

WP Plug-in Manual

For Frama-C 34.0 beta (Selenium)

Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, Allan Blanchard





Work licensed under Creative Commons BY licence
https://creativecommons.org/licenses/by/4.0/

© 2010-2026 CEA-List

Contents

Foreword 4

1 **Introduction** 5

1.1 Installation . . . 6

1.2 Tutorial . . . 6

1.3 Weakest Preconditions . . . 9

1.4 Memory Models . . . 10

1.5 Arithmetic Models . . . 11

1.6 Limitations & Roadmap . . . 12

2 **Using the WP Plug-in** 14

2.1 Supported annotations . . . 15

2.2 Graphical User Interface . . . 16

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 . . . 20

2.3.5 Replaying Scripts . . . 20

2.3.6 Tracking Scripts . . . 22

2.3.7 Strategies (DEPRECATED) . . . 22

2.3.8 General Tactics . . . 23

2.3.9 Integers & Bit-wised Tactics . . . 25

2.3.10 Domain Specific Tactics . . . 27

2.3.11 Custom Tactics and Strategies . . . 28

2.4 Command Line Options . . . 32
2.4.1 Goal Selection . . . 32
2.4.2 Program Entry Point . . . 33
2.4.3 Model Selection . . . 33
2.4.4 Computation Strategy . . . 34
2.4.5 Smoke Tests . . . 37
2.4.6 Trigger Generation . . . 38
2.4.7 Qed Simplifier Engine . . . 38
2.4.8 Prover Selection . . . 39
2.4.9 Generated Proof Obligations . . . 41
2.4.10 Probes and Counter Examples . . . 41
2.4.11 Linking ACSL Symbols to External Libraries . . . 43
2.4.12 Importing Why3 Modules . . . 45
2.4.13 Proof Session & Cache . . . 46
2.4.14 Proof Obligation Reports . . . 47
2.4.15 Persistent Options . . . 47
2.5 Proof Strategies . . . 47
2.6 Structure of JSON Scripts . . . 50
2.7 Structure of JSON Reports . . . 52
2.8 Structure of Report Templates . . . 53

3 **WP Models** 56
3.1 Language of Proof Obligations . . . 56
3.2 The Hoare Memory Model . . . 57
3.3 Memory Models with Pointers . . . 57
3.4 Hoare Variables mixed with Pointers . . . 58
3.5 Hoare Variables for Reference Parameters . . . 58
3.6 Mixed models hypotheses . . . 59
3.7 The Typed Memory Model . . . 59
3.8 The Bytes Memory Model . . . 60
3.9 The Region Memory Model . . . 60
3.10 The Caveat Memory Model . . . 61

4 **WP Simplifier** 63
4.1 Logic Normalizations . . . 63
4.2 Simplifier Engine (Qed) . . . 63
4.3 Efficient WP Computation . . . 64

5 **Builtins** 66
5.1 General Simplifications . . . 66
5.2 ACSL Built-ins . . . 66
5.3 Custom Extensions . . . 69

Foreword

This is the user manual of the WP plug-in of Frama-C[1]. The content of this document corresponds to the version 34.0 beta (Selenium), released on November 13, 2026, of Frama-C.

Acknowledgements

ANR This project has received funding from the French ANR project U3CAT (08-SEGI-021-01).

[1] http://frama-c.com

Introduction 1

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 Eva 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 3 provides a description for the included memory models. We present in Chapter 4 the simplifier module and the efficient weakest precondition engine implemented in the WP plug-in. Finally, in Chapter 5, we provide additional information on the supported ACSL built-in symbols.

1.1 Installation

The WP plug-in is distributed with the Frama-C platform. However, it also requires the Why-3 platform and 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 originally developed at INRIA and now by OCAMLPRO[1]. When using the Opam package manager, these tools are automatically installed with Frama-C.

Introduced since Frama-C 33.0 (Arsenic)

WP automatically detects available provers from `$PATH`. It also uses any prover configured from the default Why3 configuration or any additional configuration provided by the user (see options).

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 manual.i (no preprocessing)
[kernel] Parsing working_dir/swap.c (with preprocessing)
[kernel] Parsing working_dir/swap1.h (with preprocessing)
[wp] Running WP plugin...
[wp] [Valid] Goal swap_exits (Cfg) (Unreachable)
[wp] [Valid] Goal swap_terminates (Cfg) (Trivial)
[wp] Warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] [Valid] typed_swap_ensures_A (Alt-Ergo) (Cached)
[wp] [Valid] typed_swap_ensures_B (Qed)
[wp] Proved goals: 4 / 4
  Terminating: 1
  Unreachable: 1
  Qed:         1
  Alt-Ergo:    1
------------------------------------------------------------
 Functions           WP    Alt-Ergo Total Success
```

[1] Alt-Ergo: https://alt-ergo.ocamlpro.com/

```
  swap                3      1      4     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.

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 Eva 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 manual.i (no preprocessing)
[kernel] Parsing working_dir/swap.c (with preprocessing)
[kernel] Parsing working_dir/swap2.h (with preprocessing)
[wp] Running WP plugin...
[wp] Warning: Skipped RTE guards: unaligned pointers (\aligned not supported)
[wp] Warning: Skipped RTE guards: invalid function pointer calls (\valid_function not supporte
[rte:annot] annotating function swap
[wp] [Valid] Goal swap_exits (Cfg) (Unreachable)
[wp] [Valid] Goal swap_terminates (Cfg) (Trivial)
[wp] 8 goals scheduled
[wp] [Valid] typed_swap_ensures_A (Alt-Ergo) (Cached)
[wp] [Valid] typed_swap_ensures_B (Qed)
[wp] [Valid] typed_swap_assert_rte_mem_access (Alt-Ergo) (Cached)
[wp] [Valid] typed_swap_assert_rte_mem_access_2 (Qed)
[wp] [Valid] typed_swap_assert_rte_mem_access_3 (Alt-Ergo) (Cached)
[wp] [Valid] typed_swap_assert_rte_mem_access_4 (Qed)
[wp] [Valid] typed_swap_assigns_part1 (Qed)
[wp] [Valid] typed_swap_assigns_part2 (Qed)
[wp] Proved goals: 10 / 10
  Terminating: 1
  Unreachable: 1
  Qed:         5
  Alt-Ergo:    3
------------------------------------------------------------
 Functions           WP    Alt-Ergo Total Success
  swap                7      3     10     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 [...]
[wp] Warning: Skipped RTE guards: unaligned pointers (\aligned not supported)
[wp] Warning: Skipped RTE guards: invalid function pointer calls (\valid_function not supporte
[rte:annot] annotating function swap
------------------------------------------------------------
 Functions           WP    Alt-Ergo Total Success
  swap                7      3     10     100%
------------------------------------------------------------
[report] Computing properties status...

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

[ Valid ] Post-condition 'A'
          by Wp.typed.
[ Valid ] Post-condition 'B'
          by Wp.typed.
[ Valid ] Exit-condition (generated by Frama-C kernel from "working_dir/swap.c", lines 1-7)
          by Wp.typed.
[ Valid ] Termination-condition (generated by Frama-C kernel from "working_dir/swap.c", lines
          by Wp.typed.
[ Valid ] Assigns ("working_dir/swap2.h", line 4)
          by Wp.typed.
[ Valid ] Assertion 'rte,mem_access' ("working_dir/swap.c", line 3)
          by Wp.typed.
[ Valid ] Assertion 'rte,mem_access' ("working_dir/swap.c", line 4)
          by Wp.typed.
[ Valid ] Assertion 'rte,mem_access' ("working_dir/swap.c", line 4)
          by Wp.typed.
[ Valid ] Assertion 'rte,mem_access' ("working_dir/swap.c", line 5)
          by Wp.typed.
[ Valid ] Default behavior
          by Frama-C kernel.

------------------------------------------------------------------------------
--- Status Report Summary
------------------------------------------------------------------------------
   10 Completely validated
   10 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:

$$\{P\}\ stmt\ \{Q\}$$

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:

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

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

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

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 $W = wp(f, Q)$, 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.

When the `terminates` clause is specified in function contracts, WP generates proof obligations to make sure that all loops have loop variants and that all function calls terminate whenever necessary (thus requiring `decreases` clause for recursive functions). Furthermore, `exit` behaviors of a function are correctly handled by WP. More details about proof of termination in WP are provided in Section 2.4.4.

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 Eva plug-in or the RTE 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 3 for details.

`Bytes` model. (Experimental). This is a low-level memory model, where the heap is represented as an array from pointer base to array of bytes. Read and write operations from/to the heap are translated into manipulation of ranges of bytes in the heap.

By default, it behaves like the `Typed` model, using the `Hoare` model to counterbalance the huge complexity of generated formulas, but it comes with important limitations listed in Section 3.8.

`Region` model. (Experimental) This model relies on a region analysis to maximize the memory separation hypothesis and enable with the possibility to use different ways of modeling the memory in each *region*. It basically combines the capabilities of the `Typed` and `Bytes` models.

This model is currently experimental and comes with limitations that are detailed in Section 3.9.

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 3 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.

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 values are represent in a special theory with dedicated operations over `float` and `double` values and conversion from and to their `real` representation *via* rounding, as defined by the C/ACSL semantics.

Although correct with respect to the IEEE specifications, this model still provides very little support for proving properties with automated provers. You may add additional properties using *drivers* as explained later.

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 Eva analysis plug-in, or by the RTE plug-in. Consider also using `-wp-rte` option.

1.6 Limitations & Roadmap

The ambition of WP plug-in is to cover as many ACSL features as possible. However, some of them are still not available yet, because of lack of manpower or more fundamental reasons. This section provides an overview of those limitations, with roadmap indications: *easy* means that support could be provided on demand, *medium* means that significant manpower is required to implement the feature and *hard* means that a mid-term research project would be required. This list of limitations is probably not exhaustive (yet) and will be maintained over future versions of Frama-C/WP.

Global invariants. Not implemented yet (*easy*).

Type invariants. Type invariants requires to be coupled with new memory models and some memory region analysis (*hard*).

Model fields. This ACSL feature is generally coupled with type invariants and global invariants, hence it is not implemented yet. From a practical point of view, we think that ghost fields with logic types would be very complementary and easier to use. We are waiting for challenging use cases to implement these features (*medium*).

Statement contracts. No more supported since Frama-C 23 (Vanadium) because of unsoundness bugs to be fixed and ACSL restrictions. Support shall be restored on a mid-term basis (*easy*).

Non-natural loops. Loop constructed with goto are no more supported since Frama-C 23 (Vanadium) because of unsoundness bugs to be fixed. A new engine is under construction but is not yet ready (*medium*).

Dynamic allocation. All implemented memory models *are* able to deal with dynamic allocation, which is actually used internally to manage the scope of local variables. However, ACSL clauses for specifying allocation and deallocation are not implemented yet (*medium*).

Assigns. The WP strategy for proving assign clauses is based on a sound but incomplete verification: we check that side effects (writes and called assigns) are *included* in specified assigns, which is a sufficient but not necessary condition. The known workaround is to specify larger assigns and to add the missing equalities to contracts. Indeed, looking for an efficient and more permissive strategy would be challenging (*hard*).

Froms. Proving ACSL assigns-from clauses is not implemented. It is as difficult as proving functional assigns. Although, we have designed some heuristics to prove assigns-from clauses in simple cases that could be implemented on a mid-term basis (*medium*).

Per-behavior assigns. Different assigns clause associated with distinct behaviors are difficult to take into account without a deep refactoring of the WP rule of calls. We currently use a sound upper-approximation of assigns for function calls that might make correct ACSL properties not provable by lack of precision (*medium*).

Bytes, unions & casts. Memory models with non-typed access and bit- or byte-precision access would be easy to implement, although terribly inefficient. We are currently working on a new memory analysis that would provide a deep understanding of how to make *different* memory models soundly working with each others on distinct memory regions. This would deserve a brand new research plan, to be founded by collaborative projects (*hard*).

Bitfields. Adding bitfield support in the typed memory model would be easy.

Floats. A new sound but incomplete model for floats is provided since Frama-C 21 (Scandium). Here, by incomplete we means that it is generally difficult to prove arithmetic properties of float operations by lack of good support from SMT solvers. Although, recent advances on our Colibri solver open the road to a better support for float operations in a near future (*medium/hard*).

