A Lesson on Verification of IoT Software with Frama-C

Allan Blanchard
Inria Lille – Nord Europe
Villeneuve d'Ascq, France
Allan.Blanchard@inria.fr

Nikolai Kosmatov
CEA, List
Software Reliability and Security Lab
Gif-sur-Yvette, France
Nikolai.Kosmatov@cea.fr

Frédéric Loulergue
SICCS
Northern Arizona University
Flasgstaff, USA
Frederic.Loulergue@nau.edu

***Abstract*—This paper is a tutorial introduction to FRAMA-C, a framework for analysis and verification of C programs. We present value analysis, deductive verification and runtime verification of software using FRAMA-C and in particular its EVA, WP, and E-ACSL plugins. These techniques are illustrated on examples coming from real-life verification case studies for different modules of Contiki, a lightweight operating system for the Internet of Things.**

***Keywords*—software verification; C programs; value analysis; deductive verification; runtime verification; Contiki.**

I. INTRODUCTION

Among distributed systems, connected devices and services, also referred to as the Internet of Things (IoT), have proliferated very quickly in the past years. There are now billions of interconnected devices, and this number is rapidly growing. It is anticipated that by 2021, about 46 billions of devices will be in use.

Some of these devices are in service in security critical domains, and even in domains that are not necessarily critical, for example privacy issues may arise with devices collecting and transmitting a lot of personal information. Moreover, insufficiently secured devices may become a target for massive distributed denial of service attacks. This raises important security challenges. It is natural to expect that formal methods — that have been successfully used for years in highly critical domains — can now help to bring security into the IoT field.

While the correctness of an implementation with respect to a formal functional specification provides the strongest form of guarantee, it can be very costly to achieve. In practice it is therefore more common to rely on a combination of different verification techniques to achieve an appropriate degree of guarantee:

- static analysis to guarantee the absence of runtime errors,
- deductive verification for functional correctness,
- dynamic (runtime) verification for parts of code that cannot be proved using deductive verification.

This tutorial paper illustrates these three approaches using FRAMA-C.

FRAMA-C[1] [1] is a source code analysis platform that aims at conducting verification of industrial-size programs written in ISO C99 source code. FRAMA-C fully supports combinations of different approaches, by providing its users with a collection of plugins for static and dynamic analyses of safety- and security-critical software. Moreover, collaborative verification across cooperating plugins is enabled by their integration on top of a shared kernel, and their compliance to a common specification language: ACSL [2].

Recently, FRAMA-C has been successfully applied to the verification of software [3]–[5] in the context of the Internet of Things, more specifically, the verification of several modules of Contiki [6], an open-source operating system for the IoT. The purpose of this tutorial is to present several verification techniques using FRAMA-C and illustrate them on C code examples extracted from real-life case studies on verification of Contiki.

This tutorial paper is organized as follows. In Section II, we present how to check the absence of runtime errors using EVA, an automatic value analysis plugin of FRAMA-C. Verifying a library rather than a whole program, or verifying more complex (functional) properties, can require deductive verification with WP, a deductive verification plugin of FRAMA-C, that is presented in Section III. WP requires a formal specification in ACSL, so Section III also introduces the ACSL specification language. However, it is not always possible to formally specify and prove a whole software project with WP. Sometimes only some (most critical) parts are formally proved. Runtime verification with the E-ACSL2C plugin of FRAMA-C is possible even for selected modules and functions, and even for a partial specification. It can also help to check the specification on a few tests before trying to prove it. The E-ACSL2C plugin is presented in Section IV. Finally, we conclude and recommend further reading in Section V.

II. ABSENCE OF RUNTIME ERRORS USING EVA

A. EVA *on Simple Examples*

Value analysis is a program analysis technique that computes a set of possible values for every program variable at each program point. It is based on the *abstract interpretation* technique proposed by Cousot and Cousot in the 1970's [7]. Its main idea is to compute an abstract view of values of variables in the form of *abstract domains*. For example, a usual abstract view for a number value is an interval.

[1]See https://frama-c.com

```
1  int f ( int a ) {
2    int x, y;
3    int sum, result;
4    if(a == 0){
5      x = 0; y = 0;
6    }else{
7      x = 5; y = 5;
8    }
9    sum = x + y;      // sum can be 0
10   result = 10/sum; // risk of division by 0
11   return result;
12 }
```

(a) Program with a real error

```
1  int f ( int a ) {
2    int x, y;
3    int sum, result;
4    if(a == 0){
5      x = 0; y = 5;
6    }else{
7      x = 5; y = 0;
8    }
9    sum = x + y;  // sum cannot be 0
10   result = 10/sum; // no div. by 0
11   return result;
12 }
```

(b) Program with a false alarm

Fig. 1. Two toy examples where EVA reports an alarm of division by 0 at line 10.

Value analysis can be very useful to detect potential runtime errors or prove their absence. Typical examples include invalid pointers, invalid array indices, arithmetic overflows or division by zero. It can also help to prove other properties for which domain-based reasoning can be efficient.

Since the FRAMA-C Aluminium release, FRAMA-C offers a new value analysis plugin EVA (Evolved Value Analysis) [8]. It implements value analysis as a generic extendable analysis parameterized by cooperating abstract domains. Different, highly optimized domains are used to represent integers, floating-point numbers and pointers. EVA is strongly integrated into the FRAMA-C ecosystem and offers a basis for many other derived plugins that reuse the results of EVA (see [1]).

