL BODY IN A NEW FUNCTION

An approach to handle conditionals is presented in this section. As we know conditionals can have two or more branches, the goal of this approach is to get the branches in different new functions.

In the example used in this section it is still used the technique shown since the first approach about split the body in two new functions, the consequence of doing this is they can be later split as sequential code. As it is possible to see on line 22 and line 30, it is added the assert in the original function as you can see on line 18 in order the goal can be analysed.

Assert annotations are used in each split function to consider both branches of the conditional statement. It was decided to use assert annotations instead of assumes because these last ones were not working correctly at the moment of running the Jessie-why. The problem presented was that they were not considering the not part of the conditional statement. The line 48 is used in order the parameters of the suffix can be declared directly in the goal inside the *.why* files so we do not inject it manually and it is possible to see the type the Jessie analysis assign to it. Below it is possible to see the example used for this approach.

```
1 #define assumes(P) if (P) {} else miracle()
2
3 int a,b,c,glob;
```

```
4
5
6  //@ requires 7 <= a;
7  void ifFunction (int a) {
8
9        if(a<=7){
10
11            b=1;
12            c=a+b;
13       }else{
14
15           b=2;
16           c=a+b;
17       }
18 //@ assert (c>=8);
19 }
20
21
22 void ifBody1(){
23     //@ assert (7<=a);
24     //@ assert (a<=7);
25  b=1;
26  c=a+b;
27 //@ assert (c>=8);
28 }
29
30 void ifBody2(){
31 //@ assert (7<=a);
32 //@ assert (!(a<=7));
33  b=2;
34  c=a+b;
35 //@ assert (c>=8);
36 }
37
38
39 // miracle function
40 // @ requires \true;
41 // @ ensures \false;
42
43 void miracle() {
44  while (1) ;
45  return;
46 }
47
48 //@ predicate suffix1 = a && b && c;
49 int ifBody1_part2() {
```

```
50      //@ assert suffix1;
51      c=a+b;
52      //@ assert (c >= 8);
53  }
54
55
56  int ifBody1_part1() {
57      //@ assert (7<=a);
58      //@ assert (a<=7);
59      b=1;
60  //@ assert suffix1;
61  }
62
63  void main (){
64
65   ifFunction(7);
66
67  }
```

It was run the Jessie-tool in order to obtain the goals of our C code. The safety goals have been excluded and the ones of the functions are don't care as for example the goals generated for the original function. In the *.why* code below it is possible to see first the second branch of the conditional sentence, the next part of code is declaring the definition of the predicate seen on line 17.

For this part, it is detected the goal which generates automatically the suffix with the help of the predicate instruction used in the C code previously, it is just added the word predicate on it and erase the declarations that appear before it and are already included in the parameters. In the last part, it is possible to see the goal that applies the definition of the predicate created as it can be seen on line 34.

```
1  goal ifBody2_ensures_default_po_3:
2    ("JC_31": true) ->
3    ("JC_40": (7 <= integer_of_int32(a))) ->
4    ("JC_41": (not (integer_of_int32(a) <= 7))) ->
5    forall result:int32.
6    (integer_of_int32(result) = 2) ->
7    forall b:int32.
8    (b = result) ->
9    forall result0:int32.
10   (integer_of_int32(result0) = (integer_of_int32(a)
11         + integer_of_int32(b))) ->
12   forall c:int32.
13   (c = result0) ->
14   ("JC_42": ("JC_42": (integer_of_int32(c) >= 8)))
15
16
17
18 predicate suffix1(c : int32, b : int32)=
19   forall result:int32.
20   (integer_of_int32(result) = (integer_of_int32(a)
21          + integer_of_int32(b))) ->
22   forall c0:int32.
23   (c0 = result) ->
24   ("JC_69": ("JC_69": (integer_of_int32(c0) >= 8)))
25
26
27 goal ifBody1_part1_ensures_default_po_3:
28   forall c:int32.
29   ("JC_73": true) ->
30   ("JC_81": (7 <= integer_of_int32(a))) ->
31   ("JC_82": (integer_of_int32(a) <= 7)) ->
32   forall result:int32.
```

```
33    (integer_of_int32(result) = 1) ->
34    forall b:int32.
35    (b = result) ->
36    ("JC_83": ("JC_83": suffix1(c, b)))
```

Finally it is executed the Jessie-why analysis, that is going to analyse the two goals presented before and display if the predicate is valid or not. As it could be seen below the result is valid for both goals, which indicates that the predicate generated is operating when it was applied.

After running the Jessie-Why Analysis the next result was obtained:

```
1  total    :   2
2  valid    :   2 (100%)
3  invalid  :   0 (  0%)
4  unknown  :   0 (  0%)
5  timeout  :   0 (  0%)
6  failure  :   0 (  0%)
7  total wallclock time : 0.05 sec
8  total CPU time       : 0.05 sec
9  valid VCs:
10    average CPU time : 0.03
11    max CPU time     : 0.03
12 invalid VCs:
13    average CPU time : -nan
14    max CPU time     : 0.00
15 unknown VCs:
16    average CPU time : -nan
17    max CPU time     : 0.00
```

