Software Analyzers

Frama-C / WP

Phosphorus-20170501

WP Plug-in Manual

Frama-C Phosphorus-20170501

Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye

CEA LIST, Software Safety Laboratory

This work has been partially supported by the 'U3CAT' ANR project.

Contents

1 Introduction **7**

1.1 Installation 8

1.2 Tutorial 8

1.3 Weakest Preconditions 11

1.4 Memory Models 12

1.5 Arithmetic Models 14

2 Using the WP Plug-in **15**

2.1 Installing Provers 16

2.2 Graphical User Interface 17

2.3 Interactive Proof Editor 19

2.3.1 Display Modes 19

2.3.2 Tactics 20

2.3.3 Term Composer 20

2.3.4 Proof Script 21

2.3.5 Replaying Scripts 21

2.3.6 Available Tactics 23

2.3.7 Strategies 25

2.3.8 Custom Tactics and Strategies 26

2.4 Command Line Options 29

2.4.1 Goal Selection 29

2.4.2 Program Entry Point 30

2.4.3 Model Selection 30

2.4.4 Computation Strategy 31

2.4.5 Trigger Generation 32

2.4.6 Qed Simplifier Engine 32

2.4.7 Decision Procedures Interface 32

2.4.8 Generated Proof Obligations 35

2.4.9 Additional Proof Libraries 35

2.4.10 Linking ACSL Symbols to External Libraries 36
2.5 Plug-in Developer Interface 38
2.6 Proof Obligation Reports 38
2.7 Plug-in Persistent Data 41

3 WP Models **43**
3.1 Language of Proof Obligations 43
3.2 The Hoare Memory Model 44
3.3 Memory Models with Pointers 44
3.4 Hoare Variables mixed with Pointers 45
3.5 Hoare Variables for Reference Parameters 45
3.6 The Typed Memory Model 46
3.7 The Caveat Memory Model 47

4 WP Arithmetic Models **49**
4.1 Integral Models 49

5 WP Simplifier **51**
5.1 Logic Normalizations 51
5.2 Simplifier Engine (Qed) 52
5.3 Efficient WP Computation 53

Chapter 1

Introduction

This document describes the Frama-C/WP plug-in that uses external decision procedures to prove ACSL annotations of C functions.

The WP plug-in is named after *Weakest Precondition* calculus, a technique used to prove program properties initiated by Hoare [Hoa69], Floyd [Flo67] and Dijkstra [Dij68]. Recent tools implement this technique with great performance, for instance Boogie [Lei08] and Why [Fil03]. There is already a Frama-C plug-in, Jessie [MM09], developed at INRIA, that implements a weakest precondition calculus for C programs by compiling them into the Why language.

The WP plug-in is a novel implementation of such a *Weakest Precondition* calculus for annotated C programs, which focuses on parametrization *w.r.t* the memory model. It is a complementary work to the Jessie plug-in, which relies on a separation memory model in the spirit of Burstall's work [Bur72]. The Jessie memory model is very efficient for a large variety of well structured C-programs. However, it does not apply when low-level memory manipulations, such as heterogeneous casts, are involved. Moreover, Jessie operates by compiling the C program to Why, a solution that prevents the user from combining *weakest precondition calculus* with other techniques, such as the Value analysis plug-in.

The WP plug-in has been designed with cooperation in mind. That is, you may use WP for proving some annotations of your C programs, and prove others with different plug-ins. The recent improvements of the Frama-C kernel are then responsible for managing such partial proofs and consolidating them altogether.

This manual is divided into three parts. This first chapter introduces the WP plug-in, the *Weakest Precondition* calculus and *Memory Models*. Then, Chapter 2 details how to use and tune the plug-in within the Frama-C platform. Chapter 4 provides a description for the included memory models. Finally, we present in Chapter 5 the simplifier module and the efficient weakest precondition engine implemented in the WP plug-in.

1.1 Installation

The WP plug-in is distributed with the Frama-C platform. However, you should install at least one external prover in order to fulfill proof obligations. An easy choice is to install the Alt-Ergo theorem prover developed at INRIA[1]. See section 2.1 for installing other provers.

1.2 Tutorial

Consider the very simple example of a function that swaps the values of two integers passed by reference:

File **swap.c**

```
void swap(int *a,int *b)
{
  int tmp = *a ;
  *a = *b ;
  *b = tmp ;
  return ;
}
```

A simple, although incomplete, ACSL contract for this function can be:

File **swap1.h**

```
/*@ ensures A: *a == \old(*b) ;
  @ ensures B: *b == \old(*a) ;
  @*/
void swap(int *a,int *b) ;
```

You can run wp on this example with:

```
> frama-c -wp swap.c swap1.h
# frama-c -wp [...]
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing swap.c (with preprocessing)
[kernel] Parsing swap1.h (with preprocessing)
[wp] Running WP plugin...
[wp] Loading driver 'share/wp.driver'
[wp] Collecting axiomatic usage
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] [Alt-Ergo] Goal typed_swap_post_A : Valid
[wp] [Qed] Goal typed_swap_post_B : Valid
[wp] Proved goals:  2 / 2
    Qed:            1
    Alt-Ergo:       1
----------------------------------------------------------
Functions         WP  Alt-Ergo   Total  Success
swap               1     1 (12)    2      100%
----------------------------------------------------------
```

As expected, running WP for the swap contract results in two *proof obligations* (one for each 'ensures' clause). The first one is discharged internally by the Qed simplifier of WP, the second one is terminated by Alt-Ergo.

[1] Alt-Ergo: http://alt-ergo.lri.fr

You should notice the warning “Missing RTE guards”, emitted by the WP plug-in. That is, the *weakest precondition calculus* implemented in WP relies on the hypothesis that your program is runtime-error free. In this example, the `swap` function dereferences its two parameters, and these two pointers should be valid.

By default, the WP plug-in does not generate any proof obligation for verifying the absence of runtime errors in your code. Absence of runtime errors can be proved with other techniques, for instance by running the Value plug-in, or by generating all the necessary annotations with the RTE plug-in.

The simple contract for the `swap` function above is not strong enough to prevent runtime errors: pointers given as parameters might be invalid. Consider now the following new contract for `swap`:

File **swap2.h**

```
/*@ requires \valid(a) && \valid(b);
  @ ensures A: *a == \old(*b) ;
  @ ensures B: *b == \old(*a) ;
  @ assigns *a,*b ;
  @*/
void swap(int *a,int *b) ;
```

For simplicity, the WP plug-in is able to run the RTE plug-in for you, through the `-wp-rte` option. Now, WP reports that the function swap completely fulfills its contract, without any warnings regarding potential residual runtime errors:

```
> frama-c -wp -wp-rte swap.c swap2.h
# frama-c -wp -wp-rte [...]
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing swap.c (with preprocessing)
[kernel] Parsing swap2.h (with preprocessing)
[wp] Running WP plugin...
[wp] Loading driver 'share/wp.driver'
[wp] Collecting axiomatic usage
[rte] annotating function swap
[wp] 9 goals scheduled
[wp] [Alt-Ergo] Goal typed_swap_post_A : Valid
[wp] [Qed] Goal typed_swap_post_B : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access : Valid
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_2 : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access_3 : Valid
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_4 : Valid
[wp] [Qed] Goal typed_swap_assign_part1 : Valid
[wp] [Qed] Goal typed_swap_assign_part2 : Valid
[wp] [Qed] Goal typed_swap_assign_part3 : Valid
[wp] Proved goals:   9 / 9
    Qed:             6
    Alt-Ergo:        3
----------------------------------------------------------
Functions         WP   Alt-Ergo   Total   Success
swap               6      3 (17)     9      100%
----------------------------------------------------------
```

We have finished the job of validating this simple C program with respect to its specification, as reported by the *report* plug-in that displays a consolidation status of all annotations:

```
> frama-c -wp-verbose 0 [...] -then -report
# frama-c -wp -wp-rte [...]
[rte] annotating function swap
---------------------------------------------------------
Functions         WP   Alt-Ergo   Total  Success
swap               6     3 (17)     9      100%
---------------------------------------------------------
[report] Computing properties status...

-------------------------------------------------------------------------------
--- Properties of Function 'swap'
-------------------------------------------------------------------------------

[  Valid ] Post-condition 'A'
           by Wp.typed.
[  Valid ] Post-condition 'B'
           by Wp.typed.
[  Valid ] Assigns (file swap2.h, line 4)
           by Wp.typed.
[  Valid ] Assertion 'rte,mem_access' (file swap.c, line 3)
           by Wp.typed.
[  Valid ] Assertion 'rte,mem_access' (file swap.c, line 4)
           by Wp.typed.
[  Valid ] Assertion 'rte,mem_access' (file swap.c, line 4)
           by Wp.typed.
[  Valid ] Assertion 'rte,mem_access' (file swap.c, line 5)
           by Wp.typed.
[  Valid ] Default behavior
           by Frama-C kernel.

-------------------------------------------------------------------------------
--- Status Report Summary
-------------------------------------------------------------------------------
    8 Completely validated
    8 Total
-------------------------------------------------------------------------------
```

1.3 Weakest Preconditions

The principles of *weakest precondition calculus* are quite simple in essence. Given a code annotation of your program, say, an assertion Q after a statement *stmt*, the weakest precondition of P is by definition the "simplest" property P that must be valid before *stmt* such that Q holds after the execution of *stmt*.

Hoare triples. In mathematical terms, we denote such a property by a Hoare triple:

$$\{P\}\, \mathit{stmt}\, \{Q\}$$

which reads: *"whenever P holds, then after running stmt, Q holds"*.

Thus, we can define the weakest precondition as a function *wp* over statements and properties such that the following Hoare triple always holds:

$$\{\mathit{wp}(\mathit{stmt}, Q)\} \quad \mathit{stmt} \quad \{Q\}$$

For instance, consider a simple assignment over an integer local variable x; we have:

$$\{x + 1 > 0\} \quad \mathtt{x} = \mathtt{x} + \mathtt{1}; \quad \{x > 0\}$$

It should be intuitive that in this simple case, the *weakest precondition* for this assignment of a property Q over x can be obtained by replacing x with $x + 1$ in Q. More generally, for any statement and any property, it is possible to define such a weakest precondition.

Verification. Consider now function contracts. We basically have *pre-conditions*, *assertions* and *post-conditions*. Say function f has a precondition P and a post condition Q, we now want to prove that f satisfies its contract, which can be formalized by:

$$\{P\} \quad f \quad \{Q\}$$

Introducing $W = \mathit{wp}(f, Q)$, we have by definition of *wp*:

$$\{W\} \quad f \quad \{Q\}$$

Suppose now that we can *prove* that precondition P entails weakest precondition W; we can then conclude that P precondition of f always entails its postcondition Q. This proof can be summarized by the following diagram:

$$\frac{(P \Longrightarrow W) \qquad \{W\}\, f\, \{Q\}}{\{P\}\, f\, \{Q\}}$$

This is the main idea of how to prove a property by weakest precondition computation. Consider an annotation Q, compute its weakest precondition W across all the statements from Q up to the beginning of the function. Then, submit the property $P \Longrightarrow W$ to a theorem prover, where P are the preconditions of the function. If this proof obligation is discharged, then one may conclude the annotation Q is valid for all executions.

Termination. We must point out a detail about program termination. Strictly speaking, the *weakest precondition* of property Q through statement *stmt* should also ensure termination and execution without runtime errors.

The proof obligations generated by WP do not entail systematic termination, unless you systematically specify and validate loop variant ACSL annotations. Nevertheless, `exit` behaviors of a function are correctly handled by WP.

Regarding runtime errors, the proof obligations generated by WP assume your program never raises any of them. As illustrated in the short tutorial example of section 1.2, you should enforce the absence of runtime errors on your own, for instance by running the *value analysis* plug-in or the *rte generation* one and proving the generated assertions.

1.4 Memory Models

The essence of a *weakest precondition calculus* is to translate code annotations into mathematical properties. Consider the simple case of a property about an integer C-variable `x`:

```
x = x+1;
//@ assert P: x >= 0 ;
```

We can translate P into the mathematical property $P(X) = X \geq 0$, where X stands for the value of variable `x` at the appropriate program point. In this simple case, the effect of statement `x=x+1` over P is actually the substitution $X \mapsto X + 1$, that is $X + 1 \geq 0$.

The problem when applying *weakest precondition calculus* to C programs is dealing with *pointers*. Consider now:

```
p = &x ;
x = x+1;
//@ assert Q: *p >= 0 ;
```

It is clear that, taking into account the aliasing between `*p` and `x`, the effect of the increment of `x` cannot be translated by a simple substitution of X in Q.

This is where *memory models* come to rescue.

