D5.7

Minimal contract Hoare-style verification versus abstract interpretation

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D5.7/ 1.0 |
Work package contributing to the deliverable:	WP 5
Due date:	December 2019 – M36
Actual submission date:	13th January 2020

Responsible organisation:	Fraunhofer FOKUS
Editor:	Jens Gerlach
Dissemination level:	PU
Revision:	1.0

Abstract:	Report on using deductive verification compared to abstract interpretation.
Keywords:	Static analysis, deductive verification, minimal contracts, abstract interpretation

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Jens Gerlach (FOKUS)

Disclaimer

Executive Summary

In this report we summarise the experience of the Vessedia project with respect to two methods to formally verify the absence of runtime errors in C software. The first method is *abstract interpretation* which relies on an over approximation of the values that can occur while running the software. The second method consists in writing so-called *minimal contracts* which are formal function contracts that ideally are just strong enough to *deductively verify* the absence of runtime error in components. Our experience with both Vessedia's *Contiki* use case and an external case study conducted by the French cyber security agency ANSSI shows that both approaches are meaningful . Moreover, practical considerations suggest that both approaches are best used in tandem.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Goal of the Document

Nowadays, verification engineers can choose among different static analysis methods for software quality assurance, ranging from a basic level (e.g. compiler warnings), via heuristic-based tools, to mathematically rigorous formal methods. The latter can provide a high degree of confidence, but are often considered as difficult to integrate into the development process.

Some formal static analysis tools, such as Astrée, Polyspace and the EVA plugin of Frama-C use *abstract interpretation* to reliably identify potential cases of undefined behaviour in software that can lead to serious security vulnerabilities and runtime errors. These tools have the advantage to work on large programs. Their principal disadvantage is that they rely on an over-approximation of the behaviour of the analysed program. Thus, they can produce a substantial number of false alarms. This, in turn, decreases the degree of automation as an expert is then required to investigate the alarms and classify them as true or false alarms.

On the opposite, formal analysis tools relying on *deductive verification* (e.g. the WP plugin of Frama-C) can be used to verify complex software properties that go far beyond undefined behaviours. However, these tools rely on additional annotations written by the user that specify the expected behaviour of the program. Some annotations also help to guide the proof process. All of them are expressed using formal specification languages (e.g. ACSL). While this approach sounds very promising, it is common that complex properties can be successfully tackled only for relatively small and well-designed modules. Abstract interpretation and deductive verification can thus be seen as two opposite points of the spectrum of formal methods.

We argue that verifying what we call *minimal contracts* can be seen as a synthesis of both approaches combining their respective advantages, viz. local specifications and global analysis.

Our definition of a *minimal contract* says that it specifies a set of properties of a function that are necessary to verify the absence of runtime errors in this function and its callees (and sometimes also in its callers).

Minimal contracts can be provided at a reasonable effort and can be verified by both abstract interpretation and deductive verification. While the term *minimal contracts* might be new, the idea is not really. Thus, it is not surprising that minimal contracts are already in use outside our verification efforts for Contiki. A recent example is an RTE-free X.509 parser which was specified with ACSL and for which a combination of the EVA and WP plugins of Frama-C was used to verify the absence of runtime errors.

Chapter 2 A simple example of deductive verification

Frama-C is a platform dedicated to source-code analysis of C software. It has a plug-in architecture and can thus be easily extended to different kinds of analyses. The WP plugin of Frama-C allows one to deductively verify that a piece of C code satisfies its specification. This implies, of course, that the user provides a *formal specification* of what the implementation is supposed to do. Frama-C comes with its own specification language ACSL which stands for *ANSI/ISO C Specification Language*.

2.1 Computing the absolute value

We will consider the function that computes the absolute value |x| of an integer x. In order to avoid potential name clashes with the function `abs` in C standard library we use the name `abs_int`.

The mathematical definition of absolute value is very simple

$$|x| = \quad x, \text{ if } 0 \leq x$$

$$|x| = -x, \text{ if } x < 0$$

A straightforward implementation of `abs_int` is shown in Figure 1.

```
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Figure 1: An implementation of the absolute value function

In order to demonstrate that this implementation is correct we have to provide a formal specification. The listing in Figure 2 shows our first attempt for an ACSL specification of `abs_int` that is based on the mathematical definition of the function |x|.

