Documentation of Frama-C's value analysis plug-in

Pascal Cuoq with Virgile Prevosto
CEA LIST, Software Reliability Lab.
Contact: http://frama-c.cea.fr

December 12, 2008

Contents

Chapter 1

Introduction

Frama-C is a framework where static analysis tools for C programs can be built as plug-ins. This manual documents the value analysis plug-in of Frama-C. This plug-in uses abstract interpretation techniques to computes sets of possible values for the variables of the program.

1.1 First contact

The interactive graphical interface of Frama-C displays a normalized version of the analyzed code source. In this interface, the value analysis plug-in allows the user to select an expression in the code and observe an over-approximation of the set of the possible values taken by this expression at run-time. The value analysis works best on embedded code or embedded-like code without dynamic allocation nor multithreading.

Here is a simple C example:

```
int y, z=1;
int f(int x) {
  y = x + 1;
  return y;
}

void main(void) {
  for (y=0; y<2+2; y++)
    z=f(y);
}
```

The value analysis plug-in is able to guarantee that at each passage through the `return` statement of function `f`, the global variables `y` and `z` each contain either 1 ou 3. At the end of function `main`, it indicates that `y` necessarily contains 4, and the the value of `z` is again 1 or 3. When the plug-in indicates that the value of `y` is 1 or 3 at the end of function `f`, it implicitely computes the union of all the values that can be stored in `y` at each passage through this program point throughout an execution. In an actual execution of this deterministic program, there is only one passage though the end of function `main`, and therefore only one value for `z` at this point. The answer given by the value analysis is approximated but correct (the actual value, 3, is among the proposed values).

1.2 Run-time errors and the absence thereof

The analyzed application can contain run-time errors (divisions by zero, invalid pointer access,...), as in the case of the following program:

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

The value analysis emits a warning about an out-of-bound access at line 5:

```
rte.c:5: Warning: accessing out of bounds index.    \
         assert ((0 <= i) && (i < 10));
```

There is in fact an out-of-bounds access at this line in the program. It can also be the case that, because of the approximations made throughout its computations, Frama-C emits warnings for constructs that do not cause any run-time errors. These are called “false alarms”. Conversely, it should be noted that the fact that the value analysis computes correct, over-approximated sets of possible values prevents it to remain silent on a program that contains a run-time error.

1.3 Other analyses based on the value analysis

Frama-C also provides synthetic information on the behavior of analyzed functions: inputs, outputs, dependencies... This information is computed by making use of the results of the value analysis plug-in, and therefore some familiarity with the value analysis is necessary to use these features properly.

Chapter 2

What the value analysis provides

2.1 Values

2.1.1 Value domains

When the plug-in receives a query in one way or another concerning the value of a variable `x` at a given program point, it answers it by providing an over-approximation of the set of values possibly taken by `x` at this point for all possible executions. This set can take one of the following shapes:

- a set of integers, represented as:
  - ▷ an enumeration, `{`v_1`;` ... v_n`;}`,
  - ▷ an interval, `[`$m..M$`]`, that represents all the integers comprised between m and M. If `--` appears as the lower bound m (resp. the upper bound M), it means that the lower bound (resp upper bound) is $-\infty$ (resp. $+\infty$),
  - ▷ an interval with periodicity information, `[`$m..M$`],`r`%`p, that represents the set of values comprised between m and M that are equal to r modulo p (in other words, whose remainder in the Euclidean division by p is equal to r) ;
- a floating-point number or an interval of floating-point numbers:
  - ▷ f for the non-zero floating-point number f (the floating-point number `+0.0` has the same representation as the integer `0` and is identified with it),
  - ▷ `[`f_m .. f_M`]` for the interval from f_m to f_M inclusive ;
- a set of adresses denoted by `{{`a_1`;` ... a_n`;}}`, each a_i is of the form:
  - ▷ $\&x + D$, where $\&x$ is the base address corresponding to the C variable x, and D is in the domain of integer values and represents the possible offsets **expressed in bytes** with respect to the base address $\&x$,

▷ $NULL + D$, which is another notation for the set of integers D ;

- *garbled mix of* &{x_1; ... x_n;}, denoting an unknown value that was built from applying arithmetic operations to the base addresses of variables x_1 and x_2 and to integers. This notation represents the closure by arithmetic operations of the sets of addresses $\&x_1 +$ `[--..--]`, $\&x_n +$ `[--..--]`, $NULL +$ `[--..--]`.

- *ANYTHING*, that represents a completely unknown address.

Modern compilation platforms for the C language unify integer values and absolute addresses: there is not difference between the encoding of the integer 256 and that of the address `(char*)0x00000100`. Therefore, the value analysis does not make any difference between these two values either.

In floating-point computations, the value analysis considers that obtaining `Nan`, `+infinity`, or `-infinity` is an unwanted error. If it appears that these results can be produced by a given floating-point operation, the analyzer emits an alarm that excludes these possibilities, and continues the analysis with an interval representing the result obtained if excluding these possibilities. This interval, like every other results, may be over-approximated. Similarly, an alarm may be emitted for the use as a floating-point number of a value that does not ostensibly represent a floating-point number. This situation can happen for instance if a union type with both an `int` field and a `float` field is used, or in the case of a conversion from `int*` to `float*`. The emitted alarm excludes the possibility of the bit sequence used as a floating-point number representing `Nan`, an infinite, or an address.

Warning: the offsets with respect to the base addresses are expressed in bytes, regardless of the type of the C variable that is being considered.

Examples of value domains:

- `[1..256]` represents the set of integers comprised between 1 and 256, each of which can also be interpreted as an absolute address between 0x1 and 0x100.

- `[0..256],0%2` represents the set of even integers comprised between 0 and 256. This set is also the set of the addresses of the first 129 aligned 16-bit words in memory.

- `[1..255],1%2` represents the odd integers comprised between 1 and 255.

- `[--..--]` represents the set of all (possibly negative) integers.

- `3.` represents the floating-point number `3.0`.

- `[-3. .. 9.]` represents the interval of floating-point values comprised between `-3.0` and `9.0`.

- `{{ &x + { 0; } ; }}` represents the address of the variable `x`.

- `{{ &x + { 0; 1; } ; }}` represents the address of one of the first two bytes of variable `x` – assuming `x` is of a type of size at least 2 bytes. Otherwise, this notation represents a set containing the address of `x` and an invalid address.

- `{{ &x + { 0; } ;  &y + { 0; } ; }}` represents the addresses of `x` and `y`.

- `{{ &t + [0..256],0%4 ; }}`, in an application where `t` is declared as an array of 32-bit integers, represents the addresses of locations `t[0]`, `t[1]`, ..., `t[64]`.

- `{{ &t + [0..256] ; }}` represents the same values as the expression `(char*)t+i` where the variable `i` has an integer value comprised between 0 and 256.

