frama C

Software Analyzers

Value Analysis

Frama-C's value analysis plug-in

Sodium-20150201-rc2

Pascal Cuoq and Boris Yakobowski with Virgile Prevosto

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

This work has been supported by the ANR project U3CAT (ANR-08-SEGI-021-01).

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 an analyzed program. The value analysis warns about possible run-time errors in the analyzed program. Lastly, synthetic information about each analyzed function can be computed automatically from the values provided by the value analysis: these functionalities (input variables, output variables, and information flow) are 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.com/.

In technical terms, the value analysis is context-sensitive and path-sensitive. In non-technical terms, this means that it tries to offer precise results for a large set of C programs. Heterogeneous pointer casts, function pointers, and floating-point computations are handled.

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:

introduction.c

```
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 value of `z` is again 1 or 3.

When the plug-in indicates the value of `y` is 1 or 3 at the end of function `f`, it implicitly computes the union of all the values in `y` at each passage through this program point throughout any 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

An analyzed application may contain run-time errors: divisions by zero, invalid pointer accesses,...

rte.c

```
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 with `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 happen that, because of approximations during 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, 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. For instance, a particular division in the analyzed program is the object of a warning as soon as the set of possible values for the divisor contains zero. Only if the set of possible values computed by the value analysis does not contain zero is the warning omitted, and that means that the divisor really cannot be null at run-time.

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 from 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

Tutorial

Some use cases for the value analysis...

2.1 Introduction

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 during tests 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. Testing coverage criteria can help decide 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 justified empirically only.

2.2 Target code: Skein-256

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 to familiarize ourselves with Skein's reference implementation, 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.

2.3 Using the value analysis for getting familiar with Skein

2.3.1 Writing a test

After extracting the archive `skein_NIST_CD_010509.zip`, 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 with the function declarations 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 first glance is that the hash of an 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:

main_1.c

```
#include "skein.h"

#define HASHLEN (8)

int printf(const char*,...);

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;
```

```
14    Skein_256_Init( &skein_context, HASHLEN);
15    Skein_256_Update( &skein_context, msg, 80);
16    Skein_256_Final( &skein_context, hash);
17    for (i=0; i<HASHLEN; i++)
18      printf("%d\n", hash[i]);
19    return 0;
20  }
```

In order to make the test useful, we have to print the obtained hash value. Because the result we are interested in is an 8-byte number, represented as a char array of arbitrary characters (some of them non-printable), we cannot use string-printing functions, hence the `for` loop at lines 15-16.

With luck, the compilation goes smoothly and we obtain the hash value:

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

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

2.3.2 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 the following command. The analysis should not take more than a few seconds:

```
frama-c -val *.c >log
```

Frama-C has its own model of the target platform (the default target is a little-endian 32-bit platform). It also uses the host system's preprocessor. If you want to do the analysis for a different platform than the host platform, you need to provide Frama-C with a way to pre-process the files as they would be during an actual compilation.

There are analyses where the influence of host platform parameters is not noticeable. The analysis we are embarking on is not one of them. If you pre-process the Skein source code with a 64-bit compiler and then analyze it with Frama-C targetting its default 32-bit platform, the analysis will be meaningless and you won't be able to make sense of this tutorial. If you are using a 64-bit compiler, simply match Frama-C's target platform with your host header files by systematically adding the option `-machdep x86_64` to all commands in this tutorial.

The "`>log`" part of the command sends all the messages emitted by Frama-C into a file named "`log`". The value analysis is verbose for a number of reasons that will become clearer later in this tutorial. 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

it 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 of Frama-C may find issues 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.

2.3.3 Missing functions

Since we are trying out the library for the first time, something else 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 without a definition. Since these functions 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 work. Without it, it can only guess what each function does from the function's prototype, which is both inaccurate and likely incorrect.

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 missing 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 analysis. When analyzing a small part of a large codebase, 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 handle the absence of 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. On the other hand, string manipulation functions often feature difficult-to-analyze architecture-dependent optimizations. For these, 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  }
```

Some compilation platforms' headers define the names `memset` and `memcpy` in ways that make it impossible to provide your own implementation for these functions. If this happens for yours, you can try placing your own header in the analysis directory, under the name "`string.h`"

```
1  typedef unsigned int size_t;
2
3  void* memcpy(void* region1, const void* region2, size_t n);
4
5  void* memset (void* dest, int val, size_t len);
```

In some circumstances, the results of the value analysis may remain useful even when letting Frama-C guess the effects of missing functions. In the case of functions that take pointers as arguments, however, 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.

2.3.4 First meaningful analysis

If we run the analysis again with definitions for `memset` and `memcpy`, the newly obtained log 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`. Searching the source code reveals that this variable is in fact declared `static` as it should be. Frama-C can display the value of static variables (something that is annoying to do 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. The GUI displays the original source code alongside the transformed one. It also allows to inspect values of variables.

A quick inspection shows that the Skein functions use variable `ONE` 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. When the analysis takes too long, these messages help the user understand where time is spent. We can ignore these messages now. The following lines are also, in the current context, progression messages:

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

```
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` was introduced by Frama-C during source code normalization. The relation between this variable and the original source code is easiest to see in the GUI. Alternately, the command `frama-c *.c -print` shows how the original source code had been transformed by Frama-C (note, however, that the location `lib.c: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, annotated with 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 ;
  ptr = (unsigned char *)dest;
  while (1) {
    size_t tmp_0 ;
    unsigned char *tmp ;
    { /*undefined sequence*/ tmp_0 = len; len -= (size_t )1; ; }
    if (! (tmp_0 > (size_t )0)) { break; }
    { /*undefined sequence*/
      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. The important consequence is that if you get the tool to remain silent, you have *formally verified* that no run-time error could happen when running the program.

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 dangerous by the analyzer. This operation 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 misguidedly have written the `>0` test in this example as a way to test for `NULL`. Ey[2] 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 compiler places variable `a` precisely at address 1432. 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[3] 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:

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

The GUI allows to inspect the sets of values obtained during the analysis 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.

2.4 Searching for bugs

2.4.1 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 and takes no inputs, only one 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

[2] This manual uses Spivak pronouns: http://en.wikipedia.org/wiki/Spivak_pronoun
[3] The fine print is the reference part of this manual

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, `-slevel` is one of the value analysis options that can never cause it to become incorrect. Such options, when used wrongly, may make the analysis slower, provide hints that are not useful, or even possibly counter-productive (making the analysis less precise), but they cannot 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 >log
```

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:

```
...
[kernel] warning: Neither code nor specification for function printf,
        generating default assigns from the prototype
[value] Done for function printf
main_1.c:18:[kernel] warning: completely indeterminate value in
                     hash with offsets {8}.
main_1.c:18:[kernel] warning: accessing uninitialized left-value:
                     assert \ initialized (&hash[i]);
main_1.c:18:[value] Non-termination in evaluation of
                    library function call lvalue argument hash[i]
[value] Recording results for main
[value] done for function main
...
```

These messages mean that lvalue `hash[i]`, as found in line 16 of our file `main_1.c`, may be uninitialized[4]. Actually, the analyzer finds that for one execution branch, it is certain to be uninitialized. This is when reading the memory location "`hash with offset 8`" where the

[4] An lvalue is a C expression that exists in memory. Examples of lvalues are a simple variable `x`, the cell of

offset is 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 analyzer finds that function `main` does not terminate.

The next warning, `Non-termination in evaluation of ...`, 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 this situation 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.

2.4.2 Making sense of 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 this local array remained uninitialized.

Looking again at the header file `skein.h`, we may now notice that in function `Skein_256_Init`, the formal parameter for the desired hash length is named `hashBitLen`. This parameter should certainly 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 first 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 ∈ {8}
  hash[0] ∈ {224}
      [1] ∈ {56}
      [2] ∈ {146}
      [3] ∈ {251}
      [4] ∈ {183}
      [5] ∈ {62}
      [6] ∈ {26}
      [7] ∈ {48}
```

Meanwhile, compiling and executing the fixed test produces the result:

```
224
56
146
251
```

an array `t[i]`, or a pointed location `*p`. Because an lvalue lies in memory, it may have uninitialized contents. Using an lvalue when it has not yet been initialized is a forbidden behavior that compilers do not detect in all cases, and this is only one of the numerous pitfalls of C.

```
183
62
26
48
```

2.5 Guaranteeing the absence of bugs

2.5.1 Generalizing the analysis to arbitrary messages of fixed length

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. The results of this analysis only prove the absence of run-time errors[5] 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:

main_2.c

```
1  #include "skein.h"
2
3  #define HASHLEN (8)
4
5  u08b_t msg[80];
6
7  int printf(const char*,...);
8  int Frama_C_interval(int,int);
9
10 void main(void)
11 {
12   int i;
13   u08b_t hash[HASHLEN];
14   Skein_256_Ctxt_t skein_context;
15
16   for (i=0; i<80; i++) msg[i]=Frama_C_interval(0, 255);
17
18   Skein_256_Init( &skein_context, HASHLEN * 8);
19   Skein_256_Update( &skein_context, msg, 80);
20   Skein_256_Final( &skein_context, hash);
21 }
```

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 commandline must now include the file `builtin.c` from the Frama-C distribution, which defines `Frama_C_interval`:

```
frama-c -slevel 100 -val *.c `frama-c -print-share-path`/builtin.c >log 2>&1
```

[5] and the absence of conditions that *should* be run-time errors – like the uninitialized access already encountered

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 argument 64 for the size), parse an arbitrary message and produce a hash in a local `u08b_t` array of size 8.