## 5.2 FIRST CONDITIONAL APPROACH WITH ARRAYS

The next example is going to prove the same approach using arrays in the C code, in order to see if they are not a limitation for the approach. The assert annotations are used in each split function to consider both branches of the conditional statement. It was decided to use *assert* annotations instead of *assumes* because this last ones were not working correctly at the moment of running the Jessie- analysis.

```
1  #define assumes(P) if (P) {} else miracle()
2
3  int a[9],b[9],c[9];
4  int d,glob;
5
6  //@ requires 7 <= d;
7  void ifFunction (int d) {
8
9       if(d<=7){
10
11            b[1]=1;
12            a[1]=7;
13            c[1]=a[1]+b[1];
14     }else{
15
16           b[1]=2;
17           a[1]=7;
18          c[1]=a[1]+b[1];
19     }
20 //@ assert (c[1]>=8);
21 }
22
23 void ifBody1(){
24    //@ assert (7<=d);
25    //@ assert (d<=7);
26            b[1]=1;
27            a[1]=7;
28            c[1]=a[1]+b[1];
29 //@ assert (c[1]>=8);
30 }
31
32 void ifBody2(){
33 //@ assert (7<=d);
```

```
34 //@ assert (!(d<=7));
35           b[1]=2;
36           a[1]=7;
37          c[1]=a[1]+b[1];
38 //@ assert (c[1]>=8);
39 }
40
41
42 // miracle function
43 // @ requires \true;
44 // @ ensures \false;
45
46 void miracle() {
47 while (1) ;
48 return;
49 }
50
51 //@ predicate suffix1 = a && b && c && d && glob;
52 int ifBody1_part2() {
53   //@ assert suffix1;
54   c[1]=a[1]+b[1];
55   //@ assert (c[1] >= 8);
56 }
57
58
59 int ifBody1_part1() {
60   //@ assert (7<=d);
61   //@ assert (d<=7);
62   b[1]=1;
63   a[1]=7;
64 //@ assert suffix1;
65 }
66
67 void main (){
68
69 ifFunction(7);
70
71 }
```

In the *.why* code below, it is possible to see the second branch of the conditional sentence and the next part of code declaring the definition of the predicate seen on line 72. It is possible to see that compared with the previous example that doesn't include arrays, the why code is more extended in this case even if it is the same code.

```
********************** goals proof*******************

************************goal of the second branch*******
goal ifBody2_ensures_default_po_3:
 forall int_P_a_1_alloc_table:int_P alloc_table.
 forall int_P_b_2_alloc_table:int_P alloc_table.
 forall int_P_c_3_alloc_table:int_P alloc_table.
 forall int_P_int_M_a_1:(int_P, int32) memory.
 forall int_P_int_M_b_2:(int_P, int32) memory.
 forall int_P_int_M_c_3:(int_P,
 int32) memory.
 ("JC_83":
 (("JC_80": valid_a(int_P_a_1_alloc_table)) and
  (("JC_81": valid_b(int_P_b_2_alloc_table)) and
   ("JC_82": valid_c(int_P_c_3_alloc_table))))) ->
 ("JC_99": (7 <= integer_of_int32(d))) ->
 ("JC_100": (not (integer_of_int32(d) <= 7))) ->
 forall result:int32.
 (integer_of_int32(result) = 2) ->
 forall int_P_int_M_b_2_0:(int_P,
 int32) memory.
 (int_P_int_M_b_2_0 = store(int_P_int_M_b_2, shift(b, 1),
 result)) ->forall result0:int32.
 (integer_of_int32(result0) = 7) ->
 forall int_P_int_M_a_1_0:(int_P,
 int32) memory.
 (int_P_int_M_a_1_0 = store(int_P_int_M_a_1, shift(a, 1),
 result0)) ->forall result1:int32.
 (result1 = select(int_P_int_M_a_1_0, shift(a, 1))) ->
 forall result2:int32.
 (result2 = select(int_P_int_M_b_2_0, shift(b, 1))) ->
 forall result3:int32.
 (integer_of_int32(result3) = (integer_of_int32(result1)
  + integer_of_int32(result2))) ->
 forall int_P_int_M_c_3_0:(int_P,
 int32) memory.
 (int_P_int_M_c_3_0 = store(int_P_int_M_c_3, shift(c, 1),
 result3)) ->("JC_101":
 ("JC_101": (integer_of_int32(select(int_P_int_M_c_3_0,
 shift(c, 1))) >= 8)))


********************definition of the predicate**********

predicate suffix1(int_P_int_M_c_3 : (int_P, int32) memory,
```

```
 int_P_int_M_b_2 : (int_P, int32) memory,
 int_P_int_M_a_1 : (int_P, int32) memory)=
 forall int_P_a_1_alloc_table:int_P alloc_table.
 forall int_P_b_2_alloc_table:int_P alloc_table.
 forall int_P_c_3_alloc_table:int_P alloc_table.

 ("JC_136":
 (("JC_133": valid_a(int_P_a_1_alloc_table)) and
  (("JC_134": valid_b(int_P_b_2_alloc_table)) and
   ("JC_135": valid_c(int_P_c_3_alloc_table))))) ->
 forall result:int32.
 (result = select(int_P_int_M_a_1, shift(a, 1))) ->
 forall result0:int32.
 (result0 = select(int_P_int_M_b_2, shift(b, 1))) ->
 forall result1:int32.
 (integer_of_int32(result1) = (integer_of_int32(result)
+ integer_of_int32(result0))) ->
 forall int_P_int_M_c_3_0:(int_P,
 int32) memory.
 (int_P_int_M_c_3_0 = store(int_P_int_M_c_3, shift(c, 1),
 result1)) ->("JC_152":
 ("JC_152": (integer_of_int32(select(int_P_int_M_c_3_0,
  shift(c, 1))) >= 8)))


*********goal that use the definition of the predicate****

goal ifBody1_part1_ensures_default_po_3:
 forall int_P_a_1_alloc_table:int_P alloc_table.
 forall int_P_b_2_alloc_table:int_P alloc_table.
 forall int_P_c_3_alloc_table:int_P alloc_table.
 forall int_P_int_M_a_1:(int_P, int32) memory.
 forall int_P_int_M_b_2:(int_P, int32) memory.
 forall int_P_int_M_c_3:(int_P,
 int32) memory.
 ("JC_161":
 (("JC_158": valid_a(int_P_a_1_alloc_table)) and
  (("JC_159": valid_b(int_P_b_2_alloc_table)) and
   ("JC_160": valid_c(int_P_c_3_alloc_table))))) ->
 ("JC_176": (7 <= integer_of_int32(d))) ->
 ("JC_177": (integer_of_int32(d) <= 7)) ->
 forall result:int32.
 (integer_of_int32(result) = 1) ->
 forall int_P_int_M_b_2_0:(int_P,
 int32) memory.
 (int_P_int_M_b_2_0 = store(int_P_int_M_b_2, shift(b, 1),
```

```
93  result)) ->forall result0:int32.
94  (integer_of_int32(result0) = 7) ->
95  forall int_P_int_M_a_1_0:(int_P,
96  int32) memory.
97  (int_P_int_M_a_1_0 = store(int_P_int_M_a_1, shift(a, 1),
98  result0)) ->("JC_178":
99  ("JC_178": suffix1(int_P_int_M_c_3, int_P_int_M_b_2_0,
100 int_P_int_M_a_1_0)))
```