- `{{ &t + [--..--] ; }}` represents all the address obtained by shifting `t`, including misaligned and invalid ones.

2.1.2 Origins of approximations

The values that are the result of heavy approximations contain information on the origin of said approximations. The notation `V (origin: O)` indicates that the computations of the value `V` required heavy approximations. The text provided instead of `O` indicates the location and the cause of some of these approximations. An origin `O` can be one of the following:

Misaligned read

The origin `Misaligned L` indicates the set `L` of the lines of the application where misaligned reads prevented the computation to be precise. A misaligned read is a memory read-access where the bits read were not previously written as a single write that modified the whole set of bits exactly. An example of a program leading to a misaligned read is the following:

```
1  int x,y;
2  int *t[2] = { &x, &y };
3
4  int main(void)
5  {
6    return 1 + (int) * (int*) ((int) t + 2);
7  }
```

The value returned by the function `main` is
`{{ garbled mix of &{  x; y; } (origin: Misaligned {  misa.c:6; }) }}`.
Note that this result is obtained with the analyzer configured for a 32-bit architecture, and that the read memory access is not an out-of-bound access. If it was, it would cause an alarm to be emitted. The read access remains within the bounds of array `t`, but the 32-bit word read is made of two bytes from the first cell, and two bytes from the second cell of `t`.

Call to an unknown function

The origin `Library function L` is used for the result of recursive functions or calls to function pointers whose value is not known precisely.

Fusion of values with different alignments

The notation `Merge L` indicates a set `L` of lines of the analyzed code where fusions of memory states containing values with incompatible alignments take place. In the example below, the memory states from the `then` branch and from the `else` branch contain in base `t` 32-bit adresses with incompatible alignments.

```
 1  int x,y;
 2  char t[8];
 3
 4  int main(int c)
 5  {
 6    if (c)
 7      * (int**) t = &x;
 8    else
 9      * (int**) (t+2) = &y;
10    x = t[2];
11    return x;
12  }
```

The value returned by function `main` is
`{{ garbled mix of &{  x; y; } (origin: Merge {  merge.c:9; }) }}`.

Arithmetic operation

`Arithmetic L` indicates the set `L` of lines where arithmetic operations take place without the analyzer being able to represent the result precisely.

```
1  int x,y;
2  int f(void)
3  {
4    return (int) &x + (int) &y;
5  }
```

In this example, the return value for `f` is
`{{ garbled mix of &{  x; y; } (origin: Arithmetic {  ari.c:4; }) }}`.

2.2 Log messages emited by the value analysis

2.2.1 Proof obligations

The correctness of results relies on the verification of all the proof obligations generated during the analysis. In the current version of Frama-C, these proof obligations are displayed as messages that start with `Warning:...` and contains either the word `alarm` or `assert`. Such a message contains, or should contain for those which are not yet expressed as ACSL assertions, the nature and the origin of the obligation. For those alarms which are expressed as ACSL assertions, it is also possible to obtain a version of the source code annotated with the proofs obligations.

For instance, when dividing by an expression that the value analysis is not able to guarantee to be different from zero, a proof obligation is emitted. This obligation expresses that the divisor is different from zero at this point of the code.

In the particular case where zero is the only possible value for the divisor, the analysis stops for this branch. If the divisor seems to be able to take non-zero values, the analyzer is allowed to take into account the property that the divisor is different than zero when it continues the analysis after this point. It may also not take it into account, because the property is not easy to use, or just because.

Proof obligations can similarly be emitted for memory accesses, and for pointer comparisons which might break the hypotheses the memory model relies on.

Note: while the logical formulas emitted by the plug-in use ACSL's syntax, some explicit coercion operations may be missing from these formulas to make them express correctly in ACSL the condition that ensures the absence of error. This bug will be fixed in a later version.

2.2.2 Informative messages regarding the loss of precision

Some messages may warn that the analysis is making an operation that causes now or later a loss of precision. These messages are not proof obligations and it is not mandatory for the user to act on them. They are intended to help the user trace the results of the analysis, and to help him find where to help the analyzer when the results are not as precise as she hoped.

2.2.3 Informative messages regarding the experimental status of some features

Some messages may warn that a feature is experimental. This means that a part of the analyzer that gets used during the analysis is less tested or is known to have issues.

2.3 Interactive and programatic interfaces

Chapter 3

Limitations and specificities

This chapter describes how the difficult constructs in the C language are handled in the value analysis. It is with the constructs listed here that the limitations of all static analyzers in general show. Different static analysis techniques can often be positioned with respect to each other by looking only at how they handle loops, function calls and partial codebases.

3.1 Loops

The analysis of a source program always take a finite time. The fact that the source code contains loops, and that some of these loops do not terminate, can never induce the plug-in to loop forever itself[1]. In order to obtain this property, the analysis of a loop may be the place of an approximation.

Let us assume, in the following lines, that the function c is unknown:

```
n=100;
i=0;
y=0;
do {
  i++;
  if (c(i))
    y = 2*i;
  } while (i<n);
```

The value analysis plug-in could provide the best possible solution if the user instructed it explicitely to study step by step each of the hundred loop iterations. Without any such instruction from the user, the plug-in analyses the body of the loop much less than one hundred times. It is able to provide the approximated, but correct, information that after the loop, y contains an even number between 0 and 256.

[1] There are two exceptions to this rule, which are documented in the reference manual. The plug-in can loop if it is launched on a program with non-natural loops, or if the most precise modelizations of malloc are used.

Section 4.2 introduces the different ways in which the user can influence the plug-in's stategy with respect to the analysis of loops.

3.2 Functions

Without special instructions from the user, function calls are handled as if the body of the function had been expansed at the call site. In the following example, the body of f is analyzed again at each analysis of the body of the loop. The result of the analysis is as precise as the result obtained for the example in section 3.1.

```
int n, y;
void f(int x) { y = x; }

void main_1(void) {
  int i;

  n=100;
  i=0;
  y=0;
  do {
   i++;
   if (c(i))
     f(2*i);
   } while (i<n);

}
```

Recursive functions are allowed in Frama-C, but they are not handled in the current version of the value analysis plug-in.

3.3 Analyzing a partial or a complete application

The default behavior of the value analysis plug-in allows to analyze complete applications, that is, applications for which the source code is entirely available. In practice, it is sometimes desirable to limit oneself to critical subparts of the application, by using other entry points than the actual one (the main function). Besides, the code of some functions called from the application may not be available (library functions for instance). The plug-in can be used, usually with more work, in all these circumstances. The options for specifying the entry point of the analysis are detailed in the reference manual, section 4.3.

