INTRODUCTION TO C PROGRAM PROOF USING FRAMA-C AND ITS WP PLUGIN

Allan Blanchard

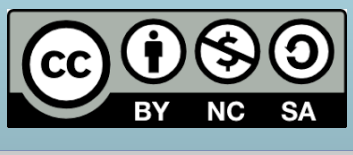
December 2017

Geste de savoir

Contents

1 Introduction

License



Please note that this document is the very first English version of the tutorial. If you find some errors, please do not hesitate to contribute at:
https://github.com/AllanBlanchard/tutoriel_wp
Please use the markdown files, the LaTeX file being generated from them.

Information

In this tutorial, some examples and some elements of organization are similar to the ones used in the TAP 2013 tutorial by Nikolai Kosmatov, Virgile Prevosto and Julien Signoles of the CEA LIST, since it is quite didactic. It also contains examples taken from *ACSL By Example* by Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans Pohl and Juan Soto from the Fraunhofer. The remaining ideas come from my personal experience with Frama-C and WP. The only requirement to this tutorial is to have a basic knowledge of the C language, and at least to be familiar with the notion of pointer.

Despite its old age, C is still a widely used programming language. Indeed, no other language can pretend to be available on so many different (hardware and software) platforms, its low-level orientation and the amount of time invested in the optimization of its compilers allows to generate very light and efficient machine code (if the code allows it of course), and that there are a lot of experts in C language, which is an important knowledge base.

Furthermore, a lot of systems rely on a huge amount of code historically written in C, that needs to be maintained and sometimes fixed, as it would be far too costly to rewrite these systems.

But anyone who has already developed with C also know that it is very hard to perfectly master this language. There are numerous reasons, but ambiguities in the ISO C, and the fact that it is extremely permissive, especially about memory management, make the development of robust C program very hard, even for an experienced programmer.

However, the C language is often chosen for critical systems (avionics, railway, armament, ...) where it is appreciated for its good performances, its technological maturity and the predictability of its compilation.

In such cases, the needs in term of code covering by tests become important. The question "is our software tested enough?" becomes a question to which it is very hard to answer. Program proof can help us. Rather than test all possible and (un)imaginable inputs to the program, we will *mathematically* prove that there cannot be any problem at runtime.

The goal of this tutorial is to use Frama-C, a tool developed at the CEA LIST, and WP, its deductive proof plugin, to learn the basics about C program proof. More than the use of the tool itself, the goal of this tutorial is to convince that it is more and more possible to write programs without any programming error, but also to sensitize to simple notions that allows to better understand and write programs.

Information

Many thanks to the different beta-testers for their constructives feedback:

- Taurre (the example in the section III - 3 - 4 has been shamefully ripped off from one of his posts)
- barockobamo
- Vayel

I thank ZesteDeSavoir validators who helped me improve again the quality of this tutorial:

- Taurre (again)
- Saroupille

Finally, many thanks to Jens Gerlach for his help during the translation of this tutorial from French to English.

2 Program proof and our tool for this tutorial: Frama-C

The goal of this first part is, in the first section, to introduce the idea of program proof without giving to much details, and then, in the second section, we will give the necessary instructions to install Frama-C and some automatic provers that we will use in this tutorial.

2.1 Program proof

2.1.1 Ensure program reliability

It is often difficult to ensure that our programs have only correct behaviors. Moreover, it is already complex to establish a good criteria that makes us confident enough to say that a program correctly works:

- beginners simply "try" to use their programs and considers that these programs work if they do not crash (which is not a really good indicator in C language),
- more advanced developers establish some test cases for which they know the expected result and compare the output they obtain,
- most of companies establish complete test bases, that covers as much code as they can ; which are systematically executed on their code. Some of them apply test driven development,
- in critical domains, such as aerospace, railway or armament, every code needs to be certified using standardized processes with very strict criteria about coding rules and code covering by the test.

In all these ways to ensure that a program produces only what we expect it to produce, a word appears to be common: *test*. We *try* different inputs in order to isolate cases that are problematic. We provide inputs that we *estimate to be representative* of the actual use of the program (note that unexpected use case are often not considered whereas there are generally the most dangerous ones) and we verify that the results we get are correct. But we cannot test *everything*. We cannot try *every* combination of *every* possible input of a program. It is then quite hard to choose good tests.

The goal of program proof is to ensure that, for any input provided to a given program, if it respects the specification, then the program will only well-behave. However, since we cannot test everything, we will formally, mathematically, establish a proof that our software can only exhibit specified behaviors, and that runtime-errors are not part of these behaviors.

A very well-known quote from Dijkstra precisely express the difference between test and proof :

> Program testing can be used to show the presence of bugs, but never to show their absence! [Dijkstra]

2.1.1.1 The developer's Holy Grail: the bug-free software

Every time we can read news about attacks on computer systems, or viruses, or bugs leading to crashes in well known apps, there is always the one same comment "the bug-free/perfectly secure program does not exist". And, if this sentence is quite true, it is a bit misunderstood.

Apart from the difference between safety and security (that we can very vaguely define by the existence of a malicious entity), we do not really define what we mean by "bug-free". Creating software always rely at least on two steps: we establish a specification of what we expect from the program and then we produce the source code of the program that must respect this specification. Both of these steps can lead to the introduction of errors.

In this tutorial, we will show how we can prove that an implementation verify a given specification. But what are the arguments of program proof, compared to program testing? First, the proof is complete, it cannot forget some corner case if the behavior is specified (program test cannot be complete, being exhaustive would be far too costly). Then, the obligation to formally specify with a logic formulation requires to exactly understand what we have to prove about our program.

One could cynically say that program proofs shows that "the implementation does not contain bugs that do not exist in the specification". But, well, it is already a big step compared to "the implementation does not contain too many bugs that do not exist in the specification". Moreover, there also exist approaches that allow to analyze specifications to find errors or under-specified behaviors. For example, with model checking techniques, we can create an abstract model from the specification and produce the set of states that can be reach according to this model. By characterizing what is an error state, we can determine if reachable states are error states.

2.1.2 A bit of context

Formal methods, as we name them, allow in computer science to rigorously, mathematically, reason about programs. There exist a lot of formal methods that can take place at different levels from program design to implementation, analysis and validation, and for all system that allow to manipulate information.

Here, we will focus on a method that allows to formally verify that our programs have only correct behaviors. We will use tools that are able to analyze a source code and to determine whether a program correctly implements what we want to express. The analyzer we will use provides a static analysis, that we can oppose to dynamic analysis.

In static analysis, the analyzed program is not executed, we reason on a mathematical model of the states it can reach during its execution. On the opposite, dynamic analyses such as program testing, require to execute the analyzed source code. Note that there exist formal dynamic analysis methods, for example automatic test generation, or code monitoring techniques that allows to instrument a source code to verify some properties about it during execution (correct memory use, for example).

Talking about static analyses, the model we use can be more or less abstract depending on the techniques, it is always an approximation of possible states of the program. The more the approximation is precise, the more the model is concrete, the more the approximation is vague, the more it is abstract.

To illustrate the difference between concrete and abstract model, we can have a look to the model of simple chronometer. A very abstract model of a chronometer could be the one presented in

Figure 2.1.

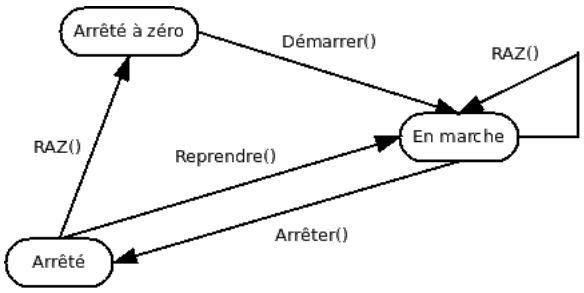


Figure 2.1: A very abstract model of a chronometer

We have a model of the behavior of our chronometer with the different states it can reach according to the different actions we can perform. However, we do not have modeled how these states are depicted inside the program (is this a C enumeration? a particular program point in the source code?), nor how is modeled the time computation (a single variable? multiple ones?). It would then be difficult to specify properties about our program. We could add some information:

- State stopped at 0 : time = 0s
- State running : time > 0s
- State stopped : time > 0s

Which gives us a more concrete model but that is still not precise enough to ask interesting questions like: "is it possible to be in the state stopped and that time is still updated?", as we do not model how the time measurement is updated by the chronometer.

On the opposite, with the source code of the program, we have a concrete model of the chronometer. The source code expresses the behavior of the chronometer since it will allow us to produce the executable. But this is still not the more concrete model! For example, the executable in machine code format, that we obtain after compilation, is far more concrete than our program.

The more a model is concrete, the more it precisely describes the behavior of our program. The source code more precisely describes the behavior than our diagram, but it is less precise than the machine code. However, the more the model is precise, the more it is difficult to have a global view of the defined behavior. Our diagram is understandable in a blink of an eye, the source code requires more time, and for the executable ... Every single person that has already opened an executable with a text editor by error knows that it is not really pleasant to read[1].

When we create an abstraction of a system, we approximate it, in order to limit the knowledge we have about it and ease our reasoning. A constraint we must respect, if we want our analysis to be correct, is to never under-approximate behaviors: we would risk to remove a behavior that contains an error. However, when we over-approximate it, we can add behaviors that cannot happen, and if we add to many of them, we could not be able to prove our program is correct, since some of them could be faulty behaviors.

In our case, the model is quite concrete. Every type of instruction, of control structure, is associated to a precise semantics, a model of its behavior in a pure logic, mathematical, world. The logic we use here is a variant of the Hoare logic, adapted to the C language and all its complex subtleties (which makes this model concrete).

[1]There also exists formal methods which are interested in understanding how executable machine code work, for example in order to understand what malwares do or to detect security breaches introduced during compilation.

2.1.3 Hoare triples

Hoare logic is a program formalization method proposed by Tony Hoare in 1969 in a paper entitled *An Axiomatic Basis for Computer Programming*. This method defines:

- axioms, that are properties we admit, such as "the skip action does not change the program state",
- rules to reason about the different allowed combinations of actions, for example "the skip action followed by the action A" is equivalent to "the action A".

The behavior of the program is defined by what we call "Hoare triples":

$$\{P\}C\{Q\}$$

Where P and Q are predicates, logic formulas that express properties about the memory at particular program points. C is a list of instructions that defines the program. This syntax expresses the following idea: "if we are in a state where P is verified, after executing C and if C terminates, then Q is verified for the new state of the execution". Put in another way, P is a sufficient precondition to ensure that C will bring us to the postcondition Q. For example, the Hoare triples that corresponds to the skip action is the following one:

$$\{P\}\ \textbf{skip}\ \{P\}$$

When we do nothing, the postcondition is the precondition.

Along this tutorial, we will present the semantics of different program constructs (conditional blocks, loops, etc) using Hoare logic. So, we will not enter into details now since we will work on it later. It is not necessary to memorize these notions nor to understand all the theoretical background, but it is still useful to have some ideas about the way our tool works.

All of this gives us the basics that allows us to say "here is what this action does" but it does not give us anything to mechanize a proof. The tool we will use rely on a technique called weakest precondition calculus.

2.1.4 Weakest precondition calculus

The weakest precondition calculus is a form of predicate transform semantics proposed by Dijkstra in 1975 in *Guarded commands, non-determinacy and formal derivation of programs*.

This sentence can appear complex but the actual meaning is in fact quite simple. We have seen before that Hoare logic gives us rules that explain the behavior of the different actions of a program, but it does not say how to apply these rules to establish a complete proof of the program.

Dijkstra reformulate the Hoare logic by explaining, in the triple $\{P\}C\{Q\}$, how the instruction, or the block of instructions, C transforms the predicate P in Q. This kind of reasoning is called *forward-reasoning*. We calculate from the precondition and from one or multiple instructions, the strongest postcondition we can reach. Informally, considering what we have in input, we calculate what we will get in output. If the postcondition we want is as strong or weaker, then we prove that there is not any unwanted behaviors.

For example:

```
int a = 2;
a = 4;
//calculated postcondition : a == 4
//wanted postcondition     : 0 <= a <= 30
```

Ok, 4 is an allowed value for a.

The form of predicate transformer semantics which we are interested in works the opposite way, we speak about *backward-reasoning*. From the wanted postcondition and the instructions we are reasoning about, we find the weakest precondition that ensures this behavior. If our actual precondition is at least as strong, that is to say, if it implies the computed precondition, then our program is correct.

For example, if we have the instruction:

$\{P\}\ x := \mathsf{a}\ \{x = 42\}$

What is the weakest precondition to validate the postcondition $\{x = 42\}$? The rule will define that P is $\{\mathsf{a} = 42\}$.

For now, let us forget about it, we will come back to these notions as we use them in this tutorial to understand how our tools work. So now, we can have a look to these tools.

2.2 Frama-C

2.2.1 Frama-C? WP?

Frama-C (FRAmework for Modular Analysis of C code) is a platform dedicated to the analysis of C programs created by the CEA List and Inria. It is based on a modular architectures allowing to use different (collaborating or not) plugins. The default plugins comprises different static analyses (that do not execute source code), dynamic analyses (that requires code execution), or combining both.

Frama-C provides a specification language called ACSL ("Axel") for ANSI C Specification Language and that allows us to express the properties we want to verify about our programs. These properties will be written using code annotations in comment sections. If one has already used Doxygen, it is quite similar, except that we write logic formulas and not text. During this tutorial, we will extensively write ASCL code, so let us just skip this for now.

The analysis we will use is provided by the WP plugin (for Weakest Precondition), it implements the technique we presented earlier: from ACSL annotations and the source code, the plugin generates what we call proof goals, that are logic formulas that must be verified to be satisfiable or not. This verification can be performed manually or automatically, here we use automatic tools.

We will use a SMT solver (statisfiability modulo theory, we will not detailed how it works). This solver is

Alt-Ergo, that was initially developed by the Laboratoire de Recherche en Informatique d'Orsay, and is today maintained and updated by OCamlPro.

2.2.2 Installation

Frama-C is developed under Linux and OSX. It is then better supported under these two. It is still possible to install it on Windows and its use would be equivalent to the one we could have on Linux but:

Warning

- the tutorial presents the use of Frama-C on Linux (or OSX) and the author did not experiment the differences that could exists with Windows,
- the "Bonus" section of this part could not be accessible under Windows.

2.2.2.1 Linux

2.2.2.1.1 Using package managers Under Debian, Ubuntu and Fedora, there exist packages for Frama-C. In such a case, it is enough to type a command like:

```
apt install frama-c # Debian-like
yum install frama-c # Fedora
```

However, these repositories are not necessarily up to date with the last version of Frama-C. This is not a big problem since there is not new versions of Frama-C every day, but it still important to know it.

Go to the section "Verify installation" to perform some tests about your installation.

2.2.2.1.2 Via opam A second option is to use Opam, a package manager for Ocaml libraries and applications.

First of all, Opam must be installed (see its documentation). Then, some packages from your Linux distribution must be installed before installing Frama-C:

- lib gtk2 dev
- lib gtksourceview2 dev
- lib gnomecanvas2 dev
- (recommended) lib zarith dev

Once it is done, we can install Frama-C and Alt-Ergo.

```
opam install alt-ergo
opam install frama-c
```

Go to the section "Verify installation" to perform some tests about your installation.

2.2.2.1.3 Via "manual" compilation

The packages we have listed in the Opam section are required (of course, Opam itself is not). It requires a recent version of Ocaml and its compiler (including compiler to native code).

After having extracted the folder available here : http://frama-c.com/download.html (Source distribution). Navigate to the folder and then execute the command line:

```
./configure && make && sudo make install
```

Go to the section "Verify installation" to perform some tests about your installation.

2.2.2.2 OSX

On OSX, the use of Homebrew and Opam is recommended to install Frama-C. The author do not use OSX, so here is a shameful copy and paste of the installation guide of Frama-C for OSX.

General Mac OS tools for OCaml:

```
> xcode-select --install
> open http://brew.sh
> brew install autoconf opam
```

Graphical User Interface:

```
> brew install gtk+ --with-jasper
> brew install gtksourceview libgnomecanvas graphviz
> opam install lablgtk ocamlgraph
```

Recommended for Frama-C:

```
> brew install gmp
> opam install zarith
```

Necessary for Frama-C/WP:

```
> opam install alt-ergo
> opam install frama-c
```

Also recommended for Frama-C/WP:

```
opam install altgr-ergo coq coqide why3
```

Go to the section "Verify installation" to perform some tests about your installation.

2.2.2.3 Windows

Currently, the installation of Frama-C for Windows requires Cygwin and an experimental version of Opam for Cygwin. So we need to install both as well as the MinGW Ocaml compiler.

Installation instructions can be found there:

Frama-C - Windows

Frama-C is then started using Cygwin.

Go to the section "Verify installation" to perform some tests about your installation.

```
/*@
  requires \valid(a) && \valid(b);
  assigns *a, *b;
  ensures *a == \old(*b);
  ensures *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 42;
  int b = 37;

  swap(&a, &b);

  //@ assert a == 37 && b == 42;

  return 0;
}
```

Figure 2.2: Code for verifying the installation

2.2.3 Verify installation

In order to verify that the installation has been correctly performed, we will use the simple code presented in Figure 2.2 in a file `main.c`:

Then, from a terminal, in the folder where the file has been created, we start Frama-C with the following command line:

```
frama-c-gui -wp -rte main.c
```

The window illustrated by Figure 2.3 should appear.

Clicking `main.c` in the left side panel to select it, we can see its content (slightly) modified, and some green bullets on different lines as illustrated by Figure 2.4.

