A gentle introduction to C code verification using the Frama-C platform

A Return of Experience from the Concorde project WP2

Christophe Garion Gautier Hattenberger Baptiste Pollien
Pierre Roux Xavier Thirioux

ENAC - ISAE-SUPAERO - ONERA - 2021

This document presents a process to verify C programs or librairies using the Frama-C verification tool. Frama-C allows to verify different types of properties on C code, such as the absence of runtime errors or more complex functional properties. The present document is the result of our work on the verification of a mathematical library of the Paparazzi autopilot [8] and is intented to be used as a first guide to formal verification of C programs for Concorde users. In addition to provide a verification process, we also give Frama-C tips and tricks to help users and clarify some points that might not be clear in the official documentation of the tool.

Keywords: program proof, deductive methods, abstract interpretation, Frama-C

Disclaimer: This document is not a tutorial about Frama-C. Allan Blanchard's tutorial that introduces Frama-C and the WP plugin [3] is a very good starting point. The official documentation of Frama-C available on its official site is a good reference to learn more deeply about Frama-C and particularly its plugins.

Contents

1 Verifying C code with Frama-C

1.1 Frama-C platform

Frama-C [7] is an analysis tool developed by CEA and Inria allowing to formally verify C code using different techniques. It is a modular platform that supports several plugins implementing analysis techniques. For instance, the EVA plugin can be used for static analysis using abstract interpretation whereas the E-ACSL plugin focuses on dynamic analysis. In this document, we focus on *static analysis*.

The typical workflow with the Frama-C platform can be summarized in 3 steps:

1. The first step consists in *specifying* the properties that has to be verified. The specification are defined by adding ACSL annotations in the C code as special comments (see section 1.2).

2. The second step is the *generation of the abstract syntax tree (AST)* of the analyzed code by Frama-C. The generated tree also contains the previously added annotations.

3. The final step is the analysis of the AST by the plugins. Some plugins might add annotations which correspond to new properties to be verified. For instance, the RTE plugin adds assertions to verify the absence of runtime errors. Other plugins, like EVA or WP, use formal techniques to verify if the specification is respected (see section 1.3).

1.2 Specifying with ACSL

ACSL [2] is the specification language used by Frama-C to define the expected properties on C code. The ACSL annotations are added in C code as special comments. Using the ACSL specification language, different types of properties can be defined, as for instance:

- *contracts* for functions. A contract is composed of

  - *preconditions*, i.e. assertions that specify the expected state of the program when entering the function. For instance, a precondition for a function taking an integer n as parameter may be "n must be positive".

    Preconditions must be verified by the caller of the function and are considered as holding by the function developer.

    Preconditions are specified using the `requires` special comment in C code.

  - *postconditions*, i.e. assertions that specify the guaranteed state of the program after the function execution. For instance, a postcondition for a function returning a floating point value may be "the result of the function is greater than a given ϵ.

    Postconditions are specified using the `ensures` special comment.

  - *frame specifications*, i.e. all the memory elements that will be modified during the execution of the function.

    Frame specifications are specified using the `assigns` special comment.

- loop annotations among which
  - *invariants* that specify properties that hold when entering the loop, at the end of each iteration of the loop and when exiting the loop. Invariants are important properties as they are the only assertions that can be used after a loop for proving properties on a program.

    Invariants are introduced using the `loop invariant` special comment.

  - *variants*, expressions that are used to prove the termination of the loop (classically an stricly decreasing expression taking its values in $\mathbb{N}$).

    Variants are introduced using the `loop variant` special comment.

- *assertions* to specify properties that must be satisfied at a particular point of the program.

  Assertions are specified using the `assert` special comment.

Note

ACSL offers also the possibility to define types, functions, lemmas and predicates to express functional properties in order to ease specification writing. See section 2.2 for more details.

In order to verify specific properties on a code, such as functional properties, annotations must be manually added in the code. However, some plugins can automatically add annotations corresponding to the verification of specific properties, as for instance the *RTE (RunTime Errors)* plugin we used that automatically adds assertions to verify the absence of runtime errors. Runtime errors are errors appearing during the execution of a program that can provoke fatal problems (the program may abort for instance). The RTE plugin supports common runtimes errors in C such as divisions by 0, overflows or dereferencement of invalid pointers. The complete list of runtime errors taken into account by RTE can be found in the plugin documentation [11]. The section 2.1 presents the process for the verification of such runtime errors.