A memory model defines a mapping from values inside the C memory heap to mathematical terms. The WP has been designed to support different memory models. There are currently three memory models implemented, and we plan to implement new ones in future releases. Those three models are all different from the one in the Jessie plug-in, which makes WP complementary to Jessie.

`Hoare` model. A very efficient model that generates concise proof obligations. It simply maps each C variable to one pure logical variable.

However, the heap cannot be represented in this model, and expressions such as `*p` cannot be translated at all. You can still represent pointer values, but you cannot read or write the heap through pointers.

`Typed` model. The default model for WP plug-in. Heap values are stored in several *separated* global arrays, one for each atomic type (integers, floats, pointers) and an additional one for memory allocation. Pointer values are translated into an index into these arrays.

In order to generate reasonable proof obligations, the values stored in the global array are not the machine ones, but the logical ones. Hence, all C integer types are represented by mathematical integers and each pointer type to a given type is represented by a specific logical abstract datatype.

A consequence of having separated arrays is that heterogeneous casts of pointers cannot be translated in this model. For instance within this memory model, you cannot cast a pointer to `int` into a pointer to `char`, and then access the internal representation of the original `int` value into memory.

However, variants of the `Typed` model enable limited forms of casts. See chapter 4 for details.

`Bytes` model. (Not Implemented Yet). This is a low-level memory model, where the heap is represented as a wide array of bytes. Pointer values are exactly translated into memory addresses. Read and write operations from/to the heap are translated into manipulation of ranges of bits in the heap.

This model is very *precise* in the sense that all the details of the program are represented. This comes at the cost of huge proof obligations that are very difficult to discharge by automated provers, and generally require an interactive proof assistant.

Thus, each *memory model* offers a different trade-off between expressive power and ease of discharging proof obligations. The `Hoare` memory model is very restricted but generates easy proof obligations, `Bytes` is very expressive but generates difficult proof obligations, and `Typed` offers an intermediate solution.

Chapter 4 is dedicated to a more detailed description of memory models, and how the WP plug-in uses and *combines* them to generate efficient proof obligations.

Remark. The original `Store` and `Runtime` memory models are no longer available since WP version `0.7`; the `Typed` model replaces the `Store` one; the `Runtime` model will be entirely re-implemented as `Bytes` model in some future release.

1.5 Arithmetic Models

The WP plug-in is able to take into account the precise semantics of integral and floating-point operations of C programs. However, doing so generally leads to very complex proof obligations.

For tackling this complexity, the WP plug-in relies on several arithmetic models:

Machine Integer Model: The kernel options are used to determine if an operation is allowed to overflow or not. In case where overflows or downcasts are forbidden, the model uses mathematical operators in place of modulo ones to perform the computations.

For example, with kernel *default* settings, addition of regular `signed int` values is interpreted by mathematical addition over unbounded `integers` ; and the addition of two `unsigned int` is interpreted by the addition modulo 2^{32}.

The user shall set the kernel options `-(no)-warn-xxx` to precisely tune the model. Using `-rte` or `-wp-rte` will generate all the necessary assertions to be verified.

Natural Model: integer operations are performed on mathematical integers. In ACSL, explicit conversions between different integer types are still translated with *modulo*, though. Size of integers is also removed from type constrained, which avoids provers to run into deep exploration of large integer domains. Only signed-ness is kept from type constraints.

Except for the lesser constrained type assumptions, this model behaves like the machine integer one when all kernel options `-warn-xxx` are set. However, the model *does not* modifies them. Hence, using `-rte` or `-wp-rte` will generate a warning if some annotation might be not generated.

Float Model: floating-point operations are *defined* to be the mathematical ones *with* a rounding operation. This is fully consistent with the IEEE semantics. Most automated provers are not able to discharge the generated proof obligations. Special support for the Gappa theorem prover is available through Why3.

Real Model: floating-point operations are *transformed* on reals, with *no* rounding. This is completely unsound with respect to C and IEEE semantics. There is no way of recovering a correct or partial property from the generated proof obligations on floating-point operations with this model.

Remark: with all models, there are conditions to meet for WP proofs to be correct. Depending on the model used and the kernel options, those conditions may change. WP do not generate proof obligations for runtime errors on its own. Instead, it can discharge the annotations generated by the Value analysis plug-in, or by the RTE plug-in. Consider also using -wp-rte option.

Chapter 2

Using the WP Plug-in

The WP plug-in can be used from the Frama-C command line or within its graphical user interface. It is a dynamically loaded plug-in, distributed with the kernel since the Carbon release of Frama-C.

This plug-in computes proof obligations of programs annotated with ACSL annotations by *weakest precondition calculus*, using a parametrized memory model to represent pointers and heap values. The proof obligations may then be discharged by external decision procedures, which range over automated theorem provers such as Alt-Ergo [CCK06], interactive proof assistants like Coq [Coq10] and the interactive proof manager Why3 [BFMP11].

This chapter describes how to use the plug-in, from the Frama-C graphical user interface (section 2.2), from the command line (section 2.4), or from another plug-in (section 2.5). Additionally, the combination of the WP plug-in with the load and save commands of Frama-C and/or the `-then` command-line option is explained in section 2.7.

2.1 Installing Provers

The WP plug-in requires external provers to work. The natively supported provers are:

Prover	Version	Download	
Alt-Ergo	0.99.1, 1.01 and 1.30	http://alt-ergo.ocamlpro.com	[CCK06]
Coq	8.5	http://coq.inria.fr	[Coq10]
Why3	0.87	http://why3.lri.fr	[BFMP11]

Recent OPAM-provided versions should work smoothly.

Other provers, like Gappa, Z3, CVC3, CVC4, PVS, and many others, are accessible from WP through Why3. We refer the user to the manual of Why3 to handle specific configuration tasks.

Provers can be installed at any time, before or after the installation of Frama-C/WP. However, when using Coq and Why-3, it is better to install them before, or to re-configure and re-install WP, as explained below.

Configuration. When using Coq and Why-3, pre-compiled libraries are built during the compilation of Frama-C, which speed up the proof process in a significant way. This behavior can be turned on/off at configure time, typically:

```
# configure --disable-wp-coq --disable-wp-why3
```

Compilation. If you want to compile the Coq and Why-3 libraries manually, you can still run:

```
# make wp-coq wp-why3
# [sudo] make wp-coq-install
```

Remark. The Why [Fil03] prover is no longer supported since WP version 0.7.

2.2 Graphical User Interface

To use the WP plug-in with the GUI, you simply need to run the Frama-C graphical user interface. No additional option is required, although you can preselect some of the WP options described in section 2.4:

```
$ frama-c-gui [options...] *.c
```

As we can see in figure 2.1, the memory model, the decision procedure, and some WP options can be tuned from the WP side panel. Other options of the WP plug-in are still modifiable from the `Properties` button in the main GUI toolbar.

To prove a property, just select it in the internal source view and choose WP from the contextual menu. The `Console` window outputs some information about the computation. Figure 2.2 displays an example of such a session.

If everything succeeds, a green bullet should appear on the left of the property. The computation can also be run for a bundle of properties if the contextual menu is open from a function or behavior selection.

The options from the WP side panel correspond to some options of the plug-in command-line. Please refer to section 2.4 for more details. In the graphical user interface, there are also specific panels that display more details related to the WP plug-in, that we shortly describe below.

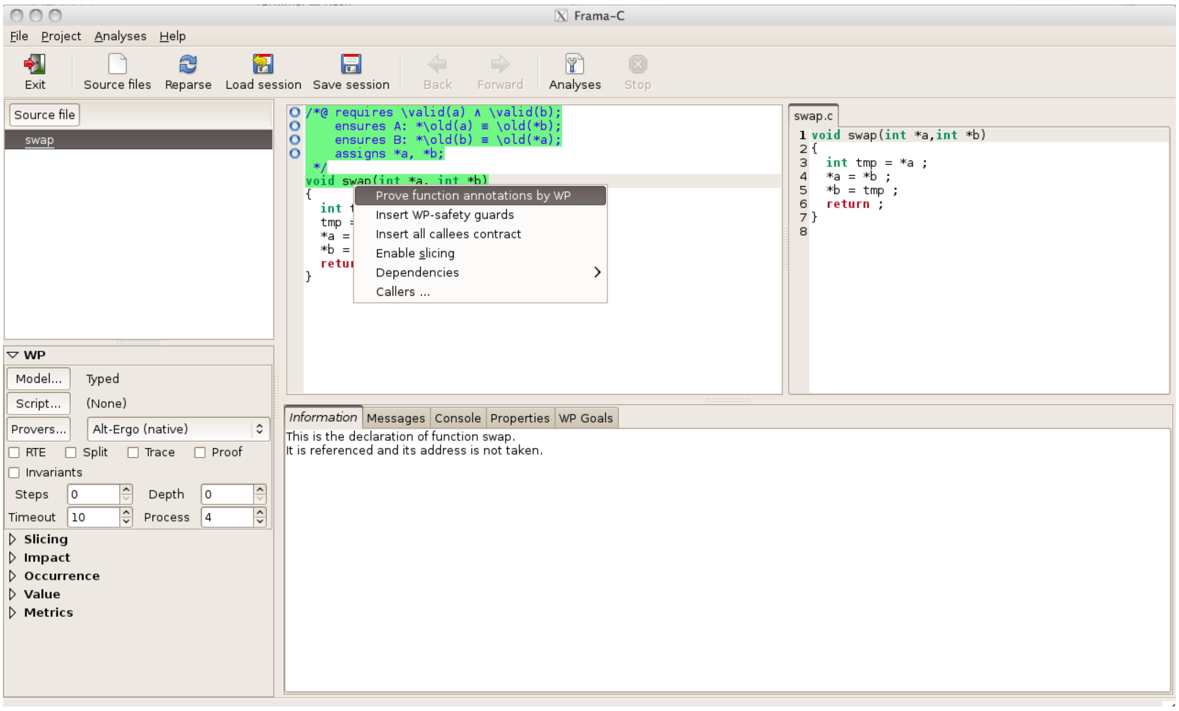
Source Panel. On the center of the Frama-C window, the status of each code annotation is reported in the left margin. The meaning of icons is the same for all plug-ins in Frama-C and more precisely described in the general user's manual of the platform. The status emitted by the WP plug-in are:

Icons for properties:	
○	No proof attempted.
🟠	The property has not been validated.
◐	The property is *valid* but has dependencies.
🟢	The property and *all* its dependencies are *valid*.

WP Goals Panel. This panel is dedicated to the WP plug-in. It shows the generated proof obligations and their status for each prover. By clicking on a prover column, you can also submit a proof obligation to a prover by hand. Right-click provides more options depending on the prover, such as proof-script editing for Coq.

Interactive Proof Editor. From the Goals Panel view, you can double-click on a row and open the *interactive proof editor* panel as described in section 2.3.

