frama C

Software Analyzers

WP 0.6 (Draft Manual)

WP Plug-in (Draft) Manual

Version 0.6 for Oxygen-20120901

Patrick Baudin, Loïc Correnson, Zaynah Dargaye

CEA LIST, Software Safety Laboratory

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

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 an experimental variant of our *weakest precondition* calculus, dedicated to proof simplification.

1.1 Installation

The WP plug-in is distributed with the Frama-C platform. However, you must install at least an external prover in order to fulfill proof obligations. You have several choices, see section 2.2.5 for details. To begin with, you may install the Alt-Ergo [CCK06] prover. You can install it from source at `http://alt-ergo.lri.fr` or with `Godi`.

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.c**

```
/*@ 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.c
[kernel] preprocessing with "gcc -C -E -I.  swap.c"
[kernel] preprocessing with "gcc -C -E -I.  swap1.c"
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[wp] warning: Missing RTE guards
[wp] 2 goals scheduled
[wp] [Alt-Ergo] Goal store_swap_post_A : Valid
[wp] [Alt-Ergo] Goal store_swap_post_B : Valid
```

As expected, Alt-Ergo discharged the two proof obligations generated by WP for the `swap` contract. 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 technique, for instance by running the *value analysis* plug-in or the *rte generation* one.

Hence, consider the following new contract for `swap`:

File **swap2.c**

```
/*@ 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.c
[kernel] preprocessing with "gcc -C -E -I.  swap.c"
[kernel] preprocessing with "gcc -C -E -I.  swap2.c"
[wp] Running WP plugin...
[wp] Collecting axiomatic usage
[rte] annotating function swap
[wp] [WP:simplified] Goal store_swap_assign : Valid
[wp] 6 goals scheduled
[wp] [Alt-Ergo] Goal store_swap_assert_rte : Valid
[wp] [Alt-Ergo] Goal store_swap_assert_rte_2 : Valid
[wp] [Alt-Ergo] Goal store_swap_assert_rte_3 : Valid
[wp] [Alt-Ergo] Goal store_swap_assert_rte_4 : Valid
[wp] [Alt-Ergo] Goal store_swap_post_A : Valid
[wp] [Alt-Ergo] Goal store_swap_post_B : Valid
```

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.c"
[rte] annotating function swap
[report] Computing properties status...

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

[  Valid  ] Post-condition 'A'
              by WP-Store.
[  Valid  ] Post-condition 'B'
              by WP-Store.
[  Valid  ] Assigns (file swap2.c, line 4)
              by WP-Store.
[  Valid  ] Assertion 'rte' (generated)
              by WP-Store.
[  Valid  ] Assertion 'rte' (generated)
              by WP-Store.
[  Valid  ] Assertion 'rte' (generated)
              by WP-Store.
[  Valid  ] Assertion 'rte' (generated)
              by WP-Store.
[  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:

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

Consider now $W = wp(f, Q)$, we have by definition of *wp*:

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

Suppose now that we can *prove* that P entails W: we can use the intermediate result of the weakest precondition calculus to prove the function contracts. This operation 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 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. Moreover, the only integer model currently implemented assumes no integer overflow at all (signed or unsigned). 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.

Provers. The WP plug-in computes proof obligations for post-conditions and assertions in C functions, and submits them to external provers.

You may discharge the generated proof obligation with automated decision procedures or an interactive proof assistant. Technically, WP is interfaced with Alt-Ergo [CCK06], Coq [Coq10], and any decision procedure supported by Why [Fil03].

1.4 Memory Models

The essence of a *weakest precondition calculus* is to translate code annotation into mathematical properties. Consider the simple case of an annotation referring to a non-pointer 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.

`Store` **model.** The default model for WP plug-in. Heap values are stored in a global array. Pointer values are translated into an index into this array.

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 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 an `int` value in memory.

`Runtime` **model.** This is a low-level memory model, where the heap is represented as a wide array of bits. 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, `Runtime` is very expressive but generates difficult proof obligations, and `Store` offers an intermediate solution.

