Verifying the Mathematical Library of a UAV Autopilot with Frama-C*

Baptiste Pollien[1], Christophe Garion[1], Gautier Hattenberger[2], Pierre Roux[3], and Xavier Thirioux[1]

[1] ISAE-SUPAERO, Université de Toulouse, France
[2] ENAC, Toulouse, France
[3] ONERA, Toulouse, France

Abstract. Ensuring safety of critical systems is crucial and is often attained by extensive testing of the system. Formal methods are now commonly accepted as powerful tools to obtain guarantees on such systems, even if it is generally not possible to formally prove the safety and correctness of the whole system. This paper presents an ongoing work on the formal verification of the Paparazzi UAV autopilot using the Frama-C verification platform. We focus on a Paparazzi mathematical library providing different UAV state representations and associated conversion functions and manage to prove the absence of runtime errors in the library and some interesting functional properties on floating-point conversion functions.

Keywords: proof of program · critical systems · deductive methods · abstract interpretation

1 Introduction

Formal methods are verification techniques based on mathematics which facilitate the formal verification of properties of hardware, software or models. They are nowadays widely accepted as an efficient complement to testing, particularly for critical systems, see for instance the DO-330 supplement to DO-178C. There are many formal methods and they can be distinguished by the properties they can help to verify, the efforts required to specify the system in order to use the verification tools, or their automation level. For instance, *abstract interpretation* is often used to prove the absence of runtime errors and is an automatic tool. *Deductive verification* is another tool that can be used to prove more complex properties, like correctness of a program given a formal specification, and generally uses automated solvers, though sometimes needs to use a proof assistant and requires human intervention.

The goal of this ongoing project is to review different formal verification techniques to define an analysis process taking advantage of these tools in order

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

to verify properties of an UAV autopilot. This analysis process is applied on the Paparazzi autopilot developed at ENAC and implemented in the C programming language [6,9].

Frama-C [7] is a C code analysis tool facilitating formal verification. Verification with Frama-C require the addition of annotations in the C code as special comments to specify the expected properties: definition of contracts for functions (*preconditions*, *postconditions* and frame specification (i.e., all the memory elements that will be modified during the execution of the function), and the definition of *invariants* and *variants* for loops and assertions. Frama-C has many plugins but three were of particular interest: WP (*Weakest Precondition*) which uses the weakest preconditions calculus, RTE (*RunTime Errors*) that automatically add assertions to verify the absence of runtime errors, and EVA (*Evolved Value Analysis*) using abstract interpretation to compute sets of possible values for each variable of the program.

Verification of the Paparazzi autopilot is currently done using the Frama-C platform (version 23.0 Vanadium) and mainly automatic provers[4]. We focused on verifying a mathematical library presented in section 2. Section 3 details the analysis concerning the absence of runtime errors. The second part of the analysis covers the verification of functional properties for some state representation transformation functions. Section 4 details the verification process using only automatic provers. As automatic provers were not able to prove some of these functions, section 5 presents how we proved such functions using the interactive prover Coq [12]. Finally, section 6 gives a conclusion and some perspectives.

2 The Paparazzi autopilot

Paparazzi [6] is an open-source autopilot under GPL license developed at ENAC since 2003. Paparazzi supports various types of drones and permits the control of several of them simultaneously. Paparazzi has also various built-in modes and offers the ability to create personalized flight plans. The mathematical library studied here provides functions converting different representations of vector rotations as rotation matrices, Euler angles, or quaternions. It also defines elementary operations on these representations. This library is written in the C programming language and each function is available in three versions: one using `double` values, another one with `float` values and the last one using `int` values to represent fixed point values.

3 Proving the absence of runtime errors

The studied library defines C structures for the different representations (rotation matrices, quaternions, vectors, etc). The library functions take only pointers

[4] Complete specified code of Paparazzi, tools versions, and installation instructions are available on https://gitlab.isae-supaero.fr/b.pollien/paparazzi-frama-c/-/tree/fmics-2021.

on such structures as inputs and always return pointers. Preconditions ensuring the validity of pointers have been added in the contracts as the functions are not designed to work with invalid pointers. It has also been necessary to specify which variables will be modified during the execution of the function. Finally, invariants on `for` loop have been added to help the provers to ensure the absence of runtime errors.