Function pointers. Limited support for function pointers is provided *via* an extension of ACSL, see Section 2.4.4 for details. Currently, a function pointer must be provably equal to a finite set of known functions. Although, this could be easily extended to support function contracts refinement (*medium*).

Using the WP Plug-in 2

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 automated theorem provers such as Alt-Ergo [CCK06], CVC5 [BBB+22] and Z3 [dMB08] or by interactive proof assistants like Coq [Coq10] and more generally, any automated or interactive prover supported by Why3 [BFMP11].

2.1 Supported annotations

The WP plug-in supports the following code annotations. We also indicate the number of verification conditions that are generated in the default configuration:

- `lemma`: 1 VC
- `axiom`: no VC (admitted with no proof)
- `ensures`: 1 VC
- `exits`: 1 VC
- `disjoint`: 1 VC
- `complete`: 1 VC
- `requires`: 1 VC for each call
- `terminates`: 1 VC for each call, 1 VC for each loop without "loop variant"
- `decreases`: 1 VC for each recursive call
- `assigns`: 1 VC for each assigned lvalue
- `admit`: no VC (admitted with no proof)
- `assert/check`: 1 VC
- `loop invariant`: 2 VCs (established, preserved)
- `loop variant` (integer): 2 VCs (positive, decreasing)
- `loop variant` (general measure): 1 VC (the measure is *assumed* to be well-founded)
- `loop assigns`: 1 VC for each assigned lvalue within the loop

The statement contracts are not supported. The number of generated verification conditions can vary depending on the selected options. See for example the options listed in 2.4.4.

In addition, the WP plug-in asks the Frama-C kernel to generate automatically some annotations when they are missing:

- `terminates`
- `exits`
- `assigns`, for functions without body

In the default Frama-C configuration, `terminates` is true, `exits` is false. These proofs are generally easily discharged, reason why we enable them by default. The `assigns` clauses for functions without body are inferred from the prototype, this is a best effort generation, and it triggers a warning since this is unsafe. One **shall** check the generated warnings. For more information about the specification generation, please refer to the options `-generated-spec-*` of the kernel and to the Frama-C User Manual [CCK+].