1.3 Verifying with EVA or WP

As briefly presented previously, Frama-C has lots of plugins (see [10] for an exhaustive list). We will focus in this document only on three of them: RTE (already presented), EVA and WP. These three plugins can be combined together for a more efficient or complete analysis. For instance, RTE adds assertions about runtime errors that can then be proved by WP or EVA, or EVA can be run before WP to help verifying assertions. This verification process is detailed in Section 2.

1.3.1 EVA

EVA (*Evolved Value Analysis*) is a plugin that uses abstract interpretation [4] to compute domains of values for each variable in the program. This plugin is very efficient for verifying the absence of numerical runtime errors such as division by 0 or overflows. EVA is able to compute accurate intervals of potential values for the program variables or intermediate results. The verification then consists in checking if the faulty values are accessible in the computed intervals.

For full information about EVA plugin, see its documentation [9].

1.3.2 WP

WP (*Weakest Precondition*) is a plugin that implements the Weakest Preconditions calculus [6] to generate verification conditions (VC) from code annotations. VCs are pure logic formulas and proving these formulas guarantees the correctness of the code. WP is interfaced with the Why3 platform [13] that is used to *discharge* VCs, i.e. prove them through theorem provers. Proofs can be done by:

- automatic solvers, like SMT solvers [1] or automatic first-order theorem provers.
- applying tactics that transform the VC if automatic provers fail to prove it (automatic provers cannot decide the truth value of all possible first-order formulas). For instance, you can apply a tactic that instantiates a bouded integer variable with all possible values and prove the property for each instantiated VC.
- manual proof assistants like Coq [14] if automatic theorem provers and tactics fail. This is of course a more difficult exercise, but is sometimes necessary to convince yourself that the VC can be proved.

The general process to conclude a proof using WP is detailed in section 3.1 and can also be found in the official documentation of the plugin [12]

2 How to verify code with Frama-C: A simple process

This section presents the verification process *we* used to verify parts of the mathematical library of the Paparazzi autopilot [8]. For each function of the library, we have defined contracts that ensure the expected properties on the behavior of the function. We focused on the verification of runtime errors (see section 2.1) and the verification of functional properties (see section 2.2) using the RTE, WP, and EVA plugins.

This section may not fit perfectly with you needs as it describes our experience with Paparazzi mathematical library verification. Depending on the code you want to verify, you may encounter situations that are not presented here. In this case, you should refer to the official documentation of Frama-C [10] and experiment by yourself. We hope however that our own experience will be helpful.

2.1 Absence of runtime errors

Runtime errors are errors that can appear during the execution of a program. A "classic" runtime error may be the dereferencing an invalid pointer, a division by 0, an overflow, a non-finite float value... These errors come generally from an incorrect implementation of a correct algorithm or borderline cases that were not taken into account.

The first goal of our verification process is to prove the absence of runtime errors. We therefore want to find the minimal preconditions for the functions of the library that will ensure that no runtime errors that will occur. This process can be summarized into 3 steps:

1. Launch the analysis on the non annotated code with Frama-C using the RTE, WP, and EVA plugins. The RTE plugin will add assertions corresponding to potential runtime errors and the plugins EVA and WP will try to verify the assertions.
2. By analyzing the assertions that are not proven by EVA nor WP, it is then possible to deduce the missing information in the contracts of the functions. It may also be necessary to specify loop variants and invariants.
3. Repeat the 2 previous steps until all assertions generated by the RTE plugin are verified.

Note

Some preconditions that have been added to verify the absence of RTE during the process may not finally be needed. In order to have minimal contracts, it thus can be necessary to consider a fourth step to clean the contract by removing these unneeded preconditions.

The RTE plugin generates assertions that can be classified into 2 main categories: assertions corresponding to possible dereferencements of invalid pointers and assertions concerning possible overflows during arithmetic operations. The verification of the first category of assertions is straightforward as it only requires to add preconditions stating that the pointers passed as parameters of the function must be valid.

The second category requires more work from the specifier. Such assertions correspond to potential overflows that can occur during the computation of arithmetic operations. These computations often involve the function parameters. Bounds for the values passed as arguments when calling the function must therefore be computed in order to guarantee the absence of overflows.