Finally the execution of the Jessie-why analysis was done to analyse the two goals presented before to display if the predicate is valid or not. As it could be seen below the result was valid for both goals, which indicates us that the predicate generated is working and even arrays were not a limitation for this approach.

After running the Jessie-Why Analysis the next result was obtained:

```
1  total    :   2
2  valid    :   2 (100%)
3  invalid  :   0 (  0%)
4  unknown  :   0 (  0%)
5  timeout  :   0 (  0%)
6  failure  :   0 (  0%)
7  total wallclock time : 0.27 sec
8  total CPU time       : 0.26 sec
9  valid VCs:
10    average CPU time : 0.13
11    max CPU time     : 0.22
12 invalid VCs:
13    average CPU time : -nan
14    max CPU time     : 0.00
15 unknown VCs:
16    average CPU time : -nan
17    max CPU time     : 0.00
```

# 6 LOOP UNROLLING

### FIRST APPROACH

## 6.1 UNROLLING A LOOP SEQUENTIALLY

The loops play an important role in the thesis, given that most of the software codes include loops. There can be loops that do not terminate and that could be a problem for the software analysis to give a correct result. In our case one of the challenges was to identify a way to unroll the loop in case it was pretended to split the function inside an internal loop. The first goal is to unroll the loop sequentially so it can be split at any point. The next code shows a common loop code.

```
1  void loopFunction () {
2
3  for (i =0; i < 8 ; i++){
4        b[i]=1;
5        c[i]=1;
6        a[i]=b[i]+c[i];
7  }
8
9      y=a[7];
10
11 }
```

After the loop is unrolled sequentially, it will look like this:

```
1  void loopFunction () {
2
3          int i,y;
4          int a[9],b[9],c[9];
5
6      b[0]=1;
7      b[1]=1;
8      b[2]=1;
9      b[3]=1;
10     b[4]=1;
11     b[5]=1;
12     b[6]=1;
13     b[7]=1;
14
15     c[0]=1;
16     c[1]=1;
17     c[2]=1;
18     c[3]=1;
19     c[4]=1;
20     c[5]=1;
21     c[6]=1;
22     c[7]=1;
23
24
25   a[0]=b[0]+c[0];
26   a[1]=b[1]+c[1];
27   a[2]=b[2]+c[2];
28   a[3]=b[3]+c[3];
29   a[4]=b[4]+c[4];
30   a[5]=b[5]+c[5];
31   a[6]=b[6]+c[6];
32   a[7]=b[7]+c[7];
33
34 }
```

The example before shows that a loop can be unrolled sequentially and it works for a small loop, in this case 9 iterations. But in case it is a bigger loop, this approach could complicate things more. In order to avoid this problem, one idea for unrolling the loop sequentially can be shown in the next structure. The idea is to convert the while loop into nested ifs in order to split the C code at any point that we want. After that, the rest of the loop can be written as a for loop as shown below:

If we have a while loop that contains a body....

```
1  while(i<=7){
2  ...
3  body
4  ...
5  }
```

It is possible to unroll it using $ifs$ until the point we want to split and then declare it as a $for$, in the next code example we can see how on lines 2 and 7 we use the ifs to unroll the loop sequentially until its second iteration, then on line 13 we use the for to just run the rest of the loop.

```
1  i=1   ;
2  if( i <= 7 ){
3     ...
4    body
5
6    i++;
7    if ( i <= 7 ){
8       ...
9      body
10
11      i++;
12   }
13  for ( i <=7   ; i ++ ){
14
15  body
16   ...
17  }
18 }
```

**EXAMPLE 1**

The following example shows the approach of unrolling a loop sequentially as explained in the section before.