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
```

The WP view can be selected from the "View & Components" sidebar configuration (see "WP view" in figure 2.1). The files and functions can be selected in the "Files" view.

From there, one can start the proof of a property. Just select it in the internal source view and choose WP from the contextual menu (see figure 2.2). After execution of the proof (see 2.3), the "WP goals" component contain the list of the goals that have been verified (or not), together with information about the solvers that have been used, execution time, etc. The computation can also be run for a bundle of properties if the contextual menu is open from a function or behavior selection.

From the WP sidebar (see selector in figure 2.3), one can finely tune WP parameters, typically, select provers, time limits, choose the runtime errors that should be detected, etc. Please refer to section 2.4 for more details. It can also be used to run proofs on all functions of the project. The help button gives access to the documentation integrated to the GUI.

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*.
	The property cannot be satisfied.

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 selecting a goal, one can see the dependencies of the property, double-clicking on a goal gives access to formula that corresponds to the goal.

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.

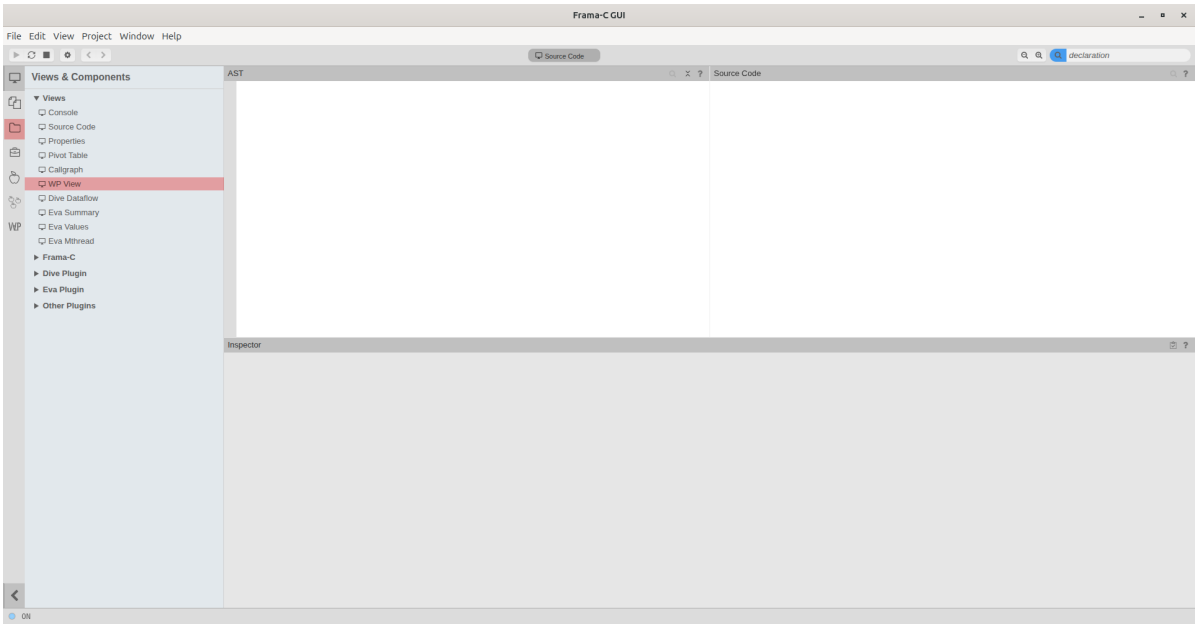


Figure 2.1: The Frama-C GUI

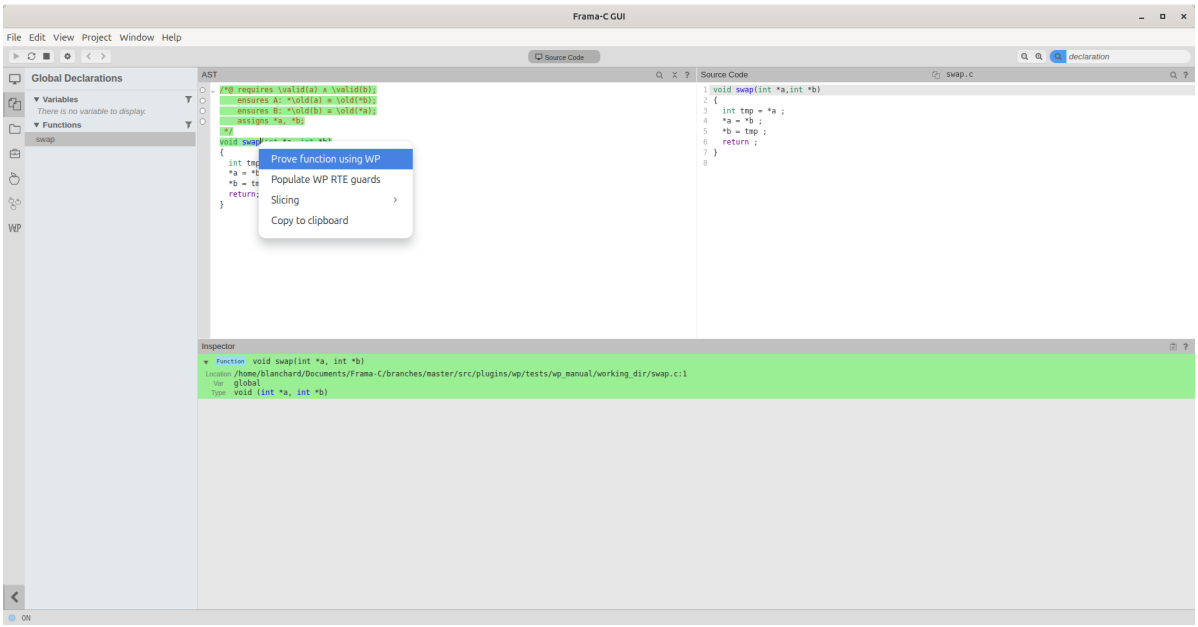


Figure 2.2: Start WP from the GUI

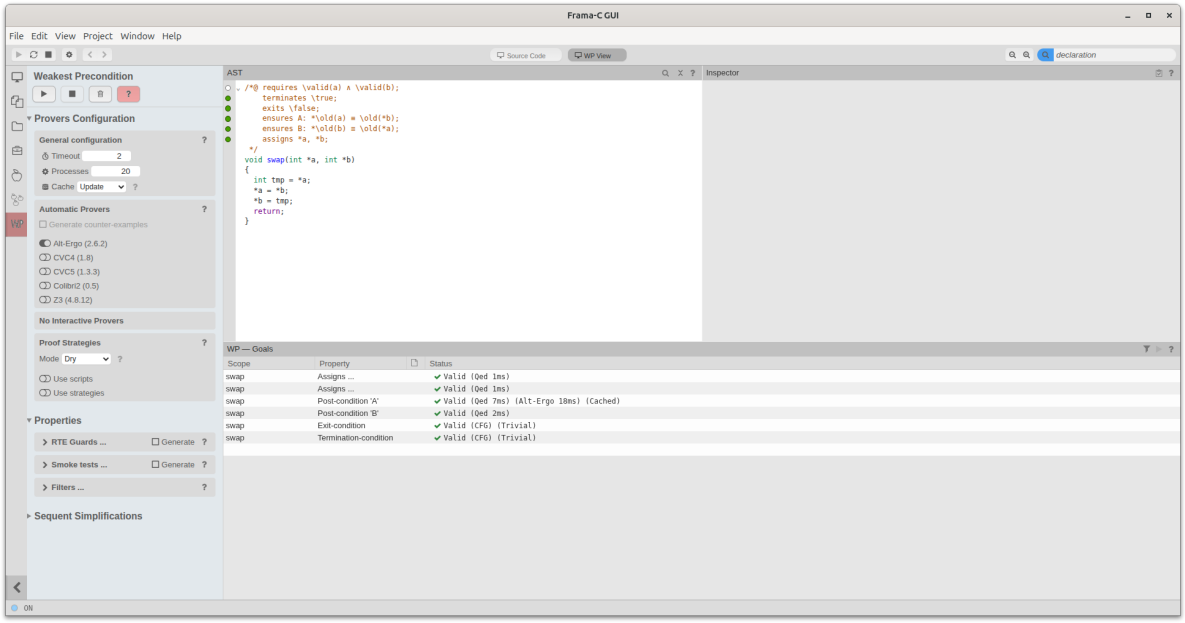


Figure 2.3: WP results and parameters

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*.

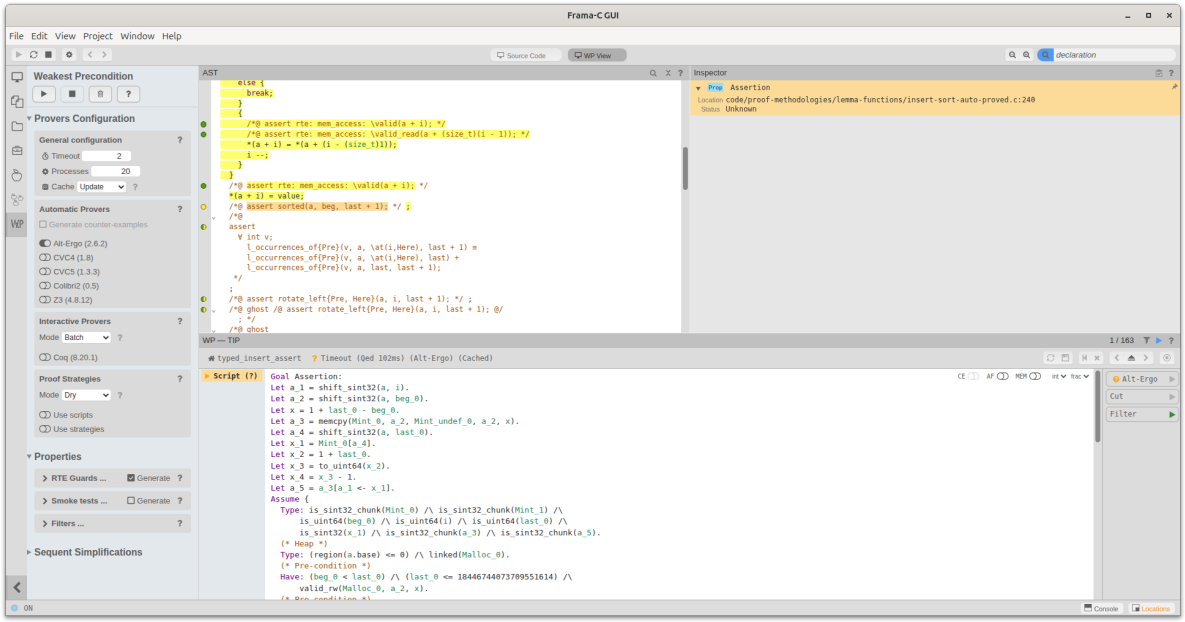


Figure 2.4: Interactive Proof Editing

The general structure of the panel is illustrated figure 2.4. 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 memory updates in executions.

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 tactics require one or several terms to be selected. This feature has been lost when migrating from the old GTK based GUI to the new GUI. If you are interested in funding this (last missing) feature of the old GUI, please contact us.

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.

Notice that, as soon as you have setup a wp-session directory, you benefit from cache facilities to speedup your proofs. Consult Section 2.4.13 for details.

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 Tracking Scripts

When working on large verification use-cases, it is common that some proof scripts become useless because the associated proof obligation does not exist anymore.

WP can track the scripts that are used during verification. For this, one has to initialize a tracking directory. This is done using the option `-wp-prepare-scripts`. This command creates a `.marks` directory in the proof session scripts directory (that can be configured using `-wp-session`), if such a directory already exists it is removed and created again.

Then, each time WP uses a proof script, it is marked as used in the `.marks` directory. In particular, it is then possible to run WP several times to verify different parts of the code, yet marking the scripts for all those different parts.

Once one is certain that the entire verification of the use case has been replayed for marking, the option `-wp-finalize-scripts` can be used to remove all orphan scripts and the marking directory. The option `-wp-dry-finalize-scripts` can be used to log the remove actions instead of actually doing them.

2.3.7 Strategies (DEPRECATED)

S trategies described below are *deprecated* since Frama-C 27.0 (Cobalt). Please use "Proof Strategies" instead (Section 2.5).

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 2.3.11 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.

It is also possible to call strategies from the command line, with option `-wp-auto`. The strategies are tried up to some depth, and while a limited number of pending goals remains unproved by Qed or the selected provers. More precisely:

`-wp-auto s,...` applies strategies `s,...` recursively to unproved goals.

`-wp-auto-depth <n>` limit recursive application of strategies to depth n (default is 5).

`-wp-auto-width <n>` limit application of strategies when there is less than n pending goals (default is 10).

`-wp-auto-backtrack <n>` when the first tried strategies do not close a branch, allows for backtracking on n alternative strategies. Backtracking is performed on goals which are closed to the root proof obligation, hence performing a kind of width-first search strategy, which tends to be more efficient in practice. Backtracking is deactivated by default ($n = 0$) and only used when `-wp-auto` is set.

The name of registered strategies is printed on console by using `-wp-auto '?'`. Custom strategies can be loaded by plug-ins, see below.

2.3.8 General Tactics

Absurd Contradict the Goal or an Hypothesis
The user can prove the goal by contradiction:

$$\Delta \models G \quad \triangleright \quad \Delta, \neg G \models \texttt{false}$$

Or the user can select a hypothesis H, and change the goal to $\neg H$:

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

Clear Remove Hypothesis
The user can select a hypothesis H, and remove it from the context:

$$\Delta, H \models G \quad \triangleright \quad \Delta \models G$$

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

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

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

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

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

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

Cut Use Intermediate 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:

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

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

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

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.

Filter Dependent Erasure of 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 heuristic 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.

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

When instantiating hypothesis with an expression e:

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

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

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

When instantiating a goal with an expression e:

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

Intuition Decompose with Conjunctive/Disjunctive Normal Form
The user can select a 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.

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.

Rewrite Replace Terms
This tactic uses an equality in a 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.

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

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:

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

$$\ldots$$

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 $a > b$:

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

2.3.9 Integers & Bit-wised Tactics

BitRange Range of logical bitwise operators
This tactical applies the two following lemmas to the current goal. The first lemma is on logical-or, and only applies to positive integers:

$$\frac{\bigwedge_i 0 \leq x_i < 2^p}{0 \leq \texttt{lor}(x_1, \ldots, x_n) \leq 2^p}$$

The second lemma is on logical-and, and applies to at-least one positive integer:

$$\frac{\bigvee_i 0 \leq x_i \quad \wedge \quad \bigwedge_i x_i \leq 2^p}{0 \leq \texttt{land}(x_1, \ldots, x_n) \leq 2^p}$$

The tactical rewrites range goals on logical and/or into the corresponding range over its parameters, by finding a suitable 2^p to apply the theorems. Such a strategy is *not* complete in general. Typically, $\texttt{land}(x, y) < 38$ is true whenever both x and y are in range $0 \ldots 31$, but this is also true in other cases.

Bit-Test Range Tighten bounds with respect to bits
The `bit_test(a,b)` function is predefined in WP and is equivalent to the ACSL expression `(a & (1 << k)) != 0`. The Qed engine has many simplification rules that applies to such patterns.

The user selects an expression $\texttt{bit_test}(n, k)$ with k a *constant* integer value greater or equal to 0 and lower than 128. The tactic uses this test to tighten the bounds of n.

$$\begin{array}{rcl} \Delta \models G & \triangleright & \Delta, T \models G \\ & & \Delta, F \models G \end{array}$$

with

$$\begin{array}{rcl} T \equiv & \texttt{bit_test}(n, k) \quad \wedge \quad (0 \leq n & \Rightarrow 2^k \leq n) \\ F \equiv & \neg\texttt{bit_test}(n, k) \quad \wedge \quad (0 \leq n < 2^{k+1} & \Rightarrow n < 2^k) \end{array}$$

Bitwise Decompose equalities over N-bits
The use selects an integer equality and a number of bits. Providing the two members of the equality are in range $0..2^N-1$, the equality is decomposed into N bit-tests equalities:

$$\Delta \models G \quad \triangleright \quad \begin{array}{rcl} \Delta & \models & 0 \leq a, b < 2^N \\ \sigma(\Delta) & \models & \sigma(G) \end{array}$$

where σ is the following substitution:

$$\sigma \equiv \left[a = b \quad \leftarrow \bigwedge_{k \in 0..N-1} \texttt{bit_test}(a,k) = \texttt{bit_test}(b,k) \right]$$

Congruence Simplify Divisions and Products
This tactic rewrites integer comparisons involving products and divisions. The tactic applies one of the following theorems to the current goal. In the following lemmas, k, k', and n are integer constants, a and b any integer terms. The notation $k|n$ stands for k divides n. The lemmas are extended to non-strict inequalities and non-positive constants in a natural way.

$$\begin{array}{rrcl} 0 < k, & a < n/k & \Longrightarrow & k.a < n \\ k|n, & a = n/k & \Longleftrightarrow & k.a = n \\ \neg(k|n), & k.a = n & \Longrightarrow & \texttt{false} \\ 0 < k, & a < k.(b+1) & \Longrightarrow & a/k < b \\ 0 < k, 0 < k', & k'.a < k.b & \Longrightarrow & a/k < b/k' \\ n|k, n|k', & (k/n).a = (k'/n).b & \Longleftrightarrow & k.a = k'.b \end{array}$$

Induction Start a proof by integer induction
The user select any integer expression e in the proof and a base value b (which defaults to 0). The tactic generates a proof by induction on e, that is, the base case $e = b$ and then the cases $e < b$ and $b < e$. Formally, the initial goal $\Delta_0 \models G_0$ is first generalized into $\Delta, P(e) \models Q(e)$. The tactic then proceed by (strong) induction over n for $G(n) \equiv P(n) \Longrightarrow Q(n)$:

$$\Delta \models G(n) \quad \triangleright \quad \begin{array}{lll} \Delta, \quad n = b & \models & G(n) \\ \Delta, \ \forall i,\, b \leq i < n \Longrightarrow G(i) & \models & G(n) \\ \Delta, \ \forall i,\, n < i \leq b \Longrightarrow G(i) & \models & G(n) \end{array}$$

Mod-Mask Rewrite bitmask into/from modulo
