RPP: Automatic Proof of Relational Properties by Self-Composition

Lionel Blatter[1], Nikolai Kosmatov[1], Pascale Le Gall[2], and Virgile Prevosto[1]

[1] CEA, LIST, Software Reliability and Security Laboratory, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[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. Self-composition provides a powerful theoretical approach to prove relational properties, i.e. properties relating several program executions, that has been applied to compare two runs of one or similar programs (in secure dataflow properties, code transformations, etc.). This tool demo paper presents RPP, an original implementation of self-composition for specification and verification of relational properties in C programs in the FRAMA-C platform. We consider a very general notion of relational properties invoking any finite number of function calls of possibly dissimilar functions with possible nested calls. The new tool allows the user to specify a relational property, to prove it in a completely automatic way using classic deductive verification, and to use it as a hypothesis in the proof of other properties that may rely on it.

Keywords: self-composition, relational properties, deductive verification, specification, Frama-C.

1 Introduction

Modular deductive verification allows the user to prove that a function respects its formal specification. For a given function f, any individual call to f can be proved 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. However, some kinds of properties are not reduced to one function call. Indeed, it is frequently necessary to express a property that involves several functions or relates the results of several calls to the same function for different arguments. We call them *relational properties*.

Different theories and techniques have been proposed to deal with relational properties in different contexts. They include Relational Hoare Logic to show the equivalence of program transformations [5] or Cartesian Hoare Logic for k-safety properties [15]. Self-composition [2] is a theoretical approach to prove relational properties relating two execution traces. It reduces the verification of a relational property to a standard verification problem of a new function. Self-composition techniques have been applied for verification of information flow properties [2, 1] and properties of two equivalent-result object methods [14]. Relational properties can be expressed on Java pure methods [11] using the JML specification language. OpenJML [8] offers a partial support for deductive verification of relational properties. The purpose of the present work is to implement and extend self-composition for specification and verification of relational properties in the context of the ACSL specification language [4] and the deductive verification plugin WP of FRAMA-C [13]. We consider a large class of relational properties (universally quantified properties invoking any finite number of calls of possibly dissimilar functions with possibly nested calls), and propose an automatic solution allowing the user not only to prove a relational property, but also to use it as a hypothesis.

Motivation. The necessity to deal with relational properties in FRAMA-C has been faced in various verification projects. Recent work [6] 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 the monotonicity of a function (e.g., $x \leq y \Rightarrow f(x) \leq f(y)$). To deal with it in FRAMA-C, [6] applies a variation of self-composition consisting in a separate verification of an additional, manually created wrapper function simulating the calls to be compared.

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. In this case, relational properties are only specified and used to verify client code, but are not verified themselves. 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 $\mathrm{Decrypt}(\mathrm{Encrypt}(Msg, PrivKey), PubKey) = Msg$. Other examples include properties of data structures, such as matrix transformations (e.g. $(A+B)^\intercal = A^\intercal + B^\intercal$ or $\det(A) = \det(A^\intercal)$), the specification of Push and Pop over a stack [7], or parallel program specification (e.g., $\mathrm{map}(\mathrm{append}(l_1, l_2)) = \mathrm{append}(\mathrm{map}(l_1), \mathrm{map}(l_2))$ in the MapReduce approach). A subclass of relational properties, *metamorphic properties*, relating multiple executions of the same function [12], are also used in a different context in order to address the oracle problem in software testing [16].

Manual application of self-composition or 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 contributions of this tool demo paper include:

- a new specification mechanism to formally express a relational property in ACSL;
- a fully-automated transformation into ACSL-annotated C code based on (an extension of) self-composition, that allows the user to prove such a property;
- a generation of an axiomatic definition and additional annotations that allow us to use a relational property as a hypothesis for the proof of other properties in a completely automatic and transparent way;
- an extension of self-composition to a large class of relational properties, including several calls of possibly dissimilar functions and possibly nested calls, and
- an implementation of this approach in a FRAMA-C plugin RPP with a sound integration of proof statuses of relational properties.

