Towards Full Proof Automation in Frama-C using Auto-Active Verification

Allan Blanchard[1] (0000−0001−7922−4880), Frédéric Loulergue[2] (0000−0001−9301−7829), and Nikolai Kosmatov[3] (0000−0003−1557−2813)

[1] Inria Lille — Nord Europe, Villeneuve d'Ascq, France
allan.blanchard@inria.fr
[2] School of Informatics Computing and Cyber Systems, Northern Arizona University, USA
frederic.loulergue@nau.edu
[3] CEA, List, Software Reliability and Security Lab, PC 174, Gif-sur-Yvette, France
nikolai.kosmatov@cea.fr

Abstract. While deductive verification is increasingly used on real-life code, making it fully automatic remains difficult. The development of powerful SMT solvers has improved the situation, but some proofs still require interactive theorem provers in order to achieve full formal verification. Auto-active verification relies on additional guiding annotations (assertions, ghost code, lemma functions, etc.) and provides an important step towards a greater automation of the proof. However, the support of this methodology often remains partial and depends on the verification tool. This paper presents an experience report on a complete functional verification of several C programs from the literature and real-life code using auto-active verification with the C software analysis platform FRAMA-C and its deductive verification plugin WP. The goal is to use automatic solvers to verify properties that are classically verified with interactive provers. Based on our experience, we discuss the benefits of this methodology and the current limitations of the tool, as well as proposals of new features to overcome them.

1 Introduction

Formal verification enables obtaining a high level of assurance in software reliability, but requires significant expertise and is still very costly to apply on real-world use cases. The purpose of this experience report is to investigate how to achieve a higher level of automation of formal verification. We address this problem in the context of FRAMA-C.

Context and Motivation. FRAMA-C [19] is a source code analysis platform that aims at conducting verification of industrial-size programs written in C. FRAMA-C offers a large range of analyzers, such as abstract interpretation based value analysis, deductive verification, dependence analysis, program slicing, runtime verification, test generation, etc., as well as a a common specification language: ACSL [3].

WP is the deductive verification plugin of FRAMA-C. It is based on the weakest precondition calculus [11]. WP can be used to prove functional correctness of C programs with respect to a specification written in the form of ACSL annotations. Given an annotated program, WP computes *verification conditions* (VCs) or *proof obligations*, that is, properties that must be proved to ensure that the program respects its specification. These verification conditions can then be proved either automatically by SMT solvers (for example, by Alt-Ergo [10], or — through the Why3 [14] platform — by CVC4 [2] or Z3 [24]) or by interactive provers (like the Coq proof assistant [28]).

Generally, the verification engineer wants to achieve a maximal level of automation, thus relying on SMT solvers whenever possible. In practice, it is not always possible. For example, when the proof of some properties requires reasoning by induction, SMT solvers often remain inefficient. Thus, most of the time the solution is to introduce generic lemmas that can be directly instantiated by SMT solvers to prove verification conditions, and to prove those lemmas using interactive provers like Coq. However, producing a Coq proof often requires extensive background in formal methods, can be time consuming and can increase the maintenance cost when a new version of the verification condition generator is released. It makes the proof much harder for many verification engineers.

Approach and Goals. Leino and Moskal [21] proposed the notion of *auto-active verification* to qualify verification approaches that suggest to enrich the code with more annotations than just necessary to express function contracts. They are available to the verification tool before the generation of the verification conditions, and so can be used to optimize this generation. However, depending on the tools, auto-active features are more or less developed and lead to different levels of automation.

In this experience report, we investigate the use of auto-active verification with FRAMA-C/WPand apply it to verify the following three case studies[4]:

1. a memory allocation module [22],
2. a linked list module [5],
3. all examples of an independent benchmark called *ACSL by Example* [7].

The first two examples are taken from recent real-life case studies on formal verification of the Contiki operating system [13]. The third artifact contains a rich set of annotated programs verified with FRAMA-C/WP, maintained by Fraunhofer FOKUS. Previous proofs of these programs essentially relied on some interactive proofs in Coq.

Our goal was to increase the level of automation and to demonstrate that, thanks to auto-active verification, interactive proof was not needed anymore to achieve full formal verification. Based on our experience, we analyze the results in terms of necessary annotations and proof effort. We identify some features that are currently not available in FRAMA-C, but appear to be necessary for a better auto-active verification. We also provide some discussion and comparison with existing approaches.

Contributions. The contributions of this work include:

– a formal specification and a *fully automatic proof* of the aforementioned programs using the auto-active verification approach;
– analysis of the effort that was needed to achieve this goal;
– identification of several features that could be helpful (and some workarounds we used to overcome their absence);
– a comparison with existing work that could be integrated into FRAMA-C to improve auto-active verification.

[4]All source code available at: http://allan-blanchard.fr/code/auto-active-nfm-2019.zip

```
1  struct memb {
2    unsigned short size;
3    unsigned short num;
4    char *busy;
5    void *mem;
6  };
7
8  /*@ requires valid_memb(m);
9      assigns m->busy[0 .. (m->num - 1)];
10     behavior free_found:
11       assumes ∃ ℤ i; 0 ≤ i < m->num ∧ m->busy[i] == 0;
12       ensures _memb_numfree(m) == \old(_memb_numfree(m)) - 1;
13     behavior full:
14       assumes ∀ ℤ i; 0 ≤ i < m->num ⇒ m->busy[i] ≠ 0;
15       ensures _memb_numfree(m) == \old(_memb_numfree(m));
16     complete behaviors; disjoint behaviors; */
17 void * memb_alloc(struct memb *m){
18   /*@ loop invariant 0 ≤ i ≤ m->num;
19       loop invariant ∀ ℤ j; 0 ≤ j < i ⇒ m->busy[j] ≠ 0;
20       loop assigns i;
21       loop variant m->num - i; */
22   for(int i = 0; i < m->num; ++i) {
23     if(m->busy[i] == 0) {
24       m->busy[i] = 1 ;
25       /*@ assert one_change{Pre, Here}(i, m->busy, 0, m->num); */
26       return address_of_block(m, i);
27     }
28   }
29   return NULL;
30 }
```

Fig. 1: The memory allocation function of MEMB and its (partial) specification

Outline. The paper is organized as follows. Section 2 presents a running example verified using a classic combination of automatic and interactive proof. Section 3 explains how to avoid interactive proofs. Section 4 further details the results of our experiments. Section 5 compares our approach with other tools. Section 6 discusses some lessons learned and necessary features. Section 7 gives a conclusion and future work.