The WP plugin of Frama-C offers different models of arithmetic that take into account more or less precisely C semantics. The verification of the `int` flavor of the functions was made using 32-bits integer arithmetic with overflows. When using the RTE plugin to verify the absence of runtime errors, assertions are automatically added to check that there is no overflow for each arithmetical operation. To verify this, each function was manually analyzed to determine the maximum possible value for the different variables. When bounds of the variables have been determined, they were added as preconditions in the function contracts. Unfortunately, WP associated with automatic provers is not able to verify these new contracts. Even if the complete memory separation of structures used in a function is specified as precondition, the solvers are unable to prove that the modification of a field in a structure does not change any other part of the memory.

To overcome this problem, we decided to associate the EVA plugin to WP. EVA has no issue dealing with pointers nor aliasing and is able to compute accurate intervals of possible values for each variable. The result is then passed to WP by Frama-C, which makes it easier to conclude some proofs. This WP limitation when pointers are extensively used as input and output parameters was also found by Vassil Todorov during his PhD thesis [13]. He also used a static analysis tool using abstract interpretation, Astrée [3], to solve the same problem. To conclude, the association of EVA and WP enables the verification of absence of runtime errors for the functions using `int` values.

WP has also an arithmetical model `real` for real arithmetic. We decided to use this model for the verification of the library functions working on floating-point values. Using the same precondition used for the `int` version of the functions, and such a model permits us to verify the absence of division by zero and that real variables do not take the `NaN` value. To perform these verifications, it was only necessary to add as preconditions the fact that each pointer refers to a valid address. The absence of these two kind of runtime errors as well as the termination of the functions have been proven, using WP and EVA, for the `float` and `double` versions of the library. Unfortunately, our verification does not offer any guarantee on the risk of floating-point overflow or on rounding errors. Moreover, the properties proven for the `real` model can only serve as an hint, but not a guarantee, that they hold for floating-point values. However, this model was particularly useful to verify functional properties as presented in section 4. Indeed, even if the model is semantically incorrect and we cannot get functional guarantees during execution, it permits at least to verify that the code is correct in the mathematical sense.

4 Functional verification using automatic provers

The goal of functional verification is to ensure properties about the behavior of functions. We will first focus here on the function `float_rmat_of_quat` to explain the process used. This function takes a normalized quaternion as input and returns the corresponding rotation matrix.

In order to specify the functional properties of such a function, types and predicates have been defined in the logic provided by ACSL [1], the language used to express Frama-C annotations. We defined types for matrices and quaternions, as well as elementary algebraic operations. We specified lemmas and then verified them to ensure that these operations are correct (e.g., we verified that matrix transposition is idempotent). Then, a logical function that converts a quaternion to a rotation matrix has been defined independently of the C code from the library. This function is based on the mathematical equation that expresses the conversion of a quaternion to a rotation matrix [5,8]. In the following, we will note `rmat_of_quat` the function that represents this conversion. `rmat_of_quat`'s semantics is expressed in ACSL as a mathematical, model-based specification. This function takes as parameter a unitary quaternion q and returns a rotation matrix. Frama-C is able to verify automatically that for any given unitary quaternion q, the rotation matrix computed by `rmat_of_quat` corresponds to the same rotation as described by the quaternion. Verification of this property has required to verify the following lemma: $\forall q \in \mathbb{H}, v \in \mathbb{R}^3,\ q(0,v)q^* = (0, \texttt{rmat\_of\_quat}(q).v)$. This lemma states that given a quaternion q and a vector v, applying the rotation with the quaternion q on vector v is equivalent to applying the rotation matrix obtained from q by `rmat_of_quat` on v.

The contract for the function `float_rmat_of_quat` has then been established using these ACSL functions. Assuming that the quaternion passed as parameter is normalized, we wanted to verify two functional properties. The first one is that the returned matrix does indeed correspond to the conversion of the quaternion passed as a parameter: our post-condition verifies that the rotation matrix generated by the C code is equal to the rotation matrix generated by our logical function `rmat_of_quat`. As presented in the previous section, we use the WP `real` model for the verification of this property, thus ignoring the differences in the results between the C version and the mathematical version which could have been introduced by rounding errors. The second verified property is that the generated matrix is indeed a rotation matrix, i.e. the transpose of the matrix is its inverse, and its determinant is equal to 1.