Properties Panel. This panel summarizes the consolidated status of properties, from various plug-ins. This panel is not automatically refreshed. You should press the `Refresh` button to update it. This panel is described in more details in the general Frama-C platform user's manual.



```
/*@ requires \valid(a) ∧ \valid(b);
    ensures A: *\old(a) ≡ \old(*b);
    ensures B: *\old(b) ≡ \old(*a);
    assigns *a, *b;
  */
void swap(int *a, int *b)
{
  int t
  tmp =
  *a =
  *b =
  retu
}
```

```
void swap(int *a,int *b)
{
  int tmp = *a ;
  *a = *b ;
  *b = tmp ;
  return ;
}
```

Model... Typed

Script... (None)

Provers... Alt-Ergo (native)

☐ RTE ☐ Split ☐ Trace ☐ Proof

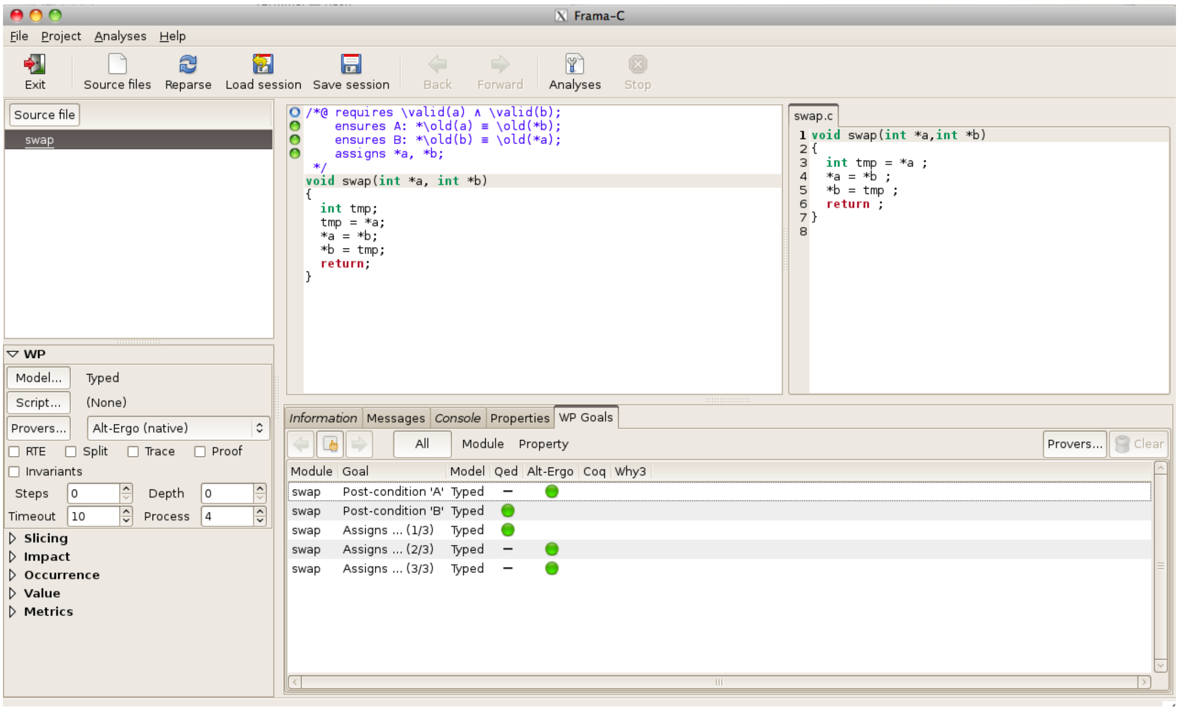
☐ Invariants

Steps 0 Depth 0

Timeout 10 Process 4

```
This is the declaration of function swap.
It is referenced and its address is not taken.
```

Figure 2.1: WP in the Frama-C GUI



```
/*@ requires \valid(a) ∧ \valid(b);
    ensures A: *\old(a) ≡ \old(*b);
    ensures B: *\old(b) ≡ \old(*a);
    assigns *a, *b;
  */
void swap(int *a, int *b)
{
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}
```

```
void swap(int *a,int *b)
{
  int tmp = *a ;
  *a = *b ;
  *b = tmp ;
  return ;
}
```

Model... Typed

Script... (None)

Provers... Alt-Ergo (native)

☐ RTE ☐ Split ☐ Trace ☐ Proof

☐ Invariants

Steps 0 Depth 0

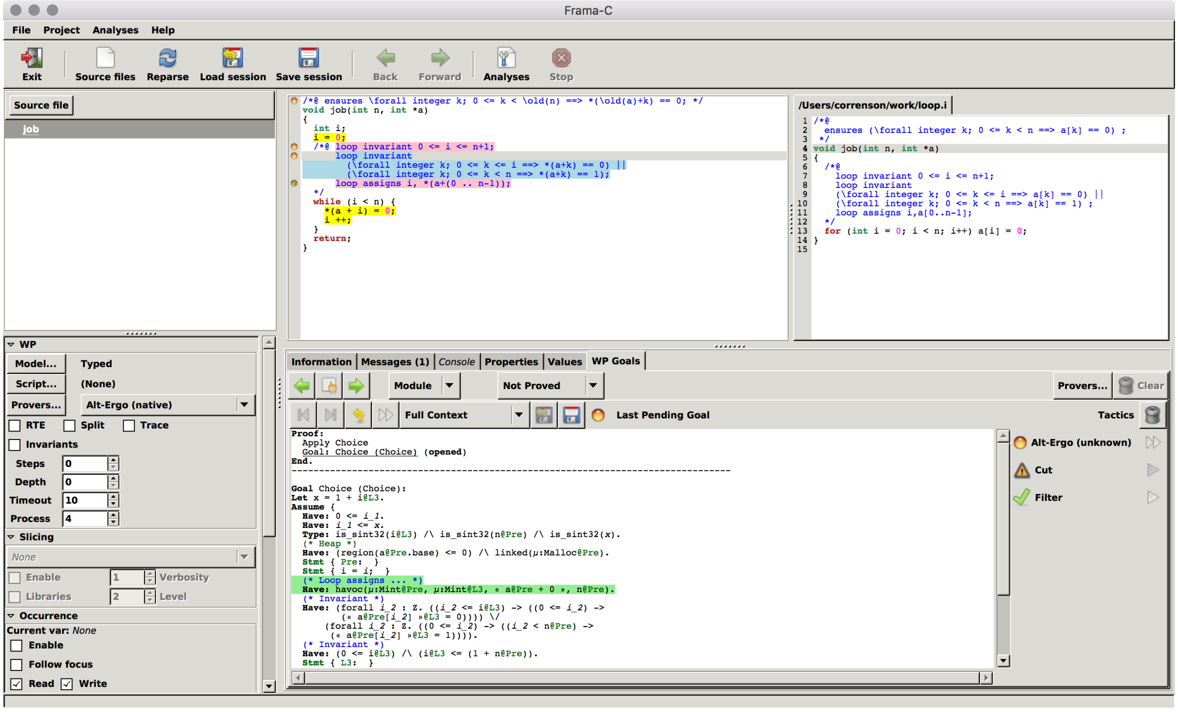
Timeout 10 Process 4

Module	Goal	Model	Qed	Alt-Ergo	Coq	Why3
swap	Post-condition 'A'	Typed	—	●		
swap	Post-condition 'B'	Typed	●			
swap	Assigns ... (1/3)	Typed	●			
swap	Assigns ... (2/3)	Typed	—	●		
swap	Assigns ... (3/3)	Typed	—	●		

Figure 2.2: WP run from the GUI

2.3 Interactive Proof Editor

This panel focus on one goal generated by WP, and allow the user to visualize the logical sequent to be proved, and to interactively decompose a complex proof into smaller pieces by applying *tactics*.



Model... Typed
Script... (None)
Provers... Alt-Ergo (native)
□ RTE □ Split □ Trace
□ Invariants
Steps 0
Depth 0
Timeout 10
Process 4
Slicing
None
□ Enable 1 Verbosity
□ Libraries 2 Level
Occurrence
Current var: *None*
□ Enable
□ Follow focus
☑ Read ☑ Write

```
/*@ ensures \forall integer k; 0 <= k < \old(n) ==> *(\old(a)+k) == 0; */
void job(int n, int *a)
{
  int i;
  i = 0;
  /*@ loop invariant 0 <= i <= n+1;
      loop invariant
        (\forall integer k; 0 <= k <= i ==> *(a+k) == 0) ||
        (\forall integer k; 0 <= k < n ==> *(a+k) == 1);
      loop assigns i, *(a+(0 .. n-1));
  */
  while (i < n) {
    *(a + i) = 0;
    i ++;
  }
  return;
}
```

/Users/correnson/work/loop.i

```
/*@
  ensures (\forall integer k; 0 <= k < n ==> a[k] == 0) ;
 */
void job(int n, int *a)
{
  /*@
    loop invariant 0 <= i <= n+1;
    loop invariant
    (\forall integer k; 0 <= k <= i ==> a[k] == 0) ||
    (\forall integer k; 0 <= k < n ==> a[k] == 1) ;
    loop assigns i,a[0..n-1];
  */
  for (int i = 0; i < n; i++) a[i] = 0;
}
```

```
Proof:
  Apply Choice
  Goal: Choice (Choice) (opened)
End.
--------------------------------------------------------------------------------

Goal Choice (Choice):
Let x = 1 + i@L3.
Assume {
  Have: 0 <= i_1.
  Have: i_1 <= x.
  Type: is_sint32(i@L3) /\ is_sint32(n@Pre) /\ is_sint32(x).
  (* Heap *)
  Have: (region(a@Pre.base) <= 0) /\ linked(μ:Malloc@Pre).
  Stmt { Pre:  }
  Stmt { i = i;  }
  (* Loop assigns ... *)
  Have: havoc(μ:Mint@Pre, μ:Mint@L3, « a@Pre + 0 », n@Pre).
  (* Invariant *)
  Have: (forall i_2 : Z. ((i_2 <= i@L3) -> ((0 <= i_2) ->
          (« a@Pre[i_2] »@L3 = 0)))) \/
      (forall i_2 : Z. ((0 <= i_2) -> ((i_2 < n@Pre) ->
       (« a@Pre[i_2] »@L3 = 1)))).
  (* Invariant *)
  Have: (0 <= i@L3) /\ (i@L3 <= (1 + n@Pre)).
  Stmt { L3:  }
```

Figure 2.3: Interactive Proof Editing

The general structure of the panel is illustrated figure 2.3. The central text area prints the logical sequent to proved. In consists of a formula to `Prove` under the hypotheses listed in the `Assume` section. Each hypothesis can consists of :

`Type:` formula expressing a typing constraint;
`Init:` formula characterizing global variable initialisation;
`Have:` formula from an assertion or an instruction in the code;
`When:` condition from a simplification performed by `Qed`;
`If:` structured hypothesis from a conditional statement;
`Either:` structured disjunction from a switch statement.
`Stmt:` labels and C-like instructions representing the memory updates during code execution;

2.3.1 Display Modes

There are several modes to display the current goal:

`Autofocus:` filter out clauses not mentioning *focused* terms (see below);
`Full Context:` disable autofocus mode — all clauses are visible;
`Unmangled Memory:` autofocus mode with low-level details of memory model;
`Raw Obligation:` no autofocus and low-level details of memory model.

Remark: the fold/unfold operations only affect the goal display. It does not *transform* the goal to be proven.

The autofocus mode is based on a ring of *focused terms*. Clicking a term of a clause automatically focus this term. Shift-clicking a term adds the term to the focus ring. When autofocus mode is active, only the clauses that contains a *focused* term are displayed. Hidden clauses are mentioned by an ellipsis `[...]`.

Low-level details of the memory model are normally hidden, and represented by C-like instructions such as:

```
    Stmt { Label A: a.f[0] = y@Pre; }
```

This reads as follows: a program point is defined by the label `A`. At this point, the left-value `a.f[0]` receives the value that variable `y` holds at label `Pre`. More generally, `lv@L` means the value of l-value `lv` at label `L:`, and for more complex expression, `« e »@L` means the expression `e` evaluated at label `L`. Redundant labels are removed when possible. This is a short-hand for ACSL notation `\at(e,L)` but is generally more readable.

Sometimes, some memory operations can not be rendered as C instructions, typically after transforming a goal so far. In such situations, the memory model encoding might appear with terms like `µ:Mint@L`.

With memory model unmangled, the encoding in logic formulae is revealed and no label are displayed.

2.3.2 Tactics

The right panel display a palette of tactics to be applied on the current goal. Tooltips are provided to help the user understanding how to configure and run tactics.

Only applicable tactics are displayed, with respect to current term or clause selected. Many tactics can be configured by the user to tune their effect. Click on the tactic button to toggle its control panel. Once a tactic is correctly configured, it can be applied by clicking its « Play » button.

2.3.3 Term Composer

Some tactic require one or several terms to be selected. In such case, the normal view display the selected term. It can be edited by buttons in the view, like a `RPN` calculator. More buttons appear with respect to already selected terms. Numerical constants can be composed, and combined with selected terms.

Typically, the composer displays a stack of values, like for instance:

```
   A: 45
   B: a[0]@Pre (int)
```

In such a case, the user can select the value `45` with the `Select A` button, or add the two numbers with the `Add A+B` button.

Sometimes, like for the Instance tactic, a *range* of numerical values can be selected. In such a case, when two numbers are selected, a special button `Select A..B` appears.

The list of all available composer buttons is displayed by the `Help` button.

A composer worth to be mentioned is `Destruct`, typically available on complex expressions. It allows to decompose a value into its sub-components. For instance, destructuring the value `B` above will reveal the address `« a+0 »@Pre` and memory `µ:Mint@Pre`.

2.3.4 Proof Script

The top toolbar upon the goal display show the current status of the goal and the number of pending goals. The media buttons allow to navigate in the proof tree.

`Next/Prev:` navigate among the list of pending (non proved) sub-goals;
`Forward:` goes to the next pending sub-goal;
`Backward:` cancel the current tactic and prover results;
`Clear:` restart all the interactive proof from the initial goal.

A sketch of current proof is displayed on top of the goal ; each step is clickable to navigate into the proof. Only the path leading to the current node is unfolded.

When all pending sub-goals have been proved, the initial goal is marked proved by Tactical in the goal list panel. It is time to save the script. A button is also available to replay the saved script, if any. Saving and replay are also accessible from the list of goals, in the popup menu of the `Script` prover column.