Source Code before being modify:

```
1  int a[9],b[9],c[9];
2  int i,y,glob;
3
4  void loopFunction () {
5
6      int i=0;
7      int y=0;
8      int a[9],b[9],c[9];
9
10 while(i<=7){
11
12     b[i]=1;
13     c[i]=1;
14     a[i]=b[i]+c[i]+c[i];
15     a[i]=a[i]-1;
16     i++;
17
18 }
19 y=a[7];
20 //printf("result: %d\n",y);
21 }
22
23 void main (){
24  loopFunction();
25 }
```

The code after being modified is shown next:

```
#define assumes(P) if (P) {} else miracle()

int a[9],b[9],c[9];
int i,y,glob;


void loopFunction () {

    int i=0;
    int y=0;
    int a[9],b[9],c[9];

while(i<=7){

   b[i]=1;
   c[i]=1;
   a[i]=b[i]+c[i]+c[i];
   a[i]=a[i]-1;
   i++;
}

y=a[7];
//printf("result: %d\n",y);
}
void main (){
 loopFunction();
}

// miracle function
// @ requires \true;
// @ ensures \false;

void miracle() {
 while (1) ;
 return;
}

//@ predicate suffix1 = a && b && c && i && y;

// 1st suffix
int loopFunction_part2() {

   for (i=3;i<=7;i++){

   b[i]=1;
   c[i]=1;
```

```
47      a[i]=b[i]+c[i]+c[i];
48      a[i]=a[i]-1;
49      i++;
50
51  }
52   //@ assert (a[1] == 2) && (a[2] == 2);
53  }
54
55
56
57  int loopFunction_part1() {
58
59  if(i<=7){
60
61      b[i]=1;
62      c[i]=1;
63      a[i]=b[i]+c[i]+c[i];
64      a[i]=a[i]-1;
65      i++;
66
67      if(i<=7){
68
69          b[i]=1;
70          c[i]=1;
71          a[i]=b[i]+c[i]+c[i];
72          a[i]=a[i]-1;
73          i++;
74      }
75
76  }
77  //@ assert suffix1;
78  }
```

The *miraclefunction* is supposed to do a nop that will not be simplified by Jessie. In the code it is possible to see how the loopFunction () is split in two parts. In the part 1 declared on line 57 it was used the sequential unroll technique using *ifs*, in the part 2 declared on line 41 it was used for just to continue the loop after the point interested in being separated sequentially. For this example the second approach explained in section 4.4 has been used. It is possible to notice how the predicate is declared with its parameters directly on line 38, also the use of *assert* to define the values of the arrays on line 52.

After running the Jessie-Why Analysis the next result was obtained:

```
total    :   2
valid    :   0 (  0%)
invalid  :   0 (  0%)
unknown  :   2 (100%)
timeout  :   0 (  0%)
failure  :   0 (  0%)
total wallclock time : 0.78 sec
total CPU time       : 0.78 sec
valid VCs:
average CPU time : -nan
max CPU time     : 0.00
invalid VCs:
average CPU time : -nan
max CPU time     : 0.00
unknown VCs:
average CPU time : 0.39
max CPU time     : 0.75
```

After executing the Jessie-why analysis it could be seen the result obtained display as unknown what for the moment indicates the approach is not wrong.

## 6.2 SEPARATING THE LOOP BODY IN AN AUXILIARY FUNCTION

In order to get a valid proof in loop unrolling, another approach is explained here. This approach consists in separating the content of the loop in an auxiliary function and add a loop invariant before the loop as shown below on line 7. This way the function that contains the body of the loop can be separated in any point as shown on line 23, it would look like this:

```
1  /* @ requires PRE
2        @ ensures POST
3
4  void f(){
5        ...
6        ...
7        // @ assert Invariant
8
9        while (cond){
10
11         f_body();
12
13   }
14       // @ assumes Invariant ^ cond
15       ...
16       ...
17 }
18
19
20 /* @ requires Invariant ^ cond
21       @ ensures Invariant */
22
23 void f_body(){
24
25       body;
26
27 }
```

**EXAMPLE 1**

A formal example using this approach is shown in this section to prove this approach works. For this example, the code below shows that the body of the loop was split first in a new function called *loopbody* with the invariant declared on line 28. Then, this new function was split in two parts as it can be seen on lines 51 and 60. *Assumes* are used again before the assert of the predicate and asserts in part 2 to declare the final value of $i$ and the array at that position as shown on lines 65 and 66. For this example the global parameters of the predicate were not declared directly.

```
1  #define assumes(P) if (P) {} else miracle()
2
3  int a[9],b[9],c[9];
4  int i,y,glob;
5
6
7
8
9  void loopFunction () {
10  //@ loop invariant 0 <= i <= 7 && \forall int m; 0 <= m < i
   ==> a[m] == b[m] + c[m] + c[m];
11        while(i<=7){
12
13              b[i]=1;
14              c[i]=1;
15              a[i]=b[i]+c[i]+c[i];
16              a[i]=a[i]-1;
17              i++;
18
19 }
20
21            y=a[7];
22
23 }
24
25
26
27 // @ requires \false;
28 // @ ensures \loop invariant 0 <= i <= 7 && \forall int m; 0 <= m < i
   ==> a[m] == b[m] + c[m] + c[m];
29 void loopBody(){
30    b[i]=1;
```

```
31    c[i]=1;
32    a[i]=b[i]+c[i]+c[i];
33    a[i]=a[i]-1;
34    i++;
35
36  }
37
38
39  // miracle function
40  // @ requires \true;
41  // @ ensures \false;
42
43  void miracle() {
44   while (1) ;
45   return;
46  }
47
48
49
50  // 1st suffix
51  int loopFunction_part2() {
52
53       a[i]=a[i]-1;
54      //@ assert (i == 7);
55      //@ assert (a[i] == 2);
56  }
57
58
59  //@ predicate suffix1 = glob;
60  int loopFunction_part1() {
61     b[i]=1;
62     c[i]=1;
63     a[i]=b[i]+c[i]+c[i];
64
65   assumes (i == 7);
66   assumes (a[i]==2);
67  //@ assert suffix1;
68  }
69
70
71  void main (){
72
73   loopFunction();
74
75  }
```