3.3.1 Entry point of a complete application

When the source code for the analysed application is entirely available, the only additional information expected by the plug-in is the entry point that the analysis should start from. Specifying the wrong entry point can lead to incorrect results. For instance, let us assume that the actual entry point for the example of section 3.2 is not the function `main_1` but the following `main_main` function:

```
void main_main(void) {
  f(15);
  main_1();
}
```

If the wrong entry point `main_1` is indicated, the plug-in will provide the same answer for variable `y` at the end of function `f` as in sections 3.1 and 3.2: the set of even numbers between 0 and 256. This set is not the expected answer if the actual entry point is the function `main_main`, because it does not contain the value 15.

The entry point of the analysed application can be specified on the command line using the option `-main`.

3.3.2 Entry point of an incomplete application

It is possible to analyze an application without starting from its actual entry point. This can be made necessary because the actual entry point is not available, for instance if the analysis is concerned with a library. It can also be a deliberate choice as part of a modular verification strategy. In this case, the option `-lib-entry`, described at section 4.4.3, should be used together with the option `-main` that sets the entry point of the analysis. In this mode, the plug-in does not assume that the global variables have kept their initial values (except for the variables with the `const` attribute).

3.3.3 Library functions

Another category of functions whose code may be missing is composed of the functions called from the application, such as the operating system primitives or the external libraries used by the application. These functions are called “library functions”. The behavior of each library function can be specified through annotations (see chapter 6). The specification of a library function can in particular be provided in terms of modified variables, and of data dependencies between these variables and the inputs of the function (section 6.2).

3.3.4 Applications relying on software interrupts

The current version of the value analysis plug-in is not able to take into account interrupts (auxiliary function that can be executed at any time during the main computation). As things stand, the plug-in may give answers that do not reflect reality if interrupts play a role in the behavior of the analyzed application. Interrupts will be taken into account in a future version.

3.3.5 Consequences of the choice between the complete application mode and the partial application mode

This section shows a small example to illustrate the pitfalls that should be considered when using the value analysis with an incomplete part of an application. This example is simplified but quite typical. This is the pattern followed by the complete application:

```
int ok1;

void init1(void) {
  ...
  if (error condition)
    error_handling1();
  else
    ok1 = 1;
}

void init2(void) {
  if (ok1) {
    ...
    }
}

void main(void) {
  init1();
  init2();
  ...
}
```

If init2 is analyzed as the entry point of a complete application, or if the function init1 is accidentally omitted, then at the time of analyzing init2, the value analysis will have no reason to believe that the global variable ok1 has not kept its initial value 0. The analysis of init2 will consist in determining that the value of the if condition is always false, and to ignore all the code that follows. Any threat in this function will therefore be missed by the analyzer. However, as long as the user is aware of these pitfalls, the analysis of incomplete sources can provide useful results. In this example, if the user wishes to analyze the function init2, he or she should use the option described in section 4.4.3.

It is also possible to use annotations to describe the state in which the analysis should be started as a precondition for the function used as entry point. The syntax and usage of preconditions is described in section 6.1. The user should pay attention to the intrinsic limitations in the way the value analysis interprets these properties (section 6.1.2).

Despite these limitations, when the specifications the user wishes to provide are simple enough to be interpreted by the plug-in, it becomes possible and useful to divide the application into several parts, and to study each part separately (by taking each part as an entry point, with the appropriate preconditions). The division of the application into parts may follow the phases in the application's behavior (initialization followed by the permanent phase) or divide it into elementary sub-pieces, the same way unit tests do.

3.4 Conventions not specified by the ISO standard

The value analysis can provide useful information even for low-level programs that rely on non-portable C construct and that depend on the size of the word and the endianness of the target architecture.

3.4.1 The C standard and its practice

There exists constructs of the C language which the ISO standard does not specify, but which are compiled in the same way by almost every compiler for almost every architecture. For some of these constructs, the behavior of the plug-in is to assume a reasonable compiler and target architecture. This way, it is possible to obtain more information about the behavior of the program than would be possible using only what is strictly guaranteed by the standard.

This stance is paradoxical for an analysis tool whose purpose is to compute only correct approximations of program behaviors. Then notion of "correctness" is necessarily relative to a definition of the semantics of the analyzed language. And, for the C language, the ISO standard is the only available definition.

However, an experimented C programmer has a certain mental model of the working habits of the compiler. This model has been acquired by experience, common sense, knowledge of the underlying architectural constraints, and sometimes the perusal of the generated assembly code. Finally, the Application Binary Interface may constrain the compiler into using representations that are not mandated by the C standard (and which the programmer should not, *a priori*, have counted on). Since most compilers make equivalent choices, this model does not vary much from one programmer to the other. The set of practices admitted by the majority of C programmers compose a kind of informal, and unwritten, standard. For each C language construct that is not completely specified by the standard, there usually exists an alternative, "portable" version. The portable version could be considered safer if the programmer did not know exactly how the non-portable version will be translated by his, or her, compiler. But the portable version may produce a code which is significantly slower and/or bigger. In practice, the constraints imposed on embedded software often lead to choosing the non-portable version. This is why, as often as possible, the value analysis uses the same standard as the one used by programmers, the unwritten one. It is the experience gained on actual industrial software, during the development of early versions of Frama-C as well as during the development of other tools, that led to this choice.

The hypotheses discussed here have to do with the conversions between integers and pointers, pointer arithmetic, the representation of `enum` types and the relations between the addresses of the fields of a same `struct`. As a very concrete example, the value analysis plug-in assumes two-complement arithmetic, which the standard does not guarantee, and whose consequences can be seen when converting between signed and unsigned types.

3.4.2 Positioning compilation parameters

Using one of the pre-configured target platforms

The option `-machdep` *platform* sets a number of parameters for the low-level description of the target platform, including the *endianness* of the target and size of each C type. The option `-machdep help` provides a list of currently supported platforms. The default is `x86_32`, an IA32 processor with what are roughly the default compilation choices of gcc.

Targeting a different platform

If you are interested in a platform that is not listed, please contact the developers. An autodetection program can be provided in order to check the hypotheses mentioned in section 3.4.1, as well as to detect the parameters of your platform. It comes under the form of a C program of a few lines, which should ideally be compiled with the same compiler as the one intended to compile the analyzed application. If this is not possible, the analysis can also be parametrized manually with the characteristics of the target architecture.

Often, the ISO standard does not provide enough guarantees to ensure that the behaviors of the compiler during the compilation of the autodetection program and during the compilation of the application are the same. It is the additional constraint that the compiler should conform to a fixed ABI that ensures the reproducibility of compilation choices.

3.5 Memory model – Bases separation

This section introduces the abstract representation of the memory the value analysis relies on. It is necessary to have at least a superficial idea of this representation in order to interact with the plug-in.