```
/*@
    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Figure 2: A first attempt to formally specify `abs_int`

The first thing to note is that ACSL specifications–or *contracts*–are placed in special C comments which start with `/*@`. Thus, they do not interfere with the executable code. The `ensures` clause in

the specification expresses *postconditions*, that is, properties that should be guaranteed *after* the execution of `abs_int`. The ACSL reserved word `\result` is used to refer to the return value of a C function. Note that we use the usual C operators `==` and `<=` to express equalities and inequalities in the specification, respectively. There is also an additional operator `==>` which expresses logical implication.

2.2 Why can Frama-C/WP not verify such a simple function?

If we can start Frama-C with the following commands

```
frama-c -wp -wp-rte abs_int.c
```

then we obtain output like this

```
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no
preprocessing)
[kernel] Parsing abs_int.c (with preprocessing)
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[rte] annotating function abs_int
[wp] 3 goals scheduled
[wp] [Qed] Goal typed_abs_int_post : Valid
[wp] [Qed] Goal typed_abs_int_post_2 : Valid
[wp] [Alt-Ergo] Goal typed_abs_int_assert_rte_signed_overflow : Unknown (105ms)
[wp] Proved goals:    2 / 3
     Qed:             2
     Alt-Ergo:        0  (unknown: 1)
```

Although the specification and implementation in the listing above (Figure 2) look perfectly right, Frama-C/WP cannot verify that the implementation actually satisfies its specification. The reason becomes clear if we look at some actual return values of `abs_int` in the following table

x	abs_int (x)	Remark
0	0	✓
1	1	✓
10	10	✓
2147483647	2147483647	✓
-1	-1	✓
-10	-10	✓
-2147483648	**-2147483648**	Oops!

Table 1: Some test results for `abs_int`

Figure 3 shows the test code whose output is listed in the table above.

```
#include <stdio.h>
#include <limits.h>

extern int abs_int(int);

void print_abs(int x)
{
  printf("%12d\t\t%12d\n", x, abs_int(x));
}

int main()
{
  printf("\n");
  print_abs(0);

  printf("\n");
  print_abs(1);
  print_abs(10);
  print_abs(INT_MAX);

  printf("\n");
  print_abs(-1);
  print_abs(-10);
  print_abs(INT_MIN);
}
```

Figure 3: Some simple test cases for `abs_int`

The offending value is in the last line of the table which basically states that `abs_int(INT_MIN)` equals the *negative value* `INT_MIN` whereas it should equal `-INT_MIN`. The problem is that the type **`int`** only presents a finite subset of the (mathematical) integers. Many computers use a two's-complement representation of integers which covers the range $[-2^{31}, 2^{31}-1]$ on a 32-bit machine. On such a machine the value `-INT_MIN` cannot be represented by a value of type **`int`**.

In a specification, Frama-C/WP interprets integers as mathematical entities. Consequently, there is no such thing as an *arithmetic overflow* when adding or multiplying them in a function contract. In other words, Frama-C/WP is perfectly right not being able to verify that `abs_int` satisfies our contract. Indeed, the implementation does not respect the given specification.

2.3 Strengthening the function contract

It is of course well known that the operation `-x` can overflow and it is the fact that Frama-C/WP can detect such overflows that helps to prevent incorrect verification results. The GNU Standard C Library clearly states that the absolute value of `INT_MIN` is undefined.[1] Under *macOS* the manual page of `abs` mentions under the field of *Bugs*:

```
The absolute value of the most negative integer remains negative.
```

In other words, our formal specification should exclude the value `INT_MIN` from the set of admissible value to which `abs_int` can be applied. In ACSL, we can use the *requires* clause to