2.5.2 Verifying functional dependencies

If we had written a formal specification for function Skein, we would soon have expressed that we expected it to modify the buffer `hash` that is passed to `Skein_256_Final` (all 8 bytes of the buffer), to compute the new contents of this buffer from the contents of the input buffer `msg` (all 80 bytes of it), and from nothing else.

During the value analysis, we have seen that all of the buffer `hash` was always modified in the conditions of the analysis: the reason is that this buffer was uninitialized before the sequence of calls, and guaranteed to be initialized after them.

We can get the complete list of locations that may be modified by each function by adding the option `-out` to the other options we were already using. These locations are computed in a quick analysis that comes after the value analysis. In the results of this analysis, we find:

```
[inout] Out (internal) for function main:
         Frama_C_entropy_source; msg[0..79]; i; hash[0..7]; skein_context; tmp
```

The "(internal)" reminds us that this list includes the local variables of `main` that have been modified. Indeed, all the variables that we could expect to appear in the list are here: the input buffer, that we initialized to all possible arbitrary 80-char messages; the loop index that we used in doing so; the output buffer for receiving the hash; and Skein's internal state, that was indirectly modified by us when we called the functions `Init`, `Update` and `Final`.

If we want the outputs of the sequence to appear more clearly, without the variables we used for instrumenting, we can put it in its own function:

```
1  u08b_t hash[HASHLEN];
2
3  void do_Skein_256(void)
4  {
5    Skein_256_Ctxt_t skein_context;
6    Skein_256_Init( &skein_context, HASHLEN * 8);
7    Skein_256_Update( &skein_context, msg, 80);
8    Skein_256_Final( &skein_context, hash);
9  }
10
11 void main(void)
12 {
13   int i;
14
15   for (i=0; i<80; i++) msg[i]=Frama_C_interval(0, 255);
16
17   do_Skein_256();
18 }
```

Using option `-out-external` in order to obtain lists of locations that exclude each function's local variables, we get:

```
[inout] Out (external) for function do_Skein_256:
         hash[0..7]
```

This means that no location other than `hash[0..7]` was modified by the sequence of calls to Skein-256 functions. It doesn't mean that each of the cells of the array was overwritten: we have to rely on the results of the value analysis when `hash` was a local variable for that result. But it means that when used in conformance with the pattern in this program, the functions do not accidentally modify a global variable. We can conclude from this analysis that the functions are re-entrant as long as the concurrent computations are being made with separate contexts and destination buffers.

Keeping the convenient function `do_Skein_256` for modelizing the sequence, let us now compute the functional dependencies of each function. Functional dependencies list, for each output location, the input locations that influence the final contents of this output location:

```
frama-c -val *.c -slevel 100 -deps
```

```
Function Skein_256_Init:
  skein_context.h.hashBitLen FROM ctx; hashBitLen
              .h{.bCnt; .T[0..1]; } FROM ctx
              .X[0..3] FROM msg[0..63];
                           skein_context{.X[0..3]; .b[0..31]; }; ctx;
                           hashBitLen; ONE[bits 0 to 7] (and SELF)
              .b[0..31] FROM ctx (and SELF)
  \result  FROM \nothing

Function Skein_256_Update:
  skein_context.h.bCnt FROM skein_context.h.bCnt; ctx; msgByteCnt
              .h.T[0] FROM skein_context.h{.bCnt; .T[0]; }; ctx; msgByteCnt
              .h.T[1] FROM skein_context{.h.bCnt; .h.T[1]; }; ctx;
                           msgByteCnt
              {.X[0..3]; .b[0..15]; } FROM msg[0..79];
                                          skein_context{.h{.bCnt; .T[0..1]; };
                                                        .X[0..3]; .b[0..31]; };
                                           ctx; msg_0; msgByteCnt (and SELF)
  \result  FROM \nothing

Function Skein_256_Final:
  hash[0..7] FROM msg[0..79]; skein_context; ctx;
                 hashVal; ONE[bits 0 to 7] (and SELF)
  skein_context.h.bCnt FROM skein_context.h.hashBitLen; ctx (and SELF)
              .h.T[0] FROM skein_context.h{.hashBitLen; .bCnt; .T[0]; }; ctx
              .h.T[1] FROM skein_context{.h.hashBitLen; .h.T[1]; }; ctx
              {.X[0..3]; .b[0..15]; } FROM msg[0..79]; skein_context;
                                          ctx; ONE[bits 0 to 7] (and SELF)
              .b[16..31] FROM skein_context.h.bCnt; ctx (and SELF)
  \result  FROM \nothing
```

The functional dependencies for the functions `Init`, `Update` and `Final` are quite cryptic. They refer to fields of the struct `skein_context`. We have not had to look at this struct yet, but its type `Skein_256_Ctxt_t` is declared in file `skein.h`.

In the results, the mention `(and SELF)` means that parts of the output location may keep their previous values. Conversely, absence of this mention means that the output location is guaranteed to have been completely over-written when the function returns. For instance, the field `skein_context.h.T[0]` is guaranteed to be over-written with a value that depends

only on various other subfields of `skein_context.h`. On the other hand, the `-deps` analysis does not guarantee that all cells of `hash` are over-written — but we previously saw we could deduce this information from the value analysis' results.

Since we don't know how the functions are supposed to work, it is difficult to tell if these dependencies are normal or reveal a problem. Let us move on to the functional dependencies of `do_Skein_256`:

```
Function do_Skein_256:
  hash[0..7] FROM msg[0..79]; ONE[bits 0 to 7] (and SELF)
```

These dependencies make the effect of the functions clearer. The `FROM msg[0..79]` part is what we expect. The `and SELF` mention is an unfortunate approximation. The dependency on global variable `ONE` reveals an implementation detail of the library (the variable is used to detect endianness dynamically). Finding out about this variable is a good thing: it shows that a possibility for a malicious programmer to corrupt the implementation into hashing the same message to different digests would be to try to change the value of `ONE` between computations. Checking the source code for mentions of variable `ONE`, we can see it is used to detect endianness and is declared `const`. On an MMU-equipped platform, there is little risk that this variable could be modified maliciously from the outside. However, this is a vivid example of how static analysis, and especially correct analyses as provided in Frama-C, can complement code reviews in improving trust in existing C code.

Assuming that variable `ONE` keeps its value, a same input message is guaranteed always to be hashed by this sequence of calls to the same digest, because option `-deps` says that there is no other memory location `hash` is computed from, not even an internal state.

A stronger property to verify would be that the sequence `Init(...)`, `Update(...,80)`, `Final(...)` computes the same hash as when the same message is passed in two calls `Update(...,40)`. This property is beyond the reach of the value analysis. In the advanced tutorial, we show how to verify easier properties for sequences of calls that include several calls to `Update`.

2.5.3 Generalizing to arbitrary numbers of Update calls

As an exercise, try to verify that there cannot be a run-time error when hashing arbitrary contents by calling `Update(...,80)` an arbitrary number of times between `Init` and `Final`. The general strategy is to modify the C analysis context we have already written in a way such that it is evident that it captures all such sequences of calls, and also in a way that launched with adequate options, the value analysis does not emit any warning.

The latter condition is harder than the former. Observing results (with the GUI or observation functions described in section 8.3) can help to iterate towards a solution. Be creative. The continuation of this tutorial can be found online at http://blog.frama-c.com/index.php?tag/skein; read available posts in chronological order.

Chapter 3

What the value analysis provides

Here begins the reference section of this manual.
This chapter categorizes and describes the outputs of the value analysis.

3.1 Values

The value analysis plug-in accepts queries for 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.

3.1.1 Interactive and programmatic interfaces

Both the user (through the GUI) or a Frama-C plug-in can request the evaluation, at a specific statement, of an lvalue (or an arbitrary expression). The variation domain thus obtained contains all the values that this lvalue or expression may have anytime an actual execution reaches the selected statement.

In fact, from the point view of the value analysis, the graphical interface that allows to observe the values of variables in the program is very much an ordinary Frama-C plug-in. It uses the same functions (registered in module `Db.Value`) as other plug-ins that interface with the value analysis. This API is not very well documented yet, but one early external plug-in author used the GUI to get a feeling of what the value analysis could provide, and then used the OCaml source code of the GUI as a reference for obtaining this information programmatically.

3.1.2 Value analysis API overview

The function `!Db.Value.access` is one of the functions provided to custom plug-ins. It takes a program point (of type `Cil_types.kinstr`), the representation of an lvalue (of type

`Cil_types.lval`) and returns a representation of the possible values for the lvalue at the program point.