3.5.1 Base address

The memory model used by the value analysis relies on the classical notion of "base address". Each variable, be it local or global, defines one and only one base address. For instance, the definitions

```
int x;
int t[12][12][12];
int *y;
```

define three base addresses, for x, t, and y respectively. The sub-arrays composing t share the same base address. The variable y defines a base address that corresponds to a memory location expected to contain an address. On the other hand, there is no base address for *y, even though dynamically, at a given time of the execution, it is possible to refer to the base address corresponding to the memory location pointed to by y.

3.5.2 Address

An address is represented as an offset (which is an integer) with respect to a base address. For instance, the addresses of the subarrays of the array t defined above are expressed as various offsets with respect to the same base address.

3.5.3 Bases separation

The strongest hypothesis that the plug-in relies on is about the representation of memory and can be expressed in this way: **It is possible to pass from one address to another through the addition of an offset, if and only if the two addresses share the same base address**.

This hypothesis is not true in the C language itself : addresses are represented with a finite number of bits, 32 for instance, and it is always possible to compute an offset to go from on address to a second one by considering them as integers and subtracting the first one from the second one. The plug-in generates all the alarms that ensure, if they are checked, that the analyzed code fits in this hypothesis. On the following example, it generates a proof obligation that means that "the comparison on line 8 is safe only if p is a valid address or if the base address of p is the same as that of &x".

```
 1  int x, y;
 2  int *p = &y;
 3
 4  void main(int c) {
 5    if (c)
 6      x = 2;
 7    else {
 8      while (p != &x) p++;
 9      *p = 3;
10      }
11  }
```

It is mandatory to check this proof obligation. When analyzing this example, the analysis infers that the loop never terminates (because p remains an offset version of the address of y and can never be equal to the address of x). It concludes that the only possible value for x at the end of function main is 2, but this answer is provided *proviso quod* the proof obligation is verified through other means. Some actual executions of this example could lead to a state where x contains 3 at the end of main: only the proof obligation generated by the plug-in and verified by the user allows to exclude these executions.

In practice, the hypothesis of base separation is unavoidable in order to analyze efficiently actual programs. For the programs that respect this hypothesis, the user should simply verify the generated proof obligations to ensure the correctness of the analysis. For the programs that voluntarily break this hypothesis, the plug-in produces proofs obligations that are impossible to lift: this kind of program can not be analyzed with the value analysis plug-in.

Here is an example of code that voluntarily breaks the base separation hypothesis. Below is the same function written in the way it should have been in order to be analyzable with Frama-C.

```
int x,y,z,t,u;

void init_non_analyzable(void)
{
  int *p;
  // initialize variables with 52
  for (p = &x; p <= &u; p++)
    *p = 52;
}

void init_analyzable(void)
{
  x = y = z = t = u = 52;
}
```

Chapter 4

Parameterizing the analysis

4.1 Command line

The parameters that determine Frama-C's behavior can be set through the command line. The command to use to launch the tool is:

```
frama-c-gui <options> <files>
```

The options that are understood by the value analysis plug-in are described in this chapter. The files are the C files containing the source code to analyze. The executable can be named `frama-c-gui` (or `frama-c-gui.exe` for Windows) if it is the version including the graphical user interface, or `frama-c` (or `frama-c.exe`) if it is the "batch" version. In both cases the use of options is identical.

4.1.1 Analyzed files and preprocessing

The analyzed files should be written in the C language. The files that do not use the `.i` extension are automatically preprocessed. The preprocessing command used by default is:

```
gcc -C -E -I.
```

It is possible that the files that do not use the `.c` extension fail to pass this stage. It is notably the case with `gcc`, to which the option `-x c` should be passed in order to analyze C files that do not have a `.c` extension. It is also possible to use another preprocessor.

The option `-cpp-command <cmd>` sets the preprocessing command to use. If the patterns `%1` and `%2` do not appear in the text of the command, the preprocessor is invoked in the following way:

```
<cmd> -o <outputfile> <inputfile>
```

In the cases where it is not possible to invoke the preprocessor with this syntax, it is possible to use the patterns `%1` and `%2` in the command's text as placeholders for the input file (respectively, the output file). Here are some examples of use of this option:

```
frama-c-gui -val -cpp-command 'gcc -C -E -I. -x c' fic1.src fic2.i
frama-c-gui -val -cpp-command 'gcc -C -E -I. -o %2 %1' fic1.c fic2.i
frama-c-gui -val -cpp-command 'copy %1 %2'  fic1.c fic2.i
frama-c-gui -val -cpp-command 'cat %1 > %2' fic1.c fic2.i
frama-c-gui -val -cpp-command 'CL.exe /C /E %1 > %2' fic1.c fic2.i
```

4.1.2 Activating the value analysis

The option `-val` activates the value analysis, and causes the values obtained for the variables at the end of each analyzed function to be displayed on the standard output.

Currently, many other functionalities provided by Frama-C rely on the computations made by the value analysis. The use of an option that relies on the results of the value analysis automatically causes the computations to be made, without it being necessary to provide the `-val` option on the command-line. It should be kept in mind that, in this case, all the other options of the value analysis remain available for tailoring its behavior and making it contribute as much as possible to the end result.

4.1.3 Saving the result of an analysis

The option `-save s` saves the state of the analyzer, after the analysis has completed, in a file named `s`. The option `-load s` lets the state saved in file `s` be loaded back into memory for visualization.

Example :

```
frama-c -val -deps -out -save result fic1.c fic2.c
frama-c-gui -load result
```

4.2 Treatment of loops

4.2.1 Controlling approximations

The default treatment of loops by the analyzer may produce results that are too approximate. This section details how precision can be improved by tuning the parameters for the treatment of loops.

When encountering a loop, the analyzer tries to compute a state that contains all the actual concrete states that may happen at run-time, including the initial concrete state just before entering the loop. This englobing state may be too imprecise by construction: typically, if the analyzed loop is initializing an array, the user does not expect to see the initial values of the array appear in the state computed by the analyzer. The solution in this case is to use one of the two unrolling options, as described in section 4.2.2.

As compared to loop unrolling, the advantage of the computation by accumulation is that it generally requires less iterations than the number of iterations of the analyzed loop.

The number of iterations does not need to be known (for instance, it allows to analyze a `while` loop with a complicated condition). In fact, this method can be used even if the termination of the loop is unclear. These advantages are obtained thanks to a technique of successive approximations. The approximations are applied individually to each memory location in the state. This technique is called "widening". Although the analyzer uses heuristics to figure out the best parameters in the widening process, it may (rarely) be appropriate to help it by providing it with the bounds that are likely to be reached, for a given variable modified inside a loop.

Stipulating bounds

The annotation `//@ loop pragma WIDEN_HINTS v1, ..., vn, e1, ..., em ;` may be placed before a loop, so as to make the analyzer use preferably the values `e1`, . . . , `em` when widening the sets of values attached to variables `v1`, . . . , `vn`.

If this annotation does not contain any variable, then the values `e1`, . . . , `em` are used as bounds for all the variables that are modified inside the loop.