This tactic is used to rewrite a bitmask into a modulo (or a modulo into a bitmask) when possible. The user selects an expression e of the form $b\%m$ (resp. $b\&m$ - `land b m`) than can be rewritten into $b\&(m+1)$ (resp. $b\%(m-1)$), if $0 \leq b$ and m is a positive power of 2 (resp. $m+1$ is a positive power of 2). When selecting an expression $x\&y$, both directions $x\%(y-1)$ and $y\%(x-1)$ can be considered.

Since establishing that m is a positive power of 2 can be hard, the tactic has several behaviors. If Qed can prove immediately that an operand m (or $m+1$ for bitmask) is a positive power of 2, the tactic only generates the guard $0 \leq b$ and the rewritten goal. If it is not the case, the tactic appears with the name "Mod-Mask (hard)" in the GUI. In this situation, the guard consists of two conditions: $0 \leq b$ and m (or $m+1$) is a positive power of 2. This guard involves an existential quantification, so it can be hard to prove. When the selected term is a bitmask, the tactic cannot decide itself the direction of the modulo, thus a checkbox is added in the configuration of the tactic to select the direction of rewriting.

Overflow Integer Conversions
This tactic split machine integer conversions into three cases: value in integer range, lower than range and upper than range. The tactic applies on expression with pattern `to_iota(e)` where `iota` is a machine-integer type name, *eg.* `to_uint32`.

$$\begin{array}{lcll} \Delta \models G & \triangleright & \sigma_{id}(\Delta) & \models & low \leq e \leq up \Rightarrow \sigma_{id}(G) \\ & & \sigma_{+}(\Delta) & \models & low < e \Rightarrow \sigma_{+}(G) \\ & & \sigma_{-}(\Delta) & \models & e < up \Rightarrow \sigma_{-}(G) \end{array}$$

where

- $[low..up]$ is the range of the `iota` integer domain,
- $\sigma_{id} = [\texttt{to_iota}(e) \mapsto e]$,
- $\sigma_{+} = [\texttt{to_iota}(e) \mapsto \texttt{overflow}(e)]$,
- $\sigma_{-} = [\texttt{to_iota}(e) \mapsto \texttt{overflow}(e)]$.

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 $e > b$:

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

Shift Transform logical shifts into arithmetics
For positive integers, logical shifts such as `a << k` and `a >> k` where `k` is a constant can be interpreted into a multiplication or a division by 2^k.

When selecting a logical-shift, the tactic performs:

$$\begin{array}{lcll} \Delta \models G & \triangleright & \Delta & \models & 0 \leq a \\ & & \sigma(\Delta) & \models & \sigma(G) \end{array}$$

where: $\sigma = [\texttt{lsl}(a, k) \leftarrow a * 2^k]$ for left-shift,
$\sigma = [\texttt{lsr}(a, k) \leftarrow a/2^k]$ for right-shifts.

2.3.10 Domain Specific Tactics

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

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

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.

Separated Expand Separation Cases
This tactic decompose a $\texttt{separated}(a, n, b, m)$ 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.

Sequence Unroll repeat-sequence operator
In this section, let us use A^n for the ACSL notation `A *^ n`, the repeat list operation, and $A \oplus l$ for the list concatenation.

This tactics is used to transform A^n sequences. Threes behaviors can be selected: unroll left that rewrites the list A^n into $A \oplus A^{n-1}$, unroll right that rewrites the list A^n into $A^{n-1} \oplus A$ and unroll sum that rewrites the list $A^{n_1+\ldots+n_k}$ into $A^{n_1} \oplus \ldots \oplus A^{n_k}$. For unroll left and right, a negative value leads to an empty list. For unroll sum, one must prove that all nI are positive.

Rule for unroll left:

$$\begin{array}{lll} \Delta \models G \quad \triangleright \quad \Delta[A^n \leftarrow A \oplus A^{n-1}], & n > 0 & \models G[A^n \leftarrow A \oplus A^{n-1}] \\ \Delta[A^n \leftarrow []], & n \leq 0 & \models G[A^n \leftarrow []] \end{array}$$

Rule for unroll right:

$$\begin{array}{lll} \Delta \models G \quad \triangleright \quad \Delta[A^n \leftarrow A^{n-1} \oplus A], & n > 0 & \models G[A^n \leftarrow A^{n-1} \oplus A] \\ \Delta[A^n \leftarrow []], & n \leq 0 & \models G[A^n \leftarrow []] \end{array}$$

Rule for unroll sum:

$$\begin{array}{lll} \Delta \models G \quad \triangleright \quad \Delta & & \models \bigwedge_{i=1}^{k} 0 \leq n_i \\ & \Delta[A^{\sum_{i=1}^{k} n_i} \leftarrow \bigoplus_{i=1}^{k} A^{n_i}] & \models G[A^{\sum_{i=1}^{k} n_i} \leftarrow \bigoplus_{i=1}^{k} A^{n_i}] \end{array}$$

Validity Unfold validity and range definitions
The user selects a validity expression (`valid_rd`, `valid_rw`, `invalid` or `included`). The expression is unfolded to a Qed term.

2.3.11 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.`$\star$	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 a full hypothesis or the full goal *)
  | Inside of clause * Lang.F.term
    (** selects a sub-term of a 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 :
  Z.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 (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 2.4.15 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 <f1,...,fn>` selects annotations of functions f_1,...,f_n (defaults to all functions).

`-wp-skip-fct <f1,...,fn>` ignores functions f_1,...,f_n (defaults to none).

`-wp-bhv <b1,...,bn>` selects annotations for behaviors b_1,...b_n (defaults to all behaviors) of the selected functions (the name `default!` can be used to select the default anonymous behavior).

`-wp-prop <p1,...,pn>` selects properties having p_1 or ...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`, `@check`, `@invariant`, `@variant`, `@breaks`, `@continues`, `@returns`, `@terminates`, `@decreases`, `@complete_behaviors`, `@disjoint_behaviors`.
Tags and categories 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.
Tag names also include the name of the function or lemma associated with the property.
Remark: properties with tag name `no_wp:` are always and automatically filtered out and never proved by WP.

`-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`).

`-wp-(no)-legacy` use the legacy WP generator, if set to `no`, WP uses a recently developed generator (default is: `no`).

`-wp-dump` does not prove selected properties, but dumps the control flow graph computed by WP, including code annotations, both DOT files and PDF files into the directory specified by `-wp-out`.

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 3 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.
Region	Select Region memory model with limitations described in its section.
Bytes	Select Bytes memory model.
`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 3.10).
`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 3 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 arithmetic. 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 model 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-(no)-alias-init` Use initializers for aliasing propagation (default is: yes).

`-wp-(no)-volatile` this option (de)activate the correct handling of volatile access. By default, accessing a volatile l-value returns an undefined value, and writing to a volatile l-value is modeled like an ACSL assigns clause. Hence, only the accessed *values* are ignored.
Setting `-wp-no-volatile` turns this behavior off: it is potentially *unsound* and makes the WP emitting a warning on each volatile access.

`-wp-(no)-warn-memory-model` this option (de)activate the warnings about memory model hypotheses for the generated proof obligations, as described in Section 3.6. 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. See also experimental option below.

`-wp-(no)-check-memory-model` this *experimental* option generates ACSL contracts for the selected memory model hypotheses, as described in Section 3.6 and listed by option `-wp-warn-memory-model`. Hence, the memory model hypothes are exposed to WP and other plugins. Disabled by default.

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.

[1] It is still correct to prove these RTE annotations with the WP plug-in.

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

`-wp-(no)-split` Splits conditional statements into separated goals (default is: `no`).

`-wp-(no)-split-switch` Splits switch statements into sepeared goals (default is: `yes`).

`-wp-split-max <n>` limits the number of sub-goals generated by `-wp-split` and `-wp-split-switch` (defaults to `1000`).

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

`-wp-split-cnf <d>` transforms all goals with in conjunctive normal form (CNF) at depth *d*. Value `-1` stands for unlimited depth. Default is 0, which means no CNF transformation.

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

`-wp-(no)-precond-weakening` discard pre-conditions of side behaviours (sound but incomplete optimisation, default is: `no`).

`-wp-unfold-assigns <n>` when proving assigns, unfold up to *n* levels of `struct` compounds types or access via ranges. `struct` are unfolded field by field, ranges are unfolded cell by cell (via a universal quantification over the index of the cell). This allows for proving that assigning a complete structure is still included into an assignment field by field, or that assigning a range is still included in a union of subranges. This option is set to `0` by default, because it is generally not necessary and it can generates a large number of verifications for structures with many (nested) fields. `-1` enables full unfolding.

`-wp-(no-)havoc` selects the behavior of the copy of compounds (default is `yes`).
Starting from version ..., `havoc`[3] has been replaced by `memcpy`[4].
Originally, the operation `havoc` was exclusively used instead of `memcpy`, where the main difference lies in the possibility to specify different locations as source and target.
When this option is set as `yes`, the semantics of replacing `havoc` by `memcpy` is unchanged. The copy of compounds still interprets as a load from the source followed by a store on a fresh map that is then assumed to be equal to the target.
When this option is set as `no`, the semantics of the copy of compounds is a direct `memcpy` from source map/location to target map/location. Therefore, it no longer uses an intermediate fresh map and lemmas expliciting equality between the modified pre map and the post map.

`-wp-(no)-variant-with-terminates` prove `loop variant` under the termination hypothesis (thus, under a `terminates \false`, any loop variant proof is trivial) (default is: `no`).

ACSL extension `@calls`

Frama-C provides an ACSL extension `@calls` that can be written before an indirect call to indicate which functions can be the target of the pointer (see [CCK+] for more information). Upon encountering such annotation, WP will perform a case analysis with the contract of each provided function.

For example:

---

[2] Proof obligations are always generated to check preconditions.

[3] it copies of a memory segment from a source map `msrc` to a target map `mtgt`, starting at the same location `loc` of size `length`, it is mainly used with a fresh source map in the context of an `assign`.

[4] it copies of a memory segment from a source map `msrc` at a source location `lsrc` to a target map `mtgt` at a target location `ltgt` of size `length`.

