Static and Dynamic Verification of Relational Properties on Self-Composed C Code

Lionel Blatter, Nikolai Kosmatov, Pascale Le Gall, Virgile Prévosto, Guillaume Petiot

Static and Dynamic Verification of Relational Properties on Self-Composed C Code

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

[1] CEA, List, Software Reliability and Security Lab, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] CentraleSupelec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
firstname.lastname@centralesupelec.fr

Abstract. Function contracts are a well-established way of formally specifying the intended behavior of a function. However, they usually only describe what should happen during a single call. Relational properties, on the other hand, link several function calls. They include such properties as non-interference, continuity and monotonicity. Other examples relate sequences of function calls, for instance, to show that decrypting an encrypted message with the appropriate key gives back the original message. Such properties cannot be expressed directly in the traditional setting of modular deductive verification, but are amenable to verification through *self-composition*. This paper presents a verification technique dedicated to relational properties in C programs and its implementation in the form of a FRAMA-C plugin called RPP and based on self-composition. It supports functions with side effects and recursive functions. The proposed approach makes it possible to prove a relational property, to check it at runtime, to generate a counterexample using testing and to use it as a hypothesis in the subsequent verification. Our initial experiments on existing benchmarks confirm that the proposed technique is helpful for static and dynamic analysis of relational properties.

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

1 Introduction

Context. Deductive verification techniques provide powerful methods for formal verification of properties expressed in Hoare Logic [11,12]. In this formalization, also known as axiomatic semantics, a program is seen as a predicate transformer, where each instruction S executed on a state verifying a property P leads to a state verifying another property Q. This is summarized in the form of *Hoare triples* as $\{P\}S\{Q\}$. In this setting, P and Q refer to states before and after a single execution of a program S. It is possible in Q to refer to the initial state of the program, for instance to specify that S has increased the value stored in variable `x`, but one cannot express properties that refer to two distinct executions of S, even less properties relating executions of different programs S_1 and S_2. As will be seen in the next sections, such properties, that we will call *relational properties* in this paper, occur quite regularly in practice. Hence,

it is desirable to provide an easy way to specify them and to verify that implementations are conforming to such specification. A simple example of a relational property is monotonicity of a function f: $x < y \Rightarrow \texttt{f}(x) < \texttt{f}(y)$.

Several theories and techniques exist for handling relational properties. First, Relational Hoare Logic [6] is mainly used to show the correctness of program transformations, *i.e.* the fact that the result of the transformation preserves the original semantics of the code. Then, Cartesian Hoare Logic [19] allows for the verification of k-safety properties, that is, properties over k calls of a function. The DESCARTES tool is based on Cartesian Hoare Logic and has been used to verify anti-symmetry, transitivity and extensionality of various comparison functions written in Java. A decomposition technique using abstract interpretation is presented in [1] for verification of k-safety properties. The method is implemented in a tool called BLAZER and used for verification of non-interference and absence of timing channel attacks. A relational program reasoning based on an intermediate program representation in LLVM is proposed by [13]. The method supports loops and recursive functions and is used for checking program equivalence. Finally, self-composition [3] and its refinement Program Products [2] propose theoretical approaches to prove relational properties by reducing the verification of relational properties to a standard deductive verification problem.

Motivation. In the context of the ACSL specification language [5] and the deductive verification plugin WP of FRAMA-C [14], the necessity to deal with relational properties has been faced in various verification projects. For example, we can extract the following quote from a work on verification of continuous monotonic functions in an industrial case study on smart sensor software [7] (emphasis ours):

> 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 authors attempt to prove the monotonicity of some functions (i.e., if $x \leq y$ then $f(x) \leq f(y)$) using FRAMA-C/WP plugin. To address the absence of support for relational properties in ACSL and WP, they perform a manual transformation [7] consisting in writing an additional function simulating the call to the related functions in the property. Broadly speaking, this amounts to manually perform self-composition. This technique is indeed quite simple and expressive enough to be used on many relational properties. However, applying it manually is relatively tedious, error-prone, and does not provide a completely automated link between three key components: *(i)* the specification of the property, *(ii)* the proof that the implementation satisfies the property, and *(iii)* the ability to use the property as hypothesis in other proofs (of relational as well as non-relational properties). 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. Finally, another motivation of this work was to obtain a solution compatible with other techniques than deductive verification, notably dynamic analysis.

Contributions. To address the absence of support for expressing relational properties in ACSL and for verifying such properties in the FRAMA-C platform, we implemented a

new plugin called RPP. This plugin allows the specification and verification of properties invoking any (finite) number of calls of possibly dissimilar functions with possibly nested calls, and to use the proved properties as hypotheses in other proofs. A preliminary version of RPP has been described in a previous short paper [8]. However, it suffered from major limitations. Notably, it could only handle pure, side-effect free functions, which in the context of the C programming language is an extremely severe constraint. Similarly, the original syntax to express relational properties is not expressive enough and requires some additional constructs, in order to properly specify relational properties of functions with side-effects. The previous work [8] did not address dynamic analysis of relational properties either.

The current paper will thus focus on the extensions that have been made to the original RPP design and implementation, as well as its evaluation. Its main contributions include:

- a new syntax for relational properties;
- handling of side effects;
- handling of recursive functions;
- evaluation of the approach over a suitable set of illustrative examples;
- experiments with runtime checking of relational properties and counterexample generation when a property cannot be proved in the context of RPP.

Outline. The remainder of this paper is organized as follows. First, in Section 2 we briefly recall the general idea of relational property verification with RPP in the case of pure functions using self-composition. Then, in Section 3, we show how to extend this technique to the verification of relational properties over functions with side effects (access to global variables and pointer dereference). Another extension, described in Section 4 allows considering recursive functions. We demonstrate the capacities of RPP by using it on the adaptation to C of the benchmark proposed for Java in [19] and our own set of test examples (Section 5). Finally, we show in Section 6 that RPP can also be used to check relational properties at runtime and/or to generate a counterexample using testing, and conclude in Section 7.