[3] http://www.systematic-paris-region.org/en/projets/pisco

```
1  /*@ assigns \nothing;
2    relational R1: ∀ int x1,x2;
3      x1 < x2 ⇒ \call(f1,x1) < \call(f1,x2);
4  */
5  int f1(int x){
6    return x + 1;
7  }
8
9  /*@ assigns \nothing;
10   relational R2: ∀ int x1, x2;
11     x1 < x2 ⇒ \call(f2,x1) < \call(f2,x2);
12   relational R3: ∀ int k;
13     \call(f1,k) < \call(f2,k);
14 */
15 int f2(int y){
16   return y + 2;
17 }
```

```
1  void relational_wrapper(int x1,int x2){
2    int tmp1 = 0;
3    int tmp2 = 0;
4    tmp1 = x1 + 1; // inlined f1(x1)
5    tmp2 = x2 + 1; // inlined f1(x2)
6  /*@ assert x1 < x2 ⇒ tmp1 < tmp2; */
7  }
8
9  /*@ axiomatic Relational_axiom{
10     logic int f1_acsl(int x);
11     lemma Relational_lemma: ∀ int x,y;
12       x < y ⇒ f1_acsl(x) < f1_acsl(y);
13     }
14 */
15
16 /*@ assigns \nothing;
17   behavior Relational_behavior:
18   ensures \result == f1_acsl(\old(x));
19 */
20 int f1(int x){
21   return x + 1;
22 }
23
24 ... // similar for f2
```

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

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 [13], a framework for analysis of C code developed at CEA LIST. FRAMA-C offers a specification language, called ACSL [4], and a deductive verification plugin, WP [3], that allow the user to specify the desired program properties as function contracts and to prove them. A typical ACSL function contract may include a precondition (**`requires`** clause stating a property supposed to hold before the function call) and a postcondition (**`ensures`** clause that should hold after the call), as well as a frame rule (**`assigns`** clause indicating which parts of the global program state the function is allowed to modify). An assertion (**`assert`** clause) can also specify a local property at any function statement.

Specification. To specify a relational property, we propose an extension of ACSL specification language with a new clause, **`relational`**. For technical, FRAMA-C-related, reasons, these clauses must be attached to a function contract. Thus, a property relating calls of different functions, such as `R3` in Figure 1a, must appear in the contract of the last function involved in the property, *i.e.* when all relevant functions are in scope. To refer to several function calls in such a property, we introduce a new construct **`\call`**`(f,<args>)` used 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 Figure 1a specify monotonicity of functions `f1,f2`, while property R3 at line 12-13 indicates that `f1(x)` is always less than `f2(x)`.