[1] See http://www.gnu.org/software/libc/manual/html_node/Absolute-Value.html

express preconditions of a function. Figure 4 shows an extended contract of `abs_int` that takes the limitations of the type `int` into account.

```
/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Figure 4: Taking integer overflows into account

Frama-C/WP is capable to verify that the implementation of `abs_int` satisfies the improved specification.

There is an important lesson that can be learned here:

> Sometimes developers provide source code and imagine that a tool like Frama-C/WP can verify the correctness of their implementation. In order to fulfil its task, however, Frama-C/WP needs an ACSL specification. Such a specification—which must be based on a reasonably precise description of the admissible inputs and expected behaviour—has to come from the requirements of the software and is not magically discovered from the source code by Frama-C/WP. The code does what it does. In order to verify that the code does what someone expects, these expectations must be clearly expressed, that is, they must be formally specified.

Of course, it might not always be the goal to verify the complete functionality of a piece of software. Sometimes, it is enough to ensure that individual software components cause no run- time errors, that is, arithmetic overflows or invalid pointer accesses. Frama-C/WP can also be used in this situation. This is were *minimal contracts* come into play. Before, however, we discuss the treatment of *side effects*.

2.4 Dealing with side effects

The next listing in Figure 5 shows an implementation of `abs_int` that writes as a side effect the argument `x` to a global variable `a`. A natural question is to ask whether this implementation with a side effect also satisfies the specification.

```
#include <limits.h>

extern int a;

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  a = x; // Is this side effect covered by the specification?
  return (x >= 0) ? x : -x;
}
```

Figure 5: An implementation with side effects

Before we answer this question we consider various uses for side effects. There are of course legitimate uses for side effects. The assignment to a memory location outside the scope of the function might be meaningful because an error condition is reported or because some data are logged as in the listing in Figure 6.

```
#include <limits.h>

extern void logging(int);

/*@
    requires x > INT_MIN;

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  logging(x);
  return (x >= 0) ? x : -x;
}
```

Figure 6: Calling a logging function from abs_int

If Frama-C/WP attempts to verify this code, then it issues the following warning:

```
Neither code nor specification for function logging,
generating default assigns from the prototype
```

Thus, it points out that the called function logging should have a proper specification that clearly indicates its side effects. There are, on the other hand, also good reasons to minimize or even forbid side effects:

- Imagine a malicious password checking function that writes the password to a global variable.
- Another reason is that side effects can make it harder to understand what the real consequences of a function call are. In particular, one must be concerned about unintended consequences that are caused by side effects. The norm IEC 61508 therefore requests in the context of software module testing and integration testing:

  > To show that all software modules, elements and subsystems interact correctly to perform their intended function and do not perform unintended functions. [§7.4.7.2,§7.7.2.9 in [1]]

Of course, it is quite difficult to ensure by testing alone that something does *not* happen and this is one of the reasons static code analysis is important.

To come back to our question about whether our first example with side effects satisfies the contract it is important to understand that Frama-C/WP verifies indeed that the implementation shown there satisfies the specification. If one wishes to forbid that a function changes global variables one can use an `assigns \nothing` clause as shown in Figure 7. Frama-C/WP will then point out that this implementation pre- vents the verification of the assigns clause.

```
extern int a;

/*@
    requires x > INT_MIN;

    assigns \nothing; // forbid any side effects

    ensures 0 <= x ==> \result ==  x;
    ensures 0 >  x ==> \result == -x;
*/
int abs_int(int x)
{
  a = x; // now illegal
  return (x >= 0) ? x : -x;
}
```

Figure 7: Specifying the absence of side effects

2.5 A minimal contract

Figure 8 shows an example of what we call a *minimal contract* for `abs_int`.

```
#include <limits.h>

/*@
    requires x > INT_MIN;