Another function, `!Db.Value.lval_to_loc`, translates the representation of an lvalue 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 lvalue (array indices, dereferenced expressions). Thanks to this and similar functions, a custom plug-in may reason entirely in terms of abstract locations, and completely avoid the problems of pointers and aliasing.

The Frama-C Plug-in Development Guide contains more information about developing a custom plug-in.

3.1.3 Variation domains for expressions

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

A set of integers

The analyzer may have determined the variation domain of a variable is 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. 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).
- `"foo"` + D, which denotes offsets from a litteral string with contents "foo".

In all three cases, " + D" is omitted if D is $\{0\}$, that is, when there is no offset.

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 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.

3.1.4 Memory contents

Memory locations can contain, in addition to the above sets of values, uninitialized data and dangling pointers. It is illegal to use these special values in computations, which is why they are not listed as possible values for an expression. Reading from a variable that appears to be uninitialized causes an alarm to be emitted, and then the set of values for the variable is made of those initialized values that were found at the memory location.

3.1.5 Interpreting the variation domains

Most modern C compilation platforms 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. It is partly for this reason that offsets D are expressed in bytes in domains of the form `{{`&x + D; ... `}}`.

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.

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 that fit within the type of the variable or expression that is being printed.

- `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 }}` 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 ; &y }}` 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 some misaligned and invalid ones.

3.1.6 Origins of approximations

Approximated values may contain information about the origin of the approximations. In this case the value is shown as "`garbled mix of &{`x_1`; ... `x_n`} (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 }) }}`.
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, an alarm would be emitted, and the analysis would go 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 }) }}`.

Well value

When generating an initial state to start the analysis from (see section 5.2.3 for details), 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.

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

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 }) }}`.

3.2 Log messages emitted by the value analysis

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

3.2.1 Results

With the batch version of Frama-C, the results of the computations are displayed on the standard output. For some computations, the results can not easily be represented in a human-readable way. In this case, they are 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.

3.2.2 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 contain the word `assert`. Frama-C comes with a common specification language for all plug-ins, called ACSL (http://frama-c.com/acsl.html). Most of the proof obligations emitted by the value analysis are expressed in ACSL. In any case, each proof obligation message describes the nature and the origin of the obligation.

When using the GUI version of Frama-C, proof obligations are inserted in the normalized source code. With the batch version, option `-print` produces a version of the analyzed source code annotated with the proofs obligations. The companion option `-ocode <file.c>` allows to specify a filename for the annotated source code to be written to. Please note that the proof obligations that are not yet expressed as ACSL properties are missing from these outputs.

Division by zero

When dividing by an expression that the analysis is not able to guarantee 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 the propagation of this execution path. If the divisor seems to be able to take non-null values, the analyzer is allowed to take into account the property that the divisor is different from 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: ... division by zero: assert (int)(x*y) != 0;
div.c:5: ... 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.

Undefined logical shift

Another arithmetic alarm is the alarm emitted for logical shift operations on integers where the second operand may be larger than the size in bits of the first operand's type. Such an operation is left undefined 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 is 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: ... invalid RHS operand for shift: assert 0 <= c < 32;
```

Overflow in integer arithmetic

By default, the value analysis emits alarms for — and reduces the sets of possible results of — signed arithmetic computations where the possibility of an overflow exists. Indeed, such overflows have an undefined behavior acccording to paragraph 6.5.5 of the ISO/IEC 9899:1999 standard. If useful, it is also possible to assume that signed integers overflow according to a 2's complement representation. The option `-no-warn-signed-overflow` can be used to this end. A reminder message is nevertheless emitted operations that are detected as potentially overflowing. Regardless of the value of option `-warn-signed-overflow`, a warning is emitted on signed arithmetic operations applied to the cast to signed int of an address, since the compiler may place variables in memory at will.

By default, no alarm is emitted for arithmetic operations on unsigned integers for which an overflow may happen, since such operations have defined semantics according to the ISO/IEC 9899:1999 standard. If one wishes to signal and prevent such unsigned overflows, option `-warn-unsigned-overflow` can be used.

Finally, no alarm is emitted for downcasts to signed or unsigned integers. In the signed case, the least significant bits of the original value are used, and are interpreted according to 2's complement representation. Frama-C's options `-warn-signed-downcast` and `-warn-unsigned-downcast` are not honored by the value analysis. The RTE plugin can be used to generate the relevant assertions before starting an analysis.

Overflow in conversion from floating-point to integer

An alarm is emitted when a floating-point value appears to exceed the range of the integer type it is converted to.

```
1 #include ".../share/builtin.h"
2
3 int main()
4 {
5   float f = Frama_C_float_interval(2e9, 3e9);
6   return (int) f;
7 }
```

```
...
ov_float_int.c:6:[kernel] warning: overflow in conversion of f
```

```
    ([2000000000. .. 3000000000.]) from floating-point to integer.
    assert -2147483649 < f < 2147483648;

...
f ∈ [2000000000. .. 3000000000.]
__retres ∈ [2000000000..2147483647]
```

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 example of this first kind of alarm can be seen in the following example.

```
1 double sum(double a, double b)
2 {
3   return a+b;
4 }
```

```
frama-c -val -main sum double_op_res.c
```

```
double_op_res.c:3:[kernel] warning: non-finite double value
    ([-1.79769313486e+308 .. 1.79769313486e+308]):
    assert \is_finite((double)(a+b));
```

An alarm is also emitted when the program uses as argument to a floating-point operation a value from memory that does not ostensibly represent a floating-point number. This can happen with a union type with both `int` and `float` fields, or in the case of a conversion from `int*` to `float*`. The emitted alarm excludes the possibility of the bit sequence used as a the argument representing `NaN`, an infinite, or an address. See the example below.

```
1 union { int i ; float f ; } bits;
2
3 float r;
4
5 int main () {
6   bits.i = unknown_fun();
7   r = 1.0 + bits.f;
8   return r > 0.;
9 }
```

```
frama-c -val double_op_arg.c
```

```
double_op_arg.c:7:[kernel] warning: non-finite float value ([--..--]):
    assert \is_finite(bits.f);
```

Uninitialized variables and dangling pointers to local variables

An alarm may be emitted if the application appears to read the value of a local variable that has not been initialized, or if it appears to manipulate the address of a local variable outside of the scope of said variable. Both issues occur in the following example:

```
1  int *f(int c)
2  {
3    int r, t, *p;
4    if (c) r = 2;
5    t = r + 3;
6    return &t;
7  }
8
9  int main(int c)
10 {
11   int *p;
12   p = f(c);
13   return *p;
14 }
```

The value analysis emits alarms for lines 5 (variable `r` may be uninitialized) and 13 (a dangling pointer to local variable `t` is used). As of the current version, only the first message is recorded as an ACSL property by Frama-C's kernel.

```
uninitialized.c:5: ... accessing uninitialized left-value: assert \initialized(&r);
uninitialized.c:13: ... accessing left-value p that contains escaping addresses;
                        assert(\defined(&p))
```

By default, the value analysis does not emit an alarm for a copy from memory to memory when the copied values include dangling addresses or uninitialized contents. This behavior is safe because the value analysis warns later, as soon as an unsafe value is used in a computation –either directly or after having been copied from another location. The copy operations for which alarms are not emitted are assignments from lvalues to lvalues (`lv1 = lv2;`), passing lvalues as arguments to functions (`f(lv1);`), and returning lvalues (`return lv1;`). An exception is made for lvalues passed as arguments to library functions: in this case, because the function's code is missing, there is no chance to catch the undefined access later; the analyzer emits an alarm at the point of the call.

The behavior documented above was implemented to avoid spurious warnings where the copied lvalues are structs or unions. In some cases, it may be normal for some fields in a struct or union to contain such dangerous contents. Option `-val-warn-copy-indeterminate` can be used to obtain a more aggressive behavior. Specifying `-val-warn-copy-indeterminate f` on the command-line will cause the analyzer to also emit an alarm on all dangerous copy operations occurring in function `f`, as long as the copied lvalues are not structs or unions. The syntax `@all` can also be used to activate this behavior for all functions.

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], *p;
2  void main()
3  {
4    for (i=0; i<=10; i++)
5      if (unknownfun()) t[i] = i;
6    p = t + 12;
7    if (unknownfun()) *p = i;
8    p[-6] = i;
9  }
```

In the above example, the analysis is not able to guarantee that the memory accesses `t[i]` and `*p` are valid, so it emits a proof obligation for each:

```
invalid.c:5: ... accessing out of bounds index [0..10]. assert i < 10;
invalid.c:7: ... out of bounds write. assert \valid(p);
```

(Notice that no alarm `assert 0 <= i` is emitted, as the analysis is able to guarantee that this always holds.)

The choice between these two kinds of alarms is influenced by option `-unsafe-arrays`, as described page 57.

Note that line 6 or 8 in this example could be considered as problematic in the strictest interpretation of the standard. The value analysis omits warnings for these two lines according to the attitude described in 4.5.1. An option to warn about these lines could happen if there was demand for this feature.