In this case, after executing the Jessie-why analysis a valid result was obtained and 4 failures. We can see that as explained in section 4.9, it is not enough to declare parameters of the predicate as global variables. It is necessary to declare them directly on the predicate ACSL annotation.

After running the Jessie-Why Analysis the next result was obtained:

```
total    :   11
valid    :   1 (  9%)
invalid  :   0 (  0%)
unknown  :   6 ( 55%)
timeout  :   0 (  0%)
failure  :   4 ( 36%)
total wallclock time : 0.47 sec
total CPU time       : 0.46 sec
valid VCs:
average CPU time : 0.03
max CPU time     : 0.03
invalid VCs:
average CPU time : -nan
max CPU time     : 0.00
unknown VCs:
average CPU time : 0.06
max CPU time     : 0.14
```

## 6.3 PROBLEMS FACED WITH THE SECOND APPROACH

The problems faced with this approach were that some failures still after executing it. After analyzing the errors displayed, it was detected that the global parameters defined in the C code at the moment of being translated to the *.why* file by the Jessie-why analysis were lost in some cases (this means not defined in the Jessie-tool in the new .why file).

**THIRD APPROACH**

## 6.4 SPLIT THE LOOP BODY IN A NEW FUNCTION AND USE ASCL ANNOTATIONS

In order to avoid the problem faced in the second approach of loop unrolling, this approach proposition is to put the parameters in the C code directly in the predicate as done in example 4 section 4.4. Doing this, the Jessie translator generates automatically the correct parameters in the *.why* file. The predicate is now declared on line 38, all the other annotations are the same as the previous example.

**EXAMPLE 1**

```
1  #define assumes(P) if (P) {} else miracle()
2
3  int a,b,c;
4  int i,y;
5
6  void loopFunction () {
7  //@ loop invariant 0 <= i <= 7 && \forall int m; 0 <= m < i
   ==> a == a + i;
8       while(i<=7){
9            a=a+i;
10          b=a;
11           i++;
12 }
13 y=b;
14 //@ assert(b <= 28);
15 }
16
17
18 // @ ensures \loop invariant 0 <= i <= 7 && \forall int i;
19 0 <= i  ==> a == a + i;
20 void loopBody(){
21 //@ assert (a<=21);
22 //@ assert (i<=7);
23    a=a+i;
24    b=a;
25    i++;
26 //@ assert(b <= 28);
```

```
27 }
28
29
30 // miracle function
31 // @ requires \true;
32 // @ ensures \false;
33 void miracle() {
34 while (1) ;
35 return;
36 }
37
38 //@ predicate suffix1 = a && b && c && i && y;
39 int loopFunction_part2() {
40   //@ assert suffix1;
41    b=a;
42    i++;
43   //@ assert(b <= 28);
44 }
45
46
47 int loopFunction_part1() {
48 //@ assert (a<=21);
49 //@ assert (i<=7);
50   a=a+i;
51 //@ assert suffix1;
52 }
53
54
55 void main (){
56 a=0;
57 loopFunction();
58
59 }
```

In the why code below, the goal referring to the loop original function could be seen first. The next part of code is declaring the definition of the predicate seen on line 29. Finally the place where the definition of the predicate is applied on line 56.

```
****************** goals proof*******************

*******************goal of the loop function


goal loopFunction_ensures_default_po_7:
 ("JC_4": true) ->
 forall a0:int32.
 forall b:int32.
 forall i0:int32.
 ("JC_22":
 (("JC_19": (0 <= integer_of_int32(i0))) and
  (("JC_20": (integer_of_int32(i0) <= 7)) and
   ("JC_21":
   (forall m:int32.
     (((0 <= integer_of_int32(m)) and
       (integer_of_int32(m) < integer_of_int32(i0))) ->
      (integer_of_int32(a0) = (integer_of_int32(a0)
        + integer_of_int32(i0)))))))))) ->
 (integer_of_int32(i0) > 7) ->
 forall y:int32.
 (y = b) ->
 ("JC_26": ("JC_26": (integer_of_int32(b) <= 28)))


***************definition of the predicate


predicate suffix1(y : int32, i : int32, b : int32, a : int32)=
 forall b0:int32.
 (b0 = a) ->
 forall result:int32.
 (integer_of_int32(result) = (integer_of_int32(i) + 1)) ->
 forall i0:int32.
 (i0 = result) ->
 ("JC_69": ("JC_69": (integer_of_int32(b0) <= 28)))



***************goal that use the definition of the predicate


goal loopFunction_part1_ensures_default_po_3:
 forall a:int32.
 forall b:int32.
 forall i:int32.
```

```
47  forall y:int32.
48  ("JC_73": true) ->
49  ("JC_82": (integer_of_int32(a) <= 21)) ->
50  ("JC_83": (integer_of_int32(i) <= 7)) ->
51  forall result:int32.
52  (integer_of_int32(result) = (integer_of_int32(a)
53        + integer_of_int32(i))) ->
54  forall a0:int32.
55  (a0 = result) ->
56  ("JC_84": ("JC_84": suffix1(y, i, b, a0)))
```

