frama c

Software Analyzers

WP 0.8

WP Plug-in Manual

Version 0.8 for Neon-20140301

Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye

CEA LIST, Software Safety Laboratory
February 2014

Contents

Chapter 1

Introduction

This document describes a Frama-C 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 performances, 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 Jessie plug-in, which relies on a separation memory model in the spirit of Burstall's work [Bur72]. The Jessie memory model is very efficient for a large variety of well structured C-programs. However, it does not apply when low-level memory manipulations, such as heterogeneous casts, are involved. Moreover, Jessie operates by compiling the C program to Why, a solution that prevents the user from combining *weakest precondition calculus* with other techniques, such as the Value analysis plug-in.

The WP plug-in has been designed with cooperation in mind. That is, you may use WP for proving some annotations of your C programs, and prove other ones with other plug-ins. The recent improvements of the Frama-C kernel are then responsible for managing such partial proofs and consolidate 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. Finally, we present in Chapter 4 the simplifier module and the efficient weakest precondition engine implemented in WP plug-in.

1.1   Installation

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

1.2   Tutorial

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

File **swap.c**

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

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

File **swap1.h**

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

You can run **wp** on this example with:

```
# frama-c -wp swap.c swap1.h
[kernel] preprocessing with "gcc -C -E -I.  swap.c"
[kernel] preprocessing with "gcc -C -E -I.  swap1.h"
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] [Alt-Ergo] Goal typed_swap_post_A : Valid (10)
[wp] [Qed] Goal typed_swap_post_B : Valid
[wp] Proved goals:    2 / 2
    Qed:            1
    Alt-Ergo:       1  (10)
```

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.

The **WP** plug-in does not generate proof obligation to prevent your program from raising a runtime error, because this property may be validated with any other techniques, for instance by running the *value analysis* plug-in or the *rte generation* one.

---

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

Hence, consider 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 generation* plug-in for you. Now, WP reports that the function `swap` fulfills its contract:

```
# frama-c -wp -wp-rte swap.c swap2.h
[kernel] preprocessing with "gcc -C -E -I.  swap.c"
[kernel] preprocessing with "gcc -C -E -I.  swap2.h"
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[rte] annotating function swap
[wp] 9 goals scheduled
[wp] [Alt-Ergo] Goal typed_swap_post_A : Valid (15)
[wp] [Qed] Goal typed_swap_post_B : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access : Valid (17)
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_2 : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access_3 : Valid (17)
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_4 : Valid
[wp] [Qed] Goal typed_swap_assign_part1 : Valid
[wp] [Alt-Ergo] Goal typed_swap_assign_part2 : Valid (22)
[wp] [Alt-Ergo] Goal typed_swap_assign_part3 : Valid (22)
[wp] Proved goals:    9 / 9
     Qed:             4
     Alt-Ergo:        5  (22)
```

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
[kernel] preprocessing with "gcc -C -E -I.  swap.c"
[kernel] preprocessing with "gcc -C -E -I.  swap2.h"
[rte] annotating function swap
[report] Computing properties status...

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

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

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

1.3   Weakest Preconditions

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

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

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

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:

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

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 $P \implies W$ 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 error.

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

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

1.4   Memory Models

The essence of a *weakest precondition calculus* is to translate code annotation 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 to deal 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` can not be translated by a simple substitution of X in Q.

This is where *memory models* comes 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 can not be represented in this model, and expressions such as `*p` can not be translated at all. You can still represent pointer values, but you can not 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 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 cast of pointers can not be translated in this model. For instance within this memory model, you can not 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 enables limited forms of casts. See chapter 3 for details.

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

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

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

Chapter 3 is dedicated to a more detailed description of memory models, and how the WP plug-in use and *combine* them to generate efficient proof obligations.

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

1.5 Arithmetics 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 rely on several arithmetic models:

Natural Model: integer operations are performed on mathematical integers. Conversions between different integer types are still translated with *modulo*, though. Generated proof obligations with this model are well supported by state-of-the-art provers. But the C operations must never overflow for the proof obligations to be correct.

This can be verified aside by running RTE plug-in with specific options, or with the `-wp-rte` option.