    assigns \nothing;
*/
int abs_int(int x)
{
  return (x >= 0) ? x : -x;
}
```

Figure 8: Minimal contract to ensure the absence of runtime errors in `abs_int`

The contract is just strong enough such that its verification ensures that no undefined behaviour such as integer overflows or buffer overflows can occurs. It also makes a clear statement about its potential side effects. A drawback of this smaller contract is that it does not really say something about the functional behaviour of the code.

Chapter 3 Using Frama-C in Contiki and beyond

Various forms of static and dynamic analyses have been conducted to verify parts of the Contiki operating system. The results of these analyses are described in [2]. With respect to sound static analyses both the deductive verification plug-in Frama-C/WP and the abstract interpretation plug-in Frama-C/EVA have been used.

3.1 Deductive verification for Contiki

Frama-C/WP was primarily applied to verify minimal contracts of Contiki components both by INRIA and Fraunhofer FOKUS in order to ensure that no runtime errors can occur. Among others the following functions from the Contiki OS support library were verified

- `os/lib/crc16.c`
- `os/lib/ifft.c`
- `os/lib/ringbuf.c`
- `os/lib/ringbufindex.c`

Another important example for the application of minimal contracts is the verification of Contiki's AES-CCM block cipher mode of operation.

However, proper functional properties have also been investigated, in particular for the `list_t` module of Contiki operations for adding and removing elements have been verified which is a considerable achievement.

3.2 Abstract interpretation for Contiki

Interestingly, Frama-C's EVA plug-in has was also applied to the above-mentioned AES-CCM module. This allowed to compare both minimal contracts and abstract interpretation on the same piece of software. The verifiers observed that "while the minimal contracts where indeed (relatively) easy to write for this module, the EVA analysis remained more efficient here" (see 4.3.3 of [3]).

In contrast, a couple of difficulties occurred when applying the Frama-C/EVA plug-in on the whole Contiki operating system. In principle, an abstract interpretation tool such as the EVA plugin is well-suited to analyse a large code base. One issue that was clear from the beginning is the fact that due to the high degree of configurability it is not meaningful to speak of analysing the whole Contiki OS. Rather, one can only specific *instances*, that is particular *configurations* of Contiki for particular hardware platforms. This also includes several applications (defined as processes) on top of it. All these components are linked together to one specific Contiki executable during compilation. Section 4.4 of Deliverable D5.2 describes in more detail the problems that arose during various analysis attempts. In particular, it became clear that some complex core modules such as `list_t` and the `MEMB` allocation module prevented achieving tangible results with Frama-C/EVA. Another reason is the *recursive* nature of the Contiki scheduler. Handling of recursion is currently not well supported in the EVA plug-in.

3.3 The X509 case study

It is interesting to have look at the efforts to implement a parser of X.509 certificates that is guaranteed to be free of runtime errors (RTE). This was conducted by researchers at ANSSI, the National Cybersecurity Agency of France.
The stated goal was achieved by

- starting with a clean state and not relying on existing implementations

- using a combination of the Frama-C/EVA and Frama-C/WP plug-ins
- employing best coding practices for the planned static analysis
- relying on a large representative test suite

The primary plug-in used in this case study was Frama-C/EVA. More advanced properties were specified with ACSL and with Frama-C/WP. Basically, the developers wrote what they call *basic function contracts* that dealt with buffer-related information such as validity and length. In other words, these contracts are what we call *minimal contracts*.

Of the overall 9000 lines, 5000 were C code while 1200 lines were (ACSL) annotations. We agree with the authors that this this *seems like a reasonable investment* for guaranteeing the complete absence of runtime errors. While Frama-C was their verification platform of choice, they point out that development rules are not tailored specifically to Frama-C.

Chapter 4 Summary and Conclusion

Both abstract interpretation and deductive verification have been essential in verifying security properties of Contiki. We have applied deductive verification with minimal contracts in a proper sense that is writing small contracts that capture only basic properties directly related to possible runtime errors. This approach, however, did not extend to the verification of the `list_t` data type. Due to the inherent complexity of these data structure minimal contracts turned out to be fully functional contracts. Nevertheless, we were able to verify these contracts, too.

The apparent advantage that abstract interpretation tools such as Frama-C/EVA can be applied for fairly large programs has been, in fact, a challenge because of complexities of the Contiki's configuration space and some very difficult to handle code artefacts of Contiki.

A possibly even more important insight comes from comparing our experience with applying Frama-C with the X.509 case study conducted by ANSSI. The rigorous bottom-up approach of the ANSSI team enabled an effective combination of *abstract interpretation* and *deductively verified minimal contracts* that culminated in an X.509 parser that is verified to be free of runtime errors. With Contiki such an approach would be possible too (at least for some well-defined core). However, it would require a very different development process where the needs of security and verification are put centre stage.

Chapter 5 List of Abbreviations

Abbreviation	Translation
RTE	Runtime Errors
WP	Work Package

Chapter 6 Bibliography

[1] IEC 61508-2010, International Electrotechnical Commission, „Functional safety of electrical/electronic/programmable electronic safety-related systems“.

[2] Jochen Burghardt, Robert Clausecker, Jens Gerlach, Annotating IoT-Software With Minimal Contracts, Preliminary Results, Internal report of the VESSEDIA project, Fraunhofer FOKUS, 2017

[3] Allan Blanchard, Inria’s use case final report, Deliverable D5.2, VESSEDIA, 2019

[4] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. Journey to a RTE-free X.509 parser, Symposium sur la Sécurité des Technologies de l’Information et des Communications (SSTIC), 2019