2 Classic Lemma-Based Verification in Frama-C

In this section, we briefly present ACSL, the specification language of FRAMA-C, its deductive verification plugin WP and the way it is generally used to prove a particular program. We illustrate it on a running example (shown in Fig. 1) taken from the Contiki operating system: function `memb_alloc` of the memory management module MEMB. This module simulates memory (de-)allocation using a set of pre-allocated blocks that can be attributed (or released) on demand. Function `memb_alloc` iterates over blocks and if it finds an available block, it marks it as allocated and returns its address.

Running Example. The `memb` structure contains the data required to manage a set of memory blocks. The `size` field (line 2 in Fig. 1) indicates the size of a block in bytes, while `num` (line 3) indicates the maximal number of allocable blocks. The pointer `mem` (line 5) refers to the pre-allocated array of blocks (of total size `num*size`). The main invariant to maintain in the module is the fact that it correctly keeps track of allocated (i.e. attributed) and free (i.e. released) blocks. This is done in the `busy` field (line 4), which contains an array of length `num` whose i-th cell indicates for the i-th block whether it is busy (value 1) or not (value 0). In the verification, we simply consider that a block of index i is free when the corresponding value at the cell of index i of `busy` is 0, and allocated otherwise. From a specification point of view, this knowledge is also

```
1  /*@ axiomatic OccArray{
2      logic ℤ occ{L}(ℤ e, char *a, ℤ from, ℤ to)
3        reads a[from .. (to - 1)];
4      axiom end_occ{L}:
5      ∀ ℤ e, char *a, ℤ from, to;
6        from ≥ to ⇒ occ{L}(e, a, from, to) == 0;
7      axiom iter_occ_true{L}:
8      ∀ ℤ e, char *a, ℤ from, to;
9        (from < to ∧ a[to-1] == e) ⇒
10       occ{L}(e, a, from, to) == occ{L}(e, a, from, to-1) + 1;
11     axiom iter_occ_false{L}:
12     ∀ ℤ e, char *a, ℤ from, to;
13       (from < to ∧ a[to-1] ≠ e) ⇒
14       occ{L}(e, a, from, to) == occ{L}(e, a, from, to-1);
15   }
16   logic ℤ _memb_numfree(struct memb *m) =
17     occ(0, m->busy, 0, m->num);
18 */
```

Fig. 2: The axiomatic definition of occurrence counting

used to ensure that after an allocation fewer memory blocks are available (and dually, that after a free, more blocks are available). In this example, we will focus on this last simple property and ignore some other aspects of the proof (for more detail see [22]).

In an ACSL contract, a precondition is specified in a **requires** clause. Here, the function requires (line 8) that the received memb structure be valid (in the sense of the memb invariant). There exist two kinds of postconditions. An **assigns** clause specifies a list of memory locations that the function is allowed to modify. An **ensures** clause specifies properties that should be verified after the function call. Here, the function can potentially assign (cf. line 9) any memory cell of the busy array (even though we know it will actually assign at most one) — the **assigns** clause has only to give a correct over-approximation. The unmodified values can be further constrained by other postconditions. We can specify different behaviors (or cases) for the function. Here, when the function is called, either there exists a free block (lines 10–12), or not (lines 13–15). The condition that defines each case is stated using an **assumes** clause. Each behavior can have its own postconditions. In this example we only consider postconditions on the number of free blocks. If some free blocks are available (line 11), the function ensures that one more block is allocated, that is, one less block is free (line 12). Otherwise, if all blocks are busy (line 14), the number of free blocks does not change (line 15). Line 16 states that the specified behaviors are complete and disjoint, that is, cover all possible cases and do not overlap. Notice that for convenience of the reader, the ACSL annotations in this paper are slightly pretty-printed (using e.g. mathematical notation $\mathbb{Z}$, $\forall$, $\exists$, $\Rightarrow$, $\neq$ instead of ACSL notation **integer**, \forall, \exists, ==>, !=, respectively).

To refer to program states at different program points in ACSL predicates, in addition to the usual *labels* of C language (see e.g. line 7 in Fig. 5), ACSL predefines a few additional labels, such as the current program point **Here** (used by default), the pre-state **Pre** on entry and the post-state **Post** on exit of the current function.