Figure 1a shows an example of a program where EVA detects an alarm of division by zero. Indeed, after the then branch, both `x` and `y` are zero, so their sum is zero as well. To run FRAMA-C/EVA on this program (in file `div1.c`), the user can type the following command

```
frama-c -val div1.c -main f
```

in order to see the results in the terminal, or

```
frama-c-gui -val div1.c -main f
```

in order to explore the results in the Graphic User Interface of FRAMA-C. The option `-main f` indicates that the entry point is function `f` (rather than the usual function `main`, not given in this toy example). EVA computes the sets of possible values of variables at each program point. After the conditional statement, the domains of `x` (and similarly for that of `y`) computed inside the `then` and `else` branches are joined into a more general domain containing both cases. That is why the domain of `x` and that of `y` at line 9 are $\{0, 5\}$. Thus, after the conditional statement, the domain of `sum` is computed as $\{0, 5, 10\}$ after the statement on line 9. Since 0 is identified as a possible value of `sum` before the division on line 10, EVA detects a potential risk of division by 0 and reports an *alarm*. It is reported by adding an ACSL assertion

```
assert sum ≠ 0;
```

before line 10 in the GUI. (For convenience of the reader, in the examples in this paper, some ACSL symbols (like `\forall`, `\exists`, `integer`, `<=`, `!=`, `&&`, etc.) are pretty-printed in the corresponding mathematical notation (resp., as $\forall$, $\exists$, $\mathbb{Z}$, $\leq$, $\neq$, $\wedge$, etc.).)

This assertion indicates that the risk of division by zero at line 10 cannot be excluded, and the given property should be verified (manually or by other means) to exclude this risk. In this case, this property cannot be proved, and the risk is actually real since we have a division by zero if the input value `a` is zero and the then branch is taken. Notice that the computed domains are over-approximated: in fact, the exact domain of `sum` after line 9 is $\{0, 10\}$, but it is computed as $\{0, 5, 10\}$ even if value 5 cannot be taken.

B. False Alarms and Parameterization

Over-approximation can lead to detecting *false alarms*, that is, situations where a potential error is reported while the error can never occur in practice. Consider the program in Figure 1b. On this program, the analysis by EVA will report a risk of division by zero on line 10 as well. Here again, after the conditional statement, the domains of `x` (and similarly for that of `y`) computed inside the then and else branches are joined into a more general domain containing both cases. Thus, the domain of `x` and that of `y` at line 9 are $\{0, 5\}$, and the domain of `sum` after line 9 is computed as $\{0, 5, 10\}$. Since it contains 0, EVA reports a risk of division by zero. Here, the exact domain of `sum` after line 9 is $\{5\}$, the other values (0 and 10) being due to over-approximation.

Several options are available to control the precision of the analysis. Let us illustrate one very useful option, called `-slevel`. It allows to maintain several abstract states in parallel along several execution traces without joining them immediately into a unique (but over-approximated) state. In other words, this option is an instance of *trace partitioning* [9]. The precision of the analysis can be improved by giving an additional option `-slevel n` that makes the analysis keep up to n states in parallel before joining them. This increase in precision comes at the cost of making the analysis potentially slower. It is possible to set this option for a specific loop, specific function or for the whole program.

Suppose EVA is run on the program of Figure 1b (in file div2.c) with an additional option `-slevel 2`:

```
1  #include "__fc_builtin.h"
2  int A, B;
3  int root(int N){
4    int R = 0;
5    while(((R+1)*(R+1)) ≤ N) {
6      R = R + 1;
7    }
8    return R;
9  }
10
11 void main(void)
12 {
13   A = Frama_C_interval(0,64);
14   B = root(A);
15 }
```

Fig. 2. A program computing an integer square root.

```
frama-c-gui -val div2.c -main f -slevel 2
```

In this case, EVA keeps precise domains for `x` and `y` after the conditional statement, and the domain of `sum` after line 9 in both cases is computed as $\{5\}$. Now, EVA proves that the risk of error is excluded, and does not report a false alarm.

Suppose now that some of the variables are not initialized in one of the branches, say, the assignment of `y` in line 5 in Figure 1b is deleted. In this case, EVA detects that variable `y` (that is, the value at address `&y`) can be read on line 9 without being necessarily initialized. This alarm is reported by adding an assertion before line 9

```
assert \initialized(&y);
```

In this case, this is a real error: providing the `-slevel` option will not remove this alarm.

Consider now the example of Figure 2. The function `root` computes the integer square root of a given integer $N \geq 0$, that is, an integer $r \geq 0$ such that $r^2 \leq N < (r+1)^2$. This function is called for the integer value `A` supposed to be between 0 and 64 (this assumption is made by the built-in function call `Frama_C_interval(0,64)` available thanks to the included header file on line 1). Application of EVA with default options by the command

```
frama-c-gui -val sqrt.c
```

on a 32-bit architecture produces two alarms before line 5:

```
assert R + 1 ≤  2147483647;
assert (R + 1)*(R + 1) ≤ 2147483647;
```

These alarms report a potential risk of arithmetic overflows in expressions `R + 1` and `(R + 1)*(R + 1)`. Indeed, because of the loop and over-approximation, the computed domain for `R` on line 5 is the interval $0, ..., 2147483647$, that is, all positive integers up to the maximal integer value (2147483647 on a 32-bit architecture). Hence, EVA cannot exclude the risk of overflows. Using EVA with an additional option `-slevel 8`, that makes the analysis keep up to 8 states[2] in parallel, improves the precision and computes the precise values of R after each iteration separately. It allows EVA to exclude the risk of overflows.

EVA can also detect potentially invalid pointers. For example, if `p` is a pointer and the pointer access in a statement `*p=10;` is not proved valid, it will report an alarm

```
assert \valid(p);
```

C. A Real-Life Example

Let us now illustrate the proof of absence of runtime errors on a real-life example of IoT software. Figure 3 shows a (partially simplified) code of the AES (Advanced Encryption Standard) module of Contiki and tests the encryption for concrete key and message. It installs an AES key and runs the encryption function to encrypt a given message **`data`** (cf. lines 44–51). Running EVA on this program allows to prove that all variables are initialized before being read, all memory accesses are valid and there are no arithmetic overflows. During the tutorial, we consider other examples with slight erroneous modifications of this program to illustrate how runtime errors can be detected by EVA. Let us briefly mention some of them. If the initialization of the arrays on lines 46 or 47 is omitted (or is not complete), EVA will report an alarm for non-initialized variables. If the size of the arrays on lines 46 or 47 is less than required (i.e. less than 16 in this example) EVA will report an alarm for incorrect indices. Finally, if the loop condition is wrong (say, `i≤11` on line 22) EVA will also detect a potential out-of-bounds array access.