Undefined pointer comparison alarms

Proof obligations can also be emitted for pointer comparisons whose results may vary from one compilation to another, such as `&a < &b` or `&x+2 != NULL`. These alarms do not necessarily correspond to run-time errors, but relying on an undefined behavior of the compiler is in general undesirable (although this one is rather benign for current compilation platforms).

Although these alarms may seem unimportant, they should still be checked, because the value analysis may reduce the propagated states accordingly to the emitted alarm. For instance, for the `&x+2 != NULL` comparison, after emitting the alarm that the quantity `&x+2` must be reliably comparable to 0, the analysis assumes that the result of the comparison is 1. The consequences are visible when analyzing the following 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 reproducible 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 analyzer can be instructed to propagate past these undefined pointer comparisons with option `-undefined-pointer-comparison-propagate-all`. With this option, in the example program above, the values `0` and `1` are considered as results for the condition as soon as `p` becomes an invalid address. Therefore, the value analysis does not predict that the program does not terminate.

Undefined side-effects in expressions

The C language allows compact notations for modifying a variable that is being accessed (for instance, `y = x++;`). The effect of these pre- or post-increment (or decrement) operators is undefined when the variable is accessed elsewhere in the same statement. For instance,

`y = x + x++;` is undefined: the code generated by the compiler may have any effect, and especially not the effect expected by the programmer.

Sometimes, it is not obvious whether the increment operation is defined. In the example `y = *p + x++;`, the post-increment is defined as long as `*p` does not have any bits in common with `x`.

```
1  int x, y, z, t, *p, *q;
2  void main(int c)
3  {
4    if (c&1)
5      y = x + x++;
6    p = c&2 ? &x : &t;
7    y = *p + x++;
8    q = c&4 ? &z : &t;
9    y = *q + x++;
10 }
```

```
frama-c -val se.c -unspecified-access
```

With option `-unspecified-access`, three warnings are emitted during parsing. Each of these only mean that an expression with side-effects will require checking after the Abstract Syntax Tree has been elaborated. For instance, the first of these warnings is:

```
se.c:5:[kernel] warning: Unspecified sequence with side effect:
                /*  <-  */
                tmp = x;
                /* x <- */
                x ++;
                /* y <- x tmp */
                y = x + tmp;
```

Then the value analysis is run on the program. In the example at hand, the analysis finds problems at lines 5 and 7.

```
se.c:5: ... undefined multiple accesses in expression. assert \separated(&x, &x);
se.c:7: ... undefined multiple accesses in expression. assert \separated(&x, p);
```

At line 5, it can guarantee that an undefined behavior exists, and the analysis is halted. For line 7, it is not able to guarantee that `p` does not point to `x`, so it emits a proof obligation. Since it cannot guarantee that `p` always points to `x` either, the analysis continues. Finally, it does not warn for line 9, because it is able to determine that `*q` is `z` or `t`, and that the expression `*q + x++` is well defined.

Another example of statement which deserves an alarm is `y = f() + g();`. For this statement, it is unspecified which function will be called first. If one of them modifies global variables that are read by the other, different results can be obtained depending on factors outside the programmer's control. At this time, the value analysis does not check if these circumstances occur, and silently chooses an order for calling `f` and `g`. The statement `y = f() + x++;` does not emit any warning either, although it would be desirable to do so if `f` reads or writes into variable `x`. These limitations will be addressed in a future version.

Partially overlapping lvalue assignment

Vaguely related to, but different from, undefined side-effects in expressions, the value analysis warns about the following program:

```
1  struct S { int a; int b; int c; };
2
3  struct T { int p; struct S s; };
4
5  union U { struct S s; struct T t; } u;
6
7  void copy(struct S *p, struct S *q)
8  {
9    *p = *q;
10 }
11
12 int main(int c, char **v){
13   u.s.b = 1;
14   copy(&u.t.s, &u.s);
15   return u.t.s.a + u.t.s.b + u.t.s.c;
16 }
```

The programmer thought ey was invoking implementation-defined behavior in the above program, using an union to type-pun between structs S and T. Unfortunately, this program returns 1 when compiled with `clang -m32`; it returns 2 when compiled with `clang -m32 -O2`, and it returns 0 when compiled with `gcc -m32`.

For a program as simple as the above, all these compilers are supposed to implement the same implementation-defined choices. Which compiler, if we may ask such a rethorical question, is right? They all are, because the program is undefined. When function `copy()` is called from `main()`, the assignment `*p = *q;` breaks C99's 6.5.16.1:3 rule. This rule states that in an assignment from lvalue to lvalue, the left and right lvalues must overlap either exactly or not at all.

The program breaking this rule means compilers neither have to emit warnings (none of the above did) nor produce code that does what the programmer intended, whatever that was. Launched on the above program, the value analysis says:

```
partially overlapping lvalue assignment (u with offsets {32}, size <96> bits;
   u with offsets {0}, size <96> bits). assert p == q || \separated(p, q);
```

By choice, the value analysis does not emit alarms for overlapping assignments of size less than `int`, for which reading and writing are deemed atomic operations. Finding the exact cut-off point for these warnings would require choosing a specific compiler and looking at the assembly it generates for a large number of C constructs. This kind of fine-tuning of the analyzer for a specific target platform and compiler can be provided as a paying service.

Invalid function pointer access

When the value analysis encounters a statement of the form `(*e)(x);` and is unable to guarantee that expression `e` evaluates to a valid function address, an alarm is emitted.

3.2.3 Informational messages regarding propagation

Some messages warn that the analysis is making an operation likely to cause loss of precision. Other messages warn the user that unusual circumstances have been encountered by the value analysis.

In both these cases, the messages are not proof obligations and it is not mandatory for the user to act on them. They can be distinguished from proof obligations by the fact that they

do **not** use the word "assert". These messages are intended to help the user trace the results of the analysis, and give as much information as possible in order to help em find when and why 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 messages that result from the apparition of imprecision in the analysis are:

```
val9.c:17:[value] assigning non deterministic value for the first time

origin.c:14:[value] assigning imprecise value to pa1.
       The imprecision originates from Arithmetic {origin.c:14}

origin.c:15:[value] writing somewhere in {ta1}
       because of Arithmetic {origin.c:14}.
```

Examples of messages that correspond to unusual circumstances are:

```
shift .c:18: ... invalid shift of 32-bit value by {5555}.
       This path is assumed to be dead.

alloc.c:20: ... all target addresses were invalid.
       This path is assumed to be dead.

origin.i:86: ... local escaping the scope of local_escape_1 through esc2
```

3.2.4 Progress messages

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

```
[kernel] preprocessing with "gcc -C -E -I. skein.c"

[value] computing for function memset <-Skein_256_Init <-main.
       Called from skein.c:83.

[value] Recording results for RotL_64

[value] Done for function Skein_256_Init
```

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

3.3 About these alarms the user is supposed to check...

When writing a Frama-C plug-in to assist in reverse-engineering source code, it does not really make sense to expect the user to check the alarms that are emitted by the value analysis. Consider for instance Frama-C's slicing plug-in. This plug-in produces a simplified version of a program. It is often applied to large unfamiliar codebases; if the user is at the point where ey needs a slicer to make sense of the codebase, it's probably a bad time to require em to check alarms on the original unsliced version.

The slicer and other code comprehension plug-ins work around this problem by defining the results they provide as "valid for well-defined executions". In the case of the slicer, this is really the only definition that makes sense. Consider the following code snippet:

```
1  x = a;
2  y = *p;
3  x = x+1;
4  // slice for the value of x here.
```

This piece of program is begging for its second line to be removed, but if `p` can be the `NULL` pointer, the sliced program behaves differently from the original: the original program exits abruptly on most architectures, whereas the sliced version computes the value of `x`.

It is fine to ignore alarms in this context, but the user of a code comprehension plug-in based on the value analysis should study the categorization of alarms in section 3.2.2 with particular care. Because the value analysis is more aggressive in trying to extract precise information from the program than other analyzers, the user is more likely to observe incorrect results if there is a misunderstanding between him and the tool about what assumptions should be made.

Everybody agrees that accessing an invalid pointer is an unwanted behavior, but what about comparing two pointers with `<=` in an undefined manner or assuming that a signed overflow wraps around in 2's complement representation? Function `memmove`, for instance, typically does the former when applied to two addresses with different bases.

Currently, if there appears to be an undefined pointer comparison, the value analysis propagates a state that may contain only "optimistic" (assuming the undefined circumstances do not occur) results for the comparison result and for the pointers. In order to get all possible behaviors, option `-undefined-pointer-comparison-propagate-all` should be used.

It is possible to take advantage of the value analysis for program comprehension, and all existing program comprehension tools need to make assumptions about undefined behaviors. Most tools do not tell whether they had to make assumptions or not. The value analysis does: each alarm, in general, is also an assumption. Still, as implementation progresses and the value analysis becomes able to extract more information from the alarms it emits, one or several options to configure it either not to emit some alarms, or not to make the corresponding assumptions, will certainly become necessary. This is a consequence of the nature of the C language, very partially defined by a standard that also tries to act as lowest common denominator of existing implementations, and at the same time used for low-level programming where strong hypotheses on the underlying architecture are needed.

Chapter 4

Limitations and specificities

Nobody is perfect.