2 Context and Main Principles

RPP (Relational Property Prover) is a solution designed and implemented as a plugin of FRAMA-C [14], an extensible framework dedicated to the analysis of C programs. FRAMA-C offers a specification language, called ACSL [5], and a deductive verification plugin, WP [4], 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 that must hold each time the function is called) and a postcondition (**`ensures`** clause that must hold when the function returns), as well as a frame rule (**`assigns`** clause indicating which parts of the global program state the function is allowed to modify). **`assigns`** clauses may be refined by **`\from`** directives, indicating for each memory location l potentially modified by the function the list of memory locations that are read in order to compute the new value

of l. Finally, an assertion (**`assert`** clause) can also specify a local property at any function statement.

WP is based on Hoare logic and generates Proof Obligations (POs) using Weakest Precondition calculus: given a property Q and a fragment of code S, it is possible to compute the minimal (weakest) condition P such that $\{P\}S\{Q\}$ is a valid Hoare triple. When S is the body of a function f, POs are formulas expressing that the precondition of f implies the weakest condition necessary for the postcondition (or assertion) to hold after executing S. POs can then be discharged either automatically by automated theorem provers (e.g. Alt-Ergo, CVC4, Z3[3]) or with some help from the user *via* a proof assistant (e.g. Coq[4]).

FRAMA-C also offers an executable subset of ACSL, called E-ACSL [10,18], that can be transformed into executable C code. It is thus compatible with dynamic analysis, such as runtime assertion checking of annotations using the E-ACSL plugin [10,20] or with counterexample generation (in case of a proof failure) using the STADY plugin [16,17].

Function contracts allow specifying the behavior of a single function call, that is, properties of the form "If $P(s)$ is verified when calling f in state s, $Q(s')$ will be verified when f returns with state s'". However, it is not possible to specify *relational properties*, that relate several function calls. Examples of such properties include monotonicity ($x < y \Rightarrow \texttt{f}(x) < \texttt{f}(y)$), anti-symmetry ($\texttt{compare}(x, y) = -\texttt{compare}(y, x)$) or transitivity ($\texttt{compare}(x, y) \leq 0 \wedge \texttt{compare}(y, z) \leq 0 \Rightarrow \texttt{compare}(x, z) \leq 0$). RPP addresses this issue by providing an extension to ACSL for expressing such properties and a way to prove them. More specifically, RPP works like a preprocessor for WP: given a relational property and the definition of the C function(s) involved in the property, it generates a new function together with plain ACSL annotations whose proof (using the standard WP process) implies that the relational property holds for the original code. As we show below, this encoding of a relational property is also compatible with dynamic analysis (runtime verification or counterexample generation).

2.1 Original Relational Specification Language

For the specification of a relational property, we initially proposed an extension [8] of the ACSL specification language with a new clause, **`relational`**. These clauses are attached to a function contract. A property relating calls of different functions, such as `R1` 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. In this new clause we introduced a new construct **`\call`**`(f,<args>)`, denoting the value returned by the call `f(<args>)` to `f` with arguments `<args>`. This allows relating several function calls in a **`relational`** clause. **`\call`** can be used recursively, i.e. a parameter of a called function can be the result of another function call. In Figure 1a, properties `R1` and `R2` at lines 7–9 and 15–17 specify properties of functions `max` and `min` respectively.

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

[4] See `http://coq.inria.fr/`

```
1  /*@ requires x > INT_MIN;
2      assigns  \nothing;
3      behavior pos:
4        assumes x ≥ 0;
5        ensures \result  == x;
6      behavior neg:
7        assumes x < 0;
8        ensures \result  == -x;*/
9  int abs (int x){
10   return (x ≥ 0) ? x : (-x);
11 }
12
13 /*@ requires INT_MIN < x+y < INT_MAX;
14     assigns \nothing;
15     relational R1: ∀ int x,y;
16      \call(max,x,y) ==
17       (x+y+\call(abs,x - y))/2; */
18 int max(int x,int y){
19   return (x ≥ y) ? x : y;
20 }
```

(a) Original source code

```
1  /*@ axiomatic Relational_axiom {
2    logic int max_acsl(int x, int y);
3    logic int abs_acsl(int x);
4    lemma Relational_lemma{L}:
5      ∀ int x, int y;
6        max_acsl(x, y) ==
7         ((x + y) + abs_acsl(x - y)) / 2;
8    }*/
9
10 void relational_wrapper(int x, int y){
11   int ret_var_1, ret_var_2;
12   ret_var_1 = (x ≥ y) ? x : y;
13   ret_var_2 = (x-y ≥ 0) ? x-y : (-(x-y));
14   /*@ assert
15        ret_var_1 ==
16        ((x + y) + ret_var_2) / 2; */
17   return;
18 }
19
20 /*@ assigns \nothing;
21     behavior Relational_behavior:
22       ensures
23        \result ==
24        max_acsl(\old(x), \old(y));
25  */
26 int max(int x, int y){ ... }
```

(b) Excerpt of the code generated by RPP

Fig. 1: Pure function with relational properties

Note however that the **`\call`** construct only allows speaking about the return value of a C function. If the function has some side effects, there is no way to express a relation between the values of memory locations that are modified by distinct calls. Section 3 describes the improvements that have been made to the initial version of the relational specification language in order to support side effects. To ensure that a function has no side effects, an **`assigns \nothing`** clause can be used.

2.2 Preprocessing of a Relational Property

The previous work [8] also proposed a code transformation whose output can be analyzed with standard deductive verification tools. This is materialized in the RPP plugin of FRAMA-C, that relies then on WP to prove the resulting standard ACSL annotations.

Going back to our example, applying the transformation to property `R1` over function `max` gives the code of Figure 1b. The generated code can be divided into three parts. First, a new function, called *wrapper*, is generated. The wrapper function is inspired by the workaround proposed in [7] and self-composition [3]. As in self-composition, this wrapper function inlines the calls occurring in the relational property under analysis, with a suitable renaming of local variables to avoid interferences between the calls.