III. DEDUCTIVE VERIFICATION USING WP

As previously pointed out, the EVA plugin can formally verify the absence of runtime errors during a whole program analysis, or detect potential errors. However, in order to formally verify a library for any call context (respecting the library requirements) or to verify more complex properties (such as functional properties), we generally need a more precise way to specify the requirements and the behavior of each function. EVA can be unable to handle complex specifications and to prove functional properties. In FRAMA-C, such properties can be specified in the ACSL language and proved using the deductive verification plugin WP.

A. ACSL *Specification Language*

ACSL is the specification language of the FRAMA-C platform. It is based on the notion of contract, like in JML or Eiffel for example, and allows users to specify functional properties of their programs. The properties that can be expressed in ACSL are roughly first order logic formulae composed of pure C expressions. ACSL also brings pure logic types (mathematical integers, reals, sets, lists, ...) and some built-in predicates, which can be useful for the specification of C-specific properties, typically, related to pointers such as pointer validity, memory separation, etc. Of course, it is also possible for users to write their own predicates and logic functions.

[2]a smaller value can be sufficient on some versions of FRAMA-C.

```
#define AES_128_BLOCK_SIZE 16
#define AES_128_KEY_LENGTH 16
typedef unsigned char uint8_t;

static const uint8_t sbox[256] =
  { /* initialization for 256 elements */ };

static uint8_t round_keys[11][AES_128_KEY_LENGTH];

static uint8_t galois_mul2(uint8_t value) {
  uint8_t xor_val = (value >> 7) * 0x1b;
  return ((value << 1) ^ xor_val);
}

static void aes_128_set_key(const uint8_t *key) {
  uint8_t i, j, rcon;
  rcon = 0x01;

  for(i = 0; i < AES_128_KEY_LENGTH; i++) {
    round_keys[0][i] = key[i];
  }
  for(i = 1; i ≤ 10; i++) {
    round_keys[i][0] = sbox[round_keys[i - 1][13]]
      ^ round_keys[i - 1][0] ^ rcon;
    round_keys[i][1] = sbox[round_keys[i - 1][14]]
      ^ round_keys[i - 1][1];
    round_keys[i][2] = sbox[round_keys[i - 1][15]]
      ^ round_keys[i - 1][2];
    round_keys[i][3] = sbox[round_keys[i - 1][12]]
      ^ round_keys[i - 1][3];

    for(j = 4; j < AES_128_BLOCK_SIZE; j++) {
      round_keys[i][j] = round_keys[i - 1][j]
        ^ round_keys[i][j - 4];
    }
    rcon = galois_mul2(rcon);
  }
}

static void aes_128_encrypt(uint8_t *state) {
  /* encryption code */
}

static void test_aes_128()
{
  uint8_t key[16] = { /* initialization */ };
  uint8_t data[16] = { /* initialization */ };

  aes_128_set_key(key);
  aes_128_encrypt(data);
}
```

Fig. 3. A test of the AES encryption module of Contiki (simplified).

```
1  /*@
2    requires \valid(a) ∧ \valid(b);
3    requires \separated(a,b);
4    assigns *a, *b;
5    ensures *a == \old(*b) ∧ *b == \old(*a);
6  */
7  void swap(int *a, int *b){
8    int tmp = *a ; *a = *b ; *b = tmp ;
9  }
10
11 int main(){
12   int x = 2;
13   int y = 4;
14   swap(&x, &y);
15   //@ assert x == 4 ∧ y = 2 ;
16 }
```

Fig. 4. Example of an ACSL specification of the `swap` function

The *contract* of a function is composed of two main parts: precondition and postcondition. A *precondition* clause is introduced by the **`requires`** keyword and describes the expected state of the system before calling the function. A *postcondition* clause is introduced by the **`ensures`** keyword and expresses how the state of the program is modified (or not) by the function, and the properties of the result. Several clauses on the same type can be written one after another, in that case they are equivalent to a single clause expressing the conjunction of the corresponding properties. A particular class of postconditions is introduced using the **`assigns`** keyword. It allows to specify which (non-local) memory locations can be modified by the function (and, consequently, which cannot be modified).

Function contracts, as well as other ACSL specifications, are added into a C file as *code annotations*, i.e. special comments started with "`/*@`" and closed with "`*/`". Figure 4 illustrates the use of ACSL to specify a swap function. The first **`requires`** clause (line 2) indicates that `a` and `b` must be valid pointers, that is to say, pointers to memory regions that can be accessed both in reading and writing. The second **`requires`** clause (line 3) expresses that the pointers must point to two non-overlapping memory regions. The **`assigns`** clause (line 4) lists the memory locations that can be modified — here, the locations referred to by `a` and `b`. Finally, the **`ensures`** clause (line 5) indicates that after the execution of the function, `*a` will be equal to the old value of `*b` (that is, its value before the execution of the function) and *vice-versa*.

B. Some Simple Examples

Once the function contracts have been expressed, deductive verification with FRAMA-C/WP can be used to prove that:

- each function respects its contract,
- each call to a function satisfies the precondition before the call.

It relies on the weakest-precondition calculus [10]. In our example, we can run WP using the command:

```
frama-c main.c -wp
```

that will automatically prove that the `swap` function correctly implements the specified contract, as well as that the call to the function on line 14 in function `main` respects the precondition of the function.

The ACSL keyword **`assert`** introduces an *assertion*, i.e. a property that must be checked at a particular program point. Here, it is used to deduce the fact that the values are indeed swapped after the call to the function.