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.

4.1 Prospective features

The value analysis does not check all the properties that could be expected of it. Support for each of these missing features could arrive quickly if the need was expressed.

4.1.1 Strict aliasing rules

The value analysis does not check that the analyzed program uses pointer casts in a way that is compatible with strict aliasing rules. This is one of the major current issues of the C world because compilers only recently began to take advantage of these rules for optimization purposes. However, the issue has passed unnoticed for critical embedded code, because for various reasons including traceability, this code is compiled with most optimizations disabled.

4.1.2 Const attribute inside complex types

Some memory locations can be read from but not written to. An example of such a memory location is a constant string literal (`"foo"`). The ACSL specification language sports a predicate

\valid_read to express that a memory location is valid for reading but does not have to be valid for writing. Accordingly, the value analysis emits alarms using \valid_read(lv) when an lvalue lv is read from, and \valid(lv) when lv is written to.

Variables with a const-qualified type are among the memory locations that the value analysis knows to be good to read from but forbidden to write to. However, the const type qualifier is currently only handled when placed at the top-level of the type of a global variable or formal argument. Other const locations are considered writable.

4.1.3 Alignment of memory accesses

The value analysis currently does not check that memory accesses are properly aligned in memory. It assumes that non-aligned accesses have only a small time penalty (instead of causing an exception) and that the programmer is aware of this penalty and is accessing (or seems to be accessing) memory in a misaligned way on purpose. A more restrictive option would be quick to implement if someone needed it. Meanwhile, instructions to explicitly check for the alignment of some accesses can be inserted in the code following the howto at http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:explicit_alignment_howto.

4.2 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.

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    {
6      i++;
7      if (c(i))
8        y = 2*i;
9    }
10 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, it 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 5.3.1 introduces different ways for the user to influence the analyzer's strategy with respect to loops.

[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).

4.3 Functions

Without special instructions from the user, function calls are handled as if the body of the function had been expanded 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 4.2.

```
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     {
12       i++;
13       if (c(i))
14         f(2*i);
15     }
16   while (i<n);
17
18 }
```

Variadic function calls are not handled very precisely. 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`.

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

4.4 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 this manual, section 5.3.2.

4.4.1 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 4.3 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 4.2 and 4.3: 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.

4.4.2 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).

4.4.3 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.

4.4.4 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 void main(void) {
16   init1();
17   init2();
18   ...
19 }
```

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.

4.4.5 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. Instead, a volatile variable with scalar type is always assumed to contain `[--..--]` when it is read from, regardless of the contents that have been previously written to it. The values of volatile pointers (*i.e* variables with pointer type and volatile specifier) are an imprecise form of the addresses that have previously been assigned to the pointer. Finally, the handling of (non-volatile) pointers to volatile locations is not finalized yet: it is not clear how the various complicated situations that can arise in this case should be handled. For instance, pointers to volatile locations may be pointing to locations that also have non-volatile access paths to them.

4.5 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.

4.5.1 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 experienced 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 may exist 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 concrete example, the value analysis plug-in assumes two-complement representation, which the standard does not guarantee, and whose consequences can be seen when converting between signed and unsigned types or with signed arithmetic overflows. These parameters are all fixed with the `-machdep` option as described in section 5.2.6.

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.

4.6 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.

4.6.1 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`.

4.6.2 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.

4.6.3 Bases separation

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

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

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

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

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

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

```
1  int x,y,z,t,u;
2
3  void init_non_analyzable(void)
4  {
5    int *p;
6    // initialize variables with 52
7    for (p = &x; p <= &u; p++)
8      *p = 52;
9  }
10
11 void init_analyzable(void)
12 {
13   x = y = z = t = u = 52;
14 }
```

4.7 What the value analysis does not provide

Values that are valid even if something bad happens

The value analysis provides sets of possible values 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 cannot 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 execution reaches the end of `main`, 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 time* `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 the analyzer may have propagated separately. 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 }
```

With the option `-slevel` described in section 5.3.1, 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 the value analysis has propagated a specific state, and therefore guarantee the absence of run-time errors under the assumptions encoded in that state, is to build the intended state oneself, 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 intended.

Chapter 5

Parameterizing the analysis

The value analysis is automatic but gives you a little bit of control. Just in case.

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 understood by the value analysis plug-in are described in this chapter. The files are the C files containing source code to analyze.

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

The options documented in this manual can be listed, with short descriptions, from the command-line. Option `-kernel-help` lists options that affect all plug-ins. Option `-value-help` lists options specific to the value analysis plug-in. The options `-kernel-help` and `-value-help` also list advanced options that are not documented in this manual.

Example:

```
frama-c -value-help
```

```
...
-slevel <n>       use <n> as number of path to explore in parallel
                  (defaults to 0)
...
```

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.`, but another preprocessor may be employed.

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.

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, Frama-C invokes the preprocessor 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' file1.src file2.i
frama-c-gui -val -cpp-command 'gcc -C -E -I. -o %2 %1' file1.c file2.i
frama-c-gui -val -cpp-command 'copy %1 %2' file1.c file2.i
frama-c-gui -val -cpp-command 'cat %1 > %2' file1.c file2.i
frama-c-gui -val -cpp-command 'CL.exe /C /E %1 > %2' file1.c file2.i
```

5.1.2 Activating the value analysis

Option `-val` activates the value analysis. Sets of values for the program's variables at the end of each analyzed function are displayed on standard output.

Many functionalities provided by Frama-C rely on the value analysis' results. Activating one of these automatically activates the value analysis, without it being necessary to provide the `-val` option on the command-line. In this case, it should be kept in mind that all other value analysis options remain available for parameterization.

5.1.3 Saving the result of an analysis

The option `-save s` saves the state of the analyzer, after the requested computations have 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 file1.c file2.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. Representative 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, these locations are allocated only up 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 ∈ [--..--]
argv ∈ {{ NULL ; &S_argv }}
S_argv[0] ∈ {{ NULL ; &S_0_S_argv }}
      [1] ∈ {{ NULL ; &S_1_S_argv }}
S_0_S_argv[0..1] ∈ [--..--]
S_1_S_argv[0..1] ∈ [--..--]
```

This is generally not what is wanted, but then again, embedded applications are generally not written against the POSIX interface. If the analyzed 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, consider an example 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);
}
```

Based on the informal specification provided in `usage`, 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);
}
```

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 arguments of the entry point are initialized with a non-deterministic value of their respective type.

Global variables of pointer type contain the non-deterministic superposition of `NULL` and of the addresses of locations allocated by the analyzer. The algorithm to generate those is the same as for formal arguments of the entry point (see previous section). The same options can be used to parameterize the generation of the pointed locations (see next section).

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 option `-context-width`.

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

For an array type, non-aliasing subtrees of values are generated for the first few cells of the array. All remaining cells are made to contain a non-deterministic superposition of the first ones. The number of initial cells for which non-aliasing subtrees are generated is also decided by the value of option `-context-width`.

Example: with the default value `2` for option `-context-width`, the declaration `int *(t[5]);` causes the following array to be allocated:

```
t[0] ∈ {{ NULL ; &S_0_t }}
 [1] ∈ {{ NULL ; &S_1_t }}
 [2..4] ∈ {{ NULL ; &S_0_t ; &S_1_t }}
```

Note that for both arrays of pointers and pointers to pointers, using option `-context-width 1` corresponds to a very strong assumption on the contents of the initial state with respect to aliasing. You should only use the argument `1` for option `-context-width` in special cases, and use at least `2` for generic, relatively representative calling contexts.

Depth of the generated tree

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

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

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

Option `-context-depth` allows to specify this limit. The default value is 2. This number is the depth at which additional variables named `S_...` 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 ∈ [--..--]
 .next ∈ {{ NULL ; &S_next_s }}
S_next_s[0].v ∈ [--..--]
       [0].next ∈ {{ NULL ; &S_next_0_S_next_s }}
       [1].v ∈ [--..--]
       [1].next ∈ {{ NULL ; &S_next_1_S_next_s }}
S_next_0_S_next_s[0].v ∈ [--..--]
...
S_next_0_S_next_0_S_next_s[0].v ∈ [--..--]
                        [0].next ∈
   {{ garbled mix of &{WELL_next_0_S_next_0_S_next_0_S_next_s} (origin: Well) }}
...
```

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 ∈ [--..--]
 .next ∈ {{ NULL ; &S_next_s }}
S_next_s[0].v ∈ [--..--]
       [0].next ∈ {{ NULL ; &S_next_0_S_next_s }}
S_next_0_S_next_s[0].v ∈ [--..--]
               [0].next ∈
               {{ garbled mix of &{WELL_next_0_S_next_0_S_next_s}
                (origin: Well) }}
WELL_next_0_S_next_0_S_next_s[bits 0 to 34359738367] ∈
                          {{ garbled mix of &{WELL_next_0_S_next_0_S_next_s}
                           (origin: Well) }}
```