In order to understand how to determine the bounds of variables, let us take a simple addition function as an example. The function takes two parameters `a` and `b` that are both signed integers coded on 32 bits and return a 32 bits signed integer value corresponding to the addition of `a` and `b`.

```
int32_t dummy_add(int32_t a, int32_t b) {
    return a + b;
}
```

We denote by `INT_MAX` the maximum value that can be represented by 32 bits signed integers. If the function `dummy_add` is called with `a` = `b` = `INT_MAX`, then there will of course be an overflow as the result cannot be represented on a 32 bits integer (`INT_MAX` + `INT_MAX` > `INT_MAX`).

There are several ways to determine the bounds of the parameters and variables of the function depending on the information about the parameters that is known by the specifier:

- some values may already be bounded (e.g. $a \in [-1; 1]$)
- some variables may be linked (e.g. $a = 2b$)
- no information on the function parameters may be known

Considering the verification of Paparazzi mathematical library, no information about the parameters of the functions were given. We thus decided to compute equal bounds for the parameters. For instance, for the previous `dummy_add` function, we search a bound M such that:

$$\begin{aligned}\forall a, b, \qquad & -M \le a \le M \\ & \wedge -M \le b \le M \\ & \implies -INT\_MAX \le a + b \le INT\_MAX\end{aligned}$$

A simple bound can easily be deduced without further information: $M = \dfrac{INT\_MAX}{2}$. The following contract guarantees therefore the absence of overflow:

```
/*@
    requires -INT_MAX/2 <= a <= INT_MAX/2;
    requires -INT_MAX/2 <= b <= INT_MAX/2;
```

```
*/
int32_t dummy_add(int32_t a, int32_t b){
    return a + b;
}
```

Note

Such contract guarantees that there will be no overflows if and only if the contract preconditions are respected when the function is called. It is therefore necessary to verify if all the call sites of the function respect its preconditions.

Depending on the code that is analyzed, it is possible to encounter specific problems. In some cases, it may be necessary to use a special *memory model* that is more fitted for the code to be proved. For example, Frama-C offers a memory model called `ref` that is particularly adapted when function parameters are passed by reference, i.e. using pointers, as WP struggles to manage pointers with its default memory model. `ref` creates aliases for the parameters passed by references allowing WP to manage these references as “simple” variables which simplifies verification. The different memory models are presented in detail in Frama-C documentation [10] and in section 3.2.

During the verification of the mathematical library of Paparazzi, we found that the WP plugin is not really adapted when functions used pointers as parameters, even with the `ref` memory model[1]. WP does not implement alias analysis, at least to our knowledge, and the `ref` model is not always used in WP even when it is manually enabled. WP is then “overloaded” by accesses to values by reference in the code and VC generated by WP concerning the absence of overflows cannot be discharged by automatic solvers. The EVA abstract interpretation plugin completely use `ref` and could easily verify the absence of overflows even when pointers are used. On the other hand, EVA cannot verify loop variants and invariants. WP and EVA have therefore to be associated to verify the assertions added by the RTE plugin[2]. Notice that this limitation of WP has also been found by Vassil Todorov in his PhD thesis [15]. He managed to eliminate it by associating WP with Astrée [5], another C static analysis tool using abstract interpretation.

2.2 Functional verification

Functional verification aims at offering guarantees on the behavior or the result of a function. Let us present a simple example to detail what kind of properties can be verified and consider a function that takes a positive value x and computes and returns its square root. We denote by r the result of the function ($r := sqrt(x)$). Here are different properties that are expected to be verified on r:

1. The result is positive: $r >= 0$,

2. The result is smaller than the argument of the function: $r <= x$,

[1]More details are given in section 3.1

[2]The section 3.3 presents how to associate EVA and WP for verification.

3. The result squared is equal to the input: $r^2 = x$.

These three points are examples of what we call "functional properties". Depending on the algorithm used to compute r, we should expect to verify the first two properties more or less easily. The third property is mathematically correct, but is no more correct in real code using floating-point arithmetics due to potential rounding errors. For instance, we know "mathematically" that $sqrt(2)$ is an irrational number and as computers are only able to represent finite digits, the result returned by the function is an approximation of the mathematical result ($r \neq \sqrt(2)$). In this case, it is therefore be impossible to verify the property.

Rounding errors are often a difficult point for functional verification, implying that some mathematical properties cannot be proved because they are wrong with floating-point arithemetics. A solution to solve this problem with Frama-C is to use a WP arithmetic model called `real`[3]. `real` considers that floating-point operations are *mathematical* operations over *real numbers*. With this model, it is possible to verify mathematical properties on a C code, like for instance the third property expected for the square root function. This model was used to verify functional properties on floating-point functions of Paparrazi mathematical library.