However, that does not prove that the program cannot fail at runtime, since we only proved that the program respects the specification we have written. We still have to prove that

```
1  /*@
2    ensures \result ≥ a ∧ \result ≥ b ;
3  */
4  int max(int a, int b){
5    return (a ≥ b) ? a : b ;
6  }
7
8  extern int x ;
9
10 int main(){
11   x = 3;
12   int r = max(4, 2);
13   //@ assert r == 4 ;
14   //@ assert x == 3 ;
15 }
```

Fig. 5. Imprecise specification for the `max` function

there are no runtime errors. We can perform this additional verification using a command line:

```
frama-c main.c -wp -then -wp-rte -wp
```

That will first prove that the program respects the specification, and then generate the assertions to prevent runtime errors and prove that they are satisfied. Note that we could directly verify both of these aspects at the same time but it is a better practice to first focus on the contracts that generally do not need the assertions about runtime errors to be proven. Thus, adding them directly could sometimes "pollute" the proof obligations provided to the automatic solvers (that would receive both the functional specification and the added runtime errors annotations). As a result, that could make the proof less efficient and sometimes report a provable property as unproven.

While WP can prove that a program respects a specification, this specification has to be carefully written. For example, Figure 5 illustrates an example of a valid, yet not precise, specification for the `max` function. The **`\result`** keyword is used to refer to the value returned by the function.

The `max` function can be proved to correctly implement the specification that has been stated. However the assertion on line 13 will not be proved. The postcondition provides the information that the returned value is greater or equal to both `a` and `b`, but it does not state that the result is equal to one of them. We have to add another **`ensures`** clause:

```
ensures \result == a ∨ \result == b ;
```

However, it is still not precise enough and does not allow WP to prove the assertion on line 14. Indeed, the **`assigns`** clause is not provided, so the function is just assumed to assign any memory location. Here, the function does not assign any memory location, so the **`assigns \nothing`** clause can be added to provide the right specification. With this specification, both assertions will be proved by WP.

```
1  /*@
2    requires 0 ≤ len;
3    requires \valid(a + (0 .. len-1));
4    assigns a[0 .. len-1];
5    ensures ∀ ℤ i ; 0 ≤ i < len ⇒ a[i] == 0;
6  */
7  void reset_array(int* a, int len){
8    /*@
9      loop invariant 0 ≤ i ≤ len ;
10     loop invariant ∀ ℤ j; 0 ≤ j < i ⇒
11                            a[j] == 0 ;
12     loop assigns i, a[0 .. len-1];
13     loop variant len - i ;
14   */
15   for(int i = 0 ; i < len ; ++i){
16     a[i] = 0 ;
17   }
18 }
```

Fig. 6. The annotated `reset_array` function

C. Loop Contracts

While EVA can analyze loops automatically without intervention of the user, it is not the case for WP. Proving a program that comprises loops requires the user to provide a *loop invariant* for each loop. A loop invariant is a property that is true before and after each iteration of the loop.

The notion of invariant is illustrated with the `reset_array` function in Figure 6. In the **`requires`** clause on line 3 we use the syntax `a + (0 .. len-1)` that means *all pointers between a+0 and a+len-1 (included)*. The postcondition on line 5 states that all elements of the array between 0 and the end are 0, while line 4 indicates that all array elements can be assigned.

The invariant is introduced using the **`loop invariant`** keyword. Here, a first invariant property (line 9) of the loop is the fact that the value of `i` is comprised between 0 and the *included* length of the array. For a loop invariant, WP generates two proofs obligations, that it is *initially true* and *preserved*:

- the property must be true at the beginning of the loop,
- provided that the property is true before, it must still be true after executing an iteration of the loop.

So here, it will generate a first proof obligation to check that 0 is comprised between 0 and the array length, which is trivially true, and a second one to check that when `0 ≤ i ≤ len` holds before an iteration, it holds after. That is true because:

- if `i == len`, the body of the loop is not executed so the property holds,
- if `0 ≤ i < len`, since we only increment `i`, we have `1 ≤ i ≤ len` after the body, and the property holds as well.

This first invariant is not enough to be able to prove the postcondition of the function. When WP tries to prove the proof obligation that is related to the postcondition of a function containing a loop, it does not analyze the body of the loop, but only relies on the loop invariant. That is why

```
1 int i = 42;
2 /*@
3   loop invariant 0 ≤ i ≤ 42 ;
4   loop assigns i ;
5   loop variant i ;
6 */
7 while(i > 0){
8   i = i - ((rand()%i)+1) ;
9 }
```

Fig. 7. A loop variant that is not the number of remaining iterations

we have to specify an invariant that will allow to deduce the property we want to prove about the function. Typically, it specifies the information that has been ensured by the loop so far.

Here, the goal is to show that at the end of the execution of the function, each cell of the array `a` has been set to 0 (line 5). This property is established by the function by successively writing 0 in each cell. Therefore, the information we know after an iteration is which cells have been visited and set to 0 so far. So, the second important invariant (line 10) is:

```
∀ ℤ j; 0 ≤  j <  i ⇒  a[j] ==  0
```

One can easily verify that this invariant is established and maintained by the loop. With this loop invariant, the tool can prove the postcondition. Indeed, the combination of the first invariant and the negation of the loop condition implies that `i ==  len` and therefore, by the second invariant, we get the postcondition.

In order to prove the **`assigns`** clause of the function (line 4), we also have to annotate the loop with assignment information. It is introduced with a **`loop assigns`** clause. While local variables are not required in the **`assigns`** clause of a function, it is necessary to consider them in the case of the **`loop assigns`** clause, since the loop contract is the only information WP has about the loop. Without this clause, it would not be possible to prove the postcondition because it would not be possible to prove that `len` has not been modified during the execution of the loop. Of course, it is possible to provide this property as a loop invariant but the **`loop assigns`** clause is meant to do this for all non-modified variables at the same time. In this loop, the assigned memory location are `i` (which is incremented at each iteration), and all array elements of `a` at indices between `0` and `len - 1`, leading to the **`loop assigns`** clause on line 12.