In addition, the wrapper records the results of the calls in fresh local variables. Then, in the spirit of calculational proofs [15], we state an assertion equivalent to the relational property (lines 14–16 in Figure 1b). The proof of such an assertion is possible with a classic deductive verification tool (WP with Alt-Ergo as back-end prover in our case).

However, the wrapper function only provides a solution to prove relational properties. The ability to use these properties as hypotheses in other proofs (relational or not)

```
1  /*@ assigns \nothing;*/
2  int Crypt(int m,int key){
3    return m + key;
4  }
5
6  /*@ assigns \nothing;
7      relational R3:
8        ∀ int m, key;
9          \call(Decrypt,
10           \call(Crypt,m,key),
11           key)
12       == m;*/
13 int Decrypt(int m,int key){
14   return m - key;
15 }
16
17 /*@ assigns \nothing;
18     ensures \result == m;
19     relational R4:
20      ∀ int m,key;
21       \call(run,
22         \call(run,m,key),
23         key)
24      == m;*/
25 int run(int m,int key){
26   int crypt, decrypt;
27   crypt = Crypt(m,key);
28   decrypt = Decrypt(crypt,key);
29   return decrypt;
30 }
```

(a) Original source code

```
1  /*@ axiomatic Relational_axiom {
2    logic int run_acsl(int m, int key);
3
4    lemma Relational_lemma{L}:
5      ∀ int m, int key;
6      run_acsl(
7        run_acsl(m, key),
8        key)
9     == m; }*/
10
11 void relational_wrapper(int m, int key){
12   int tmp_1, tmp_2, tmp_3, tmp_4;
13   tmp_1 = Crypt_aux_2(m,key);
14   tmp_2 = Decrypt_aux_2(tmp_1,key);
15   tmp_3 = Crypt_aux_2(tmp_2,key);
16   tmp_4 = Decrypt_aux_2(tmp_3,key);
17   /*@ assert tmp_4 == m;*/
18   return; }
19
20 /*@ ensures \result == \old(m);
21     assigns \nothing;
22     behavior Relational_behavior:
23       ensures \result ==
24       run_acsl(\old(m), \old(key));*/
25 int run(int m, int key){
26   int crypt;
27   int decrypt;
28   crypt = Crypt(m,key);
29   decrypt = Decrypt(crypt,key);
30   return decrypt; }
```

(b) Transformed code

Fig. 2: Functions `Crypt` and `Decrypt`, used by function `run`.

must be reached otherwise. For this purpose, RPP generates an ACSL axiomatic definition (cf. **`axiomatic`** section at lines 1–8 in Figure 1b) introducing a logical reformulation of the relational property as a lemma (cf. lines 4–7) over otherwise unspecified logic functions (`max_acsl` and `abs_acsl` in the example). Furthermore, new postconditions are generated in the contracts of the C functions involved in the relational property. They specify that there is an exact correspondence between the original C function and its newly generated logical ACSL counterpart. Thanks to this axiomatic, POs over functions calling `max` and `abs` will have the lemma in their environment and thus will be able to take advantage of the proven relational property. Note that the correspondence between `max` and `max_acsl` (respectively `abs` and `abs_acsl`) can only be done because `max` and `abs` do not access global memory (neither for writing nor for reading). Indeed, since `max_acsl` and `abs_acsl` are pure logic functions, they do not have side effects and their result only depends on their parameters.

To illustrate the use of relational properties in the proof of other specifications, we can consider the postcondition and property `R4` of function `run` of Figure 2a (inspired by the PISCO project[5]) whose proof needs to use property `R3`. Thanks to their reformulation as lemmas and to the link between ACSL and C functions, WP automatically proves the assertion at line 17 (for property `R4`) and the postcondition at line 20 of Figure 2b.

[5] See `http://www.projet-pisco.fr/`.

```
/*@ assigns \result \from x, y;
    relational R1:
       \forall int x1, y1;
          \callset(\call(max, x1, y1, id1),\call(abs, x1 - y1, id2)) ==>
           \callresult(id1) == (x1 + y1 + \callresult(id2)) / 2;
*/
int max(int x,int y) { ... }
```

Fig. 3: Annotated C function with `relational` annotations

2.3 Soundness of the transformation

Since our transformation is introducing an ACSL **`axiomatic`**, care must be taken to avoid introducing inconsistencies in the specification. More precisely, the **`axiomatic`** specifies the intended behavior of the ACSL counterpart of the C functions under analysis. The corresponding ACSL functions are then only used in the contracts of those C functions. In particular, since the wrapper is inlining the body of the functions concerned by the relational property, the **`lemma`** of the **`axiomatic`** cannot be used to prove the **`assert`** annotation inside the wrapper.

3 Functions with Side Effects

As mentioned above, the initial RPP approach only works for relational properties over pure functions. More precisely, it allows proving relational properties of the form:

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

for an arbitrary predicate P invoking $N \geq 1$ calls of non-recursive functions without side effects. In the context of the C programming language, handling only pure functions is a major limitation. We thus propose an extension of both the specification language and the transformation technique in order to let RPP tackle a wider, more representative, class of C functions.

3.1 New Grammar for Relational Properties

Relational properties are still introduced by a **`relational`** clause inside an ACSL contract. However, since we might now refer to memory locations in either the pre- or the post-state of any call implied in the relational property, we need to be able to make explicit references to these states, and not only to the value returned by a given call. Although more verbose, the new syntax can also be used for pure functions. For instance, property `R1` of Figure 1a can be rewritten as shown in Figure 3.

