Deductive Verification with Relational Properties

Lionel Blatter\textsuperscript{1}, Nikolai Kosmatov\textsuperscript{1}, Pascale Le Gall\textsuperscript{2}, and Virgile Prevosto\textsuperscript{1}

\textsuperscript{1} CEA, LIST, Software Reliability and Security Laboratory, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

\textsuperscript{2} Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 92295 Châtenay-Malabry France
firstname.lastname@centralesupelec.fr

Abstract. Modular deductive verification provides a sound and powerful technique to establish that any call to a given function respects its given specification. However, relational properties, i.e., properties relating several function calls, are not supported. This short paper presents an original automated technique for specification and verification of such properties using the classic deductive verification approach. We illustrate the proposed technique by comprehensive examples and present its implementation as a FRAMA-C plugin, named RPP.

Keywords: deductive verification, relational properties, specification, FRAMA-C.

1 Introduction

Deductive verification is a powerful formal verification technique that allows the user to prove that a given program respects its formal specification. In modular deductive verification, functions are specified and verified separately, following the concept of design-by-contract [16] and generally using weakest-precondition calculus [9]. For a given function \( f \), any individual call to \( f \) can be proven to respect the contract of \( f \), that is, basically an implication: if the given precondition is true before the call, the given postcondition is true after it. This proof relies on contracts of called functions (that, in turn, can be proven separately or just assumed). Examples of deductive verification tools include ESC-Java [4], Spec# [1], Dafny [15], OpenJML [6], Why/Why3 [10], Jessie and WP plugins of FRAMA-C [14], Krakatoa [9], Spark2014 [13], Verifast [12].

Motivation. However, some kinds of properties cannot be easily specified and verified as function contracts. Indeed, functions are often linked by algebraic specifications defining their relationships. It is frequently necessary to express a property that invokes several functions or relates the results of several calls to the same function for different arguments. We call them relational properties.

The necessity to deal with relational properties has been faced in various verification projects. Recent work [3] reports on verification of continuous monotonic functions in an industrial case study on smart sensor software. The authors write: “After reviewing around twenty possible code analysis tools, we decided to use FRAMA-C, which fulfilled all our requirements (apart from the specifications involving the comparison of function calls).” The relational property in question is monotonicity of a function (e.g., \( x \leq y \Rightarrow f(x) \leq f(y) \)). To deal with it in FRAMA-C, [3] used a workaround with a separate verification of an additional, manually created wrapper function simulating the calls to be compared. This workaround prevented better automation of the verification.
Relational properties can often be useful to give an expressive specification of library functions or hardware-supported functions, when the source code is not available and workarounds become even more delicate or impossible. For instance, in the PISCO project\(^3\), an industrial case study on verification of software using hardware-provided cryptographic primitives (PKCS#11 standard) required tying together different functions with properties such as \(\text{Decrypt(Encrypt(Msg, PrivKey), PubKey)} = Msg\). Other examples include properties of data structures, such as matrix transformations (e.g., \(\text{det}(A) = \text{det}(A^T)\)), the specification of Push and Pop over a stack \(^5\), or parallel program specification (e.g., \(\text{map(append(l_1, l_2)) = append(map(l_1), map(l_2))}\) in the MapReduce approach). A subclass of relational properties, metamorphic properties, relating multiple executions of the same function \(^{11}\), are also used in a different context in order to address the oracle problem in software testing \(^{17}\).

Function contracts and classic deductive verification methods focusing on one function call at a time are not suitable for specification and verification of relational properties. Possible workarounds reduce the level of automation, can be error-prone and do not provide a complete automated link between three key components: (i) the property specification, (ii) its proof, and (iii) its usage as a hypothesis in other proofs. Thus, the lack of support for relational properties can be a major obstacle to a wider application of deductive verification in academic and industrial projects. The purpose of this work is to address this problem in the context of the ACSL specification language \(^2\), FRAMA-C and its deductive verification plugin WP \(^{14}\).

**Contributions.** This work proposes a convenient solution for support of relational properties using deductive verification with FRAMA-C. It includes:

- a new specification mechanism to formally express a relational property in ACSL;
- a fully-automated transformation into ACSL-annotated C code with an axiomatic definition that allow the user both to prove the property and to use it as a hypothesis;
- this technique naturally benefits from existing deductive verification tools;
- a proof-of-concept implementation of this approach in a FRAMA-C plugin RPP with a sound integration of proof statuses of relational properties;
- its application for verification of several examples with relational properties.

2 **The Method and the Tool**

2.1 **Specification and Preprocessing of a Relational Property**

The proposed solution is designed and implemented on top of FRAMA-C \(^{14}\), a framework for analysis of C code developed at CEA LIST. FRAMA-C offers a specification language, called ACSL \(^2\), and a deductive verification plugin, WP, that allow the user to specify the desired program properties as function contracts and to prove them. A classic ACSL function contract may include a precondition (\texttt{requires} clause stating a property supposed to hold before the function call) and a postcondition (\texttt{ensures} clause that should be proved after the call), as well as a frame rule (\texttt{assigns} clause indicating which parts of the global program state the function is allowed to modify). An assertion (\texttt{assert} clause) can also specify a local property at any function statement.