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] or interactive proof assistant like Coq [Coq10].

This chapter describes how to use the plug-in, from the Frama-C graphical user interface (section 2.1), from the command line (section 2.2), or from another plug-in (section 2.3). 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 2.4.

2.1 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 2.2:

```
$ 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 2.2 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.

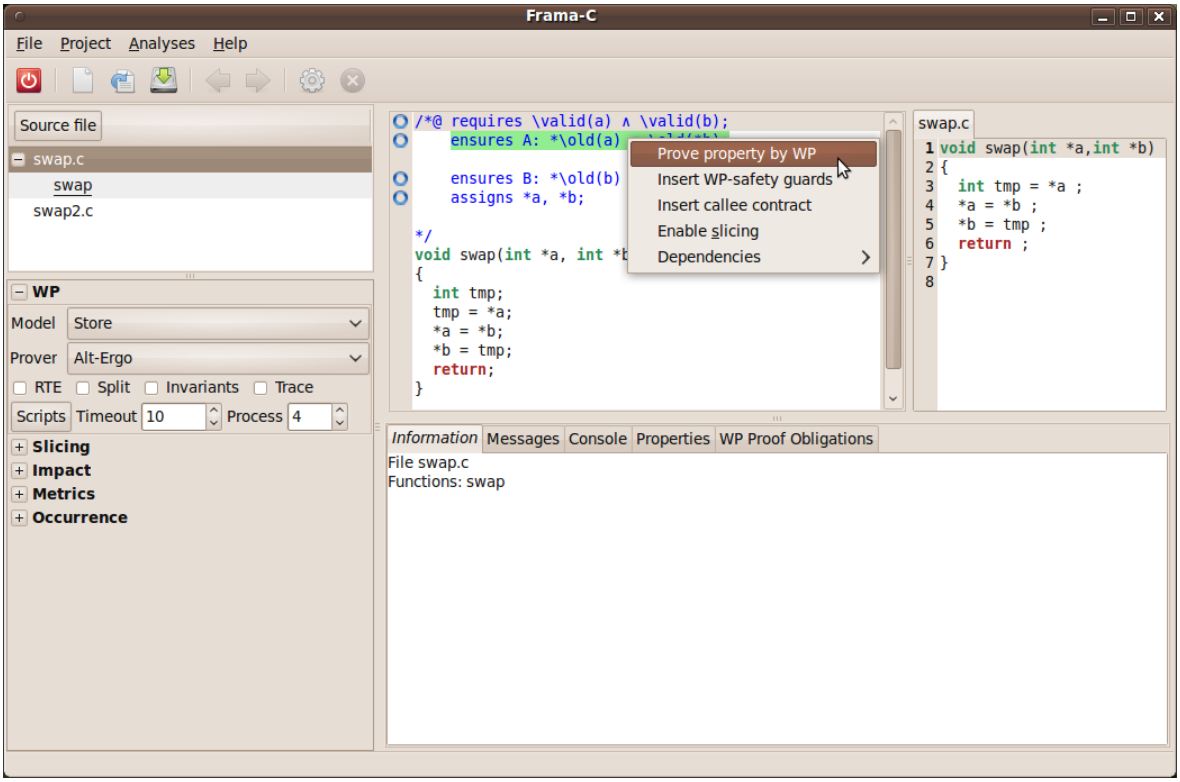

Figure 2.1: WP in the Frama-C GUI

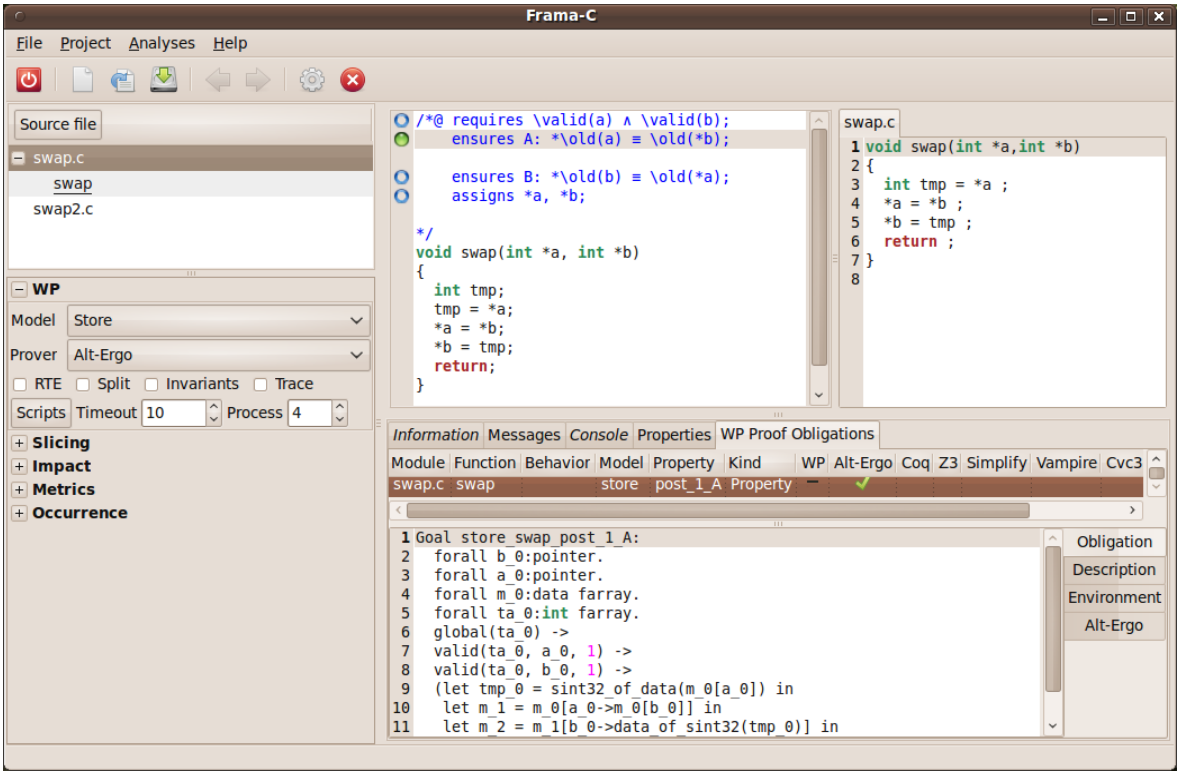

Figure 2.2: WP run from the GUI

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*.

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.

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.

2.2 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 2.4 for details.

2.2.1 Goal Selection

This group of options refines the selection of annotations for which proof obligations are generated. By default, all annotations are selected. By default, a property which is already proved – by WP plug-in or any other – does not lead to any proof-obligation generation.

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

`-wp-fct <f1,...,fn>` selects annotations of functions `f1`,...,`fn` (defaults to all functions).

`-wp-bhv <b1,...,bn>` selects annotation for behaviors `b1`,...`bn` (defaults to all behaviors) of the selected functions.

`-wp-prop <p1,...,pn>` selects properties having `p1` or ...`pn` as tagname (defaults to all properties). You may also replace a tagname by a `@<category>` of properties.
Recognized categories are: `@requires`, `@assigns`, `@ensures`, `@exits`, `@assert`, `@invariant`, `@variant`, `@breaks`, `@continues`, `@returns`, `@complete_behaviors`, `@disjoint_behaviors`.
Starts by a minus character to remove properties or tags from the selection.
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.

2.2.2 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.

2.2.3 Model Selection

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

`-wp-model <m>` sets the memory model among `Hoare`, `Store` (default memory model) or `Runtime`. For more information about the models and how to choose it, see section 1.4.

`-wp-(no)-logicvar` activates optimization for variables whose address is never taken, for which WP calculus uses the Hoare model (default is: `yes`).

`-wp-(no)-byreference` activates detection of arguments passed by reference (as pointers), for which WP calculus may also use the Hoare model under some hypotheses (default is: `no`).

`-wp-assigns <m>` sets the method for proving `assigns` clauses. Possible methods are:

`effect:` Each statement with side-effect produces one sub-goal. The locations written by each statement are checked to be included in the assigns clause. This gives a result stronger than required, but the proof obligations are generally simple and sufficient in practice. So, this is the default method.

`memory:` use the ACSL definition of `assigns` clauses, where memory states are compared before and after the considered block. Generates much more complex proof obligations than `effect`.