Example:

```
int i,j;

void main(void)
{
  int n = 13;
  /*@ loop pragma WIDEN_HINTS i, 12, 13; */
  for (i=0; i<n; i++)
    {
      j = 4 * i + 7;
    }
}
```

4.2.2 Loop unrolling

There are two different options for forcing the value analysis to unroll the effects of the body of the loop, as many times as specified, in order to obtain a precise representation of the effects of the loop itself. If the number of iterations is sufficient, the analyzer is thus able to determine that each cell in the array is initialized, as opposed to the approximation techniques from the previous section.

Syntactic unrolling

The option `-ulevel n` indicates that the analyzer should unroll the loops syntactically `n` times before starting the analysis. If the provided number `n` is larger than the number of iterations of the loop, then the loop is completely unrolled and the analysis will not observe any loop in that part of the code.

Providing a large value for `n` makes the analyzed code bigger: this may cause the analyzer to use more time and memory. This option can also make the code exponentially bigger in presence of nested loops. A large value should therefore not be used in this case.

It is possible to control the syntactic unrolling for each loop in the analyzed code thanks to the annotation `//@ loop pragma UNROLL n;`. This annotation should be placed in the source code for the application, at the point that precedes the loop which should be unrolled `n` times. The annotation `loop pragma UNROLL` has priority over the option `-ulevel`.

Semantic unrolling

The option `-slevel n` 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. An effect of this option is that the states corresponding to the first, second,. . . iterations in the loop remain separated, as if the loop had been unrolled.

The number which should be passed to this option depends on the nature of the control flow graph of the function to analyze. If the only control structure is a loop of `m` iterations, then `-slevel m` allows to unroll the loop completely. The presence of other loops or of `if-then-else` constructs multiplies the number of paths a state may correspond to, and thus the number of states it is necessary to keep separated in order to unroll a loop completely. For instance, the nested simple loops in the following example require the option `-slevel 54` in order to be completely unrolled:

```
int i,j,t[5][10];

void main(void)
{
  for (i=0;i<5;i++)
    for (j=0;j<10;j++)
      t[i][j]=1;
}
```

When the loops are sufficiently unrolled, the result obtained for the contents of array `t` are the optimally precise:

```
t[0..4][0..9] ∈ {1; }
```

The number to pass the option `-slevel` is of the magnitude of the number of values for `i` (the 6 integers between 0 and 5) times the number of possible values for `j` (the 11 integers comprised between 0 and 10). If a value much lower than this is passed, the result of the initialization of array `t` will only be precise for the first cells. The option `-slevel 27` gives for instance the following result for array `t` :

```
t{[0..1][0..9]; [2][0..4]; } ∈ {1; }
 {[2][5..9]; [3..4][0..9]; } ∈ {0; 1; }
```

In this result, the effects of the first iterations of the loops (for the whole of `t[0]`, the whole of `t[1]` and the first half of `t[2]`) have been computed precisely. The effects on the rest of `t` were computed with approximations. Because of these approximations, the analyzer can not tell if each of those cells was initialized (contains 1) or not (still contains its initial value 0).

4.3 Treatment of functions

4.3.1 Reusing the analysis of a function

If the user notices, in the application he or she is studying, a function `f` whose analysis takes a long time, while the impact of this function on the behavior of the application as a whole remains limited, it is possible for him/her to launch the analysis with the option `-mem-exec f`. The analyzer will then analyze 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 will make the analysis:

- faster, and
- less precise concerning everything affected by `f`.

If the function `f` has pointers as inputs, the generic analysis uses the adresses of specially allocated non-aliasing locations as the values for these pointers, similarly to what is done for the formal arguments of the entry point of a complete application (section **??**). During the actual analysis of the application, each time a call to function `f` is encountered, the analyzer determines if the state at the call site can be seen as an instance of the generic state, and re-uses the results of the generic analysis if this is the case.

This option has not been intensively tested, and should be considered as experimental at this point. The computation of the values of expressions is supposed to be correct when it is used. However, there are known problems making the computations of inputs, outputs and dependencies (section 5) incorrect for the functions that call a function analyzed with this option.

4.3.2 Dealing with library functions

When the value analysis plug-in encounters a call to a library function, it makes assumptions about the effects of this call. If no ACSL contract is provided for the function, it uses the type of the function as its source of information for making informed assumptions.

The behavior of library functions can be described more precisely by writing a contract for the function (chapter 6), especially using an assigns clause (section 6.2).

4.4 Describing the codebase

4.4.1 Specification of the entry point

The option `-main f` specifies that `f` should be used as the entry point for the analysis. If this option is not specified, the analyzer uses the function called `main` as the entry point.

4.4.2 Analysis of a complete application

By default (when the option `-lib-entry` is *not* set), the analysis starts from a state in which initialized global variables contain their initial values, and uninitialized ones contain zero. This only makes sense if the entry point (see section 4.4.1) is the actual starting point of this analyzed application. Each formal argument of the function used as entry point contains in the initial state a non-deterministic value that corresponds to its type. Non-aliasing locations are generated for the arguments with a pointer type, and the value of the pointer argument is the union of the address of this location and of `NULL`. For chain-linked structures, the allocation of such locations is done only to a fixed depth.

4.4.3 Analysis of an incomplete application

The option `-lib-entry` specifies that the analyzer should not use the initial values for globals in its initial state (except for those qualified with the keyword `const`). With this option, the analysis starts with an initial state where the integer components of global variables (without the `const` qualifier) and parameters of `f` are initialized with a non-determinitic value of their respective type. Their components of pointer types contain the non-deterministic superposition of `NULL` and of the addresses of special non-aliasing locations allocated by the analyzer, similarly to what is done for formal arguments of the entry point when analyzing a complete application.

4.5 Parameterizing the modelization of the C language

When Frama-C's user has *a priori* knowledge concerning the behavior of the analyzed code, this knowledge can sometimes be put to use to adjust the modelization to the analyzed code, and improve the results. Currently, only two such general properties about the analyzed code can be translated into modelization parameters:

- valid absolute addresses in memory, and
- the absence of arithmetic overflow during the program's computations.

Generally speaking, if these options are activated for the analysis of programs that do not respect the corresponding restrictions, the analysis will produce incorrect results **without emitting any warning**. This kind of option should therefore be used carefully.

Valid absolute addresses in memory

By default, the value analysis assumes that the absolute addresses in memory are all invalid. This assumption can be too restrictive, because in some cases there exist a limited number of absolute addresses which are intended to be accessed by the analyzed program, for instance in order to communicate with hardware.

The option `-absolute-valid-range m-M` specifies that the only valid absolute addresses (for reading or writing) are those comprised between `m` and `M` inclusive. This option currently allows to specify only a single interval, although it could be improved to allow several intervals in a future version.