The definition of the _memb_numfree is provided in Fig. 2 (line 16–17). It consists in counting the number of occurrences of value 0 in the busy array of the MEMB structure. The counting operation occ{L}(e,a,from,to) is defined axiomatically (lines 1–15): it counts the number of occurrences of value e in array a at program label L, from an index from to an (excluded) index to. Label L can be omitted (cf. line 18) and is then by default **Here**, the current program point. The definition considers

```
/*@
lemma occ_split{L}:
  ∀ ℤ e, char *t, ℤ from, cut, to;
  from ≤ cut ≤ to ⇒ occ{L}(e,t,from,to) ==
                     occ{L}(e,t,from,cut)+occ{L}(e,t,cut,to);

predicate same_elems{L1,L2}(char *t, ℤ from, ℤ to) =
  ∀ ℤ j; from ≤ j < to ⇒ \at(t[j], L1) == \at(t[j], L2);

lemma same_elems_means_same_occ{L1, L2}:
  ∀ ℤ e, char *t, ℤ from, to;
  same_elems{L1,L2}(t,from,to) ⇒
    occ{L1}(e, t, from, to) == occ{L2}(e, t, from, to);

predicate one_change{L1,L2}(ℤ index, char *t, ℤ from, ℤ to) =
  from ≤ index < to ∧
  same_elems{L1,L2}(t, from, index) ∧ same_elems{L1,L2}(t, index+1, to) ∧
  \at(t[index], L1) ≠ \at(t[index], L2);

lemma one_change_means_inc_and_dec{L1, L2}:
  ∀ ℤ index, char *t, ℤ from, to;
    \let old_val = \at(t[index],L1); \let new_val = \at(t[index],L2);
    (one_change{L1,L2}(index, t, from, to)) ⇒ (
      occ{L1}(old_val, t, from, to)-1 == occ{L2}(old_val, t, from, to) ∧
      occ{L1}(new_val, t, from, to)+1 == occ{L2}(new_val, t, from, to) ∧
      (∀ ℤ other; other ≠ old_val ∧ other ≠ new_val ⇒
         occ{L1}(other, t, from, to) == occ{L2}(other, t, from, to))
    );
*/
```

Fig. 3: Predicates and lemmas to reason about occurrence counting

three cases. If the index range is empty then the number of occurrences is 0 (lines 4–6). If there is at least one cell to visit then the number of occurrences is (recursively) the number of occurrences in the array prefix of indices from `from` to `to-1` (excluded), incremented or not depending if the last value included in the initial range (that is, `a[to-1]`) is equal to `e` (lines 8–10) or not (lines 12–14). The **`reads`** clause (line 3) indicates the values that impact the result of the function and thus determines when a memory modification invalidates the property.

Lines 18–21 of Fig. 1 define in a classic way a loop contract specifying which variables can be modified (line 20) and what other properties remain valid (lines 18–19) after any number of complete loop iterations (i.e. *each time the loop condition is evaluated*), as well as a loop variant used to prove termination (line 21).

Lemmas and Assertions. Consider the instruction on line 24 of Fig. 1. It assigns a particular value of the `busy` array inside a loop. This value was previously 0 (line 23) and becomes 1. Thus the number of 0's decreases by 1. However, proving this property requires some kind of reasoning by induction, to show how the counting can be split into three segments: segment $[0,\texttt{i})$ preceding the value (possible empty), the modified value itself, and segment $[i+1,\texttt{m->num})$ that follows it (possibly empty again). We also need to reason by induction to show that since the first and the last array segments remain unchanged, the number of occurrences in them does not change either.

SMT solvers are not good at reasoning by induction. In our example, they cannot prove the contract of the function if we do not give them more help. The classic approach with WP is to provide lemmas that capture the inductive reasoning. These lemmas can be directly instantiated by SMT solvers to deduce the required conclusion from the premises without reasoning by induction.

Let us illustrate it with the three lemmas presented in Fig. 3. We state the lemma `one_change_means_inc_and_dec` to prove our property on the number of occurrences after a change in the array. This lemma states that if one (and only one) value changes in an array from a label `L1` to another label `L2`, then the number of occurrences of the old value decreases, the number of occurrences of the new value increases and the number of occurrences of other values does not change. While we could directly prove this property in Coq, it is easier to state two more lemmas to ease the process. Lemma `occ_split` allows splitting the total range of indices into two subranges (segments). Lemma `same_elems_means_same_occ` states that when all the values of an array remain unchanged between two labels, the number of occurrences does not change either. All these lemmas cannot be proved directly by most SMT solvers, so classically they are proved using an interactive prover, e.g. the Coq proof assistant.

Once these lemmas have been stated, they can be automatically used by SMT solvers to discharge the different verification conditions generated by WP. However, depending on the property that needs to be proved, and on the proof context (i.e., the different properties known about the program at the corresponding program point), SMT solvers can fail to automatically *trigger* the use of these lemmas. For example, in the case when the function modifies the array, the ensures clause on line 12 will not be proved because of the complexity of the proof context. In this case, we use a first feature of auto-active verification: we add an assertion (line 25 in Fig. 1) that makes the link between the current state (label **`Here`**) and the state before the function starts (label **`Pre`**). It directly states the premise of Lemma `one_change_means_inc_and_dec` (cf. line 23 in Fig. 3) and will allow SMT solvers to trigger its application and to deduce that the number of 0's decreased (i.e. to deduce line 12 in Fig. 1 from line 24 in Fig. 3).

The more complex the proof context is, the harder is the automatic triggering of lemmas. So it is common to add assertions, but sometimes, even with assertions, triggering of lemmas can be hard to predict. The question is then to know whether we can further increase proof automation, both in efficient triggering of SMT solvers and in the ability to prove complex properties without interactive provers. Since this approach can require adding more annotations, the question is also whether these annotations are more costly to add compared to an interactive proof or not.

3 Auto-Active Verification Illustrated for the Running Example

As many other verification tools, FRAMA-C provides the ability to add *ghost code* in annotations of the source code being analyzed. Ghost code is regular C code, except that it is only visible by the verification tools. Ghost code must not interfere with the behavior of the program: it can basically observe the real world but cannot influence it.

Lemma functions are a particular form of ghost code. They consist of ghost functions that serve the same purpose as lemmas: to deduce some conclusions from some general premises. Like for lemmas, the properties are proved separately — here as function contracts — and can then be used in a particular context by simply calling the corresponding lemma function. Moreover, in lemma functions, some tricks can help to avoid the limitations of SMT solvers, for example, on reasoning by induction.