2.3.5 Replaying Scripts

Editing scripts interactively allows the user to finish the proofs. Once proofs are saved, he must be able to replay them from the command line. To ease the process, the following options are available to the user:

`-wp-session <dir>` to setup a directory where scripts are saved in;
`-wp-prover tip` for incrementally building and updating the session scripts;
`-wp-prover script` for replaying saved scripts only, as they are;

The `script` prover only runs the proof scripts edited by the user from the TIP, including the scripts being complete or known to being stuck at some sub-goal. The other proof obligations are transmitted to other provers, if some are provided.

This mode is well suited for replaying a proof bench, by using a combination of provers such as `-wp-prover script,alt-ergo`. Moreover, the `script` prover never modifies the proof session and the proof scripts.

The `tip` prover is similar, except that it never runs sub-goals that are known to be stuck but updates the proof scripts on success or when an automated proof fails. Using the `tip` prover is less time consuming and eventually prepares new scripts for failed proofs to be edited later under the TIP.

A typical proof session consists then in the following stages:

a. Collecting the automated proofs and preparing for the TIP.

```
frama-c [...] -wp-prover tip,alt-ergo
```

This runs all existing scripts (none at the very beginning) in success-mode only, and try Alt-Ergo on the others. Failed proofs lead to new empty scripts created.

b. Running the TIP.

```
frama-c-gui [...] -wp-prover tip
```

This mode only runs existing scripts (typically prepared in the previous phase) in success-mode only, which is quite fast. Finally, the GUI is opened and the user can enter the TIP and edit the proofs.

Most goals are reported not to be proved, because automated proof is deactivated since no other prover than `tip` is specified. However, by filtering only those proof scripts that requires completion, only the relevant goals appear. The user has to save its edited proof scripts to re-run them later.

Any number of phase a. and b. can be executed and interleaved. This incrementally builds the set of proof scripts that are required to complement the automated proofs.

c. Consolidating the Bench.

```
frama-c [...] -wp-prover script,alt-ergo
```

This mode replays the automated proofs and the interactive ones, re-running Alt-Ergo on every WP goals and every proof tactic sub-goals. The user scripts are never modified — this is a replay mode only.

2.3.6 Available Tactics

Absurd Contradict an Hypothesis
The user can select an hypothesis H, and change the goal to $\neg H$:

$$\Delta, H \models G \qquad \triangleright \qquad \Delta \models \neg H$$

Array Decompose array access-update patterns
The use select an expression $e \equiv a[k_1 \mapsto v][k_2]$. Then:

$$\Delta \models G \qquad \triangleright \qquad \begin{array}{ll} \Delta,\ k_1 = k_2,\ e = v & \models G \\ \Delta,\ k_1 \neq k_2,\ e = a[k_2] & \models G \end{array}$$

Choice Select a Goal Alternative
When the goal is a disjunction, the user select one alternative and discard the others:

$$\Delta \models \Gamma, G \qquad \triangleright \qquad \Delta \models G$$

Compound Decompose compound equalities
When the user select an equality between two records, it is decomposed field by field.

$$a = b \qquad \triangleright \qquad \bigwedge a.f_i = b.f_i$$

Contrapose Swap and Negate Hypothesis with Conclusion
The user select an hypothesis (typically, a negation) and swap it with the goal.

$$\Delta, H \models G \qquad \triangleright \qquad \Delta, \neg G \models \neg H$$

Cut Use Intermerdiate Hypothesis The user introduce a new clause C with the composer to prove the goal. There two variants of the tactic, made available by a menu in the tactic panel.

The Modus-Ponens variant where the clause C is used as an intermediate proof step:

$$\Delta \models G \qquad \triangleright \qquad \begin{array}{ll} \Delta & \models C \\ \Delta, C & \models G \end{array}$$

And the Case Analysis variant where the clause C is used with a split:

$$\Delta \models G \qquad \triangleright \qquad \begin{array}{l} \Delta,\ \ C \models G \\ \Delta, \neg C \models G \end{array}$$

Filter Erase Hypotheses
The tactic is always applicable. It removes hypotheses from the goal on a variable used basis. When variables are compounds (record and arrays) a finer heuristics is used to detect which parts of the variable is relevant. A transitive closure of dependencies is also used. However, it is always possible that too many hypotheses are removed.

The tactic also have a variant where only hypotheses *not relevant* to the goal are retained. This is useful to find absurd hypotheses that are completely disjoint from the goal.

Havoc Go Through Assigns
This is a variant of the `Lemma` tactic dedicated to `Havoc` predicate generate by complex assigns clause. The user select an address, and if the address is not assigned by the `Havoc` clause, the memory at this address is unchanged.

Instance Instantiate properties
The user selects an hypothesis with one or several ∀ quantifiers, or an ∃ quantified goal. Then, with the composer, the use choose to instantiate one or several of the quantified parameters. In case of ∀ quantifier over integer, a range of values can be instantiated instead.

When instantiating hypothesis with an expression e:

$$\Delta, \forall x\, P(x) \models G \qquad \triangleright \qquad \Delta, P(e) \models G$$

When instantiating with a range of values $n \ldots m$:

$$\Delta, \forall x\, P(x) \models G \qquad \triangleright \qquad \Delta, P(n) \ldots P(m) \models G$$

When instantiating a goal with an expression e:

$$\Delta \models \exists x\, G(x) \qquad \triangleright \qquad \Delta \models G(e)$$

Lemma Search & Instantiate Lemma
The user start by selecting a term in the goal. Then, the search button in the tactic panel will display a list of lemma related to the term. Then, he can instantiate the parameters of the lemma, like with the Instance tactic.

Intuition Decompose with Conjunctive/Disjunctive Normal Form
The user can select an hypothesis or a goal with nested conjunctions and disjunctions. The tactics then computes the conjunctive or disjunctive normal form of the selection and split the goal accordingly.

Range Enumerate a range of values for an integer term
The user select any integer expression e in the proof, and a range of numerical values $a \ldots b$. The proof goes by case for each $e = a \ldots e = b$, plus the side cases $e < a$ and $e > b$:

$$\Delta \models G \qquad \triangleright \qquad \begin{array}{ll} \Delta, e < a & \models G \\ \Delta, e = a & \models G \\ & \vdots \\ \Delta, e = b & \models G \\ \Delta, e > b & \models G \end{array}$$

Rewrite Replace Terms
This tactic uses an equality in hypothesis to replace each occurrence of term by another one. The tactic exists with two variants: the left-variant which rewrites a into b from equality $a = b$, and the right-variant which rewrites b into a from equality $a = b$. The original equality hypothesis is removed from the goal.

$$\Delta, a = b \models G \qquad \triangleright \qquad \Delta[a \leftarrow b] \models G[a \leftarrow b]$$

Separated Expand Separation Cases
This tactic decompose a `separated`(a, n, b, m) predicate into its four base cases: a and b have different bases, $a + n \leq b$, $b + m \leq a$, and $a[0..n-1]$ and $b[0..m-1]$ overlaps. The regions are separated in the first three cases, and not separated in the overlapping case. This is kind of normal disjunctive form of the separation clause.

Split Decompose Logical Connectives and Conditionals
This is the most versatile available tactic. It decompose merely any logical operator following the sequent calculus rules. Typically:

$$\begin{array}{rcl}
\Delta, (H_1 \vee H_2) \models G & \triangleright & \Delta, H_1 \models G \\
 & & \Delta, H_2 \models G \\
\Delta \models (G_1 \wedge G_2) & \triangleright & \Delta \models G_1 \\
 & & \Delta \models G_2 \\
\Delta, H?P : Q \models G & \triangleright & \Delta, \quad H, P \models G \\
 & & \Delta, \neg H, Q \models G \\
\ldots & &
\end{array}$$

When the user selects a arbitrary boolean expression e, the tactic is similar to the Cut one:

$$\begin{array}{rcl}
\Delta \models G & \triangleright & \Delta, \quad e \models G \\
 & & \Delta, \neg e \models G
\end{array}$$

Finally, when the user select a arithmetic comparison over a and b, the tactics makes a split over $a = b$, $a < b$ and $a > b$:

$$\begin{array}{rcl}
\Delta \models G & \triangleright & \Delta, a < b \models G \\
 & & \Delta, a = b \models G \\
 & & \Delta, a > b \models G
\end{array}$$

Definition Unfold predicate and logic function definition
The user simply select a term $f(e_1, \ldots, e_n)$ or a predicate $P(e_1, \ldots, e_n)$ which is replaced by its definition, when available.

2.3.7 Strategies

Strategies are heuristics that generate a prioritized bunch of tactics to be tried on the current goal. Few built-in strategies are provided by the WP plug-in ; however, the user can extends the proof editor with custom ones, as explained in section 2.3.8 below.

To run strategies, the interactive proof editor provide a single button `Strategies` in the tactic panel. Configure the heuristics you want to include in your try, then click the button. The generated with highest priority is immediately applied. The proof summary now display `backtrack` buttons revealing proof nodes where alternative tactics are available. You can use those backtracking button to cycle over the generated tactics.

Of course, strategies are meant to be used multiple times, in sequence. Recall that strategies apply highest priority tactic first, on the current goal. When using strategies several times, you shall see several `backtrack`ing buttons in your proof script. You backtrack from any point at any time.

You shall also alternate strategies *and* manually triggered tactics. Though, strategies are only used to *infer* or *suggest* interesting tactics to the user. Once your are finished with your proved, only the tactics are saved in the script, not the strategies used to find them. Hence, replaying a script generated with strategies would not involve backtracking any more. The script will directly replay your chosen alternatives.

2.3.8 Custom Tactics and Strategies

The proof editor and script runner can be extended by loading additional plug-ins. These plug-ins are regular OCaml files to be loaded with the kernel `-load-module` option. They will be compiled by Frama-C against its API. The WP plug-in exports a rich API to extend the proof editor with new tactics, strategies, and even term-composer tools.

It is not possible to reproduce here the complete API ; it is better to use the automatically generated HTML documentation from WP's sources. We only provide here a quick tour of this API, as a tutorial on how to implement a basic custom strategy.

The main extension points of the WP plug-in's API are the following ones:

`Wp.Tactical.tactical`	Base-class definition of custom Tactic.
`Wp.Strategy.heuristic`	Base-class definition of custom Strategy.
`Wp.Auto.*`	Pre-defined tactics and strategies.
`Wp.Tactical.register`	Registration point for custom Tactics.
`Wp.Strategy.register`	Registration point for custom Strategies.
`Wp.Tactical.add_composer`	Registration point for custom term composer.

Warning: It is technically possible to break the logical soundness of WP when using custom tactics crafted by hand. Fortunately, using only pre-defined tactics in custom strategies will be always safe, even if your heuristic generates crazily stupid alternatives to solve a goal. The point with custom *tactics* is that you might transform a sequent *without preserving* the equivalence with the original goal if you make some mistakes into your custom code. This is the same problem as using ACSL axioms instead of lemmas. So, use custom tactics carefully, and prefer custom strategies when possible.

To build a custom strategy, the typical boilerplate code is as follows:

```
 (* file : dummy.ml *)
open Wp

class dummy : Strategy.heuristic =
object
    method id = "MyStrategy.dummy" (* required, must be unique *)
    method title = "Split Goal"  (* visible in Strategy panel *)
    method descr = "Simply split conjunctions in goal" (* idem *)
    method search push sequent = (* heuristic code *)
      let goal = snd sequent in
      match Repr.pred goal with
      | And _ →
          let selection = Tactical.(Clause(Goal goal)) in
          push (Auto.split ~priority:2.0 selection)
      | _ → ()
end

(* Register the strategy *)
```

```
let () = Strategy.register (new dummy)
```

Then, simply extend your command line with the following options to make your strategy available from the interactive proof editor:

```
> frama-c-gui -load-module dummy.ml [...]
```

Note: Loading custom strategies is only required when running the graphical user interface (`frama-c-gui`). When replaying scripts from the command line (`frama-c`), only custom tactics and custom composers actually involved in proofs are required to be loaded.

The example custom strategy above is structured as follows. First we open the module `Wp` to simplify access to the API. A custom strategy must be an instance of class-type `Strategy.heuristic`, and we use a coercion here to explicit types. Methods #`id`, #`title` and #`descr` are required and describes the strategy, making it available from the tactic panel in the graphical user interface.

The actual heuristic code takes place in method #`search` which has the following type (consult the html API for details):

```
    method search : (Strategy.strategy → unit) → Conditions.sequent → unit
```

This method takes two parameters: a strategy registration callback and the sequent to prove. Each heuristic is supposed to register any number of strategies to be tried on the provided sequent. In turn, each strategy is a record consisting of a priority, a tactic, a target selection for the tactic and its arguments. It is possible to build such a record by hand, using custom or predefined tactics. However, it is much more convenient to use the helper functions from module `Auto` that directly build strategies.