Absence of arithmetic overflows

The option `-no-overflow` instructs the analyzer to assume that integers are not bounded and that the analyzed program's arithmetic is exactly that of mathematical integers. This option should only be used for codes that do not depend on specific sizes for integer types and do not rely on overflows. For instance, the following program is analyzed as "non-terminating" in this mode.

```
void main(void) {
  int x=1;
  while(x++);
  return;
}
```

This option should only be activated when it is guaranteed that the sizes of integer types do not change the concrete semantics of the analyzed code. It can be difficult to be certain of this property. For instance, the following function's behavior is sensitive to the size of integers and to overflows:

```
int abs(int x) {
  if (x<0) x = -x;
  return x;
}
```

With the `-no-overflow` option, the result of this function is a positive integer, for whatever integer passed to it as an argument. This property is not true for a conventional architecture, where `abs(MININT)` overflows and returns `MININT`.

The option `-no-overflow` may be modified or suppressed in a later version of the plug-in.

Chapter 5

Inputs, outputs and dependencies

Frama-C can compute and display the inputs (adresses of locations read from), outputs (adresses of locations written to), and the dependencies between outputs and inputs, for each function. The options to use are `-out` for the display of locations written to by each function and `-deps` for the functional dependencies between outputs and inputs.

The inputs, outputs and dependencies computed are as of now incorrect if the option `-mem-exec` is used (section 4.3.1).

5.1 Dependencies

An example of dependencies as obtained with the option `-deps` is as follows:
`y FROM x; z; (and default:false)`

This clause means that in this example, the variable `y` may have changed at the end of the function, and that the variables `x` and `z` are used in order to compute the new value of `y`. The text `(and default:false)` means that `y` may not have kept its previous value, whereas `(and default:true)` would mean that `y` *may* have been modified, and that if it was, then its new value depends only on `x` and `z`.

The dependencies computed by `-deps` express relations between the values of modified variables when the function terminates and the values the input variables had when entering the function. This is illustrated in the following example:

```
int b,c,d,e;

void loop_branch(int a)
{
  if (a)
    b = c;
  else
    while (1) d = e;
}
```

The dependencies of function `loop_branch` are `b FROM c; (and default:false)`, which means that when the function terminates, the variable `b` has been modified and its new value depends on `c`. The variables `d` and `e` do not appear in the dependencies of `loop_branch` because they are only used in branches that do not terminate. A function for which the analyzer is able to infer that it does not terminate has empty dependencies.

The set of variables that appear on the right-hand side of the dependencies of a function are called the "functional inputs" of this function. In the example below, the dependency of `double_assign` is `a FROM c;`. The variable `b` is not a functional input because the final value of `a` depends only on `c`.

```
int a, b, c;

void double_assign(void)
{
  a = b;
  a = c;
}
```

5.2 Imperative inputs

The imperative inputs of a function are the locations that may be read during the execution of this function. The analyzer computes an over-approximation of the set of these locations with the option `-input`. For the function `double_assign` of the previous section, Frama-C gives `b; c;` as imperative inputs, which is the exact answer.

A location is accounted for in the imperative inputs even if it is read only in a branch that does not terminate. When asked to compute the imperative inputs of the function `loop_branch` of the previous section, Frama-C answers `c; e;`, which is again the exact answer.

5.3 Imperative outputs

The imperative outputs of a function are the locations that may be written to during the execution of this function. The analyzer computes an over-approximation of this set with the option `-out`. For the function `loop_branch` from above, Frama-C gives the imperative outputs `b; d;`, which is the exact answer.

5.4 Operational inputs

The name "operational inputs of a function" is currently given to the locations that have been read before having been written to in the cases where the function terminates. This notion may change in future versions.

As they stand, the operational inputs can be used in particular to decide which variables should be initialized at least in order to be able to execute the function, if it is known by other means that the execution terminates on the input values that are provided. An over-approximation of the operational inputs is computed with the option `-inout`.

```
int b, c, d, e, *p;

void op(int a)
{
  a = *p;
  a = b;
  if (a)
    b = c;
  else
    while (1) d = e;
}
```

This example, when analyzed with the options `-inout -lib-entry op`, is found to have the operational inputs `b; c; p; star_p;` for function `op`. The variable `p` is among the operational inputs, although it is not a functional input, because it is read (in order to be dereferenced) without having been previously overwritten. The variable `a` is not among the operational inputs because its value has been overwritten before being read. This means that an actual execution of the function `op` requires to initialize `p` (which has an influence on the execution by causing, or not, an an illicit memory access), whereas on the other hand the analyzer guarantees that it is not necessary to initialize `a`.

Chapter 6

Annotations

The language for annotations is ACSL (`http://www.frama-c.cea.fr/acsl.html`). Only a subset of the properties that can be expressed in ACSL can effectively be of service to or checked by the value analysis plug-in.

6.1 Preconditions, postconditions and assertions

6.1.1 Truth value of a property

When an annotation of the form precondition, postcondition or assertion is encountered by the analysis, it evaluates the truth value of the annotation in the current analysis state. The result of this evaluation can be:

- `valid`, indicating that the property is verified for the current state;
- `invalid`, indicating that the property is certainly false for the current state;
- `unknown`, indicating that the imprecision of the current state and/or the complexity of the property do not allow to conclude in one way or the other.

If a property obtains the evaluation `valid` every time the analyzer goes through the point to which it is attached, it means that it is valid under the hypotheses made by the analyzer. On the other hand, the evaluation `invalid` for a property may not necessarily indicate a problem: the property is false only for the state corresponding to the path that the analyzer is currently considering. It is possible that this path does not to occur for any real execution. The fact that the analyzer is considering this path may be a consequence of a previous approximation.

6.1.2 Reduction of the state by a property

After displaying its estimation of the truth value of a property *P*, the analyzer uses *P* to refine the current state. In other words, it relies on the fact that the validity of *P* will be established through other means, even if it itself is not able to ensure that the property *P* holds.

Let us consider for instance the following function, analyzed with the options `-val -slevel 12 -lib-entry f`.

```
 1  int t[10],u[10];
 2
 3  void f(int x)
 4  {
 5    int i;
 6    for (i=0; i<10; i++)
 7      {
 8        //@ assert x >= 0 && x < 10;
 9        t[i] = u[x];
10      }
11  }
```

Frama-C displays the following two warnings:

```
reduction.c:8: Warning: Assertion got status unknown.
reduction.c:8: Warning: Assertion got status valid.
```

The first warning is emitted at the first iteration through the loop, with a state where it is not certain that `x` is in the interval `[0..9]`. The second warning is for the following iterations. For these iterations, the value of `x` is in the considered interval, because the property has been taken into account at the first iteration and the variable `x` has not been modified since. Similarly, there are no warnings for the memory access `u[x]` at line 9, because under the hypothesis of the assertion at line 8, this access may not cause a run-time error. The only property left to be proved through other techniques is therefore the assertion at line 8.