Finally, the **`loop variant`** clause (line 12) allows the tool to prove that a loop terminates. A loop variant is an expression that must be positive whenever an iteration starts, and strictly decreasing after each execution of the body of the loop. From a loop variant clause, WP generates two proof obligations. First, it requires to prove that the value is indeed positive, second, that it is a decreasing value when the body of the loop is executed. Here, `len - i` is positive or nul because `i ≤ len`, and it is easy to prove that this value is decreasing because `len - (i+1)` is less than `len - i`.

```
1  /*@
2    requires 0 ≤ N ≤ 1000000000;
3    assigns \nothing;
4    ensures \result * \result ≤ N ;
5    ensures N < (\result+1) * (\result+1);
6  */
7  int root(int N){
8    int R = 0;
9    /*@
10     loop invariant 0 ≤ R * R ≤ N;
11     loop assigns R;
12     loop variant N-R;
13   */
14   while(((R+1)*(R+1)) ≤ N) {
15     R = R + 1;
16   }
17   return R;
18 }
```

Fig. 8. The specified `root` function

The loop variant can be seen as an upper bound on the number of remaining loop iterations. However it does not necessarily express the exact number of remaining iterations, as it does in the `reset_array` example. For example, in Figure 7, on line 5, `i` is a correct variant of the loop: it is indeed positive and decreasing, however it is not necessarily the number of remaining iterations since the loop can end at the first iteration if `rand()%i` can be 41.

Consider, again, the integer square root example (cf. Figure 2). We assume here that `N` is less or equal to 1,000,000,000 to avoid overflows on line 14. This example contains non-linear expressions, and EVA was not able to prove the absence of runtime errors automatically without unrolling the loop. It can be done by WP with the specification illustrated by Figures 8 and 13. Note that the first part of the postcondition (line 4) is guaranteed by the invariant (line 10) and that the second part of the postcondition (line 5) is guaranteed by the condition of the loop (line 14).

D. A Real-Life Example

During the tutorial, we specify and prove a real-life example, a function extracted from the memory management module of Contiki: the memory allocation function, that is shown in Figure 9. In Contiki, memory blocks are taken in pre-allocated memory regions. These regions are associated to a `memb` structure. This structure defines 4 fields:

- the `size` field represents the size of each memory block in the region (and will be the size of a memory block that is returned by an allocation),
- the `num` field represents the number of blocks in the region,
- the field `count` is an array of `num` values, whose cells indicate if the corresponding block is free or not,
- the field `mem` points to a memory region whose size is `size * num`.

The behavior of the memory allocation function is simple: when it is called, it iterates over the blocks to determine if one

```
struct memb {
  unsigned short size;
  unsigned short num;
  char *count;
  void *mem;
};

void *
memb_alloc(struct memb *m)
{
  int i;

  for(i = 0; i < m->num; ++i) {
    if(m->count[i] == 0) {
      m->count[i] = 1 ;
      int loc = i * m->size ;
      return (void *)((char *)m->mem + loc);
    }
  }

  return NULL;
}
```

Fig. 9. The MEMB allocation function

of them is free. If such a block is found, the corresponding cell in `count` is marked as a busy block, and the corresponding address of the block is returned. If the function cannot find a free block, it returns `NULL`.

In order to prove this function, it is necessary to express the invariant of the data structure, that basically states that the memory blocks are valid with the right sizes. Then, two cases can be distinguished: either there exists an available block or not. In the first case, we have to show that the function has indeed allocated a block, that all previously allocated blocks are still allocated and that the block we allocated is not one of them. In the second case, we have to show that the function has not modified the data structure.

IV. Runtime Verification Using E-ACSL2C

FRAMA-C was initially designed as a static analysis platform, but it was later extended with plugins for dynamic analysis. One of these plugins is E-ACSL, a runtime verification tool.

E-ACSL supports runtime assertion checking [11]. Assertions are very convenient for detecting errors and providing information about their locations. It is the case even when such an error does not result in a failure during execution.

In FRAMA-C, E-ACSL is both the name of the assertion language and the name of a plugin that generates C code to check these assertions at runtime. For the sake of clarity from now on we will use E-ACSL only for the language, and E-ACSL2C for the plugin.

E-ACSL is a subset of ACSL: the specifications written in this subset can therefore be used both by WP and E-ACSL2C. WP tries to prove the correctness of these assertions *statically* using automated provers, while E-ACSL2C is used to translate these assertions into C code that can then be executed. In this case the assertions are checked *dynamically*.

```
int main(void){
  f(42);
  f(0);
  return 0;
}
```

Fig. 10. A program calling `f` of Figure 1

A. Some Simple Examples

Let us first consider the programs of Figure 1. It is possible to check the additional assertion **`assert`** `sum ≠ 0;` (added before Line 10) when `f` is called in the `main` function of Figure 10.

Assuming the file `main.c` contains both one of the definitions of `f` of Figure 1 and the `main` of Figure 10 the following command

```
frama-c -e-acsl main.c -then-last \
   -print -ocode monitored_main.c
```

generates a file `monitored_main.c` where the assertion **`assert`** `sum ≠ 0;` has been converted to C executable code. In particular this file contains, after this assertion, the following code:

```
e_acsl_assert(sum ≠ 0, "Assertion", "f",
              "sum␣≠␣0", 10);
```

This can be considered as an evolved version of the C **`assert`** macro. This function checks the boolean expression `sum ≠ 0`, as the C **`assert`** macro does. In addition, if the expression is false it prints the kind of E-ACSL annotation this checks comes from (here an assertion). It also prints the function in which this assertion is located (here `f`), a string representing the condition itself (here the string `"sum␣≠␣0"`), and finally the line of the source code where the E-ACSL annotation is (in this case 10).