[^3]: \(\url{http://www.systematic-paris-region.org/en/projets/pisco}\)
/* @assigns \nothing; relational R1: \forall int x1,x2; x1 < x2 \Rightarrow \call(f1,x1) < \call(f1,x2); */

int f1(int x){
    return x + 1;
}

/* @assigns \nothing; relational R2: \forall int x1, x2; x1 < x2 \Rightarrow \call(f2,x1) < \call(f2,x2); relational R3: \forall int k; \call(f1,k) < \call(f2,k); */

int f2(int y){
    return y + 2;
}

Fig. 1: (a) Two monotonic functions f1, f2 with three relational properties (file f.c), and (b) extract of their transformation by RPP for deductive verification

**Specification.** To specify a relational property as part of a function contract, we propose an extension of ACSL specification language with a new clause, relational. To refer to several function calls in such a property, we introduce a new construct \call(f, <args>) to indicate the value returned by the call f(<args>) to f with arguments <args>. \call can be used recursively, i.e. a parameter of a called function can be the result of another function call. For example, properties R1, R2 at lines 2–3, 10–11 of Fig. 1a specify monotonicity of functions f1, f2, while lines 5–6 of Fig. 1b specify correct encryption-decryption.

**Preprocessing and Proof Status Propagation.** Since this new syntax is not supported by classic deductive verification tools, we have designed a code transformation allowing the user to prove the property with one of these tools. We illustrate the transformation for function f1 and its monotonicity property (see Fig. 1b).

The transformation result (Fig. 1b) consists of three parts. First, a new function, called wrapper, is generated. It inlines the function calls occurring in the relational property, records their results in local variables and states an assertion equivalent to the relational property (lines 1–7 in Fig. 1b). The proof of such an assertion is possible with a classic deductive verification tool (WProve it in this example).

The wrapper function is inspired by the workaround proposed in [3] and allows a provable reformulation of the relational property. However, it is not sufficient if we need to use the relational property as a hypothesis in other proofs and to make their support fully automatic and transparent for the user.

For this purpose, we generate an axiomatic definition (cf. axiomatic section at lines 9–14) to give a logical reformulation of the relational property as a lemma (cf. lines 11–12). This logical formulation can be used in over proofs (as we illustrate below). Lemmas can refer to several function calls, but only for logic functions. Therefore,
#include "f.c"

/*@
relational Rg: ∀ int x1,x2; x1 < x2 ⇒ call(g,x1) < call(g,x2);
*/

int g(int x){
  return f1(x)+f2(x);
}

/*@
relational Rh: ∀ int x1,x2; x1 < x2 ⇒ call(h,x1) < call(h,x2);
*/

int h(int x){
  return f1(f2(x));
}

/*@
assert x1 < x2 ⇒ tmp1 < tmp2;*/

void relational_wrapper(int x1,int x2){
  int tmp1 = 0;
  int tmp2 = 0;
  tmp1 = f1(x1) + f2(x1); // g(x1)
  tmp2 = f1(x2) + f2(x2); // g(x2)
  /*@ assert x1 < x2 ⇒ tmp1 < tmp2;*/
}

/*@ axiomatic Relational_axiom{
logic int g_acsl(int x);
lemma relational_lemma: ∀ int x,y;
  x < y ⇒ g_acsl(x) < g_acsl(y);
}*/

int g(int x){
  return x + 1;
}

... // similar for h

Fig. 2: (a) Two monotonic functions g, h with two relational properties, and (b) extract of their transformation by RPP for deductive verification

a logic counterpart (with _acsl suffix) is declared for each C function involved in a relational property (cf. line 10). The ACSL function is partially specified via lemmas corresponding to the relational properties of the original C function. Note that the correspondence between f and f_acsl implies that f does not access global memory (neither for writing nor for reading). Indeed, since f_acsl is a pure logic function, it has no side effect and its result only depends on its parameters. Extending our approach for this case can rely on assigns...\from... clauses [2] for adding to the ACSL function parameters representing the relevant parts of the program state. This extension is left as future work.

Finally, to create a bridge between the C function and its logic counterpart, we add a postcondition (an ensures clause, placed in a separate behavior for readability) to state that they always return the same result (cf. line 18 relating f1 and f1_acsl).

To make the proposed solution as transparent as possible for the user and to ensure automatic propagation of proof statuses in the FRAMA-C property database [7], two additional rules are necessary. First, the postconditions making the link between C functions and their associated logic counterparts are always supposed valid (so the clause of line 18 is declared as valid). Second, the logic reformulation of a relational property in a lemma (lines 11–12) is declared valid as soon as the assertion (line 6) at the end of the wrapper function is proved.

Fig. 2b gives another example of transformation for function g of Fig. 2a.