Case analysis

When *semantic unrolling* is used (section 4.2.2), if an assertion is in the shape of a disjonction, then the reduction of the state by the assertion may be computed independently for each sub-formula in the disjonction. This multiplies the number of states in the same way that the analysis of the `if-then-else` does with semantic unrolling. Likewise, the states are kept separated only if the limit (the numerical value passed to option `-slevel`) has not been reached yet in that point of the program. This treatment may improve the analysis' precision. In particular, it can be used to provide hints to the analyzer, as shown in the following example.

```
int main(void)
{
  int x = Frama_C_interval(-10, 10);
4    //@ assert x <= 0 || x >= 0 ;
5    return x * x;
6  }
```

With the option `-slevel 2`, the analysis finds the result of this computation to be in `[0..100]`. Without the option, or without the annotation on line 4, the result found is `[-100..100]`. Both are correct, but the former is optimal considering the available information and the representation of large sets as intervals, while the latter is approximated.

Limitations

Attention should be paid to the two following limitations:

- a precondition or assertion only constrains the state that the analyzer has computed by itself. In particular in the case of a precondition for a function analyzed with the option `-lib-entry`, the precondition can only reduce the generic state that the analyzer would have used had there not been an annotation. It can not make the state more general. For instance, it is not possible to specify that there can be aliasing between two pointer arguments of the function analyzed with the option `-lib-entry`, because it would be a generalization, as opposed to a restriction, of the initial state generated automatically by the analyzer;
- the interpretation of an ACSL formula by the value analysis may be approximated. The state effectively used after taking the annotation into account is a superset of the state described by the user. In the worse case (for instance if the formula is too complicated for the analyzer to exploit), this superset is the same as the original state. It appears as if the annotation is not taken into account at all.

The two functions below illustrate both of these limitations:

```
1  int a;
2  int b;
3  int c;
4
5  //@ requires a == (int)&b || a == (int)&c;
6  int generalization(void)
7  {
8    b = 5;
9    *(int*)a = 3;
10 }
11
12 //@ requires a != 0;
13 int not_reduced(void)
14 {
15   return a;
16 }
```

If the analyzer is launched with the option `-lib-entry generalization`, the initial state generated for the analysis of function `generalization` contains an interval of integers (no adresses) for the variable `a` of type `int`.
The precondition `a == (int)&b || a == (int)&c` will probably not have the effect expected by the user: his/her intention appears to be to generalize the initial state, which is not possible.

If the analyzer is launched with the option `-lib-entry not_reduced`, the result for variable `a` is the same as if there was no precondition. The interval computed for the returned value, `[--..--]`, seems not to take the precondition into account because the analyzer can not represent the set of non-zero integers.

Note: the set of values computed by the analyzer remains correct, because it is a superset of the set of the value that can effectively happen at run-time with the precondition. When an annotation appears to be ignored for the reduction of the analyzer's state, it is not in a way that could lead to incorrect results.

6.2 The "assigns" clauses

Assign clauses in the contract of a function indicate which variables that may be modified by a function, and optionally the dependencies of the new values of these variables.

In the following example, the `assigns` clause indicates that the `withdraw` function does not modify any memory cell other than `p->balance`.

```
/*@ assigns p->balance;
  @*/
void withdraw(purse *p,int s) {
  p->balance = p->balance - s;
}
```

Assign clauses can be put to use to describe the behavior of functions for which the source code is not part of the analysis project. This happens when the function is really not available (we call these "library functions") or if it was purposelly removed because it did something that was complicated and unimportant for the analysis at hand.

Chapter 7

Primitives

It is possible to interact with the analyzer through the insertion in the analyzed source code of calls to special pre-defined functions. There are three reasons to want to insert one such call:

- emulating standard C library functions
- parameterizing the analysis
- observing the results of the analysis.

7.1 Standard C Library

The application under analysis may call functions such as `malloc`, `strncpy`, `atan`,... The source code for these functions is not necessarily available, as they are part of the system rather than of the application itself. In theory, it would be possible for the user to give a C implementation of these functions, but those implementations might prove difficult to analyze for the value analysis plug-in. A more pragmatic solution is to use a primitive function of the analyzer for each standard library call that would model as precisely as possible the effects of the call.

Currently, the primitive functions available this way are all inspired from the POSIX interface. It would however be possible to model other system interfaces. Existing primitives are described in the rest of this section.

7.1.1 `malloc` Function

The file `share/malloc.c` contains various models for the `malloc` function. To choose a given model, one of the following symbol, `FRAMA_C_MALLOC_INDIVIDUAL`, `FRAMA_C_MALLOC_POSITION`, `FRAMA_C_MALLOC_CHUNKS` or `FRAMA_C_MALLOC_HEAP`, must be defined before `#includ`ing this file. Their particularities are described in the `malloc.c` file itself.

Generally speaking, better results are achieved when each loop containing calls to malloc is entirely unrolled. Still, some models are more robust than the other ones if this condition is not met. FRAMA_C_MALLOC_POSITION is the most robust option with respect to loops that are not unrolled. If some loops containing calls to malloc are not entirely unrolled, FRAMA_C_MALLOC_INDIVIDUAL and FRAMA_C_MALLOC_CHUNKS might lead the analysis to enter an infinite computation.

```
#define FRAMA_C_MALLOC_INDIVIDUAL
#include "share/malloc.c"

void main(void)
{
  int * p = malloc(sizeof(int));
  ...
}
```

7.1.2 Mathematical Operations over Floating-Point Numbers

Few functions are currently available. Their prototype is given in share/math.h. In order to use these functions, share/math.c must be added on the command line.

7.1.3 String Manipulation Functions

Few functions are currently available. Their prototype is given in share/libc.h. In order to use these functions, share/libc.c must be added on the command line.

7.2 Parameterizing the Analysis

7.2.1 Adding some Non-Determinism

The following functions, declared in share/builtin.c allows to introduce some non-determinism in the analysis. The results given by the plug-in are valid **for all values proposed by the user**, as opposed to what a test-generation tool would typically do. Such a tool would indeed pick up some values among the billions of possible entries to execute the application.

```
int Frama_C_nondet(int a, int b)

void *Frama_C_nondet_ptr(void *a, void *b)

int Frama_C_interval(int min, int max)

float Frama_C_float_interval(float min, float max);
```

The implementation of these functions might change in future versions of Frama-C, but their types and their behavior will stay the same.

Example of use of the functions introducing non-determinism:

```
#include "share/builtin.h"

int A,B,X;
void main(void)
{
  A = Frama_C_nondet(6, 15);
  B = Frama_C_interval(-3, 10);
  X = A * B;
}
```

With the command

```
toplevel.opt -val -cpp-command "gcc -C -E -I .../ppc/share" \
    ex_nondet.c .../ppc/share/builtin.c
```

The obtained result for `X` is `[-45..150],0%3`.

