frama C

Software Analyzers

Value Analysis

Frama-C's value analysis plug-in

Lithium release

Pascal Cuoq with Virgile Prevosto

CEA LIST, Software Reliability Laboratory, Saclay, F-91191

This work has been supported by the 'CAT' ANR project (ANR-05-RNTL-0030x).

Contents

Chapter 1

Introduction

Frama-C is a modular static analysis framework for the C language. This manual documents the value analysis plug-in of Frama-C. The value analysis plug-in automatically computes sets of possible values for the variables of the program. Synthetic information about each analyzed function can be computed automatically from the values provided by the value analysis. This information is also documented here.

The framework, the value analysis plug-in and the other plug-ins documented here are all Open Source software. They can be downloaded from `http://frama-c.cea.fr`.

1.1 First contact

Frama-C comes with two interfaces: batch and interactive. The interactive graphical interface of Frama-C displays a normalized version of the analyzed source code. 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 values this expression can take at run-time.

Here is a simple C example:

```
1  int y, z=1;
2  int f(int x) {
3    y = x + 1;
4    return y;
5  }
6
7  void main(void) {
8    for (y=0; y<2+2; y++)
9      z=f(y);
10 }
```

If either interface of Frama-C is launched with options `-val introduction.c`, 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 or 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 implicitly 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).

The theoretical framework on which the value analysis is founded is called Abstract Interpretation and has been the subject of extensive research during the last thirty years.

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 }
```

When launched for instance with the command `frama-c -val rte.c`, 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”. On the other hand, note that the fact that the value analysis computes correct, over-approximated sets of possible values prevents it from remaining 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, and 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 get the most of these computations.

Chapter 2

What the value analysis provides

2.1 Values

The value analysis plug-in can process queries regarding the value of a variable `x` at a given program point. It answers such a query with an over-approximation of the set of values possibly taken by `x` at the designated point for all possible executions.

2.1.1 Interactive and programmatic interfaces

Both the user (through the GUI) or a custom plug-in (through calls to the functions registered in module `Db.Value`) can request the evaluation, at a specific statement, of an l-value (or an arbitrary expression). The variation domain thus obtained contains all the values that this l-value or expression may have in an actual execution any time the point just before the selected statement is reached.

The function `!Db.Value.access` is one example of the functions provided to custom plug-ins. It takes a program point (of type `Cil_types.kinstr`), the representation of an l-value (of type `Cil_types.lval`) and returns a representation of the possible values for the l-value at the program point.

Another function, `!Db.Value.lval_to_loc`, translates the representation of an l-value into a location (of type `Locations.location`), which is the analyzer's abstract representation for a place in memory. The location returned by this function is free of memory accesses or arithmetic. The provided program point is used for instantiating the values of variables that appear in expressions inside the l-value (indices of array and dereferenced expressions). Thanks to this and similar functions, a custom plug-in may reason entirely in terms of abstract locations, and completely avoid dealing with the problems of pointers and aliasing.

More information about writing a custom plug-in can be found in the Frama-C Plug-in Development Guide.

2.1.2 Variation domains

The variation domain of a variable or expression can take one of the shapes described below.

A set of integers

The variation domain of a variable may have been determined by the analysis to be a set of integers. This usually happens for variables of an integer type, but may happen for other variables if the application contains unions or casts.

Such a set of integers can be represented as:

- an enumeration, `{`v_1; ... v_n`;}`,
- an interval, `[`l..u`]`, that represents all the integers comprised between l and u. If "`--`" appears as the lower bound l (resp. the upper bound u), it means that the lower bound (resp upper bound) is $-\infty$ (resp. $+\infty$),
- an interval with periodicity information, `[`l..u`],`r`%`m, that represents the set of values comprised between l and u whose remainder in the Euclidean division by m is equal to r. For instance, `[2..42],2%10`, represents the set that contains 2, 12, 22, 32, and 42.

A floating-point value or interval

A location in memory (typically a floating-point variable) may also contain 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_l .. f_u`]` for the interval from f_l to f_u inclusive.

A set of addresses

A variation domain (for instance for a pointer variable) may be a set of addresses, denoted by `{{`a_1; ... a_n`;}}`. Each a_i is of the form:

- $\&x + D$, where $\&x$ is the base address corresponding to the 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 denotes absolute addresses (seen as offsets with respect to the base address NULL).

An imprecise mix of addresses

If the application involves, or seems to involve, unusual arithmetic operations over addresses, many of the variation domains provided by the analysis may be imprecise sets of the form `garbled mix of` $\&${x_1; ... x_n;}. This expression denotes an unknown value that was built from applying arithmetic operations to the addresses of variables x_1,...,x_n and to integers.

Absolutely anything

You should not observe it in practice, but sometimes the analyzer is not able to deduce any information at all on the value of a variable, in which case it displays `ANYTHING` for the variation domain of this variable.

2.1.3 Interpreting the variation domains

Most modern compilation platforms for the C language unify integer values and absolute addresses: there is no difference between the encoding of the integer 256 and that of the address `(char*)0x00000100`. Therefore, the value analysis does not distinguish between these two values either.

In floating-point computations, the value analysis considers that obtaining NaN, $+\infty$, or $-\infty$ is an unwanted error. The floating-point intervals provided by the analysis are always intervals of finite floating-point values.

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

Examples of variation 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 at least 2 bytes in size. 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 addresses obtained by shifting `t`, including misaligned and invalid ones.

2.1.4 Origins of approximations

The values resulting from heavy approximations contain information about the origin of these approximations. In this case the value is shown as "`V (origin: ...)`". The text provided after "`origin:`" indicates the location and the cause of some of these approximations. An origin can be one of the following:

Misaligned read

The origin `Misaligned L` indicates a set `L` of lines in 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 the analyzer is by default configured for a 32-bit architecture, and that consequently the read memory access is not an out-of-bound access. If it was, it would cause an alarm to be emitted, which would take the analysis in a different direction.

With the default target platform, the read access remains within the bounds of array `t`, but due to the offset of two bytes, the 32-bit word read is made of the last two bytes from `t[0]` and the first two bytes from `t[1]`.

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 in the analyzed code where memory states with incompatible alignments are fused together. In the example below, the memory states from the then branch and from the else branch contain in the array `t` some 32-bit addresses 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

The origin `Arithmetic L` indicates a 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; }) }}`.

Well value

In some circumstances, the analyzer has to generate a set of possible values for a variable with only its type for information. Some recursive or deeply chained types may force the generated contents for the variable to contain imprecise, absorbing values called well values. See section 5.2 for details.

Computations that are imprecise because of a well value are marked as `origin: Well`.

2.2 Log messages emitted by the value analysis

This section categorizes the messages displayed by the value analysis in the batch version of the analyzer (`frama-c`). When using the graphical interface (`frama-c-gui`), the messages are intercepted and directed to different panels for easier access.

2.2.1 Results

With the batch version of Frama-C, the results of the computations are displayed on the standard output. For some computations, the information computed can not easily be represented in a human-readable way. In this case, it is not displayed at all, and can only be accessed through the GUI or programmatically. In other cases, a compromise had to be reached. For instance, although variation domains for variables are available for any point in the execution of the analyzed application, the batch version only displays, for each function, the values that hold whenever the end of this function is reached.

2.2.2 Proof obligations

The correctness of the results provided by the value analysis is guaranteed only if the user verifies 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`. Frama-C comes with a common specification language for all plug-ins, called ACSL (`http://www.frama-c.cea.fr/acsl.html`). Most of the proof obligations emitted by the value analysis are expressed in ACSL. Each proof obligation message contains the nature and the origin of the obligation.

It is also possible to obtain a version of the analyzed source code annotated with the proofs obligations. Please note that the proof obligations that are not yet expressed in ACSL are missing from the output source code. For those alarms which are expressed as ACSL assertions, do also note that while ACSL's syntax is used, the value analysis' support for ACSL

is still partial in the sense that 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.

Division by zero

When dividing by an expression that the analysis is not able to guarantee to be non-null, 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. The property expressed by an alarm may also not be taken into account when it is not easy to do so.