Let us illustrate on our running example how to completely remove the need for the Coq proof assistant. Basically, we can provide a way of performing the reasoning

```
1  /*@ requires from ≤ cut ≤ to;
2      ensures occ(e,a,from,to) == occ(e,a,from,cut)+occ(e,a,cut,to);
3      assigns \nothing; */
4  void occ_split(int e, char * a, int from, int cut, int to){
5    /*@ loop invariant cut ≤ i ≤ to;
6        loop invariant occ(e,a,from,i) == occ(e,a,from,cut)+occ(e,a,cut,i);
7        loop assigns i;
8        loop variant to - i; */
9    for(int i = cut ; i < to ; i++){ }
10 }
```

Fig. 4: The lemma function `occ_split` with its proving code

```
1  void * memb_alloc(struct memb *m){
2    /*@ ... */ // loop contract as in Fig.1
3    for(int i = 0; i < m->num; ++i) {
4      /*@ ghost occ_split(0, m->busy, 0, i, m->num); */
5      /*@ ghost occ_split(0, m->busy, i, i+1, m->num); */
6      if(m->busy[i] == 0) {
7      BA:
8        m->busy[i] = 1 ;
9
10       /*@ ghost same_elems_means_same_occ(BA, Here, 0, m->busy, 0, i) ; */
11       /*@ ghost same_elems_means_same_occ(BA, Here, 0, m->busy, i+1, m->num) ; */
12
13       /*@ ghost occ_split(0, m->busy, 0, i, m->num); */
14       /*@ ghost occ_split(0, m->busy, i, i+1, m->num); */
15
16       return address_of_block(m, i);
17     }
18   }
19   return NULL;
20 }
```

Fig. 5: Function `memb_alloc` fully proved in auto-active fashion

of the Coq proof of our lemma but using ghost code. The first step of the proof is to split our array again into three segments, the first and the last being unchanged, and the second containing the modified cell. We define a function that mimics the previously introduced `occ_split` lemma as shown in Fig. 4.

The contract of the function expresses the lemma. The premise of the lemma is specified in a precondition, and the conclusion as a postcondition. A lemma function is pure, and thus **`assigns \nothing`**. The **`for`** loop provides, thanks to its loop invariant (lines 5–6), a way to reason by induction on the distance between `cut` and `to`. Indeed, the VC generation for this invariant creates the two cases to prove: either the distance is zero, or the relation is known for rank i and must be verified for $i + 1$. The function is fully proved by SMT solvers.

The lemma can then be instantiated in the `memb_alloc` function by calling it with suitable parameters, as illustrated in Fig. 5, on lines 4–5 and 13–14. The first call splits the global range of counting into segments $[0,i)$ and $[i,$`m->num`$)$, and then the latter is split again into $[i, i + 1)$ and $[i + 1,$`m->num`$)$. Notice that we need to split the range before and after the assignment since it invalidates any knowledge about `occ` over the whole array (as defined by the **`reads`** clause on line 3 of Fig. 2).

To prove that the unmodified segments $[0,i)$ and $[i + 1,$`m->num`$)$ contain the same number of 0's before and after the assignment, the notion of labels must be taken into account, as is done for the lemma on lines 10–13 of Fig. 3. Therefore, we cannot use a lemma function here since *we cannot specify arbitrary labels in a function call.*

```
1  #define same_elems_means_same_occ(_L1, _L2, _value, _array, _from, _to) \
2    /@                                                                     \
3      loop invariant _from ≤ _k ≤ _to ;                                    \
4      loop invariant occ{_L1}(_value, _array, _from, _k) ==                \
5                     occ{_L2}(_value, _array, _from, _k) ;                 \
6      loop assigns _k ;                                                    \
7      loop variant _to - _k ; @/                                           \
8    for(int _k = _from ; _k < _to ; ++ _k) ;                              \
9    /@ assert occ{_L1}(_value, _array, _from, _to) ==                     \
10             occ{_L2}(_value, _array, _from, _to) ; @/
```

Fig. 6: The `same_elems_means_same_occ` lemma macro

As a workaround, we directly *inject* proof-carrying code at the program point where we want to prove that the number of occurrences did not change (cf. lines 10–11 of Fig. 5) between labels `BA` and **`Here`**. This is done via a C macro[5]. This macro is defined in Fig. 6, it receives two labels `_L1` and `_L2`, the counted `_value`, and the `_array` with the bounds of the considered segment. The assertion on lines 9–10 states the equality to be proved, and the **`for`** loop enables this proof thanks to the invariant on lines 3–4. We use the term *lemma macro* for such a proof-carrying macro.

Notice however that with this workaround, the proof is done each time we need an instance of the lemma: each instance of the lemma macro generates annotations to be proved. In contrast, the original `occ_split` lemma is proved once and for all, and then instantiated as many times as necessary.

Another issue in creating a lemma macro in more complex cases is that it can be difficult to design and to prove directly for a large proof context (where the proof can be sensitive to other annotations). We propose to use a wrapper function that creates the smallest possible relevant context for such a lemma macro.

Let us illustrate this point with the helper function (given by Fig. 7) that we used to build the lemma macro `same_elems_means_same_occ.` The purpose of the function on lines 9–13 is to create a sufficiently generic context that only establishes the premises of the original lemma, in order to validate that our lemma macro can indeed prove our property of interest in isolation of any other context. The lemma `same_elems_means_same_occ` (Fig. 3, lines 10–13) states that if `same_elems` holds for an array for two program labels `L1` and `L2` then the number of occurrences does not change between `L1` and `L2`. Let us consider that `L1` is the **`Pre`** label of the function, and `L2`, the **`Post`** label. We have to create a context that assures that the **`Pre`** and **`Post`** labels of the function are two different memory states related by `same_elems`, and that our lemma macro can assure the conclusion of the lemma.