More generally, we introduce the grammar shown in Figure 4. A relational clause is composed of three parts. First, we declare a set of universally quantified variables, that will be used to express the arguments of the calls that are related by the clause. Then, we specify the set of calls on which we will work in the *relational-def* part. As shown in Figure 4a, each call is then associated to an identifier *call-id*. In the property `R1` of

⟨*call-id*⟩ ::= id

⟨*bin-rel*⟩ ::= `==` | `!=` | `<=` | `>=` | `>` | `<`

⟨*function-parameter*⟩ ::= ⟨*relational-call-terms*⟩+

⟨*function-name*⟩ ::= poly-id

⟨*function-call*⟩ ::= `\call(`⟨*inlining-option*⟩`,`
⟨*function-name*⟩`,`
⟨*function-parameter*⟩`,`
⟨*call-id*⟩`)`

⟨*call-parameter*⟩ ::= ⟨*function-call*⟩+

⟨*relational-def*⟩ ::= `\callset(`⟨*call-parameter*⟩`)`

⟨*relational-pred*⟩ ::= `\true` | `\false`
| ⟨*relational-terms*⟩ ⟨*bin-rel*⟩ ⟨*relational-terms*⟩
| ⟨*relational-pred*⟩ `&&` ⟨*relational-pred*⟩
| ⟨*relational-pred*⟩ `||` ⟨*relational-pred*⟩
| ⟨*relational-pred*⟩ `==>` ⟨*relational-pred*⟩
| `!`⟨*relational-pred*⟩
| `\forall` ⟨*binders*⟩ `;`⟨*relational-pred*⟩
| `\exists` ⟨*binders*⟩ `;`⟨*relational-pred*⟩

⟨*relational-annot*⟩ ::= `relational` ⟨*relational-clause*⟩

⟨*relational-clause*⟩ ::=
`\forall` ⟨*binders*⟩ `;`
⟨*relational-def*⟩ `==>` ⟨*relational-pred*⟩

(a) Grammar of relational predicates

⟨*literal*⟩ ::= `\true` | `\false` | int | float

⟨*relational-label*⟩ ::= `Post_`⟨*call-id*⟩
| `Pre_`⟨*call-id*⟩

⟨*bin-op*⟩ ::= `+` | `-` | `*` | `/` |

⟨*result-reference*⟩ ::= `\callresult(` ⟨*call-id*⟩ `)`

⟨*pure-function-parameter*⟩ ::= ⟨*relational-call-terms*⟩+

⟨*inlining-option*⟩ ::= int

⟨*pure-function-name*⟩ ::= poly-id

⟨*pure-function-call*⟩ ::=
`\callpure(`⟨*inlining-option*⟩`,`
⟨*pure-function-name*⟩`,`⟨*pure-function-parameter*⟩`)`

⟨*relational-call-terms*⟩ ::= ⟨*literal*⟩
| ⟨*pure-function-call*⟩
| ⟨*relational-call-terms*⟩ ⟨*bin-op*⟩ ⟨*relational-call-terms*⟩

⟨*relational-terms*⟩ ::= ⟨*literal*⟩
| ⟨*relational-terms*⟩ ⟨*bin-op*⟩ ⟨*relational-terms*⟩
| ⟨*result-reference*⟩
| `\at(`⟨*poly_id*⟩ `,` ⟨*relational-label*⟩ `)`
| ⟨*pure-function-call*⟩

(b) Grammar of relational terms

Fig. 4: Grammar for relational properties

Figure 3, two function calls are explicitly specified in the **`\callset`** construct and not directly in the predicate. Each call has its own identifier (`id1` and `id2` respectively). Finally, the relational property itself is given as an ACSL predicate in the *relational-pred* part. As described in Figure 4a, in addition to standard ACSL constructs, three new terms can be used. First, **`\callpure`** can be used to indicate the value returned by a pure function as was done with the **`\call`** built-in in the original version of RPP. This allows specifying relational properties over pure functions without the overhead required for handling side-effects. As before, nested **`\callpure`** are allowed. Second, **`\callresult`**, as used in Figure 3, takes a *call-id* as parameter and refers to the value returned by the corresponding call in *relational-def*. Finally, each such *call-id* gives rise to two logic labels. Namely, `Pre_`*call-id* refers to the pre-state of the corresponding call, and `Post_`*call-id* to its post-state. These labels can in particular be used in the ACSL term **`\at`**`(e,`L`)` that indicates that the term `e` must be evaluated in the context of the program state linked to logic label L. Figure 5a below shows an example of their use.

3.2 Global Variables Accesses

As said before, the new syntax for relational properties enables us to speak about the value of global variables at various states of the execution, thanks to the newly defined logic labels bound to each call involved in the **`\callset`** of the property. This is for instance the case in the relational property of Figure 5a, which indicates that `h` is mono-

```
1  int y;
2
3  /*@ assigns y \from y;
4      relational R1:
5       \callset(\call(h,id1),
6                \call(h,id2))
7       ==>
8        \at(y,Pre_id1) < \at(y,Pre_id2)
9        ==>
10       \at(y,Post_id1) < \at(y,Post_id2);
11 */
12 void h(){
13   int a = 10;
14   y = y + a;
15   return;
16 }
```

(a) Annotated C function with **relational** annotations

```
1  int y;
2
3  /*@ axiomatic Relational_axiom_1 {
4    predicate
5       h_acsl(int y_pre, int y_post);
6
7    lemma Relational_lemma_1:
8    \forall int y_id2_pre, y_id2_post,
9                y_id1_pre, y_id1_post;
10    h_acsl(y_id2_pre, y_id2_post)
11    ==> h_acsl(y_id1_pre, y_id1_post)
12    ==> y_id1_pre < y_id2_pre
13    ==> y_id1_post < y_id2_post; }*/
14
15 /*@ assigns y \from y;
16     behavior Relational_behavior_1:
17       ensures h_acsl(\at(y,Pre),
18       \at(y,Post));*/
19 void h(void){ ... }
20
21 int y_id1;
22 int y_id2;
23
24 void relational_wrapper_1(void){
25     int a_1 = 10;
26     y_id1 = y_id1 + a_1;
27     int a_2 = 10;
28     y_id2 += y_id2 + a_2;
29   /*@ assert Rpp:
30     \at(y_id1,Pre) < \at(y_id2,Pre) ==>
31     \at(y_id1,Here) < \at(y_id2,Here); */
32   return;
33 }
```

(b) Transformed code for verification and use of relational properties with side effect

Fig. 5: Relational property on a function with side-effect

tonic with respect to `y`, in the sense that if a first call to `h` is done in a state `Pre_id1` where the value of `y` is strictly less than in the pre-state `Pre_id2` of a second call, this will also be the case in the respective post-states `Post_id1` and `Post_id2`.