Note

`real` allows to verify functional properties, but it is important to know that it is an uncorrect model of floating-point computations. Particularly, it does not respect the C programming language semantics and therefore, verification with this model does not guarantee any of the results. It is not possible to ensure that properties are verified, but at least it allows to verify that the code has a correct meaning in a mathematical sense.

Functional properties verification process can then be separated into two parts: (i) specification of the expected properties and (ii) verification of these properties.

The first step requires a formal specification language with more expressiveness. ACSL, the language used to define the contracts of functions can be used and allows to define:

- *Types*[4] that can describe mathematical types corresponding to types defined in the code, e.g. vectors, matrix, quaternions...
- *Elementary functions* that can operate on user defined types, e.g. addition of vectors.
- *Lemmas* to ensure that the mathematical definitions are correct by expressing properties, e.g. commutativity of the vectors addition.

[3]Section 3.2 presents the model and how to enable it.
[4]Section 3.4 explains how to define types in ACSL.

- *Predicates* that use elementary functions to express properties on the defined types, e.g. a predicate that states that a matrix M is a rotation matrix if and only if $M.M = I$, where I is the identity matrix.

Predicates can then be used in functions contracts to specify the functional properties that must be verified.

The second part of the process is the verification of the functional part of the functions contracts, but also of the lemmas. Lemmas must of course be verified to ensure that the functions are correct, but also can be used to facilitate the verification of the contracts. This verification step is presented in section 3.1.

3 Tips & Tricks about Frama-C

This section presents some answers to questions/problems that might arise during the verification of a library or a program using Frama-C. These answers correspond to solutions we discovered when verifying an extract of the mathematical library of Paparazzi [8].

Warning

All these problems and solutions presented are based on our experiences with *Frama-C 23.0 (Vanadium)*. In future versions, they may no longer be valid.

3.1 How to help WP to discharge VC?

The WP plugin generates Verification Conditions that cannot always be discharged by automatic provers. This section presents several techniques that might help WP to prove them.

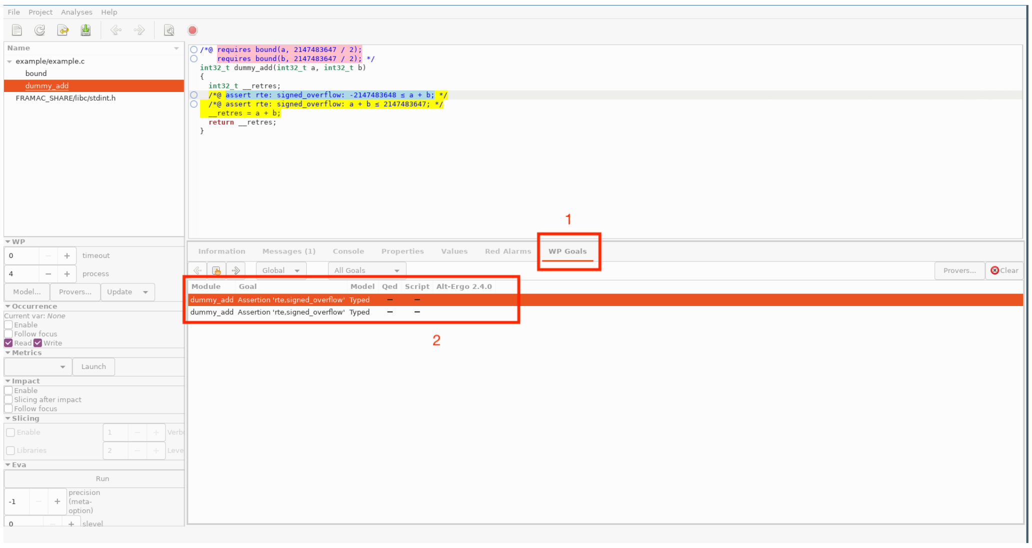
3.1.1 Using SMT solvers

WP generates VC for the Why3 platform [13]. As stated before, Why3 is a platform for deductive verification which allows using automatic SMT solvers like Alt-Ergo, Z3 or CVC4. Each SMT solver has its pros and cons: for instance, Z3 is better for solving pure SAT problems whereas Alt-Ergo has a built-in floating-point library. It is therefore advised to attempt proving a VC with all SMT solvers in parallel.