Warning

The graphical user interface of Frama-C does not allow source code edition

Information

For color blinds, it is possible to start Frama-C with another theme where color bullets are replaced:

```
$ frama-c-gui -gui-theme colorblind file.c
```

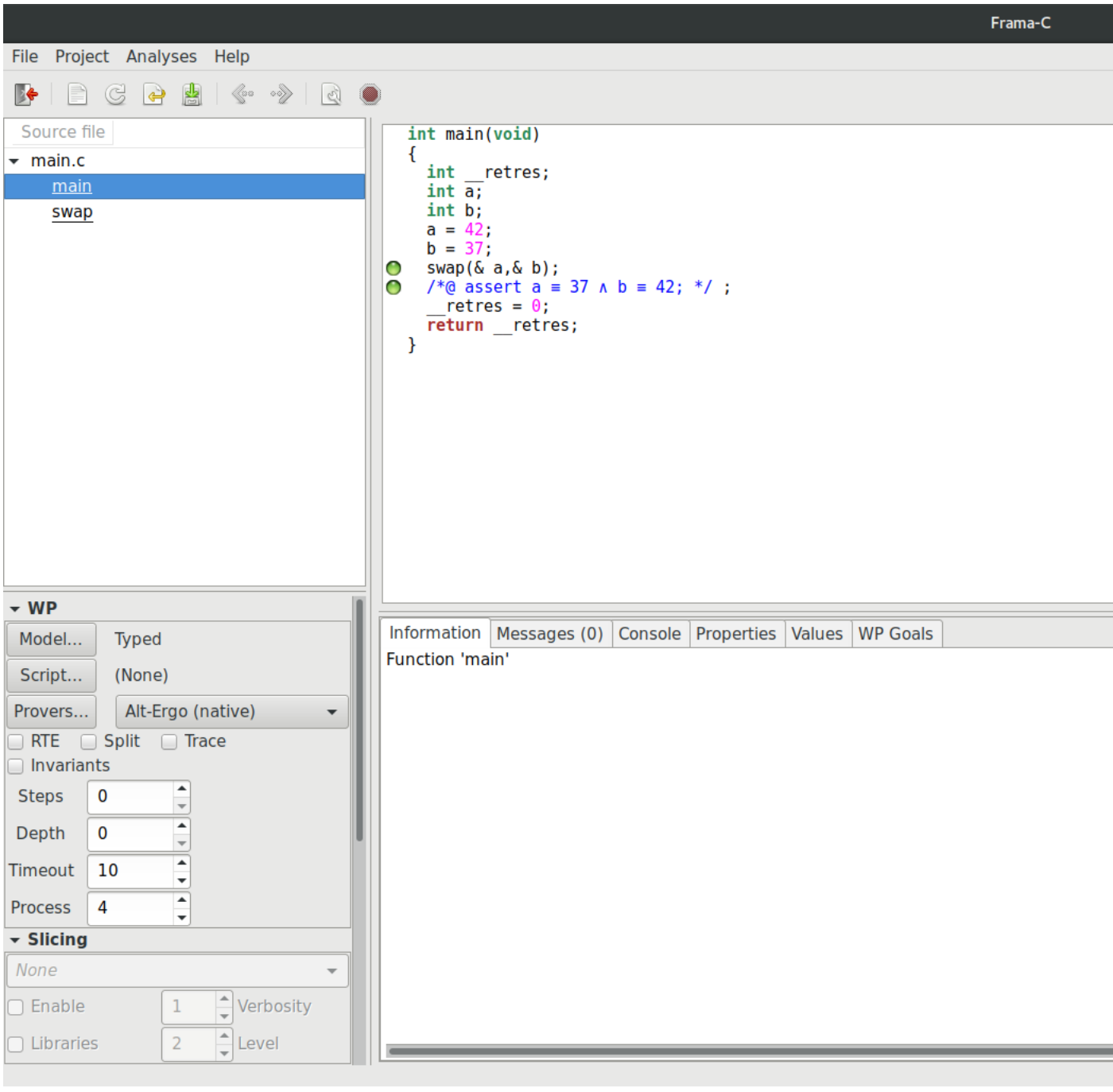


Figure 2.3: Verify installation 1

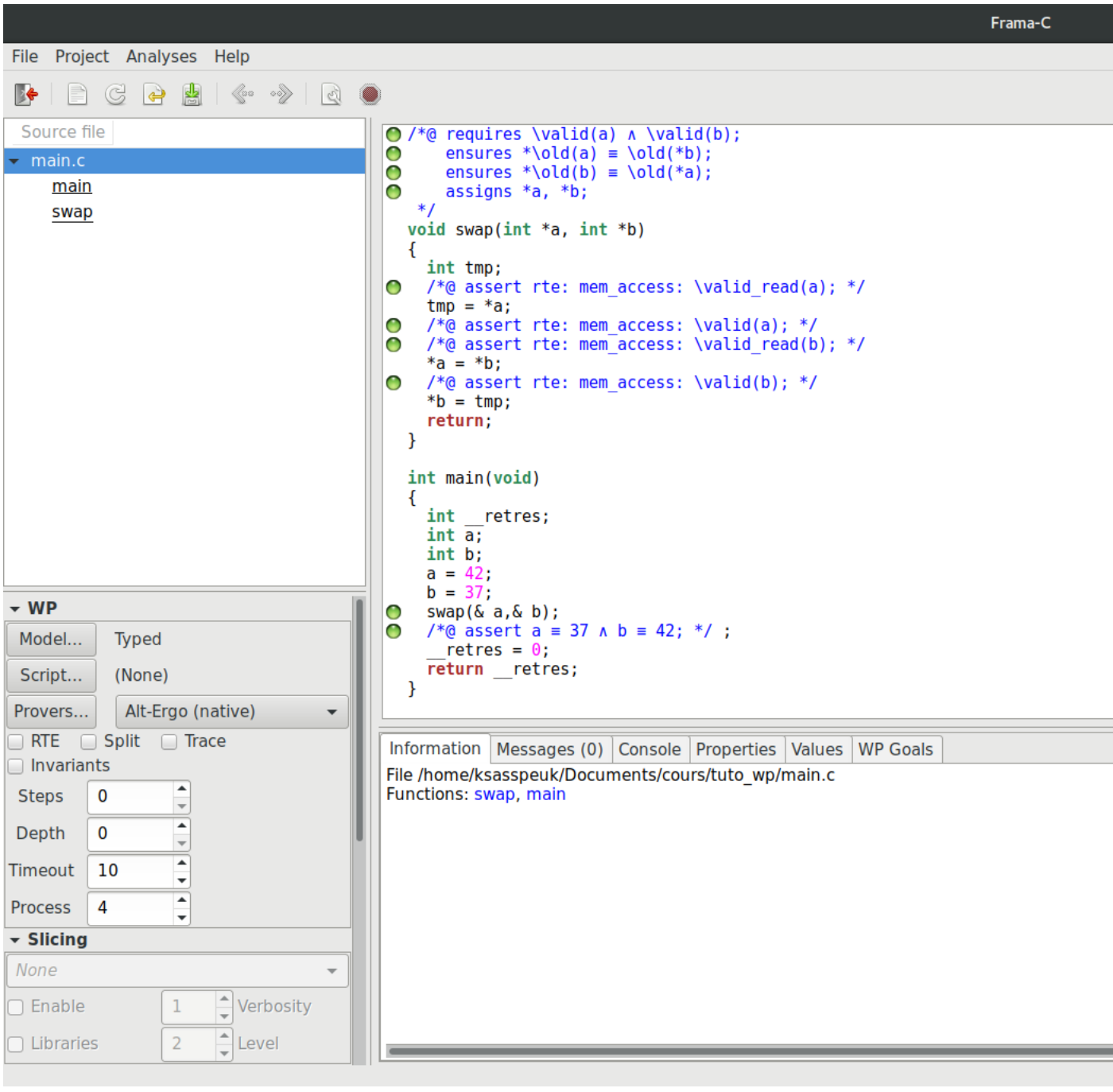


Figure 2.4: Verify installation 2

2.2.4 (Bonus) Some more provers

This part is optional, nothing in this section should be particularly useful *in the tutorial*. However, when we start to be interested in proving more complex programs, it is often possible to reach the limits of Alt-Ergo, which is basically available, and we would thus need some other provers.

2.2.4.1 Coq

Coq, which is developed by Inria, is a proof assistant. Basically, we write the proofs ourselves in a dedicated language and the platform verify (using typing) that the proof is actually a valid proof.

Why would we need such a tool? Sometimes, the properties we want to prove can be too complex to be solved automatically by SMT solvers, typically when they requires careful inductive reasoning with precise choices at each step. In this situation, WP allows us to generate proof goals translated in Coq language, and to write the proof ourselves.

To learn Coq, we would recommend this tutorial.

Information

If Frama-C has been installed using the package manager of a Linux distribution, Coq could be automatically installed.

If one needs more information about Coq and its installation, this page can help: The Coq Proof Assistant.

When we want to use Coq for a proof with Frama-C, we have to select it using the left side panel, in the WP part (see Figure 2.5).

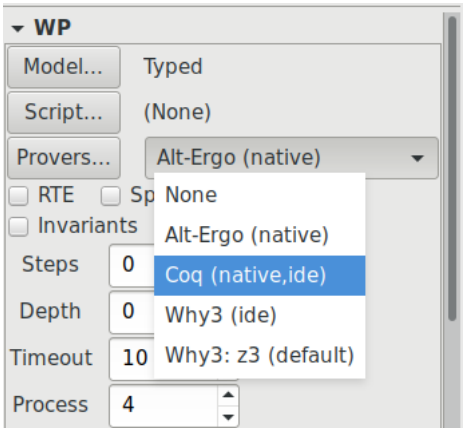


Figure 2.5: Select the Coq proof assistant

Information

The author does not know if it works under Windows.

2.2.4.2 Why3

Warning

To the author's knowledge, it is not possible (or, at least, not easy at all) to install Why3 under Windows. The author cannot be charged for injuries that could result of such an operation.

Why3 is deductive proof platform developed by the LRI in Orsay. It provides a programming language and a specification language, as well as a module that allows to interact with a wide variety of automatic and interactive provers. This point is the one that interest us here. WP can translate its proof goals to the Why3 language and then use Why3 to interact with solvers.

The Why3 website provides all information about it. If Opam is installed, Why3 is available using it, else, there is an another installation procedure.

On this website, we can find the list of supported provers. We recommend to install Z3 which is developed by Microsoft Research, and CVC4 which is developed by many research teams (New York University, University of Iowa, Google, CEA List). Those two provers are very efficient and somewhat complementary.

To use these provers, the procedure is explained in the Coq part that describes the selection of a prover for the proof. Notice that it could be necessary to ask the detection of freshly installed provers using the "Provers" button and then "Detect Provers" in the window that should pop.

Our tools are now installed and ready to be used.

The goal of this part, apart of the installation of our tools was to put in relief two main ideas:

- program proof is a way to ensure that our programs only have correct behaviors, described by our specification,
- it is still our work to ensure that this specification is correct.

3 Function contract

It is time to enter the heart of the matter. Rather than starting with basic notions of the C language, as we would do for a tutorial about C, we will start with functions. First because it is necessary to be able to write functions before starting this tutorial (to be able to prove that a code is correct, being able to write it correct is required), and then because it will allow us to directly prove some programs.

After this part about functions, we will on the opposite focus on simple notions like assignments or conditonal structures, to understand how our tool really works.

In order to be able to prove that a code is valid, we first need to specify what we expect of it. Building the proof of our program consists in ensuring that the code we wrote corresponds to the specification that describes its job. As we previously said, Frama-C provides the ACSL language to let the developer write contracts about each function (but that is not its only purpose, as we will see later).

3.1 Contract definition

The goal of a function contract is to state the conditions under which the function will execute. That is to say, what the function expect from the caller to ensure that it will correctly behave: the precondition, the notion of "correctly behave" being itself defined in the contract by the postcondition.

These properties are expressed with ACSL, the syntax is relatively simple if one have already developed in C language since it shares most of the syntax of boolean expressions in C. However, it also provides:

- some logic constructs and connectors that do not exists in C, to ease the writing of specifications,
- built-in predicates to express properties that are useful about C programs (for example: a valid pointer),
- as well as some primitive types for the logic that are more general than primitive C types (for example: mathematical integer).

We will introduce along this tutorial a large part of the notations available in ACSL.

ACSL specifications are introduced in our source code using annotations. Syntactically, a function contract is integrated in the source code with this syntax:

```
/*@
  //contract
*/
void foo(int bar){

}
```

Notice the @ at the beginning of the comment block, this indicates to Frama-C that what follows are annotations and not a comment block that it should simply ignore.

Now, let us have a look to the way we express contracts, starting with postconditions, since it is what we want our function to do (we will later see how to express precondition).

3.1.1 Postcondition

The postcondition of a function is introduced with the clause `ensures`. We will illustrate its use with the following function that returns the absolute value of an input. One of its postconditions is that the result (which is denoted with the keywork `\result`) is greater or equal to 0.

```
/*@
  ensures \result >= 0;
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

(Notice the ; at the end of the line, exactly as we do in C).

But that it is not the only property to verify, we also need to specify the general behavior of a function returning the absolute value. That is: if the value is positive or 0, the function returns the same value, else it returns the opposite of the value.

We can specify multiple postconditions, first by combining them with a && as we do in C, or by introducing a new `ensures` clause, as we illustrate here:

```
/*@
  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val ) &&
          (val <  0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

This specification is the opportunity to present a very useful logic connector provided by ACSL and that does not exist in C: the implication $A \Rightarrow B$, that is written `A ==> B` in ACSL. The truth table of the implication is the following:

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

That means that an implication $A \Rightarrow B$ is true in two cases: either A is false (and in this case, we do not check the value of B), or A is true and then B must also be true. The idea finally being "I want to know if when A is true, B also is. If A is false, I don't care, I consider that the complete

formula is true".

Another available connector is the equivalence $A \Leftrightarrow B$ (written `A <==> B` in ACSL), and it is stronger. It is conjunction of the implication in both ways $(A \Rightarrow B) \wedge (B \Rightarrow A)$. This formula is true in only two cases: A and B are both ture, or false (it can be seen as the negation of the exclusive or).

Information

Let's give a quick reminder about all truth tables of usual logic connectors in first order logic ($\neg$ = !, $\wedge$ = &&, $\vee$ = ||) :

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
F	F	V	F	F	V	V
F	V	V	F	V	V	F
V	F	F	F	V	F	F
V	V	F	V	V	V	T

We can come back to our specification. As our files become longer and contains a lot of specifications, if can be useful to name the properties we want to verify. So, in ACSL, we can specify a name (without spaces) followed by a `:`, before stating the property. It is possible to put multiple levels of names to categorize our properties. For example, we could write this:

```
/*@
  ensures positive_value: function_result: \result >= 0;
  ensures (val >= 0 ==> \result == val) &&
          (val < 0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

In most of this tutorial, we will not name the properties we want to prove, since they will be generally quite simple and we will not have too many of them, names would not give us much information.

We can copy and paste the function `abs` and its specification in a file `abs.c` and use Frama-C to determine if the implementation is correct against the specification. We can start the GUI of Frama-C (it is also possible to use the command line interface of Frama-C but we will not use it during this tutorial) by using this command line:

```
$ frama-c-gui
```

Or by opening it from the graphical environment.

It is then possible to click on the button "Create a new session from existing C files", files to analyze can be selected by double-clicking it, the OK button ending the selection. Then, adding other files will be done by clicking Files > Source Files.

Notice that it is also possible to directly open file(s) from the terminal command line passing them to Frama-C as parameter(s):

```
$ frama-c-gui abs.c
```

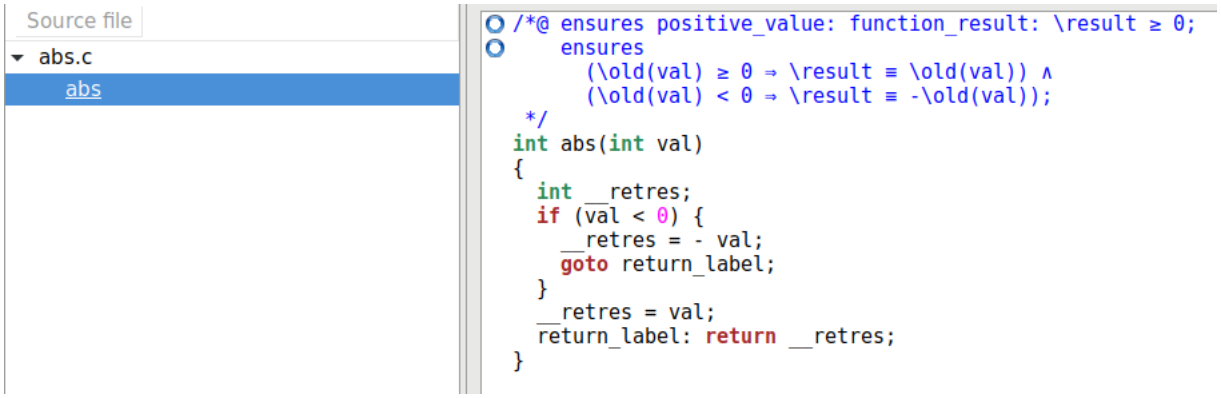


Figure 3.1: The side panel gives the files and functions tree

The window of Frama-C opens and in the panel dedicated to files and functions, we can select the function `abs`. At each `ensures` line, we can see a blue circle, it indicates that no verification has been attempted for these properties (see Figure 3.1).

We ask the verification of the code by right-clicking the name of the function and "Prove function annotations by WP" (see Figure 3.2).

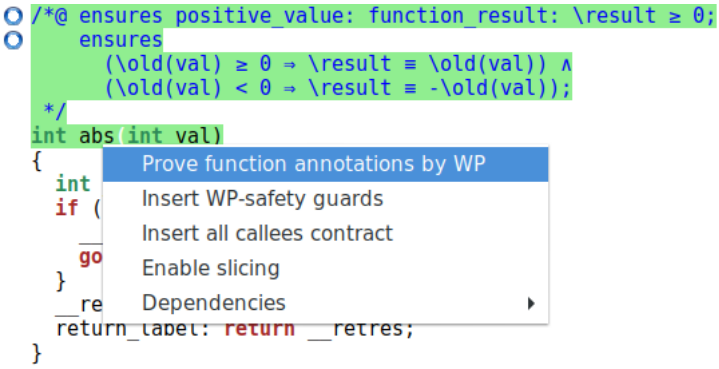


Figure 3.2: Start the verification of `abs` with WP

We can see that blue circles become green bullets, indicating that the specification is indeed ensured by the program. We can also prove properties one by one by right-clicking on them and not on the name of the function.

But is our code really bug free ? WP gives us a way to ensure that a code respects a specification, but it does not check for runtime errors (RTE). This is provided by another plugin that we will use here and that is called RTE. Its goal is to add, in the program, some controls to ensure that the program cannot create runtime errors (integer overflow, invalid pointer dereferencing, 0 division, etc).

To active these controls, we check the box pointed by the screenshot (in the WP panel). We can also ask Frama-C to add them in a function by right-clicking on its name and then click "Insert RTE guards" (see Figure 3.3).

Finally, we execute the verification again (we can also click on the "Reparse" button of the toolbar, it will deletes existing proofs).

We can then see that WP fails to prove the absence of arithmetic underflow for the computation of `-val`. And, indeed, on our architectures, `-INT_MIN` (-2^{31}) > `INT_MAX` ($2^{31} - 1$) (see Figure 3.4).

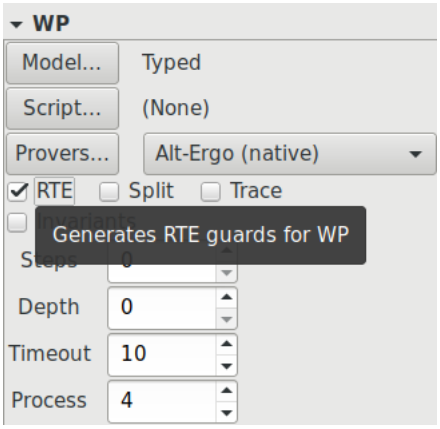


Figure 3.3: Activate runtime error absence verification

```
/*@ ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ val; */
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

Figure 3.4: Incomplete proof of `abs`

Information

We can notice that the underflow risk is real for us, since our computers (for which the configuration is detected by Frama-C) use the Two's complement implementation of integers, which do not defined the behavior of under and overflows.

Here, we can see another type of ACSL annotation. By the line `//@ assert property ;`, we can ask the verification of property at a particular program point. Here, RTE inserted for us, since we have to verify that `-val` does not produce an underflow, but we can also add such an assertion manually in the source code.

In this screenshot, we can see two new colors for our bullets: green+brown and orange.

The green+brown color indicates that the proof has been produced but it can depend on some properties that have not been verified.

If the proof has not been entirely redone after adding the runtime error checks, these bullets must still be green. Indeed, the corresponding proofs have been realized without the knowledge of the property in the assertion, so they cannot rely on this unproved property.

When WP transmits a proof obligation to an automatic prover, it basically transmits two types of properties : G, the goal, the property that we want to prove, and A_1 ... A_n, the different assumptions we can have about the state of the memory at the program point where we want to verify G. However, it does not receive (in return) the properties that have been used by the prover to validate G. So, if A_3 is an assumption, and if WP did not succeed in getting a proof of A_3, it indicates that G is true, but only if we succeed in proving A_3.

The orange color indicates that no prover could determine if the property is verified. There are two possibles reasons:

- the prover did not have enough information,
- the prover did not have enough time to compute the proof and encountered a timeout (which can be configured in the WP panel).

In the bottom panel, we can select the "WP Goals" tab (see Figure 3.5), it shows the list of proof obligations, and for each prover the result is symbolized by a logo that indicates if the proof has been tried and if it succeeded, failed or encountered a timeout (here we can see a try with Z3 where we had a timeout on the proof of absence of RTE).