```
1 int A, B;
2 void main(int x, int y)
3 {
4   A = 100 / (x * y);
5   B = 333 % x;
6 }
```

```
div.c:4: Warning: division by zero: assert (x*y != 0);
...
div.c:5: Warning: division by zero: assert (x != 0);
```

In the above example, there is no way for the analyzer to guarantee that `x*y` is not null, so it emits an alarm at line 4. In theory, it could avoid emitting the alarm `x != 0` at line 5 because this property is a consequence of the property emitted as an alarm at line 4. Redundant alarms happen – even in cases simpler than this one. Do not be surprised by them.

Unspecified logical shift

Another arithmetic alarm is the alarm emitted for logical shift operations on integers where the second argument may be larger than the size of the target type. Such an operation is left unspecified by the ISO/IEC 9899:1999 standard, and indeed, processors are often built in a way that such an operation does not produce the `0` or `-1` result that could have been expected. Here are an example of program with such an issue, and the resulting alarm:

```
1 void main(int c){
2   int x;
3   c = c ? 1 : 8 * sizeof(int);
4   x = 1 << c;
5 }
```

```
shift.c:4: Warning: invalid shift: assert ((c >= 0) && (c < 32));
```

Floating-point alarms

When it appears that a floating-point operation can result in an infinite value or NaN, 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 any other result, may be over-approximated.

An alarm may also be emitted when the application uses as a floating-point number 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.

Uninitialized variables and dangling pointers

An alarm may be emitted if the application seems to read the value of a local variable that has not been initialized, or if it seems to manipulate the address of a local variable outside of the scope of said variable. Both issues appear in the following example. The value analysis should emit alarms for lines 5 (variable `r` may be uninitialized) and 11 (a dangling pointer to local variable `t` is used). As of the current version, the messages are not yet expressed as ACSL properties.

```
1  int *f(int c)
2  {
3    int r, t;
4    if (c) r = 2;
5    t = r + 3;
6    return &t;
7  }
8
9  int main(int c)
10 {
11   return *(f(c));
12 }
```

Invalid memory accesses

Whenever the value analysis is not able to establish that a dereferenced pointer is valid, it emits an alarm that expresses that the pointer needs to be valid at that point.

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

In the above example, the analysis is not able to guarantee that the memory access `t[i]` is valid, so it emits the proof obligation

```
assert ((0 <= i) && (i < 10));
```

Memory model alarms

Proof obligations can also be emitted for pointer comparisons which might break the memory model of the value analysis. These alarms do not necessarily correspond to run-time errors, but they should still be checked, because the results of the value analysis are otherwise incorrect. Consider the example:

```
1  int x,y,*p;
2  main(){
3    p = &x;
4    while (p++ != &y);
5  }
```

The value analysis finds that this program does not terminate. This seems incorrect because an actual execution will terminate on most architectures. However, the value analysis' conclusion is conditioned by an alarm emitted for the pointer comparison. The value analysis only allows pointer comparisons that give deterministic results — that is, the possibility of obtaining an

unspecified result for a pointer comparison is considered as an unwanted error, and is excluded by the emission of an alarm. The memory model is described in more detail in section 4.5.

It would also be possible to authorize these unspecified pointer comparisons and to take both possible results into account, but no user expressed this wish yet.

2.2.3 Experimental status messages

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. An example of such a message is:

```
Warning: float support is experimental
```

2.2.4 Informational messages regarding the loss of precision

Some messages may warn that the analysis is making an operation likely to cause 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 give as much information as possible in order to help em[1] find when the analysis becomes imprecise. These messages are only useful when it is important to analyze the application with precision. The value analysis remains correct even when it is imprecise.

Examples of such messages are:

```
val9.c:10: Warning: assigning non deterministic value for the first time

origin.c:102: Warning: assigning imprecise value to q2.
 The imprecision originates from Misaligned {origin.c:102;}

origin.c:102: Warning: extracting bits of a pointer

origin.c:102:
Warning: reading left-value *((int **)((char *)(v.t) + 3)).
The location is {{ v -> {88; } ;}}.
It contains a garbled mix of {y; }
because of Arithmetic {tests/misc/origin.c:102; }.
```

2.2.5 Progress messages

Some messages are only intended to inform the user of the progress of the analysis. Here are examples of such messages:

```
Parsing

[preprocessing] running gcc -C -E -I.   tests/misc/alias.c

[values] computing for function f <-main
[values] called from tests/misc/alias.c:46