Machine Integer Model: integer operations are *always* performed with *modulo*. This model removes the constraint of generating the guards against overflows. But the generated proof obligations are generally difficult to discharge.

Real Model: floating-point operations are *transformed* on real, 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.

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

By default, the WP plug-in uses the Natural model for integers and the (unsound) Real model for floating-points.

Chapter 2

Using WP Plug-in

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

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

This chapter describes how to use the plug-in, from the Frama-C graphical user interface (section 2.2), from the command line (section 2.3), or from another plug-in (section 2.4). Additionally, the combination of the WP plug-in with the load and save commands of Frama-C and/or the -then command-line option is explained in section 2.6.

2.1 Installing Provers

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

Prover	Version	Download	
Alt-Ergo	0.95.2+	http://alt-ergo.ocamlpro.com	[CCK06]
Coq	8.4.pl[12]	http://coq.inria.fr	[Coq10]
Why3	0.81+	http://why3.lri.fr	[BFMP11]

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

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

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

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

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

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

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

2.2 Graphical User Interface

To use WP plug-in under 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 2.3:

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

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

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

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

The options from the WP side panel correspond to some options of the plug-in command-line. Please refer to section 2.3 for more details. In the graphical user interface, there are also specific panels that display more details related to WP plug-in, that we shortly describe below.

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

Icons for properties:

	No proof attempted.
	The property has not been validated.
	The property is *valid* but has dependencies.
	The property and *all* its dependencies are *valid*.

Proof Obligations Panel. This panel is dedicated to the WP plug-in. It shows the generated proof obligations and their status for each prover. By double-clicking an annotation, you can view its mathematical definition in a human readable format. By clicking on a prover column, you can also submit a proof obligation to a prover by hand. Right-click provides more option depending on the prover, such as proof-script editing for Coq.

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.

Property Dependency Graph. By double-clicking on the status column of a property in the properties panel, you can display a dependency graph for this property. The graph displays the property, its status, which plug-in has participated in the proof, and on which properties the proof directly depends on.
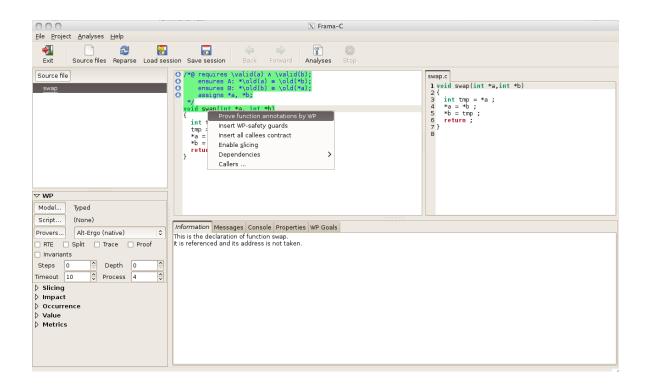
17

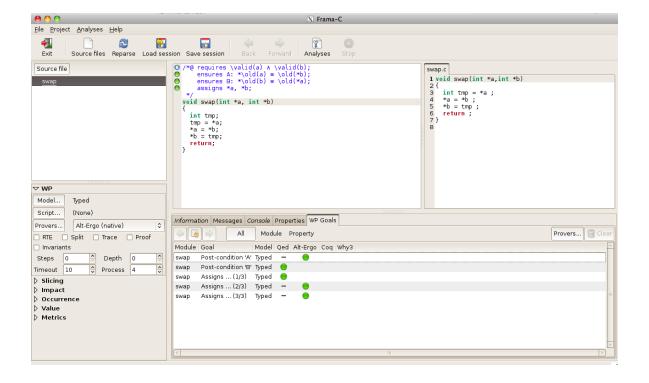Figure 2.1: WP in the Frama-C GUI

Figure 2.2: WP run from the GUI

2.3 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 hypothesis 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 pass on the project. See section 2.6 for details.

2.3.1 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 <f_1,...,f_n>` selects annotations of functions f_1,...,f_n (defaults to all functions).

`-wp-skip-fct <f_1,...,f_n>` remove annotations of functions f_1,...,f_n (defaults to none).