In the first column, we have the name of the function the proof obligation belongs to. The second column indicates the name of proof obligation. For example here, our postcondition is named "Post-condition 'positive_value,function_result' ", we can notice that if we select a property in this list, it is also highlighted in the source code. Unnamed properties are automatically named by WP with the kind of wanted property. In the third column, we see the memory model that is used for the proof, we will not talk about it in this tutorial. Finally, the last columns represent the different provers available through WP.

In these provers, the first element is Qed. It is not really a prover. In fact, if we double-click on the property "absence of underflow" (highlight in blue in the last screenshot), we can see corresponding proof obligation (see Figure 3.6).

This is the proof obligation generated by WP about our property and our program, we do need to understand everything here, but we can get the general idea. It contains (in the "Assume" part) the

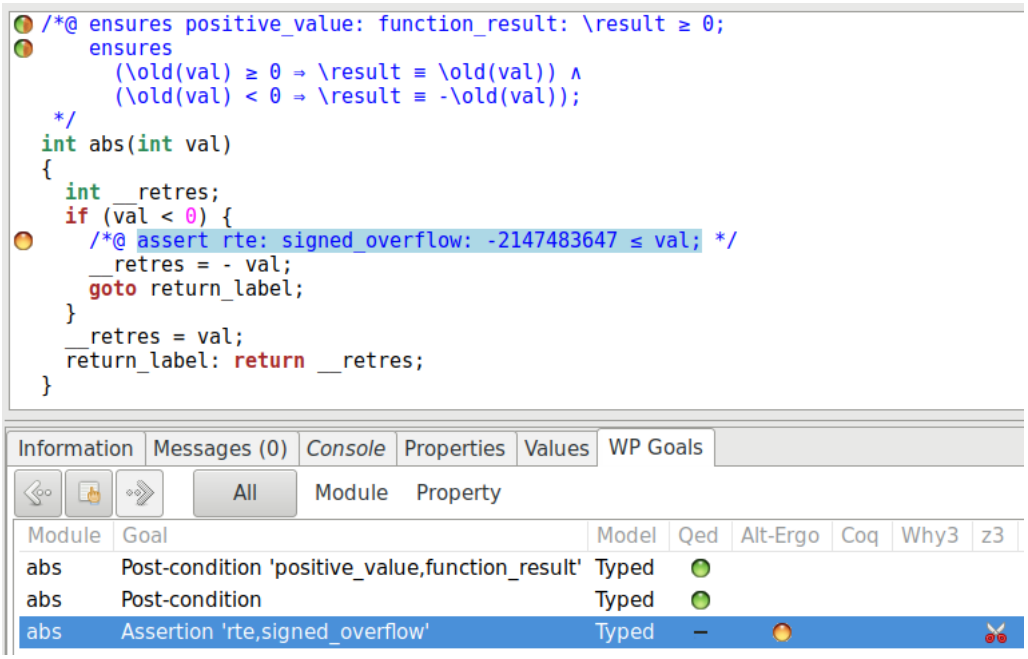


Figure 3.5: Proof obligations panel of WP for `abs`

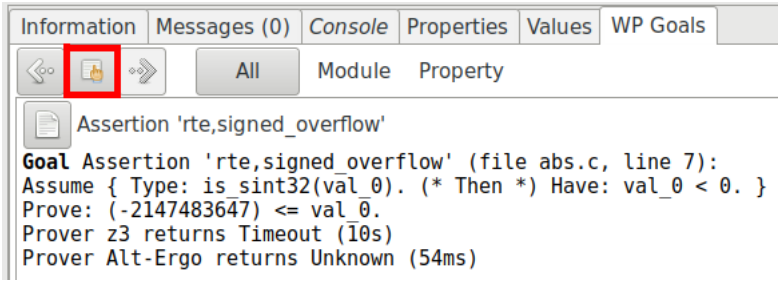


Figure 3.6: Proof obligation associated to the verification of absence of underflow in `abs`

assumptions that we have specified and those that have been deduced by WP from the instructions of the program. It also contains (in the "Prove" part) the property that we want to verify.

What does WP do using these properties ? In fact, it transforms them into a logic formula and then asks to different provers if it is possible to satisfy this formula (to find for each variable, a value that can make the formula true), and it determines if the property can be proved. But before sending the formula to provers, WP uses a module called Qed, which is able to perform different simplifications about it. Sometimes, as this is the case for the other properties about `abs`, these simplifications are enough to determine that the property is true, in such a case, WP do not need the help of the automatic solvers.

When automatic solvers cannot ensure that our properties are verified, it it sometimes hard to understand why. Indeed, provers are generally not able to answer something other than "yes", "no" or "unknown", they are not able to extract the reason of a "no" or an "unknown". There exists tools that can explore a proof tree to extract this type of information, currently Frama-C do not provide such a tool. Reading proof obligations can sometimes be helpful, but it requires a bit of practice to be efficient. Finally, one of the best way to understand the reason why a proof fails is to try to do it interactively with Coq. However, it requires to be quite comfortable with this language to not being lost facing the proof obligations generated by WP, since these obligations need to encode some elements of the C semantics that can make them quite hard to read.

If we go back to our view of proof obligations (see the squared button in the last screenshot), we can see that our hypotheses are not enough to determine that the property "absence of underflow" is true (which is indeed currently impossible), so we need to add some hypotheses to guarantee that our function will well-behave: a call precondition.

3.1.2 Preconditon

Preconditions are introduced using `requires` clauses. As we could do with `ensures` clauses, we can compose logic expressions and specify multiple preconditions:

```
/*@
  requires 0 <= a < 100;
  requires b < a;
*/
void foo(int a, int b){

}
```

Preconditions are properties about the input (and eventually about global variables) that we assume to be true when we analyze the function. We will verify that they are indeed true only at program points where the function is called.

In this small example, we can also notice a difference with C in the writing of boolean expressions. If we want to specify that `a` is between 0 and 100, we do not have to write `0 <= a && a < 100`, we can directly write `0 <= a < 100` and Frama-C will perform necessary translations.

If we come back to our example about the absolute value, to avoid the arithmetic underflow, it is sufficient to state that `val` must be strictly greater than `INT_MIN` to guarantee that the underflow will never happen. We then add it as a precondition of the function (notice that it is also necessary to include the header where `INT_MIN` is defined):

```
#include <limits.h>

/*@
  requires INT_MIN < val;

  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val) &&
          (val < 0 ==> \result == -val);
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

Warning

Reminder: The Frama-C GUI does not allow code source modification.

Information

For Frama-C NEON and older, the pre-processing of annotations is not activated by default. We have to start Frama-C with the option `-pp-annot`:

```
$ frama-c-gui -pp-annot file.c
```

Once we have modified the source code with our precondition, we click on "Reparse" and we can ask again to prove our program. This time, everything is validated by WP, our implementation is proved (see Figure 3.7).

```
/*@ requires val > -2147483647 - 1;
    ensures positive_value: function_result: \result ≥ 0;
    ensures
      (\old(val) ≥ 0 ⇒ \result ≡ \old(val)) ∧
      (\old(val) < 0 ⇒ \result ≡ -\old(val));
 */
int abs(int val)
{
  int __retres;
  if (val < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ val; */
    __retres = - val;
    goto return_label;
  }
  __retres = val;
  return_label: return __retres;
}
```

Figure 3.7: Proof of `abs` performed

We can also verify that a function that would call `abs` correctly respects the required precondition:

```
void foo(int a){
   int b = abs(42);
   int c = abs(-42);
   int d = abs(a);       // False : "a" can be INT_MIN
   int e = abs(INT_MIN); // False : the parameter must be strictly greater than INT_MIN
}
```

(The result is presented in Figure 3.8).

```
void foo(int a)
{
  int b;
  int c;
  int d;
  int e;
  b = abs(42);
  c = abs(-42);
  d = abs(a);
  e = abs(-2147483647 - 1);
  return;
}
```

Figure 3.8: Precondition checking when calling `abs`

We can modify this example by revering the last two instructions. If we do this, we can see that the call `abs(a)` is validated by WP if it is placed after the call `abs(INT_MIN)`! Why?

We must keep in mind that the idea of the deductive proof is to ensure that if preconditions are verified, and if our computation terminates, then the post-conditon is verified.

If we give a function that surely breaks the precondition, we can deduce that the postconditon is false. Knowing this, we can prove absolutely everything because this "false" becomes an assumption of every call that follows. Knowing false, we can prove everything, because if we have a proof of false, then false is true, as well as true is true. So everything is true.

Taking our modified program, we can convince ourselves of this fact by looking at proof obligations generated by WP for the bad call and the subsequent call that becomes verified (Figures 3.9 and 3.10).

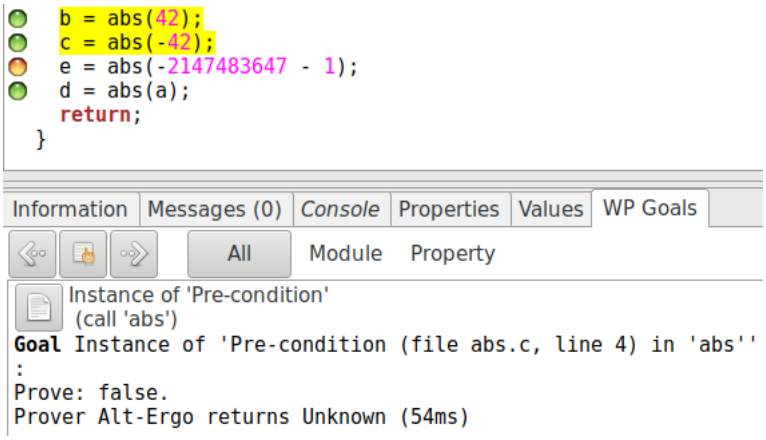


Figure 3.9: Generated proof obligation for the bad call

We can notice that for function calls, the GUI highlights the execution path that leads to the call for which we want to verify the preconditon. Then, if we have a closer look to the call `abs(INT_MIN)`, we can notice that, simplifying, Qed deduced that we try to prove "False". Consequently, the next call `abs(a)` receives in its assumptions the property "False". This is why Qed can immediately deduce "True".

The second part of the question is then: why our first version of the calling function (`abs(a)` and then `abs(INT_MIN)`) did not have the same behavior, indicating a proof failure on the second call? The answer is simply that the call `abs(a)` can, or not, produce an error, whereas `abs(INT_MIN)` necessarily leads to an error. So, while `abs(INT_MIN)` necessarily gives us the knowledge of "false", the call `abs(a)` does not, since it can succeed.

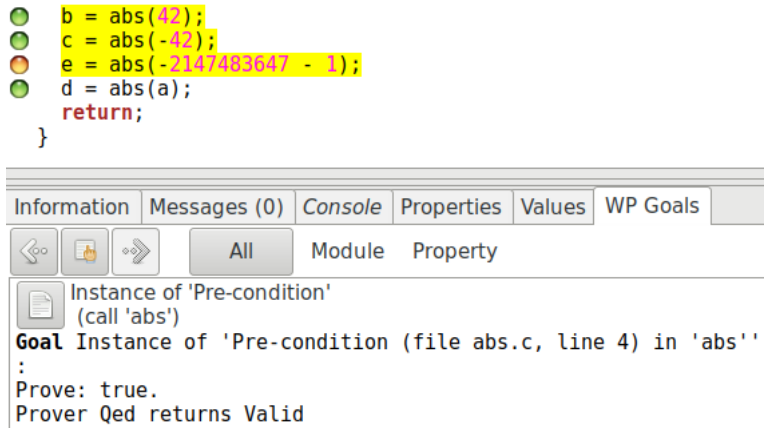


Figure 3.10: Generated proof obligation for the call that follows

Produce a correct specification is then crucial. Typically, by stating false precondition, we can have the possibility to create a proof of false:

```
/*@
  requires a < 0 && a > 0;
  ensures  \false;
*/
void foo(int a){

}
```

If we ask WP to prove this function, it will accept it without a problem since the assumption we give in precondition is necessarily false. However, we will not be able to give an input that respects the precondition so we will be able to detect this problem by carefully reading what we have specified.

Some notions we will see in this tutorial can expose us to the possibility to introduce subtle incoherence. So, we must always be careful specifying a program.

3.1.2.1 Finding the right preconditions

Finding the right preconditions for a function is sometimes hard. The most important idea is to determine these preconditions without taking in account the content of the function (at least, in a first step), in order to avoid building a specification that would contain the same bugs currently existing in the source code, for example taking in account an erroneous conditional structure. In fact, it is generally a good practice to work with someone else. One specifies the function and the other implements it (even if they previously agreed on a common textual specification).

Once these precondition has been stated, then we work on the specifications that are due to the constraints of our language and our hardware. For example, the absolute value do not really have a precondition, this is our hardware that adds the condition we have given in precondition due to the two's complement on which it relies.

3.1.3 Some elements about the use of WP and Frama-C

In the two preceding sections, we have seen a lot of notions about the use of the GUI to start proofs. In fact, we can ask WP to immediately prove everything at Frama-C's startup with the option `-wp`:

```
$ frama-c-gui file.c -wp
```

Which will collect all properties to be proved inside `file.c`, generate all proof obligations and try to discharge them.

About runtime-errors, it is generally advised to first verify the program without generating RTE assertions, and then to generate them to terminate the verification with WP. It allows WP to "focus" on the functional properties in a first step without having in its knowledge base purely technical properties, that are generally not useful for the proof of functional properties. Again, it is possible to directly produce this behavior using the command line:

```
$ frama-c-gui file.c -wp -then -rte -wp
```

"Start Frama-C with WP, then create assertions to verify the absence of RTE and start WP again".

3.2 Well specified function

3.2.1 Correctly write what we expect

This is certainly the hardest part of our work. Programming is already an effort that consists in writing algorithms that correctly respond to our need. Specifying request the same kind of work, except that we do not try to express *how* we respond to the need but *what* is exactly our need. To prove that our code implements what we need, we must be able to describe exactly what we need.

From now, we will use an other example, the `max` function:

```
int max(int a, int b){
  return (a > b) ? a : b;
}
```

The reader could write and prove their own specification. We will start using this one:

```
/*@
  ensures \result >= a && \result >= b;
*/
int max(int a, int b){
  return (a > b) ? a : b;
}
```

If we ask WP to prove this code, it will succeed without any problem. However, is our specification really correct? We can try to prove this calling code:

```
void foo(){
  int a = 42;
  int b = 37;
  int c = max(a,b);

  //@assert c == 42;
}
```

There, it will fail. In fact, we can go further by modifying the body of the `max` function and notice that the following code is also correct with respect to the specification:

```
#include <limits.h>

/*@
  ensures \result >= a && \result >= b;
*/
int max(int a, int b){
  return INT_MAX;
}
```

Our specification is too permissive. We have to be more precise. We do not only expect the result to be greater or equal to both parameters, but also that the result is one of them:

```
/*@
  ensures \result >= a && \result >= b;
  ensures \result == a || \result == b;
*/
int max(int a, int b){
  return (a > b) ? a : b;
}
```

3.2.2 Pointers

If there is one notion that we permanently have to confront with in C language, this is definitely the notion of pointer. Pointers are quite hard to manipulate correctly, and they still are the main source of critical bugs in programs, so they benefit of a preferential treatment in ACSL.

We can illustrate with a swap function for C integers:

```
/*@
  ensures *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

3.2.2.1 History of values in memory

Here, we introduce a first built-in logic function of ACSL: `old`, that allows us to get the old value of a given element. So, our specification defines that the function must ensure that after the call, the value of `*a` is the old (that is to say, before the call) value of `*b` and conversely.

The `\old` function can only be used in the post-condition of a function. If we need this type of information somewhere else, we will use `at` that allows us to express that we want the value of a variable at a particular program point. This function receives two parameters. The first one is the variable (or memory location) for which we want to get its value and the second one is the program point (as a C label) that we want to consider.

For example, we could write:

```
  int a = 42;
 Label_a:
  a = 45;

  //@assert a == 45 && \at(a, Label_a) == 42;
```

Of course, we can use any C label in our code, but we also have 6 built-in labels defined by ACSL that can be used, however WP does not support all of them currently:

- `Pre`/`Old` : value before function call,
- `Post` : value after function call,
- `LoopEntry` : value at loop entry (not supported yet),
- `LoopCurrent` : value at the beginning of the current step of the loop (not supported yet),
- `Here` : value at the current program point.

Information

The behavior of `Here` is in fact the default behavior when we consider a variable. Its use with `at` with generally let us ensure that what we write is not ambigous, and is more readable, when we express properties about values at different program points in the same expression.

Whereas `\old` can only be used in function post-conditions, `\at` can be used anywhere. However, we cannot use any program point with respect to the type annotation we are writing. `Old` and `Post` are only available in function post-conditions, `Pre` and `Here` are available everywhere. `LoopEntry` and `LoopCurrent` are only available in the context of loops (which we will detail later in this tutorial).

At the moment, we will not need `\at` but it can often be useful, if not essential, when we want to make our specification precise.

3.2.2.2 Pointers validity

If we try to prove that the swap function is correct (comprising RTE verification), our post-condition is indeed verified but WP failed to prove some possibilities of runtime-error, since we perform access to some pointers that we did not indicate to be valid pointers in the precondition of the function.

We can express that the dereferencing of a pointer is valid using the `\valid` predicate of ACSL which receives the pointer in input:

```
/*@
  requires \valid(a) && \valid(b);
  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

Once we have specified that the pointers we receive in input are valid, dereferencing is assured to not produce undefined behaviors.

As we will see later in this tutorial, `\valid` can take more than one pointer in parameter. For example, we can give it an expression such as: `valid(p + (s .. e))` which means "for all `i` between included `s` and `e`, `p+i` is a valid pointer. This kind of expression will be extremely useful when we will specify properties about arrays in specifications.

If we have a closer look to the assertions that WP adds in the swap function comprising RTE verification, we can notice that there exists another version of the `\valid` predicate, denoted

`\valid_read`. As opposed to `valid`, the predicate `\valid_read` indicates that a pointer can be dereferenced, but only to read the pointed memory. This subtlety is due to the C language, where the downcast of a const pointer is easy to write but is not necessarily legal.

Typically, in this code:

```
/*@ requires \valid(p); */
int unref(int* p){
  return *p;
}

int const value = 42;

int main(){
  int i = unref(&value);
}
```

Dereferencing `p` is valid, however the precondition of `unref` will not be verified by WP since dereferencing `value` is only legal for a read-access. A write access will result in an undefined behavior. In such a case, we can specify that the pointer `p` must be `\valid_read` and not `\valid`.

3.2.2.3 Side effects

Our `swap` function is provable with regard to the specification and potential runtime errors, but is it however specified enough? We can slightly modify our code to check this (we use `assert` to verify some properties at some particular points):

```
int h = 42; //we add a global variable

/*@
  requires \valid(a) && \valid(b);
  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(){
  int a = 37;
  int b = 91;

  //@ assert h == 42;
  swap(&a, &b);
  //@ assert h == 42;
}
```

The result is not exactly what we expect, as we can see in Figure 3.11.

Indeed, we did not specify the allowed side effects for our function. In order to specify side effects, we use an `assign` clause which is part of the postcondition of a function. It allows us to specify which **non local** elements (we verify side effects) can be modified during the execution of the function.

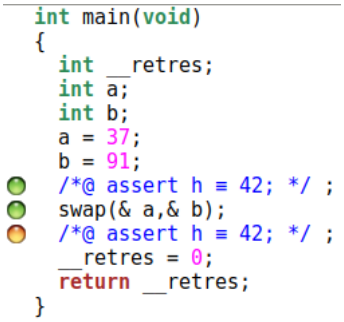


Figure 3.11: Proof failure on the property of a global variable which is not modified by `swap`

By default, WP considers that a function can modify everything in the memory. So, we have to specify what can be modified by a function. For example, our `swap` function will be specified like this:

```
/*@
  requires \valid(a) && \valid(b);

  assigns *a, *b;

  ensures  *a == \old(*b) && *b == \old(*a);
*/
void swap(int* a, int* b){
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

If we ask WP to prove the function with this specification, it will be validated (including with the variable added in the previous source code).