[values] Recording results for f

[values] done for function f
```

Progress messages are informational only. If you find the analysis fast enough, there is no reason to read them at all. If it seems too slow, these messages can help find where time is spent.

[1] Spivak pronouns are used throughout this manual: `http://en.wikipedia.org/wiki/Spivak_pronoun`

Chapter 3

Tutorial

3.1 Introduction

This chapter displays some uses of the value analysis plug-in of Frama-C. This chapter is in the process of being written (that is, even more so than the other chapters). Once it is finished, throughout this tutorial, we will see on a single example how to use the value analysis for the following tasks :

1. to get familiar with foreign code,
2. to produce documentation automatically,
3. to search for bugs,
4. to guarantee the absence of bugs.

It is useful to stick to a single example in this tutorial, and there is a natural progression to the list of results above, but in real life, a single person would generally focus on only one or two of these four tasks for a given codebase. For instance, if you need Frama-C's help to reverse engineer the code as in tasks 1 and 2, you have probably not been provided with the quantity of documentation and specifications that is appropriate for meaningfully carrying out task 4.

Frama-C helps you to achieve tasks 1-3 in less time than you would need to get the same results using the traditional approach of writing tests cases. Task 4, formally guaranteeing the absence of bugs, can in the strictest sense not be achieved at all using tests for two reasons. Firstly, many forms of bugs that occur in a C program (including buffer overflows) may cause the behavior of the program to be non-deterministic. As a consequence, even when a test

suite comes across a bug, the faulty program may appear to work correctly while it is being tested and fail later, after it has been deployed. Secondly, the notion of coverage of a test suite in itself is an invention made necessary because tests aren't usually exhaustive. Assume a function's only inputs are two 32-bit integers, each allowed to take the full range of their possible values. Further assume that this function only takes a billionth of a second to run (a couple of cycles on a 2GHz processor). A pencil and the back of an envelope show that this function would take 600 years to test exhaustively. The best that can be done is to invent testing coverage criteria in order to be able to decide that a test suite is "good enough", but there is an implicit assumption in this reasoning. The assumption is that a test suite that satisfies the coverage criteria finds all the bugs that need to be found, and this assumption is generally justified empirically only.

3.2 Target code

A single piece of code, the reference implementation for the Skein hash function, is used as an example throughout this document. As of this writing, this implementation is available at `http://www.schneier.com/code/skein.zip`. The Skein function is Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker's entry in the NIST cryptographic hash algorithm competition for SHA-3.

Cryptographic hash functions are one of the basic building blocks from which cryptographic protocols are designed. Many cryptographic protocols are designed with the assumption that an implementation for a function h from strings to fixed-width integers is available with the following two properties:

- it is difficult to find two distinct, arbitrary strings s_1 and s_2 such that $h(s_1) = h(s_2)$,
- for a given integer i, it is difficult to build a string s such that $h(s) = i$.

A function with the above two properties is called a cryptographic hash function. It is of the utmost importance that the function actually chosen in the implementation of a cryptographic application satisfies the above properties! Any weakness in the hash function can be exploited to corrupt the application.

Proofs of security for cryptographic hash functions are complicated mathematical affairs. These proofs are made using a mathematical definition of the function. Using static analysis for verifying that a piece of code corresponds to the mathematical model which was the object of the security proofs is an interesting topic but is outside the scope of this tutorial. In this document, we will not be using the value analysis for verifying cryptographic properties.

We will, however, use the value analysis for familiarizing ourselves with the reference implementation of Skein, and we will see that Frama-C's value analysis can be a more useful tool than, say, a C compiler, to this effect. We will also use the value analysis to look for bugs in the implementation of Skein, and finally prove that the functions that implement Skein may never cause a run-time error for a general use pattern. Because of the numerous pitfalls of the C programming language, any amount of work at the mathematical level can not exclude the possibility of problems such as buffer overflows in the implementation. It is a good thing to be able to rule these out with Frama-C's value analysis.

Frama-C's value analysis is most useful for embedded or embedded-like code. Although the Skein library does not exactly fall in this category, it does not demand dynamic allocation and uses few functions from external libraries, so it is well suited to this analysis.

3.3 Using the value analysis for getting familiar with Skein

3.3.1 Writing a test

Once the archive `skein_NIST_CD_010509.zip` has been extracted, listing the files in `NIST/CD/Reference_Implementation` shows:

```
-rw-r--r--  1 pascal  501   4984 Oct 14 00:52 SHA3api_ref.c
-rw-r--r--  1 pascal  501   2001 Oct 14 00:54 SHA3api_ref.h
-rw-r--r--  1 pascal  501   6141 Sep 30 00:28 brg_endian.h
-rw-r--r--  1 pascal  501   6921 May 17  2008 brg_types.h
-rw-r--r--  1 pascal  501  34990 Jan  6 01:39 skein.c
-rw-r--r--  1 pascal  501  16290 Nov  9 05:48 skein.h
-rw-r--r--  1 pascal  501  18548 Oct  7 20:02 skein_block.c
-rw-r--r--  1 pascal  501   7807 Oct 10 02:47 skein_debug.c
-rw-r--r--  1 pascal  501   2646 Oct  9 23:44 skein_debug.h
-rw-r--r--  1 pascal  501   1688 Jul  3  2008 skein_port.h
```

The most natural place to go next is the file `skein.h`, since its name hints that this is the header that declare the functions that should be called from outside the library. Scanning the file quickly, we may notice declarations such as

```
typedef struct /*  256-bit Skein hash context structure */
    {
      [...]
    } Skein_256_Ctxt_t;

/*   Skein APIs for (incremental) "straight hashing" */
int  Skein_256_Init  (Skein_256_Ctxt_t *ctx, size_t hashBitLen);
[...]
int  Skein_256_Update(Skein_256_Ctxt_t *ctx, const u08b_t *msg,
                                                 size_t msgByteCnt);
[...]
int  Skein_256_Final (Skein_256_Ctxt_t *ctx, u08b_t * hashVal);
```

The impression we get at the first glance is that the hash of a 80-char message containing the string "People of Earth, your attention, please" can be computed as simply as declaring a variable of type `Skein_256_Ctxt_t`, letting `Skein_256_Init` initialize it, passing `Skein_256_Update` a representation of the string, and calling `Skein_256_Final` with the address of a buffer where to write the hash value. Let us write a C program that does just that:

```
#include "skein.h"

#define HASHLEN (8)

u08b_t msg[80]="People of Earth, your attention, please";

int main(void)
{
  u08b_t hash[HASHLEN];
  int i;
  Skein_256_Ctxt_t skein_context;
  Skein_256_Init( &skein_context, HASHLEN);
  Skein_256_Update( &skein_context, msg, 80);
  Skein_256_Final( &skein_context, hash);
  for (i=0; i<HASHLEN; i++)
    printf("%d\n", hash[i]);
  return 0;
}
```

Note that in order to make the test useful, we have to print the obtained hash value. Because the result we are interested in is a 8-byte number, represented as a char array of containing arbitrary characters (some of them non-printable), we can not use string-printing functions. The compilation and execution go smoothly and we obtain the hash value that we were looking for:

```
gcc *.c
./a.out
```

```
215
215
189
207
196
124
124
13
```

3.3.2 First try at a value analysis

Let us now see how Frama-C's value analysis works on the same example. The value analysis can be launched with a command that (for most Unix shells) looks as follows. The analysis should not take more than a few seconds:

```
frama-c -val *.c >log 2>&1
```

The "`>log 2>&1`" part of the command is meant to send all the messages emitted by Frama-C into a single log file named "`log`". The value analysis is verbose for a number of reasons that will become clearer later. The best way to make sense of the information produced by the analysis is to send it to a log file. There is also a Graphical User Interface for creating analysis projects and visualizing the results. The GUI is still evolving quickly at this stage and it would not make much sense to describe its manipulation in detail in this document. You are however encouraged to try it if is is available on your platform. One way to create an analysis project in the GUI is to pass the command `frama-c-gui` the same options that would be passed to `frama-c`. In this first example, the command `frama-c-gui -val *.c` launches the same analysis and then opens the GUI for inspection of the results.

The initial syntactic analysis and symbolic link phases done by Frama-C may find issues in the analyzed program, such as inconsistent types for the same symbol in different files. Because of the way separate compilation is implemented, the issues are not detected by standard C compilers. It is a good idea to check for these issues in the first lines of the log.

3.3.3 Missing functions

Especially since we are trying out the library for the first time, another thing to look for in the log file is the list of functions for which the source code is missing:

```
grep "No code for" log
```

```
No code for function memset, default assigns generated
No code for function memcpy, default assigns generated
No code for function printf, default assigns generated
```

These are the three functions for which no definition were found. Since these function are expected to be in system libraries, it is normal not to find any definition for them. Still, the value analysis needs source code in order to be able to do its work. Without it, it can only guess what each function does from the function's prototype, which is both inaccurate and potentially wrong.

Note that it is also possible to obtain a list of missing function definitions by using the command:

```
frama-c -metrics *.c
```

This command computes, among other pieces of information, a list of undefined functions using a syntactic analysis. It is not exactly equivalent to grepping the log of the value analysis because it lists all the functions that are missing, while the log of the value analysis only cites the functions that would have been necessary to an execution. When analyzing a small part of a large library, the latter list may be much shorter than the former. In this case, relying on the information displayed by `-metrics` means spending time hunting for functions that are not actually necessary.

The best way to deal with the missing functions `memset` and `memcpy` is to provide source code for them. Sometimes it is a good idea to provide the exact source code that will actually be used in an actual running environment, so as to detect subtle bugs in the interactions between the program and the system libraries. For string manipulation functions, which often feature difficult-to-analyze architecture-dependent optimizations, it is better to provide a simpler but equivalent implementation. In this particular case, let us provide these two functions in a file that we will place with the others and name "`lib.c`".