We first use a function call `side_effect_to_L2` (line 10) to introduce a different state before the use of the lemma macro. The function is specified (and does not need an implementation) on lines 1–3. It assigns the array to invalidate any previous knowledge about it (line 1) and ensures the premise that relates the two labels of the lemma (line 2). By calling this function in the wrapper function, we ensure that there exist two program states **`Pre`** (that corresponds to `L1` in the lemma) and `SideEffectHappened` (that corresponds to `L2` in the lemma) that are different because some side effect happened in between, and related by the predicate `same_elems`. Since the remaining part of the code is the call to the lemma macro

[5]Given a macro **`#define`** `name(p1,...,pN) text`, the C preprocessor replaces any occurrence `name(e1,...,eN)` by `text` where named parameters are replaced accordingly.

```
1  /*@ assigns array[from .. to-1] ;
2      ensures same_elems{Pre, Post}(array, from, to) ; */
3  void side_effect_to_L2(char* array, int from, int to) ;
4
5  /*@ assigns array[from .. to-1] ;
6      ensures occ{Pre}(value, array, from, to) ==
7              occ{Post}(value, array, from, to) ;
8  */
9  void prove_same_elems_means_same_occ(int value, int* array, int from, int to){
10   side_effect_to_L2(array, from, to) ;
11 SideEffectHappened:
12   //@ ghost same_elems_means_same_occ(Pre, Here, value, array, from, to) ;
13 }
```

Fig. 7: Code produced to build the `same_elems_means_same_occ` macro (Fig. 6)

which is pure, the label **`Post`** is equivalent to `SideEffectHappened`. Finally, the role of this wrapper function is thus to create a context where some labels `L1` (= **`Pre`**) and `L2` (= `SideEffectHappened` = **`Post`**) are related by `same_elems` in order to deduce, using the lemma macro we want to create, that the number of occurrences between these two labels did not change (lines 6–7). Using this wrapper function, we can now design the lemma macro of Fig. 7 and prove the wrapper function with it.

As the attentive reader might notice, this wrapper function is not pure anymore, and the labels considered by the lemma macro are supposed to be ordered (while specified labels in ACSL lemmas are not). We further discuss these aspects in Section 6.

4 Experiments using Auto-Active Verification

In order to evaluate the use of auto-active verification with FRAMA-C/WP and to compare it to the classic lemma-based approach, we applied the auto-active approach to verify several previously verified examples from the literature and real-life code: (1) the memory allocation module `MEMB` (briefly presented in Section 2) and (2) the linked list module [5] of Contiki, as well as (3) all programs of an independent benchmark *ACSL by Example* [7]. For case studies (1) and (3), auto-active proofs were done without knowing the earlier Coq proofs. The list module appears to be the most complex case study. *In all these case studies, we managed to achieve a fully automatic proof with the auto-active approach*, except only for two lemmas explained below.

Approach. For each case study, we begin with the annotated code previously verified using interactively proved lemmas. First, we remove all lemmas from the annotations. Then, we identify the assertions that are not proved anymore. For each assertion, we consider the lemma(s) that are required to enable the verification. If such a lemma can be fully automatically proved (without relying on other lemmas that require interactive proof), it is preserved. Otherwise, we replace such a lemma by a lemma function or a lemma macro necessary to enable the verification, and adapt the annotations of the functions to verify.

For case studies (1) and (2), we also separately expressed and verified, in an auto-active style, the lemmas that were not required anymore (and thus were not preserved) with the auto-active approach. Our goal was to measure the time to verify all lemmas of these case studies. (For *ACSL by Example*, this was not done because of the large number of such lemmas.) In other statistics presented in the tables below for the auto-active approach we count only lemmas that were *required* for the verification.

	Lemmas, incl. lemma functions & lemma macros	Generated goals	Goals proved with Coq	Lines of code		Execution time
				lemmas, incl. l.fun./macros	guiding annotations	
Case study (1). The memory management module `MEMB` (70 lines of C code)						
Classic	15	134	15	33	20	47 sec.
Auto-active	3	217	1	25	25	19 sec.
Case study (2). The linked list module (176 lines of C code)						
Classic	24	805	19	163	708	24 min.
Auto-active	17	1631	1	366	629	21 min.
Case study (3). *ACSL by Example,* v. 17.2.0 (630 lines of C code)						
Classic	87	1398	40	594	485	92 min.
Auto-active	53	1790	0	670	611	78 min.

Table 1: Statistics for the considered case studies

Measured Indicators and Statistics. The `MEMB` module is composed of 5 functions; we needed to adapt the proofs for 2 of them. This adaptation to the auto-active approach took about 6 person-hours (including the proof of lemma functions that were not required). The verified version of the list module is composed of 11 functions and 2 auxiliary functions. We had to write one more auxiliary function, and to adapt the verification for 9 functions. In total, the adaptation took about 20 person-hours. This effort includes the time to validate that the proof remains valid on different list data-types (see [5] for more detail). The *ACSL by Example* repository contains 76 proved examples (some functions are proved with two different contracts, methods, or implementations), 12 of which needed to be adapted. This adaptation took about 30 person-hours.

Table 1 sums up the amount of annotation required by each approach for each use case. The second column indicates the number of declared lemmas (including lemma functions or lemma macros if any). The third column gives the number of proof obligations generated by WP from the annotated program. The fourth column indicates how many of these goals required an interactive proof. The fifth column indicates how many lines of code were required to write the lemma functions/macros (including function bodies, assertions, and loop invariants for them). The sixth column corresponds to the number of lines written to guide the proof, including loop invariants, and excluding the lines needed to prove lemma functions/macros (as we count them in column 5). The last column indicates the time needed to replay entirely the proofs of the goals[6].

Analysis of Results. We notice that in each case study, *we need fewer lemmas (including lemma functions) in the full auto-active approach*. The main reason for this result is that for complex proofs it is common to "duplicate" a lemma with slightly different premises or conclusions in order to make them easier to trigger by SMT solvers in the context of a particular proof. Each version must be separately proved in Coq. In the auto-active approach, since lemmas are manually instantiated, this duplication is not needed.