In the example above, we inspect the structure of the goal, and when a conjunction is detected (`And _`), we decide to register a split tactic, thanks to the helper function `Auto.split`. Default priority is `1.0` by convention. Pre-installed strategies would only use range $[0.5 \ldots 2.0]$ of priorities. You can use any value you want inside or outside this range. In the example below, we assign a high priority to the split of goal conjunctions, meaning that such a split should be tried first.

Using Selections. Tactics always need a `selection` target. Moreover, some tactics require additional parameters, also to be provided as `selection` values. Typically, consider the `Auto.range` tactic:

```
   val Auto.range : ?priority:float -> selection -> vmin:int -> vmax:int -> strategy
```

Here the selection argument shall targets the expression to be enumerated in range `vmin..vmax`. Selections must refer to a term that pre-exists in the sequent. You must indicate to the WP proof engine how to rebuild this term from the sequent. Hence, if the C-code or the ACSL specification change, WP still has a chance to rebuild the same selection from the updated sequent.

Selections are easy to build. There are five basic forms, as described below:

```
  type Tactical.selection =
    | Empty (** no selection *)
    | Clause of clause (** selects an full hypothesis or the full goal *)
    | Inside of clause * Lang.F.term (** selects a sub−term of an hypothesis or goal *)
    | Compose of compose (** a calculus from several sub−selections *)
```

```
and Tactical.clause =
  | Goal of Lang.F.pred
  | Step of Conditions.step
```

It is also possible to build selections from sequent by explicit lookup:

```
val Strategy.select_e : Conditions.sequent → Lang.F.term → Tactical.selection
val Strategy.select_p : Conditions.sequent → Lang.F.pred → Tactical.selection
```

Composition allows you to build new terms from existing ones, like when using the term composer from the graphical user interface. You access composers by their name, like in the term composer. The API for building new terms is as follows:

```
val Tactical.int : int → Tactical.selection
val Tactical.cint : Integer.t → Tactical.selection
val Tactical.range : int → int → Tactical.selection
val Tactical.compose : string → Tactical.selection list → Tactical.selection
```

For instance, provided you have two selected terms `a` and `b`, you can build their sum using `compose "wp:add"[a;b]`. The name of composers can be obtained from the graphical user interface.

Exploring Sequents. The clauses refer to parts of the provided sequent. Typically, a sequent consists of a sequence of hypothesis, and a goal to prove. Each hypothesis is represented by a `step`, consisting either of single hypothesis or a more structured condition (branch, cases, *etc.*):

```
type Conditions.sequent = sequence * Lang.F.pred
and sequence = .... step list ... (* private type *)
and step = { condition : condition ; ... }
and condition =
      | Have of Lang.F.pred (** hypothesis *)
      | Init of Lang.F.pred (** C−initializer initialization clause *)
      | Type of Lang.F.pred (** C/ACSL type constraints *)
      | Core of Lang.F.pred (** Common hypothesis factorization from WP *)
      | When of Lang.F.pred (** hypothesis from tactical or simplification *)
      | Branch of Lang.F.pred * sequence * sequence (** If−Then−Else *)
      | Either of sequence list (** Disjunction of Cases *)
val iter : (step → unit) → sequence → unit
```

When you walk through a sequence of hypothesis, you shall reuse the provided steps to build clauses and selections.

Exploring Formulae . The constituent of hypothesis and goals are logical terms and predicates. You shall use module `Repr.term` and `Repr.pred` to access the internal representation of formulae. There are many constructors for terms and predicates, simply browse the html documentation to have an overview. Many properties about terms and predicates are directly obtained from the `Lang.F` module. The API is quite rich, so use the html documentation to get details.

2.4 Command Line Options

The best way to know which options are available is to use:

```
# frama-c -wp-help
```

The WP plug-in generally operates in three steps:

1. Annotations are selected to produce a control-flow graph of elementary statements annotated with hypotheses and goals.
2. Weakest preconditions are computed for all selected goals in the control-flow graph. Proof obligations are emitted and saved on disk.
3. Decision procedures (provers) are run to discharge proof obligations.

The WP options allow to refine each step of this process. It is very convenient to use them together with the standard `-then` option of Frama-C, in order to operate successive passes on the project. See section 2.7 for details.

2.4.1 Goal Selection

This group of options refines the selection of annotations for which proof obligations are generated. By default, all annotations are selected. A property which is already proved – by WP or by any other plug-in – does not lead to any proof-obligation generation, unless the property is individually selected from the graphical user interface of the programmatic API.

`-wp` generates proof obligations for all (selected) properties.

`-wp-fct <`$\mathtt{f}_1$`,...,`$\mathtt{f}_n$`>` selects annotations of functions $\mathtt{f}_1$,...,$\mathtt{f}_n$ (defaults to all functions).

`-wp-skip-fct <`$\mathtt{f}_1$`,...,`$\mathtt{f}_n$`>` removes annotations of functions $\mathtt{f}_1$,...,$\mathtt{f}_n$ (defaults to none).

`-wp-bhv <`$\mathtt{b}_1$`,...,`$\mathtt{b}_n$`>` selects annotations for behaviors $\mathtt{b}_1$,...$\mathtt{b}_n$ (defaults to all behaviors) of the selected functions.

`-wp-prop <`$\mathtt{p}_1$`,...,`$\mathtt{p}_n$`>` selects properties having $\mathtt{p}_1$ or ...$\mathtt{p}_n$ as tagname (defaults to all properties). You may also replace a tagname by a `@<category>` of properties.
Recognized categories are: `@lemma`, `@requires`, `@assigns`, `@ensures`, `@exits`, `@assert`, `@invariant`, `@variant`, `@breaks`, `@continues`, `@returns`, `@complete_behaviors`, `@disjoint_behaviors`.
Properties can be prefixed with a minus sign to *skip* the associated annotations. For example `-wp-prop="-@assigns"` removes all `assigns` and `loop assigns` properties from the selection.

`-wp-(no)-status-all` includes in the goal selection all properties regardless of their current status (default is: `no`).

`-wp-(no)-status-valid` includes in the goal selection those properties for which the current status is already 'valid' (default is: `no`).

`-wp-(no)-status-invalid` includes in the goal selection those properties for which the current status is already 'invalid' (default is: `no`).

`-wp-(no)-status-maybe` includes in the goal selection those properties with an undetermined status (default is: `yes`).

Remark: options `-wp-status-xxx` are not taken into account when selecting a property by its name or from the GUI.

2.4.2 Program Entry Point

The generic Frama-C options dealing with program entry point are taken into account by WP plug-in as follows:

`-main <f>` designates `f` to be the main entry point (defaults to `main`).

`-lib-entry` the main entry point (as defined by option `-main`) is analyzed regardless of its initial context (default is no).

These options impact the generation of proof-obligations for the "`requires`" contract of the main entry point. More precisely, if there is a main entry point, *and* `-lib-entry` is not set:

- the global variables are set to their initial values at the beginning of the main entry point for all its properties to be established;
- special proof obligations are generated for the preconditions of the main entry point, hence to be proved with globals properly initialized.

Otherwise, initial values for globals are not taken into account and no proof obligation is generated for preconditions of the main entry point.

2.4.3 Model Selection

These options modify the underlying memory model that is used for computing weakest preconditions. See chapter 4 for details. Models are identified by a combination of *selectors* which are defined below:

Selector	Description
Hoare	Select Hoare memory model.
Typed	Select Typed memory model with limited casts.
`cast`	Select Typed memory model with unlimited casts (unsound).
`nocast`	Select Typed memory model with *no* casts.
`raw`	Disable the combination of memory models.
`var`	Combination of memory models based on variable analysis.
`ref`	Activate the detection of pointer variables used for reference passing style.
`caveat`	Caveat memory model (see 3.7).
`int`	Use machine integers when overflows and downcasts might occurs.
`nat`	Integer model without bounds (no overflow assumed).
`float`	Use floating-point operations.
`real`	Use mathematical reals instead of floating point.

Refer to Section 1.5 for details on arithmetic models and Chapter 4 for a description of memory models.

The available WP command-line options related to model selection are:

`-wp-model <spec...>` specifies the models to use. The specification is a list of *selectors*. Selectors are usually separated by '`,`' although other separators are accepted as well: '`+`', '`_`', spaces, newlines, tabs and parentheses '`(`', '`)`'.
Selectors are *case insensitive*. The option `-wp-model` can be used several times. All provided selectors are processed from left to right, possibly canceling previous ones.
Default setting corresponds to `-wp-model "Typed+var+int+float"`.

`-warn-(un)signed-(overflow|downcast)` those kernel options are used by the (default) arithmetic model `-wp-model +int` to interpret integer arithmetics. See section 1.5 for details.

`-wp-literals` exports the contents of string literals to provers (default: `no`).

`-wp-extern-arrays` gives an arbitrary large size to arrays with no dimensions. This is a modelization of infinite size arrays (default is: `no`).

`-wp-(alias|unalias|ref|context)-vars <var,...>` these options can be used to finely tweak the memory model inferred by WP. Each variable with a given name can be forced to be modeled as follows:

`alias`:	the variable is known to have aliases and modeled by `Typed`.
`noalias`:	the variable is known to have *no* alias and modeled with `Hoare`.
`ref`:	the variable is a constant pointer and is modeled by the `Ref`.
`context`:	the variable is initially non-aliased and uses a fresh global in `Typed`.

`-wp-print-separation` this option activate the calculus of separation hypotheses for the generated proof obligations. For each model supporting this feature, and each concerned function, an ACSL specification is printed on output. Currently, only the `Caveat`, `Typed` and `Ref` memory models support this feature.

2.4.4 Computation Strategy

These options modify the way proof obligations are generated during weakest precondition calculus.

`-wp-(no)-rte` generates RTE guards before computing weakest preconditions. This option calls the *rte generation* plug-in before generating proof obligations. The generated guards, when proved[1], fulfill the requirements for using the WP plug-in with the default machine-integer domain (default is: `no`). Using this option with `-wp-model nat` is tricky, because `rte` uses the kernel options to generate guards, and they might be not strong enough to meet the natural model requirements. In this case, a warning is emitted for potential runtime errors. Refer to Section 1.5 for details.

`-wp-(no)-split` conjunctions in generated proof obligations are recursively split into sub-goals. The generated goal names are suffixed by "`part<`*n*`>`" (defaults to `no`).

`-wp-(no)-callee-precond` includes preconditions of the callee after[2] a call (default is: `yes`).

`-wp-(no)-init-const` uses initializers for global `const` variables (default is: `no`).

`-wp-(no)-dynamic` handles calls *via* function pointers (experimental, default is: `no`).

[1] It is still correct to prove these RTE annotations with the WP plug-in.
[2] Proof obligations are always generated to check preconditions.

2.4.5 Trigger Generation

The ACSL language does not provide the user with a syntax for declaring *triggers* associated to lemmas and axioms. However, triggers are generally necessary for SMT solvers to discharge efficiently the generated proof obligations.

There is a limited support for triggers in WP. The *sub-terms* and *sub-predicates* marked with label `"TRIGGER"` in an axiom or lemma are collected to generate a multi-trigger for their associated free variables.

2.4.6 Qed Simplifier Engine

These options control the simplifications performed by the WP plug-in before sending proof obligations to external provers.

`-wp-(no)-simpl` simplifies constant expressions and tautologies (default is: `yes`).

`-wp-(no)-clean` removes unused terms and variables from proof obligations (default is: `yes`).

`-wp-(no)-let` propagates equalities by substitutions and let-bindings (default is: `yes`).

`-wp-(no)-pruning` eliminates trivial branches of conditionals (default is: `yes`).

`-wp-(no)-bits` simplifies bitwise operations (default is: `yes`).

`-wp-(no)-simplify-is-cint` eliminates redundant constraints on integers (default: `no`). Variables of C-integer types in `\forall` are promoted to `integer` variables when possible. Typically, provided `n` is already an integer, `\forall unsigned x; x < n ==> ...` will be simplified into `\forall integer x; 0 <= x < n ==> ...`.

2.4.7 Decision Procedures Interface

The generated proof obligations are submitted to external decision procedures. If proof obligations have just been generated, by using `-wp`, `-wp-fct`, `-wp-bhv` or `-wp-prop`, then only the new proof obligations are sent. Otherwise, all unproved proof obligations are sent to external decision procedures.

`-wp-prover <dp,...>` selects the decision procedures used to discharge proof obligations. See below for supported provers. By default, `alt-ergo` is selected, but you may specify another decision procedure or a list of to try with. Finally, you should supply `none` for this option to skip the proof step.