```
1  #include <string.h>
2
3  void* memcpy(void* region1, const void* region2, size_t n)
4  {
5    char *dest =  (char*)region1;
6    const char* first = (const char*)region2;
7    const char* last = ((const char*)region2) + n;
8    char* result = (char*)region1;
9    while (first != last)
10     *dest++ = *first++;
11   return result;
12 }
13
14 void* memset (void* dest, int val, size_t len)
15 {
16   unsigned char *ptr = (unsigned char*)dest;
17   while (len-- > 0)
18     *ptr++ = val;
19   return dest;
20 }
```

In some circumstances, the results of the value analysis may remain useful even when Frama-C is left to guess the effects of missing functions, but in the case of functions that take pointers as arguments, the possibilities are too numerous for the analyzer to be able to guess right. In this example, the function `printf` is safe to leave without an implementation, because unlike the other two, it does not have effects on the variables of the program.

3.3.4 First meaningful analysis

If we run the analysis again now that the missing functions have been provided, the new log that we obtain no longer mentions any missing function other than `printf`[1].

The log next contains a description of the initial values of variables. Apart from the variables that we defined ourselves, there is a rather strange one, named `ONE` and indeed containing `1`. A quick grep reveals that this variable is in fact declared `static` as it should be. Frama-C displays the value of static variables (something that is annoying when using a C compiler for testing). Frama-C may virtually rename a static variable in order to distinguish it from another variable with the same name. In the GUI, where it is also possible to inspect values of variables, the original source code is displayed alongside the transformed one.

[1] It is a good idea to check again for missing functions because new execution paths could have been activated by the functions that were previously missing

A quick inspection shows that the variable `ONE` is used in the Skein library to detect endianness. Frama-C assumes a little-endian architecture by default, so the value analysis is only analyzing the little-endian version of the library (Frama-C also assumes an IA-32 architecture, so we are only analyzing the library as compiled and run on this architecture). The big-endian version of the library could be analyzed by reproducing the same steps we are taking here for a big-endian configuration of Frama-C.

Next in the log are a lot of entries that simply indicate the value analysis' progression. These messages are useful to understand where time is spent when the analysis is too long. We can ignore them now. The following lines are also, in this context, progression messages:

```
lib.c:17: Warning: entering loop for the first time
lib.c:17: Warning: assigning non deterministic value for the first time
```

The next log entry which is not a progression message is a warning concerning variable `tmp`

```
lib.c:18: Warning: out of bounds write. assert \valid(tmp);
```

There is no variable `tmp` in our file `lib.c`. The variable `tmp` that should be a valid address at line 18 in file `lib.c` is a variable that has been introduced by Frama-C when it was normalizing the source code. The relation this variable has to the original source code is easiest to see in the GUI, or alternately, the command `frama-c *.c -print` can be used to see how the original source code had been transformed by Frama-C (note, however, that the location "line 18" refers to the original source file). Here, it is the statement `*ptr++ = val;` in the function `memset` that has been transformed by Frama-C into the sequence below, so that it would not be ambiguous which value of `ptr` has to be a valid pointer.

```
  tmp = ptr;
  ptr ++;
  *tmp = (unsigned char )val;
```

The command `frama-c *.c -val -print` launches the value analysis and then prints the transformed source code, which is then annotated with comments that indicate the alarms raised by the value analysis. In our case, the function `memset` is transformed in

```
void *memset(void *dest , int val , size_t len )
{ unsigned char *ptr ;
  unsigned char *tmp ;
  size_t tmp_0 ;

  {ptr = (unsigned char *)dest;
  while (1) {{{tmp_0 = len;
              len -= (size_t )1;}

              ;}
    if (! (tmp_0 > (size_t )0)) {break;}

    {{tmp = ptr;
     ptr ++;}

     /*@ assert \valid(tmp);
          // synthesized
     */
     *tmp = (unsigned char )val;}}

  return (dest);}

}
```

As we will find out later in this tutorial, this alarm is a false alarm, an indication of a potential problem in a place where there is in fact none. On the other hand, the value analysis never remains silent when a risk of run-time error is present, unlike many other static analysis tools.

It may seem that the access to `*tmp` is the only dangerous operation in the function `memset`, and therefore that the analyzer is not doing a very good job of pointing only the operations that are problematic. While it is true that the results obtained on this example are not very precise, the comparison `>` is also considered as a dangerous operation by the analyzer. It may be unspecified when applied to an invalid pointer. Consider a program that does the following:

```
1 int main()
2 {
3   int a;
4   return ((unsigned int)(&a - 1432)) > 0;
5 }
```

The programmer might have written the `>0` test in this example as a convoluted way to test for `NULL`. Ey might expect this program always to return 1, since `(&a - 1432)` is not `NULL`. But this program may in fact return 0 if, out of bad luck, the variable `a` is placed precisely at address 1432 by the compiler. This kind of bug would be very difficult to find and/or reproduce by testing.

The analyzer does not emit any alarm for the comparison `tmp_0 > (size_t )0` in the normalized source code for `memset`, and this means that it guarantees[2] that this particular comparison is never non-deterministic because of an issue such as the one just described. This conclusion can only be reached by looking at the arguments actually passed to the function `memset`. Therefore, this conclusion is only valid for all the possible executions coming from the `main` function that we provided, and not for all the possible calls to function `memset` that a programmer could write.

One should not spend too much time at this stage trying to determine if the dozen or so alarms emitted by the analyzer are true alarms that indicate an actual problem or false alarms that don't. The most important information is that the analysis did not take an unreasonable time. The second most important information is that the analysis seems to have explored all the paths we intended for it to explore, as seen in the list of functions for which values are printed or in the log entries such as:

```
[values] computing for function RotL_64 <-Skein_256_Process_Block
                                                <-Skein_256_Final <-main
[values] called from skein_block.c:100
```

The GUI can be used to inspect the values of the variables at different points of the execution of the program and to get a feeling of how it works. Right-clicking on the function name in a function call brings a contextual menu that allows to go to the function's definition and to inspect it in turn. In order to return to the caller, right-clicking on the name of the current function at the top of the normalized code buffer brings a contextual menu with a list of callers.

3.4 Searching for bugs

3.4.1 Increasing the precision with option `-slevel`

Because we compiled and executed the same program that we are now analyzing, we are confident that most of the alarms displayed by the value analysis are false alarms that do not correspond to actual problems. In fact, because the program is deterministic, only one

[2] The fine print is in Frama-C's value analysis reference manual

of the alarms can be a true alarm in the strictest sense. The analysis stops when an error is encountered (what would it even mean to continue the analysis after, for instance, `NULL` has been dereferenced?). It is only because the value analysis is uncertain about the errors encountered here that it continues and finds more possible errors.

Before we spend any of our time looking at each of these alarms, trying to determine whether it is true or false, it is a good idea to make the analyzer spend more of its time trying to determine whether each alarm is true or false. There are different settings that influence the compromise between precision and resource consumption. If you chose to remember only one of these settings, it would have to be the option `-slevel`. This option has two visible effects: it makes the analyzer unroll loops, and it makes it propagate separately the states that come from the `then` and `else` branches of a conditional statement. This makes the analysis more precise (at the cost of being slower) for almost every program that can be analyzed.

The value analysis has different ways to provide information on the loss of precision that causes the false alarms. This is another reason why it is so verbose during an analysis: it tries to provide enough information for a motivated user to be able to understand what is happening during the analysis, where the imprecise values that cause the false alarms came from, and what instructions in the program made them imprecise.

But instead of spending precious human time making use of the provided information, the brute-force approach of blindly applying the `-slevel` option must be tried first. Indeed, with some rare exceptions that are clearly marked as such in the reference manual, the options of the value analysis never cause it to become incorrect. When they are used wrongly, the options can make the analysis slower, provide hints that are not useful, or even possibly counter-productive (making the analysis less precise), but they can not prevent the analyzer to report a problem where there is one.

The `-slevel` option is useful for unrolling loops. We do not know without further inspection how many iterations the loops inside the program need, but the input message is 80 characters long, and we can assume that it is read inside a loop, so using the option `-slevel 100` should have a good chance of unrolling at least that loop. Furthermore, it does not make the analysis slower to use a number greater than the value actually necessary to unroll every loop and conditional in the program. We are limiting ourselves to 100 in this first try because we do not know how much time the analysis will take with this number of unrolled branches. If, with the parameter `-slevel 100`, the precision of the analysis still isn't satisfactory and the time spent by the analyzer remains reasonable, we can always try a higher value. This should be progressive, because a value higher than a few hundreds, when there really are that many branches to unroll in the program, can make the analysis very slow.

```
frama-c -slevel 100 -val *.c  >L 2>&1
```

The analysis goes rather far without finding any alarm, but when it is almost done (after the analysis of function `Skein_256_Final`), it produces:

```
[values] Recording results for Skein_256_Final
[values] done for function Skein_256_Final
[values] computing for function printf <-main
[values] called from main_1.c:16
No code for function printf, default assigns generated
[values] done for function printf
main_1.c:16: Warning: (TODO: emit a proper alarm) accessing
                                uninitialized left-value: hash[i]
main_1.c:16: Warning: completely unspecified value in {{
hash -> {8; } ;}} (size:<8>). This path is assumed to be dead.
main_1.c:16: Warning: Evaluation of argument led to bottom in function call
[values] Recording results for main
main_1.c:16: Warning: non termination detected in function main
[values] done for function main
```

These messages mean that the l-value `hash[i]` as found in line 16 of the file `main_1.c` that we provided may be uninitialized. Actually, the analyzer finds that for one execution branch, it is certain to be uninitialized. This is when reading the memory location `{{ hash -> {8; } ;}} (size:<8>)`, where the offsets and sizes are expressed in bits – in other words, when reading the second char of array `hash`. The analysis of this branch is therefore stopped, because from the point of view of the analyzer reading an uninitialized value should be a fatal error. And there are no other execution branches that reach the end of this loop, which is why the function `main` is found not to terminate.

The next warning, "`Evaluation of argument led to bottom in function call`", is an indication that it is the evaluation of the arguments of `printf` before the call, and not the call itself, that failed to terminate and produce a result. This distinction is made because in a situation as we have here where a function call does not terminate, it is natural to start searching for an explanation for the non-termination inside the code of the called function, and this would be a wrong track here.

3.4.2 Making sense of the alarms

There is only one alarm in the analysis we now have, and the program is also found not to terminate, which means that every execution either encounters the problem from line 16 in `main_1.c`, or an infinite loop somewhere else in the program. We should, therefore, pay particular attention to this alarm. The GUI can be used to inspect the value of `hash` before the loop in `main`, and this way we may notice that `Skein_256_Final` only wrote in the first character of `hash`. The rest of the array remained uninitialized, and since `hash` is a local variable, could not be expected to contain zeros.

Looking again at the header file `skein.h`, we may notice that the formal parameter for the length expected by the function `Skein_256_Init` is named `hashBitLen` and should presumably be expressed in bits. We were inadvertently asking for a 1-char hash of the message since the beginning, and the test that we ran as our first step failed to notice it. Our first imprecise analysis did find it – the same alarm that pointed out the problem is present among the other alarms produced by the first analysis. Note that because of the imprecisions, the analysis was not able to conclude that the uninitialized access at `hash[2]` was certain, making it much less noticeable among the others.

The bug can be fixed by passing 8*HASHLEN instead of HASHLEN as the second argument of `Skein_256_Init`. With this fix in place, the analysis with `-slevel 100` produces no alarms and gives the following result:

```
Values for function main:
  i IN {8; }
  hash[0] IN {224; }
       [1] IN {56; }
       [2] IN {146; }
       [3] IN {251; }
       [4] IN {183; }
       [5] IN {62; }
       [6] IN {26; }
       [7] IN {48; }
```

Meanwhile, recompiling and executing the test produces the result:

```
224
56
146
251
183
```

```
62
26
48
```

3.5 Guaranteeing the absence of bugs

3.5.1 Generalizing the analysis

The analysis we have done so far is very satisfying because it finds problems that are not detected by a C compiler or by testing. It only proves the absence of run-time errors[3] when the particular message that we chose is being hashed, though. It would be much more useful to have the assurance that there are no run-time errors for any input message, especially since the library might be under consideration for embedding in a device where anyone (possibly a malicious user) will be able to choose the message to hash.

A first generalization of the previous analysis is to include in the subject matter the hashing of all possible 80-character messages. We can do this by separating the analyzed program in two distinct phases, the first one being the construction of a generalized analysis context and the second one being made of the sequence of function calls that we wish to study:

```
1  #include "skein.h"
2
3  #define HASHLEN (8)
4
5  u08b_t msg[80];
6
7  void main(void)
8  {
9    int i;
10   u08b_t hash[HASHLEN];
11   Skein_256_Ctxt_t skein_context;
12
13   for (i=0; i<80; i++) msg[i]=Frama_C_interval(0, 255);
14
15   Skein_256_Init( &skein_context, HASHLEN * 8);
16   Skein_256_Update( &skein_context, msg, 80);
17   Skein_256_Final( &skein_context, hash);
18 }
```

From this point onward the program is no longer executable because of the call to built-in primitives such as `Frama_C_interval`. We therefore dispense with the final calls to `printf`, since the value analysis offers simpler ways to observe intermediate and final results. The analysis is launched with the same command again:

```
frama-c -slevel 100 -val *.c  >L 2>&1
```

Again, there is no alarm emitted during the analysis. This time, the absence of alarms is starting to be really interesting: it means that it is formally excluded that the functions `Skein_256_Init`, `Skein_256_Update`, and `Skein_256_Final` produce a run-time error when they are used, in this order, to initialize a local variable of type `Skein_256_Ctxt_t` (with the argument 64 for the size), parse an arbitrary message and produce a hash in a local `u08b_t` array of size 8,

[3] nor conditions that *should* be run-time errors – like the uninitialized access that we encountered

Chapter 4

Limitations and specificities

This chapter describes how the difficult constructs in the C language are handled in the value analysis. The constructs listed here are difficult for all static analyzers in general. 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. The value analysis works best on embedded code or embedded-like code without dynamic allocation nor multithreading, although it is usable (with modelization work from the user) on applications that feature either.

Variadic function calls are not handled with much precision. On the other hand, many applications do not call any variadic function other than `printf`, and the observation primitives of Frama-C (section 8.3) and the capabilities built in the GUI more than make up for the absence of `printf`.

4.1 Loops

The analysis of a source program always takes a finite time. The fact that the source code contains loops, and that some of these loops do not terminate, can never induce the analyzer itself to loop forever [1]. In order to guarantee this property, the analyzer may need to introduce approximations when analyzing a loop.

[1] There are two exceptions to this rule. The analyzer can loop if it is launched on a program with non-natural loops (i.e. gotos from outside to inside a loop), or if the most precise modelizations of `malloc` are used (details in section 8.1.1).

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

```
1   n=100;
2   i=0;
3   y=0;
4   do {
5     i++;
6     if (c(i))
7       y = 2*i;
8     } while (i<n);
```

The value analysis plug-in could provide the best possible sets of values if the user explicitly instructed it 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. This is an over-approximation of the most precise correct result, which is "an even number between 0 and 200".

Section 5.3 introduces the different ways in which the user can influence the plug-in's strategy with respect to the analysis of loops.

4.2 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 4.1.

```
1  int n, y;
2  void f(int x) { y = x; }
3
4  void main_1(void) {
5    int i;
6
7    n=100;
8    i=0;
9    y=0;
10   do {
11    i++;
12    if (c(i))
13      f(2*i);
14    } while (i<n);
15
16 }
```

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

4.3 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 the analysis to critical subparts of the application, by using other entry points than the actual one (the `main` function). Besides, the source code of some of the functions used by 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 5.4.

4.3.1 Entry point of a complete application

When the source code for the analyzed application is entirely available, the only additional information expected by the plug-in is the name of the function 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 4.2 is not the function `main_1` but the following `main_main` function:

```
17 void main_main(void) {
18   f(15);
19   main_1();
20 }
```

If the user specifies the wrong entry point `main_1`, the plug-in will provide the same answer for variable `y` at the end of function `f` as in sections 4.1 and 4.2: 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 analyzed application can be specified on the command line using the option `-main` (section 5.2.1). In the GUI, it is also possible to specify the name of the entry point at the time of launching the value analysis.

4.3.2 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 5.2.3, 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).

4.3.3 Library functions

Another category of functions whose code may be missing is composed of the operating system primitives and functions from external libraries. These functions are called "library functions". The behavior of each library function can be specified through annotations (see chapter 7). 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 7.2). An alternative way to specify a library function is to write C code that models its behavior, so that its code is no longer missing from the point of view of the analyzer.

4.3.4 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. There is preliminary support for using the value analysis in this context in the form of support for volatile variables. Unlike normal variables, the analyzer does not assume that the value read from a volatile variable is identical to the last value written there.

4.3.5 Choosing between complete and partial application mode

This section uses 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:

```
1  int ok1;
2
3  void init1(void) {
4    ...
5    if (error condition)
6      error_handling1();
7    else
8      ok1 = 1;
9  }
10
11 void init2(void) {
12   if (ok1) {
13     ...
14     }
15 }
16
17 void main(void) {
18   init1();
19   init2();
20   ...
21 }
```

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` consists of determining that the value of the `if` condition is always false, and to ignore all the code that follows. Any possible run-time error 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, one way to analyze the function `init2` is to use the option `-lib-entry` described in section 5.2.3.

It is also possible to use annotations to describe the state in which the analysis should be started as a precondition for the entry point function. The syntax and usage of preconditions is described in section 7.1. The user should pay attention to the intrinsic limitations in the way the value analysis interprets these properties (section 7.1.2). A simpler alternative for specifying an initial state is to build it using the non-deterministic primitives described in section 8.2.1.

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 initial state). The division of the application into parts may follow the phases in the application's behavior (initialization followed by the permanent phase) or break it down into elementary sub-pieces, the same way unit tests do.

4.4 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.

4.4.1 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 value analysis plug-in assumes a reasonable compiler and target architecture. This design choice makes it 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 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 composes 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 eir 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.

4.4.2 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 IA-32 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 auto-detection program can be provided in order to check the hypotheses mentioned in section 4.4.1, 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 parameterized 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 auto-detection 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.

4.5 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.

4.5.1 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

```
1 int x;
2 int t[12][12][12];
3 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`.

4.5.2 Address

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

4.5.3 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`".

```
int x, y;
int *p = &y;

void main(int c) {
  if (c)
    x = 2;
  else {
    while (p != &x) p++;
    *p = 3;
    }
}
```

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;
}
```

4.6 What the value analysis does not provide

Values that are valid even if something bad happens

Sets of possible values are provided by the analysis under the assumption that the alarms emitted during the analysis have been verified by the user (if necessary, using other techniques). If during an actual execution of the application, one of the assertions emitted by the value analysis is violated, values other than those predicted by the value analysis may happen. See also questions 2 and 3 in chapter 9.

Termination or reachability properties

Although the value analysis sometimes detects that a function does not terminate, it can not be used to prove that a function terminates. Generally speaking, the fact that the value

analysis provides non-empty sets of values for a specific statement in the application does not imply that this statement is reachable in a real execution. Currently, the value analysis is not designed to prove the termination of loops or similar liveness properties. For instance, the following program does not terminate:

```
1 int x, y = 50;
2 void main()
3 {
4   while(y<100)
5     y = y + (100 - y)/2;
6   x = y;
7 }
```

If this program is analyzed with the default options of the value analysis, the analysis finds that every time the end of `main` is reached, the value of `x` is in `[100..127]`. This does not mean that the function always terminates or even that it may sometimes terminate (it does neither). When the value analysis proposes an interval for `x` at a point P, it should always be interpreted as meaning that if P is reached, then at that moment `x` is in the proposed interval, and not as implying that P is reached.

Propagation of the displayed states

The values available through the graphical and programmatic interfaces do not come from a single propagated state but from the union of several states that may have been propagated separately during the analysis. As a consequence, it should not be assumed that the "state" displayed at a particular program point has been propagated. In the following example, the value analysis did not emit any alarm for the division at line 8. This means that the divisor was found never to be null during an actual execution starting from the entry point. The values displayed at the level of the comment should not be assumed to imply that `(x - y)` is never null for arbitrary values `x`$\in \{0; 1;\}$ and `y`$\in \{0; 1;\}$.

```
1 int x, y, z;
2 main(int c){
3   ...
4   ...
5 /* At this point the value analysis guarantees:
6    x IN {0; 1}
7    y IN {0; 1}; */
8   z = 10 / (x - y);
9 }
```

If the analysis is done with the option `-slevel` described in section 5.3.2, the lines leading up to this situation may for instance have been:

```
3   x = c ? 0 : 1;
4   y = x ? 0 : 1;
```

Identically, the final states displayed by the batch version of Frama-C for each function are an union of all the states that reached the end of the function when it was called during the analysis. It should not be assumed that the state displayed for the end of a function `f` is the state that was propagated back to a particular call point. The only guarantee is that the state that was propagated back to the call point is included in the one displayed for the end of the function.

The only way to be certain that a specified state has been propagated by the value analysis, and therefore guarantee the absence of run-time errors under the assumptions encoded in that state, is to build the intended state, for instance with non-deterministic primitives (section 8.2.1). However, the intermediate results displayed in the GUI can and should be used for cross-checking that the state actually built looks like the one intended.

Chapter 5

Parameterizing the analysis

5.1 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>
```

Most parameters can also be set after the tool is launched, in the graphical interface.