`-wp-bhv <b_1,...,b_n>` selects annotation for behaviors b_1,...b_n (defaults to all behaviors) of the selected functions.

`-wp-prop <p_1,...,p_n>` 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`, `@invariant`, `@variant`, `@breaks`, `@continues`, `@returns`, `@complete_behaviors`, `@disjoint_behaviors`.
Properties can be prefixed with a minus sign to *skip* the associated annotations. For example `-wp-prop="-@assigns"` removes all `assigns` and `loop assigns` properties from the selection.

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

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

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

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

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

2.3.2   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.

2.3.3   Model Selection

These options modify the underlying memory model that is used for computing weakest preconditions. See chapter 3 for details.

`-wp-model <spec...>` specify 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, potentially reverting previous selectors. The available selectors are:

Selector	Description
Hoare	Select Hoare memory model.
Typed	Select Typed memory model with limited casts.
cast	Select Typed memory model with unlimited casts (unsound).
nocast	Select Typed memory model with *no* casts.
raw	Disable the combination of memory models.
var	Combination of memory models based on variable analysis.
ref	Activate the detection of pointer variables used for reference passing style.
nat	Use natural integer model (no overflows).
int	Use machine integers (modulo arithmetics).
cint	alias for int.
real	Use mathematical reals instead of floating-points.
float	Use floating-point operations.
cfloat	alias for float.

Default settings corresponds to `-wp-model "Typed,var,nat,real"`.

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

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

2.3.4 Computation Strategy

These options modifies 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 with the following options: `-rte-mem`, `-rte-div`, `-rte-signed` and `-rte-unsigned-ov`. The generated guards, when proved[1], fulfill the requirements for using the WP plug-in with natural integer domain (default is: `no`).

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

`-wp-(no)-invariants` computes proof obligations for arbitrary invariants inside loops. Also modifies the calculus for proper loop invariants[2] (default is: `no`).

2.3.5 Trigger Generation

The ACSL language does not provide user with 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.

---

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

[2] To be efficient, it is better to put all the loop invariants inside only one annotation. Otherwise, Frama-C insert them at different program points. Then, the WP calculus cuts the generated proof obligations at each invariant instead of proving all of them inside the same induction scheme. Notice that, when using the ACSL-Importer plug-in, all the loop invariants are placed at one unique program point, and are treated efficiently by WP plug-in.

2.3.6 Qed Simplifier Engine

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

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

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

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

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

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

2.3.7 Decision Procedures Interface

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

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

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

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

`-wp-gen` only generates proof obligations, do 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. Defaults is
4 processes. With `-wp-par 1`, the order of logged results is fixed. With more processes,
the order is runtime dependent.

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

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

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

Alt-Ergo. Direct support for the Alt-Ergo prover is provided. You need at least version `0.95` of the prover. It is also the default selected prover.

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

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

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

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

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

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

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

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

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

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

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

`-wp-tactic <ltac>` specifies the Coq tactic to try with when no user-script is found. The default tactical is `"auto with zarith"`. See also how to load external libraries and user-defined tactics in section 2.3.9.

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

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

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

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

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

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

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

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

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

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

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

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

2.3.8 Generated Proof Obligations

Your proof obligations are generated and saved to several text files. With the `-wp-out` option, you can specify a directory of your own where all these files are generated. By default, this output directory is determined as follows: under 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 at Frama-C exit.

The other options controlling the output of generated proof obligations are:

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

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

2.3.9 Additional Proof Libraries

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

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

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

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

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

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

2.3.10 Linking ACSL Symbols to External Libraries

In addition to additional proof libraries, it is also possible to *link* declared ACSL symbols to external or predefined symbols. In such a 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, see section 2.3.9.

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

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

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

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

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

C-Comments are allowed in the file. For overloaded ACSL symbols, it is necessary to provide one "*link*" symbol for each existing signature. The same "*link*" symbol is used for all provers, and must be defined in the specified libraries, or in the external ones (see 2.3.9).