It is possible to ask for several decision procedures to be tried. For each goal, the first decision procedure that succeeds cancels the other attempts.

`-wp-proof <dp,...>` **deprecated** alias for `-wp-prover` for backward compatibility with WP version `0.6`.

`-wp-gen` only generates proof obligations, does not run provers. See option `-wp-out` to obtain the generated proof obligations.

`-wp-par <n>` limits the number of parallel process runs for decision procedures. Default is 4 processes. With `-wp-par 1`, the order of logged results is fixed. With more processes, the order is runtime dependent.

`-wp-filename-truncation <n>` truncates the basename of proof obligation files to the first `n` characters. Since numbers can be added as suffixes to ensure unique filenames, their length can be longer than `n`. No truncation is performed when the value equals zero.

`-wp-(no)-proof-trace` asks for provers to output extra information on proved goals when available (default is: `no`).

`-wp-(no)-unsat-model` asks for provers to output extra information when goals are not proved (default is: `no`).

`-wp-timeout <n>` sets the timeout (in seconds) for the calls to the decision prover (defaults to 10 seconds).

Alt-Ergo. Direct support for the Alt-Ergo prover is provided. You need at least version `0.99` of the prover, but more recent versions `1.01` or `1.30` are preferrable. It is also the default selected prover.

`-wp-prover alt-ergo` selects Alt-Ergo.

`-wp-prover altgr-ergo` opens the graphical interface of Alt-Ergo when the goal is not proved.

`-wp-steps` <*n*> sets the maximal number of Alt-Ergo steps. This can be used as a machine-independent alternative to timeout.

`-wp-depth` <*n*> sets '*stop*' and '*age-limit*' parameters of Alt-Ergo such that *n* cycles of quantifier instantiations are enabled.

`-wp-alt-ergo-opt <opt,...>` passes additional options to Alt-Ergo (default: none).

`-wp-alt-ergo='<cmd>'` override the `alt-ergo` command.

`-wp-altgr-ergo='<cmd>'` override the `altgr-ergo` command.

Coq. Direct support for the Coq proof assistant is provided. The generated proof obligations are accepted by Coq version `8.4`. When working with Coq, you will enter an interactive session, then save the proof scripts in order to replay them in batch mode.

`-wp-prover coq` runs `coqc` with the default tactic or with the available proof script (see below).

`-wp-prover coqide` first tries to replay some known proof script (if any). If it does not succeed, then a new interactive session for `coqide` is opened. As soon as `coqide` exits, the edited proof script is saved back (see below) and finally checked by `coqc`.

The only part of the edited file retained by WP is the proof script between "`Proof.`...`Qed.`".

`-wp-script <f.script>` specifies the file which proof scripts are retrieved from, or saved to. The format of this file is private to the WP plug-in. It is, however, a regular text file from which you can cut and paste part of previously written script proofs. The WP plug-in manages the content of this file for you.

`-wp-(no)-update-script` if turned off, the user's script file will *not* be modified. A warning is emitted if the script data base changed.

`-wp-tactic <ltac>` specifies the Coq tactic to try with when no user script is found. The default tactic is `"auto with zarith"`. See also how to load external libraries and user-defined tactics in section 2.4.9.

`-wp-tryhints` When both the user-provided script and the default tactic fail to solve the goal, other scripts for similar goals can be tried instead.

`-wp-hints <n>` sets the maximal number of suggested proof scripts.

`-wp-coq-timeout <n>` sets the maximal time in seconds for running the `coqc` checker. Does not apply to `coqide` (default: 30s).

`-wp-coq-opt <opt,...>` additional options for `coqc` and `coqide` (default: none).

`-wp-coqc='<cmd>'` override the `coqc` command.

`-wp-coqide='<cmd>'` override the `coqide` command. If the command line contains the `emacs` word (case-insensitive), coq-options are not passed to the command, but a coq-project is used instead. This conforms to Proof General 4.3 settings. The project file can be changed (see below).

`-wp-coq-project='<name>'` override the `_CoqProject` file name for Emacs and Proof General.

Why3. Since WP version `0.7`, native support for Why3 and Why3-Ide are provided. The older system Why `2.x` is *no* longer supported.

`-wp-prover "why3ide"` runs Why3-Ide with all generated goals. On exit, the WP plug-in reads back your Why3 session and updates the proof obligation status accordingly.

`-wp-prover "<p>"` runs a Why3 prover named `<p>`.

`-wp-prover "why3:<p>"` useful alias when `"<p>"` can be ambiguous. It is actually different to run `alt-ergo` or `coq` directly from WP or through Why3.

`-wp-detect` lists the provers available with Why3. This command calls `why3 -list-provers` but you have to configure Why3 on your own before, for instance by using `why3config`. Consult the Why3 user manual for details. The listed prover names can be directly used with the `-wp-prover` option.

`-wp-why3='<cmd>'` override the `why3` command.

Sessions. Your Why3 session is saved in the "project.session" sub-directory of -wp-out. You may run why3ide by hand by issuing the following command:

```
# why3ide -I <frama-c-share>/wp <out>/project.session
```

Proof recovering features of Why3 are fully available, and you can interleave proving from WP with manual runs of why3ide. Interactive proofs with Why3 are completely separated from those managed by the native WP interface with Coq.

2.4.8 Generated Proof Obligations

Your proof obligations are generated and saved to several text files. With the -wp-out option, you can specify a directory of your own where all these files are generated. By default, this output directory is determined as follows: in the GUI, it is <home>/.frama-c-wp where <home> is the user's home directory returned by the HOME environment variable. In command-line, a temporary directory is automatically created and removed when Frama-C exits.

Other options controlling the output of generated proof obligations are:

-wp-(no)-print pretty-prints the generated proof obligations on the standard output. Results obtained by provers are reported as well (default is: no).

-wp-out <dir> sets the user directory where proof obligations are saved. The directory is created if it does not exist yet. Its content is not cleaned up automatically.

2.4.9 Additional Proof Libraries

It is possible to add additional bases of knowledge to decision procedures. This support is provided for Alt-Ergo, Why3 and Coq thanks to the following options:

-wp-share <dir> modifies the default directory where resources are found. This option can be useful for running a modified or patched distribution of WP.

-wp-include <dir,...,++sharedir> (**deprecated** use driver instead) sets the directories where external libraries and driver files are looked for. The current directory (implicitly added to that list) is always looked up first. Relative directory names are relative to the current directory except for names prefixed by the characters ++. In such a name, the directory is relative to the main FRAMAC_SHARE directory.

-wp-alt-ergo-lib <f,...> (**deprecated** use altergo.file in driver instead) looks for Alt-Ergo library files "f.mlw" and inlines them into the proof obligation files for Alt-Ergo.

-wp-coq-lib <f,...> (**deprecated** use coq.file in driver instead) looks for Coq files "f.v" or "f.vo". If "f.vo" is not found, then WP copies "f.v" into its working directory (see option -wp-out) and compiles it locally.

-wp-why-lib <f,...> (**deprecated** use why3.file in driver instead) looks for Why3 library file "f.why" and opens the library "f.F" for the proving the goals.

2.4.10 Linking ACSL Symbols to External Libraries

Besides additional proof libraries, it is also possible to *link* declared ACSL symbols to external or predefined symbols. In such case, the corresponding ACSL definitions, if any, are not exported by WPs.

External linkage is specified in *driver files*. It is possible to load one or several drivers with the following WP plug-in option:

`-wp-driver <file,...>` load specified driver files, replacing deprecated features from section 2.4.9.

Each driver file contains a list of bindings with the following syntax:

```
library "lib": "lib" ..."lib"
    group.field   := string
    group.field   += string
           type   symbol                           = "link" ;
           ctor   type symbol ( type,...,type )    = "link" ;
          logic   type symbol ( type,...,type )    = property-tags "link" ;
      predicate   symbol ( type,...,type )         = "link" ;
```

It is also possible to define *aliases* to other ACSL symbols, rather than external links. In this case, you may replace `=` by `:=` and use an ACSL identifier in place of the external "*link*". No property tag is allowed when defining aliases.

Library specification is optional and applies to subsequent linked symbols. If provided, the WP plug-in automatically loads the specified external libraries when linked symbols are used in a goal. Dependencies among libraries can also be specified, after the ':'.

Generic *group.field* options have a specific value for each theory. The binding applies to the current theory. Binding with the `:=` operator resets the option to the singleton of the given string and binding with the `+=` operator adds the given string to the current value of the option. The following options are defined by the plugin: `coq.file`, `altergo.file`, `why3.file` and `why3.import`.

C-Comments are allowed in the file. For overloaded ACSL symbols, it is necessary to provide one "*link*" symbol for each existing signature. The same "*link*" symbol is used for all provers, and must be defined in the specified libraries, or in the external ones (see 2.4.9).

It is also possible to specify different names for each prover, with the following syntax: `{coq="`*a*`";altergo="`*b*`";why3="`*c*`"}`. Alternatively, a link-name can be an arbitrary string with patterns substituted by arguments, `"(%1+%2)"` for instance.

When a library "*lib*" is specified, the loaded module depends on the target solver:

	Option	Format
Coq:	`coq.file`	`[dir:]path.v`
Alt-Ergo:	`altergo.file`	`path.mlw`
Why3:	`why3.file`	`path.why[:name][:as]`
	`why3.import`	`theory[:as]`

Precise meaning of formats is given by the following examples (all filenames are relatives to the driver file's directory):

`coq.file="mydir/bar.v"` Imports module `Bar` from file `mydir/bar.vo`.

`coq.file="mydir/foo:Foo.v"` Loads coq library `foo.Foo` from file `mydir/foo/Foo.vo`.

`why3.file="mydir/foo.why"` Imports theory `foo.Foo` from directory `mydir`.

`why3.file="mydir/foo.why:Bar"` Imports theory `foo.Bar` from directory `mydir`.

`why3.file="mydir/foo.why:Bar:T"` Imports theory `foo.Bar as T` from directory `mydir`.

`why3.import="foo.Bar"` Imports theory `foo.Bar` with no additional includes.

`why3.import="foo.Bar:T"` Imports theory `foo.Bar as T` with no additional includes.

See also the default WP driver file, in `[wp-share]/wp.driver`.

Optional *property-tags* can be given to `logic` "*link*" symbols to allow the WP to perform additional simplifications (See section 5). Tags consist of an identifier with column (‘`:`’), sometimes followed by a link (‘"*link*";’). The available tags are depicted on figure 2.4.

Tags	Operator Properties
`commutative:`	specify a commutative symbol: $x \odot y = y \odot x$
`associative:`	specify an associative symbol: $(x \odot y) \odot z = x \odot (y \odot z)$
`ac:`	shortcut for `associative: commutative:`
`left:`	balance the operator on left during export to solvers (requires the associative tag): $x \odot y \odot z = (x \odot y) \odot z$
`right:`	balance the operator on right during export to solvers (requires the associative tag): $x \odot y \odot z = x \odot (y \odot z)$
`absorbant:` "*a-link*":	specify "*a-link*" as being the absorbant element of the symbol: "*a-link*" $\odot x =$ "*a-link*" $x \odot$ "*a-link*" $=$ "*a-link*"
`neutral:` "*e-link*":	specify "*e-link*" as being the neutral element of the symbol: "*e-link*" $\odot x = x$ $x \odot$ "*e-link*" $= x$
`inversible:`	specify simplification relying on the existence of an inverse: $x \odot y = x \odot z \iff y = z$ $y \odot x = z \odot x \iff y = z$
`idempotent:`	specify an idempotent symbol: $x \odot x = x$
`injective:`	specify an injective function: $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \Longrightarrow \forall i\ x_i = y_i$
`constructor:`	specify an injective function, that constructs different values from any other constructor. Formally, whenever f and g are two distinct constructors, they are both injective and: $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ forall x_i and y_j.

Figure 2.4: Driver Property Tags

2.5 Plug-in Developer Interface

The WP plug-in has several entry points registered in the Dynamic[3] module of Frama-C.

Those entry points are however deprecated. Instead, a full featured OCaml API is now exported with the Wp plug-in. As documented in the Frama-C user manual, simply add the directive PLUGIN_DEPENDENCIES+=Wp to the Makefile of your plug-in.

The high-level API for generating and proving properties is exported in module Wp.API. The logical interface, compilers, and memory models are also accessible. See the generated HTML documentation of the platform for details.

2.6 Proof Obligation Reports

The WP plug-in can export statistics on generated proof obligations. These statistics are called *WP reports* and are distinct from those *property reports* generated by the Report plug-in. Actually, *WP reports* are statistics on proof obligations generated by WP, whereas *property reports* are consolidated status of properties, generated by the Frama-C kernel from various analyzers. We only discuss *WP reports* in this section.