Compiling `monitored_main.c` requires several libraries, but FRAMA-C provides a script to easily generate and compile such a file. With the command

```
e-acsl-gcc.sh main.c -c -O monitored_main
```

an executable file `monitored_main.e-acsl` is generated. If executed, for function `f` of Figure 1a, we obtain:

```
Assertion failed at line 10 in function f.
The failing predicate is:
sum != 0.
Aborted (core dumped)
```

If `main.c` contains function `f` of Figure 1b instead, there is no output as both calls to `f` are correct.

The second example illustrates two features of E-ACSL2C: function contracts and segmentation faults. Figure 11 presents one function of the list API of Contiki: `list_init`. The contract we give here is partial: we just require as a precondition that its argument is a valid pointer, and that the function assigns the dereferencement of this pointer. The main function does not contain any assertion, however as `list_init` has a contract, the precondition is checked at each call. The first

```
1  #include "stdlib.h"
2
3  struct list {
4    struct list *next;
5    int value;
6  };
7
8  /*@
9    requires \valid(list);
10   assigns *list;
11 */
12 void list_init(struct list ** list) {
13   *list = NULL;
14 }
15
16 int main(void){
17   struct list ** l = malloc(sizeof(void *));
18   list_init(l);
19   free(l);
20   list_init(l);
21 }
```

Fig. 11. The `init_list` Function

call is correct, but after Line 19 the pointer `l` is dangling, and the verification of the precondition for the second call fails:

```
Precondition failed at line 8 in function
  list_init.
The failing predicate is:
\valid(list).
Aborted (core dumped)
```

Handling memory related constructs such as **`\valid`** requires to query the program memory at runtime. Queries include checking whether some data has been fully initialized, getting the length of a memory block, or getting the offset of a pointer from its base address.

In order to support this kind of queries, E-ACSL2C comes with a dedicated memory runtime library [12], [13]. `e-acsl-gcc.sh` takes care of linking this library against the generated code. This code records program memory modifications in a dedicated data structure, which can then be queried to evaluate memory-related E-ACSL constructs.

This instrumentation is expensive. In order to limit it, E-ACSL2C implements static analyses to over-approximate the memory locations to be monitored [14]. It is then unnecessary to track all the other locations.

B. E-ACSL *Specification Language*

In Section III we mentioned that the ACSL language contains basically first order logic formulae composed of pure C expressions as well as pure logic types (mathematical integers, reals, sets, lists, ...) and some built-in predicates. In addition, as we have seen, it is possible to define logical functions, but also predicates. Logicial functions can be defined in an axiomatic way, and predicates can be defined inductively.

Axiomatic functions and inductively defined predicates are not part of E-ACSL. The pure logical types are part of E-ACSL, however most tools only support mathematical integers. In E-ACSL2C they are translated into C using the GNU Multi-Precision library, when it is necessary. Finally, all quantifications in E-ACSL should be bounded, as the formula of Line 5 in Figure 6.

E-ACSL2C currently ignores logical functions and predicates in annotations.

C. A Real-Life Example

During the tutorial, we will consider the following scenario: some API has been verified using WP, and is used to implement applications. Verifying these applications using WP is time consuming, but we would like at least to check that the API calls are made correctly using runtime assertion checking. This allows to ensure, at least for the considered test suite, that the verified API is not called on inadmissible inputs for which its behavior was neither specified nor verified.

We will consider case studies similar to the one presented in Figure 12, where `list_chop` is a function of the linked list API of Contiki. As this API also contains a function `int length(`**`struct`**` list **)`, we assume that the maximal length for a list is `INT_MAX`. The tail recursive logical function `length_aux` takes that into account to stop the recursion if its accumulative parameter has reached the limit. In this case, even if there is still a next element, the function stops and returns `-1`. The `length` function simply calls `length_aux` with an initial accumulative value of `0`.

The main function first builds a circular list. However we will see that, due to the design of the logical function `length_aux`, the runtime checking shows that the precondition `0` $\leq$ `length(l)` is not satisfied for the call on line 56.

V. CONCLUSION AND FURTHER READING

A. Concluding Remarks

With the expansion of connected devices in the modern world, formal verification of IoT software attracts a growing interest. The goal of this tutorial is to show how formal verification can be applied to IoT software using the FRAMA-C verification platform and its plugins for value analysis, deductive verification and runtime assertion checking.

Value analysis allows users to prove the absence of runtime errors and does not require annotations. It relies on an over-approximation and can therefore report false alarms (or false positives). They can sometimes be eliminated by increasing the precision of the analysis, but this can lead to making the analysis much slower.

Value analysis is, however, often unable to verify more complex functional properties or deal with complex (e.g. non-linear) programs. When it is necessary, deductive verification can be used to treat them and to formally prove that a program respects its formal specification. It proceeds in a modular way and requires a carefully annotated program. When successful, it ensures that each function respects its specification, and function calls respect the corresponding preconditions.

Unfortunately, in a big software product, formal specification and deductive verification often remain partial, done for the most critical parts of the code. To take benefit from a partial

```c
#include "limits.h"
#include "stdlib.h"

struct list
{
  struct list *next;
  int value;
};

/*@
  logic int length_aux{L}(struct list * l,
                            int n)=
    n < (int)0 ? ((int)-1) :
      l == NULL ? n :
        n < INT_MAX ?
          length_aux(l->next, (int)(1+n)) :
          ((int)-1);

  logic int length{L}(struct list * l) =
    length_aux(l, (int)0);
*/

/*@
  requires \valid(list);
  requires 0 ≤ length(*list);
*/
struct list * list_chop(struct list ** list){
  struct list *l, *r;

  if(*list == NULL) {
    return NULL;
  }

  if((*list)->next == NULL) {
    l = *list;
    *list = NULL;
    return l;
  }

  l = *list;
  while(l->next->next ≠ NULL){
    l = l->next;
  }
  r = l->next;
  l->next = NULL;
  return r;
}

int main(void){
  struct list node;
  node.value = 1;
  node.next  = &node;

  struct list * l = &node;

  l = list_chop(&l);
}
```

Fig. 12. The `list_chop` Function

formal specification (of a library, module, etc.), the user can use runtime verification of the provided specifications during testing. In that case, a complete specification is not required for the whole project, and violations of the provided (even partial) specifications can be detected during the execution of a test suite.