```
/*@ ensures \result == x ; */
int f1(int x);
/*@ ensures \result == x+1 ; */
int f2(int x);

/*@ requires fp == &f1 || fp == &f2 ;
    ensures POST: \result == 4 || \result == 5; */
int caller(int (*fp)(int)){
  //@ calls f1, f2 ;
  return (*fp)(4);
}
```

These annotations split post-conditions to the dynamic call into two sub-goals: one for call to `f1` and a second for the call to `f2`. A last goal is generated at the call site: one must prove that `fp` is either `f1` or `f2`.

Proof of `@terminates`

In ACSL, a function can be specified to terminate when some condition `P` holds in pre-condition.

```
/*@ terminates P ; */
void function(void){
  ...
}
```

If terminates verification is active (depending on `-wp` options) WP collects all function calls and loops and generate verification conditions (VCs) to prove them, as detailed below.

Loop variants. When a loop has a variant, no particular verification is generated for termination. However, if `-wp-variant-with-terminates` is active, the variant is verified only when `P` holds in pre-condition. Intuitively, this option means that variant has to decrease only if the function has to terminate. For example:

```
//@ terminates i >= 0 ;
void positive(int i){
  /*@ loop invariant \at(i, Pre) >= 0 ==> 0 <= i <= \at(i, Pre) ;
      loop assigns i ;
      loop variant i ; // verified when i >= 0 at pre
  */
  while(i) --i;
}
```

Note that when `terminates \false` is specified, any loop variant would trivially holds.

When a loop does not have a variant, WP checks that the loop is unreachable when `P` holds in precondition, which means that a loop is allowed to not terminate when the function does not have to terminate. For example:

```
//@ terminates i >= 0 ;
void function(int i){
  if(i < 0){
    // note i >= 0 && i < 0 ==> \false
    while(1);
  }
}
```

Function call. When a function `f` is required to terminate on `P`, and calling function `g` only terminates when `Q`, WP tries to prove that `P@pre ==> Q@call-point`. Indeed, for `f` to terminates while calling `g`, then `g` shall terminate. Note that it means that if the function under verification shall terminate, any call to a non-terminating function shall be unreachable. For example:

```
//@ terminates i <= 0 ;
void f(int i) ;

//@ terminates i >  0 ;
void g(int i) ;