Despite the use of the `real` arithmetic model, WP could not verify this contract. It was therefore necessary to manually review the code. We noticed that the C code used a constant `M_SQRT2` to represent $\sqrt{2}$. By analyzing the calculations done in the code, we realized that the constant `M_SQRT2` was every time multiplied by itself. We therefore suggested a code modification that replaces `M_SQRT2` * `M_SQRT2` by 2. This modification does not change the number of multiplications in the C code but permits to reduce the rounding errors propagated by the function. With this code change and the arithmetic model `real`, WP verifies the contract of the function `float_rmat_of_quat`.

5 Functional verification using interactive provers

The same verification has been attempted for the inverse function `float_quat_of_rmat` that converts a rotation matrix into a quaternion. There are different equations to perform this conversion in the literature but we use the four formulae using Shepperd's method [11,10]. These equations are directly deduced from the formula that converts a quaternion into a rotation matrix and are defined according to the diagonal values of the rotation matrix: one is defined when the trace is strictly positive, the three other ones are defined when the trace is negative and correspond to the possible choices for the greatest element of the diagonal of the matrix. We defined each of these formulae by an ACSL logical function. When defining postconditions of C functions, we uses ACSL `behavior` feature to specify one sub-contract per Shepperd's case, specifying as preconditions in the sub-contract the conditions for which the corresponding Shepperd's function is defined. For instance, we defined a behavior that requires as precondition that the trace of the input matrix is positive. These behaviors then ensure, as postconditions, that the quaternion computed by the C function is equal to the quaternion computed using the corresponding logical function. Another feature offered by Frama-C is the possibility to verify that the behaviors are disjoint and complete. I.e. for every input matrix, there is one and only one behavior such that its preconditions are fulfilled by the matrix. With this contract, we were able to verify that the function returns a quaternion that corresponds to the same rotation as the matrix used as input.

Let us denote by `quat_of_rmat` the mathematical function that returns the quaternion corresponding to a given rotation matrix. Let us consider here the case where the rotation matrix used as input has a positive trace. Verifying that `quat_of_rmat` is correct in this case is equivalent to verifying that the property described on the following lemma holds: $\forall R \forall q \ ||q|| = 1 \wedge Tr(R) > 0 \rightarrow (R = \texttt{rmat\_of\_quat}(q) \leftrightarrow q = \texttt{quat\_of\_rmat}(R))$. This lemma can be read as follows: for all rotation matrices R with a positive trace and for all unitary quaternions q, R represents the rotation matrix obtained from $\texttt{rmat\_of\_quat}(q)$, if and only if the function `quat_of_rmat` will return q when applied to R. It has then been translated into an ACSL lemma, as well as the similar equations resulting from the three other cases.

Unfortunately, Frama-C is not able to prove these lemmas using only automatic SMT solvers, even after extending the timeout value considerably. The proof requires specific transformations, such as factorization, that the solvers might not be able to find. We therefore had to use the interactive mode of WP. This mode generates incomplete proof scripts for each unproven goal. The scripts contain all the definitions and lemmas that have already been proved by Frama-C and the solvers. The theorem corresponding to an unproven goal needs to be verified with some interactive prover where in our case, we use Coq [12]. The implication that if R is obtained from $\texttt{rmat\_of\_quat}(q)$, then the function `quat_of_rmat` will return q has been verified with Coq for the four lemmas. The reverse implication has not been proved yet. However, by considering the verification of the function `rmat_of_quat`, this proof is sufficient to guarantee that the

result of the function `quat_of_rmat` describe the same rotation than the input matrix.

We also wanted to verify the function implementing the conversion from the Euler representation of a rotation to a rotation matrix. In the library, there are two functions, `float_rmat_of_eulers_321` and `float_rmat_of_eulers_312`, that implement this conversion. These two functions differ on the order of Euler angles (given the $(\boldsymbol{z}, \boldsymbol{y}, \boldsymbol{x})$ axis for the `321` function and given the $(\boldsymbol{z}, \boldsymbol{x}, \boldsymbol{y})$ axis for the `312` function). The contracts defined for these functions ensure that the matrix must be special orthogonal, i.e., a rotation matrix. In order to verify these contracts, we started using only automatic SMT solvers. The first problem we faced was that the code for the conversion uses the `cosf` and `sinf` trigonometric functions from the C standard library. Frama-C equips these built-in functions with contracts, but these contracts do not provide enough information. We decided to add an hypothesis in the contract stating that the result of these functions was equal to the result obtained with the corresponding mathematical trigonometric function of ACSL (defined by `\cos` and `\sin`). This hypothesis might not be correct. However, as we use the `real` model, this hypothesis permits to use properties of trignometric functions (for instance $\forall a \in \mathbb{R},\ \cos a^2 + \sin a^2 = 1$).