Preprocessing and Proof Status Propagation. Since this new syntax is not supported by classic deductive verification tools, we have designed a code transformation, inspired by self-composition, allowing the user to prove the property with one of these tools.

```
/*@ relational R1: ∀ int x1,x2;
  x1 < x2 ⇒ \call(f1,x1) < \call(f1,x2);
*/
int f1(int x);

/*@ relational R2: ∀ int x1,x2;
  x1 < x2 ⇒ \call(f2,x1) < \call(f2,x2);
*/
int f2(int x);

/*@ 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));
}
```

```
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);
    }
*/

/*@ behavior Relational_behavior:
  ensures \result == g_acsl(\old(x));
*/
int g(int x){
  return f1(x)+f2(x);
}

... // 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

We illustrate the transformation for function f1 and property R1 (see Figure 1a). The transformation result (Figure 1b) consists of three parts. First, a new function, called *wrapper*, is generated. The wrapper function is inspired by the workaround proposed in [6] and self-composition. 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 Figure 1b). The proof of such an assertion is possible with a classic deductive verification tool (WP can prove it in this example).

However, a wrapper function 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 subsequent proofs (as we illustrate below). Lemmas can refer to several function calls, but only for *logic* functions. Therefore, 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, similarly to what is proposed in [10], for adding to f_acsl 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 readibility) 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 [9], 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[4] as soon as the assertion (line 6) at the end of the wrapper function is proved.

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 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 [13]: proof obligations are generated and can be either discharged automatically by automatic theorem provers (e.g. Alt-Ergo, CVC4, Z3[5]) or proven interactively (e.g. in Coq[6]).

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

$$\forall\ \texttt{<args1>}, \ldots, \forall\ \texttt{<argsN>},$$
$$P(\ \texttt{<args1>}, \ldots, \texttt{<argsN>}, \textbf{\texttt{\textbackslash call}}\texttt{(f\_1,<args1>)}, \ldots, \textbf{\texttt{\textbackslash call}}\texttt{(f\_N,<argsN>)})$$

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. 1a 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 Figure 2a for slightly more complex functions (inspired by [6]) 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 11–12, 18 of Figure 1b for `f1`), WP automatically proves the assertion at line 6 of Figure 2b and validates property `Rg` as proven. The proof for `Rh` is similar.

Notice that in examples of Figure 2, functions `f1,f2` can be undefined since only their (relational) specification is required, which is suitable for specification of library or hardware-provided functions that cannot be specified without relational properties.

The RPP tool has also been successfully tested on several other examples such as cryptographic properties like $\mathrm{Decrypt}(\mathrm{Encrypt}(Msg, PrivKey), PubKey) = Msg$, squeeze lemma condition (i.e. $\forall x,\ f_1(x) \leq f_2(x) \leq f_3(x)$), median function properties (e.g. $\forall a, b, c,\ \mathrm{Med}(a, b, c) = \mathrm{Med}(a, c, b)$), properties of determinant for matrices of order 2 and 3 (e.g. $\det(A) = \det(A^{\intercal})$), matrix equations like $(A+B)^{\intercal} = A^{\intercal} + B^{\intercal}$, etc. Some of them include loops whose loop invariants are automatically transferred by RPP into the wrapper function to make possible its automatic proof.

[4] Technically, a special "valid under condition" status is used in this case in FRAMA-C.

[5] See, resp., https://alt-ergo.ocamlpro.com, http://cvc4.cs.nyu.edu, https://z3.codeplex.com/

[6] See http://coq.inria.fr/

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 impossibility 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.

Future work includes extending the tool to support complex data structures and functions with side-effects, 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.

Acknowledgment. Part of the research work leading to these results has received funding for DEWI project (www.dewi-project.eu) from the ARTEMIS Joint Undertaking under grant agreement No. 621353, and for the S3P project from French DGE and BPIFrance.

References

1. Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. Journal of Logical and Algebraic Methods in Programming (2016)
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science (2011)
3. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z.: WP Plugin Manual v1.0 (2016)
4. 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
5. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL (2004)
6. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: A case study. In: ISSRE (2013)
7. Burghardt, J., Gerlach, J., Lapawczyk, T.: ACSL by Example (2016), http://www.fokus.fraunhofer.de/download/acsl_by_example
8. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE (2014)
9. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: FMICS (2012)
10. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V.: Functional dependencies of C functions via weakest pre-conditions. STTT 13(5), 405–417 (2011)
11. Darvas, A., Müller, P.: Reasoning about method calls in JML specifications. Journal of Object Technology (2006)
12. Hui, Z.W., Huang, S.: A formal model for metamorphic relation decomposition. In: WCSE (2013)
13. 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
14. Leino, K.R.M., Müller, P.: Verification of Equivalent-Results Methods. In: ESOP (2008)
15. Sousa, M., Dillig, I.: Cartesian hoare logic for verifying k-safety properties. In: PLDI (2016)
16. Weyuker, E.J.: On testing non-testable programs. Comput. J. 25(4), 465–470 (1982)