The possibility of invalid pointers

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 continuations for the

list. Also, the `context-width` parameter for the generated memory areas has to be understood as controlling the maximum width for which the analyzer assumes the pointed area may be valid. However, in order to assume as little as possible on the calling context for the function, the analyzer also considers the possibility that any part of the pointed area might *not* be valid. Thus, by default, any attempt to access such an area results in an alarm signaling that the access may have been invalid.[1]

The option `-context-valid-pointers` causes those pointers to be assumed to be valid (and `NULL` therefore to be omitted from the possible values) at depths that are less than the context width. It also 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 ∈ [--..--]
 .next ∈ {{ &S_next_s }}
S_next_s[0].v ∈ [--..--]
       [0].next ∈ {{ &S_next_0_S_next_s }}
S_next_0_S_next_s[0].v ∈ [--..--]
               [0].next ∈ {0}
```

5.2.5 State of the IEEE 754 environment

The IEEE 754 standard specifies a number of functioning modes for the hardware that computes floating-point operations. These modes determine which arithmetic exceptions can be produced and rounding direction for operations that are not exact.

At this time, obtention of an infinite or NaN as result of a floating-point operation is always treated as an unwanted error. Consequently, FPU modes related to arithmetic exceptions are irrelevant for the analysis.

The value analysis currently offers two modelizations of the floating-point hardware's rounding mode. The possibilities to analyze floating-point computations in C programs with the value analysis are as follows:

Compilers that transform floating-point computations as if floating-point addition or multiplication were associative, or transform a division into a multiplication by the inverse when the inverse cannot be represented exactly as a floating-point number, are not supported by the value analysis.

Programs that use only `double` and `float` floating-point types, do not change the default (nearest-even) floating-point rounding mode and are compiled with compilers that neither generate `fmadd` instructions nor generate extended precision computations (historical x87 instruction set) can be analyzed with the default modelization of floating-point rounding.

Programs that use only `double` and `float` floating-point types, but either do not stay during the whole execution in the "nearest" floating-point rounding mode, or are compiled with a compiler that may silently insert `fmadd` instructions when only multiplications and additions appear in the source code, or are compiled for the x87 FPU, can be analyzed correctly by using the `-all-rounding-modes` analysis option.

The `-all-rounding-modes` option forces the analyzer to take into account more behaviors. The results obtained may therefore seem less precise with this option. Still, the user must resist

[1] For technical reasons, this also means that it is not possible to reduce the values of those areas through user-provided assertions (section 7.1.2).

the temptation to analyze eir program without this option when ey is not in the conditions that allow it, because the analysis may then be incorrect (the analysis may fail to predict some behaviors of the analyzed program, and for instance fail to detect some possible run-time errors).

5.2.6 Setting 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 your target platform is not listed, please contact the developers. An auto-detection program can be provided in order to check the hypotheses mentioned in section 4.5.1, 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.

5.2.7 Parameterizing the modelization of the C language

The following options are more accurately described as pertaining to Frama-C's modelization of the C language than as a description of the target platform, but of course the distinction is not always clear-cut. The important thing to remember is that these options, like the previous one, are dangerous options that should be used, or omitted, 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.

Overflow in array accesses

The value analysis assumes that when an array access occurs in the analyzed program, the intention is that the accessed address should be inside the array. If it can not determine that this is the case, it emits an `out of bounds index` alarm. This leads to an alarm on the following example:

```
1 int t[10][10];
2
3 int main() {
```

```
4    return t[0][12];
5  }
```

The value analysis assumes that writing `t[0][...]`, the programmer intended the memory access to be inside `t[0]`. Consequently, it emits an alarm:

```
array_array.c:4: ... accessing out of bounds index {12}. assert 12 < 10;
```

The option `-unsafe-arrays` tells the value analysis to warn only if the address as computed using its modelization of pointer arithmetics is invalid. With the option, the value analysis does not warn about line 4 and assumes that the programmer was referring to the cell `t[1][2]`.

The default behavior is stricter than necessary but often produces more readable and useful alarms. To reiterate, in this default behavior the value analysis gets hints from the syntactic shape of the program. Even in the default mode, it does not warn for the following code.

```
4    int *p=&t[0][12];
5    return *p;
6  }
```

5.2.8 Dealing with library functions

When the value analysis plug-in encounters a call to a library function, it may need to make assumptions about the effects of this call. The behavior of library functions can be described precisely by writing a contract for the function (chapter 7), especially using an assigns clause (section 7.2).

If no ACSL contract is provided for the function, the analysis uses the type of the function as its source of information for making informed assumptions. However, no guarantee is made that those assumptions over-approximate the real behavior of the function. The inferred contract should be verified carefully by the user.

5.3 Controlling approximations

Unlike options of section 5.2, which used wrongly may cause the value analysis to produce incorrect results, all the options in this section give the user control over the tradeoff between the quality of the results and the resources taken by the analyzer to provide them. The options described in this section can, so to speak, not be used wrongly. They may cause Frama-C to exhaust the available memory, to take a nearly infinite time, or, in the other direction, to produce results so imprecise that they are useless, but **never** to produce incorrect results.

5.3.1 Treatment of loops

Default treatment of loops

The default treatment of loops by the analyzer may produce results that are too approximate. Tuning the treatment of loops can improve precision.

When encountering a loop, the analyzer tries to compute a state that contains all the concrete states possible 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 5.3.1.

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 user may place an annotation `//@ loop pragma WIDEN_HINTS` $v_1, \ldots, v_n$`,` $e_1, \ldots, e_m$ `;` before a loop to indicate the analyzer should 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 change 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 }
```

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 Frama-C should unroll the loops syntactically `n` times before starting any 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;`. The user should place this annotation in the source code, just before the loop. This annotation causes Frama-C's front-end to unroll the loop `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 `n` to use 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+1` 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 55` in order to be completely unrolled:

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

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 28` 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 whether each of those cells contains 1 or its original value 0. 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.

The option `-slevel-function f:n` tells the analyzer to apply semantic unrolling level `n` to function `f`. This fine-tuning option allows to force the analyzer to invest time precisely analyzing functions that matter, for instance `-slevel-function crucial_initialization:2000`. Oppositely, options `-slevel 100 -slevel-function trifle:0` can be used together to avoid squandering resources over irrelevant parts of the analyzed application. The `-slevel-function` option can be used several times to set the semantic unrolling level of several functions.

Loop invariants

A last technique, complementary to the unrolling ones above, consists in using the loop invariant construct of ACSL. Loop invariants describe some properties that hold at the beginning of each execution of a loop, as explained in the ACSL specification, §2.4.2. They can be used to gain precision when analyzing loops in the following way: when the analysis is about to widen the value of a variable, it intersects the widened domain with the one inferred through the loop invariant. Thus, the invariant can be used to limit the (possibly overly strong) approximation induced by the widening step.

```
1  #define N 10
2
3  void main (int c) {
4    int t[N+5];
5    if (c >= 1 && c < N) {
6      int *p=t;
7      /*@ loop invariant p < &t[N-1]; */
8      while(1) {
9        *(++p) = 1;
10       if(p >= t+c) break;
11     }
12   }
13 }
```

In the example above, without a loop invariant, the pointer `p` is determined to be in the range `&t + [4..60],0%4`. In C terms, this means that `p` points between `&t[0]` and `&t[15]`, inclusive. In particular, the analysis is not able to prove that the access on line 9 is always valid.

Thanks to the loop invariant `p < &t[N-1]`, the variable `p` is instead widened to the range `&t[0]..&t[N-2]`. This means that the access to `t` through `p` at line 7 occurs within bounds[2]. Thus, this technique is complementary to the use of `WIDEN_HINTS`, which only accepts integers — hence not `&t[N-1]`. Compared to the use of `-slevel` or `-ulevel`, the technique above does not unroll the loop; thus the overall results are less precise, but the analysis can be faster.

Notice that, in the fonction above, a more precise loop invariant would be

```
5  /*@ loop invariant p < &t[c-1]; */
```

However, this annotation cannot be effectively used by the value analysis if the value for `c` is imprecise, which is the case in our example. As a consequence, the analysis cannot prove this improved invariant. We have effectively replaced an alarm, the possible out-of-bounds access at line 7, by an invariant, that remains to be proven.

5.3.2 Treatment of functions

Skipping the recording of the results of a function

When a function `f` is called many times, or, in presence of semantic unrolling, if `f` contains large/nested loops, significant time may be spent recording the values taken by the program's variables at each of `f`'s statements. If the user knows that these values will not be useful later, ey can instruct the value analysis not to record these with the `-no-results-function f` option. The `-no-results-function` option can be used several times, to omit the results of several functions. The option `-no-results` can be used to omit all results. The time savings can be important, as in the following example.