Launching Frama-C with the option `-wp-detect` displays all solvers that are installed and detected by Frama-C. In order to enable multiple automatic solvers during an analysis with WP, the command line argument `-wp-prover <PROVERS>` can be used, where `PROVERS` is the list of the provers to enable.

Note

It can also be necessary to use the command line option `-wp-timeout` to increase the solvers timeout.



```
/*@ requires bound(a, 2147483647 / 2);
    requires bound(b, 2147483647 / 2); */
int32_t dummy_add(int32_t a, int32_t b)
{
  int32_t __retres;
  /*@ assert rte: signed_overflow: -2147483648 ≤ a + b; */
  /*@ assert rte: signed_overflow: a + b ≤ 2147483647; */
  __retres = a + b;
  return __retres;
}
```

Module	Goal	Model	Qed	Script	Alt-Ergo 2.4.0
dummy_add	Assertion 'rte,signed_overflow'	Typed	–	–	
dummy_add	Assertion 'rte,signed_overflow'	Typed	–	–	

Figure 1: Select `WP goals` tab (1) and then chose a unproven goal (2).

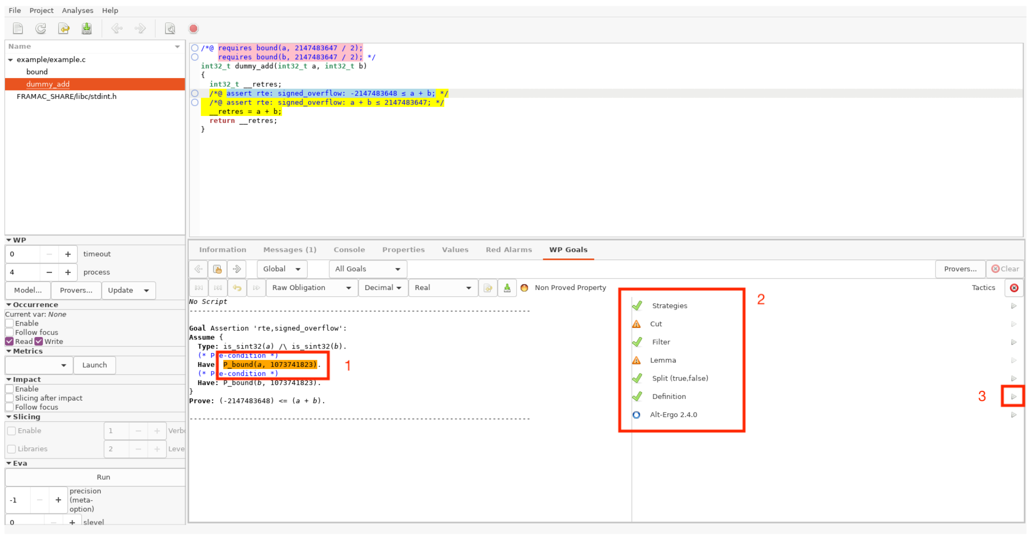
3.1.2 Apply general tactics

WP has several predefined interactive tactics that can be applied on unproven goals. The idea of tactics is to "simplify" the current goal in order to help the SMT solvers. To apply these tactics, you need to launch Frama-C graphical interface, go in the `WP goals` tab and then double click on an unproven goal (see figure 1).

The interface will show the context for the proof and the goal that need to be proved. It is then possible to click on an expression or a term and a menu will appear on the right. You will find in the menu tactics that can be applied on the selected expression or term and a green triangle button to apply them (see figure 2). When a tactic is applied, Frama-C will re-launch the solvers on the newly generated goal(s).

The complete list of WP tactics in documented in [10]. Here are some useful tactics:

- *Definition*: unfolds predicate or logic function definition. If there are many predicates/logic functions in the context/goal and if they are defined using also predicates/logic functions, just using this tactic recursively can sometimes be sufficient as it helps the solvers to discover "links" between terms by unfolding definitions.
- *Filter*: erases hypothese. This will simplify the context and may help the solvers if the hypothese is not useful for the proof.
- *Split*: separates a conjunctive goal into two simple subgoals. The solvers will be launched on both subgoals independently.
- *Shift*: turns the bit-shifting operations into mathematical divisions or multiplications. When applied on a left shift operation, an assertion is added to verify that the shifted value is positive. Indeed, if the value is negative, the shift might produce


```
/*@ requires bound(a, 2147483647 / 2);
    requires bound(b, 2147483647 / 2); */
int32_t dummy_add(int32_t a, int32_t b)
{
  int32_t __retres;
  /*@ assert rte: signed_overflow: -2147483648 ≤ a + b; */
  /*@ assert rte: signed_overflow: a + b ≤ 2147483647; */
  __retres = a + b;
  return __retres;
}
```

```
No Script
--------------------------------------------------------------------------------

Goal Assertion 'rte,signed_overflow':
Assume {
  Type: is_sint32(a) /\ is_sint32(b).
  (* Pre-condition *)
  Have: P_bound(a, 1073741823).
  (* Pre-condition *)
  Have: P_bound(b, 1073741823).
}
Prove: (-2147483648) <= (a + b).

--------------------------------------------------------------------------------
```

Figure 2: Select an expression in the unproven goal (1). There will be a list of the available tactics (2). You can then apply the chosen tactic using the play button (3).

a finite value instead of 0. If you want to disable this verification, you should call Frama-C with option `-no-warn-left-shift-negative`.

When a goal is proved using tactics, you must save the proof script using the "save" button in the graphical interface. All applied tactics will be saved in a JSON file located in the `.frama-c/wp/script` folder. When relaunching Frama-C, you must add the `tip` solver to the list of solvers to use. `tip` will automatically apply the tactics saved in the JSON file on the corresponding goals to replay the proof script.

3.1.3 Enable interactive mode to use Coq

Automatic solvers are sometimes unable to verify a goal, even with a large timeout and with the help of tactics. There are two cases:

- the goal cannot effectively be proven because it is false
- automatic solvers cannot prove the goal due to its complexity or their limitations

We only consider here the second case, i.e. that the goal requires specific reasoning unreachable by automatic SMT solvers. For instance, automatic solvers are not able to verify the following mathematical lemma, even if the context contains the trigonometric property $\forall a \in \mathbb{R}, \cos(a)^2 + \sin(a)^2 = 1$:

$$\begin{aligned}
&\forall a, b, c \in \mathbb{R}, \\
&\qquad \sin(a)^2 \cos(b)^2 \\
&\qquad + (\sin(a)\sin(b)\cos(c) - \sin(c)\cos(a))^2 \\
&\qquad + (\cos(c)\cos(a) + \sin(a)\sin(b)\sin(c))^2 = 1
\end{aligned}$$

This lemma can easily be verified by factoring and developing the formula in a specific order. To verify this type of property, Frama-C must be used in "interactive mode" in which the Coq proof assistant can be used to write the proof.

Here the different step to use Coq in Frama-C[5]:

- First, Coq must be added to the list of the provers used and then Frama-C must be launched with the graphical interface.
- In the `WP goals` tab and in the Coq column, right-click on the line of the goal you want to prove and then click on `Edit proof`.
- Coqide will open a proof file. This file contains all the context (the definitions and lemmas that have been proved previously by the solvers or Coq) and some theorems about arithmetics etc. At the end of the file, the goal is representd by a to be proved. You can edit the file and prove the theorem.
- When the proof is finished, you save it. The file will be automatically saved in the `.frama-c/wp/interactive` folder. The Frama-C graphical interface must have updated the status of the goal.
- This proof script will be automatically called the next time Frama-C is launched.

3.2 Which WP model to use?

WP has different memory models representing more or less precisely memory and arithmeric models to represent how to interpret arithmetic operations. You can choose models to have a more or less precise analysis.

By default WP uses:

- the `Typed` memory model. This model represents the memory by global arrays for every atomic type (integers, floating-point values, pointers).
- the machine arithmetic model for the computation of integer and floating-point values (respecting IEEE754 specification).

These models provide sound results and offer the best compromise between verification time and respect of the C programming language semantics. Here are some other model which can also be useful:

[5] Of course, Coq and an editor like Coqide must be installed in order to use it from Frama-C.

ref This model allows WP the detects pointer variables that are used for passing argument by reference, which will simplify the verification as previously stated in section 2.1. As this model is only valid under some memory hypotheses, WP will show warnings when it is enabled. For example, if the `ref` model is used on the following function

```
void int32_quat_comp_inv(struct Int32Quat *a2b,
                         struct Int32Quat *a2c,
                         struct Int32Quat *b2c);
```

then Frama-C should display the following warning:

```
Memory model hypotheses for function 'int32_quat_comp_inv':
  /*@
    behavior wp_typed_cast_ref_real:
    requires \valid(a2b);
    requires \separated(a2b, {a2c + (..), b2c + (..)});
  */
  void int32_quat_comp_inv(struct Int32Quat *a2b,
                           struct Int32Quat *a2c,
                           struct Int32Quat *b2c);
```

The `ref` model requires that pointers passed as parameters must be separated in memory. The previous warning shows the contract that must be verified to guarantee these hypotheses.