We have demonstrated how these techniques can be applied using FRAMA-C and illustrated them on real-life examples extracted from IoT software.

B. Further Reading

1) On FRAMA-C *verification platform:* The first author wrote a longer tutorial focused on WP plugin [15]. Burghardt and Gerlach authored and regularly update their book "ACSL by Example" [16] giving many interesting examples of specification in ACSL. Several other tutorial papers present various analysis techniques using FRAMA-C: deductive verification [17], runtime verification [18], [19], test generation [20] and analysis combinations [21]. Finally, user manuals for FRAMA-C and its different analyzers can be found on the website http://frama-c.com.

2) On FRAMA-C *Applied to IoT Verification:* Several modules of Contiki have been verified with FRAMA-C:

- a memory allocation module [3],
- a linked list module [4], [22],
- the AES-CCM* modules [5].

Other verification projects are in progress.

REFERENCES

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[2] P. Baudin, P. Cuoq, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*. [Online]. Available: http://frama-c.com/acsl.html

[3] F. Mangano, S. Duquennoy, and N. Kosmatov, "A memory allocation module of Contiki formally verified with Frama-C. A case study," in *11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016)*, ser. LNCS, vol. 10158. Springer, 2016, pp. 114–120.

[4] A. Blanchard, N. Kosmatov, and F. Loulergue, "Ghosts for lists: A critical module of contiki verified in Frama-C," in *Nasa Formal Methods*, ser. LNCS, vol. 10811. Springer, 2018.

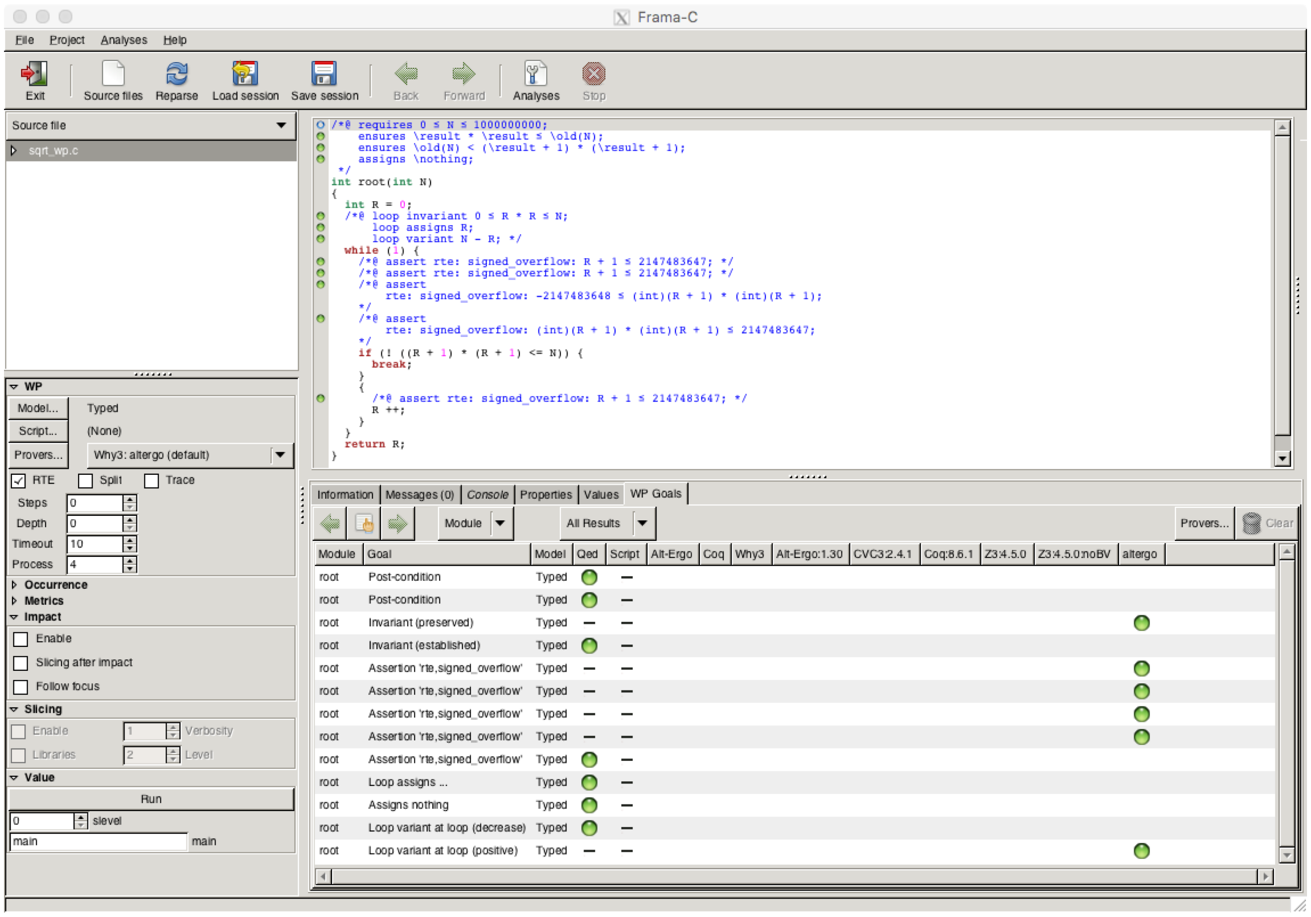
[5] A. Peyrard, S. Duquennoy, N. Kosmatov, and S. Raza, "Towards formal verification of Contiki: Analysis of the AES–CCM* modules with Frama-C," in *2nd International Workshop on Recent Advances in Secure Management of Data and Resources in the IoT (RED-IoT 2017) co-located with the International Conference on Embedded Wireless Systems and Networks (EWSN 2018)*. ACM, 2018, to appear.