[2] And more precisely between `t[1]` and `t[9]`, since `p` is incremented before the affectation to `*p`.

```
1  int t[2000], i;
2
3  void unimportant_function(void)
4  {
5    for(i=0; i<2000; i++)
6      t[i]=i;
7  }
8
9  int main() {
10   unimportant_function();
11
12   /* now the important stuff: */
13   return t[143];
14 }
```

```
time frama-c -val nor.c -slevel 200
```

```
[value] Values for function unimportant_function:
          t[0] ∈ {0}
           [1] ∈ {1}
           ...
           [1999] ∈ {1999}
          i ∈ {2000}
[value] Values for function main:
          t[0] ∈ {0}
           [1] ∈ {1}
           ...
           [1999] ∈ {1999}
          i ∈ {2000}
          __retres ∈ {143}

user    0m3.238s
```

Launched with option `-slevel 2001` so that the value returned by `main` can be computed precisely, the analysis takes 3.2 seconds. Contrast with:

```
time frama-c -val nor.c -slevel 2001 -no-results-function unimportant_function
```

```
[value] Values for function unimportant_function:
          NO INFORMATION
[value] Values for function main:
          Cannot filter: dumping raw memory (including unchanged variables)
          t[0] ∈ {0}
           [1] ∈ {1}
           ...
           [199] ∈ {1999}
          i ∈ {2000}
          __retres ∈ {143}

user    0m0.817s
```

When instructed, with option `-no-results-function unimportant_function`, that the values for the function `unimportant_function` do not need to be kept, the value analysis takes less than a second to produce its results. This shows that in the earlier analysis, most of the time was not spent in the analysis itself but in recording the values of `unimportant_function`.

Note that the function `unimportant_function` was analyzed with as much precision as before, and that the result for `main` is as precise as before. The recording of some results is

omitted, but this does not cause any loss of precision *during* the analysis[3]. The results for `unimportant_function` are, as expected, unavailable. The "outputs" of `main` (the "outputs" computation is described in chapter 6) cannot be computed, because they depend on the outputs of `unimportant_function`, which require the values of that function to be computed. These outputs would automatically be used for displaying only the relevant variables if they were available. Instead, all the program's variables are displayed.

The meaning of option `-no-results-function unimportant_function` is different from that of option `-slevel-function unimportant_function:0`. In the case of the latter, values are computed with less precision for `unimportant_function` but the computed values are kept in memory as usual. All derived analyses (such as outputs) are available, but the values predicted by the analysis are less precise, because `unimportant_function` has been analyzed less precisely.

Using the specification of a function instead of its body

In some cases, one can estimate that the analysis of a given function `f` takes too much time. This can be caused by a very complex function, or because `f` is called many times during the analysis. If the solutions of the previous section are not sufficient, a radical approach consists in replacing the entire body of the function by an ACSL contract, that describes its behavior. See for example section 7.2 for how to specify which values the function reads and writes.

Alternatively, one can leave the body of the function untouched, but still write a contract. Then it is possible to use the option `-val-use-spec f`. This instructs the value analysis to use the specification of `f` instead of its body each time a call to `f` is encountered. This is equivalent to deleting the body of `f` for the value analysis, but not for the other plug-ins of Frama-C, that may study the body of `f` on their own.

Notice that option `-val-use-spec f` loses some guarantees. In particular, the body of `f` is not studied at all by the value analysis. Neither are the functions that `f` calls, unless they are used by some other functions than `f`. Moreover, the value analysis does not attempt to check that `f` conforms to its specification. It is the responsability of the user to verify those facts, for example by studying `f` on its own in a generic context.

5.4 Analysis cutoff values

5.4.1 Cutoff between integer sets and integer intervals

Option `-val-ilevel <n>` is used to indicate the number of elements below which sets of integers should be precisely represented as sets. Above the user-set limit, the sets are represented as intervals with periodicity information, which can cause approximations.

Example:

```
1 int t[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 27, 29, 31};
2
3 main(){
4   return t[Frama_C_interval(0, 11)];
5 }
```

[3]The emission of run-time error alarms and the evaluation of user-provided ACSL properties are done during the analysis and are not influenced by the `-no-results*` options. If you make use of only these functionalities, then you can speed up the analysis with option `-no-results` without detrimental effects.

With the default limit of 8 elements for sets of integers, the analysis of the program shows a correct but approximated result:

```
[value] Values at end of function main:
          __retres ∈ [2..31]
```

Studying the program, the user may decide to improve the precision of the result by adding `-val-ilevel 16` to the commandline. The analysis result then becomes:

```
[value] Values at end of function main:
          __retres ∈ {2; 3; 5; 7; 11; 13; 17; 19; 23; 27; 29; 31}
```

Chapter 6

Inputs, outputs and dependencies

Frama-C can compute and display the inputs (memory locations read from), outputs (memory locations written to), and precise dependencies between outputs and inputs, for each function. These computations use the value analysis for resolving array indices and pointers.

6.1 Dependencies

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

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 SELF)`" means that `y` *might* have been modified, and that if it had, its new value would only depend on `x` and `z`, whereas the absence of such a mention means that `y` has necessarily been overwritten.

The dependencies computed by `-deps` hold if and when the function terminates. The list of variables given for an output `y` contains all variables whose initial values can influence the final value of `y`.

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

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

```
int a, b, c;

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

The dependencies of a function may also be listed as `NO EFFECTS`. This means that the function has no effects when it terminates. There are various ways this can happen; in the example below, all three functions `f`, `g` and `h` get categorized as having no effects. Function `f` does it the intuitive way, by not having effects. Function `g` and `h` get categorized as not having effects in a more subtle way: the only executions of `g` that terminate have no effects. Function `h` never terminates at all, and neither `*(char*)0` nor `x` and `y` need be counted as locations modified on termination.

```
int x, y;

void f(void)
{
  0;
}

void g(int c)
{
  if (c)
  {
    x = 1;
    while (x);
  }
}

void h(void)
{
  x = 1;
  y = 2;
  *(char*)0 = 3;
}
```

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 offers `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.

Variant `-input` omits function parameters from the displayed inputs.
Variant `-input-with-formals` leaves function parameters in the displayed inputs (if they indeed seem to be accessed in the function).

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 provides the imperative outputs `b; d` which is the exact answer.

Variant `-out` includes local variables, and variant `-out-external` omits them.

6.4 Operational inputs

The name "operational inputs of a function" is given to the locations that are read without having been previously written to. These are provided in two flavors: a superset of locations that can be thus read in all terminating executions of the function, and a superset of the locations thus read including non-terminating executions.

Operational inputs can for instance be used to decide which variables to initialize in order to be able to execute the function. Both flavors of operational inputs are displayed with the option `-inout`, as well as a list of locations that are guaranteed to have been over-written when the function terminates; the latter is a by-product of the computation of operational inputs that someone may find useful someday.

```
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 -main op`, is found to have on termination the operational inputs `b; c; p; S_p[0]` for function `op`. Operational inputs for non-terminating executions add `e`, which is read inside an infinite loop, to this list.

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

influences the execution by causing, or not, an illicit memory access), whereas on the other hand, the analyzer guarantees that initializing `a` is unnecessary.

Chapter 7

Annotations

Frama-C's 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

Each time it encounters a precondition, postcondition or user assertion, the analyzer evaluates the truth value of the property in the state it is propagating. 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 it, this means that the property is valid under the hypotheses made by the analyzer.

When a property evaluates to `invalid` for some passages of the analysis, it does not necessarily indicate a problem: the property is false only for the execution paths that the analyzer was considering these times. It is possible that these paths do not to occur for any real execution. The fact that the analyzer is considering these paths may be a consequence of a previous approximation. However, if the property evaluates to `invalid` for *all* passages of the analysis, then the code at that point is either dead or always reached in a state that does not satisfy the property, and the program should be examined more closely.

The analysis log contains property statuses relative to the state currently propagated (this is to help the user understand the conditions in which the property fails to verify). The same property can appear several times in the log, for instance if its truth value is `valid` for some passages of the value analysis and `invalid` for others.[1]

The graphical interface also displays a summary status for each property that encompasses the truth values that have been obtained during all of the analysis.

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 propagated state. In other words, the analyzer relies on the fact that the user will establish the validity of P 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.

```
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 -val -slevel 12 -lib-entry -main f reduction.c
```

The value analysis emits 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 prove is therefore the assertion at line 8.

Case analysis

When using *semantic unrolling* (section 5.3.1), if an assertion is a disjunction, then the reduction of the state by the assertion may be computed independently for each disjunct. This multiplies the number of states to propagate in the same way that analyzing an `if-then-else` construct does. Again, the analyzer keeps the states separate only if the limit (the numerical value passed to option `-slevel`) has not been reached yet in that point of the program.

This treatment often improves the analysis' precision. It can be used with tautological assertions to provide hints to the analyzer, as shown in the following example.

[1] Conversely, as Frama-C's logging mechanism suppresses the printing of identical message, the property is printed only once with each status.

```
1 int main(void)
2 {
3   int x = Frama_C_interval(-10, 10);
4   //@ assert x <= 0 || x >= 0 ;
5   return x * x;
6 }
```

```
frama-c -val -slevel 2 .../share/builtin.c sq.c
```

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

Limitations

Attention should be paid to the following two 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 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 use a precondition to add more aliasing in an initial state generated by option `-lib-entry`, because it would be a generalization, as opposed to a restriction;
- 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 worst case (for instance if the formula is too complicated for the analyzer to exploit), this superset is the same as the original state. In this case, 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 options `-lib-entry -main 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 options `-lib-entry -main 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 cannot represent the set of non-zero integers. 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.