Good practise

It is recommended to use the command line argument `-wp-warn-memory-model` that will add and verify automatically these hypotheses in the contract.

cast This model allows WP to perform unsafe pointers casts. It is useful for the verification of programs that use cast on pointers, typically when using `void *` pointers.

real This model uses real arithmetic on floating-point variables, i.e. without no rounding errors or overflows. This model does not respect the C semantics and IEEE specification and thus can produce unsound proofs (cf. section 2.2).

The graphical interface of Frama-C offers the possibility to change the model used by WP with the button [`Model...`]. The model can also be specified using the command line arguments `-wp-model <model+...>`.

Warning

If predicates are used in contracts of function, especially in preconditions, and the `ref` model is enabled, Frama-C/WP might not use the `ref` model. It cannot verify the contracts or the assertions, as it do not implement alias analysis by default. In this case, the unfolding of predicates will have no effect. It is generally necessary to remove the use of the predicates and re-launch Frama-C.

3.3 How to associate EVA and WP for the verification of a library?

The EVA plugin using abstract interpretation can efficiently compute domains of possible values for each variable of the program, but it needs an entry point in the program. Generally, this entry point is the `main` function of a C program. Unfortunately, when verifying a library, there is generally no `main` function.

EVA has the possibility to use any function as an entry point using the command line arguments `-lib-entry -main FUNCTION`, where `FUNCTION` is the name of the function.

Functions of a library, at least those appearing int the API of the library, are independent. EVA needs thus to be launched on each of these functions. This process can be automatised with a shell script that launches the verification command for every function. The following bash script retrieves the list of functions defined in a header file.

```bash
# Get all the function names in $1 file.
# The result is written in $2 file.
get_function_names () {
    ctags -x --declaration --no-members $1  \
      | grep -v -e "#define" -e "struct" \
      | cut -f -1 -d " " > $2
}

# Get all functions declared in header file
get_function_names $HEADER $TMP1

# Pre-process the header and get all
# function names
gcc -E $INCLUDE -I. $DEFINE $HEADER > $TMP_HEADER
get_function_names $TMP_HEADER $TMP2

# Get all the function names by removing functions
# suppressed by the preprocessing to ignore
# functions that cannot be verified by Frama-C.

FUNCTIONS=$(grep -Fxf $TMP1 $TMP2)
```

To use these commands, the following variables must be defined:

- `HEADER`: the names of the header file containing all the name of the function

- `DEFINE`: the name of the constant that needs to be defined if the preprocessing macro `ifdef` is used to remove some functions that should not be verified. These functions will not appear on the computed list.

- `INCLUDE`: the path to a specific folder containing header files of the project. This variable might not be needed.

- `TMP_HEADER`,`TMP1` and `TMP2`: temporary files that can be removed.

The names of the functions declared in the `HEADER` file will be stored in `FUNCTIONS` variable.

We supply a complete script that launches the analysis on each function of a given C file: `frama-c-analysis.sh`. In order to use this script, the environment variable `FILES` should contain the list of C files that need to be verified. The command line arguments for Frama-C may be updated if needed. Set the environment variable `SMOKE` to `1` in order to enable smoke tests that verify if there are inconsistencies in the ACSL definitions or in the contract[6].

This bash script uses a Python script (`output-frama-c-analysis.py`) to nicely display the results of the analysis for every C file.

3.4 How to define user types in ACSL?

According to ACSL documentation, users should be able to define their own types for specification purposes in ACSL. Powerful construct, like inductive types, should be available, but the Vanadium version of Frama-C does not allow to define all of them. Nonetheless, it is possible to define simple types that will only be used in specifications. The following example shows how to define a 3D vector type.

First, it is necessary to define a C structure as *ghost code* (ghost code is code that is only used in specifications).

```
/*@ ghost
      struct RealVect3_s {float x;
                          float y;
                          float z;};
*/
```

Then, an ACSL type can be defined using the structure.

```
//@ type RealVect3 = struct RealVect3_s;
```

An `axiomatic` expression defining a logical function that returns an arbitrary vector and a macro `REALVECT3` that instantiates a vector are defined.