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.

For advanced users (or plug-in authors), there exists a "batch" version of Frama-C. The executable is then named `frama-c` (or `frama-c.exe`). All the options of the value analysis work identically for the GUI version and the batch version of Frama-C.

5.1.1 Analyzed files and preprocessing

The analyzed files should be syntactically correct C. The files that do not use the `.i` extension are automatically pre-processed. The preprocessing command used by default is:

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

It is possible that files without a `.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 pre-process 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 place-holders 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
```

5.1.2 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.

5.1.3 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` loads the state saved in file `s` back into memory for visualization or further computations.

Example :

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

5.2 Describing the analysis context

5.2.1 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.

5.2.2 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 5.2.1) is the actual starting point of this analyzed application. In the initial state, each formal argument of the entry point contains a non-deterministic value that corresponds to its type. Non-aliasing locations are generated for 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.

Example: for an application written for the POSIX interface, the prototype of `main` is:

```
int main(int argc, char **argv)
```

The types of arguments `argc` and `argv` translate into the following initial values:

```
argc IN [--..--]
argv IN {{ &NULL ; &star_argv ;}}
star_argv[0] IN {{ &NULL ; &star_star_argv_0nth ;}}
         [1] IN {{ &NULL ; &star_star_argv_1nth ;}}
star_star_argv_0nth[0..1] IN [--..--]
star_star_argv_1nth[0..1] IN [--..--]
```

This is generally not what is wanted, but then again, embedded applications are generally not written against the POSIX interface. If your application expects command-line arguments, you should probably write an alternative entry point that creates the appropriate context before calling the actual entry point. That is, assume you want to analyze an application whose source code looks like this:

```
int main(int argc, char **argv)
{
  if (argc != 2) usage();
  ...
}

void usage(void)
{
  printf("this application expects an argument "
     "between '0' and '9'\n");
  exit(1);
}
```

You should make use of the non-deterministic primitives described in section 8.2.1 to write an alternative entry point for the analysis like this:

```
int analysis_main(void)
{
  char *argv[3];
  char arg[2];
  arg[0]=Frama_C_interval('0', '9');
  arg[1]=0;
  argv[0]="Analyzed application";
  argv[1]=arg;
  argv[2]=NULL;
  return main(2, argv);
}
```

Note that for this particular example, the initial state that was automatically generated includes the desired one. This may however not always be the case. Even when it is the case, it is desirable write an analysis entry point that positions the values of `argc` and `argv` to improve the relevance of the alarms emitted by the value analysis.

Although the above method is recommended for a complete application, it remains possible to let the analysis automatically produce values for the arguments of the entry point. In this case, the options described in section 5.2.4 below can be used to tweak the generation of these values to some extent.

5.2.3 Analysis of an incomplete application

The option `-lib-entry` specifies that the analyzer should not use the initial values for globals (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-deterministic value of their respective type. Variables with a pointer type 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.

5.2.4 Tweaking the automatic generation of initial values

This sections describes the options that influence the automatic generation of initial values of variables. The concerned variables are the arguments of the entry point and, in `-lib-entry` mode, the non-const global variables.

Width of the generated tree

For a variable of a pointer type, there is no way for the analyzer to guess whether the pointer should be assumed to be pointing to a single element or to be pointing at the beginning of an array — or indeed, in the middle of an array, which would mean that it is legal to take negative offsets of this pointer.

By default, a pointer type is assumed to point at the beginning of an array of two elements. This number can be changed with the `-context-width` option.

Example: if the prototype for the entry point is `void main(int *t)`, t is assumed by the analyzer to point to an array `int star_t[2]`.

Depth of the generated tree

For variables of a type pointer to pointers, the analyzer limits the depth at which initial chained structures are generated. This is necessary for recursive types such as the following.

```
struct S { int v; struct S *next; };
```

This limit may also be observed for non-recursive types if they are deep enough.

The limit can be changed with the option `-context-depth`. The default value is 2. This number is the depth at which additional variables named `star_...` are allocated, so two is plenty for most programs.

For instance, here is the initial state displayed by the value analysis in `-lib-entry` mode if a global variable `s` has type `struct S` defined above:

```
s.v IN [--..--]
 .next IN {{ &NULL ; &star_s__next ;}}
star_s__next[0].v IN [--..--]
             [0].next IN {{ &NULL ; &star_star_s__next_0nth__next ;}}
             [1].v IN [--..--]
             [1].next IN {{ &NULL ; &star_star_s__next_1nth__next ;}}
star_star_s__next_0nth__next[0].v IN [--..--]
                             [0].next IN
...
```

In this case, if variable `s` is the only one which is automatically allocated, it makes sense to set the option `-context-width` to one. The value of the option `-context-depth` represents the length of the linked list which is modelized with precision. After this depth, an imprecise value (called a well) captures all the possible continuations in a compact but imprecise form.

Below are the initial contents for a variable `s` of type `struct S` with options `-context-width 1 -context-depth 1`:

```
s.v IN [--.--]
 .next IN {{ &NULL ; &star_s__next ;}}
star_s__next[0].v IN [--.--]
            [0].next IN {{ &NULL ; &star_star_s__next_0nth__next ;}}
star_star_s__next_0nth__next
   [0].v IN [--.--]
   [0].next IN {{ garbled mix of &{star_...__next_WELL;
                                          } (origin: Well) }}
star_...__next_WELL[bits 0 to 549755813887] IN
               {{ garbled mix of &{star_...__next_WELL;
                                          } (origin: Well) }}
```

The NULL possibility

In all the examples above, `NULL` was one of the possible values for all pointers, and the linked list that was modelized ended with a well that imprecisely captured all continuation for the list. The option `-context-valid-pointers` causes `NULL` to be omitted from the possible values at depths that are less than the context width, and causes the list to end with a `NULL` value instead of a well. When analyzed with options `-context-width 1 -context-depth 1 -context-valid-pointers`, a variable `s` of type `struct S` receives the following initial contents, modeling a chained list of length exactly 3:

```
s.v IN [--.--]
 .next IN {{ &star_s__next ;}}
star_s__next[0].v IN [--.--]
            [0].next IN {{ &star_star_s__next_0nth__next ;}}
star_star_s__next_0nth__next[0].v IN [--.--]
                            [0].next IN {0; }
```

5.3 Treatment of loops

5.3.1 Controlling approximations

The default treatment of loops by the analyzer may produce results that are too approximate. The 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 5.3.2.

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` $v_1, \ldots, v_n$, $e_1, \ldots, e_m$ `;` may be placed before a loop, so as to make the analyzer use preferably the values $e_1, \ldots, e_m$ when widening the sets of values attached to variables $v_1, \ldots, v_n$.

If this annotation does not contain any variable, then the values $e_1, \ldots, e_m$ are used as bounds for all the variables that are modified inside the loop.

Example:

```
1  int i,j;
2
3  void main(void)
4  {
5    int n = 13;
6    /*@ loop pragma WIDEN_HINTS i, 12, 13; */
7    for (i=0; i<n; i++)
8      {
9        j = 4 * i + 7;
10     }
11 }
```

5.3.2 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 with 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. It causes this particular loop to be unrolled `n` times.

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:

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

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). Note that the value proposed for the cells `t[2][5]` and following is imprecise but correct. The set `{0; 1; }` does contain the actual value `1` of the cells.

5.4 Treatment of functions

5.4.1 Reusing the analysis of a function

If the user notices, in the application ey 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 em to launch the analysis with the option `-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 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 addresses 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 5.2.2). 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 rendering the computations of inputs, outputs and dependencies (section 6) incorrect for the callers of a function analyzed with this option.

5.4.2 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 7), especially using an assigns clause (section 7.2).

5.5 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.