Generation of the wrapper function is more complicated in presence of side-effects. As presented in [3], each function call must operate on its own memory state, separated from the other calls in order for self-composition to work. We thus create as many duplicates of global variables as needed to let each part of the wrapper use its own set of copies. However, to avoid useless copies, RPP requires that each function involved in a relational property has been equipped with a proper set of ACSL **`assigns`** clauses, including **`\from`** components. This constraint is similar to what is proposed in [9], and ensures that only the parts of the global state that are accessed (either for writing or for reading) by the functions under analysis are subject to duplication. As an example, the wrapper function corresponding to our `h` function of Figure 5a is shown in lines 24–33 of Figure 5b.

Finally, the generated axiomatic definition enabling the use of the relational property in other POs must also be modified. The original transformation uses a logic function that is supposed to return the same **`\result`** as the C function. However, since logic functions are always pure, this mechanism is not sufficient to characterize side effects in the logic world. Instead, we declare a predicate that takes as parameters not

only the returned value and the formal parameters of the C function, but also the relevant parts of the program states that are involved in the property. As for the wrapper function, these additional parameters are inferred from the **`assigns ... \from ...`** clauses of the corresponding C functions. For instance, predicate `h_acsl`, on line 5 of figure 5b, takes two arguments representing the values of `y` before and after and execution of `h`. This link between the ACSL predicate and the C function is again materialized by an **`ensures`** clause (lines 17–18). The lemma defining the ACSL predicate is more complex too, since we have to quantify over the values of all the global variables at all relevant program states. In the example, this is shown on lines 7–13, where we have 4 quantified variables representing the value of global variable `y` before and after both calls involved in the relational property.

3.3 Support of Pointers

In the previous section, we have shown how to specify relational properties in presence of side effects over global variables, and how the transformations for both proving and using a property are performed. However, support of pointer dereference is more complicated. Again, as proven in [3] Self-Composition works if the memory footprint of each call is separated from the others. Thus, in order to adapt our method, we must ensure that pointers that are accessed during two distinct calls point to different memory locations. As above, such accesses are given by **`assigns ... \from ...`** clauses in the contract of the corresponding C functions. An example of a relational property on a function `k` using pointers (monotonicity with respect to the content of a pointer) is given in Figure 6a, where `k` is specified to assign `*y` using only its initial content.

Memory separation is enforced using ACSL's built-in predicate **`\separated`**. For the wrapper function, we add a **`requires`** clause stating the appropriate **`\separated`** locations. This can be seen on Figure 6b, line 20, where we request that the copies of pointer `y` used for the inlining of both calls to `k` points to two separated area in the memory. Similarly, in the axiomatic part, the lemma adds separation constraints over the universally quantified pointers (line 9 in the Figure 6b).

We also need to refine the declaration of the predicate in presence of pointer accesses. First, the predicate now needs to explicitly take as parameters the pre- and post-states of the C function. In ACSL, this is done by specifying *logic labels* as special parameters, surrounded by braces, as shown in line 3 of Figure 6b. Second, a **`reads`** clause allows one to specify the footprint of the predicate, that is, the set of memory accesses that the validity of the predicate depends on (line 4). Similarly, the lemma on lines 6–13 takes 4 logic labels as parameters, since it relates two calls to `k`, each of them having a pre- and a post-state.

It should be noted that the memory separation assumption makes the tool verify relational properties without pointer aliasing. Support of properties with pointer aliasing is left as future work.

4 Recursive Functions

We have shown in the previous section how we handle functions with side effects. Let us now focus on another class of functions, namely recursive functions. Support for

```
1  /*@ assigns *y \from *y;
2      relational R1:
3      \callset(
4        \call(k,id1),
5        \call(k,id2))
6       ==>
7         \at(*y,Pre_id1) <
8         \at(*y,Pre_id2)
9         ==>
10         \at(*y,Post_id1) <
11         \at(*y,Post_id2);
12 */
13 void k(int *y){
14   *y = *y + 1;
15   return;
16 }
```

(a) Original annotated C function

```
1  /*@ axiomatic Relational_axiom_1 {
2  predicate
3  k_acsl{pre, post}(int *y)
4    reads \at(*y,post), \at(*y,pre);
5
6  lemma Relational_lemma_1
7    {pre_id2, post_id2, pre_id1, post_id1}:
8  \forall int *y_id2, int *y_id1;
9  \separated(y_id1,y_id2)
10   ==> k_acsl{pre_id2, post_id2}(y_id2)
11   ==> k_acsl{pre_id1, post_id1}(y_id1)
12   ==> \at(*y_id1,pre_id1) < \at(*y_id2,pre_id2)
13   ==> \at(*y_id1,post_id1) < \at(*y_id2,post_id2);
14 }*/
15 /*@ assigns *y \from *y;
16     behavior Relational_behavior_1:
17       ensures k_acsl{Pre, Post}(y);*/
18 void k(int *y){  ... }
19
20 /*@ requires \separated(y_id1, y_id2);*/
21 void relational_wrapper_1(int *y_id1, int *y_id1){
22   *y_id1 = *y_id1 + 1;
23
24   *y_id2 = *y_id2 + 1;
25
26   /*@ assert Rpp:
27   \at(*y_id1,Pre) < \at(*y_id2,Pre) ==>
28   \at(*y_id1,Here) < \at(*y_id2,Here);*/
29   return;
30 }
```