2.2 Implementation and Illustrative Examples

Implementation. A proof-of-concept implementation of the proposed technique has been realized in a FRAMA-C plugin RPP (Relational Property Prover). RPP works like

---

4 Technically, a special “valid under condition” status is used in this case in FRAMA-C.
```c
int Crypt(int m, int key) {
    return m + key;
}

int Decrypt(int m, int key) {
    return m - key;
}

int Test_crypt(int m, int key) {
    int cryp, decryp;
    cryp = Crypt(m, key);
    decryp = Decrypt(cryp, key);
    return decryp;
}
```

Fig. 3: Relational property for two cryptographic functions (implemented here by Caesar cipher) used to prove the postcondition of function Test_crypt.

A preprocessor for WP: after its execution on a project containing relational properties, the proof on the generated code proceeds like any other proof with WP [14]: proof obligations are generated and can be either discharged automatically by automatic theorem provers (e.g. Alt-Ergo, CVC3, CVC4, Z3) or proven interactively (e.g. in Coq).

Thanks to the proposed code transformation no significant modification was required in FRAMA-C and WP. RPP currently supports relational properties of the form

$$\forall \langle \text{args}_1 \rangle, \ldots, \forall \langle \text{args}_N \rangle, P(\langle \text{call}(f_1, \langle \text{args}_1 \rangle)\ldots, \langle \text{call}(f_n, \langle \text{args}_N \rangle)\rangle)$$

for an arbitrary predicate $P$ invoking $N \geq 1$ calls of non-recursive functions without side effects and complex data structures.

**Illustrative Examples.** After preprocessing with RPP, FRAMA-C/WP automatically validates properties R1-R3 of Fig. 1b by proving the assertions in the generated wrapper functions and by propagating proof statuses.

To show how relational properties can be used in another proof, consider properties Rg, Rh of Fig. 2b for slightly more complex functions (inspired by [3]) whose proof needs to use properties R1, R2. Thanks to their reformulation as lemmas and to the link between logic and C functions (cf. lines 10–11, 18 of Fig. 1b for $f_1$), WP automatically proves the assertion at line 6 of Fig. 2b and validates property Rg as proven. The proof for Rh is similar.

Fig 3 shows another simplified, but representative example (inspired by the PISCO project) where the relational property is used to prove the postcondition at line 11. Notice that in examples of Fig. 2 and 3 functions $f_1, f_2, Crypt, Decrypt$ can be undefined since only their (relational) specification is required, which is suitable for specification of library or hardware-provided functions that were not specifiable without relational properties.

The RPP tool has also been successfully tested on several other examples such as squeeze lemma condition (i.e. $\forall x, f_1(x) \leq f_2(x) \leq f_3(x)$), median function properties (e.g. $\forall a, b, c, Med(a, b, c) = Med(a, c, b)$), properties of determinant for matrices of order 2 and 3 (e.g. $\det(A) = \det(A^T)$), etc.

### 3 Conclusion and Future Work

We proposed a novel technique for specification and proof of relational properties for C programs in FRAMA-C. We implemented it in a FRAMA-C plugin RPP and illustrated its capacity to treat a large range of examples coming from various industrial and academic projects that were suffering from the incapacity to express relational properties.
One benefit of this approach is its capacity to rely on sound and mature verification tools like FRAMA-C/WP, thus allowing for automatic or interactive proof from the specified code. Thanks to an elegant transformation into auxiliary C code and logic definitions accompanied by a property status propagation, the user can treat complex relational properties and observe the results in a convenient and fully automatic manner. Another key benefit is that this approach is suitable for verification of programs relying on library or hardware-provided functions whose source code is not available.

This technique opens several exciting work perspectives. One of them is extending the tool to support complex data structures and functions with side-effects. Other research directions include support of recursive functions, studying other variants of generated code (e.g. avoiding function inlining in some cases), as well as further experiments on real-life programs.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: CASSIS 2004
2. 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](http://frama-c.com/acsl.html)
3. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: A case study. In: ISSRE 2013
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. STTT 7(3), 212–232 (2005)
5. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example (2016). [https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample/blob/master/ACSL-by-example.pdf](https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample/blob/master/ACSL-by-example.pdf)
6. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE 2014
7. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: FMICS (2012)
8. Dijkstra, E.W.: A constructive approach to program correctness. BIT Numerical Mathematics 8(1), 42–64 (1968)
9. Filliâtre, J.C., Marché, C.: The why/krakatoa/caduceus platform for deductive program verification. In: CAV 2007
10. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: ESOP 2013
11. Hui, Z.W., Huang, S.: A formal model for metamorphic relation decomposition. In: WCSE 2013
12. Jacobs, B., Piessens, F.: The Verifast Program Verifier. Tech. Rep. CW-520, KU Leuven (2008)
13. Kanig, J., Chapman, R., Comar, C., Guittion, J., Moy, Y., Rees, E.: Explicit assumptions - a prenup for marrying static and dynamic program verification. In: TAP 2014
14. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Framac: A software analysis perspective. Formal Asp. Comput. 27(3), 573–609 (2015). [http://frama-c.com](http://frama-c.com)
15. Leino, K.R.M., Wüstholz, V.: The Dafny integrated development environment. In: F-IDE 2014, pp. 3–15
16. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Inc. (1988)
17. Weyuker, E.J.: On testing non-testable programs. Comput. J. 25(4), 465–470 (1982)