```
1 void main(void) {
2   int x=1;
3   while(x++);
4   return;
5 }
```

The option `-no-overflow` should only be activated when it is guaranteed that the sizes of integer types do not change the concrete semantics of the analyzed code. Beware: voluntary overflows that are a deliberate part of the implemented algorithm are easy enough to recognize and to trust during a code review. Unwanted overflows, on the other hand, are rather difficult to spot using a code review. The next example illustrates this difficulty.

Consider the function `abs` that computes the absolute value of its `int` argument:

```
1 int abs(int x) {
2   if (x<0) x = -x;
3   return x;
4 }
```

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`. Without the `-no-overflow` option, on the other hand, the value analysis detects that the value returned by this function `abs` may not be a positive integer if `MININT` is among the arguments.

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

Chapter 6

Inputs, outputs and dependencies

Frama-C can compute and display the inputs (addresses of locations read from), outputs (addresses 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 5.4.1).

6.1 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` has certainly been overwritten, whereas "`(and default:true)`" would mean that `y` *might* have been modified, and that if it had, its new value would only depend 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:

```
1  int b,c,d,e;
2
3  void loop_branch(int a)
4  {
5    if (a)
6      b = c;
7    else
8      while (1) d = e;
9  }
```

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`.

```
1  int a, b, c;
2
3  void double_assign(void)
4  {
5    a = b;
6    a = c;
7  }
```

6.2 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.

6.3 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.

6.4 Operational inputs

The name "operational inputs of a function" is currently given to the locations that are read without having been previously 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`.

```
1  int b, c, d, e, *p;
2
3  void op(int a)
4  {
5    a = *p;
6    a = b;
7    if (a)
8      b = c;
9    else
10     while (1) d = e;
11 }
```

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 illicit memory access), whereas on the other hand the analyzer guarantees that it is not necessary to initialize `a`.

Chapter 7

Annotations

The language for annotations is ACSL. Only a subset of the properties that can be expressed in ACSL can effectively be of service to or be checked by the value analysis plug-in.

7.1 Preconditions, postconditions and assertions

7.1.1 Truth value of a property

When a precondition, postcondition or assertion is encountered, the analyzer 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.

7.1.2 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 5.3.2), if an assertion is in the shape of a disjunction, then the reduction of the state by the assertion may be computed independently for each sub-formula in the disjunction. 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.

```
1  int main(void)
2  {
3    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 addresses) 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: eir 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 that 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.

7.2 The "assigns" clauses

Assign clauses in the contract of a function indicate which variables 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`.

```
1 /*@ assigns p->balance;
2   @*/
3 void withdraw(purse *p,int s) {
4   p->balance = p->balance - s;
5 }
```

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 purposely removed because it did something that was complicated and unimportant for the analysis at hand.

Chapter 8

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.

8.1 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.

8.1.1 The `malloc` function

The file `share/malloc.c` contains various models for the `malloc` function. To choose a model, one of the following symbol must be defined before `#including` the file `share/malloc.c`:
`FRAMA_C_MALLOC_INDIVIDUAL`,
`FRAMA_C_MALLOC_POSITION`,
`FRAMA_C_MALLOC_CHUNKS` or
`FRAMA_C_MALLOC_HEAP`.

The particularities of each modelization 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 others when 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, the modelizations `FRAMA_C_MALLOC_INDIVIDUAL` and `FRAMA_C_MALLOC_CHUNKS` may lead the analysis to enter an infinite computation, which would eventually result in an "out of memory" error for the analyzer.

```
1 #define FRAMA_C_MALLOC_INDIVIDUAL
2 #include "share/malloc.c"
3
4 void main(void)
5 {
6   int * p = malloc(sizeof(int));
7   ...
8 }
```

8.1.2 Mathematical operations over floating-point numbers

Few functions are currently available. The available functions are listed in `share/math.h`. In order to use these functions, `share/math.c` must be added to the list of files that compose the analysis project.

8.1.3 String manipulation functions

Few functions are currently available. The available functions are listed in `share/libc.h`. In order to use these functions, `share/libc.c` must be added to the list of files that compose the analysis project.

8.2 Parameterizing the analysis

8.2.1 Adding non-determinism

The following functions, declared in `share/builtin.c`, allow to introduce some non-determinism in the analysis. The results given by the value analysis are valid **for all values proposed by the user**, as opposed to what a test-generation tool would typically do. A tool based on testing techniques would indeed necessarily pick only a subset of 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:

```
1 #include "share/builtin.h"
2
3 int A,B,X;
4 void main(void)
5 {
6   A = Frama_C_nondet(6, 15);
7   B = Frama_C_interval(-3, 10);
8   X = A * B;
9 }
```

With the command

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

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

8.3 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 5.3.2). 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.

8.3.1 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()`.

8.3.2 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 place-holder "`name`" can be replaced by an arbitrary identifier `s`. This identifier will appear in the output of the analyzer along with the value of the argument. It is recommended to use different identifiers 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 9

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 may hamper its manipulation. On the other hand, syntactic unrolling, by explicitly separating iteration steps, allows to use the graphical user interface `frama-c-gui` to observe values or express properties for a specific iteration step of the loop.

The `-slevel` option does not allow to observe a specific iteration step of the loop. In fact, this option may even be a little confusing for the user when the program contains loops for which the analysis can not decide easily the truth value of the condition, 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 works with loops that are built using gotos instead of for or while. The `-slevel` option requires less memory than `-ulevel`, and as a consequence it can sometimes be 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.

[1] if-then-else statements are "unrolled" in a manner similar to loops

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?

The answers to these questions are "yes" and "yes". Let us consider the following example:

```
1  int x,y;
2  void main(void)
3  {
4    int *p=NULL;
5    x = *p;
6    y = x / 0;
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. This technique is called debugging. Once the issue has been corrected in the source code under analysis — more precisely, once the user has convinced emself 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 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 happens 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 }
```

Analyzing this example with the default options produces:

```
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 emself that the threat on line

7 is false, ey 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 pre-processed. What should I do?

The annotations being contained inside C comments, they *a priori* aren't affected by the pre-processing. 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 pre-process the annotations, the following command-line should be used:

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

Acknowledgments

Dillon Pariente has provided helpful suggestions on this document since it was an early draft. I am especially thankful to him for finding the courage to read it version after version. Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, David Delmas, Benjamin Monate, Anne Pacalet, Julien Signoles provided useful comments in the process of getting the text 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 pre-processed code and inhibits the generation of `#line` directives respectively.