/*@ requires i <= 0 ;
    terminates \true ; */
void caller(int i){
  f(i); // succeeds
  g(i); // fails, thus global termination is not proved
}
```

When a called function does not have a terminates specification, WP assumes that it does not terminate.

(Mutually) recursive function. When a function is involved in a recursive cluster (as defined in the ACSL manual) and shall terminate, WP checks that a `decreases` clause is available for the function. Proving that existing recursive calls might be unreachable is delegated to the proof of the `decreases` clause.

Function pointers. On a call via a function pointer, WP checks that a `@calls` is specified for this function pointer. If so, WP just behaves as explained before for function calls. If such a specification is not available, WP considers that the pointed function might call the function under verification and thus that it is part of a recursive cluster and shall provide a `decreases` clause.

2.4.5 Smoke Tests

During modular deductive verification, inconsistencies in function requirements can be difficult to detect until you actually call it. Although, such inconsistencies make its post-conditions provable, while its pre-conditions would never be provable.

The WP plug-in can generate smoke-tests to detect such inconsistencies. Basically, it consists in checking if `\false` is provable under the requirements or assumptions of a behavior, or under the invariants of a loop. Also, a simple reachability analysis is performed to track dead-code statements and calls that neither terminates nor exits.

This is best-effort verification : if at least one prover succeed in proving `\false`, an inconsistency is detected. Otherwise, the test is not conclusive, and you can never be sure that the ACSL annotations are free of inconsistencies.

In case any smoke-test fails, a "*False if reachable*" status is put on the inconsistent requirements, or on the loop with inconsistent invariants, and WP generates a user warning.

`-wp-(no)-smoke-tests` generates checks to detect inconsistencies (default is: `no`).

`-wp-smoke-timeout` timeout to be used for trying to prove `\false` on smoke-tests (default is `2` seconds).

`-wp-(no)-smoke-dead-assumes` includes smoke tests for dead assumes (default is: `yes`).

`-wp-(no)-smoke-dead-call` includes smoke tests for non-terminating calls (default is: `yes`).

`-wp-(no)-smoke-dead-code` includes smoke tests for dead code (default is: `yes`).

`-wp-(no)-smoke-dead-local-init` includes smoke tests for dead local variables initialization (default is: `no`).

`-wp-(no)-smoke-dead-loop` includes smoke tests for inconsistent loop invariants (default is: `yes`).

When reporting prover results for smoke-tests, the WP displays "Failed" when some prover succeeds in discharging the `\false` proof-obligation and "Passed" when all the provers result are unknown or interrupted. In the final prover statistics, the interrupted smoke tests are *not* reported, since they are considered valid tests.

2.4.6 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.

2.4.7 Qed Simplifier Engine

These options control the simplifications performed by the WP plug-in before sending proof obligations to external provers. The default simplifiers can be controlled by the following options:

`-wp-(no)-qed` enable Qed simplifications (default is: `yes`).

`-wp-(no)-subst` extract and propagate equalities from the proof context (default is: `yes`).

`-wp-(no)-filter` filter non used variables and related hypotheses (default is: `yes`).

`-wp-(no)-core` factorize common properties between branches (default is: `yes`).

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

`-wp-(no)-clean` removes unused terms and variables from proof obligations (only available when qed simplifications are globally disabled) (default is: `yes`).

`-wp-(no)-ground` replace ground values in equalities (default is: `yes`).

`-wp-(no)-extensional` use extensional equality on compounds (default is: `yes`).

`-wp-(no)-reduce` replace functions with precedence to constructors and operators (default is: `yes`).

`-wp-(no)-parasite` eliminate parasite variables (default is: `yes`).

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

`-wp-(no)-init-summarize-array` summarize contiguous initializers with quantified formulae (default: `yes`).

`-wp-(no)-simplify-is-cint` eliminates redundant constraints on integers (default: `yes`).

`-wp-(no)-simplify-land-mask` tight constants in logical-and with unsigned integers (default: `yes`).

`-wp-(no)-prenex` normalize nested quantifiers into prenex-form (default: `no`).

`-wp-(no)-simplify-forall` eliminates integer ranges in quantifiers (*unsound*, to be used with caution, default is: `no`).

`-wp-(no)-simplify-type` remove type constraints from proof obligation (*incomplete*, default is: `no`).

`-wp-bound-forall-unfolding <n>` instantiates statically `n` instances of k for hypothesis $\forall k \in [n_1..n_2], a = b$ (default is: `1000`).

2.4.8 Prover Selection

The generated proof obligations are submitted to external decision procedures run through the Why-3 platform. 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.

Support for Why-3 IDE is no longer provided.

Since Frama-C 22.0 (Titanium) support for Coq interactive prover has been added and might also work with other interactive provers. See `-wp-interactive <mode>` option for details.

`-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.

You may also select provers `"tip"` to use proof strategies and `"script"` to use proof scripts (without applying new proof strategies), see option `"-wp-script"` below for more details.

`-wp-script <mode>` selects how to manage proof strategies and proof scripts session. Several modes are available:

- `"batch"` use existing scripts, but do not store new scripts. This is the default mode for prover `"script"`.
- `"update"` use existing scripts and store new or updated scripts in proof session. This is the default mode for prover `tip`.
- `"init"` don't use existing scripts and store the newly generated scripts in proof session.
- `"dry"` don't use existing script and don't store any new scripts in proof session.

This parameter can be customized using the `FRAMAC_WP_SCRIPT` environment variable. If both the variable and the option are positioned, the option has the priority.

`-wp-interactive <mode>` selects the interaction mode with interactive provers such as Coq (while it could work for other interactive provers, it has not been tested so far). Five modes are available:

- `"batch"` mode only check existing scripts (the default);
- `"update"` mode update the script and then checks it;
- `"edit"` mode opens the default prover editor on each generated goal;
- `"fix"` mode only opens editor for non-proved goals;
- `"fixup"` combines `"update"` and `"fix"` (if necessary).

New scripts are created in the `interactive` directory of WP session (see option `-wp-session <dir>`).

`-wp-list-provers` lists all available provers. The option prints all the available provers from Why3 configuration and from the current `PATH` with their respective version and `-wp-prover <p>` code name. See `-wp-why3-config` and `-wp-why3-detect` for details.

`-wp-(no)-run-all-provers` Run all specified provers on each goal not proved by Qed. When a prover succeeds in proving the goal, the others are not stopped. (default is: no).

`-wp-strategy s,...` specifies the default ones to be used, see Section 2.5 for details.

`-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 the number of logical cores on the machine. 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. (default is: 60)

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

`-wp-timeout <t>` sets the timeout (in seconds) for the calls to the decision prover (defaults to 2 seconds).

`-wp-memlimit <m>` sets the memory-limit (in Mb) for the calls to the decision prover (defaults to 1000 Mb).

`-wp-fct-timeout <f1:t1,f2:t2,...>` customize the timeout for each specified function (`t1` for `f1`, `t2` for `f2`, etc).

`-wp-interactive-timeout <n>` sets the timeout (in seconds) for checking edited scripts with interactive provers (defaults to 30 seconds).

`-wp-(no)-script-on-stdout` displays generated proof scripts on stdout instead of saving them on disk (default is: `no`).

`-wp-time-extra <n>` additional time allocated to provers when replaying a script. This is used to cope with variable machine load. Default is `5s`.

`-wp-time-margin <n>` margin time for considering a proof to be replayable without a script. When a proof succeed within `timeout-margin` seconds, it is considered fully automatic. Otherwise, a script is created by prover `tip` to register the proof time. This is used to decrease the impact of machine load when proof time is closed to the timeout. Default is `5s`.

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

`-wp-why3-config=<file>` use the specified Why3 configuration file for configuring provers. By default, uses `'$HOME/.why3.conf'` if it exists.

`-wp-(no)-why3-detect` collect all available provers from `$PATH` in addition to the specified or default Why3 configuration (set by default).

`-wp-why3-opt='options,...'` provides additional options to the `why3` command.

`-wp-why3-extra-config='files,...'` provides additional configuration files for `why3`.

`-wp-prepare-scripts` initialize a script tracking directory in the session directory.

`-wp-finalize-scripts` remove untracked scripts according to the tracking directory if it does exist (does not remove anything otherwise).

`-wp-dry-finalize-scripts` scripts that might be removed by `-wp-finalize-scripts` are kept, a message is printed instead for each file. The marks directory is kept.

Example of using provers. Suppose you have the following configuration:

```
# frama-c -wp-list-provers
[wp] Prover Alt-Ergo 2.0.0 [alt-ergo|altergo|Alt-Ergo:2.0.0]
[wp] Prover   CVC4 1.6 [cvc4|CVC4:1.6]
[wp] Prover   CVC4 1.6 [cvc4-ce|CVC4:1.6,counterexamples]
[wp] Prover Eprover 2.4 [eprover|Eprover:2.4]
[wp] Prover    Z3 4.8.7 [z3-ce|Z3:4.8.7:counterexamples]
```

Then, to use (for instance) CVC4 1.6, you can use `-wp-prover cvc4` or `-wp-prover CVC4:1.6`. Similarly, if you want to use Z3 4.6.0 without bitvectors, you can use `-wp-prover z3-nobv`. Finally, since Why-3 also provides the alias `altergo` for this prover, `-wp-prover altergo` will also run it *via* Why-3.

If you require on command line a specific version of a prover which is not installed, WP will emit an error. When a script contains a couple prover/version that is not installed, WP will try to find another version of the same prover and fallback on this solver (emitting a warning) if it finds one. In this case, one should update the scripts.

Notice that Why-3 provers benefit from a cache management when used in combination with a WP-session, see Section 2.4.13 for more details.

2.4.9 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-print` pretty-prints the generated proof obligations on the standard output. Results obtained by provers are reported as well.

`-wp-status` pretty-prints only proof obligations that are not (yet) proved on the standard output. In case of script, prints all the non-proved sub-goals of the script.

`-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.

2.4.10 Probes and Counter Examples

Introduced since Frama-C 29.0 (Copper)

Depending on prover capabilities, WP can trigger counter example generation for non successful goals. Provers supporting counter example generation can be identified by the `"counterexamples"` variant after their name. For example, in the prover selection example above, `cvc4-ce` and `z3-ce` are such provers.

To enable counter examples generation by WP you shall use `-wp-counter-examples` option. Then, for each selected solver that has a counter-examples variant, this variant will be used instead and counter-examples will be asked to the solver in case of unknown result. Notice that you can also use counter-examples solvers *without* asking for counter-examples. For instance, with option `-wp-prover cvc4-ce`, WP will use this version of the prover, even if counter examples is not active; the prover will not be asked to report models, unless counter examples *is* activate. Notice that solvers variants with counter-examples support are generally a bit slower than their original version.

A proof obligation for which a counter-example has been generated is reported by WP by the string `"(Model)"` after its proof status. For instance:

```
# ./bin/frama-c -wp ~/work/wrong.i -wp-prover cvc4 -wp-counter-examples
[...]
[wp] 6 goals scheduled
[wp] [Unknown] typed_g_ensures (Qed 4ms) (CVC4) (No Cache) (Model)
[wp] [Cache] not used
[wp] Proved goals: 6 / 7
  Unreachable: 1
  Qed:         5 (0.72ms)
  Unknown:     1
```

Notice that cache usage is disabled for provers asked to generate a counter-example, since models are currently not stored in WP's cache[5].

A counter-example consists in a *Model*, that is, a map from *Variables* or *Probes* (see below) to *Values*, for which the proof obligation formula is expected to be `False`. More precisely, the Model reports on *necessary* conditions for the proof obligation to be wrong, although they might not be *sufficient* in practice.

Generated models are reported by WP along with prover results when either `-wp-print` or `-wp-status` options are used. The interactive proof transformer (TIP) also displays models in the graphical user interface, when available.

Probes are generalization of variables. They are named expressions for which the prover is asked to output a value in the model. WP will automatically introduce probes for formal parameters of functions and universally quantified variables of lemmas. Developers can introduce their own probes by using the `@probe` annotation. For instance:

```
  int a[4][5];
  /*@
    ensures \false;
    assigns \nothing;
  */
  int value(int i, int j) {
    //@ probe Offset: &a[i][j] - &a[0][0];
    return a[i][j];
  }
}
```

Probes *must* be named ACSL terms. As a short cut, for very simple probes such as variables, the name can be omitted. Several probes might have the same name, although it is not really convenient for debugging. The general syntax for the annotation is:

$$\texttt{probe}\ a_1 : t_1, \ldots, a_n : t_n\ ;$$

Probes are normalized by Qed during simplification of proof obligations by WP. Their (simplified) term definitions are reported when printing proof obligations, and their model values are reported when printing proof results, when available. For instance:

```
# frama-c [...] -wp-status
[...]
Goal Post-condition (file /Users/correnson/work/matrix.i, line 3) in 'value':
Assume {
  Type: is_sint32(i) /\ is_sint32(j).
  Probe i = i.
  Probe j = j.
  Probe Offset = ((4 * j) + (20 * i)) / 4.
}
```

[5] This feature might be added in future versions of the plugin.

```
Prove: false.
Prover CVC4 1.8 returns Unknown (Model) (Qed:2ms)
Model i = 0
Model j = 0
Model Offset = 0
[...]
```

User probes are always reported, even if `-wp-counter-examples` is not activated. This feature reveals to be extremely useful for debugging proofs in practice, even when counter examples is not available or not activated. It also helps in understanding C, ACSL and WP models and encoding.

You may freely add as many probes as you want: probes do not modify the proof obligations submitted to the provers, unless `-wp-counter-examples` is activated. For instance, in the previous example:

```
# frama-c [...] -wp-status -wp-no-counter-examples
[...]
Goal Post-condition (file /Users/correnson/work/matrix.i, line 3) in 'value':
Assume {
  Type: is_sint32(i) /\ is_sint32(j).
  Probe Offset = ((4 * j) + (20 * i)) / 4.
}
Prove: false.
Prover CVC4 1.8 returns Unknown (Qed:2ms) (Cached)
[...]
```

Summary of WP options related to probes and counter-examples generation:

`-wp-(no)-counter-examples` enables counter examples generation for supporting provers. Also generate probes for formal parameters of functions and universal quantifiers of lemmas. Not set by default.

`-wp-print, -wp-status` pretty-prints the probes and the associated models, when available.

`-wp-solver xxx-ce` use the specified variant of prover, even when counter-examples is not activated.

`-wp-(no)-probes` registers the `@probe` ACSL extension in Frama-C kernel. This option is set by default, but you may want to deactivate it to avoid any clash with extensions from other plugins.

2.4.11 Linking ACSL Symbols to External Libraries

DEPRECATED. You shall now import a Why3 module instead of using custom drivers. See Section 2.4.12 for details.

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.

Declared logic types, constructors, logic functions and logic predicates can be linked to external Why3 library symbols, or aliased to other existing ACSL builtins or declared symbols.

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

```
      type  symbol                      = ;
predicate  symbol ( type,...,type )     = "why3-symbol";
      ctor  type symbol ( type,...,type ) = "why3-symbol";
     logic  type symbol ( type,...,type ) = property-tags "why3-symbol";
```

It is also possible to define *aliases* with other ACSL symbols, rather than external links. In this case, you may replace = by `:=` and use an ACSL identifier in place of the external `"why3-symbol"`. No property tag is allowed when defining aliases.

C-Comments are allowed in the file. For overloaded ACSL symbols, it is necessary to provide one `"why3-symbol"`symbol for each existing signature. See also the default WP driver file for examples of external links, in file `wp.driver` of the shared library.

Optional *property-tags* can be given to `logic "why3-symbol"`symbols to allow the WP plugin to perform additional simplifications (See section 4). Tags consist of an identifier with column (‘`:`’), sometimes followed by an ACSL symbol. The available tags are depicted on figure 2.5.

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:`
`absorbent` e`:`	specify the ACSL absorbent element of the operator: $e \odot x = x \odot e = e$
`neutral` e`:`	specify the ACSL neutral element of the operator: $e \odot x = x \odot e = x$
`invertible:`	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: $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \implies \forall i\ x_i = y_i$
`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 2.5: Driver Property Tags

2.4.12 Importing Why3 Modules

The WP plug-in provides an ACSL module importer that can import Why3 theories and modules into ACSL. See Frama-C user manual, Section § 6.3 for details. Here is an example of use:

```
//@ import \wp::why3: number::Gcd;

/*@
  assigns \nothing;
  ensures \result == Gcd::gcd(a,b);
*/
int euclid(int a, int b)
{
  int r;
  /*@
    loop assigns a, b, r;
    loop invariant Gcd::gcd(a,b) == \at( Gcd::gcd(a,b), Pre );
    loop variant \abs(b);
  */
  while( b != 0 ) {
    r = b ;
    b = a % b ;
    a = r ;
  }
  return a < 0 ? -a : a;
}
```

The Why3 load-path can be specified from the command-line:

`-wp-library <dir,...>` add the specified directory to the Why3 load-path.

Notice that the Why3 standard library is already in the load-path by default.

2.4.13 Proof Session & Cache

The WP plugin can use a session directory to store information to be used from one execution to another one. It is used to store proof scripts edited from the TIP (see Section 2.3) and to replay them from the command line. And it is also used to speedup the invocation of provers by reusing previous runs.

Actually, running provers can be demanding in terms of memory and CPU resources. When working interactively or incrementally, it is often the case where most proof obligations remain unchanged from one WP execution to the other. To reduce this costs, a cache of prover results can be used and stored in your session.

There are different ways of using the cache, depending on your precise needs. The main option to control cache usage is `-wp-cache`, documented below:

`-wp-cache <mode>` selects the cache mode to use with Why-3 provers. The default mode is `update` if there is WP session available, and `none` otherwise. The environment variable `FRAMAC_WP_CACHE` can be used for the same purpose, if both the environment variable and the option are configured, the option is prioritary.

`-wp-cache-dir <dir>` select the directory where cached results are stored. By default, the cache is store in the WP session directory. The environment variable `FRAMAC_WP_CACHEDIR` can be used to customize the directory, if both the environment variable and the option are configured, the option is prioritary.

The available cache modes are described below:

`-wp-cache update` : use cache entries or run the provers, and store new results in the cache. This mode is useful when you are working interactively : proofs can be partial and volatile, and you want to accumulate and keep as much previous results as you can.

`-wp-cache cleanup` : same as `update` mode but at the end of Frama-C execution, any cache entry that was not used nor updated will be deleted. This mode shall be only used when you want to cleanup your cache with old useless entries, typically at the end of an interactive session.

`-wp-cache replay` : same as `update` mode but new results are *not* stored in the cache. This mode is useful for continuous integration, when you are not sure your cache is complete but don't want to modify it.

`-wp-cache offline` : similar to `replay` mode but cache entries are the unique source of results. Provers are never run and missing cache entries would result in a « Failed » verdict. This mode is useful to fasten continuous integration and enforcing cache completeness.

`-wp-cache rebuild` : force prover execution and store all results in the cache. Previous results will be replaced with new ones, but entries for non relevant proofs would be kept and you might need a cleanup stage after. This mode is useful when you modify your Why-3 or prover installation and you don't want to reuse your previous cache entries.

`-wp-cache none` : do not use nor modify cache entries; provers are run normally. This option must be used if you have a session set and you don't want to use the cache, since the default is mode `update` in this case. But you probably always to benefit from a cache when you have an (interactive) session.

When using cache with a non-`offline` mode, time and steps limits recorded in the cache are compared to the command line settings to produce meaningful and consistent results. Hence, if you provide more time or more steps from the command line than before, the prover would be run again. If you provide less or equal limits, the cache entries are reused, but WP still report the cached time and step limits to inform you of your previous attempts. For instance, if you have in the cache a « Valid » entry with time 12.4s and re-run it with a timeout of 5s, you will have a « Timeout » result with time 12.4s printed on the console.

2.4.14 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 report template specification file. See Section 2.8 for details on their structure.

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

`-wp-report-json <file>.json` output the proof report in JSON format. See Section 2.7 for details on the JSON report format.

2.4.15 Persistent Options

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 these 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-provers` This option configure the provers to use for generated proof-obligations. It is cleared after WP run so that a Frama-C run like `-wp-fct f -wp-prover z3 -then -wp-fct g -wp-prover cvc5` indeed uses Z3 to prove `f` and CVC5 to prove `g` (and not Z3 + CVC5).

`-wp-print, -wp-status.` These options configure feedback on generated proof obligations, generally they are used for debugging proofs thus positioned to debug particular proofs not all of them. Thus, they are not saved and cleared after a WP run.

`-wp-list-provers, -wp-tactic` These options are used to display information about available provers and tactics, they are run when Frama-C exits.

`-wp-gen -wp-dump` These options are mostly advanced debugging options. Thus, they are not saved and cleared after a WP run.

`-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.

2.5 Proof Strategies

Introduced since Frama-C 28.0 (Nickel)

Proof obligations generated by WP are usually discharged by an SMT solver specified by the user through command line option `-wp-prover`. As described in previous sections, complex proof obligations may also be split into simpler sub-goals by applying *Tactics* from the TIP user interface.

Proof strategies provide user-defined heuristics to automatically try various combinations of provers, timeouts and tactics, depending on the proof context. This is a much more effective technique than relying on manually edited scripts through the TIP user interface. Here are some benefits of using proof strategies:

- Proof strategies are automatic: there is no need for entering GUI session.
- Proof strategies can be associated to individual functions, lemmas or properties, or tried globally.
- Tactics are applied following patterns: depending on your case study, you can define fine-tuned strategies to solve your common issues.
- Proof strategies are compatible with TIP-based scripts: you can use strategies to complete scripts and conversely.
- Scripts are easier to replay and to update: each proof node now records which strategy was involved in the past, so that incomplete or broken proofs can be recovered easily.

To activate proof strategies and search for new scripts for unproven goals, user shall activate the `tip` prover from the command line, with `-wp-prover tip`, or use the interfactive TIP from the graphical user interface.

Proof Strategies are defined thanks to dedicated ACSL extensions and controlled by specific command-line options. Note that those extensions are only available unless command-line option `-wp-no-strategy-engine` is specified. The syntax of ACSL extensions for proof strategies is defined below:

`@\wp::strategy` *name*:
alternative,
...,
alternative;

`@\wp::proof` *name*: *target*, ..., *target*;

A *strategy* must be defined by a unique name and consists of a list of *alternatives* that will be tried in order until proof completion.

Proof hints can be specified to associate strategies to lemma, function or property names. Hints consists of a strategy name to be used for one or many proof targets. Targets follow the same conventions than `-wp-prop` command-line option, although quotes might be necessary, e.g. `"@requires"`. A proof hint specification applies for all properties matching the property filter. Many proof hints might apply, in which case, proof hints will be tried in order.

Strategies shall be defined *before* being used as proof hints. It is recommended to write strategies in separated `*.c` or `*.i` files and to write proof hints right after their target definition.

Strategy consists of list of alternatives with the following syntax:

alternative ::=
| `\prover("p",...,"p",` *timeout*)
| `\tactic("id",` *param*, ..., *param*)
| `\auto("id",...,"id")`
| `\default`
| *strategy*

Each form *alternative* consists of one possible solution to discharge a proof obligation:

- `\provers(p,...,p,time)` will try the specified provers in sequence with a specified timeout. Provers and/or timeout can be omitted, in such a case the command-line options `-wp-prover` and `-wp-timeout` are used instead. Sub-second timeout can also be specified, eg. `0.2` for 200 milliseconds. Prover names must be individually quoted, with the same syntax as `-wp-prover` command line option.
- `\tactic(id,prm...)` will try to apply the specified tactic with the associated parameters. Tactic parameters allow to select the target of the tactic, and the strategy to be used on generated sub-goals.

- `\auto(id,...)` will try to apply the (deprecated) auto-strategies available in WP versions prior to Cobalt.
- *name* or `\default` tries the specified strategy.

The default strategy consists in applying the strategies specified through the command-line option `-wp-strategy s,...`, in order.

Tactic parameters are specified using the following syntax:

param ::=
| `\goal`(*pattern*,...)
| `\when`(*pattern*,...)
| `\ingoal`(*pattern*,...)
| `\incontext`(*pattern*,...)
| `\pattern`(*pattern*,...)
| `\select`(*value*)
| `\param`(*id*,*value*)
| `\child`(*id*,*strategy*)
| `\children`(*strategy*)

Tactic parameters are interpreted as follows:

- `\goal(ps)`, `\when(ps)` and other patterns are used to match parts of the current proof node. The form `\when(ps)` looks for top-level hypotheses patterns, `\goal(ps)` for top-level goal patterns. Form `\ingoal(ps)` and `\incontext(ps)` respectectively matches patterns deeper inside the goal or the hypotheses, and finally `\pattern(ps)` searches in the entire proof obligation. When several patterns are given, all must apply and agree on common pattern variables. Pattern syntax is specified below.
- `\select(v)` defines the main target of the selected tactic. The value `v` uses the same syntax as patterns, with pattern variables replaced with their matched values. Few tactics support multiple selection, in which case, multiple values can be provided. When no selection is specified, the first specified pattern is used as an implicit target. When no selection nor pattern is selected, the default selection is the proof goal.
- `\param(p,v)` configures the parameter value `p` of the selected tactic with the associated value.
- `\child(a,s)` applies strategy `s` to prove the sub-goal named `a` of the tactic.
- `\children(s)` applies strategy `s` on (other) sub-goals of the tactic. If unspecified, the strategy used for the current node is applied.

Tactics identifiers and their parameters can be obtained from the command line:

- `frama-c -wp-tactic ?` lists all the registered tactic identifiers;
- `frama-c -wp-tactic id,...` prints detailed infos for the selected tactics.

Patterns and values are a closed to raw proof obligation formulae, with the following restrictions and meaning:

- variables are pattern variables;
- pattern _ matches any value;
- pattern `x:p` matches pattern `p` and bind its value to variable `x`;
- pattern `a..b` matches integer constants in the specified (constant) range.
- pattern `\any(p,...)` matches any of the specified patterns.
- patterns `p[k]` and `p[k->v]` matches array operations, e.g. for memory models;
- ACSL operators have their usual meaning;
- ACSL list operators `[|...|]`, `^` and `*^` are also recognized;
- function calls must use their printed name, as they appear in proof nodes;
- for function patterns, trailing dots can be used to match any number of arguments, like in `f(x,y,(..))` for instance.

It is highly recommended using the graphical user interface or `-wp-print` to build patterns expression and values.

Smoking Strategies. When smoke-tests are activated by `-wp-smoke-tests`, strategies are deactivated by default on smoking goals. Indeed, smoke tests are proof obligations that are expected to *fail*. However, it might be interesting to try strategies on smoking proof goals in order to stress-test further your code. To activate strategies for smoke-tests, you shall add the `smoke` name to the strategies to use:

`@\wp::proof` *strategy* : smoke,*target* , . . . , *target* ;

2.6 Structure of JSON Scripts

The proof scripts generated by using the Interactive Proof Editor (Section 2.3) are saved in the Frama-C/WP session as JSON files. This section defines the format of session scripts precisely.

JSON scripts files are located in the `<session>/wp/scripts` sub-directory of the Frama-C session directory (set with command line option `-session`) or in the `<wpsession>/scripts` sub-directory of the Frama-C/WP session directory (set with command line option `-wp-session`).

The proof scripts for goal named `<goal>` is stored in a single file `<goal>.json` ; the format of the `<goal>` name is identical to the one used by the Frama-C/Report plug-in for its JSON output.

The content of each script file is an array of alternatives, each alternative being a record with two possible structures, that might represent a *prover* attempt or the application of a *tactic*:

$$\begin{aligned} \mathit{wp.script} &::= [\mathit{wp.alternative}, \dots] \\ \mathit{wp.alternative} &::= \mathit{wp.prover} \mid \mathit{wp.tactic} \end{aligned}$$

Prover Attempts The proof script records previous attempts to discharge a proof obligation with external provers (also called decision procedures). Each attempt is represented by a JSON record with the following fields:

$$\begin{aligned} \mathit{wp.prover} ::= & \left\{ \begin{array}{rl} \texttt{"prover"}: & \mathit{string}, \\ \texttt{"verdict"}: & \mathit{wp.verdict}, \\ \texttt{"time"}: & \mathit{number}? \\ \texttt{"steps"}: & \mathit{number}? \\ \texttt{"depth"}: & \mathit{number}? \end{array} \right\} \\ \mathit{wp.verdict} ::= & \texttt{"none"} \mid \texttt{"valid"} \mid \texttt{"unknown"} \mid \\ & \texttt{"timeout"} \mid \texttt{"stepout"} \mid \texttt{"invalid"} \mid \texttt{"failed"} \end{aligned}$$

Remark that, since the proof scripts do not record the proof obligation that was exercised, there is no guarantee that replaying the script on a (potentially new) proof obligation would issue the same result. Hence, prover attempts *must* not be used as a cache, but can serve as a hint for choosing among several alternatives.

Proof Tactic Applying a tactic is represented by a JSON record with the following fields:

$$\mathit{wp.tactic} ::= \left\{ \begin{array}{rl} \texttt{"tactic"}: & \mathit{string}, \\ \texttt{"header"}: & \mathit{string}? \\ \texttt{"select"}: & \mathit{wp.selection}, \\ \texttt{"params"}: & \{\ \mathit{string} : \mathit{wp.param}, \dots\ \}? \\ \texttt{"children"}: & \{\ \mathit{string} : \mathit{wp.script}, \dots\ \}? \\ \texttt{"strategy"}: & \mathit{string}? \end{array} \right\}$$

The `header` field is just a descriptive title and has no special meaning. The `tactic` identifies the tactic, as it has been registered in the Frama-C/WP internal API. The `selection` is an opaque encoding of the target of the tactic inside the proof obligation structure. The `params` array stores the value of tactic parameters, if any. The `children` stores the proof scripts associated with each sub-goal generated by applying the specified tactic on a proof obligation. Finally, the `strategy` fields is used to recall which user-defined proof strategy was used to trigger the tactic (See Section 2.5).

Tactic Parameters Values of tactical parameters are encoded with the following JSON format, depending on the internal type of the parameter value:

$$
\begin{aligned}
\mathit{wp.param} ::= {} & \mathit{bool} \mid \mathit{number} \mid \mathit{string} \\
& \mid \mathit{wp.named} \mid \mathit{wp.selection} \\
\mathit{wp.named} ::= {} & \texttt{null} \mid \left\{ \begin{array}{rl} \text{"}\texttt{id}\text{"}: & \mathit{string}, \\ \text{"}\texttt{title}\text{"}: & \mathit{string}? \\ \text{"}\texttt{description}\text{"}: & \mathit{string}? \end{array} \right\}
\end{aligned}
$$

Named items (*wp.named*) corresponds, for instance, to searched lemmas. Selections corresponds to terms or expressions selected by the user *via* the graphical user interface, like the target of the tactic, as described below.

Tactic Selection The target of the tactic is identified by a complex structure encoding on which part of the proof obligation it shall be applied. Since the exact structure of the proof obligation may vary from time to time, this structure allows for searching in the proof obligation a *pattern* that resembles the original target that was originally selected by the user during an interactive session with the Frama-C/WP graphical user interface.

Such *patterns* are encoded as follows:

$$
\begin{aligned}
\mathit{wp.selection} ::= {} & \texttt{null} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{clause-goal}\text{"},\ \mathit{wp.pattern}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{inside-goal}\text{"},\ \mathit{wp.occur},\ \mathit{wp.pattern}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{clause-step}\text{"},\ \mathit{wp.at},\ \mathit{wp.kind},\ \mathit{wp.pattern}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{inside-step}\text{"},\ \mathit{wp.at},\ \mathit{wp.kind},\ \mathit{wp.occur},\ \mathit{wp.pattern}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{range}\text{"},\ \text{"}\texttt{min}\text{"} : \mathit{number},\ \text{"}\texttt{max}\text{"} : \mathit{number}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{kint}\text{"},\ \text{"}\texttt{val}\text{"} : \mathit{number}\ \} \\
& \mid \{\ \text{"}\texttt{select}\text{"} : \text{"}\texttt{compose}\text{"},\ \text{"}\texttt{id}\text{"} : \mathit{string},\ \text{"}\texttt{args}\text{"} : [\mathit{wp.selection}, \ldots]\ \} \\
\mathit{wp.pattern} ::= {} & \text{"}\texttt{pattern}\text{"} : \mathit{string} \\
\mathit{wp.occur} ::= {} & \text{"}\texttt{occur}\text{"} : \mathit{number} \\
\mathit{wp.at} ::= {} & \text{"}\texttt{at}\text{"} : \mathit{number} \\
\mathit{wp.kind} ::= {} & \text{"}\texttt{kind}\text{"} : \text{"}\texttt{have}\text{"}\mid\text{"}\texttt{type}\text{"}\mid\text{"}\texttt{init}\text{"}\mid\text{"}\texttt{branch}\text{"}\mid\text{"}\texttt{either}\text{"}\mid\text{"}\texttt{state}\text{"}
\end{aligned}
$$

The various components of selection patterns have the following meaning: *wp.pattern* encodes a term or predicate pattern; *wp.occur* designates which occurrence of the pattern to select, in case of there is many of them inside the selected goal or hypothesis; *wp.at* designates the order of the selected hypothesis inside the proof obligation (the pattern is searched around this value), and *wp.kind* its kind.

The *wp.pattern* is a simple string that encodes the head of the structure of the selected term or formulæ inside the designated goal or hypothesis. It is the concatenation of the first 32 head nodes of width-first traversal of the selected term, each node being represented by the following string:

- constant are represented by their value,
- free variables by their base-name prefixed with `$`,
- bound variables by their de-Bruijn index prefixed with `#`,
- true and false by `T` and `F`, and quantifiers by `\F` and `\E`,
- operators by `&`, `|`, `!`, `~`, `+`, `-`, `*`, `/`, `%<`, `>`, `=`, `>=`, `/`, `?`
- array operations by `[]`, `[.]` and `[=]`
- record operations by `.fd` and `{fd,...}`
- function calls by their name

For instance, the term $(x_1 \leq x_2 + 1)$ will be represented by the pattern `"<=$x+$x1"`.

Script Priority When several alternatives are available for discharging a proof obligation, the `script` and `tip` provers of Frama-C/WP choose which one to apply first according to the following heuristic:

1. try internal prover `"Qed"` with a `"valid"` verdict;
2. try any SMT prover with a `"valid"` verdict;
3. try the `"Coq"` proof assistant with a `"valid"` verdict;
4. try any Tactic alternative;
5. finally try the remaining Prover alternatives.

Inside the same level of priority, alternatives are kept in their original order.

2.7 Structure of JSON Reports

The generated JSON report format consists of an array of JSON records, one for each generated toplevel goal. Each record consists of the following fields, some of them being optional when indicated:

`"goal":<string>` goal identifier (same as in console report);

`"property":<string>` property identifier (see Frama-C/Report documentation);

`"file":<string>` source file location;

`"line":<number>` source line location;

`"axiomatic"?:<string>` axiomatic name (optional);

`"function"?:<string>` function name (optional);

`"behavior"?:<string>` behavior name (optional, non-default);

`"smoke":<boolean>` smoke-tests goal (**warning:** Cf. fields `"verdict"` and `"passed"` below);

`"verdict":<string>` prover returned status. Can be `"none"`, `"valid"`, `"failed"`, `"unknown"`, `"stepout"` or `"timeout"` (**warning:** Cf. field `"passed"` below);

`"passed":<boolean>` is `"true"` when the prover verdict is valid for non-smoke tests, or when the prover verdict is *not* valid for smoke tests;

`"script"?:<string>` script file, when available (optional);

`"tactics"?:<number>` number of tactics involved in the script (optional);

`"provers":[<prover>,...]` prover detailed output. Each entry of the array is a record that consists of the following fields:

- `"prover":<string>` prover name (with version, see option `-wp-prover`);
- `"time":<number>` prover time, in milliseconds. For `Qed` prover, includes also the total simplification time for other provers and tactics that contributes to the goal;
- `"success":<number>` number of valid sub-goals the prover contributes to.

`"subgoals"?:<number>` number of associated sub-goals (optional, when more than one);

`"proved"?:<number>` number of sub-goals proved (optional, when non-zero);

`"failed"?:<number>` number of sub-goals ending with prover failure or failed smoke-tests (optional, when non-zero);

`"timeout"?:<number>` number of sub-goals ending in timeout or step-out (except for smoke-tests) (optional, when non-zero);

`"unknown"?:<number>` number of sub-goals with unknown verdict (except for smoke-tests) (optional, when non-zero);

`"cached"?:<number>` number of cached sub-goals verdicts (including `Qed` proved sub-goals) (optional, when non-zero).

Notice that the JSON report file is printed in compact form without extra space nor indentation, unless verbose level is greater than 2 or debug level is greater than 1.

2.8 Structure of Report Templates

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 `"-"`.

`@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
`range`	range of maximal steps used by prover (more stable)
`success`	percentage of discharged PO

Remarks: `&ergo` is a shortcut for `&alt-ergo`. Formatters can be written `"%.."` or `"%{..}"`. When `range` is used instead of `steps`, the maximal number n of steps is printed as a range $a..b$ that contains n. When option `-wp-report-json` is used, the previous rank a and b are kept when available

and still fits with the new maximal step number. Otherwise, a and b are re-adjusted following an heuristics designed to increase the stability for non-regression testing.

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.

WP Models 3

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 $m_1 \dots m_k$ 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:

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

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:

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

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:

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

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

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

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

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

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 Mixed models hypotheses

For the previously presented Ref model, but also for the Typed, and Caveat models presented later, WP lists the separation and validity hypotheses associated to the choice that it make of dispatching each pointer and variable either to the `Hoare` or to the model $\mathcal{M}$ used for the heap.

Consequently, in addition to user-defined function `requires`, WP also assumes, and thus states as `requires` for the function caller, that:

- `Hoare` variables are separated from each other,
- `Hoare` variables are separated from the locations in $\mathcal{M}$,
- references are valid memory locations,
- locations assigned via a pointer are separated from `Hoare` variables whose address is not taken by the function (including via its contract).

Furthermore, the function must ensure that:

- locations assigned via a pointer (including the returned value when it is a pointer) are separated from `Hoare` variables whose address is not taken by the function (including via its contract),
- pointers assigned by the function (including the returned value when it is a pointer) are separated from function parameters and `Hoare` variables whose address is not taken by the function (including via its contract).

In order to precisely generate these hypotheses, WP needs precise `assigns` specification. In particular each function under verification and all its callees needs an `assigns` specification. Furthermore, if the function assigns or returns a pointer, WP needs a correct `\from` specification for those pointers. If the specification is incomplete, a warning `wp:pedantic-assigns` is triggered. Note that WP does not verify that the `\from` is correct.

The hypotheses are displayed when the option `-wp-warn-memory-model` is enable (it is enabled by default). They can be verified by WP using the experimental option `-wp-check-memory-model`.

3.7 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).

3.8 The Bytes Memory Model

The behavior of the Bytes memory model is essentially the same as the Typed memory model except that, while Typed dispatches the aliased memory according to the types of the values, the Bytes memory model has a single memory which is an array from pointer base to memory block, memory blocks being arrays from pointer offset to byte.

Modeling. When a value is written to the memory, it first encoded as a sequence of bytes. More precisely, unsigned integers are directly encoded into bytes via modulo and division. For other fundamental types, WP first translate them into an unsigned and then encode the obtained unsigned, the following applies for the translation to unsigned:

- signed integers are translated using (morally) an ACSL cast,
- for pointers, we use the physical translation axiomatized in WP,
- for float, we assume that there exists such a function without providing it.

The sequence of bytes that we obtained is then written bytes by bytes in memory.

When a value is read from the memory, the opposite applies: we read the bytes from the memory, build the unsigned value and then translate it back to the expected type.

This modeling is **super costly** for proof, since the read and writes involves the encoding + decoding of values. It makes formulas quite complex.

Mitigation. There are two main mitigations.

First, we provide a lot of lemmas about load/store operations so that when the code does not exhibit heterogeneous load/store, the SMT solvers do not have to reason about the encoding. Furthermore, there are also shortcuts for load/store when the only heterogeneous behavior is the presence of a sign or not.

Second, we use the Hoare memory model when there is no alias. It avoids modeling bytes for every single variable of the program.

Limitations. The model has three main limitations currently:

- it assumes little-endian,
- it cannot really deal with low-level operations for pointers and floats, because WP assumes that a conversion function exists (which makes sense) but does not exhibit this function,
- in default mode (+ Hoare), if a union is not aliased it is **not** sent into the Bytes model but into the Hoare one that does not take in account heterogeneous operations. One can use `+raw` to use the Bytes memory model everywhere, but be prepared to the fact that properties will be extremely hard to prove.

3.9 The Region Memory Model

The Region memory model dispatches each pointer to a different memory model. For instance, an `int*` pointer that is always accessed through `int*` accesses (with possible aliases) need only to rely on the Typed memory model. On the contrary, a value containing an union whose fields are accessed with heterogeneous types shall be handled using the Bytes memory model.

Usage. To activate this model, simply use '`-wp-model Region`'. It will automatically launch the plugin Region, even without the usage of '`-region`'.

Addresses. Each address in the Region memory model is a pair of the representation of its physical address (computed by the Bytes memory model) and the region in which it is included (over-approximation).

Plugin Region. The plugin Region statically computes for each function a static region map that associates a node (or region) to expressions of pointers. Each node of the map represents a set of pointer's addresses and includes the computation of an over-approximation of the list of access types (e.g. write 32bits int, read 64bits float, cast 16bits int, etc...).

Bytes memory model & Region plugin. The Region memory model uses the region map produced by the Region plugin to statically associate each pointer to a region. Then, for each pointer whose region's accesses are not heterogeneous, the Region memory model behaves the same way as the Typed memory model ; whereas for all regions with heterogeneous accesses, the Region memory model sends off all its pointers to the Bytes memory model.

Limitations. The Region memory model is provided with the following current limitations:

- Function calls are not yet implemented in the Region plugin.
- A structure copy might be rejected by the model if the structure contains fields with pointers.
- Logic is not yet included and only C code is currently analyzed by the Region plugin. Accesses (read, write, and cast) happening only inside ACSL annotations are not included in the analysis. As a consequence, developers and formalists shall add (possibly ghost) C code for field accesses that only appear inside ACSL annotations.

3.10 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*. Note that it means that the pointer is considered valid. If one needs to accept `NULL` for this pointer, a `wp_nullable` ghost annotation or the clause `wp_nullage_args` can be used to bring this information to WP:

File **nullable.c**