7.3 Observing Intermediate Results

In addition to using the graphical user interface, it is also possible to obtain information about the value of variables at a particular point of the program in log files. This is done by inserting at the relevant points in the source code calls to the functions described below.

Currently, functions displaying intermediate results all have an immediate effect, *i.e* their effect is to display the particular state that the analyzer is propagating at the moment where it reaches the call. Thus, these functions might expose some undocumented aspects of the behavior of the analyzer. This is in particular the case if they are used together with semantic unrolling (see section 4.2.2). The results displayed might be found counter-intuitive by the user. It is recommended to attach a greater importance to the union of the values displayed during the whole analysis than to the particular order during which the subsets composing these unions are propagated in practice.

7.3.1 Displaying the entire memory state

Displaying the current memory state each time the analyzer reaches a given point of the program is done with a call to the function `Frama_C_dump_each()`.

7.3.2 Displaying the value of an expression

Displaying the values of an expression `expr` each time the analyzer reaches a given point of the program is done with a call to the function `Frama_C_show_each_name(expr)`.

The placeholder “name” can be replaced by an arbitrary identifier s, which appears in the output of the analyzer. It is recommended to use different identifier for each use of these functions, as shown in the following example:

```
void f(int x)
{
  int y;
  y = x;
  Frama_C_show_each_x(x);
  Frama_C_show_each_y(y);
  Frama_C_show_each_delta(y-x);
  ...
}
```

Chapter 8

FAQ

Q.1 Which option should I use to improve the handling of loops in my program, `-ulevel` or `-slevel`?

The options `-ulevel` and `-slevel` have different sets of advantages and drawbacks. The main drawback of `-ulevel` is that it performs a syntactic modification of the analyzed source code, which might hamper its manipulation. On the other hand, syntactic unrolling, by explicitly separating iteration steps, allows in the graphical user interface `frama-c-gui` to easily check values or express properties for a given iteration step of the loop.

`-slevel` option does not allow to observe separately a given iteration step of the loop. In fact, this option might be a little disturbing for the user when the program contains loops for which the analysis can not decide the truth value of the condition for a given step, nested loops, or `if-then-else` statements[1]. The main advantages of this option are that it leaves the source code unchanged and that it applies also to loops that are built using `goto`s instead of `for` and `while`. `-slevel` option requires less memory, and as a consequence is often faster. A current drawback of `-slevel`, which should disappear in future versions of Frama-C, is that it concerns the entire source code under analysis.

Q.2 Alarms that occur after a true alarm in the analyzed code are not detected. Is that normal? May I give some information to the tool so that it detects those alarms?

In both cases the answer is “yes”. Let us consider the following example:

```
int x,y;
void main(void)
{
  int *p=NULL;
  x = *p;
  y = x / 0;
```

[1] `if-then-else` statements are “unrolled” in a similar manner as loops

```
 7  }
```

When this example is analyzed by Frama-C, the value analysis does not emit an alarm on line 6. This is perfectly correct, since no error occurs at run time on line 6. In fact, line 6 is not reached at all, since execution stops at line 5 when attempting to dereference the NULL pointer. It is unreasonable to expect Frama-C to perform a choice over what may happen after dereferencing NULL. It is possible to give some new information to the tool so that analysis can continue after a true alarm. It is called debugging. Once the issue has been corrected in the source code under analysis — more precisely once the user has convinced him/herself that there is no problem at this point in the source code — it becomes possible to trust the alarms that occur after the given point, or the absence thereof (see next question).

Q.3 Can I trust the alarms (or the absence of alarms) that occur after a false alarm in the analyzed code? May I give some information to the tool so that it detects these alarms?

The answers to both these questions are respectively "yes" and "there is nothing special to do". If an alarm might be spurious, the value analysis automatically goes on. If the alarm is really a false alarm, the result given in the rest of the analysis can be considered with the same level of trust than if Frama-C had not displayed the false alarm. One should however keep in mind that this applies only in the case of a false alarm. Deciding whether the first alarm is a true or a false one is the responsibility of the user. This situation can be shown in the following example:

```
 1  int x,y,z,r,i,t[101]={1,2,3};
 2
 3  void main(void)
 4  {
 5    x = Frama_C_interval(-10,10);
 6    i = x * x;
 7    y = t[i];
 8    r = 7 / (y + 1);
 9    z = 3 / y;
10  }
```

```
false_al.c:7: Warning: accessing out of bounds index.
assert ((0 <= i) && (i < 101));

false_al.c:9: Warning: division by zero: assert (y != 0);
```

On line 7, the tool is only capable to detect that i lies in the interval -100..100, which is approximated but correct. The alarm on line 7 is false, because the values that i can take at run-time lie in fact in the interval 0..100. As it proceeds with the analysis, the plug-in detects that line 8 is safe, and that there is an alarm on line 9. These results must be interpreted thus: assuming that the array access on line 7 was legitimate, then line 8 is safe, and there is a threat on line 9. As a consequence, if the user can convince him/herself that the threat on line 7 is false, he or she can trust these results (*i.e.* there is nothing to worry about on line 8, but line 9 needs further investigation).

Q.4 In my annotations, macros are not preprocessed. What should I do?

The annotations being contained inside C comments, they *a priori* aren't affected by the preprocessing. It is possible to instruct Frama-C to launch the preprocessing on annotations with the option `-pp-annot`. However, this option still is experimental at this point. In particular, it requires the preprocessor to be GNU `cpp`, the GCC preprocessor[2]. This restriction might disappear in future versions of Frama-C. Moreover, the preprocessing is then made in two passes (the first pass operating on the code only, and the second operating on the annotations only). For this reason, the options passed to `-cpp-command` in this case should only be commands that can be applied several times without ill side-effects. For instance, the `-include` option to `cpp` is a command-line equivalent of the `#include` directive. If it was passed to `cpp-command` while the preprocessing of annotations is being used, the corresponding header file would be included twice. The Frama-C option `-cpp-extra-args <args>` allows to pass `<args>` at the end of the command for the first pass of the preprocessing.

Example: In order to force `gcc` to include the header `mylib.h` and to preprocess the annotations, the following commandline should be used:

```
frama-c-gui -val -cpp-command 'gcc -C -E -I.'    \
       -cpp-extra-args '-include mylib.h'        \
       -pp-annot file1.c
```

Acknowledgements

Patrick Baudin, Géraud Canet, David Delmas, Benjamin Monate, Anne Pacalet, Julien Signoles provided useful comments in the process of getting this document to where it is today.

[2] More precisely, the preprocessor must understand the `-dD` and the `-P` options that outputs macros definitions along with the preprocessed code and inhibits the generation of `#line` directives respectively.