7.1.3 An example: evaluating postconditions

To treat the postconditions of a function, the analysis proceeds as follows. Let us call B the memory state that precedes a call to a function f, and A the state before the post-conditions[2], *i.e* the union of the states on all `return` statements. Postconditions are evaluated in succession. Given a postcondition P, the analyzer computes the truth value $\mathcal{V}$ of P in the state A. Then:

- If $\mathcal{V}$ is `valid`, the analysis continues on the next postcondition, using state A.
- If $\mathcal{V}$ is `unknown`, the postcondition may be invalid for some states $A_I \subset A$. The analyzer attemps to reduce A according to P, as explained in section 7.1.2. The resulting state A_V is used to evaluate the subsequent postconditions. Notice that P does *not* necessarily hold in A_V, as reduction is not guaranteed to be complete.
- if $\mathcal{V}$ is `invalid`, the postcondition is invalid for all the real executions that are captured by B, and reducing by P leads to an empty set of values. Thus, further postconditions are skipped, and the analyzer assumes that the call did not terminate.

  Notice that an `invalid` status can have three origins:

  ▷ The postcondition is invalid, *i.e* it does not accurately reflect what the body of the function does. Either the postcondition or the body (or both!) can actually be incorrect with respect to what the code is intended to do.

  ▷ The call to f never terminates, but the analyzer has not been able to deduce this fact, due to approximations. Thus, the postcondition is evaluated on states A that do not match any real execution.

  ▷ The entire call actually never takes place on a real execution (at least for the states described by B). This is caused by an approximation in the statements that preceded the call to f.

  In all three cases, $\mathcal{V}$ is recorded by Frama-C's kernel, for further consolidation.

7.2 Assigns clauses

An `assigns` clause in the contract of a function indicate which variables may be modified by the 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`. Furthermore, the final value of `p->balance` has been computed only through its initial value, and the value of the parameter `s`.

[2] B and A stand respectively for *before* and *after*.

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

An assigns clause can describe the behavior of functions whose 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 removed on purpose because it did something that was complicated and unimportant for the analysis at hand.

The value analysis uses the assigns clauses provided for library functions. It neither takes advantage of nor tries to check assigns clauses of functions whose implementation is also available, except when option `-val-use-spec` is used (section 5.3.2). The effect of an assigns clause `assigns loc1 \from loc2` on a memory state S are evaluated by the analyzer as follows:

1. the contents of `loc2` in S are evaluated into a value v;
2. v is *generalized* into a value v'. Roughly speaking, scalar values (integer or floating-point) are transformed into the set of all possible scalar values. Pointer values are transformed into pointers on the same memory base, but with a completely imprecise offset.
3. `loc1` is evaluated into a set of locations L;
4. each location $l \in L$ of S is updated so that it contains the values present at l in S, but also v'. This reflects the fact that l *may* have been overwritten.
5. furthemore, if `loc1` and `loc2` are distinct locations in S, the locations in L are updated to contain *exactly* v'. Indeed, in this case, `loc1` is necessarily overwritten.[3]

Notice that the values written in `loc1` are *entirely* determined by `loc2` and S. The value analysis does not attempt to "invent" generic values, and it is very important to write \from clauses that are sufficiently broad to encompass all the real behaviors of the function.

Assigns clauses and derived analyses For the purposes of options `-deps`, `-input`, `-out`, and `-inout`, the assigns clauses provided for library functions are assumed to be accurate descriptions of what the function does. For the latter three options, there is a leap of faith in the reasoning: an assigns clause only specifies *functional* outputs and inputs. As an example, the contract `assigns \nothing;` can be verified for a function that reads global `x` to compute a new value for global `y`, before restoring `y` to its initial value. Looking only at the prototype and contract of such a function, one of the `-input`, `-out`, or `-inout` computation would mistakenly assume that it has empty inputs and outputs. If this is a problem for your intended use of these analyses, please contact the developers.

Postconditions of library functions When evaluating a function contract, the value analysis handles the evaluation of a postcondition in two different ways, depending on whether the function f being evaluated has a source body or just a specification.

[3] A correct assigns clause for a function that either writes `y` into `x`, or leaves `x` unchanged, is `assigns x \from y, x;`.

Functions with a body First, the body of f is evaluated, in the memory state that precedes the call. Then the analyzer evaluates the postconditions and computes their truth values, following the approach outlined in section 7.1.3. The memory state obtained after the evaluation of the body may also be reduced by the postconditions. After the evaluation of each property, its truth value for this call is printed on the analysis log.

Functions with only a specification The analyzer evaluates the assigns ... \from ... clauses of f, as explained at the beginning of this section. This results in a memory state S that corresponds to the one after the analysis of the hypothetical body of f. Then, since postconditions cannot be proven correct without a body, they are *assumed* correct, and the value analysis does not compute their truth values. Accordingly, no truth value is output on the analysis log, except when we suspect the postcondition of being incorrect.

Although the analyzer does *not* use the postconditions to create the final memory state, it interprets them in order to *reduce*. Hence, it becomes possible to constrain a posteriori the very broad state S. A typical contract for a function successor[4] would be

```
1 /*@ assigns \result \from x;
2     ensures \result == x + 1; */
3 int succ(int x);
```

Since postconditions are used solely to *reduce* S, not to create it, a postcondition ensures \result == arg; will cause the evaluation to stop if S does not verify $[\![\backslash\text{result}]\!] \cap [\![\text{arg}]\!] \neq \emptyset$ – in which case the postcondition could not hold. It is thus of paramount importance to write assigns ... \from ... clauses that are correct –hence broad enough. A tricky example is the specification of a function that needs to return the address of a global variable.

```
1 int x;
2
3 void *addr_x(); // Returns (void*)&x;
```

The specification assigns \result \from &x; is unfortunately incorrect, as \from clauses can only contain lvalues. Instead, the following workaround can be used;

```
1 int x;
2 int * const __p_x = &x;
3
4 /*@ assigns \result \from __p_x;
5     ensures \result == &x; */
6 void *addr_x();
```

[4] Ignoring problems related to integer overflow.

Chapter 8

Primitives

It is possible to interact with the analyzer through insertion in the analyzed code of calls to pre-defined functions.

There are three reasons to use primitive functions: emulating standard C library functions, transmitting analysis parameters to the plug-in, and observing results of the analysis that wouldn't otherwise be displayed.

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.

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

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

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 constitute the analysis project.

8.1.3 String manipulation functions

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 constitute 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);
    /* returns either a or b */

void *Frama_C_nondet_ptr(void *a, void *b);
    /* returns either a or b */

int Frama_C_interval(int min, int max);
    /* returns any value in the interval from min to max inclusive */
```

```
float  Frama_C_float_interval(float min, float max);
    /* returns any value in the interval from min to max inclusive */

double Frama_C_double_interval(double min, double max);
    /* returns any value in the interval from min to max inclusive */
```

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

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

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 state that the analyzer is propagating at the time it reaches the call. Thus, these functions might expose some undocumented aspects of the behavior of the analyzer. This is especially visible when they are used together with semantic unrolling (see section 5.3.1). Displayed results may be counter-intuitive to 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 by the analyzer.

8.3.1 Displaying the entire memory state

The current memory state each time the analyzer reaches a given point of the program can be displayed 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. 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

Well, for some value of "frequently"...

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. It also improves precision when evaluating if or switch conditionals. A current drawback of semantic unrolling is that it can only be specified crudely at the function level, whereas syntactic unrolling can be specified loop by loop.

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". Consider the following example:

```
1 int x,y;
```

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

```
2  void main(void)
3  {
4    int *p=NULL;
5    x = *p;
6    y = x / 0;
7  }
```

When analyzing this example, 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:[kernel] warning: accessing out of bounds index [-100..100].
                      assert 0 <= i < 101;
false_al.c:9:[kernel] warning: division by zero: assert y != 0;
```

On line 7, the tool is only capable of detecting 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 preprocessing. It is possible to instruct Frama-C to launch the preprocessing on annotations with the option `-pp-annot`. However, this option still is experimental at this point. In particular, it requires the preprocessor to be GNU `cpp`, the GCC preprocessor[2].

A typical example is as follows.

array.c

```
1  #define N 12
2
3  int t[12];
4
5  /*@ ensures 0 <= \result < N ; */
6  int get_index(void);
7
8  main()
9  {
10   int i = get_index();
11   t[i] = 3;
12 }
```

It is useful to use the pre-processor constant `N` in the post-condition of function `get_index()`. Use the command-line:

```
frama-c -val array.c -pp-annot
```

Note that when using this option, the preprocessing is 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, Jochen Burghardt, Géraud Canet, Loïc Correnson, David Delmas, Florent Kirchner, David Mentré, Benjamin Monate, Anne Pacalet, Julien Signoles provided useful comments in the process of getting the text to where it is today. Speaking of the

[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.

text, I find awful, awkward constructs in it each time a new Frama-C release warrants a new pass on it. On the plus side, I am usually happy with it for a few days after each release, until I notice another blunder was left in. Do not hesitate to report documentation bugs at http://bts.frama-c.com/main_page.php.