Finally, we sometimes want to specify that a function is side effect free. We specify this by giving `\nothing` to `assigns`:

```
/*@
  requires \valid_read(a) && \valid_read(b);

  assigns  \nothing;

  ensures \result == *a || \result == *b;
  ensures \result >= *a && \result >= *b;
*/
int max_ptr(int* a, int* b){
  return (*a > *b) ? *a : *b ;
}
```

The careful reader will now be able to take back the examples we presented until now to integrate the right `assigns` clause.

3.2.2.4 Memory location separation

Pointers bring the risk of aliasing (multiple pointers can have access to the same memory location). For some functions, it will not cause any problem, for example when we give two identical pointers to the `swap` function, the specification is still verified. However, sometimes it is not that simple:

```
#include <limits.h>

/*@
  requires \valid(a) && \valid_read(b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b;
  ensures  *b == \old(*b);
*/
void incr_a_by_b(int* a, int const* b){
  *a += *b;
}
```

If we ask WP to prove this function, we get the result illustrated in Figure 3.12.

```
/*@ requires \valid(a) ∧ \valid_read(b);
    ensures *\old(a) ≡ \old(*a) + *\old(b);
    ensures *\old(b) ≡ \old(*b);
    assigns *a;
 */
void incr_a_by_b(int *a, int const *b)
{
  *a += *b;
  return;
}
```

Figure 3.12: Proof failure: potential aliasing

The reason is simply that we do not have any guarantee that the pointer `a` is different of the pointer `b`. Now, if these pointers are the same,

- the property `*a == \old(*a) + *b` in fact means `*a == \old(*a) + *a` which can only be true if the old value pointed by `a` was 0, and we do not have such a requirement,
- the property `*b == \old(*b)` is not validated because we potentially modify this memory location.

Why is the `assign` clause validated ?

The reason is simply that `a` is indeed the only modified memory location. If `a != b`, we only modify the location pointed by `a`, and if `a == b`, | that is still the case: `b` is not another location.

In order to ensure that pointers address separated memory locations, ACSL gives use the predicate `\separated(p1, ...,pn)` that receives in parameter a set of pointers and that ensures that these pointers are non-overlapping. Here, we specify:

```
#include <limits.h>

/*@
  requires \valid(a) && \valid_read(b);
  requires \separated(a, b);
  assigns  *a;
  ensures  *a == \old(*a)+ *b;
  ensures  *b == \old(*b);
*/
void incr_a_by_b(int* a, int const* b){
  *a += *b;
}
```

```
/*@ requires \valid(a) ∧ \valid_read(b);
    requires \separated(a, b);
    ensures *\old(a) ≡ \old(*a) + *\old(b);
    ensures *\old(b) ≡ \old(*b);
    assigns *a;
 */
void incr_a_by_b(int *a, int const *b)
{
  *a += *b;
  return;
}
```

Figure 3.13: Solved aliasing problems

And this time, the function is verified, as we can see in Figure 3.13.

We can notice that we do not consider the arithmetic overflow here, as we do not focus on this question in this section. However, if this function was part of a complete program, it would be necessary to define the context of use of this function and the precondition guaranteeing the absence of overflow.

3.3 Behaviors

Sometimes, a function can have behaviors that can be quite different depending on the input. Typically, a function can receive a pointer to an optional resource: if the pointer is `NULL`, we will have a certain behavior, which will be different of the behavior expected when the pointer is not `NULL`.

We have already seen a function that have different behaviors: the `abs` function. We will use it again to illustrate behaviors. We have two behaviors for the `abs` function: either the input is positive or it is negative.

Behaviors allow us to specify the different cases for postconditions. We introduce them using the `behavior` keyword. Each behavior will have a name, the assumptions we have for the given case introduced with the clause `assumes` and the associated postcondition. Finally, we can ask WP to verify that behaviors are disjoint (to guarantee determinism) and complete.

Behaviors are disjoint if for any (valid) input of the function, it corresponds to the assumption (`assumes`) of a single behavior. Behaviors are complete if any (valid) input of the function corresponds to at least one behavior.

For example, for `abs` we can write the specification presented in Figure 3.14.

It can be useful to experiment two possibilities to understand the exact meaning of `complete` and `disjoint`:

- replace the assumption of `pos` with `val > 0`, in this case, behaviors will be disjoint but incomplete (we will miss `val == 0`),
- replace the assumption of `neg` with `val <= 0`, in this case, behaviors will be complete but not disjoint (we will have two assumptions corresponding to `val == 0`.

```
/*@
  requires val > INT_MIN;
  assigns  \nothing;

  behavior pos:
    assumes 0 <= val;
    ensures \result == val;

  behavior neg:
    assumes val < 0;
    ensures \result == -val;

  complete behaviors;
  disjoint behaviors;
*/
int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

Figure 3.14: A specification using behaviors for `abs`

Warning

Even if `assigns` is a postcondition, indicating different assigns in each behavior is currently not well-handled by WP. If we need to specify this, we will:

- put our `assigns` before the behaviors (as we have done in our example) with all potentially modified non-local element,
- add in post-condition of each behaviors the elements that are in fact not modified by indicating their new value to be equal to the `\old` one.

Behaviors are useful to simplify the writing of specifications when functions can have very different behaviors depending on their input. Without them, specification would be defined using implications expressing the same idea but harder to write and read (which would be error-prone).

On the other hand, the translation of completeness and disjointness would be necessarily written by hand which would be tedious and again error-prone.

3.4 WP Modularity

The end of this part will be dedicated to function call composition, where we will start to have a closer look to WP. We will also have a look to the way we can split our programs in different files when we want to prove them using WP.

Our goal will be to prove the `max_abs` function, that return the maximum absolute value of two values:

```
int max_abs(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

  return max(abs_a, abs_b);
}
```

Let us start by (over-)splitting the function we already proved in pairs header/source for `abs` and `max`. We will obtain, for `abs` the files presented in Figure 3.15 and 3.16.

We can notice that we put our function contract inside the header file. The goal is to be able to import the specification at the same time than the declaration when we need it in another file. Indeed, WP will need it to be able to prove that the precondition of the function is verified when we call it.

We can create a file using the same format for the `max` function. In both cases, we can open the source file (we do not need to specify header files in the command line) with Frama-C and notice that the specification is indeed associated to the function and that we prove it.

Now, we can prepare our files for the `max_abs` function with the header in Figure 3.17 and its source file in Figure 3.18.

We can open the source file in Frama-C. If we look at the side panel, we can see that the header files we have included in `abs_max` correctly appear and if we look at the function contracts for them, we can see some blue and green bullets as illustrated by Figure 3.19.

```
#ifndef _ABS
#define _ABS

#include <limits.h>

/*@
  requires val > INT_MIN;
  assigns  \nothing;

  behavior pos:
    assumes 0 <= val;
    ensures \result == val;

  behavior neg:
    assumes val < 0;
    ensures \result == -val;

  complete behaviors;
  disjoint behaviors;
*/
int abs(int val);

#endif
```

Figure 3.15: Header for `abs`

```
#include "abs.h"

int abs(int val){
  if(val < 0) return -val;
  return val;
}
```

Figure 3.16: Source file for `abs`

```
#ifndef _MAX_ABS
#define _MAX_ABS

int max_abs(int a, int b);

#endif
```

Figure 3.17: Header for `max_abs`

```
#include "max_abs.h"
#include "max.h"
#include "abs.h"

int max_abs(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

  return max(abs_a, abs_b);
}
```

Figure 3.18: Source file for `max_abs`

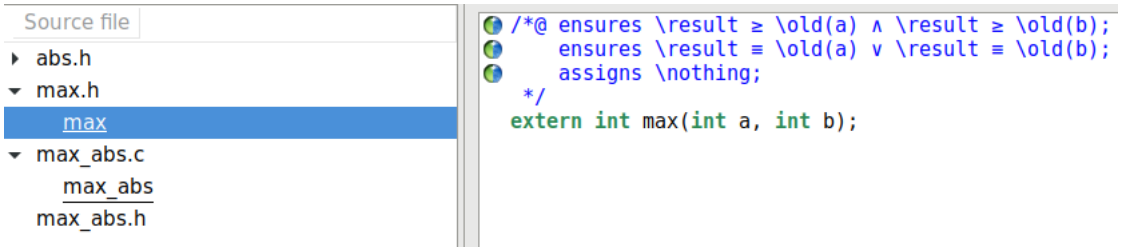


Figure 3.19: The contract of `max` is assumed to be valid

These bullets indicate that, since we do not have the implementation, they are assumed to be true. It is an important strength of the deductive proof of programs compared to some other formal methods: function are verified isolated from each other.

When we are not currently performing the proof of a function, its specification is considered to be correct: we do not try to prove it when we are proving another function, we will only verify that the precondition is correctly established when we call it. It provides very modular proofs and specifications that are therefore more reusable. Of course, if our proof rely on the specification of another function, it must be provable to ensure that the proof of the program is complete. But, we can also consider that we trust a function that comes from an external library that we do not want to prove (or for which we do not even have the source code).

The careful reader could specify and prove the `max_abs` function.

A solution is provided there (we also add the implementation as a reminder):

```
/*@
  requires a > INT_MIN;
  requires b > INT_MIN;

  assigns \nothing;

  ensures \result >= 0;
  ensures \result >= a && \result >= -a && \result >= b && \result >= -b;
  ensures \result == a || \result == -a || \result == b || \result == -b;
*/
int abs_max(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

  return max(abs_a, abs_b);
}
```

During this part of the tutorial, we have studied how we can specify functions using contracts, composed of a pre and a post-condition, as well as some features ACSL provides to express those properties. We have also seen why it is important to be precise when we specify and how the introduction of behaviors can help us to write more understandable and concise specification.

However, we do not have studied one important point: the specification of loops. Before that, we should have a closer look to the way WP works.

4 Basic instructions and control structures

Information

This part is more formal than what we have seen so far. If the reader wishes to concentrate on the usage of the tool, he can skip the introduction and the first two sections (about the basic instructions and the bonus training) of this chapter. If what we presented so far has been difficult for the reader from a formal point of view, it is well possible to reserve the introduction and the two sections for a later reading.
The sections on loops, however, are indispensable. We will highlight the more formal parts of these sections.

We will associate with every C programming construct

- the corresponding inference rule together,
- its governing rule from the weakest precondition calculus and
- examples that show its usage.

Not necessarily in that order and sometimes only with a loose connection to the tool. Since the first rules are quite simple, we will discuss them in a fairly theoretical manner. Later on, however, our presentation will rely more and more on the tool, in particular when we begin dealing with loops.

4.0.1 Inference rules

An inference rule is of the form

$$\frac{P_1 \quad \ldots \quad P_n}{C}$$

and means that in order to assure that the conclusion C is true, first the truth of the premises P_1, ..., and P_n has to be established. In case that the rule has no premises

$$\frac{}{C}$$

then nothing has to be assured in order to conclude the truth of C.

On the other hand, in order to prove that a certain premise is true, it might be necessary to employ other inference rules which would lead to something like this:

$$\frac{\dfrac{}{P_1} \qquad \dfrac{P_{n_1} \quad P_{n_2}}{P_n}}{C}$$

This way, we obtain step by step the *deduction tree* of our reasoning. In our case, the premises and conclusions under consideration will in general be *Hoare triples*.

4.0.2 Hoare triples

We are now returning to concept of a Hoare triple:

$$\{P\} \quad C \quad \{Q\}$$

In the beginning of this tutorial we have seen that this triple expresses the following: if the property P holds before the execution of C and if C terminates, then the property Q holds too. For example, if we take up again our (slightly modified) program for the computation of the absolute value:

```
/*@
  ensures \result >= 0;
  ensures (val >= 0 ==> \result == val ) && (val <  0 ==> \result == -val);
*/
int abs(int val){
  int res;
  if(val < 0) res = - val;
  else        res = val;

  return res;
}
```

The rules of Hoare logic tell us that in order to show that our program satisfies its contract we have to verify the properties shown in the braces. (We have omitted one postcondition in order to simplify the presentation.)

```
int abs(int val){
  int res;
// { P }
  if(val < 0){
// {  (val < 0) && P }
    res = - val;
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }
  } else {
// { !(val < 0) && P }
    res = val;
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }
  }
// { \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val }

  return res;
}
```

Yet, Hoare logic does not tell us how we can automatically obtain the property `P` of our program `abs`. Dijkstra's *weakest-precondition calculus*, on the other hand, allows us to compute from a given postcondition Q and a code snippet C the minimal precondition P that ensures Q after the execution of C. We are thus in a position to determine for our example `abs` the desired property `P`.

In this chapter we present the different cases of the function wp which, starting from a given postcondition and a program (or statement), computes the *weakest* precondition that allows us to establish the validity of the postcondition. We will use the following notation to define the computation that corresponds to one ore several statements:

$wp(Instruction(s), Post) := WeakestPrecondition$

The function wp will guarantee that the Hoare triple

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

really is a valid triple.

We will thereby often use ACSL assertions in order to represent the upcoming concepts:

```
//@ assert my_property;
```

These assertions correspond in fact to possible intermediate steps for the properties in our Hoare triples. We can, for example, replace the properties of our function `abs` by corresponding ACSL assertions (we have omitted here the property `P` because it is just `true`):

```
int abs(int val){
  int res;
  if(val < 0){
    //@ assert val < 0 ;
    res = - val;
    //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
  } else {
    //@ assert !(val < 0) ;
    res = val;
    //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
  }
  //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;

  return res;
}
```

4.1 Assignment, sequence and conditional

4.1.1 Assignment

Assignment is the most basic operation one can have in language (leaving aside the "do nothing" operation that is not particularly interesting). The weakest precondition calculus associates the following computation to an assignment operation;

$$wp(x = E, Post) := Post[x \leftarrow E]$$

Here the notation $P[x \leftarrow E]$ means "the property P where x is replaced by E". In our case this corresponds to "the postcondition $Post$ where x is replaced by E". The idea is that the postcondition of an assignment of E to x can only be true if replacing all occurrences of x in the formula by E is true. For example:

```
// { P }
x = 43 * c ;
// { x = 258 }
```

$$P = wp(x = 43 * c, \{x = 258\}) = \{43 * c = 258\}$$

The function wp allows us to compute as weakest precondition of the the assignment the formula $\{43 * c = 258\}$, thus obtaining the following Hoare triple:

```
// { 43*c = 258 }
x = 43 * c ;
// { x = 258 }
```

In order to compute the precondition of the assignment we have replaced each occurrence of x in the postcondition by the assigned value $E = 43 * c$. If our program were of the form:

```
int c = 6 ;
// { 43*c = 258 }
x = 43 * c ;
// { x = 258 }
```

we could submit the formula " $43 * 6 = 258$ " to our automatic prover in order to determine whether it is really valid. The answer would of course be "yes" because the property is easy to verify. If we had, however, given the value 7 to the variable `c` the prover's reply would be "no" since the formula $43 * 7 = 258$ is not true.

Taking into account the weakest precondition calculus, we can now write the inference rule for the Hoare triple of an assignment as

$$\overline{\{Q[x \leftarrow E]\} \quad x = E \quad \{Q\}}$$

We note that there is no precondition to verify. Does this mean that the triple is necessarily true? Yes. However, it does not mean that the precondition is respected by the program to which the assignment belongs or that the precondition is at all possible. Here the automatic provers come into play.

For example, we can ask Frama-C to verify the following line

```
int a = 42;
//@ assert a == 42;
```

which is, of course, directly proven by Qed, since it is a simple applications of the assignment rule.

Information

We remark that according to the C standard, an assignment is in fact an expression. This allows us, for example, to write `if( (a = foo()) == 42)`. In Frama-C, an assignment will always be treated as a statement. Indeed, if an assignment occurs within a larger expression, then the Frama-C preprocessor, while building the abstract syntax tree, systematically performs a *normalization step* that produces a separate assignment statement.

4.1.2 Composition of statements

For a statement to be valid, its precondition must allow us by means of executing the said statement to reach the desired postcondition. Now we would like to execute several statements one after another. Here the idea is that the postcondition of the first statement is compatible with the required precondition of the second statement and so on for the third statement.

The inference rule that corresponds to this idea utilizes the following Hoare triples:

$$\frac{\{P\}\ \ S1\ \ \{R\}\quad \{R\}\ \ S2\ \ \{Q\}}{\{P\}\ \ S1;\ S2\ \ \{Q\}}$$

In order to verify the composed statement $S1;\ S2$ we rely on an intermediate property R that is at the same time the postcondition of $S1$ and the precondition of $S2$. (Please note that $S1$ and $S2$ are not necessarily simple statements; they themselves can be composed statements.) The problem is, however, that nothing indicates us how to determine the properties P and R.

The weakest-precondition calculus now says us that the intermediate property R can be computed as the weakest precondition of the second statement. The property P, on the other hand, then is computed as the weakest precondition of the first statement. In other words, the weakest precondition of the composed statement $S1; S2$ is determined as follows:

$$wp(S1;\ S2, Post) := wp(S1, wp(S2, Post))$$

The WP plugin of Frama-C performs all these computations for us. Thus, we do not have to write the intermediate properties as ACSL assertions between the lines of codes.

```
int main(){
  int a = 42;
  int b = 37;

  int c = a+b; // i:1
  a -= c;      // i:2
  b += a;      // i:3

  //@assert b == 0 && c == 79;
}
```

4.1.2.1 Proof tree

When we have more than two statements, we can consider the last statement as second statement of our rule and all the preceding ones as first statement. This way we traverse step by step backwards the statements in our reasoning. With the previous program this looks like:

$$\frac{\dfrac{\{P\}\ \ i_1;\ \ \{Q_{-2}\}\quad \{Q_{-2}\}\ \ i_2;\ \ \{Q_{-1}\}}{\{P\}\ \ i\_1;\ \ i\_2;\ \ \{Q_{-1}\}}\qquad \{Q_{-1}\}\ \ i_3;\ \ \{Q\}}{\{P\}\ \ i\_1;\ \ i\_2;\ \ i\_3;\ \ \{Q\}}$$

The weakest-precondition calculus allows us to construct the property Q_{-1} starting from the property Q and statement i_3 which in turn enables us to derive the property Q_{-2} from the property Q_{-1} and statement i_2. Finally, P can be determined from Q_{-2} and i_1.

Now that we can verify programs that consists of several statements it is time to add some structure to them.

4.1.3 Conditional rule

For a conditional statement to be true, one must be able to reach the postcondition through both branches. Of course, for both branches the same precondition (of the conditional statement) must hold. In addition we have that in the if-branch the condition is true while in the else-branch it is false.

We therefore have, as in the case of composed statements, two facts to verify (in order to avoid confusion we are using here the syntax $if\ B\ then\ S1\ else\ S2$):

$$\frac{\{P \wedge B\}\quad S1\quad \{Q\} \qquad \{P \wedge \neg B\}\quad S2\quad \{Q\}}{\{P\}\quad if\quad B\quad then\quad S1\quad else\quad S2\quad \{Q\}}$$

Our two premises are therefore that we can both in the if-branch and the else-branch reach the postcondition from the precondition.

The result of the weakest-precondition calculus for a conditional statement reads as follows:

$$wp(if\ B\ then\ S1\ else\ S2, Post) := (B \Rightarrow wp(S1, Post)) \wedge (\neg B \Rightarrow wp(S2, Post))$$

This means that the condition B has to imply the weakest precondition of $S1$ in order to safely arrive at the postcondition. Analogously, the negation of B must imply the weakest precondition of $S2$.

4.1.3.1 Empty else-branch

Following this definition we obtain for case of an empty else-branch the following rule by simply replacing the statement $S2$ by the empty statement `skip`.

$$\frac{\{P \wedge B\}\quad S1\quad \{Q\} \qquad \{P \wedge \neg B\}\quad skip\quad \{Q\}}{\{P\}\quad if\quad B\quad then\quad S1\quad else\quad skip\quad \{Q\}}$$

The triple for `else` is:

$$\{P \wedge \neg B\}\quad skip\quad \{Q\}$$

which means that we need to ensure:

$$P \wedge \neg B \Rightarrow Q$$

In short, if the condition B of `if` is false, this means that the postcondition of the complete conditional statement is already established before entering the else-branch (since it does not do anything).

As an example, we consider the following code snippet where we reset a variable c to a default value in case it had not been properly initialized by the user.