`none:` skip proof of `assigns` clauses.

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

2.2.4 Computation Strategy

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

`-wp-(no)-invariants` computes proof obligations for arbitrary invariants inside loops. Also modifies the calculus for proper loop invariants. This option automatically turns splitting on (see `-wp-split`)[1] (default is: `no`).

`-wp-huge <s>` cuts off proof terms with size exceeding 2^s (default size: 2^{30}). The size of a term is linearly related to its size on the disk, and to the size of proof obligation sent to decision procedures.

[1] 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 scheeme.

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.

`-wp-norm <m>` sets the normalization method applied to let-bindings in obligations generated for Alt-Ergo and Coq:

`Eqs:` let-bindings are replaced by universally-quantified fresh variables with the associated defining equalities in hypothesis (default method).

`Let:` let-bindings are preserved.

`Exp:` let-bindings are expanded.

`Cc:` let-bindings are replaced by function call or predicates by closure conversion.

`-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[2], fulfill the requirements for using the WP plug-in (default is: `no`).

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

`-wp-(no)-split` conjunctions in generated proof obligations are recursively split into subgoals. The generated goal names are suffixed by "`part`n". Notice that this option is set by default for `assigns` clauses when using the `effect` assigns method (see `-wp-assigns` above). Otherwise the option defaults to `no`.

`-wp-split-dim <d>` limits the number of generated sub-goals for assigns goals when using `-wp-split`. The number of generated sub-goals will not exceed 2^d proof obligations. Default is 2^6.

2.2.5 Decision Procedures Interface

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

`-wp-check <dp>` only checks the syntax of generated proof obligations for a family of decision procedures. Possible values of `dp` are: `alt-ergo`, `coq` and `why`.

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

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

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

`-wp-(no)-trace` keeps user labels in generated proof obligations. This option can be useful for tracing where the proof obligations come from, especially when using `-wp-split` option or interactive proof assistants (default is: `no`).

Alt-Ergo Direct support for the Alt-Ergo prover is provided. You need at least version 0.94 of the prover. It is also the default selected prover.

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

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

When using the new experimenal Typed model (see chapter 4), the following options are available with Alt-Ergo:

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

`-wp-proof-trace` prints Alt-Ergo proof trace.

`-wp-unsat-model` prints Alt-Ergo models for undischarged proof obligations.