After running the Jessie-Why Analysis the next result was obtained:

```
1  total    :   2
2  valid    :   2 (100%)
3  invalid  :   0 (  0%)
4  unknown  :   0 (  0%)
5  timeout  :   0 (  0%)
6  failure  :   0 (  0%)
7  total wallclock time : 0.15 sec
8  total CPU time       : 0.06 sec
9  valid VCs:
10    average CPU time : 0.03
11    max CPU time     : 0.03
12 invalid VCs:
13    average CPU time : -nan
14    max CPU time     : 0.00
15 unknown VCs:
16    average CPU time : -nan
17    max CPU time     : 0.00
```

After executing this approach, no errors were displayed. The result *allvalidproof* was generated by the Jessie-why tool and 0 unknown, this means the definition of the predicate and the goal that were applied work.

## 6.5 PROBLEMS FACED WITH THIS APPROACH

The problem with this approach was it is still needed to do manually some steps. The goal in which the suffix appears with the parameters inside it must be identified. After that, it is needed look for the parameters that appear inside (the ones that are declared before the line in which it appears). Then it is just needed to put inside the predicate the missing statements.

# 7 LIMITATIONS

During the development of each approach some limitations important to be considered appeared. One of the limitations found was that the pointers, arrays and memory allocation produce problems in the value and Jessie analysis plug-ins. Given these conditions, most of the attempts were tested over simple C codes trying not to include the structures mentioned.

## 7.1 LIMITATIONS OF THE TOOLS

One of the first limitations found during the development of the thesis was the tool (Frama-C), as it is an open source software without a final version. It can be even improved and may contain limitations not found yet. In our specific case, when the value analysis plug-in was used with help of a special tool (value-wrapper) in order to obtain the data at each line of the code some problems were founded: for example at the moment of analyzing special C codes structures. An example was with the values of the linked list. When this structure was analyzed the tool was only able to define the head value, being not able to follow the linked list. As a consequence the values in each node were not able to be seen by the tool. An example is shown here:

```
1  #include<stdio.h>
2
3  struct list_el {
4     int val;
5     struct list_el * next;
6  };
7
8
9  struct list_el *head = 1;
10 struct list_el *lp;
11
12  struct list_el node10 ={1, NULL};
13  struct list_el node9 = {2, &node10};
14  struct list_el node8 = {3, &node9};
15  struct list_el node7 = {4, &node8};
16  struct list_el node6 = {5, &node7};
17  struct list_el node5 = {6, &node6};
18  struct list_el node4 = {7, &node5};
19  struct list_el node3 = {8, &node4};
20  struct list_el node2 = {9, &node3};
21  struct list_el node1 = {10,&node2};
22
23 void main() {
24
25 for(lp = head; lp != NULL; lp = lp->next)
26    printf("%d\n", lp->val);
27
28 }
```

Result obtained with Value wrapper:

On lines 16, 20, 24 and 28 the computed values are shown by the tool and it is confirmed that for the linked list the node values were not displayed at each line.

```
1  Function main
2  [value-wrapper] Stmt id 1: lp = head;
3                  ---
4                  lp in {0; }
5                  my try as ACSL predicate: (lp      0)
6  [value-wrapper] Stmt id 1: lp = head;
```

```
7                   ---
8                   head in {1; }
9                   my try as ACSL predicate: (head      1)
10  [value-wrapper] Stmt id 4: if (! (lp != (void *)0)) { break; }
11                  ---
12                  lp in {1; }
13                  my try as ACSL predicate: (lp      1)
14  [value-wrapper] Stmt id 6: printf((char const    *)"%d\n",lp->val);
15                  ---
16                  printf in {{}}
17                  my try as ACSL predicate: \false
18  [value-wrapper] Stmt id 6: printf((char const    *)"%d\n",lp->val);
19                  ---
20                  lp->val in {{}}
21                  my try as ACSL predicate: \false
22  [value-wrapper] Stmt id 7: lp = lp->next;
23                  ---
24                  lp in {{}}
25                  my try as ACSL predicate: \false
26  [value-wrapper] Stmt id 7: lp = lp->next;
27                  ---
28                  lp->next in {{}}
29                  my try as ACSL predicate: \false
```

Another limitation found was that the value inside each array in a loop was complicated to compute for the tool. When this scenario was tried, the value wrapper tool displayed the range of the possible values, but not the value at each iteration, in other words it was just getting the whole amount of possible values, but not specific at each point as desired. In order to avoid this problem the loop unrolling approaches in section 6.0 were born. An example of this scenario is shown below:

```
1  int i , t[10];
2
3  void main ( void ) {
4  for( i =0; i <=10; i ++)
5
6  t[i]= i ;
7
8  }
```