[6] Smoke tests verify if `false` is provable at different places in the code. If the assertion is proved, it means that there are contradictions in the preconditions in the contracts or in the definitions of predicates/logic functions.

```
//@ axiomatic Arbitrary_RealVect3 { logic RealVect3 empty_vect3;}

#define REALVECT3(vx, \
                  vy, \
                  vz) \
                  {{{empty_vect3 \
                  \with .x = (float) (vx)} \
                  \with .y = (float) (vy)} \
                  \with .z = (float) (vz)}
```

Finally, logic functions can be defined and used in specifications.

```
/*@
  logic RealVect3 l_Vect_of_FloatVect3(float x, float y, float z) =
      REALVECT3(x, y, z);
*/

/*@
  logic RealVect3 mult_scalar(real scal, RealVect3 v) =
      REALVECT3(scal * v.x, scal * v.y, scal * v.z);
*/
```

4 Conclusion

This document summarizes the verification process we have applied on a library of the Paparrazi autopilot using the Frama-C platform with its EVA, WP and RTE plugins. The goal of this process is to ensure the absence of runtime errors and some functional properties on non-trivial C code. As Frama-C is a complete and complex tool, this document presents tips and tricks that can help beginners to use Frama-C effectively.

Formally annotating the code with ACSL to specify contracts for functions is always a good practice, even if some assertions or lemmas might not be trivial to prove with automatic solvers or Coq. The definition of contracts can be seen as a formal specification and allows to formally define the properties that must respected by the input (preconditions) and the expected results of the functions (postconditions) and should be used as formal documentation.

Aknowledgment

This work was supported by the Defense Innovation Agency (AID) of the French Ministry of Defense (research project grant CONCORDE N 2019 65 0090004707501).

References

[1] Clark Barrett et al. “Satisfiability Modulo Theories”. In: *Handbook of Satisfiability*. Ed. by Armin Biere et al. Vol. 185. IOS Press, Feb. 2009. Chap. 26, pp. 825–885.

[2] Patrick Baudin et al. *ACSL: ANSI/ISO C specification language*. Version 1.17. 2021. URL: `https://www.frama-c.com/download/acsl.pdf`.

[3] Allan Blanchard. *Introduction à la preuve de programmes C avec Frama-C et son greffon WP*. 2020. URL: `https://allan-blanchard.fr/publis/frama-c-wp-tutoriel-fr.pdf`.

[4] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *POPL*. 1977, pp. 238–252.

[5] Patrick Cousot et al. “Why does Astrée scale up?” In: *Formal Methods in System Design* 35.3 (2009), pp. 229–264.

[6] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (1975), pp. 453–457.

[7] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal aspects of computing* 27 (2015), pp. 573–609. DOI: `10.1007/s00165-014-0326-7`.

[8] Baptiste Pollien et al. “Verifying the Mathematical Library of an UAV Autopilot with Frama-C”. In: *26th International Conference on Formal Methods for Industrial Critical Systems - FMICS 2021*. Paris, France, Aug. 2021. DOI: `10.1007/978-3-030-85248-1\_10`. URL: `https://hal.archives-ouvertes.fr/hal-03344191`.

[9] The Frama-C development team. *EVA documentation*. 2021. URL: `https://frama-c.com/download/frama-c-eva-manual.pdf`.

[10] The Frama-C development team. *Frama-C documentation*. 2021. URL: `https://www.frama-c.com/html/documentation.html`.

[11] The Frama-C development team. *RTE documentation*. 2021. URL: `https://frama-c.com/download/frama-c-rte-manual.pdf`.

[12] The Frama-C development team. *WP documentation*. 2021. URL: `https://frama-c.com/download/frama-c-wp-manual.pdf`.

[13] The Why3 Development Team. *Why3 Documentation*. 2020. URL: `http://why3.lri.fr/manual.pdf`.

[14] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*. Version 8.8.2. Apr. 2018. DOI: `10.5281/zenodo.1219885`. URL: `https://hal.inria.fr/hal-01954564`.

[15] Vassil Todorov. “Automotive embedded software design using formal methods”. PhD Thesis. Université Paris-Saclay, Dec. 2020. URL: `https://tel.archives-ouvertes.fr/tel-03082647`.