Coq. Direct support for the Coq proof assistant is provided. The generated proof obligations are accepted by Coq version 8.3 but should also work with prior versions of the proof assistant. When working with Coq, you will enter interactive session, then save the proof scripts in order to replay them in batch mode.

`-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 nnot 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 2.2.7.

`-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>` Set the maximal number of suggested proof scripts.

`-wp-proof coq` only runs `coqc` on proof scripts found in the script file. If the generated goal (or the default one) is not correctly typed-checked by `coqc`, the `coq` prover fails to discharge the proof obligation.

`-wp-proof 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. During this session, several files are opened for you:

`<goal>.v` the proof obligation to discharge.

`<model>_env<n>.v` the environment generated during weakest precondition calculus (already compiled by `coqc`): type definitions, global variables, etc.

`<model>_model.v` the definitions and properties of the memory model used (already compiled by `coqc`).

`f.script` the script file where all your proofs are stored. This is useful for reusing parts from previous scripts on similar goals.

As soon as `coqide` exits, the edited proof script is saved back to the script file, and finally checked by `coqc`. Do not forget to save your proof before exiting `coqide`.

Why. Finally, a wide range of automated provers are supported by WP plug-in thanks to the Why 2.29 prover interface. Both the `why` translation tool and the `why-dp` utility are required. You also need to install external provers by your own. The accepted values corresponding to these provers for the `-wp-proof` option are: `simplify`, `yices`, `cvc3`, `vampire`, `z3`, `zenon`.

2.2.6 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.

2.2.7 Additional Proof Libraries

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

`-wp-include <dir,...>` sets the directories where external libraries are looked for. The current directory (implicitly added to that list) is always looked up first.

`-wp-coq-lib <file,...>` looks for Coq files (`".v"`, the extension can be omitted) in included directories, and copies them into WP working directory (see option `-wp-out`). The files are then compiled in the same environment than for proof obligations, and imported for proving each goal. In particular, it is possible to use external libraries within `-wp-coq-tactic` command.

`-wp-why-lib <file,...>` looks for Why library files (`".why"`, the extension can be omitted) in included directories, and copies them into the proof obligation files for Why based SMT solvers.

`-wp-alt-ergo-lib <file,...>` looks for Why library files (`".why"`, the extension can be omitted) in included directories, and copies them into the proof obligation files for Alt-Ergo.

2.2.8 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-(no)-warnings` displays details when warnings are emitted during proof obligation generation (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.