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

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

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

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

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

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

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

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

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

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

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

Optional *property-tags* can be given to `logic "link"` symbols to allow the WP to performs additional simplifications (See section 4). Tags consists of an identifier with column (':'), sometimes followed by a link ('"*link*";'). The available tags are depicted on figure 2.3.

Tags	Operator Properties
`commutative:`	specify a commutative symbol: $x \odot y = y \odot x$
`associative:`	specify an associative symbol: $(x \odot y) \odot z = x \odot (y \odot z)$
`ac:`	shortcut for `associative: commutative:`
`left:`	balance the operator on left during export to solvers (requires the associative tag): $x \odot y \odot z = (x \odot y) \odot z$
`right:`	balance the operator on right during export to solvers (requires the associative tag): $x \odot y \odot z = x \odot (y \odot z)$
`absorbant: "`*a-link*`":`	specify "*a-link*" as being the absorbant element of the symbol: "*a-link*" $\odot\, x =$ "*a-link*" $x \odot$ "*a-link*" $=$ "*a-link*"
`neutral: "`*e-link*`":`	specify "*e-link*" as being the neutral element of the symbol: "*e-link*" $\odot\, x = x$ $x \odot$ "*e-link*" $= x$
`inversible:`	specify simplification relying on the existence of an inverse: $x \odot y = x \odot z \iff y = z$ $y \odot x = z \odot x \iff y = z$
`idempotent:`	specify an idempotent symbol: $x \odot x = x$
`injective:`	specify an injective function: $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \implies \forall i\; x_i = y_i$
`constructor:`	specify an injective function, that construct 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 2.3: Driver Property Tags

2.4 Plug-in Developer Interface

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

All the entry points are documented in the `Frama-C html` documentation, in the "Wp API" page of the plug-in section.

2.5 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 `Frama-C` kernel from various analyzers. We only discuss *WP reports* in this section.

Reports are generated with the following command-line options:

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

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

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

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

Configuration section consists of optional commands, one per line, among:

`@CONSOLE` the report is printed on 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

---

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

`@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 contents can be specified.

Textual contents use special formatters that will be replaced by actual statistics 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 finally proved properties in section
`%prop:total`	number of covered properties
`%prop:valid`	number of finally proved properties
`%prop:failed`	number of remaining unproved properties
`%<`*prover*`>`	discharged PO by *prover*
`%<`*prover*`>:<`*stat*`>`	statistics for *prover* in section

Provers	
(*<prover>*)	A prover name (see `-wp-prover`)

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

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

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

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

2.6 Plug-in Persistent Data

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

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

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

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

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

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

Chapter 3

WP Models

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

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

The interest of developing several memory models is to manage the trade-off between the precision of the heap's values representation and the difficulty of discharging the generated proof obligations by external decision procedures. If you chose 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 be hardly 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 consists 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:

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

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

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

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

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

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

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

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

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 previous section. But we can do even better. Remark that none of the pointer values contained in variables `a` and `b` is stored in memory. The only occurrences of these pointer values are in expressions `*a` and `*b`. Thus, there can be no alias with these pointer values elsewhere in memory, provided they are not aliased initially.

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

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

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

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

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

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

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

3.6   The Typed Memory Model

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

Addresses.   We now use native records of $\mathcal{L}$ and provers to encode addresses as pairs of base and offset (integers). This simplify greatly 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).

Chapter 4

WP Simplifier

The logical language $\mathcal{L}$ used to build proof obligations is now equipped with build-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 simplifier and how to use it with WP plug-in. It also present 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.

The maximal sub-term sharing are 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 can turn a proof obligation to be trivialy 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 access and updates.

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

4.2 Simplifier Engine (Qed)

Build on top of our normalizing logic language $\mathcal{L}$, we have a simplifier engine named Qed. The simplifier engine is used by WP plug-in to simplify the generated proof contexts and proof obligations. The basic feature of Qed is to manage a base of knowledge 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 for basic laws involving the simplifier engine. Each law is applied with respect to a local base of knowledge 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. An 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 build 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 for 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 this extra cost to be valuable.

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.

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 is duplicated inside W_A and W_B. Moreover, this common post conditions 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 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 to 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 variables 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 post-condition 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 into the number of program statements.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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