(b) Code transformation

Fig. 6: Relational property in presence of pointers

recursive functions in RPP is interesting because it is very natural to specify such functions with relational properties. For example, a naive specification of a `fact` function computing the factorial of an integer can be written as

$$\begin{cases} \forall x.\ x \leq 1 \implies fact(x) = 1, \\ \forall x.\ x > 1 \implies fact(x) = fact(x-1) * (x) \end{cases}$$

The corresponding relational properties are given in Figure 7a. The proof of the `Induction` property requires a modification to the generation of the wrapper function, that can be observed in Figure 7b. Indeed, we do not want to inline the second call to `fact` on line 12, in order to take advantage of the fact that, since `fact` is a pure function that does not read anything from the global environment, this call returns the same value as the one of line 9, obtained by inlining the call to `fact(x1)`. This is why, as was indicated on Figure 4, there is an optional argument to the **`\callpure`** construct, that indicates the maximal depth that the inlining can reach in the wrapper. The default value of `1`, which is also used explicitly in our example for the first call, on line 9 of Figure 7a, means that we inline the body of the function once (i.e. if the function calls other functions, including itself, these calls themselves will not be inlined). When this parameter is set to `0`, as is the case for the second call in our example (line 10), we keep the call as such in the wrapper.

Support for recursive functions is not limited to pure functions. Recursive functions with side effects can also be handled. In particular, as shown in the grammar, each **`\call`** appearing in a **`\callset`** can also have an inlining directive. For instance, we can consider another implementation of the factorial, whose result is this time recorded

```
/*@ assigns \result \from x;
    relational Base:
      \forall int x1;
        x1 <= 1 ==>
          \callpure(1,fact,x1) == 1;
    relational Induction:
      \forall int x1;
        x1 > 1 ==>
        \callpure(1,fact,x1) ==
          \callpure(0,fact,x1-1)*x1;
*/
int fact(int x) {
        if(x <= 1){
                return 1;
        }
        else{
                return fact(x-1)*x;
        }
}
```

(a) Annotated recursive C function with `relational` clauses

```
void relational_wrapper_2(int x1){
int return_var_rela_2;
int return_var_rela_3;
{
  if (x1 <= 1) {
    return_var_rela_2 = 1;
  }
  else {
    return_var_rela_2 = fact(x1-1)*x1;
  }
}
return_var_rela_3 = fact(x1-1);
/*@ assert Rpp:
     x1 > 1 ==>
     return_var_rela_2 ==
     return_var_rela_3*x1;
*/
return;
}
```

(b) Code transformation for the proof of the second relational property

Fig. 7: Relational property on recursive C function without side effects

in a global variable `r` (Figure 8). The corresponding relational properties (lines 5–9) are similar to the pure case. However, the proof is slightly different, since the function has side effects, we cannot use logic function equality. Instead, we use the relational property as an induction hypothesis and inline both functions.

Note that in this case, a call to the function itself appears in the wrapper, contrarily to the situation detailed in section 2.3. However, under the assumption that the function always terminates, this call is performed on arguments that are strictly smaller than the ones of the wrapper itself. Hence, the **`axiomatic`** can be used as an induction hypothesis in the sense that the wrapper allows us to prove that if the relational property holds for arguments smaller than `x`, then it holds for `x`.

5 llustrative Examples

We have seen how to express relational properties over a large class of C functions and how RPP can generate C code and plain ACSL specifications for proving and using these properties through a standard WP process. To check that this approach works in practice, we have tested our tool on different benchmarks. These tests aim at confirming:

- the ability to specify various relational properties over a large class of functions;
- the capacity to prove and use such properties using the generated transformation;
- the support of a large range of function implementations;
- the ability to use other techniques (runtime checks, test generation for invalidating the property) when WP fails to discharge a corresponding PO.

The first subsection will present our own benchmark composed of a mix of different types of relational properties. This benchmark is mainly designed to validate the two first items. The second subsection will show how RPP has performed on the benchmark proposed in [19]. This will confirm the second and third points. Finally, we will present in Section 6 our use of the E-ACSL and STADY plugins assessing the last point.

```
int r;

/*@ requires x >= 0;
    assigns r \from r,x;
    relational \forall int x1;
     \callset(\call(1,fact,x1,id1)) ==> x1 <= 1 ==> \at(r,Post_id1) == 1;
    relational \forall int x1;
     \callset(\call(1,fact,x1,id2), \call(1,fact,x1-1,id3))
     ==> x1 > 1 ==> \at(r,Post_id2) == \at(r,Post_id3)*x1;
*/
void fact(int x) {
  if(x <= 1){
    r = 1;
    return;
  }
  else{
    fact(x-1);
    r = r * x;
    return;
 }
}
```

Fig. 8: Relational property on recursive C function with side effects

5.1 Internal Examples

As stated previously, we have tested RPP on a set of relational properties extracted from real case studies. This includes in particular encryption, as presented in Section 2, monotonicity (Section 3) or the factorial of Section 4, but also properties found in map/reduce, as the one in row 6 in Figure9, stating that the choice of the partitioning for the initial set of data should not play a role in the final result. The benchmark is also composed of more academic examples like linear algebraic properties of matrices, over functions containing loops (rows 7 and 8), or the property of row 10, that states the symmetry of the median of three numbers.

Figure 9 summarizes the results obtained on the benchmark. The first three columns indicate respectively whether the corresponding property could be specified and the corresponding code transformation generated, proved and used as an hypothesis in other proofs. The last three columns show what kind of C constructs are used in the implementation of the functions under analysis, namely side effects, presence of loops (which are always difficult for WP-related verification techniques, due to the need for loop invariants), and presence of recursive functions.