Reports are generated with the following command-line options:

-wp-report <Rspec$_1$,...,Rspec$_n$> specifies the list of reports to export. Each value Rspec$_i$ is a *WP report* specification file (described below).

-wp-report-basename <name> set the basename for exported reports (described below).

Reports are created from user defined wp-report specification files. The general format of a wp-report file is as follows:

```
<configuration section...>
@HEAD
<head contents...>
@CHAPTER
<per chapter contents...>
@SECTION
<per section contents of a chapter...>
@TAIL
<tail contents...>
@END
```

The configuration section consists of optional commands, one per line, among:

@CONSOLE the report is printed on the standard output.
Also prints all numbers right-aligned on 4 ASCII characters.

@FILE "<*file*>" the report is generated in file *file*.

@SUFFIX "<*ext*>" the report is generated in file *base.ext*,
where *base* can be set with -wp-report-basename option.

@ZERO "<*text*>" text to be printed for 0-numbers. Default is "-".

[3] See the *plug-in development guide*

`@GLOBAL_SECTION "<`*text*`>"` text to be printed for the chapter name about globals

`@AXIOMATIC_SECTION "<`*text*`>"` text to be printed for the chapter name about axiomatics

`@FUNCTION_SECTION "<`*text*`>"` text to be printed for the chapter name about functions

`@AXIOMATIC_PREFIX "<`*text*`>"` text to be printed before axiomatic names. Default is `"Axiomatic"` (with a trailing space).

`@FUNCTION_PREFIX "<`*text*`>"` text to be printed before function names. Default is empty.

`@GLOBAL_PREFIX "<`*text*`>"` text to be printed before global property names. Default is `"(Global)"` (with a trailing space).

`@LEMMA_PREFIX "<`*text*`>"` text to be printed before lemma names. Default is `"Lemma"` (with a trailing space).

`@PROPERTY_PREFIX "<`*text*`>"` text to be printed before other property names.

The generated report consists of several optional parts, corresponding to Head, Chapter and Tail sections of the wp-report specification file. First, the head contents lines are produced. Then the chapters and their sections are produced. Finally, the Tail content lines are printed.

The different chapters are about globals, axiomatics and functions. Outputs for these chapters can be specified using these directives:

`@CHAPTER` <*chapter header...*>

`@GLOBAL` <*global section contents...*>

`@AXIOMATIC` <*per axiomatic section contents...*>
For each axiomatic, a specific section is produced under the chapter about axiomatics.

`@FUNCTION` <*per function section contents...*>
For each function analyzed by WP, a specific section is produced under the chapter about functions.

`@SECTION` <*default section contents...*>

`@PROPERTY` <*per property contents...*>
For each property of a section, a specific textual content can be specified.

Textual contents use special formatters that will be replaced by actual statistic values when the report is generated. There are several categories of formatters (PO stands for *Proof Obligations*):

Formatters	**Description**
`&<`*col*`>:`	insert spaces up to column *col*
`&&`	prints a `"&"`
`%%`	prints a `"%"`
`%<`*stat*`>`	statistics for section
`%prop`	percentage of proved properties in section
`%prop:total`	number of covered properties
`%prop:valid`	number of proved properties
`%prop:failed`	number of remaining unproved properties
`%<`*prover*`>`	PO discharged by *prover*
`%<`*prover*`>:<`*stat*`>`	statistics for *prover* in section

Provers	
(<*prover*>)	A prover name (see `-wp-prover`)
Statistics	
(<*prover*>)	
`total`	number of generated PO
`valid`	number of discharged PO
`failed`	number of non-discharged PO
`time`	maximal time used by prover for one PO
`steps`	maximal steps used by prover for one PO
`success`	percentage of discharged PO

Remarks: `&ergo` is a shortcut for `&alt-ergo`. Formatters can be written `"%.."` or `"%{..}"`.

Textual contents can use naming formatters that will be replaced by current names:

Names	**Description**
`%chapter`	current chapter name
`%section`	current section name
`%global`	current global name (under the chapter about globals)
`%axiomatic`	current axiomatic name (under the chapter about axiomatics)
`%function`	current function name (under the chapter about functions)
`%name`	current name defined by the context: - property name inside `@PROPERTY` contents, - function name inside `@FUNCTION` contents, - axiomatic name inside `@AXIOMATIC` contents, - global name inside `@GLOBAL` contents, - section name inside `@SECTION` contents, - chapter name inside `@CHAPTER` contents.

2.7 Plug-in Persistent Data

As a general observation, almost *none* of the internal WP data is kept in memory after each execution. Most of the generated proof-obligation data is stored on disk before being sent to provers, and they are stored in a temporary directory that is removed upon Frama-C exit (see also `-wp-out` option).

The only information which is added to the Frama-C kernel consists in a new status for those properties proved by WP plug-in with their dependencies.

Thus, when combining WP options with `-then`, `-save` and `-load` options, the user should be aware of the following precisions:

`-wp, -wp-prop, -wp-fct, -wp-bhv.` These options make the WP plug-in generate proof-obligations for the selected properties. The values of theses options are never saved and they are cleared by `-then`. Hence, running `-wp-prop A -then -wp-fct F` does what you expect: properties tagged by `A` are proved only once.

`-wp-print, -wp-prover, -wp-gen, -wp-detect.` These options do not generate new proof-obligations, but run other actions on all previously generated ones. For the same reasons, they are not saved and cleared by `-then`.

`-wp-xxx.` All other options are tunings that can be easily turned on and off or set to the desired value. They are saved and kept across `-then` commands.

Chapter 3

WP Models

Basically, a memory model is a set of datatypes, operations and properties that are used to abstract the values living inside the heap during the program execution.

Each memory model defines its own representation of pointers, memory and data actually stored in the memory. The memory models also define some types, functions and properties required to translate C programs and ACSL annotations into first order logic formulæ.

The interest of developing several memory models is to manage the trade-off between the precision of the heap's values representation and the difficulty of discharging the generated proof obligations by external decision procedures. If you choose a very accurate and detailed memory model, you shall be able to generate proof obligations for any program and annotations, but most of them would not be discharged by state-of-the art external provers. On the other hand, for most C programs, simplified models are applicable and will generate less complex proof obligations that are easier to discharge.

A practical methodology is to use the simpler models whenever it is possible, and to up the ante with more involved models on the remaining, more complex parts of the code.

This chapter is dedicated to the description of the memory models implemented by the WP plug-in. In this manual, we only provide a high-level description of the memory models you might select with option `-wp-model` (section 3.2 and 3.3). Then we focus on two general powerful optimizations. The first one, activated by default (section 3.4), mixes the selected memory model with the purely logical Hoare model for those parts of your program that never manipulate pointers. The second one (section 3.5) is dedicated to those pointers that are formal parameters of function passed by reference.

3.1 Language of Proof Obligations

The work of WP consists in translating C and ACSL constructs into first order logical formulæ. We denote by $\mathcal{L}$ the logic language for constructing proof obligations. Shortly, this logical language is made of terms (t : term) and propositions (P : prop) that consist of:

- Natural, signed, unbounded integer constants and their operations;
- Natural real numbers and their operations;
- Arrays (as total maps) and records (tuples with named fields);
- Abstract (polymorphic) data types;

- Anonymous function symbols with (optional) algebraic properties;
- Logical connectors;
- Universally and existentially quantified variables.

Actually, the task of the memory model consists in mapping any heap C-values at a given program point to some variable or term in the logical $\mathcal{L}$ language.

3.2 The Hoare Memory Model

This is the simplest model, inspired by the historical definition of *Weakest Precondition Calculus* for programs with no pointers. In such programs, each global and local variable is assigned a distinct variable in $\mathcal{L}$.

Consider for instance the statement `x++;` where `x` has been declared as an `int`. In the `Hoare` memory model, this C-variable will be assigned to two $\mathcal{L}$-variables, say x_1 before the statement, and x_2 after the statement, with the obvious relation $x_2 = x_1 + 1$ (if no overflow occurred).

Of course, this model is not capable of handling memory reads or writes through pointer values, because there is no way of representing aliasing.

You select this memory model in the WP plug-in with the option `-wp-model Hoare`; the analyzer will complain whenever you attempt to access memory through pointers with this model.

3.3 Memory Models with Pointers

Realistic memory models must deal with reads and writes to memory through pointers. However, there are many ways for modeling the raw bit stream the heap consists of. All memory models $\mathcal{M}$ actually implement a common signature:

Pointer Type: τ, generally a pair of a base address and an offset.

Heap Variables: for each program point, there is a set of logical variables to model the heap. For instance, you may have a variable for the values at a given address, and another one for the allocation table. The heap variables $m_1 \ldots m_k$ are denoted by $\overline{m}$.

Read Operation: given the heap variables $\overline{m}$, a pointer value $p : \tau$, and some C-type T, the model will define an operation:

$$\mathrm{read}_T(\overline{m}, p) : \mathrm{term}$$

that defines the representation in $\mathcal{L}$ of the value of C-type T which is stored at address p in the heap.

Write Operation: given the heap variables $\overline{m}$ before a statement, and their associated heap variables $\overline{m}'$ after the statement, a pointer value $p : \tau$ and a value v of C-type T, the model will define a relation:

$$\mathrm{write}_T(\overline{m}, p, v, \overline{m}') : \mathrm{prop}$$

that relates the heap before and after writing value v at address p in the heap.

Typically, consider the statement `(*p)++` where `p` is a C-variable of type `(int*)`. The memory model $\mathcal{M}$ will assign a unique pointer value $P : \tau$ to the address of `p` in memory.

Then, it retrieves the actual value of the pointer p, say A_p, by reading a value of type `int*` into the memory variables $\overline{m}$ at address P:

$$A_p = \mathrm{read}_{\texttt{int*}}(\overline{m}, P)$$

Next, the model retrieves the previous `int`-value at actual address A_p, say V_p:

$$V_p = \mathrm{read}_{\texttt{int}}(\overline{m}, A_p)$$

Finally, the model relates the final memory state $\overline{m}'$ with the incremented value $V_p + 1$ at address P:

$$\mathrm{write}_{\texttt{int}}(\overline{m}, A_p, V_p + 1, \overline{m}')$$

3.4 Hoare Variables mixed with Pointers

As illustrated above, a very simple statement is generally translated by memory models into complex formulæ. However, it is possible in some situations to mix the Hoare memory model with the other ones.

For instance, assume the address of variable `x` is never taken in the program. Hence, it is not possible to create a pointer aliased with `&x`. It is thus legal to manage the value of `x` with the Hoare memory model, and other values with another memory-model $\mathcal{M}$ that deals with pointers.

Common occurrences of such a situation are pointer variables. For instance, assume `p` is a variable of type `int*`; it is often the case that the value of `p` is used (as in `*p`), but not the address of the variable `p` itself, namely `&p`. Then, it is very efficient to manage the value of `p` with the Hoare memory model, and the value of `*p` with a memory model with pointers.

Such an optimization is possible whenever the address of a variable is never taken in the program. It is activated by default in the WP plug-in, since it is very effective in practice. You can nevertheless deactivate it with selector "`-wp-model raw`".

3.5 Hoare Variables for Reference Parameters

A common programming pattern in C programs is to use pointers for function arguments passed by reference. For instance, consider the `swap` function below:

```
void swap(int *a,int *b)
{
  int tmp = *a ;
  *a = *b ;
  *b = tmp ;
}
```

Since neither the address of `a` nor the one of `b` are taken, their values can be managed by the Hoare Model as described in the previous section. But we can do even better. Remark that none of the pointer values contained in variables `a` and `b` is stored in memory. The only occurrences of these pointer values are in expressions `*a` and `*b`. Thus, there can be no alias with these pointer values elsewhere in memory, provided they are not aliased initially.

Hence, not only can `a` and `b` be managed by the Hoare model, but we can also treat `(*a)` and `(*b)` expressions as two independent variables of type `int` with the Hoare memory model.

For the callers of the `swap` function, we can also take benefit from such by-reference passing arguments. Typically, consider the following caller program:

```
void f(void)
{
  int x=1,y=2 ;
  swap(&x,&y);
}
```

Strictly speaking, this program takes the addresses of `x` and `y`. Thus, it would be natural to handle those variables by a model with pointers. However, `swap` will actually always use `*&x` and `*&y`, which are respectively `x` and `y`.

In such a situation it is then correct to handle those variables with the Hoare model, and this is a very effective optimization in practice. Notice however, that in the example above, the optimization is only correct because `x` and `y` have disjoint addresses.