Furthermore, in the auto-active approach it is easier to know exactly which lemmas, and in particular lemma functions, are used during the verification. An instance of a lemma function is explicit: this is a function call (or an occurrence of a lemma macro). On the contrary, to detect the required lemmas in the classic approach, we need to proceed iteratively: (1) remove a lemma; (2) re-run all proofs to see whether some of them fail; (3) if that is the case, put the lemma back; and (4) go back to (1). This process

[6]Executed on a Core i7 6700HQ with 16GB DDR4. Versions of the tools: Frama-C 18 Argon, Why3 0.88.3, Alt-Ergo 2.2, CVC3 2.4.1, CVC4 1.6, Z3 4.8.1, E Prover 2.1, Coq 8.7.2.

is time-consuming, and it cannot be done before everything has been proved because triggering of lemmas can be impacted e.g. when we add more assertions in the code to prove. Thus, most of the time it is not done.

While the auto-active approach needs fewer lemmas/lemma-functions, *lemma functions often require more code to be written*. This was expected: we need to express a function contract, which is already longer than a lemma due to the format of the contract. We also need to provide a body for each lemma function that basically carries out the proof of the lemma. In our study, we noticed that we spent less time producing the auto-active proofs of the lemmas that were previously proved with Coq. For both case studies on Contiki modules, we proved all lemma function versions of the lemmas used in the classic approach (even those that we removed later since they were not required), it took about a day, while the corresponding Coq proofs took at least a week.

The number of generated proof obligations is greater for the auto-active approach. For the first two case studies, this number doubles. This increase was expected. Indeed, in the classic approach, the proof of a lemma generates a single proof obligation. In the auto-active approach, we need a proof obligation for the postcondition, but the body of the function also generates other proof obligations (for assertions, function calls, loop invariants or variants). The strong advantage is that *all these obligations are automatically verified by SMT solvers.*

Moreover, as we already mentioned, we use lemma macros to inline the proof-carrying code for lemmas that require multiple labels. Each time we use an instance of such a lemma, we perform a new complete proof of its statement. This significantly increases the number of proof obligations. The experiment on *ACSL by Example*, where the number of proof obligations increased less significantly, tends to confirm this intuition. Indeed, the semantic properties of the **`reads`** clause — stating that arrays with unmodified values between two labels have the same properties — were simply stated as axioms in the *ACSL by Example* case study, whereas we chose to prove them as lemmas in the other two case studies. These lemmas required creating additional lemma macros, resulting in more proof obligations.

Regarding guiding annotations needed in the verified code (that include assertions, loop invariants and ghost code), the results of the first and the third case studies indicate that *the auto-active approach needs more guiding annotations*. This is consistent with the idea that lemmas must be triggered manually. For the list module the opposite situation is observed. To explain that, we presume that during the adaptation of the proofs, we aggressively removed a lot of assertions to get faster results from SMT solvers. In this way, we removed some assertions that were superfluous in the classic approach.

Overall, *we expect that auto-active proofs will be easier to achieve for non-experts.* Using the auto-active approach does not require more expertise in FRAMA-C/WP than needed for the classic approach (proving a lemma function is just like proving any other C function). On the other hand, in the interactive approach, one also needs a strong expertise in Coq to prove the Coq goals generated from ACSL lemmas by WP, which involve a (relatively technical) encoding of the C memory model in Coq as well.

Resisting Lemmas. We were able to prove the great majority of lemmas without interactive proof. *Only two lemmas had to be proved in Coq.* The first one, used in the MEMB module, states: $\forall a, b \in \mathbb{Z},\ a \geq 0 \Rightarrow b > 0 \Rightarrow (a * b)/b = a$. Indeed, the reasoning of

SMT solvers can become harder here since WP encodes the division with a particular logic function that represents the C semantics of the division.

The second resisting lemma, used in the linked list module, is related to the encoding of inductive definitions by WP. We illustrate the issue with a simple inductive predicate (checking if a given vector `a` of length `size` contains only zeroes):

```
inductive null_vector(int* a, size_t size){
case len_0: ∀ int* a;
  null_vector(a, 0);
case len_n: ∀ int* a, size_t s;
  s > 0 ⇒ (null_vector(a+1, s-1) ∧ a[0] == 0) ⇒ null_vector(a, s);
}
```

To reason about the cases of such an inductive definition, we need the following fact:

```
axiom empty_or_not: ∀ int* a, size_t s;
  null_vector(a, s) ⇒ ( (s == 0) ∨ (s > 0 ∧ null_vector(a+1, s-1) ∧ a[0] == 0) );
```

Such a property is readily provided by Coq, but not by WP. This is why we have to explicitly state it as a lemma and establish it with a simple one-line proof[7] in Coq.

5 Related Work

The term *auto-active verification* was coined by Leino and Moskal [21] to designate methods of verification that rely on specific annotations added into the source code to help the generation of verification conditions. Auto-active is there pointed out as one of the most efficient solutions to verify real-life programs.

This observation mainly came from several projects that involved some auto-active verification. They include VCC [8], a verifier for concurrent C code where annotations are extensively used to manipulate concurrency or ownership related properties, and Spec# [1] for the object-oriented C# language. The latter strongly influenced the language Dafny [20] and its verifier, which were specifically designed to ease verification yet enabling the construction of realistic systems. It was for example used to verify the secure execution of applications [16]. In object-oriented verification, AutoProof [15], part of EVE (Eiffel Verification Environment), targets Eiffel language. It was used to verify a container library [25]. OpenJML [9], for verification of Java programs, also supports introducing lemma functions and assertions to assist an automatic proof.

In the field of functional languages, the tools Leon [4] for Scala and Why3 [14] for the WhyML language are both based on auto-active features, in particular lemma functions. The Why3 platform has for example been used to verify binary heaps [27]. It is also used as a backend for FRAMA-C/WP to enable the use of different SMT solvers.

With FRAMA-C, we target the C programming language, mostly for critical applications. In addition to VCC that we already cited, the closest tool is the GNATprove [17] verifier, used to prove programs written in SPARK 2014 [23]. The auto-active features of GNATprove are extensively used for SPARK, in particular ghost code. It was for example used to verify a red-black tree data structure [12]. Verifast [18] is a verifier for C and Java based on separation logic, which is particularly powerful to verify heap-related properties, the counterpart being an extensive use of code annotations that are not related to the application of lemma functions or ghost code.