Num	Relational Property	Specified / Generated	Verified	Used	Side effect	Loop	Recursive
1	$\forall x1, x2 \in \mathbb{Z}:$ $x1 < x2 \Rightarrow f(x1) < f(x2)$	✓	✓	✓	✓	✗	✗
2	$\forall\, x;$ $f(x+1) = f(x) * (x+1)$	✓	✓	✓	✓	✗	✓
3	$\forall x,\ f_1(x) \leq f_2(x) \leq f_3(x)$	✓	✓	✗	✗	✗	✗
4	$\forall x, f(f(x)) = f(x)$	✓	✓	✗	✗	✓	✗
5	$\forall\, Msg, Key;$ $Decrypt(Encrypt(Msg, Key), Key) = Msg$	✓	✓	✓	✓	✓	✗
6	$\forall\, t, sub_{t1}, ..., sub_{tn};$ $t = sub_{t1} \cup ... \cup sub_{tn} \Rightarrow$ $max(t) = max(max(sub_{t1}), ..., max(sub_{tn}))$	✓	✓	✗	✓	✓	✗
7	$\forall\, A, B;$ $(A+B)^T = (A^T + B^T)$	✓	✓	✗	✗	✓	✗
8	$\det(A) = \det(A^\intercal)$	✓	✓	✗	✗	✓	✗
9	$\forall x1, x2, y, f(x1 + x2, y) = f(x1, y) + f(x2, y)$	✓	✓	✓	✗	✗	✓
10	$\forall a, b, c,\ \mathrm{Med}(a, b, c) = \mathrm{Med}(a, c, b)$	✓	✓	✗	✗	✗	✗

Fig. 9: Summary of relational properties considered by RPP

5.2 Comparator Functions

We also evaluated RPP on the benchmark proposed in [19]. It is composed of a collection of flawed and corrected implementations of comparators over a variety of data types written in Java, inspired from a collection of Stackoverflow [6] questions. Translating the Java code into C was straightforward and fully preserved the semantics of the functions. We focused on the same properties as [19], that is anti-symmetry (P1), transitivity (P2) and extensionality (P3). Mathematically, these properties can be expressed as such:

$$P1 : \forall\, s1, s2.\ compare(s1, s2) = -compare(s2, s1)$$
$$P2 : \forall\, s1, s2, s3.\ compare(s1, s2) > 0 \wedge compare(s2, s3) > 0 \Rightarrow compare(s1, s3) > 0$$
$$P3 : \forall\, s1, s2, s3.\ compare(s1, s2) = 0 \Rightarrow (compare(s1, s3) = compare(s2, s3))$$

Results are depicted in Table 10. For each comparator, we indicate whether the properties P1, P2 and P3 hold according to RPP (✓ and ✗ show whether the property was proved valid by WP). We get similar results as [19], with the exception of PokerHand, for which the generated wrapper function seems currently out of reach for WP (limits of scalability due to the combinatorial explosion of self-composition). However, by rewriting the function in a more modular way, WP was able to handle the example.

6 Dynamic Verification

6.1 Counterexample Generation

For the properties that do not hold in the comparator benchmark, we have been able to find counterexamples thanks to the proposed encoding of a relational property by self-composed code and using another FRAMA-C plugin, STADY [17]. STADY[7] is a testing-based counterexample generator. In particular, STADY tries to find an input vector that will falsify an ACSL annotation for which WP could not decide whether it holds, thereby showing that the code is not conforming to the specification.

We apply STADY to try to find a test input such that the **`assert`** clause at the end of the `wrapper` function is false. The results are shown in the STADY columns of Figure 10. Obviously, STADY does not try to find counterexamples for properties that are proved valid by WP. For properties that are not proved valid, ✓ indicates that a counterexample is found (within a timeout of 30 seconds), while ✄ indicated the only case where a counterexample is not generated before a 30-second timeout. A longer timeout (60 minutes) did not improve the situation in that case. Symbol ✐ denotes two cases where the code translation uses features that are currently not yet supported by STADY. As shown in the table, thanks to the RPP translation, STADY was able to find counterexamples for almost all unproven properties. Notice that some examples

[6] `https://stackoverflow.com`

[7] See `https://github.com/gpetiot/Frama-C-StaDy`

Benchmark	Proof (WP) P1	P2	P3	Counterex. gen. (STADY) P1	P2	P3
ArrayInt-false.c	✓	✓	✗	–	–	✓
ArrayInt-true.c	✓	✓	✓	–	–	–
CatBPos-false.c	✗	✗	✗	✓	✓	✓
Chromosome-false.c	✓	✗	✗	–	✂	✓
Chromosome-true.c	✓	✓	✓	–	–	–
ColItem-false.c	✗	✗	✗	✓	✓	✓
ColItem-true.c	✓	✓	✓	–	–	–
Contact-false.c	✓	✗	✗	–	✓	✓
Container-false-v1.c	✗	✓	✓	✓	–	–
Container-false-v2.c	✗	✗	✗	✓	✓	✓
Container-true.c	✓	✓	✓	–	–	–
DataPoint-false.c	✗	✗	✗	✓	✓	✓
FileItem-false.c	✓	✓	✗	–	–	✓
FileItem-true.c	✓	✓	✓	–	–	–
IsoSprite-false-v1.c	✗	✗	✗	✓	✓	✓
IsoSprite-false-v2.c	✗	✗	✓	✓	✓	–
Match-false.c	✗	✓	✗	✓	–	✓
Match-true.c	✓	✓	✓	–	–	–
NameComparator-false.c	✗	✓	✓	✓	–	–
NameComparator-true.c	✓	✓	✓	–	–	–
Node-false.c	✓	✓	✗	–	–	✓
Node-true.c	✓	✓	✓	–	–	–
NzbFile-false.c	✗	✓	✓	✓	–	–
NzbFile-true.c	✓	✓	✓	–	–	–
PokerHand-false.c	✓	✗	✗	–	✎	✎
PokerHand-true.c	✓	✓	✓	–	–	–
Solution-false.c	✓	✓	✗	–	–	✓
Solution-true.c	✓	✓	✓	–	–	–
TextPosition-false.c	✓	✗	✗	–	✓	✓
TextPosition-true.c	✓	✓	✓	–	–	–
Time-false.c	✗	✓	✓	✓	–	–
Time-true.c	✓	✓	✓	–	–	–
Word-false.c	✗	✗	✓	✓	✓	–
Word-true.c	✓	✓	✓	–	–	–

Fig. 10: Comparator properties analysed with WP and STADY after RPP translation

required minor modifications so that STADY can be used. To be able to use testing, we had of course to add bodies for unimplemented functions. Other modifications consisted in reducing the input space to a representative smaller domain (by limiting the size of an input array) for some examples to facilitate counterexample generation [17].