```
int c;

// ... some code ...

if(c < 0 || c > 15){
  c = 0;
}
//@ assert 0 <= c <= 15;
```

Let

$wp(if \neg(c \in [0;15])\ then\ c := 0, \{c \in [0;15]\})$

$:= (\neg(c \in [0;15]) \Rightarrow wp(c := 0, \{c \in [0;15]\})) \wedge (c \in [0;15] \Rightarrow wp(skip, \{c \in [0;15]\}))$

$= (\neg(c \in [0;15]) \Rightarrow 0 \in [0;15]) \wedge (c \in [0;15] \Rightarrow c \in [0;15])$

$= (\neg(c \in [0;15]) \Rightarrow true) \wedge true$

The property can be verified: independent of the evaluation of $\neg(c \in [0;15])$, the implication will hold.

4.2 [Bonus Stage] Consequence and constancy

4.2.1 Consequence rule

It can sometimes be useful to strengthen a postcondition or to weaken a precondition. If the former will often be established by us to facilitate the work of the prover, the second is more often verified by the tool as the result of computing the weakest precondition.

The inference rule of Hoare logic is the following:

$$\frac{P \Rightarrow WP \quad \{WP\} \quad c \quad \{SQ\} \quad SQ \Rightarrow Q}{\{P\} \quad c \quad \{Q\}}$$

(We remark that the premises here are not only Hoare triples but also formulas to verify.)

For example, if our post-condition is too complex, it may generate a weaker precondition that is, however, too complicated, thus making the work of provers more difficult. We can then create a simpler intermediate postcondition SQ, that is, however, stricter and implies the real postcondition. This is the part $SQ \Rightarrow Q$.

Conversely, the calculation of the precondition will usually generate a complicated and often weaker formula than the precondition we want to accept as input. In this case, it is our tool that will check the implication between what we want and what is necessary for our code to be valid. Th is is the part $P \Rightarrow WP$.

We can illustrate this with the following code. Note that here the code could be proved by WP without the weakening and strengthening of properties because the code is very simple, it is just to illustrate the rule of consequence.

```
/*@
  requires P: 2 <= a <= 8;
  ensures  Q: 0 <= \result <= 100 ;
  assigns  \nothing ;
*/
int constrained_times_10(int a){
  //@ assert P_imply_WP: 2 <= a <= 8 ==> 1 <= a <= 9 ;
  //@ assert WP:         1 <= a <= 9 ;

  int res = a * 10;

  //@ assert SQ:         10 <= res <= 90 ;
  //@ assert SQ_imply_Q: 10 <= res <= 90 ==> 0 <= res <= 100 ;

  return res;
}
```

(Note: We have omitted here the control of integer overflow.)

Here we want to have a result between 0 and 100. But we know that the code will not produce a result outside the bounds of 10 and 90. So we strengthen the postcondition with an assertion that at the end `res`, the result, is between 0 and 90. The calculation of the weakest precondition of this property together with the assignment `res = 10 * a` yields a weaker precondition `1 <= a <= 9` and we know that `2 < = a <= 8` gives us the desired guarantee.

When there are difficulties to carry out a proof on more complex code, then it is often helpful to write assertions that produce stronger, yet easier to verify, postconditions. Note that in the previous code, the lines `P_imply_WP` and`SQ_imply_Q` are never used because this is the default reasoning of WP. They are just here for illustrating the rule.

4.2.2 Constancy rule

Certain sequences of instructions may concern and involve different variables. Thus, we may initialize and manipulate a certain number of variables, begin to use some of them for a time, before using other variables. When this happens, we want our tool to be concerned only with variables that are susceptible to change in order to obtain the simplest possible properties.

The rule of inference that defines this reasoning is the following:

$$\frac{\{P\} \quad c \quad \{Q\}}{\{P \wedge R\} \quad c \quad \{Q \wedge R\}}$$

where c does not modify any input variable in R. In other words: "To check the triple, let's get rid of the parts of the formula that involves variables that are not influence by c and prove the new triple." However, we must be careful not to delete too much information, since this could mean that we are not able to prove our properties.

As an example,let us consider the following code (here gain, we ignore potential integer overflows):

```
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    a++ ;
  } else {
    a += b ;
  }
  return a ;
}
```

If we look at the code of the `if` block, we notice that it does not use the variable `b`. Thus, we can completely omit the properties about `b` in order to prove that `a` will be strictly greater than 0 after the execution of the block:

```
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    //@ assert a >= 0; // no mentioning of b
    a++ ;
  } else {
    a += b ;
  }
  return a ;
}
```

On the other hand, in the `else` block, even if `b` were not modified, formulating properties only about `a` would render a proof impossible for humans. The code would be:

```
/*@
  requires a > -99 ;
  requires b > 100 ;
  ensures  \result > 0 ;
  assigns  \nothing ;
*/
int foo(int a, int b){
  if(a >= 0){
    //@ assert a >= 0; // no mentioning of b
    a++ ;
  } else {
    //@ assert a < 0 && a > -99 ; // no mentioning of b
    a += b ;
  }
  return a ;
}
```

In the `else` block, knowing that`a` lies between `-99` and 0, but knowing nothing about `b`, we could hardly know if the operation `a += b` produces a result that is greater than 0.

The WP plug-in will, of course, prove the function without problems, since it produces by itself the properties that are necessary for the proof. In fact, the analysis which variables are necessary or not (and, consequently, the application of the constancy rule) is conducted directly by WP.

Let us finally remark that the constancy rule is an instance of the consequence rule:

$$\frac{P \wedge R \Rightarrow P \quad \{P\} \quad c \quad \{Q\} \quad Q \Rightarrow Q \wedge R}{\{P \wedge R\} \quad c \quad \{Q \wedge R\}}$$

If the variables of R have not been modified by the operation (which, on the other hand, may modify the variables of P to produce Q), then the properties $P\ wedgeR\ RightarrowP$ and $Q\ RightarrowQ\ wedgeR$ hold.

4.3 Loops

Loops needs a particular treatment in deductive verification of programs. These are the only control structures that will require an important work from us. We cannot avoid this because without loops, it is difficult to write and prove interesting programs.

Before we look at the way we specify loop, we can answer to a rightful question: why are loops so complex?

4.3.1 Induction and invariant

The nature of loops makes their analysis complex. When we perform our backward reasoning, we need a rule to determine from a post-condition, what is the precondition to a given sequence of instructions. Here, the problem is that we cannot *a priori* deduce how many times a loop will iterate, and consequently, we cannot know neither how many times variables will be modified.

we will then proceed using an inductive reasoning. We have to find a property that is true before we start to execute the loop and that, if it is true at the beginning of an iteration, remains true at the end (and that is consequently true at the beginning of the next iteration).

This type of property is called a loop invariant. A loop invariant is a property that must be true before and after each loop instruction. For example with the following loop:

```
for(int i = 0 ; i < 10 ; ++i){ /* */ }
```

The property $0 <= i <= 10$ is a loop invariant. The property $-42 <= i <= 42$ is also an invariant (even if it is far less precise). The property $0 < i <= 10$ is not an invariant because it is not true at the beginning of the execution of the loop. The property $0 <= i < 10$ **is not a loop invariant**, it is not true at the end of the last iteration that sets the value of `i` to 10.

To verify an invariant I, WP will then produce the following "reasoning":

- verify that I is true at the beginning of the loop (establishment)
- verify that if I is true before an iteration, then I is true after (preservation).

4.3.1.1 [Formal] Inference rule

Let us note the invariant I, the inference rule corresponding to loops is defined as follows:

$$\frac{\{I \wedge B\}\, c\, \{I\}}{\{I\}\, while(B)\{c\}\, \{I \wedge \neg B\}}$$

And the weakest precondition calculus is the following:

$$wp(while(B)\{c\}, Post) := I \wedge ((B \wedge I) \Rightarrow wp(c, I)) \wedge ((\neg B \wedge I) \Rightarrow Post)$$

Let us detail this formula:

- (1) the first I corresponds to the establishment of the invariant, in layman's terms, this is the "precondition" of the loop,
- the second part of the conjunction ($(B \wedge I) \Rightarrow wp(c, I)$) corresponds to the verification of the operation performed by the body of the loop:
  - the precondition that we know of the loop body (let us note KWP, "Known WP") is ($KWP = B \wedge I$). That is the fact we have entered the loop (B is true), and that the invariant is verified at this moment (I, is true before we start the loop by (1), and we want to verify that it will be true at the end of the body of the loop in (2)),
  - (2) it remains to verify that KWP implies the actual precondition* of the body of the loop ($KWP \Rightarrow wp(c, Post)$). What we want at the end of the loop is the preservation of the invariant I (B is maybe not true anymore however), formally $KWP \Rightarrow wp(c, I)$, that is to say $(B \wedge I) \Rightarrow wp(c, I)$,
  - * it corresponds to the application of the consequence rule previously explained.
- finally, the last part ($(\neg B \wedge I) \Rightarrow Post$) expresses the fact that when the loop ends($\neg B$), and the invariant I has been maintained, it must imply that the wanted postcondition of the loop is

In this computation, we can notice that the wp function do not indicate any way to obtain the invariant I. We have to specify ourselves this property about our loops.

4.3.1.2 Back to the WP plugin

There exist tools that can infer invariant properties (provided that these properties are simple, automatic tools remains limited). It is not the case for WP. We will have to manually annotate our programs to specify the invariant of each loop. To find and write invariant for our loops will always be the hardest part of our work when we want to prove programs.

Indeed if, when there are no loops, the weakest precondition calculus function can automatically provide the verifiable properties of our programs, it is not the case for loop invariant properties for which we do not have computation procedures. We have to find and express them correctly, and depending on the algorithm, they can be quite subtle and complex.

In order to specify a loop invariant, we add the following annotations before the loop:

```
int main(){
  int i = 0;

  /*@
    loop invariant 0 <= i <= 30;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
}
```

Warning

REMINDER : The invariant is: i <= 30 !

Why? Because along the loop, `i` will be comprised between 0 and **included** 30. 30 is indeed the value that allows us to leave the loop. Moreover, one of the properties required by the weakest precondition calculus is that when the loop condition is invalidated, by knowing the invariant, we can prove the postcondition (Formally $(\neg B \wedge I) \Rightarrow Post$).

The postcondition of our loop is `i == 30` and must be implied by $\neg$ `i < 30` $\wedge$ `0 <= i <= 30`. Here, it is true since: `i >= 30 && 0 <= i <= 30 ==> i == 30`. On the opposite, if we exclude the equality to 30, the postcondition would be unreachable.

Again, we can have a look to the list of proof obligations in "WP Goals" (illustrated by Figure 4.1).

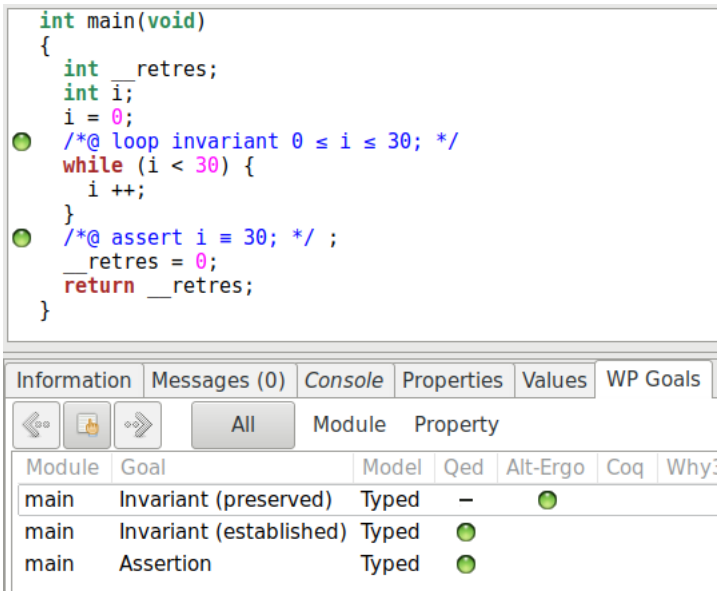


Figure 4.1: Proof obligations generated to verify our loop

We note that WP produces two different proof obligations: the establishment of the invariant and its preservation. WP produces exactly the reasoning we previously described to prove the assertion. In recent versions of Frama-C, Qed has become particularly aggressive and powerful, and the generated proof obligation does not show these details (showing directly "True"). Using the option `-wp-no-simpl` at start, we can however see these details (see Figure 4.2).

But is our specification precise enough?

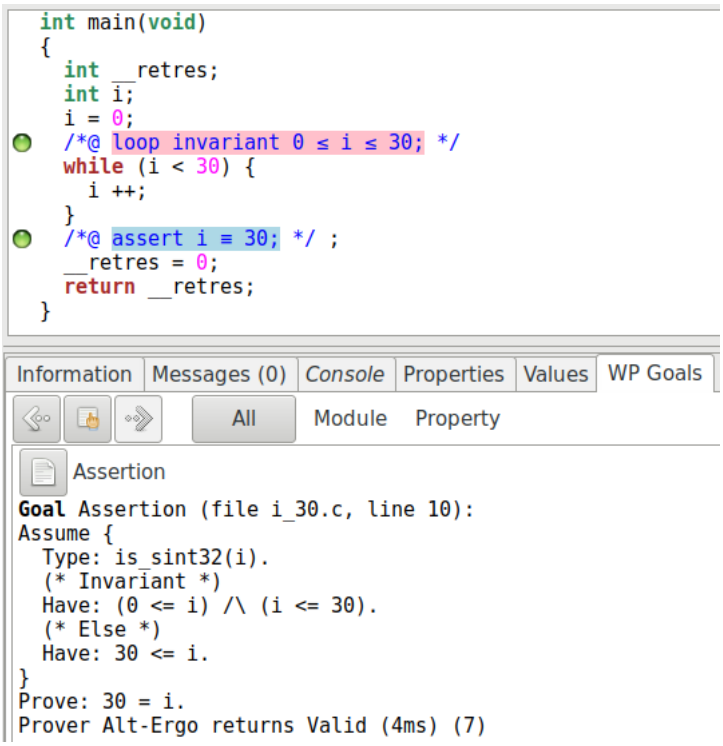


Figure 4.2: Proof of the assertion, knowing the invariant and the invalidation of the loop condition

```
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

And the result is:

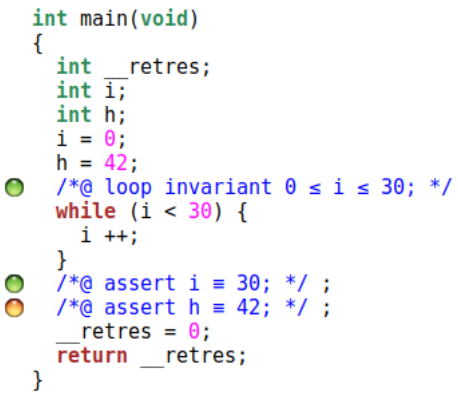


Figure 4.3: Side-effects, again

It seems not (sse Figure 4.3).

4.3.2 The assigns clause ... for loops

In fact, considering loops, WP **only** reasons about what is provided by the user to perform its reasoning. And here, the invariant does not specify anything about the way the value of `h` is modified (or not). We could specify the invariant of all program variables, but it would be a lot of work. ACSL simply allows to add `assigns` annotations for loops. Any other variable is considered to keep its old value. For example:

```
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
    loop assigns i;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

This time, we can establish the proof that the loop correctly behaves. However, we cannot prove that it terminates. The loop invariant is not enough to perform such a proof. For example, in our program, we could modify the loop, removing the loop body:

```
/*@
  loop invariant 0 <= i <= 30;
  loop assigns i;
*/
while(i < 30){

}
```

The invariant is still verified, but we cannot prove that the loop ends: it is infinite.

4.3.3 Partial correctness and total correctness - Loop variant

In deductive verification, we find two types of correctness, the partial correctness and the total correctness. In the first case, the formulation of the correctness property is “if the precondition is valid, and **if** the computation terminates, then the postcondition is valid”. In the second case, “if the precondition is valid, **then** the computation terminates and the postcondition is valid”. By default, WP considers only partial correctness:

```
void foo(){
  while(1){}
  //assert \false;
}
```

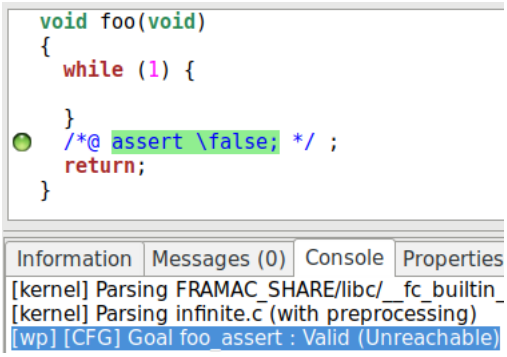


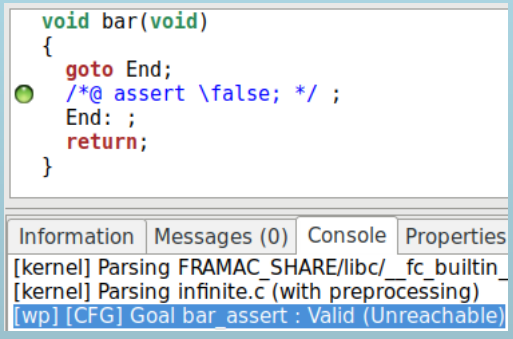
Figure 4.4: Proof of false by non termination

If we try to verify this code activating the verification of absence of RTE, we get the result presented in Figure 4.4.

The assertion "False" is proved! For a very simple reason: since the condition of the loop is "True" and no instruction of the loop body allows to leave the loop, it will not terminate. As we are proving the code with partial correctness, and as the execution does not terminate, we can prove anything about the code that follows the non terminating part of the code. However, if the termination is not trivially provable, the assertion will probably not be proved.

Information

Note that an (provably) unreachable assertion is always proved to be true:



And this is also the case when we trivially know that an instruction produces a runtime error (for example dereferencing `NULL`), or inserting "False" in post-condition as we have already seen with `abs` and the parameter `INT_MIN`.

In order to prove the termination of a loop, we use the notion of loop variant. The loop variant is not a property but a value. It is an expression that involves the element modified by the loop and that provides an upper bound to the number of iteration that have to be executed by the loop at each iteration. Thus, it is an expression greater or equal to 0, and that strictly decreases at each loop iteration (it will also be verified by induction by WP).

If we take our previous example, we add the loop variant with this syntax:

```
int main(){
  int i = 0;
  int h = 42;