[7]by introducing the variables and hypotheses, inverting the inductive hypothesis and automatically verifying all remaining proof goals.

The closest work to the present study has been done recently in VerKer project [30]. In this work Volkov et al. use another deductive verification plugin of FRAMA-C, called AstraVer, which is a fork of the Jessie plugin. They introduce the notion of lemma function in FRAMA-C and the corresponding verification process. The main idea consists in giving the ability to declare a function as a "lemma" with a specific syntax, the contract of the corresponding function being then automatically converted into a lemma. This work also takes advantage of the fact that the AstraVer plugin supports function variant clause **`decreases`** to prove recursion termination that makes the verifying code easier to write. They tested the approach on a set of 8 functions, all of them being related to string manipulations.

Our work continues these efforts and performs a large verification study over 23 different functions, proved almost fully automatically with FRAMA-C/WP, some of which are related to array manipulations (in `MEMB` module and *ACSL by Example*), others to linked-list manipulations, and yet others to intricate axiomatic definitions (in *ACSL by Example* again). A particular focus of our work is a detailed comparison with an earlier classic lemma-based verification of the same case studies based on our personal experience and the discussion of the difficulties encountered and possible solutions.

6 Discussion

Logic types. As pointed out by Volkov et al. [30], lemma functions in the particular case of FRAMA-C often lead to a loss of generality. Indeed, contrary to other languages that allow the use of logic types in the code, FRAMA-C does not, even in ghost code. That means that we cannot use logic types when we declare lemma functions. For example, if we compare the `occ_split` lemma (lines 2–5 of Fig. 3) and its equivalent lemma function (Fig. 4), we can notice that the bounds are no longer mathematical integers but machine integers. In all the cases we have currently verified, this is not a big problem. It often suffices to declare functions with the same types as the ones used in the program under verification (or with the largest one when multiple types exist), or to simply duplicate the lemma with the suitable type, which is not really elegant, but works.

However, the current support of ghost code by FRAMA-C limits the ability to use auto-active style depending on the needed ACSL features. For example, in a recent work, we verified the linked list module of Contiki with another formalization, using ACSL logic lists [6]. As some lemmas rely on purely logic types (the logic list), we would be unable to express them as lemma functions in FRAMA-C, and to use them in ghost code. The ACSL language allows the use of logic types in ghost code, however, adding support to FRAMA-C for this feature would require a significant implementation effort.

Lemmas with unique labels. For the particular case of lemma functions with a single label, a solution was proposed in the AstraVer tool. A function is declared as follows:

```
/*@ lemma
  requires premise_1 ; ... requires premise_N ;
  ensures conclusion ; */
void lemma_func_name(type_1 p1, ... type_M pM){ /* pure proof-carrying code */}
```

The **`lemma`** keyword indicates that the following function is a lemma, and consequently that it must be pure (which has to be proved). From this function, the plugin automatically generates an ACSL lemma supposed to be valid as soon as the lemma function is proved:

```
/*@ lemma lemma_func_name{L}: ∀ type_1 p1, ... type_M pM ;
        premise_1 ∧ ... ∧ premise_N ⇒ conclusion ; */
```

Lemmas with Multiple Labels. Our study allowed us to identify this point as an important challenge used for many considered functions. In the case of multi-label lemmas, the translation into a lemma function is still an open problem. Unlike other tools like Why3 or SPARK/GNATprove, FRAMA-C and ACSL support the use of C labels in annotations, including lemmas and predicates, and this notion is often used to specify the behavior of a section of code that involves a memory modification. Currently, we use lemma macros (cf. Section 3 and Fig. 6) to directly inject the proof when it is needed. However, this is not completely satisfactory. Indeed, first, the system has to prove the lemma again each time it is used, which is not modular, and second, it makes the proof context bigger. In complex proofs, it can make the job of SMT solvers more difficult, meaning that we need more time to get results during verification and, since SMT solvers are sensitive to the context, making the proof less robust to new versions of the VC generator.

As mentioned in the end of Section 3, to create and prove a lemma macro more easily before inserting it in a more complex function. As a workaround we proposed to use a wrapper function to prepare the lemma macro in isolation of the rest of the proof.

The first important observation is the fact that contrary to a lemma function, the wrapper function we use to build the lemma macro should not be pure. For example, it Fig. 7, the wrapper function can modify the whole array (line 5). This is not a problem: the wrapper is not meant to be called, it is a helper to build the lemma macro. If it was pure, the states before and after the call would, from a memory point of view, correspond to the same memory state, and we precisely do not want to have the same memory state to validate that our lemma macro can indeed relate to different program labels. However, the role of the `side_effect_to_L2` function is not really to produce side-effects. It is just meant to create a new memory state and to invalidate the knowledge we had at the beginning of the function and at the same time to establish that we have some new knowledge that relates previous labels and the new label we reached. The proof-carrying code is still pure.

Second, in a wrapper function, the labels are ordered, while this is not the case for the labels specified in a lemma. In our study, since the code of the proof is directly injected, it does not introduce a risk of unsoundness. However, if we want to provide a lemma function mechanism that considers multiple labels, we must ensure that the VC generation does not take into account a specific ordering, or at least force the labels to be ordered at the calling point, which can be checked using the control flow graph.

Let us sum up the difficulties on multi-label lemma functions. First, adapting ghost functions to multiple labels would probably require modifying the kernel of FRAMA-C. Second, since preconditions of a usual function contract describe only one state (**`Pre`**), taking into account multi-label premises will require a new way to generate the VCs. On the side of the proof-carrying code of the lemma function itself, we will have to consider different memory states related by some predicates. On the call site, the verification of the multi-label premises will require us to provide a suitable verification condition at each label considered by the premises of the lemma. It radically differs from the way WP currently generates those conditions.