6.2 Runtime Assertion Checking

The code transformation technique of RPP also enables runtime verification of relational properties through the E-ACSL plugin [10,20]. More precisely, the E-ACSL plugin translates ACSL annotations into C code that will check them at runtime and

abort execution if one of the annotations fails. We tested the E-ACSL plugin on the test inputs generated by STADY in order to check that each generated counterexample does indeed violate the relational property. As expected, the obtained results validate those of the previous section. Since counterexample generation with STADY [17] basically includes a runtime assertion checking step for each test datum considered during the test generation process, we do not present the results of this step in separate columns.

7 Conclusion and Future Work

We have presented a major extension to an existing verification technique for relational properties, implemented in the FRAMA-C plugin RPP. The extension adds support for functions with side effects (access to global variables and pointer dereferences) and recursive functions. RPP relies on FRAMA-C/WP for automatic or interactive proof of the relational properties and offers the ability to use them as hypothesis in other proofs. Moreover, beyond WP, RPP also allows users to take advantage of E-ACSL and STADY plugins to verify relational properties at runtime and to produce a test input exhibiting the issue when a function does not respect the specified relational property. We have also shown that our implementation can handle a wide variety of properties and code: we consider a large class of relational properties with several, possibly nested, function calls.

However, there are still some limitations, inherent to our use of sequential self-composition. First, in the case of relational properties linking functions with large bodies or a large number of functions, the size of the generated wrapper function may explode, leading to POs that cannot be handled by automated theorem provers or even generated by weakest precondition calculus. A first solution for this problem is to use the modularity of the approach to reduce the size of the function and prove sub-properties. However, it is not always possible to modify an existing implementation. Alternative methods, based on a generalization of the technique proposed in [9] for verifying **`\from`** clauses, and that do not rely on the generation of a wrapper function seem thus desirable. The notation of relational properties in the presence of side effects can be seen somewhat heavy to use. To make this notation more succinct, some shorthands for most common usages will be useful. The possibility to use runtime verification and testing is an important benefit in situations where the proof does not conclude. Furthermore, treatment of loops needs to be improved. In particular, it is not possible yet to specify "relational invariants" that would allow relating the behavior of a loop in two different contexts, while this is often necessary to complete the proof of a relational property. Solutions based on program products [2] look promising. Finally, as already mentioned, we need to extend our technique to handle potential aliases across the executions involved in a relational property.

Acknowledgment. The authors thank the FRAMA-C and PATHCRAWLER teams for providing the tools and support. Special thanks to François Bobot, Loïc Correnson, and Nicky Williams for many fruitful discussions, suggestions and advice. Many thanks to the anonymous referees for their helpful comments.

References

1. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Proc. of the 38th SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017). pp. 362–375. ACM (2017)
2. Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. J. Log. Algebr. Meth. Program. 85(5), 847–859 (2016)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. J. Mathematical Structures in Computer Science 21(6), 1207–1252 (2011)
4. Baudin, P., Bobot, F., Correnson, L., Dargaye, Z.: WP Plugin Manual v1.0 (2017), `http://frama-c.com/download/frama-c-wp-manual.pdf`
5. 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`
6. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004). pp. 14–25 (2004)
7. Bishop, P.G., Bloomfield, R.E., Cyra, L.: Combining testing and proof to gain high assurance in software: A case study. In: 24th International Symposium on Software Reliability Engineering (ISSRE 2013). pp. 248–257. IEEE (2013)
8. Blatter, L., Kosmatov, N., Gall, P.L., Prevosto, V.: RPP: automatic proof of relational properties by self-composition. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). vol. 10205, pp. 391–397. Springer (2017)
9. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V.: Functional dependencies of C functions via weakest pre-conditions. International Journal on Software Tools for Technology Transfer (STTT 2011) 13(5), 405–417 (2011)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proc. of the ACM Symposium on Applied Computing (SAC 2013). pp. 1230–1235. ACM (2013)
11. Floyd, R.W.: Assigning meanings to programs. Proc. of the American Mathematical Society Symposia on Applied Mathematics 19 (1967)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10) (1969)
13. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR - combining static verification and dynamic analysis. J. Autom. Reasoning 60(3), 337–363 (2018)
14. 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`
15. Leino, K.R.M., Polikarpova, N.: Verified calculations. In: 5th International Conference, on Verified Software: Theories, Tools, Experiments (VSTTE 2013), Revised Selected Papers. vol. 8164, pp. 170–190. Springer (2013)
16. Petiot, G., Botella, B., Julliand, J., Kosmatov, N., Signoles, J.: Instrumentation of annotated C programs for test generation. In: 14th International Working Conference on Source Code Analysis and Manipulation, (SCAM 2014). pp. 105–114. IEEE (2014)
17. Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., Julliand, J.: Your proof fails? testing helps to find the reason. In: Proc. of the 10th International Conference on Tests and Proofs (TAP 2016). vol. 9762, pp. 130–150. Springer (2016)

18. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language (2012), http://frama-c.com/download/e-acsl/e-acsl.pdf
19. Sousa, M., Dillig, I.: Cartesian Hoare Logic for Verifying k-safety Properties. In: Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016). pp. 57–69. ACM (2016)
20. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proc. of the International Symposium on Memory Management (ISMM 2017). pp. 47–58. ACM (2017)