The example shown before just shows the execution filling an array inside a loop between the values 0 to 10 inside an array of 11 positions. It is possible to see marked on lines 4, 8, 12, 16, 20 and 24 the range given by the value-wrapper tool after applying it to the code shown before, but not the value in each array position.

```
1
2  Function main
3  [value-wrapper] Stmt id 1: i = 0;
4                  ---
5                  i in {0; }
6    my try as ACSL predicate: (i      0)
7  [value-wrapper] Stmt id 4: if (! (i <= 10)) { break; }
8                  ---
9                  i in [0..11]
10   my try as ACSL predicate: ((i      0)      ((i      11)      \true))
11 [value-wrapper] Stmt id 6: t[i] = i;
12                 ---
13                 t[i] in [0..10]
14   my try as ACSL predicate: ((t[i]      0)      ((t[i]      10)      \true))
15 [value-wrapper] Stmt id 6: t[i] = i;
16                 ---
17                 i in [0..10]
18   my try as ACSL predicate: ((i      0)      ((i      10)      \true))
19 [value-wrapper] Stmt id 7: i ++;
20                 ---
21                 i in [0..10]
22   my try as ACSL predicate: ((i      0)      ((i      10)      \true))
23 [value-wrapper] Stmt id 7: i ++;
24                 ---
25                 i in [0..10]
26   my try as ACSL predicate: ((i      0)      ((i      10)      \true))
```

Working with memory allocation was another of the limitation found at the moment of using it on the value wrapper tool. The final result crashed and it was not found any possible value by the value wrapper tool.

An example that shows this scenario is shown next, a linked list is coded using memory allocation.

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  struct list_el {
5     int val;
6     struct list_el * next;
7  };
8
9  typedef struct list_el item;
10
11 void main() {
12    item * curr, * head;
13    int i;
14
15    head = NULL;
16
17    for(i=1;i<=10;i++) {
18       curr = (item *)malloc(sizeof(item));
19       curr->val = i;
20       curr->next  = head;
21       head = curr;
22    }
23
24    curr = head;
25
26    while(curr) {
27       printf("%d\n", curr->val);
28       curr = curr->next ;
29    }
30 }
```

It is possible to see marked on lines 18, 46, 51, 56,69 and 74 that the values in each node are not defined by the tool. As a result after running the value wrapper tool it is got:

```
1  Function main
2  [value-wrapper] Stmt id 1: head = (item *)((void *)0);
3                  ---
4                  head in {{}}
5                  my try as ACSL predicate: \false
6  [value-wrapper] Stmt id 2: i = 1;
7                  ---
8                  i in {{}}
9                  my try as ACSL predicate: \false
```

```
[value-wrapper] Stmt id 5: if (! (i <= 10)) { break; }
                ---
                i in [1..11]
                my try as ACSL predicate:
                ((i      1)      ((i      11)      \true))
[value-wrapper] Stmt id 7: curr =
                (item *)malloc((unsigned long )sizeof(item ));
                ---
                curr in {{ &NULL + [--..--] ;
                &alloced_return_malloc + [0..2147483647] ;}}
                my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 7: curr = (item *)
                malloc((unsigned long )sizeof(item ));
                ---
                malloc in {{}}
                my try as ACSL predicate: \false
[value-wrapper] Stmt id 8: curr->val = i;
                ---
                curr->val in {{ garbled mix of &{alloced_return_malloc; }
                (origin: Library function
                {linkedListDyn.c:18;
                        }) }}
                my try as ACSL predicate: \true
[value-wrapper] Stmt id 8: curr->val = i;
                ---
                i in [1..10]
   my try as ACSL predicate: ((i      1)      ((i      10)      \true))
[value-wrapper] Stmt id 9: curr->next = head;
                ---
                curr->next in {{ garbled mix of &{alloced_return_malloc; }
                (origin: Misaligned
                {
                  linkedListDyn.c:19;
                }) }}
                my try as ACSL predicate: \true
[value-wrapper] Stmt id 9: curr->next = head;
                ---
                head in {{ &NULL + [--..--] ;
                &alloced_return_malloc + [0..2147483647] ;}}
                my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 10: head = curr;
                ---
                head in {{ &NULL + [--..--] ;
                &alloced_return_malloc + [0..2147483647] ;}}
                my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 10: head = curr;
```

```
                  ---
                  curr in {{ &NULL + [--..--] ;
                  &alloced_return_malloc + [0..2147483647] ;}}
                  my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 11: i ++;
                  ---
                  i in [1..10]
  my try as ACSL predicate: ((i      1)      ((i      10)      \true))
[value-wrapper] Stmt id 11: i ++;
                  ---
                  i in [1..10]
  my try as ACSL predicate: ((i      1)      ((i      10)      \true))
[value-wrapper] Stmt id 12: curr = head;
                  ---
                  curr in {{ &NULL + [--..--] ;
                  &alloced_return_malloc + [0..2147483647] ;}}
                  my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 12: curr = head;
                  ---
                  head in {{ &NULL + [--..--] ;
                  &alloced_return_malloc + [0..2147483647] ;}}
                  my try as ACSL predicate: (\true      \true)
[value-wrapper] Stmt id 15: if (! curr) { break; }
                  ---
                  curr in {{ garbled mix of &{alloced_return_malloc; }
                  (origin: Misaligned
                  {linkedListDyn.c:19;
                    linkedListDyn.c:20;
                      linkedListDyn.c:28;
                       }}
                  my try as ACSL predicate: \true
[value-wrapper] Stmt id 17: printf((char const    *)"%d\n",curr->val);
                  ---

                  printf in {{}}
                  my try as ACSL predicate: \false
[value-wrapper] Stmt id 17: printf((char const    *)"%d\n",curr->val);
                  ---
                  curr->val in {{ garbled mix of &{alloced_return_malloc;
                   } (origin: Misaligned
                           {linkedListDyn.c:19;
                                   linkedListDyn.c:20;
                                       }) }}
                  my try as ACSL predicate: \true
[value-wrapper] Stmt id 18: curr = curr->next;
                  ---
```