  /*@
    loop invariant 0 <= i <= 30;
    loop assigns i;
    loop variant 30 - i;
  */
  while(i < 30){
    ++i;
  }
  //@assert i == 30;
  //@assert h == 42;
}
```

Again, we can have a look to the generated proof obligations (see Figure 4.5).

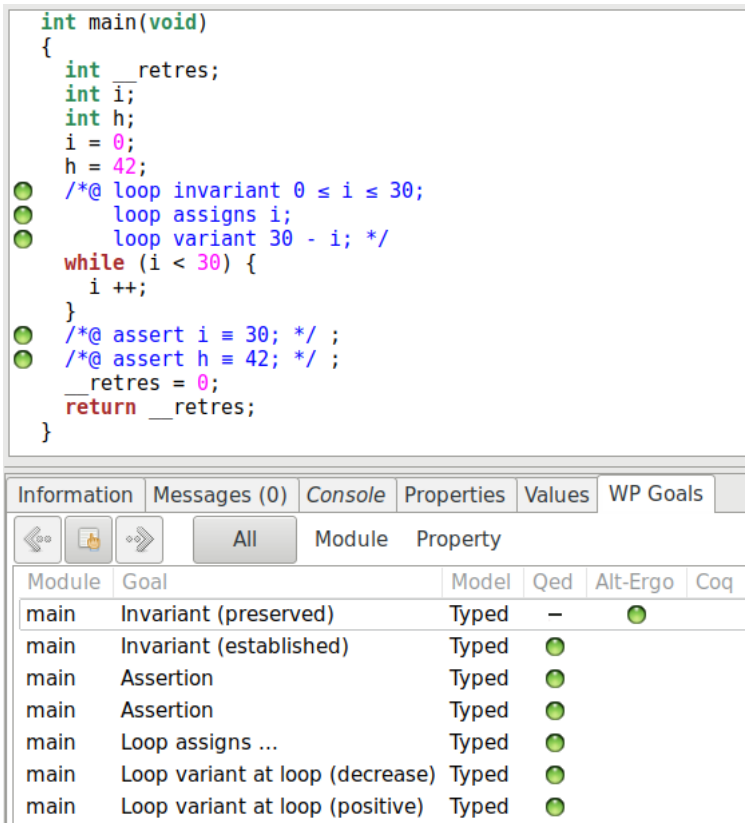


Figure 4.5: Our loop entirely specified and proved

The loop variant generates two proof obligations: verify that the value specified in the variant is positive, and prove that it strictly decreases during the execution of the loop. And if we delete the line of code that increments `i`, WP cannot prove anymore that `30 - i` strictly decreases.

We can also note that being able to give a loop invariant does not necessarily induce that we can give the exact number of remaining iterations of the loop, as we do not always have a so precise knowledge of the behavior of the program. We can for example build an example like this one:

```
#include <stddef.h>

/*@
  ensures min <= \result <= max;
*/
size_t random_between(size_t min, size_t max);

void undetermined_loop(size_t bound){
  /*@
    loop invariant 0 <= i <= bound ;
    loop assigns i;
    loop variant i;
   */
  for(size_t i = bound; i > 0; ){
    i -= random_between(1, i);
  }
}
```

Here, at each iteration, we decrease the value of the variable `i` by a value comprised between 1 and `i`. Thus, we can ensure that the value of `i` is positive and strictly decreases during each loop iteration, but we cannot say how many loop iteration will be executed.

The loop variant is then only an upper bound on the number of iteration, not an expression of their exact number.

4.3.4 Create a link between post-condition and invariant

Let us consider the following specified program. Our goal is to prove that this function returns the old value of `a` plus 10.

```
/*@
    ensures \result == \old(a) + 10;
*/
int add_10(int a){
    /*@
        loop invariant 0 <= i <= 10;
        loop assigns i, a;
        loop variant 10 - i;
    */
    for (int i = 0; i < 10; ++i)
        ++a;

    return a;
}
```

The weakest precondition calculus does not allow to deduce information that is not part of the loop invariant. In a code like the one presented in Figure 4.6, by reading the instructions backward from the postcondition, we always keep all knowledge about `a`. On the opposite, as we previously mentioned, outside the loop, WP only considers the information provided by the invariant. Consequently, our "add_10" function cannot be proved: the invariant does not say anything about `a`. To create a link between the postcondition and the invariant, we have to add this knowledge. See, for example, the code illustrated by Figure 4.7.

```
/*@
    ensures \result == \old(a) + 10;
*/
int add_10(int a){
    ++a;
    ++a;
    ++a;
    //...
    return a;
}
```

Figure 4.6: Unrolled loop

```
/*@
    ensures \result == \old(a) + 10;
*/
int add_10(int a){
    /*@
        loop invariant 0 <= i <= 10;
        loop invariant a = \old(a) + i; //< ADDED
        loop assigns i, a;
        loop variant 10 - i;
    */
    for (int i = 0; i < 10; ++i)
        ++a;

    return a;
}
```

Figure 4.7: Strenghtened invariant

Information

This need can appear as a very strong constraint. This is not really the case. There exists strongly automated analysis that can compute loop invariant properties. For example, without a specification, an abstract interpretation would easily compute `0 <= i <= 10` and `old(a) <= a <= \old(a)+10`. However, it is often more difficult to compute the relations that exists between the different variables of a program, for example the equality expressed by the invariant we have added.

4.4 Loops - Examples

4.4.1 Examples with read-only arrays

Array is the most common data structure when we are working with loops. It is then a good example base to exercise with loops, and these examples allow to rapidly show interesting invariant and will allow us to introduce some important ACSL constructs.

We can for example use the search function that allows to find a value inside an array:

```
#include <stddef.h>

/*@
  requires 0 < length;
  requires \valid_read(array + (0 .. length-1));

  assigns  \nothing;

  behavior in:
    assumes \exists size_t off ; 0 <= off < length && array[off] == element;
    ensures array <= \result < array+length && *\result == element;

  behavior notin:
    assumes \forall size_t off ; 0 <= off < length ==> array[off] != element;
    ensures \result == NULL;

  disjoint behaviors;
  complete behaviors;
*/
int* search(int* array, size_t length, int element){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i ==> array[j] != element;
    loop assigns i;
    loop variant length-i;
  */
  for(size_t i = 0; i < length; i++)
    if(array[i] == element) return &array[i];
  return NULL;
}
```

There are enough ideas inside this example to introduce some important syntax.

First, as we previously presented, the `\valid_read` predicate (as well as `\valid`) allows us to specify not only the validity of read-only address but also to state that a range of contiguous addresses is valid. It is expressed using this expression:

```
//@ requires \valid_read(a + (0 .. length-1));
```

This precondition states that all addresses a+0, a+1, ..., a+length-1 are valid read-only locations.

We also introduced two notations that are used almost all the time in ACSL, the keywords `\forall` (∀) and `\exists` (∃), the universal logic quantifiers. The first allows to state that for any element, some property is true, the second allows to say that there exists some element such that the property is true. If we comment a little bit the corresponding lines in our specification, we can read them this way:

```
/*@
// for all "off" of type "size_t", such that IF "off" is comprised between 0 and "length"
//                                  THEN the cell "off" in "a" is different of "element"
\forall size_t off ; 0 <= off < length ==> a[off] != element;

// there exists "off" of type "size_t", such that "off" is comprise between 0 and "length"
//                                       AND the cell "off" in "a" equals to "element"
\exists size_t off ; 0 <= off < length && a[off] == element;
*/
```

If we want to sum up the use of these keyword, we would say that on a range of values, a property is true, either about at least one of them or about all of them. A common scheme is to constraint this set using another property (here: `0 <= off < length`) and to prove the actual interesting property on this smaller set. **But using `exists` and `forall` is fundamentally different**.

With `\forall type a ; p(a) ==> q(a)`, the constraint `p` is followed by an implication. For all element, if a first property `p` is verified about it, then we have to verify the second property `q`. If we use a conjunction, as we do for "exists" (that we will later explain), that would mean that all element verify both `p` and `q`. Sometimes, it could be what we want to express, but it would then not correspond anymore to the idea of constraining a set for which we want to verify some other property.

With `\exists type a ; p(a) && q(a)`, the constraint `p` is followed by a conjunction. We say there exists an element such that it satisfies the property `p` at the same time it also satisfies `q`. If we use an implication, as we do for "forall", such an expression will always be true if `p` is not a tautology! Why? Is there a "a" such that p(a) implies q(a) ? Let us take a "a" such that p(a) is false, the implication is true.

This part of the invariant deserves a particular attention:

```
//@ loop invariant \forall size_t j; 0 <= j < i ==> array[j] != element;
```

Indeed, it defines the treatment performed by our loop, it indicates to WP what happens inside the loop (or more precisely: what we learn) along the execution. Here, this formula indicates that at each iteration of the loop, we know that for each memory location between 0 and the next location to visit (`i` excluded), the memory location contains a value different of the element we are looking for.

The proof obligation associated to the preservation of this invariant is a bit complex and it is not really interesting to precisely look at it, on the contrary, the proof that the invariant is established before executing the loop is interesting (see Figure 4.8).

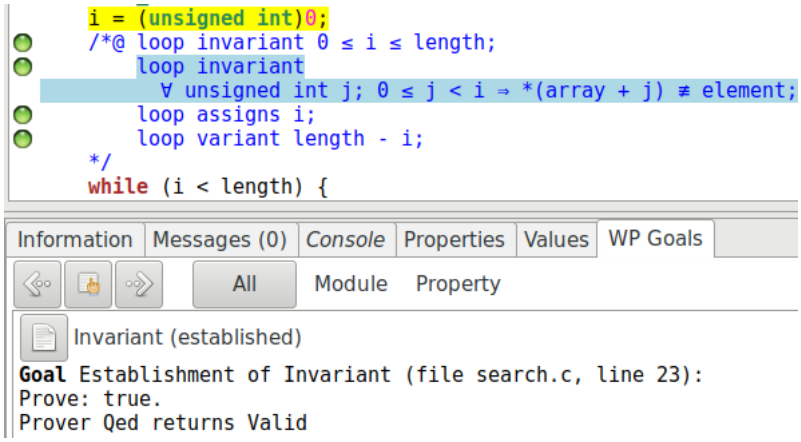


Figure 4.8: Trivial goal

We note that this property, while quite complex, is proved easily proved by Qed. If we look at the parts of the programs on which the proof relies, we can see that the instruction `i = 0` is highlighted and this is, indeed, the last instruction executed on `i` before we start the loop. And consequently if we replace the value of `i` by 0 inside the formula of the invariant, we get:

" For all j, greater or equal to 0 and strictly lower than 0 ", this part of the formula is necessarily false, our implication is then necessarily true.

4.4.2 Examples with mutable arrays

Let us present two examples with mutation of arrays. One with a mutation of all memory locations, the other with selective modifications.

4.4.2.1 Reset

Let us have a look to the function that resets an array of integer to 0 (see Figure 4.9).

Let us just highlight the function and loop assign clauses. Again, we can use the notation `n .. m` to indicate which parts of the array are modified.

4.4.2.2 Search and replace

The last example we will detail to illustrate the proof of functions with loops is the algorithm "search and replace". This algorithms will selectively modify values in a range of memory locations. It is generally harder to guide the tool in such a case, because on one hand we must keep track of what is modified and what is not, and on the other hand, the induction relies on this fact.

As an example, the first specification we can write for this function could be the one illustrated by Figure 4.10.

We use the logic function '@(v, Label)`that gives us the value of the variablevat the program point`Label'. If we look at the usage of this function here, we see that in the invariant we try to establish a relation between the old values of the array and the potentially new values:

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  \forall size_t i; 0 <= i < length ==> array[i] == 0;
*/
void raz(int* array, size_t length){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i ==> array[j] == 0;
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    array[i] = 0;
}
```

Figure 4.9: Function that resets an array

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns array[0 .. length-1];

  ensures \forall size_t i; 0 <= i < length && \old(array[i]) == old
            ==> array[i] == new;
  ensures \forall size_t i; 0 <= i < length && \old(array[i]) != old
            ==> array[i] == \old(array[i]);
*/
void search_and_replace(int* array, size_t length, int old, int new){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                    ==> array[j] == new;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                    ==> array[j] == \at(array[j], Pre);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i){
    if(array[i] == old) array[i] = new;
  }
}
```

Figure 4.10: Specified search and replace function

```
/*@
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                   ==> array[j] == new;
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                   ==> array[j] == \at(array[j], Pre);
*/
```

For memory location, if it contained the value that must be replaced, then it now contains the new value, else the value remains unchanged. In fact, if we try to prove this invariant with WP, it fails. In such a case, the simpler method is to add different assertions that will express the different intermediate properties using assertions, that we expect to be easily proved and that implies the invariant. Here, we can easily notice that WP do not succeed in maintaining the knowledge that we have not modified the end of the array yet:

```
for(size_t i = 0; i < length; ++i){
    //@assert array[i] == \at(array[i], Pre); // proof failure
    if(array[i] == old) array[i] = new;
}
```

We can add this information as an invariant:

```
/*@
  loop invariant 0 <= i <= length;
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                   ==> array[j] == new;
  loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                   ==> array[j] == \at(array[j], Pre);

  // The end of the array remains the same:
  loop invariant \forall size_t j; i <= j < length
                     ==> array[j] == \at(array[j], Pre);
  loop assigns i, array[0 .. length-1];
  loop variant length-i;
*/
for(size_t i = 0; i < length; ++i){
  if(array[i] == old) array[i] = new;
}
```

And this time the proof will succeed. Note that if we try to prove this invariant directly with the verification of the absence of RTE, Alt-Ergo may not succeed (CVC4 succeeds without problem). In this case, we can launch these proofs separately (first without, and then with the absence of RTE checking) or else add assertions that allows to guide the proof inside the loop:

```
for(size_t i = 0; i < length; ++i){
  if(array[i] == old) array[i] = new;

  /*@ assert \forall size_t j; i < j < length
               ==> array[j] == \at(array[j], Pre);                    */
  /*@ assert \forall size_t j; 0 <= j <= i && \at(array[j], Pre) == old
               ==> array[j] == new;                                   */
  /*@ assert \forall size_t j; 0 <= j <= i && \at(array[j], Pre) != old
               ==> array[j] == \at(array[j], Pre);                    */
}
```

As we will try to prove more complex properties, particularly when programs involve loops, there will be a part of “trial and error” in order to understand what the provers miss to establish the proof.

It can miss hypotheses. In this case, we can try to add assertions to guide the prover. With some experience, we can read the content of the proof obligations or try to perform the proof with the Coq interactive prover to see whether the proof seems to be possible. Sometimes, the prover just need more time, in such a case, we can (sometimes a lot) augment the timeout value. Of course, the property can be too hard for the prover, and in this case, we will have to write the proof ourselves with an interactive prover.

Finally, the implementation can be indeed incorrect, and in this case we have to fix it. Here, we will use test and not proof, because a test allows us to prove the presence of a bug and to analyze this bug.

Dans cette partie nous avons pu voir comment se traduisent les affectations et les structures de contrôle d'un point de vue logique. Nous nous sommes beaucoup attardés sur les boucles parce que c'est là que se trouvent la majorité des difficultés lorsque nous voulons spécifier et prouver un programme par vérification déductive, les annotations ACSL qui leur sont spécifiques nous permettent d'exprimer le plus précisément possible leur comportement.

Pour la suite, nous allons nous attarder plus précisément sur les constructions que nous offre le langage ACSL du côté de la logique. Elles sont très importantes parce que ce sont elles qui vont nous permettre de nous abstraire du code pour avoir des spécifications plus compréhensibles et plus aisément prouvables.

5 ACSL - Properties

From the beginning of this tutorial, we have used different predicates and logic functions provided by ACSL : `\valid`, `\valid_read`, `\separated`, `\old` and `at`. There are others built-in predicates but we will not present them all, the reader can refer to the documentation (ACSL implementation) (note that everything is not necessarily supported by WP).

ACSL allows us to do something more than "just" specify our code using existing predicates and functions. We can define our own predicates, functions, relations, etc. Doing this, we can have more abstract specifications. It also allows us to factor specifications (for example defining what is a valid array), which have two pleasant consequences: our specifications are more readable and more understandable, and we can reuse existing proofs to ease the proof of new programs.

5.1 Some logical types

ACSL provides two types that will allow us to write properties or functions without having to take about constraints due to the size of the representation of primitive C types in memory. These types are `integer` and `real`, which respectively represent mathematical integers and reals (that are modeled to be as near the reality we can, but this notion cannot be perfectly handled).

From now, we will often use integers instead of classical C `int`s. The reason is simply that a lot of properties and definitions are true regardless the size of the machine integer we have in input.

On the other hand, we will not talk about the differences that exists between `real` and `float/double`. It would require to speak about precise numerical calculus, and about proofs of programs that rely on such calculus which could deserve an entire dedicated tutorial.

5.2 Predicates

A predicate is a property about different objects that can be true or false. To sum up, we are writing predicates from the beginning of this tutorial in precondition, postcondition, assertion and loop invariant. ACSL allows us to name these predicates, as we could do for a boolean function in C, for example. An important difference, however, is that predicates (as well as functions that we will see later) must be pure, they cannot produce side effects by modifying a pointed value for example.

These predicates can receive some parameters. Moreover, they can also receive some C labels that will allow us to establish relations between different program points.

5.2.1 Syntax

Predicates are introduced using ACSL annotations. The syntax is the following:

```
/*@
  predicate named_predicate {Lbl0, Lbl1, ..., LblN}(type0 arg0, type1 arg1, ..., typeN argN) =
    //a logic relations between all these things
*/
```

For example, we can define the predicate that checks whether an integer in memory is changed between to particular program points:

```
/*@
  predicate unchanged{L0, L1}(int* i) =
    \at(*i, L0) == \at(*i, L1);
*/
```

Warning

Keep in mind that passing a value to a predicate is done, as it is done in C, by value. We cannot write this predicate by directly passing `i` in parameter:

```
/*@
  predicate unchanged{L0, L1}(int i) =
    \at(i, L0) == \at(i, L1);
*/
```

Since `i` is just a copy of the received variable.

We can verify this code using our predicate:

```
int main(){
  int i = 13;
  int j = 37;

 Begin:
  i = 23;

  //@assert ! unchanged{Begin, Here}(&i);
  //@assert   unchanged{Begin, Here}(&j);
}
```

We can also have a look to the goals generated by WP and notice that, even it is slightly (syntactically) modified, the predicate is not unrolled by WP. The provers will determine if they need to use the definition of the predicate.

As we said earlier, one important use of predicates (and logic functions) is to make our specifications more readable and to factor it. An example can be to write a predicate that expresses the validity of an array in read or write. It allows us to avoid writing the complete expression every time we need it and to make it readable quickly:

```
/*@
  predicate valid_range_rw(int* t, integer n) =
    n >= 0 && \valid(t + (0 .. n-1));
  predicate valid_range_ro(int* t, integer n) =
    n >= 0 && \valid_read(t + (0 .. n-1));
*/
/*@
  requires 0 < length;
  requires valid_range_ro(array, length);
  //...
*/
int* search(int* array, size_t length, int element)
```

In these specification, we do not give an explicit label to predicates for their definition, nor for their use. For the definition, Frama-C automatically creates an implicit label. At predicate use, the given label is implicitly `Here`. The fact we do not explicitly define the label in the definition of a predicate does not forbid to explicitly give a label when we use it.

Of course, predicates can be defined in header files in order to produce an utility library for specification for example.

5.2.2 Abstraction

An other important use of predicates is to define the logical state of our data structures when programs start to be more complex. Our data structures must usually respect an invariant (again) that each manipulation function must maintain in order to ensure that the data structure will always remain coherent and usable through future calls.

It allows us to ease the reading of specifications. For example, we can define the specification required to ensure the safety of a fixed size stack. It could be done as illustrated there:

```
struct stack_int{
  size_t top;
  int    data[MAX_SIZE];
};

/*@
  predicate valid_stack_int(struct stack_int* s) = // to be defined ;
  predicate empty_stack_int(struct stack_int* s) = // to be defined ;
  predicate full_stack_int(struct stack_int* s) =  // to be defined ;
*/

/*@
  requires \valid(s);
  assigns *s;
  ensures valid_stack_int(s) && empty_stack_int(s);
*/
void initialize(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !full_stack_int(s);
  assigns  *s;
  ensures valid_stack_int(s);
*/
void push(struct stack_int* s, int value);

/*@
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns \nothing;
*/
int  top(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns *s;
  ensures valid_stack_int(s);
*/
void pop(struct stack_int* s);
```

```
/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> empty_stack_int(s);
*/
int  is_empty(stack_int_t s);


/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> full_stack_int(s);
*/
int  is_full(stack_int_t s);
```

Here, the specification does not express functional properties. For example, we do not specify that when we perform the push of a value, and then we ask for the top of the stack, we get the same value. But we already have enough details to ensure that, even if we cannot prove that we always get the right result (behaviors such as "if I push v, top returns v"), we can still guarantee that we do not produce runtime errors (if we provide correct predicates for the stack, and to prove that the implementation of our functions ensures that no runtime error can occur).

5.3 Logic functions

Logic functions are meant to describe functions that can only be used in specifications. It allows us, first, to factor those specifications and, second, to define some operations on `integer` or `real` with the guarantee that they cannot overflow since they are mathematical types.

Like predicates, they can receive different labels and values in parameter.

5.3.1 Syntax

To define a logic function, the syntax is the following:

```
/*@
  logic return_type my_function{ Label0, ..., LabelN }( type0 arg0, ..., typeN argN ) =
    formula using the arguments ;
*/
```

We can for example define a mathematical linear function using a logic function:

```
/*@
  logic integer ax_b(integer a, integer x, integer b) =
    a * x + b;
*/
```

And it can be used to prove the source code of the following function:

```
/*@
  assigns \nothing ;
  ensures \result == ax_b(3,x,4);
*/
int function(int x){
  return 3*x + 4;
}
```

```
/*@ logic ℤ ax_b(ℤ a, ℤ x, ℤ b) = a * x + b;

*/
/*@ ensures \result ≡ ax_b(3, \old(x), 4);
    assigns \nothing; */
int function(int x)
{
  int __retres;
  /*@ assert rte: signed_overflow: (int)(3 * x) + 4 ≤ 2147483647; */
  /*@ assert rte: signed_overflow: -2147483648 ≤ 3 * x; */
  /*@ assert rte: signed_overflow: 3 * x ≤ 2147483647; */
  __retres = 3 * x + 4;
  return __retres;
}
```

Figure 5.1: Some overflows seem to be possible

This code is indeed proved but some runtime-errors seems to be possible (see Figure 5.1). We can again define some mathematical logic function, that will, from a logic point of view, provide the bounds of the affine function according to the machine type we use? It allows us to then add our bound checking in preconditon or the function.

```
/*@
  logic integer limit_int_min_ax_b(integer a, integer b) =
    (a == 0) ? (b > 0) ? INT_MIN : INT_MIN-b :
    (a <  0) ? (INT_MAX-b)/a :
               (INT_MIN-b)/a ;

  logic integer limit_int_max_ax_b(integer a, integer b) =
    (a == 0) ? (b > 0) ? INT_MAX-b : INT_MAX :
    (a <  0) ? (INT_MIN-b)/a :
               (INT_MAX-b)/a ;
*/

/*@
  requires limit_int_min_ax_b(3,4) < x < limit_int_max_ax_b(3,4);
  assigns \nothing ;
  ensures \result == ax_b(3,x,4);
*/
int function(int x){
  return 3*x + 4;
}
```

N

ote that, as in specifications, computations are done using mathematical integers, we then do not need to care about some overflow risk with the computation or `INT_MIN-b` or `INT_MAX-b`.

Once this specification is provided, everything is fine. Of course, we could provide manually these bounds every time we create an linear logic function. But, by creating these bound computation function, we directly get a way to compute them automatically which is quite comfortable.

5.3.2 Recursive functions and limits of logic functions

Logic functions (as well as predicates) can be recursively defined. However, such an approach will rapidly show some limits in their use for program proof. Indeed, when automatic solver will

reason on such logic properties, if such a function is met, it will be necessary to evaluate it, and SMT solvers are not meant to be efficient for this task, which will be generally costly, producing to long proof resolution and eventually timeouts.

We can have a concrete example with the factorial function, using logic and using C language:

```
/*@
  logic integer factorial(integer n) = (n <= 0) ? 1 : n * factorial(n-1);
*/

/*@
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
unsigned facto(unsigned n){
  return (n == 0) ? 1 : n * facto(n-1);
}
```

Without checking overflows, this function is easy and fast to prove. If we add runtime error checking, RTE does not add any assertion to check the absence of overflow on unsigned integers, since it is specified by the C norm (and is then a defined behavior). If we want to add an assertion at this point, we can ask WP to generate its own bound checking by right-clicking on the function and then asking "insert WP-safety guards". And in this case, an overflow is not proved to be absent.

On `unsigned`, the maximum value for which we can compute factorial is 12. If we go further, it overflows. We can then add this preconditon:

```
/*@
  requires n <= 12 ;
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
unsigned facto(unsigned n){
  return (n == 0) ? 1 : n * facto(n-1);
}
```

If we ask for a proof on this input, Alt-ergo will probably fail, whereas Z3 can compute the proof in less than a second. The reason is that in this case, the heuristics that are used by Z3 consider that it is a good idea to spend a bit more of time on the evaluation of the function. We can for example change the maximum value of `n` to see how the different provers behave. With a `n` fixed to 9, Alt-ergo produces a proof in less than 10 seconds, whereas with a value of 10, even a minute is not enough.

Logic functions can then be defined recursively but without some more help, we are rapidly limited by the fact that provers will need to perform evaluation or to "reason" by induction, two tasks for which they are not efficient, which will limit our possibility for program proofs.

5.4 Lemmas

Lemmas are general properties about predicates or functions. Once these properties are expressed, their proof can be performed (one time) and the provers will then be able to use this result to perform other proofs without requiring to perform again all steps needed to perform the original proof if it appears in a much longer proof about an other property.

For example, lemmas allow us to express properties about recursive functions in order to get easier proofs when we are interested in proving properties that use such functions.

5.4.1 Syntax

Again, we introduce lemmas using ACSL annotations. The syntax is following:

```
/*@
  lemma name_of_the_lemma { Label0, ..., LabelN }:
    property ;
*/
```

This time, the properties we want to express do not depend on received parameters (except for labels). So we will express these properties on universally quantified variables. For example, we can state this lemma, which is true, even if it is trivial:

```
/*@
  lemma lt_plus_lt:
    \forall integer i, j ; i < j ==> i+1 < j+1;
*/
```

This proof can be performed using WP. The property is, of course, proved using only Qed.

5.4.2 Example: properties about linear functions

We can come back to our linear functions and express some interesting properties about them:

```
/* @
  lemma ax_b_monotonic_neg:
    \forall integer a, b, i, j ;
      a <  0 ==> i <= j ==> ax_b(a, i, b) >= ax_b(a, j, b);
  lemma ax_b_monotonic_pos:
    \forall integer a, b, i, j ;
      a >  0 ==> i <= j ==> ax_b(a, i, b) <= ax_b(a, j, b);
  lemma ax_b_monotonic_nul:
    \forall integer a, b, i, j ;
      a == 0 ==> ax_b(a, i, b) == ax_b(a, j, b);
*/
```

For these proofs, Alt-ergo, will probably not be able to discharge generated goals. In this case, Z3 will certainly perform it. We can then write some code examples:

```
/*@
  requires a > 0;
  requires limit_int_min_ax_b(a,4) < x < limit_int_max_ax_b(a,4);
  assigns \nothing ;
  ensures \result == ax_b(a,x,4);
*/
int function(int a, int x){
  return a*x + 4;
}

/*@
  requires a > 0;
  requires limit_int_min_ax_b(a,4) < x < limit_int_max_ax_b(a,4) ;
```

```
  requires limit_int_min_ax_b(a,4) < y < limit_int_max_ax_b(a,4) ;
  assigns \nothing ;
*/
void foo(int a, int x, int y){
  int fmin, fmax;
  if(x < y){
    fmin = function(a,x);
    fmax = function(a,y);
  } else {
    fmin = function(a,y);
    fmax = function(a,x);
  }
  //@assert fmin <= fmax;
}
```

If we do not give the lemmas provided earlier, Alt-ergo will not be able to prove the proof that `fmin` is lesser or equal to `fmax`. With the lemmas it is however very easy for it since the property is the simply an instance of the lemma `ax_monotonic_pos`, the proof is then trivial as our lemma is considered to be true when are not currently proving it.

In this part of the tutorial, we have seen different ACSL construct that allow us to factor our specifications and to express general properties that can be used by our solver to easy their work.

All techniques we have talk about are safe, since they do not *a priori* allow us to write false or contradictory specifications. At least if the specification only use such logic constructions and if every lemma, precondition (at call site), every postcondition, assertion, variant and invariant is correctly proved, the code is correct.

However, sometimes, such constructions are not enough to express all properties we want to express about our programs. The next constructions we will see give us some new possibilities about it, but it will be necessary to be really careful using them since an error would allow us to introduction false assumptions or silently modify the program we are verifying.

6 ACSL - Logic definitions and code

In this part of the tutorial, we will present two important notions of ACSL:

- axiomatic definitions,
- ghost code.

In some cases, this two notions are absolutely needed to ease the process of specification and, more importantly, proof. On one hand by forcing some properties to be more abstract when an explicit modeling involve to much computation during proof, on the other hand by forcing some properties to be explicitly modeled since they are harder to reason with when they are implicit.

Using this two notions, we expose ourselves to the possibility to make our proof irrelevant if we make mistakes writing specification with it. The first one allows us to introduce "false" in our assumptions, or to write imprecise definitions. The second one opens the risk to silently modify the verified program ... making us prove another program, which is not the one we want to prove.

6.1 Axiomatic definitions

Axioms are properties we state to be true no matter the situation. It is a good way to state complex properties that will allow the proof process to be more efficient by abstracting their complexity. Of course, as any property expressed as an axiom is assumed to be true, we have to be very careful when we use them to defined properties: if we introduce a false property in our assumptions, "false" becomes "true" and we can then prove anything.

6.1.1 Syntax

Axiomatic definitions are introduced using this syntax:

```
/*@
  axiomatic Name_of_the_axiomatic_definition {
    // here we can define or declare functions and predicates

    axiom axiom_name { Label0, ..., LabelN }:
      // property ;

    axiom other_axiom_name { Label0, ..., LabelM }:
      // property ;

    // ... we can put as many axioms we need
  }
*/
```

For example, we can write this axiomatic block:

```
/*@
  axiomatic lt_plus_lt{
    axiom always_true_lt_plus_lt:
      \forall integer i, j; i < j ==> i+1 < j+1 ;
  }
*/
```

And we can see that in Frama-C, this property is actually assumed to be true (see Figure 6.1).

```
/*@
axiomatic lt_plus_lt {
  axiom always_true_lt_plus_lt: ∀ ℤ i, ℤ j; i < j ⇒ i + 1 < j + 1;

  }

*/
```

Figure 6.1: First axiom, assumed to be true by Frama-C

Off topic

Currently, our automatic solvers are not powerful enough to compute *the Answer to the Ultimate Question of Life, the Universe, and Everything*. We can help them by stating it as an axiom! Now, we just have to understand the question to determine in which case this result can be useful ...

```
/*@
  axiomatic Ax_answer_to_the_ultimate_question_of_life_the_universe_and_everything {
    logic integer the_ultimate_question_of_life_the_universe_and_everything{L} ;

    axiom answer{L}:
      the_ultimate_question_of_life_the_universe_and_everything{L} = 42;

*/
```

6.1.1.1 Link with lemmas

Lemmas and axioms allows to express the same types of properties. Namely, properties expressed about quantified variables (and possibly global variables, but it is quite rare since it is often difficult to find a global property about such variables being both true and interesting). Apart this first common point, we can also notice that when we are not considering the definition of the lemma itself, lemmas are assumed to be true by WP exactly as axioms are.

The only difference between lemmas and axioms from a proof point of view is that we must provide a proof that each lemma is true, whereas an axiom is always assumed to be true.

6.1.2 Recursive function or predicate definitions

Axiomatic definition of recursive functions and predicates are particularly useful since they will prevent provers to unroll the recursion when is is possible.

The idea is then not to define directly the function or the predicate but to declare it and then to define axioms that will specify its behavior. If we come back to the factorial function, we can define it axiomatically as follows:

```
/*@
  axiomatic Factorial{
    logic integer factorial(integer n);

    axiom factorial_0:
      \forall integer i; i <= 0 ==> 1 == factorial(i) ;

    axiom factorial_n:
      \forall integer i; i > 0 ==> i * factorial(i-1) == factorial(i) ;
  }
*/
```

In this axiomatic definition, our function do not have a body. Its behavior is only defined by the axioms we have stated about it.

A small subtlety is that we must take care about the fact that if some axioms state properties about the content of some pointed memory cells, we have to specify considered memory blocks using the `reads` notation in the declaration. If we omit such a specification, the predicate or function will be considered to be stated about the received pointers and not about pointer memory blocks. So, if the code modifies the content of an array for which we had proven that the predicate or function give some result, this result will not be considered to be potentially different.

If, for example, we want to define that an array only contains 0s, we have to write it as follows:

```
/*@
  axiomatic A_all_zeros{
    predicate zeroed{L}(int* a, integer b, integer e) reads a[b .. e-1];

    axiom zeroed_empty{L}:
      \forall int* a, integer b, e; b >= e ==> zeroed{L}(a,b,e);

    axiom zeroed_range{L}:
      \forall int* a, integer b, e; b < e ==>
        zeroed{L}(a,b,e-1) && a[e-1] == 0 <==> zeroed{L}(a,b,e);
  }
*/
```

And we can again prove that our reset to 0 is correct with this new definition:

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  zeroed(array,0,length);
*/
void reset(int* array, size_t length){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant zeroed(array,0,i);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    array[i] = 0;
}
```

Depending on the Frama-C or automatic solvers versions, the proof of the preservation of the invariant could fail. A reason to this fail is the fact that the prover forget that cells preceding the

one we are currently processing are actually still set to 0. We can add a lemma in our knowledge base, stating that if a set of values of an array did not change between two program points, the first one being a point where the property "zeroed" is verified, then the property is still verified at the second program point.

```
/*@
  predicate same_elems{L1,L2}(int* a, integer b, integer e) =
    \forall integer i; b <= i < e ==> \at(a[i],L1) == \at(a[i],L2);

  lemma no_changes{L1,L2}:
    \forall int* a, integer b, e;
      same_elems{L1,L2}(a,b,e) ==> zeroed{L1}(a,b,e) ==> zeroed{L2}(a,b,e);
*/
```

Then we can add an assertion to specify what did not change between the beginning of the loop block (pointed by the label `L` in the code) and the end (which is `Here` since we state the property at the end):

```
for(size_t i = 0; i < length; ++i){
  L:
  array[i] = 0;
  //@ assert same_elems{L,Here}(array,0,i);
}
```

Note that in this new version of the code, the property stated by our lemma is not proved using automatic solver, that cannot reason by induction. If the reader is curious, the (quite simple) Coq proof can be found there:

Preuve Coq

```
(* Generated par WP *)
Definition P_same_elems (Mint_0 : farray addr Z) (Mint_1 : farray addr Z)
    (a : addr) (b : Z) (e : Z) : Prop :=
    forall (i : Z), let a_1 := (shift_sint32 a i%Z) in ((b <= i)%Z) ->
      ((i < e)%Z) -> (((Mint_0.[ a_1 ]) = (Mint_1.[ a_1 ]))%Z).
Goal
  forall (i_1 i : Z), forall (t_1 t : farray addr Z), forall (a : addr),
  ((P_zeroed t a i_1%Z i%Z)) ->
    ((P_same_elems t_1 t a i_1%Z i%Z)) ->
      ((P_zeroed t_1 a i_1%Z i%Z)).
(* Our proof *)
Proof.
  intros b e.
  (* By induction on the distance between b and e *)
  induction e using Z_induction with (m := b) ; intros mem_l2 mem_l1 a Hz_l1 Hsame.
  (* Base case: "empty" axiom *)
  + apply A_A_all_zeros.Q_zeroed_empty ; assumption.
  + replace (e + 1) with (1 + e) in * ; try omega.
    (* We use the range axiom *)
    rewrite A_A_all_zeros.Q_zeroed_range in * ; intros Hinf.
    apply Hz_l1 in Hinf ; clear Hz_l1 ; inversion_clear Hinf as [Hlast Hothers].
    split.
    (* Sub-range considered by Hsame *)
    - rewrite Hsame ; try assumption ; omega.
    (* Induction hypotheses *)
    - apply IHe with (t := mem_l1) ; try assumption.
      * unfold P_same_elems ; intros ; apply Hsame ; omega.
Qed.
```

In this case study, using an axiomatic definition is not efficient: our property can be perfectly expressed using the basic constructs of the first order logic as we did before. Axiomatic definitions are meant to be used to write definitions that are not easy to express using the basic formalism provided by ACSL. It is here used to illustrate their use with a simple example.

6.1.3 Consistency

By adding axioms to our knowledge base, we can produce more complex proofs since some part of these proofs, expressed by axioms, do not need to be proved anymore (they are already specified to be true) shortening the proof process. However, using axiomatic definitions, **we must be extremely careful**. Indeed, even a small error could introduce false in the knowledge base, making our whole reasoning futile. Our reasoning would still be correct, but relying on false knowledge, it would only learn incorrect things.

The simplest example is the following:

```
/*@
  axiomatic False{
    axiom false_is_true: \false;
  }
*/

int main(){
  // Examples of proven properties

  //@ assert \false;
  //@ assert \forall integer x; x > x;
  //@ assert \forall integer x,y,z ; x == y == z == 42;
  return *(int*) 0;
}
```

And everything is proved, comprising the fact that the dereferencing of 0 is valid (see Figure 6.2).

```
int main(void)
{
  int __retres;
  /*@ assert \false; */ ;
  /*@ assert ∀ ℤ x; x > x; */ ;
  /*@ assert ∀ ℤ x, ℤ y, ℤ z; x ≡ y ≡ z ≡ 42; */ ;
  /*@ assert rte: mem_access: \valid_read((int *)0); */
  __retres = *((int *)0);
  return __retres;
}
```

Figure 6.2: Different false things proved to be true

Of course, this example is extreme, we would not write such an axiom. The problem is in fact that it is really easy to write an axiomatic definition that is subtly false when we express more complex properties, or adding assumptions about the global state of the system.

When we start to create axiomatic definitions, it is worth adding assertions or postconditions requiring a proof of false that we expect to fail to ensure that the definition is not inconsistent. However, it is often not enough! If the subtlety that creates the inconsistency is hard enough to find, provers could need a lot of information other than the axiomatic definition itself to be able to find and use the inconsistency, we then need to always be careful!

More specifically, specifying the values read by a function or a predicate is important for the consistency of an axiomatic definition. Indeed, as previously explained, if we do not specify what is read when a pointer is received, an update of a value inside the array do not invalidate a property known about the content of the array. In such a case, the proof is performed but since the axiom do not talk about the content of the array, we do not prove anything.