Remaining Interactive Proofs. As two of our case studies show, with the current versions of the tools, it is not always possible to completely remove interactive proofs. The first resisting lemma is about arithmetic properties. Its proof in Coq basically proceeds in two steps. First, it relates the Coq encoding of the type of FRAMA-C logic integers to the `Z` type of the Coq standard library. Then it applies a lemma that states the same property for multiplication and division on `Z`. In this respect, it seems that this lemma could be part of a standard library of FRAMA-C lemmas. The second lemma was necessary only because WP currently does not generate the axiom we mentioned. WP could be extended to generate this kind of axiom for any inductive definition. This extension would avoid the need for interactive proof for the second resisting lemma as well.

Soundness. Soundness of axioms and inductive definitions is crucial for the verification both for the classic and auto-active approach, so we do not discuss it in this comparison. We put maximal effort in using sound statements for the case studies in this work.

7 Conclusion and Future Work

This experience report makes a step forward towards a better proof automation with FRAMA-C/WP. While the classic approach still allows to get good results with a combination of automatic and interactive proofs, we are convinced that auto-active verification can provide a usable solution for users that do not have a strong background in formal methods. While writing contracts can still remain relatively difficult, proving and calling lemma functions to deduce necessary properties seems to be easier than interactive proof since it avoids the need for a double expertise in both WP and Coq.

In this work, following the auto-active approach, we verified 23 functions from two real-life modules and a rich suite of examples proved with FRAMA-C and WP. These programs include functions that manipulate linked data-structures which are known to be hard to verify with WP. The corresponding proofs were adapted rather fast, even for examples that we had never verified by ourselves. This paper also reports on the recorded results and identified limitations. In particular, we pointed out the problem of multi-label lemmas, often needed for verification of real-life C programs involving non-trivial memory manipulations, and proposed lemma macros as a workaround.

Regarding future work, while implementing the extension proposed in AstraVer [30] is appealing, we plan to directly consider multi-label lemmas that would be more general than the AstraVer extension. On the long run, allowing logic types in ghost code seems to be another important feature to implement if we want to have more general and more easily reusable lemmas. Future work also includes creating a detailed methodology for auto-active verification, realizing an extensive user study to compare the auto-active and classic approaches, as well as experiments on real-life code verification applying the recently improved capacities of solvers to perform induction (e.g. [26, 29]).

Acknowledgment. This work was partially supported by a grant from CPER DATA and the project VESSEDIA, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453. The authors thank the FRAMA-C team for providing the tools and support, as well as Patrick Baudin, François Bobot and Loïc Correnson for fruitful discussions and advice. Many thanks to David Cok, Denis Efremov, Marieke Huisman and the anonymous referees for their helpful comments.

References

1. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the Spec# experience. Commun. ACM 54(6), 81–91 (2011)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Springer
3. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
4. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the Leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013. pp. 1:1–1:10 (2013)
5. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for lists: A critical module of contiki verified in Frama-C. In: Proc. of the 10th NASA Formal Methods Symposium (NFM 2018). LNCS, vol. 10811, pp. 37–53. Springer (2018)
6. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: Comparison of two proof approaches for a list module. In: Proceedings of the 34th Annual ACM Symposium on Applied Computing, SAC 2019. ACM (2019), to appear
7. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example (2016), https://github.com/fraunhoferfokus/acsl-by-example/blob/master/ACSL-by-Example.pdf
8. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. pp. 23–42. Springer (2009)
9. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE (2014)
10. Conchon, S., Contejean, E., Iguernelala, M.: Canonized Rewriting and Ground AC Completion Modulo Shostak Theories : Design and Implementation. Logical Methods in Computer Science (2012)
11. Dijkstra, E.W.: A constructive approach to program correctness. BIT Numerical Mathematics Springer (1968)
12. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings. pp. 68–83 (2017)
13. Dunkels, A., Gronvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: LCN 2014. IEEE (2004)
14. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: ESOP 2013. LNCS, vol. 7792. Springer (2013)
15. Furia, C.A., Nordio, M., Polikarpova, N., Tschannen, J.: Autoproof: auto-active functional verification of object-oriented programs. STTT 19(6), 697–716 (2017)
16. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014. pp. 165–181 (2014)
17. Hoang, D., Moy, Y., Wallenburg, A., Chapman, R.: SPARK 2014 and GNATprove - A competition report from builders of an industrial-strength verifying compiler. STTT 17(6), 695–707 (2015)
18. Jacobs, B., Piessens, F.: The Verifast program verifier. Tech. Rep. CW-520, KU Leuven (2008)
19. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015), http://frama-c.com

20. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR 2010. pp. 348–370. Springer (2010)
21. Leino, K.R.M., Moskal, M.: Usable auto-active verification (2010), http://fm.csl.sri.com/UV10/
22. Mangano, F., Duquennoy, S., Kosmatov, N.: A memory allocation module of Contiki formally verified with Frama-C. A case study. In: CRiSIS 2016. LNCS, vol. 10158. Springer (2016)
23. McCormick, J., Chapin, P.: Building High Integrity Applications with SPARK. Cambridge University Press (2015), https://books.google.fr/books?id=Yh9TCgAAQBAJ
24. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008. pp. 337–340 (2008)
25. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. Formal Asp. Comput. 30(5), 495–523 (2018)
26. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India. pp. 80–98 (2015)
27. Tafat, A., Marché, C.: Binary Heaps Formally Verified in Why3. Research Report RR-7780, INRIA (2011), https://hal.inria.fr/inria-00636083
28. The Coq Development Team: The Coq proof assistant, http://coq.inria.fr
29. The Imandra team: The Imandra verification tool, https://docs.imandra.ai/
30. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-C: C programs as proofs. In: Proceedings of the 2018 Ivannikov ISPRAS Open Conference (ISPRAS-2018). pp. 31–38 (2018)