```
102                 curr in {{ garbled mix of &{alloced_return_malloc; }
103                 (origin: Misaligned
104                 {linkedListDyn.c:19;
105                    linkedListDyn.c:20;
106                        linkedListDyn.c:28;
107                                          }) }}
108                 my try as ACSL predicate: \true
109 [value-wrapper] Stmt id 18: curr = curr->next;
110                 ---
111                 curr->next in {{ garbled mix of &{alloced_return_malloc; }
112                 (origin: Misaligned
113                    {
114                      linkedListDyn.c:19;
115                          linkedListDyn.c:20;
116                     }) }}
```

## 7.2 LIMITATIONS FIRST AND SECOND APPROACH

When the first approach in section 4.2 was done, some limitations appear at the moment of trying to prove that it was possible at the moment of injecting data. The procedure was done manually before running the Jessie-why analysis in order to get valid proofs. In the C codes the problems appear especially with arrays and memory allocation. The script done to automate the procedure was not completed specially in the step in which the .why files generated by the Jessie-why tool needed to be modified in order to link the variables. This approach was limited by this issue.

The second approach explained in section 4.3 has the limitation that even it was already avoided the procedure of linking variables manually there was still needed to be done some modifications in the .why files generated by the Jessie-why tool, so the script shell that automates this step was not completed at that moment.

## 7.3 LIMITATIONS LOOP UNROLLING

After testing the approaches in loop unrolling, the main limitation at that moment was also to automate the procedure. The procedure worked at this moment, but the modifications in the predicate definition inside the *.why* files needed to be done manually.

# 8 CONCLUCIONS

After developing some approaches for different code structures and test them with the tools mentioned on this thesis, it is possible to conclude that the combination of abstract interpretation and deductive methods is possible as it was proposed in section 4.0. The extraction of the data was one of the first steps of the project, the value analysis plug-in can be used to extract computed values from C codes considering the limitations presented until the creation of this thesis and previously explained. It is important to notice that Frama-C is still being an open source software and that some of the limitations found could not be a problem any more in the near future. It was proved that sequential codes were able to be manipulated inside the *.why* files with the approach explained in section 4.2 as well as the data injection using the ACSL annotations for defining the predicates that could be applied later on the goals.

*Predicates* and *asserts* were used in order to avoid the problems of linking the variables defined by the Jessie-why plug-in and the original ones with the help of ACSL as explained in section 4.4. In this approach, *assumes* were used to prove a pre-condition. The declaration of the *predicate* in the C code was used in order to define the parameters directly inside the *.why* file. To automate the process a plug-in must be done containing the commands typed directly from the terminal. Shell scripts could be used in order to be called later by the final plug-in that will combine both plug-ins (value analysis and Jessie plug-in).

Abstract interpretation [3] and the use of weakest precondition Floyd-Hoare [4] are two different techniques that could be implemented in a couple of years in order to do it right. With the help of the proposed methods in this thesis, the implementation could be achieved to improve the precision of the analysis of software used in industrial processes inside embedded systems.

In the case of applying the approaches to conditional codes, satisfactory results were obtained given that they present valid results also at the moment of testing them with codes that include arrays. In other words, it is possible to conclude that the best way to split a conditional statement is to consider the branches in new function. Doing this a sequential split could be done in order to generate a pre-condition. The loop unrolling approach presented in section 6.4 is also working based in the principle that a loop invariant is used. The body of the loop is copied inside any function to create a new function in order to split it sequentially later.

As it can be seen the thesis requires testing different C code structures. It was tried to cover most of the cases as possible. Simple examples were used in order the approaches then can be tested in more complex codes. Finally, it is concluded that the results got during the development of this thesis prove the static analysis could be combined in a practical manner, the studies can be followed in order to automate the process and that it can finally be used in embedded systems, containing critical software parts.

# References


[1] Patrick Baudin, Jean C. Fillitre, Thierry Hubert, Claude March, Benjamin Monate, Yannick Moy, and Virgile Prevosto, "ACSL: ANSI C Specification Language (preliminary design V1.2)", preliminary edition, May 2008.

[2] CEA LIST and LRI, "Frama-c", http://www.frama-c.cea.fr/.

[3] P. Cousot, "Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, Program Flow Analysis: Theory and Applications, chapter 10, pages 303342", rentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[4] C.A.R Hoare, "An axiomatic basis for computer programming", Commu. ACM, 12(10):576-580, 1969.

[5] E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, 1976.

[6] Pascal Cuoq with Virgile Prevosto, "Frama-cs value analysis plug-in 20100401 Boron version ", 2010 CEA LIST, Frama-c. http://www.frama-c.com/.

[7] Claude March, Yannick Moy, "Jessie Plugin Tutorial Frama-C version: Boron Jessie plugin version: 2.26", May 31, 2010 INRIA, Frama-c. http://www.frama-c.com/.