For example, in the function that resets an array to 0, if we modify the axiomatic definition, removing the specification of the values that are read by the predicate (`reads a[b .. e-1]`), the proof will still be performed, but will not prove anything about the content of the arrays.

6.1.4 Example: counting occurrences of a value

In this example, we want to prove that an algorithm actually counts the occurrences of a value inside an array. We start by axiomatically define what is the number of occurrences of a value inside an array:

```
/*@
  axiomatic Occurrences_Axiomatic{
    logic integer l_occurrences_of{L}(int value, int* in, integer from, integer to)
      reads in[from .. to-1];

    axiom occurrences_empty_range{L}:
      \forall int v, int* in, integer from, to;
        from >= to ==> l_occurrences_of{L}(v, in, from, to) == 0;

    axiom occurrences_positive_range_with_element{L}:
      \forall int v, int* in, integer from, to;
        (from < to && in[to-1] == v) ==>
      l_occurrences_of(v,in,from,to) == 1+l_occurrences_of(v,in,from,to-1);

    axiom occurrences_positive_range_without_element{L}:
      \forall int v, int* in, integer from, to;
        (from < to && in[to-1] != v) ==>
      l_occurrences_of(v,in,from,to) == l_occurrences_of(v,in,from,to-1);
  }
*/
```

We have three different cases:

- the considered range of values is empty: the number of occurrences is 0,
- the considered range of values is not empty and the last element is the one we are searching: the number of occurrences is the number of occurrences in the current range without the last element, plus 1,
- the considered range of values is not empty and the last element is not the one we are searching: the number of occurrences is the number of occurrences in the current range without the last element.

Then, we can write the C function that computes the number of occurrences of a value inside an array and prove it:

```
/*@
  requires \valid_read(in+(0 .. length));
  assigns  \nothing;
  ensures  \result == l_occurrences_of(value, in, 0, length);
*/
size_t occurrences_of(int value, int* in, size_t length){
  size_t result = 0;

  /*@
    loop invariant 0 <= result <= i <= length;
    loop invariant result == l_occurrences_of(value, in, 0, i);
    loop assigns i, result;
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i)
    result += (in[i] == value)? 1 : 0;

  return result;
}
```

An alternative way to specify, in this code, that `result` is at most `i`, is to express a more general lemma about the number of occurrences of a value inside an array, since we know that it is comprised between 0 and the size of maximum considered range of values:

```
/*@
lemma l_occurrences_of_range{L}:
  \forall int v, int* array, integer from, to:
    from <= to ==> 0 <= l_occurrences_of(v, a, from, to) <= to-from;
*/
```

An automatic solver cannot discharge this lemma. It would be necessary to prove it interactively using Coq, for example. By expressing, generic manually proved lemmas, we can often add useful tools to provers to manipulate more efficiently our axiomatic definitions, without directly adding new axioms that would augment the chances to introduce errors. Here, we still have to realize the proof of the lemma to have a complete proof.

6.1.5 Example: sort

We will prove a simple selection sort:

```
size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;
  for(size_t i = beg+1; i < end; ++i)
    if(a[i] < a[min_i]) min_i = i;
  return min_i;
}

void swap(int* p, int* q){
  int tmp = *p; *p = *q; *q = tmp;
}

void sort(int* a, size_t beg, size_t end){
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

The reader can exercise by specifying and proving the search of the minimum and the swap function. We hide there a correct version of these specification and code, we will focus on the specification and the proof of the sort function that is a interesting use case for axiomatic definitions.

Answer

```
/*@
  requires \valid_read(a + (beg .. end-1));
  requires beg < end;

  assigns  \nothing;

  ensures  \forall integer i; beg <= i < end ==> a[\result] <= a[i];
  ensures  beg <= \result < end;
*/
size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;

  /*@
    loop invariant beg <= min_i < i <= end;
    loop invariant \forall integer j; beg <= j < i ==> a[min_i] <= a[j];
    loop assigns min_i, i;
    loop variant end-i;
  */
  for(size_t i = beg+1; i < end; ++i){
    if(a[i] < a[min_i]) min_i = i;
  }
  return min_i;
}

/*@
  requires \valid(p) && \valid(q);
  assigns  *p, *q;
  ensures  *p == \old(*q) && *q == \old(*p);
*/
void swap(int* p, int* q){
  int tmp = *p; *p = *q; *q = tmp;
}
```

Indeed, a common error we could do, trying to prove a sort function, would be to write the (true!) specification illustrated by Figure 6.3.

This specification is true. But if we recall correctly the part of the tutorial about specifications, we have to *precisely* express what we expect of the program. With this specification, we do not prove all required properties expected for a sort function. For example, the function presented in Figure 6.4 correctly answers to the specification:

Our specification does not express the fact that all elements initially found inside the array must still be found inside the array after executing the sort function. That is to say: the sort function produces a sorted permutation of the original array.

Defining the notion of permutation is easily done using an axiomatic definition. Indeed, to determine that an array is the permutation of an other one, we can limit us to a few cases. First, the array is a permutation of itself, then swapping to values of the array produces a new permutation if we do not change anything else. And finally if we create the permutation p_2 of p_1, and then the permutation p_3 of p_2, then by transitivity p_3 is a permutation of p_1.

```
/*@
  predicate sorted(int* a, integer b, integer e) =
    \forall integer i, j; b <= i <= j < e ==> a[i] <= a[j];
*/

/*@
  requires \valid(a + (beg .. end-1));
  requires beg < end;
  assigns  a[beg .. end-1];
  ensures sorted(a, beg, end);
*/
void sort(int* a, size_t beg, size_t end){
  /*@ //annotation de l'invariant */
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

Figure 6.3: Underspecified `sort` function

```
/*@
  requires \valid(a + (beg .. end-1));
  requires beg < end;

  assigns  a[beg .. end-1];

  ensures sorted(a, beg, end);
*/
void fail_sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg <= i <= end;
    loop invariant \forall integer j; beg <= j < i ==> a[j] == 0;
    loop assigns i, a[beg .. end-1];
    loop variant end-i;
  */
  for(size_t i = beg ; i < end ; ++i)
    a[i] = 0;
}
```

Figure 6.4: Function that satisfies the specification of Figure 6.3

The corresponding axiomatic definition is the following:

```
/*@
  predicate swap_in_array{L1,L2}(int* a, integer b, integer e, integer i, integer j) =
    b <= i < e && b <= j < e &&
    \at(a[i], L1) == \at(a[j], L2) && \at(a[j], L1) == \at(a[i], L2) &&
    \forall integer k; b <= k < e && k != j && k != i ==> \at(a[k], L1) == \at(a[k], L2);

  axiomatic Permutation{
    predicate permutation{L1,L2}(int* a, integer b, integer e)
     reads \at(*(a+(b .. e - 1)), L1), \at(*(a+(b .. e - 1)), L2);

    axiom reflexive{L1}:
      \forall int* a, integer b,e ; permutation{L1,L1}(a, b, e);

    axiom swap{L1,L2}:
      \forall int* a, integer b,e,i,j ;
        swap_in_array{L1,L2}(a,b,e,i,j) ==> permutation{L1,L2}(a, b, e);

    axiom transitive{L1,L2,L3}:
      \forall int* a, integer b,e ;
        permutation{L1,L2}(a, b, e) && permutation{L2,L3}(a, b, e) ==> permutation{L1,L3}(a, b, e);
  }
*/
```

We can then specify that our sort function produces the sorted permutation of the original array and we can then prove it by providing the invariant of the function:

```
/*@
  requires beg < end && \valid(a + (beg .. end-1));
  assigns  a[beg .. end-1];
  ensures sorted(a, beg, end);
  ensures permutation{Pre, Post}(a,beg,end);
*/
void sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg <= i <= end;
    loop invariant sorted(a, beg, i) && permutation{Begin, Here}(a, beg, end);
    loop invariant \forall integer j,k; beg <= j < i ==> i <= k < end ==> a[j] <= a[k];
    loop assigns i, a[beg .. end-1];
    loop variant end-i;
  */
  for(size_t i = beg ; i < end ; ++i){
    //@ ghost begin: ;
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
    //@ assert swap_in_array{begin,Here}(a,beg,end,i,imin);
  }
}
```

This time, our property is precisely defined, the proof is relatively easy to produce, only requiring to add an assertion in the block of the loop to state that it only performs a swap of values inside the array (and then giving the transition to the next permutation). To define this swap notion, we use a particular annotation (at line 16), introduced using the keyword `ghost`. Here, the goal is to introduce a label in the code that in fact does not exists from the program point of view, and is only visible from a specification point of view. This is the topic of the next section.

6.2 Ghost code

Behind this title, that seems to be an action movie, we find in fact a way to enrich our specification with information expressed as regular C code. Here, the idea is to add variables and source code that will not be part of the actual program but will model logic states that will only be visible from a proof point of view. Using it, we can make explicit some logic properties that were previously only known implicitly.

6.2.1 Syntax

Ghost code is added using annotations that will contains C code introduced using the `ghost` keyword:

```
/*@
  ghost
  // C source code
*/
```

The only rules we have to respect in such a code, is that we cannot write a memory block that is not itself defined in ghost code, and that the code must close any block it would open. Apart of this, any computation can be inserted provided it *only* modifies ghost variables.

Here are some examples of ghost code:

```
//@ ghost int ghost_glob_var = 0;

void foo(int a){
  //@ ghost int ghost_loc_var = a;

  //@ ghost Ghost_label: ;
  a = 28 ;

  //@ ghost if(a < 0){ ghost_loc_var = 0; }

  //@ assert ghost_loc_var == \at(a, Ghost_label) == \at(a, Pre);
}
```

We must again be careful using ghost code. Indeed, the tool will not perform any verification to ensure that we do not write in the memory of the program by mistake. This problem being, in fact, an undecidable problem, this analysis would require a proof by itself. For example, this code is allowed in input of Frama-C even if it explicitly modifies the state of the program we want to verify:

```
int a;

void foo(){
  //@ ghost a = 42;
}
```

We then need to be really careful about we are doing using ghost code (by making it simple).

6.2.2 Make a logical state explicit

The goal of ghost code is to make explicit some information that are without them implicit. For example, in the previous section, we used it to get an explicit logic state known at a particular point of the program.

Let us take a more complex example. Here, we want to prove that the following function returns the value of the maximal sum of subarrays of a given array. A subarray of an array `a` is a contiguous subset of values of `a`. For example, for an array `{ 0 , 3 , -1 , 4 }`, some subarrays can be `{}`, `{ 0 }`, `{ 3 , -1 }`, `{ 0 , 3 , -1 , 4 }`, ... Note that as we allow empty arrays for subarrays, the sum is at least 0. In the previous array, the subarray with the maximal sum is `{ 3 , -1 , 4 }`, the function would then return 6.

```
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0)   cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

In order to specify the previous function, we will need an axiomatic definition about sum. This is not too much complex, the careful reader can express the needed axioms as an exercise:

```
/*@ axiomatic Sum {
  logic integer sum(int *array, integer begin, integer end) reads a[begin..(end-1)];
}*/
```

Some correct axioms are hidden there:

Answer

```
/*@
  axiomatic Sum_array{
    logic integer sum(int* array, integer begin, integer end)
      reads array[begin .. (end-1)];

    axiom empty:
      \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
    axiom range:
      \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
  }
*/
```

The specification of the function is the following:

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
```

For any bounds, the value returned by the function must be greater or equal to the sum of the elements between these bounds, and there must exist some bounds such that the returned value

is exactly the sum of the elements between these bounds. About this specification, when we want to add the loop invariant, we will realize that we miss some information. We want to express what are the values `max` and `cur` and what are the relations between them, but we cannot do it!

Basically, our postcondition needs to know that there exists some bounds `low` and `high` such that the computed sum corresponds to these bounds. However, in our code, we do not have anything that express it from a logic point of view, and we cannot *a priori* make the link between this logical formalization. We will then use ghost code to record these bounds and express the loop invariant.

We will first need two variables that will allow us to record the bounds of the maximum sum range, we will call them `low` and `high`. Every time we will find a range where the sum is greater than the current one, we will update our ghost variables. This bounds will then corresponds to the sum currently stored by `max`. That induces that we need other bounds: the ones that corresponds to the sum store by the variable `cur` from which we will build the bounds corresponding to `max`. For these bounds, we will only add a single ghost variable: the current low bound `cur_low`, the high bound being the variable `i` of the loop. The specified code is illustrated by Figure 6.5.

The invariant `BOUNDS` expresses how the different bounds are ordered during the computation. The invariant `REL` expresses what the variables `cur` and `max` mean depending on the bounds. Finally, the invariant `POST` allows us to create a link between the invariant and the postcondition of the function.

The reader can verify that this function is indeed correctly proved without RTE verification. If we add RTE verification, the overflow on the variable `cur`, that is the sum, seems to be possible (and it is indeed the case).

Here, we will not try to fix this because it is not the topic of this example. The way we can prove the absence of RTE here strongly depends on the context where we use this function. A possibility is to strongly restrict the contract, forcing some properties about values and the size of the array. For example, we could strongly limit the maximal size of the array and strong bounds on each value of the different cells. An other possibility would be to add an error value in case of overflow (-1 for example), and to specify that when an overflow is produced, this value is returned.

In this part, we have covered some advanced constructions of the ACSL language that allows to express and prove more complex properties about programs.

Badly used, this features can make our analyses incorrect, we then need to be careful manipulating them and not hesitate to check them again and again, or eventually express properties to verify about them to assure that we are not introducing an incoherence in our program or our assumptions.

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  //@ ghost size_t cur_low = 0;
  //@ ghost size_t low = 0;
  //@ ghost size_t high = 0;

  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;

    loop invariant REL :   cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POST:   \forall integer l;    0 <= l <= i      ==> sum(a,l,i) <= cur;
    loop invariant POST:   \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;

    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) {
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

Figure 6.5: Specified `max_sub_array` function

7 Conclusion

Voilà, c'est fini ... [Jean-Louis Aubert, *Bleu Blanc Vert*, 1989]

... for this introduction to the proof of C programs using Frama-C and WP.

Along this tutorial, we have seen how we can use these tools to specify what we except of our programs and verify that the source code we have produced indeed corresponds to the specification we have provided. This specification is provided using annotations of our functions that includes the contract they must respect. These contracts are properties required about the input to ensure that the function will correctly work, which is specified by properties about the output of the function.

Starting from specified programs, WP is able to produce the weakest precondition of our programs, provided what we want in postcondition, and to ask some provers if the specified precondition is compatible with the computed one. The reasoning is completely modular, which allows to prove function in isolation from each other and to compose the results. WP cannot currently work with dynamic allocation. A function that would use it could not be proved.

However, even without dynamic allocation, a lot of function can be proved since they work with data-structures that are already allocated. And these functions can then be called with the certainty that they perform a correct job. If we cannot or do not want to prove calling code, we can still write something like this:

```
/*@
  requires some_properties_on(a);
  requires some_other_on(b);

  assigns ...
  ensures ...
*/
void ma_fonction(int* a, int b){
  //this is indeed the "assert" defined in "assert.h"
  assert(/*properties on a*/ && "must respect properties on a");
  assert(/*properties on b*/ && "must respect properties on b");
}
```

Which allows us to benefit from the robustness of our function having the possibility to debug and incorrect call in a source code that we cannot or do want to prove.

Writing specifications is sometimes long or tedious. Higher-level features of ACSL (predicates, logic functions, axiomatizations) allow us to lighten this work, as well as our programming languages allow us to define types containing other types, functions call functions, bringing us to the final program. But, despite this, write specification in a formal language, no matter which one, is generally a hard task.

However, this **formalization** of our need is **crucial**. Concretely, such a formalization is a work every developer should do. And not only in order to prove a program. Even the definition of tests for a function requires to correctly understand its goal if we want to test what is necessary and only what

is necessary. And writing specification in a formal language is incredibly useful (even if it can be sometimes frustrating) to get a clear specification.

Formal languages, that are close to mathematics, are precise. Mathematics have this: they are precise. What is more terrible that reading a specification written in a natural language, with complex sentences, using conditional forms, imprecise terms, ambiguities, compiled in administrative documents composed of hundreds of pages, and where we need to determine, "so, what this function is supposed to do ? And have I to validate about it ?".

Formal methods are probably not enough used currently. Sometimes because of mistrust, sometimes because of ignorance, sometimes becuase of prejudices based on ideas born at the begining of the tools, 20 years ago. Our tools evolve, the way we develop change, probably faster than in any other technical domain. Saying that these tools could never be used for real life programs would certainly be a too big shortcut. After all, we see everyday how much development is different from what it were 10 years, 20 years, 40 years ago and can barely imagine how much it will be different in 10 years, 20 years, 40 years.

During the past few years, safety and security questions have become more and more actual and crucial. Formal methods also progressed a lot, and the improvement they bring for these questions are greatly appreciated. For example, this hyperlink brings to the report of a conference about security that brought together people from academic and industrial world, in which we can read:

> Adoption of formal methods in various areas (including verification of hardware and embedded systems, and analysis and testing of software) has dramatically improved the quality of computer systems. We anticipate that formal methods can provide similar improvement in the security of computer systems.
>
> [...]
>
> **Without broad use of formal methods, security will always remain fragile.** [Formal Methods for Security, 2016]

8 Going further

8.1 With Frama-C

Frama-C provides different ways to analyse programs. In these tools, the most commonly used and interesting to know from a static and dynamic verification point of view are certainly those ones:

- abstract interpretation analysis using EVA,
- the transformation of annotation into runtime verification using E-ACSL.

The goal of the first one is to compute the domain of the different variables at each program point. When we precisely know these domains, we can determine if these variables can produce errors when they are used. However this analysis is executed on the whole program and not modularly, it is also stongly dependent of the type of domain we use (we will not enter into details here) and it is not so good at keeping the relations between variables. On the other side, it is really completely automatic, we do not even need to give loop invariant ! The most manual part of the work is to determine whether or not an alarm is a true error or a false positive.

The second analysis allows to generate from an original program, a new program where the assertions are transformed into runtime verifications. If these assertions fail, the program fails. If they are valid, the program have the same behavior it would have without the assertions. An example of use is to generate the verification of absence of runtime errors as assertions using `-rte` and then to use E-ACSL to generate the program containing the runtime verification that these assertions do not fail.

There exists different other plugins for very different tasks:

- impact analysis of a modification,
- data-flow analysis to visit it efficiently,
- ...

Finally, a last possibility that will motivate the use of Frama-C is the ability to develop their own analysis plugins using the API provided by the Frama-C kernel. A lot of tasks can be realized by the analysis of the source code and Frama-C allows to build different analyses.

8.2 With deductive proof

Along this tutorial we used WP to generate proof obligation starting from programs with their specification. Next we have used automatic solvers to assure that these properties were indeed verified.

When we use other solvers than Alt-Ergo and Coq, the communication with this solver in provided by a translation to the Why3 language that will next be used to bridge the gap to automatic solvers.

But this is not the only way to use Why3. It can also be used itself to write programs and prove them. It especially provides a set of theories for some common data structures.

There is some proofs that cannot be discharged by automatic solvers. In such a case, we have to provide these proofs interactively. WP, like Why3, can extract its goals to Coq, and it is very interesting to study this language. In the context of Frama-C, we produce lemmas libraries already proved that we can reuse. But Coq can also be used for many different tasks, including programming. Note that Why3 can also extract its proof obligations to Isabelle or PVS that are also proof assistants.

Finally, there exists other program logics, for example separation logic or concurrent program logics. Again this notion are interesting to know in the context of Frama-C: if we cannot directly use them, they can inspire the way we specify our program in Frama-C for the proof with WP. They could also be implemented into new plugins to Frama-C.

A whole new world of methods to explore.