[6] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," in *Proc. of the 29th Annual IEEE Conference on Local Computer Networks (LCN 2004)*. IEEE Computer Society, 2004, pp. 455–462.

[7] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. of the 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*. ACM, 1977, pp. 238–252.

[8] D. Bühler, P. Cuoq, and B. Yakobowski, *EVA - The Evolved Value Analysis plug-in*. [Online]. Available: http://frama-c.com/download/frama-c-value-analysis.pdf

[9] L. Mauborgne and X. Rival, "Trace partitioning in abstract interpretation based static analyzers," in *Proc. of the European Symposium on Programming (ESOP 2005)*, ser. LNCS, vol. 3444. Springer, 2005, pp. 5–20.



WP

Model...	Typed
Script...	(None)
Provers...	Why3: altergo (default)

[✓ RTE] Split Trace

Steps	0
Depth	0
Timeout	10
Process	4

```
/*@ requires 0 ≤ N ≤ 1000000000;
    ensures \result * \result ≤ \old(N);
    ensures \old(N) < (\result + 1) * (\result + 1);
    assigns \nothing;
 */
int root(int N)
{
  int R = 0;
  /*@ loop invariant 0 ≤ R * R ≤ N;
      loop assigns R;
      loop variant N - R; */
  while (1) {
    /*@ assert rte: signed_overflow: R + 1 ≤ 2147483647; */
    /*@ assert rte: signed_overflow: R + 1 ≤ 2147483647; */
    /*@ assert
        rte: signed_overflow: -2147483648 ≤ (int)(R + 1) * (int)(R + 1);
    */
    /*@ assert
        rte: signed_overflow: (int)(R + 1) * (int)(R + 1) ≤ 2147483647;
    */
    if (! ((R + 1) * (R + 1) <= N)) {
      break;
    }
    {
      /*@ assert rte: signed_overflow: R + 1 ≤ 2147483647; */
      R ++;
    }
  }
  return R;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo	Coq	Why3	Alt-Ergo:1.30	CVC3:2.4.1	Coq:8.6.1	Z3:4.5.0	Z3:4.5.0:noBV	altergo
root	Post-condition	Typed	●	—									
root	Post-condition	Typed	●	—									
root	Invariant (preserved)	Typed	—	—									●
root	Invariant (established)	Typed	●	—									
root	Assertion 'rte,signed_overflow'	Typed	—	—									●
root	Assertion 'rte,signed_overflow'	Typed	—	—									●
root	Assertion 'rte,signed_overflow'	Typed	—	—									●
root	Assertion 'rte,signed_overflow'	Typed	—	—									●
root	Assertion 'rte,signed_overflow'	Typed	●	—									
root	Loop assigns ...	Typed	●	—									
root	Assigns nothing	Typed	●	—									
root	Loop variant at loop (decrease)	Typed	●	—									
root	Loop variant at loop (positive)	Typed	—	—									●

Fig. 13. The `root` Function in Frama-C GUI

[10] E. W. Dijkstra, "A constructive approach to program correctness," *BIT Numerical Mathematics*, vol. Springer, 1968.

[11] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 25–37, May 2006.

[12] N. Kosmatov, G. Petiot, and J. Signoles, "An optimized memory monitoring for runtime assertion checking of C programs," in *Proc. of the 4th International Conference on Runtime Verification (RV 2013)*, ser. LNCS, vol. 8174. Springer, 2013, pp. 167–182.

[13] K. Vorobyov, J. Signoles, and N. Kosmatov, "Shadow state encoding for efficient monitoring of block-level properties," in *Proc. of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM 2017)*. ACM, Jun. 2017, pp. 47–58.

[14] A. Jakobsson, N. Kosmatov, and J. Signoles, "Fast as a shadow, expressive as a tree: Optimized memory monitoring for C," *Sci. Comput. Program.*, vol. 132, pp. 226–246, 2016, special Issue, Revised Selected Papers of SAC-SVT 2015,.

[15] A. Blanchard, "Introduction to C program proof using Frama-C and its WP plugin," december 2017. [Online]. Available: https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf

[16] J. Burghardt and J. Gerlach, "ACSL by example," 2018. [Online]. Available: https://github.com/fraunhoferfokus/acsl-by-example

[17] N. Kosmatov, V. Prevosto, and J. Signoles, "A lesson on proof of programs with Frama-C. invited tutorial paper," in *Proc. of the 7th International Conference on Tests and Proofs (TAP 2013)*, ser. LNCS, vol. 7942. Springer, Jun. 2013, pp. 168–177.

[18] N. Kosmatov and J. Signoles, "A lesson on runtime assertion checking with Frama-C," in *Proc. of the 4th International Conference on Runtime Verification (RV 2013)*, ser. LNCS, vol. 8174. Springer, 2013, pp. 386–399.

[19] ——, "Runtime assertion checking and its combinations with static and dynamic analyses – tutorial synopsis," in *Proc. of the 8th International Conference on Tests and Proofs (TAP 2014), Held as Part of STAF 2014*, ser. LNCS, vol. 8570. Springer, 2014, pp. 165–168.

[20] N. Kosmatov, N. Williams, B. Botella, M. Roger, and O. Chebaro, "A lesson on structural testing with PathCrawler-online.com," in *Proc. of the 6th International Conference on Tests and Proofs (TAP 2012)*, ser. LNCS, vol. 7305. Springer, May 2012, pp. 169–175.

[21] N. Kosmatov and J. Signoles, "Frama-C, A collaborative framework for C code verification: Tutorial synopsis," in *Proc. of the 7th International Conference on Runtime Verification (RV 2016)*, ser. LNCS, vol. 10012. Springer, Sep. 2016, pp. 92–115.

[22] F. Loulergue, A. Blanchard, and N. Kosmatov, "Ghosts for lists: from axiomatic to executable specifications," in *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018), Held as Part of STAF 2018*, ser. LNCS, vol. 10889. Springer, Jun. 2018, to appear.