These optimizations can be activated in the WP plug-in with selector "`-wp-model ref`", and the necessary separation conditions are generated on-the-fly. This memory model first detects pointer or array variables that are always passed by reference. The detected variables are then assigned to the Hoare memory model.

This optimization is not activated by default, since the non-aliasing assumptions at call sites are sometimes irrelevant.

3.6 The Typed Memory Model

This memory model is actually a reformulation of the `Store` memory model used in previous versions of the WP plug-in. In theory, its power of expression is equivalent. However, in practice, the reformulation we performed makes better usage of built-in theories of Alt-Ergo theorem prover and Coq features. The main modifications concern the heap encoding and the representation of addresses.

Addresses. We now use native records of $\mathcal{L}$ and provers to encode addresses as pairs of base and offset (integers). This greatly simplifies reasoning about pointer separation and commutation of memory accesses and updates.

Store Memory. In the `Store` memory model, the heap is represented by one single memory variable holding an array of *data* indexed by *addresses*. Then, integers, floats and pointers must be boxed into *data* and unboxed from *data* to implement read and write operations. These boxing-unboxing operations typically prevent Alt-Ergo from making maximal usage of its native array theory.

Typed Memory. In the Typed memory model, the heap is now represented by *three* memory variables, holding respectively arrays of integers, floats and addresses indexed by addresses. This way, all boxing and unboxing operations are avoided. Moreover, the native array theory of Alt-Ergo works very well with its record native theory used for addresses: memory variables access-update commutation can now rely on record theory to decide that two addresses are different (separated).

3.7 The Caveat Memory Model

This memory model simulates the behavior of the Caveat analyser, with additional enhancements. It is implemented as an extension of the `Typed` memory model, with a specific treatment of *formal* variables of pointer type.

To activate this model, simply use '`-wp-model Typed+Caveat`' or '`-wp-model Caveat`' for short.

A specific detection of variables to be treated as *reference parameters* is used. This detection is more clever than the standard one since it only takes into account local usage of each function (not global ones).

Additionally, the `Caveat` memory model of Frama-C performs a local allocation of formal parameters with pointer types that cannot be treated as *reference parameters*.

This makes explicit the separation hypothesis of memory regions pointed by formals in the Caveat tool. The major benefit of the WP version is that aliases are taken into account by the `Typed` memory model, hence, there are no more suspicious *aliasing* warnings.

Warning: using the `Caveat` memory model, the user *must* check by manual code review that no aliases are introduced *via* pointers passed to formal parameters at call sites.

However, WP can output the implicit separation hypotheses required by the memory model *via* the `-wp-print-separation` option.

Chapter 4

WP Arithmetic Models

Arithmetic models are responsible for modeling the integral and floating point operations of C and ACSL into mathematical integers and reals. The exact representation of machine operation into logic is straightforward by using ACSL interpretation. However, this leads to very difficult proof obligations for usual solvers which are not that efficient with modular arithmetic.

Hence, the WP proposes an aggressive simplification by using separation of concerns: most machine operations are identical to their pure mathematical counterpart unless an overflow occurs, and most overflows are subject to unexpected behavior of the source code – excluding explicit conversions.

To cope with various usage and needs, the WP plug-in proposes three integral models and two floating point models:

`Cint`	an exact model for machine integers
`Natural`	a model assuming the absence of overflow
`NoRange`	a model forgetting all integral bounds
`Float`	an *opaque* model for floating points
`Real`	a model where all rounding are discarded

The selection of the model is performed *via* the `-wp-model` option (Cf. 2.4).

4.1 Integral Models

We distinguish various kinds of machine-integral operations: the domains of a C-integer types, the conversions of integer into a C-integer, the usual arithmetic operations (addition, *etc.*), and the logical bitwise operations. Each model correspond to a specific choice for each kind of operations.

The real machine operations depend on the *signedness* and *size* of the C-integer types of its operands and result. We known detail how each model interpret machine operations.

Domain. Actually, any *signed* $2n$-bits integer lives in $[-2^n \ldots 2^n - 1]$; and an *unsigned* integer with the same size lives in range $[0 \ldots 2^{2n} - 1]$. This is exactly rendered by the `Cint` and `Natural` model. For the `NoRange` model, we simply erase the bounds but keep their signedness, such that *unsigned* integers are considered living in $[-\infty \ldots + \infty]$, and *signed* ones in $[0 \ldots + \infty]$.

Conversions. Conversions explicitly introduced by casts are rendered exactly by a modulus extraction in the `Cint` and `Natural` models. For the `NoRange` model, since bounds are erased, no modulus is applied and the conversion is replaced by the identity function.

Arithmetics. Usual operations (addition, multiplication, division, modulus) are correctly rendered with a modulus extraction in the `Cint` model, such that $a \oplus b \equiv (\texttt{int})(a+b)$ for the additions of two `int` expressions a and b. In the `Natural` and `NoRange` models, the modulus is simply erased: in the natural model, we assume the absence of overflow, and without range, the conversion is considered an identity.

Bitwise. Logical bitwise operations (logical-or, logical-not, shifts, *etc.*) are correctly rendered with the `Cint` and `Natural` model by a modulus extraction of their ACSL counterpart, which are precisely defined over natural integers. Notice that in practice, only few *unsigned* operations actually require a modulus. With the `NoRange` domain, modulus is simply never applied.

The various interpretation of each kind of machine operation by the various models are summarized in the table below, with deviations from the actual behavior in italic:

	`Cint`	`Natural`	`NoRange`
Domain	bounds	bounds	*signed*
Conversion	modulus	modulus	*identity*
Bitwise	modulus	modulus	*identity*
Arithmetic	modulus	*integral*	*integral*

Chapter 5

WP Simplifier

The logical language $\mathcal{L}$ used to build proof obligations is now equipped with built-in simplifications. This allows for proof obligations to be simplified *before* being sent to external provers, and sometimes to be reduced to trivial goals.

This chapter is dedicated to the description of the simplifier and how to use it with the WP plug-in. It also presents how combinatorial explosion of path exploration is now tackled down thanks to *passive form* transformation and automated sub-terms *factorization* [FS01, Lei03]. This also leads to more compact and (somehow) more readable proof obligations, with less memory, less disk usage and lower external prover time overhead, compared to WP versions `0.6` and lower.

5.1 Logic Normalizations

The new logic language $\mathcal{L}$ is naturally equipped with term normalization and maximal sub-term sharing. It is only used with new memory models, not with the standard ones.

Maximal sub-term sharing is responsible for the introduction of let-bindings whenever a sub-expression appears several times in the generated proof obligations. The occupied memory and disk usage of WP is also reduced compared to other models.

The normalization rules can not be turned off, and are responsible for local simplifications. Although modest, they allow a proof obligation to be trivially discharged.

Logic normalization by commutativity and associativity; absorption and neutral elements; elimination of redundant facts; propagation of negations (Morgan laws); simplification of conditionals.

Arithmetic normalization by commutativity and associativity; absorption and neutral elements; factorization with linear forms; constant folding; normalization of linear equalities and inequalities.

Array elimination of consecutive accesses and updates.

Record elimination of consecutive accesses and updates;
simplification of structural equalities and inequalities.

5.2 Simplifier Engine (Qed)

Built on top of our normalizing logic language $\mathcal{L}$, we have a simplifier engine named Qed[Cor14]. The simplifier engine is used by the WP plug-in to simplify the generated proof contexts and proof obligations. The basic feature of Qed is to manage a knowledge base Γ. It is possible to add new facts (hypotheses) to Γ, and to simplify (rewrite) a term of a property with respect to Γ.

By default, the only rewriting performed by Qed is the propagation of equality classes by normalization. The Qed engine can be enriched by means of plug-ins to perform more dedicated simplifications. For instance, we have developed a simplifier plug-in for array and record theories, and a prototype for linear inequalities.

WP uses the simplification engine to simplify proof contexts by recursively combining basic laws involving the simplifier engine. Each law is applied with respect to a local knowledge base Γ (initially empty).

Adding a new fact H to Γ is denoted by $\Gamma \oplus H$; rewriting a term of predicate e into e' with respect to Γ is denoted by $\Gamma \models e \triangleright e'$.

Inference Law. A hypothesis is simplified and added to the knowledge base to simplify the goal.

$$\frac{\Gamma \models\ H \triangleright H' \qquad \Gamma \oplus H' \models\ G \triangleright G'}{\Gamma \models\ (H \rightarrow G) \triangleright (H' \rightarrow G')}$$

Conjunction Law. Each side of a conjunction is simplified with the added knowledge of the other side. This law scales up to the conjunction of n facts, and simplifications can be performed incrementally.

$$\frac{\Gamma \oplus B \models\ A \triangleright A' \qquad \Gamma \oplus A \models\ B \triangleright B'}{\Gamma \models\ (A \wedge B) \triangleright (A' \wedge B')}$$

Conditional Law. The conditional expression is simplified, before simplifying each branch under the appropriate hypothesis.

$$\frac{\Gamma \models\ H \triangleright H' \qquad \Gamma \oplus H' \models\ A \triangleright A' \qquad \Gamma \oplus \neg H' \models\ B \triangleright B'}{\Gamma \models\ (H\,?A : B) \triangleright (H'\,?A' : B')}$$

Inside the WP plug-in, the proof contexts are only built in terms of conjunctions, conditional and inference rules. Hence, these laws are sufficient to perform proof context simplifications. Technically, simplification has a quadratic complexity in the width and depth of the proof formula. Options will be added to control the risk of combinatorial explosion. In practice, simplification is delayed until submission of the proof obligation to external provers, that have similar complexity. Since we account on simplification for enhancing prover efficiency, we expect it to be worth the cost.

The power of the simplification process depends on the simplification plug-ins loaded in the Qed engine, and will be the purpose of further developments.

5.3 Efficient WP Computation

During the *Weakest Precondition* calculus, proof obligations are constructed backwardly for each program instruction. Conditional statements are of particular interest, since they introduce a fork in the generated proof contexts.

More precisely, consider a conditional statement `if` (e) A `else` B. Let W_A be the weakest precondition calculus from block A, and W_B the one from block B. Provided the translation of expression e in the current memory model leads to assumption E, the naive weakest precondition of the conditional is: $(E\,?W_A : W_B)$.

With this formula, the *weakest preconditions* of the program after the conditional are duplicated inside W_A and W_B. Moreover, these common postconditions have been transformed by the effects of A and B. Then, the factorization of common sub-terms of logic language $\mathcal{L}$ is *not* capable of avoiding the duplication. In the presence of successive conditionals, proof obligations generated become twice as big at each conditional statement.

To tackle this problem, the solution is to put the program in *passive form* [FS01, Lei03]. Each variable of the program is assigned a different logic variable in each branch. The different variables are joined at conditionals into new fresh variables and equality conditions.

In practice, the passive form transformation is done during the *weakest precondition* calculus, together with the translation of C and ACSL by the memory model. Hence, a translation map σ is maintained at each program point from memory model variables to $\mathcal{L}$ logic variables.

Joining maps σ_1 and σ_2 from the branches of a conditional leads to a new map σ assigning a new logic variable x to memory variable m whenever $\sigma_1(m)$ and $\sigma_2(m)$ are different. This join also produces the two sets of equalities H_1 and H_2 associated to this variable renaming. Hence $\sigma(m) = \sigma_1(m)$ below is member of H_1 and $\sigma(m) = \sigma_2(m)$ is member of H_2.

Now, if W is the postcondition of the conditional program below, W_A and W_B can always be decomposed into: $W_A = W_A^0 \wedge W$ and $W_B = W_B^0 \wedge W$. Finally, the weakest precondition of the conditional is:

$$(\,E\,?\;H_1 \wedge W_A^0\;:\;H_2 \wedge W_B^0\,) \wedge W$$

This form actually factorizes the common postcondition to A and B, which makes the *weakest precondition* calculus linear in the number of program statements.

Bibliography

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. *First International Workshop on Intermediate Verification Languages*, August 2011.

[Bur72] R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 1972.

[CCK06] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006.

[Coq10] Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2010.

[Cor14] Loïc Correnson. Qed. computing what remains to be proved. In *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, pages 215–229, 2014.

[Dij68] Edsger W. Dijkstra. A constructive approach to program correctness. *BIT Numerical Mathematics*, Springer, 1968.

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, 1967.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[Lei03] K. Rustan Leino. Efficient weakest preconditions, 2003.

[Lei08] K. Rustan M. Leino. *This is Boogie 2.* Microsoft Research, 2008.

[MM09] Yannick Moy and Claude Marché. *Jessie Plugin Tutorial,* Beryllium *version.* INRIA, 2009. http://www.frama-c.cea.fr/jessie.html.