Unfortunately, Frama-C was not able to prove the postcondition of the conversion functions, even with this hypothesis and WP tactics. WP tactics are a feature offered by WP that applies basic transformations on the goals to simplify them in order to discharge the solvers (for instance, definitions can be unfolded or goals splitted into subgoals). Even by using tactics, there were remaining unproven subgoals. Instead of using Coq to verify the whole postcondition, we decided to define generic lemmas in ACSL which correspond to the subgoals unproven by SMT solvers and verify them in Coq. Such a lemma is for instance $\forall a, b, c \in \mathbb{R}\ \sin a^2 * \cos b^2 + (\sin a * \sin b * \cos c - \sin c * \cos a)^2 + (\cos c * \cos a + \sin a * \sin b * \sin c)^2 = 1$, which can easily be proved by hand using factorization and properties of trigonometric functions.

6 Conclusion

We have presented in this paper an ongoing work on formal verification of a mathematical library of the open-source autopilot Paparazzi. We have mainly focused on the verification of the absence of runtime errors, but also have proven interesting properties of rather complex functions.

In future work, we plan on completing the proof remaining presented in section 5. Some functional properties of other functions from the same mathematical library of Paparazzi should also be verified. We want especially to focus on verifying rounding errors, and therefore do not use WP `real` model but rather using a model that represents accurately the floating-point numbers. We should also compare our approach to *autoactive proofs*, where interactive provers are not used, but SMT solvers are guided by assertions inserted by developers to help the provers [2,4]. Finally, we plan to tackle formal verification of the Paparazzi flight plan generator.

References

1. Baudin, P., Filliâtre, J., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008)
2. Blanchard, A., Loulergue, F., Kosmatov, N.: Towards Full Proof Automation in Frama-C Using Auto-active Verification. In: NFM 2019 - 11th Annual NASA Formal Methods Symposium. pp. 88–105. Springer, Houston, TX, United States (May 2019). https://doi.org/10.1007/978-3-030-20652-9_6, https://hal.inria.fr/hal-02317055
3. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does Astrée scale up? Formal Methods in System Design **35**(3), 229–264 (2009)
4. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NASA Formal Methods. pp. 68–83. Springer International Publishing, Cham (2017)
5. Grubin, C.: Derivation of the quaternion scheme via the Euler axis and angle. Journal of Spacecraft and Rockets **7**(10), 1261–1263 (1970). https://doi.org/10.2514/3.30149, https://doi.org/10.2514/3.30149
6. Hattenberger, G., Bronz, M., Gorraz, M.: Using the Paparazzi UAV System for Scientific Research. In: IMAV 2014, International Micro Air Vehicle Conference and Competition 2014. pp. pp 247–252. Delft, Netherlands (Aug 2014). https://doi.org/10.4233/uuid:b38fbdb7-e6bd-440d-93be-f7dd1457be60, https://hal-enac.archives-ouvertes.fr/hal-01059642
7. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal aspects of computing **27**, 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
8. Klumpp, A.R.: Singularity-free extraction of a quaternion from a direction-cosine matrix. Journal of Spacecraft and Rockets **13**(12), 754–755 (1976). https://doi.org/10.2514/3.27947, https://doi.org/10.2514/3.27947
9. Paparazzi UAV Team: Paparazzi – the free autopilot (2021), https://paparazzi-uav.readthedocs.io/en/latest/
10. Sarabandi, S., Thomas, F.: Accurate computation of quaternions from rotation matrices. In: Lenarcic, J., Parenti-Castelli, V. (eds.) Advances in Robot Kinematics 2018. pp. 39–46. Springer International Publishing, Cham (2019)
11. Shepperd, S.W.: Quaternion from rotation matrix. Journal of Guidance and Control **1**(3), 223–224 (1978). https://doi.org/10.2514/3.55767b, https://doi.org/10.2514/3.55767b
12. The Coq Development Team: The Coq Proof Assistant, version 8.8.0 (Apr 2018). https://doi.org/10.5281/zenodo.1219885, https://hal.inria.fr/hal-01954564
13. Todorov, V.: Automotive embedded software design using formal methods. Phd thesis, Université Paris-Saclay (Dec 2020), https://tel.archives-ouvertes.fr/tel-03082647