```
void foo(int* p __attribute__((wp_nullable)),
         int* q __attribute__((wp_nullable))){

}

// or equivalently:
//@ \wp::nullable_args p, q ;
void foo(int* p, int* q){

}
```

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 warns about the implicit separation hypotheses required by the memory model *via* the `-wp-warn-memory-model` option, set by default.

WP Simplifier 4

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.

4.1 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.

4.2 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.

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

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. Chapter 5 provides additional information on the implemented simplifications and the supported ACSL built-in operations.

4.3 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.

Builtins 5

This chapter provides additional information on the supported ACSL built-in symbols and, more generally, on the simplification rules implemented in the Qed simplifier, as introduced in Chapter 4.

5.1 General Simplifications

Arithmetics operators $(+, \times, \dots)$ are normalized with respect to their associativity and commutativity rules. Neutral and absorbent operands are also taken into account, and operations on numerical constants are propagated as well. Moreover, linear forms are strongly normalized. Hence, it is not possible to write term $(1 + x) - 1 + x$ since it will be automatically reduced into $2x$.

Comparison operators are also simplified. First, linear normalization is performed on both sides of the comparison, with negative factors or constants moved on the opposite side. Finally, on integer comparison, an off-by-one normalization is applied to transform, *eg.* $1 < x$ into $0 \leq x$ (on integers only).

Comparison between injective functions are simplified in the expected way. For instance, $f(x, y) = f(x', z')$ will be rewritten into $x = x' \wedge y = y'$ when f is injective, and similarly with dis-equalities.

Idempotent operators f are normalized with $f(x, x) = x$. Associative and commutative operators are normalized as usual (terms are sorted structurally).

5.2 ACSL Built-ins

The ACSL built-ins supported by Frama-C/WP are listed in Figure 5.1. In the table, the reference implementation of the function is given in terms of the Why-3 standard library[1], from which all the properties are automatically imported and communicated to the provers.

Below in the section, additional information is given for each supported builtin, with a brief description of the implemented Qed simplifiers.

[1] Available online `http://why3.lri.fr/stdlib`

ACSL	Why-3 Reference	Qed
(real) x	real.FromInt	p.67
\ceil, \floor	real.Truncate	p.67
\abs	int.Abs, real.Abs	p.67
\min, \max	int.MinMax, real.MinMax	p.67
\sqrt	real.Square	p.67
\exp, \log, \log10	real.ExpLog	p.68
\pow	real.PowerReal	p.68
\sin, \cos, \tan	real.Trigonometry	p.68
\asin, \acos, \atan	real.Trigonometry	p.68
\sinh, \cosh, \tanh	real.Hyperbolic	p.68
\atan2, \hypot	real.Polar	p.68

Figure 5.1: Supported ACSL builtins

(real) x for the conversion from integers to reals. Constants, including `float` and `double` literals, are exactly converted to rationals. This operation is also declared to be injective.

\ceil(x), \floor(x) for the conversion from reals to integers. Constants (rationals) are truncated towards the expected direction. Other simplifications are:

$$\begin{aligned} \texttt{\ceil}((\texttt{real})\, n) &= n \quad (\forall n \in \mathbb{Z}) \\ \texttt{\floor}((\texttt{real})\, n) &= n \quad (\forall n \in \mathbb{Z}) \end{aligned}$$

Other conversion simplifications will be performed when using conversions to C-integer types. Actually, the truncation operation defined in `real.Truncate` from Why-3 is used for these other conversions and combine well together.

\abs(x) for both integers and reals. The implemented simplifiers are:

$$\begin{aligned} \texttt{\abs}(x) &\geq 0 \\ \texttt{\abs}(k.x) &= |k|.\texttt{\abs}(x) \\ \texttt{\abs}(x) &= x && \texttt{when } 0 \leq x \\ \texttt{\abs}(x) &= -x && \texttt{when } x < 0 \\ \texttt{\abs}(x) &= 0 && \texttt{iff } x = 0 \end{aligned}$$

\min(x, y), \max(x, y) are implemented on both real and integers as idempotent, associative and commutative operators.

\sqrt(x) is declared to be injective, monotonically increasing and is associated with the following simplifiers:

$$\begin{aligned} \texttt{\sqrt}(0) &= 0 \\ \texttt{\sqrt}(1) &= 1 \\ \texttt{\sqrt}(x) &\geq 0 \\ \texttt{\sqrt}(x) &= 0 && \texttt{iff } x = 0 \\ \texttt{\sqrt}(x \times x) &= \texttt{\abs}(x) \end{aligned}$$

Moreover, the following lemmas are added to complement the reference ones:

$$\begin{array}{rcll} \texttt{\sqrt}(x) & < & x & \text{when } 1 < x \\ \texttt{\sqrt}(x) & > & x & \text{when } 0 \leq x < 1 \end{array}$$

\exp(x), \log(x) and \log10(x) are declared to be injective, monotonically increasing and are associated with the following simplifiers:

$$\begin{array}{rcll} \texttt{\log}(1) & = & 0 & \\ \texttt{\exp}(0) & = & 1 & \\ \texttt{\exp}(x) & > & 0 & \\ \texttt{\exp}(\texttt{\log}(x)) & = & x & \text{when } 0 < x \\ \texttt{\log}(\texttt{\exp}(x)) & = & x & \end{array}$$

However, we found necessary to complement the reference definitions with the following lemma, especially when \tanh is involved:

$$\texttt{\exp}(x) \geq 0$$

\pow(x, n) is associated with the following simplifiers:

$$\begin{array}{rcl} \texttt{\pow}(x, 0) & = & 1 \\ \texttt{\pow}(x, 1) & = & x \end{array}$$

Other algebraic simplifications are limited to cases where $x > 0$, and there is no obvious choice of normalization when combining \pow with \exp and \log.

$$\texttt{\log}(\texttt{\pow}(x, n)) \quad = \quad n.\texttt{\log}(x) \quad \text{when } 0 < x$$

However, algebraic rules for such combinations are well established in the reference lemmas imported from Why-3.

\sin(x), \cos(x) and \tan(x) trigonometric operations. Useful lemmas are already defined in the reference implementation from Why-3 library.

\asin(x), \acos(x) and \atan(x) trigonometric operations. Except definition of \atan which is available from reference, definition for arc-\sin and arc-\cos have been added. Moreover, all the arc-trigonometric operations are declared to be injective. In addition, the following simplifiers are also registered:

$$\begin{array}{rcll} \texttt{\sin}(\texttt{\asin}(x)) & = & x & \text{when } -1 \leq x \leq 1 \\ \texttt{\cos}(\texttt{\acos}(x)) & = & x & \text{when } -1 \leq x \leq 1 \\ \texttt{\tan}(\texttt{\atan}(x)) & = & x & \end{array}$$

Notice that definitions for \asin and \acos are not (yet) available from the original Why-3 reference and are actually provided by custom extensions installed in the WP shared directory.

\sinh(x), \cosh(x) and \tanh(x) trigonometric operations. Useful lemmas are already defined in the reference implementation from Why-3 library.

`\atan2`(x, y) **and** `\hypot`(x, y) for dealing with polar coordinates. Definitions imported from the reference implementation of Why-3 library.

`\le_float`(x, y), `\ge_float`(x, y), `\lt_float`(x, y), `\gt_float`(x, y), `\eq_float`(x, y), `\ne_float`(x, y), `\le_double`(x, y), `\ge_double`(x, y), `\lt_double`(x, y), `\gt_double`(x, y), `\eq_double`(x, y), **and** `\ne_double`(x, y) for dealing with floating point comparisons. They are similar to comparisons over the real numbers if both x and y are finite, but obey IEEE semantics for infinities and NaNs

5.3 Custom Extensions

DEPRECATED. You shall import a Why3 module instead of using custom drivers. See Section 2.4.12 for details.

As explained in Section 2.4.11, it is possible to extend all the properties mentioned above. Section 2.3.11 also provides hints on how to define tactics that can be used in the interactive prover of Frama-C/WP.

However, Why-3 offers the easiest way of adding new theorems for interactive proving. Since all the builtin symbols of ACSL are actually linked to the standard library of Why-3, any user theory also referring to the standard symbols can be added to the proof environment by using option `-wp-why-lib 'file...'`.

Otherwise, a driver can be written. For instance, the additional lemma regarding `\exp` p.68 for Why3 is simply defined in the following way:

```
// MyExp.mlw file
axiom exp_pos : forall x:real. (0.0 < exp(x))
```

Of course, this piece of Why3 input file must be integrated after the symbol `exp` has been defined. The corresponding driver is then:

```
// MyExp.driver file
library exponential:
why3.file += "MyExp.mlw" ;
```

Such a driver, once loaded with option `-wp-driver`, instructs Frama-C/WP to append your extension file to the other necessary resources for library `"exponential"`, to which the logic ACSL builtin `\exp` belongs to.

The file `share/wp/wp.driver` located into the shared directory of Frama-C, and the associated files in sub-directory `share/wp/why3` shall provide all the necessary hints for extending the capabilities of Frama-C/WP in a similar way.

Bibliography

[BBB+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022. Proceedings*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[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.

[CCK+] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*. CEA List, Software Reliability Laboratory. https://www.frama-c.com/download/frama-c-user-manual.pdf.

[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.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[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.