`-wp-(no)-dot` generates a graphical representation of the CFG in the dot format used by the GraphViz tools[3] (default is: `no`).

The output directory contains a lot of files. All files are generated with the following naming convention:

`<goal>_head.txt` a summary of the generated proof obligation. This file contains the warning emitted during weakest precondition calculus.

`<goal>_body.txt` a human-readable description of the proof obligation.

`<goal>_log_<prover>.txt` a log from the prover when it has been run on the goal.

The complete goal submitted to external provers:

`<goal>_po.why` for WHY.

`<goal>_po_why.<ext>` generated by WHY for external decision procedures.

`<goal>_po_ergo.why` for Alt-Ergo without arrays.

`<goal>_po_aergo.why` for Alt-Ergo with arrays.

`<goal>_po.v` for Coq.

Each complete goal actually consists of the specification of the model, an environment describing the C definitions of your program, and the elementary goal itself. The environments are:

`<env>.txt` in human-readable description of the environment.

`<env>.why` for Why.

`<env>_ergo.why` for Alt-Ergo without arrays.

[3] http://www.graphviz.org

`<env>_aergo.why` for Alt-Ergo with arrays.

`<env>.v` for Coq.

The elementary goals are:

`<goal>.why` the elementary goal generated for WHY.

`<goal>_ergo.why` the elementary goal generated for Alt-Ergo without arrays.

`<goal>_aergo.why` the elementary goal generated for Alt-Ergo with arrays.

`<goal>.v` the elementary goal generated for Coq.

Finally, definitions and properties of the memory model are distributed in the Frama-C `share/wp` directory with similar naming conventions. Their Coq instances are copied on the temporary directory for separate compilation purposes.

To discharge a proof obligation, WP plug-in assembles an input for the external decision prover composed of three inputs: the resources for selected memory model, the resources from the environment of the goal, and the goal itself.

Remark: to save space on disk, when generating proof obligations from the command line, the proof obligations are only generated for the requested prover format. This behavior is turned off under the GUI and in debug mode. Hence, you still get all formats available for all provers in these cases.

2.3 Plug-in Developer Interface

The WP plug-in has several entry points registered in the `Dynamic`[4] module of Frama-C.

`Wp.run` runs the weakest precondition calculus using the options to know what to compute. This is similar to using `-wp` on the command line;

`Wp.wp_clear` erases all internal data of plug-in WP. Properties proved by WP are not erased, but all generated proof obligations are lost. This is a safe workaround for working with WP on multiple projects.

`Wp.wp_compute kf_opt bhv_list_opt prop_opt` where:

- `kf_opt` is an optional kernel function;
- `bhv_list_opt` specifies an optional behavior list;
- `prop_opt` specifies an optional property;

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

2.3.1 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:

```
<config...>
@HEAD
<head contents...>
@FUNCTION
<per function contents...>
@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 `"-"`.

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

`@LEMPREFIX "<`*text*`>"` text to be printed before names of lemmas. Default is `"(Lem.)"` (with a trailing space).

The generated report consists of three optional parts, corresponding to Head, Function and Tail sections of the wp-report specification file. First, the head contents lines are produced. Then, for each function analyzed by WP, the content lines of Function section are printed. Finally, the Tail content lines are printed.

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
`%function`	current function name

Provers	
(`<`*prover*`>`)	A prover name (see `-wp-proof`)
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 `"%{..}"`.

2.4 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-proof, -wp-check.` 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 preliminary version of the 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 and controlled by option `-wp-(no)-logicvar` (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, controled by option `-wp-byreference` (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) made of:

- Natural, signed, unbounded integer constants and their operations;
- Natural real numbers and their operations;
- Arrays (as total maps) and tuples;

- Abstract (polymorphic) data types;
- Anonymous function symbols;
- 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:

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

There are two such models with pointers available with the WP plug-in:

`-wp-model Store` : a simple memory model with two heap-variables. One is for the allocation table that deals with pointer validity. The second one stores numerical and pointer values into an array indexed by pointers. This model is not capable of handling unions and casts of pointer types.

`-wp-model Runtime` : a low-level memory model, also with two heap-variables. One is for the allocation table, and second one stores all values of the heap as an array of bytes indexed by pointers. All operations can be handled by this model, but the generated proof obligations are generally untractable by automated decision procedures.

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 option `-wp-no-logicvar`.

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 the `-wp-byreference` option, and the necessary separation conditions are generated on-the-fly. To summarize, the `-wp-byreference` option:

- detects pointer or array variables that are always passed by reference.
- generates additional pre-conditions to prevent aliasing between arguments at call sites.
- assigns the detected variables passed by reference to the Hoare memory model.

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

Chapter 4

WP Simplifier

(Experimental).

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. The simplification technology involved a complete refactoring of WP internals, such that it can not be used with the usually available memory models. Instead, the simplifier is activated by using a novel implementation of Store model, called the Typed model. Logic variables and reference parameters have been already ported on this new memory model, and Runtime memory model will be available in a near future. These novel implementations of memory models make better usage of build-in theories of provers, especially the array and record theories of the recent Alt-Ergo 0.94 prover.

Not only the memory models, but also the *weakest precondition* calculus have been enhanced to take advantages from the new logic language and its simplifications. Hence, 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.

4.1 Specific Options

The WP behavior is slightly different when using these new features. This section summarize those differences and few specific options.

`-wp-model Typed` activates the new memory model with the new logical language and its basic simplifications.

`-wp-qed` activates advanced simplifications in $\mathcal{L}$. Goals are pre-simplified before being submitted to the provers. If the residual of simplification is a trivial goal (*true* goal or *false* hypothesis), the proof is discharged without any external prover.

`-wp-byreference` activates the detection of reference parameters as discussed in section 3.5.

`-wp-split` deactivates the factorization of proof contexts on conditional statements. This provides opportunities for more aggressive simplifications on each conditional branches, but at the price of a potential combinatorial explosion of generated proof obligations.

Support of Alt-Ergo prover is also enhanced in several ways. Time and proof steps are now reported, and it is possible to limit the number of proof steps with a new option `-wp-steps`. The graphical user interface of Alt-Ergo can also be launched, by selecting `-wp-proof altgr-ergo`. The command line prover `alt-ergo` is tried first, then the GUI (if installed).

In the graphical user interface of WP plug-in, running by hand alt-ergo on a proof obligations also launches `altgr-ergo` when available.

4.2 Logic Normalization

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.

4.3 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.

4.4 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.

4.5 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).

4.6 Conclusion

This experimental simplifier actually comes with a very novel *weakest precondition* calculus that will be extended to all memory models and standard features of WP in a near future.

Bibliography

[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 V8.3*, 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.