Software Analyzers

# Frama-C/RPP

# Frama-C's relational property prover plug-in

v 0.0.1

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

[1] CEA Tech List, Software Safety and Security Laboratory, Saclay, F-91191
[2] CentraleSupelec, Université Paris-Saclay, 91190 Gif-sur-Yvette Franace

# Contents

Chapter 1

# Introduction

This documentation describes the grammar supported by the Frama-C/RPP plug-in and examples of usage.

RPP is a Frama-C plugin designed for the proof of relational properties. Relational properties are properties invoking any finite number of function calls of possibly dissimilar functions with possible nested calls. Informal examples include:

- the monotony of function $f$: $\forall x1, x2; x1 < x2 \Rightarrow f(x1) < f(x2)$
- decrypt an encrypted message: $\forall msg, key; msg = Decrypt(Encrypt(msg, key), key)$

RPP is based on the technique of self-composition and works like a preprocessor for the WP plug-in. Briefly speaking, the plug-in generates a wrapper function in which the various calls involved in the property are inlined. After that, the proof on the generated code proceeds like any other proof with WP[1]: proof obligations are generated and can be either discharged automatically by automatic theorem provers (e.g. *Alt-Ergo*, *CVC4*, *Z3*) or proven interactively (e.g. in *Coq*).

The plug-in also generates an axiomatic definition and additional annotations that allow using a relational property as a hypothesis for the proof of other properties in a completely automatic and transparent way.

[1] See WP own manual at https://frama-c.com/download/wp-manual-Chlorine-20180501.pdf

Chapter 2

# Grammar

RPP introduces an extension of the ACSL[1] specification language with new global annotations introduced by the keyword relational . These annotation must appear after the last function involved in the property, *i.e.* when all relevant functions are in scope.

A relational annotation consists in an extension to ACSL's grammar for predicates and terms that is shown in Figure 2.3 and Figure 2.4. A relational annotation is generally composed of three parts. First, it declares a set of universally quantified variables, that will be used to express the arguments of the calls that are related by the annotation. Then, it explicitly specifies the set of calls which are involved in the property of interest (the *relational-def* part of the annotation), using the \callset construct. Each call is defined using the \call construct and has its own identifier. Finally, the relational property itself is given as an ACSL predicate in the *relational-pred* part. As described in Figure 2.4, in addition to standard ACSL constructs, three new terms can be used. First, \callresult takes a *call-id* as parameter and refers to the value returned by the corresponding call in *relational-def*. Second, \callpure , denoting the value returned by a call `f(<args>)` to a pure function `f` with arguments `<args>`. This allows specifying relational properties over pure functions without the overhead required for handling side-effects. Figure 2.1 show two equivalent relational annotations where \callpure and \callresult are used respectively. Note that the *relational-def* part can be omitted if all involved calls are directly referred to as \callpure in the predicate. \callpure can be used recursively, i.e. a parameter of a called function can be the result of another function call.

```
/*@ assigns \result \from x;*/
int f(int x){
  return x*2;
}

/*@ relational \forall int x1,x2;
       \callset (\call (f,x1,id1),\call(f,x2,id2)) ==>
       x1 < x2 ==> \callresult (id1) < \callresult (id2);
    relational \forall int x1,x2; x1 < x2 ==> \callpure(f,x1) < \callpure(f,x2);
*/
```

Figure 2.1: Monotony of a function without side effect. The two relational annotations specify the same property.

[1] https://github.com/acsl-language/acsl/releases/latest

```
int y;

/*@ assigns y \from y;*/
void f(){
  y = y*2;
}

/*@ relational \callset (\call (f,id1),\call(f,id2)) ==>
          \at(y,Pre_id1) < \at(y,Pre_id2) ==> \at(y,Post_id1) < \at(y,Post_id2);
*/
```

Figure 2.2: Monotony of a function with side effect

| | | |
|---|---|---|
| *call-id* | ::= | *id* |
| *funct-param* | ::= | *relational-call-terms*$^+$ |
| *funct-name* | ::= | *poly-id* |
| *funct-call* | ::= | `\call` (*inline-opt*, *funct-name*, *funct-param*, *call-id*) |
| *call-parameter* | ::= | *funct-call*$^+$ |
| *relational-def* | ::= | `\callset` (*call-parameter*) |
| *relational-pred* | ::= | `\true` \| `\false` |
| | \| | *relational-terms* `==` *relational-terms* |
| | \| | *relational-terms* `!=` *relational-terms* |
| | \| | *relational-terms* `<=` *relational-terms* |
| | \| | *relational-terms* `>=` *relational-terms* |
| | \| | *relational-terms* `>` *relational-terms* |
| | \| | *relational-terms* `<` *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* |
| | \| | *relational-def* `==>` *relational-pred* |
| | \| | *relational-pred* |

Figure 2.3: Grammar for predicates in relational annotations

Finally, each *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 $L$. An example is shown in Figure 2.2.

| | | |
|---|---|---|
| *literal* | ::= | \true \| \false \| *int* \| *float* |
| *relational-label* | ::= | `Post_` *call-id* |
| | \| | `Pre_` *call-id* |
| *bin-op* | ::= | `+` \| `-` \| `*` \| `/` |
| *result-reference* | ::= | \callresult (*call-id*) |
| *pure-function-param* | ::= | *relational-call-terms*$^+$ |
| *inline-opt* | ::= | *int* |
| *pure-funct-name* | ::= | *poly-id* |
| *pure-funct-call* | ::= | \callpure (*inline-opt*, *pure-funct-name*, *pure-funct-param*) |
| *relational-call-terms* | ::= | *literal* |
| | \| | *pure-funct-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-funct-call* |

Figure 2.4: Grammar for terms in relational annotations

# Chapter 3

# Inlining option

`\callpure` and `\call` constructs have an optional argument that indicates the maximal depth that the inlining can reach in the wrapper. The default value of `1`, which is also used explicitly in Figure 3.1, for the first relational annotation, means that we inline the body of the functions once (i.e. if the function calls other functions, including itself, these calls themselves will not be inlined). However, since function `g` is annotation free, we will not be able to prove this property. This can be solved using a inlining parameter set to `2`. In that case, both functions `f` and `g` will thus be inlined. In the case this parameter is set to `0`, the call is kept as such in the wrapper.

```
int g(int x){
  return x*2;
}


/*@ assigns \result \from x;*/
int f(int y, int x) {
  return g(x-1)*y*3;;
}

/*@ relational \forall int x1,x2,y;
      x1 < x2 ==> \callpure(1,f,y,x1) < \callpure(1,f,y,x2);
    relational \forall int x1,x2,y;
      x1 < x2 ==> \callpure(2,f,y,x1) < \callpure(2,f,y,x2);*/
```

Figure 3.1: Example of relational properties for a recursive function

Chapter 4

# Side effect

By construction, self-composition require each function call to be operated on its own memory state, separated from the other calls in order to work. We thus create as many duplicates of all global memory chunks 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` annotations, including `\from` components. This constraint 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. Moreover, it allows detecting if a function is pure or has side effect.

```
int y;
int z;
int x;

/*@ assigns y \from y,z;*/
void f(){
  y = y*z;
}

/*@ relational \callset (\call(f,id1),\call(f,id2)) ==>
         \at(y,Pre_id1) < \at(y,Pre_id2) ==>
         \at(z,Pre_id1) < \at(z,Pre_id2) ==>
         \at(y,Post_id1) < \at(y,Post_id2);*/
```

Figure 4.1: Function `f` depends on `y` and `z` but not on `x`