Jessie Plugin Tutorial
Beryllium Version

Yannick Moy Claude Marché

September 21, 2009

Chapter 1

Introduction

1.1 Batch Mode vs. GUI Mode

The Jessie plug-in allows to perform deductive verification of C programs inside Frama-C. The C file possibly annotated in ACSL is first checked for syntax errors by Frama-C core, before it is translated to various intermediate languages inside the Why Platform embedded in Frama-C, and finally verification conditions (VC) are generated and a prover is called on these, as sketched in Figure 1.1.

By default, the Jessie plug-in launches the GUI mode. To invoke this mode on a file `ex.c`, just type

```
> frama-c -jessie ex.c
```

The GUI of the Why Plaform (a.k.a. GWhy) is called. It presents each VC on a line, with available provers on columns.

To invoke the plug-in in batch mode, use the `-jessie-atp` with the prover name as argument, e.g.

```
> frama-c -jessie -jessie-atp simplify ex.c
```

runs the prover Simplify on the generated VCs. Valid identifiers for provers are documented in Why, it includes `alt-ergo`, `cvc3`, `yices`, `z3`. See also the prover tricks page .

Finally, you can use the generic name `goals` to just ask for generation of VCs without running any prover.

1.2 Basic Use

A program does not need to be complete nor annotated to be analyzed with the Jessie plug-in. As a first example, take program `max`:

```
int max(int i, int j) {
```

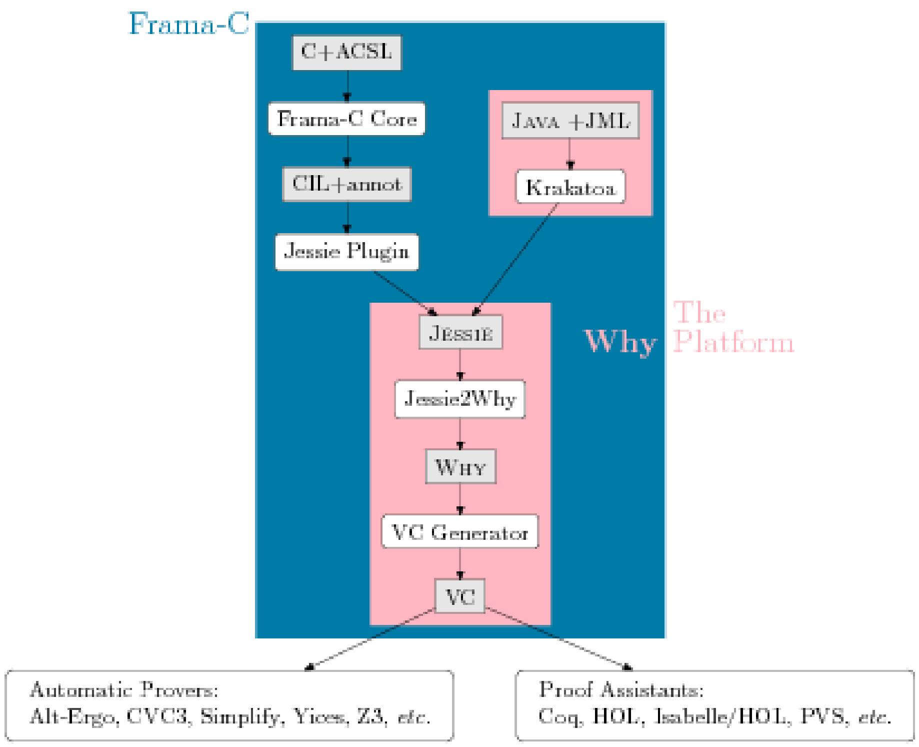


Figure 1.1: Frama-C and the Why Platform

```
  return (i < j) ? j : i;
}
```

Calling the Jessie plug-in in batch mode generates the following output:

```
frama-c -jessie -jessie-atp simplify max.c
Parsing
[preprocessing] running gcc -C -E -I. -include
/usr/local/share/frama-c/jessie/jessie_prolog.h -dD max.c
Cleaning unused parts
Symbolic link
Starting semantical analysis
Starting Jessie translation
Producing Jessie files in subdir max.jessie
File max.jessie/max.jc written.
File max.jessie/max.cloc written.
Calling Jessie tool in subdir max.jessie
Generating Why function f
Calling VCs generator.
```

```
why -simplify [...] why/max.why
Running Simplify on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
simplify/max_why.sx           :  (0/0/0/0/0)
```

The result of calling prover Simplify is succinctly reported as a sequence of symbols `.`, `*`, `?`, `#` and `!` which denote respectively that the corresponding VC is valid, invalid or unknown, or else that a timeout or a failure occured. By default, timeout is set to 10 s for each VC. This result is summarized as a tuple (v,i,u,t,f) reporting the total number of each outcome.

Here, summary `(0/0/0/0/0)` says that there were no VC to prove. If instead we call the Jessie plug-in in GUI mode, we get no VC to prove. Indeed, function `max` is safe and we did not ask for the verification of a functional property.

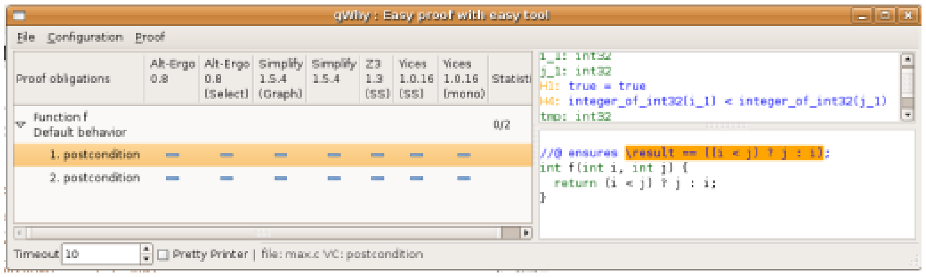
Consider now adding a postcondition to function `max`:

```
//@ ensures \result == ((i < j) ? j : i);
int max(int i, int j) {
  return (i < j) ? j : i;
}
```

This ACSL annotation expresses the fact function `max` returns the maximum of its parameters `i` and `j`. Now, running the Jessie plug-in in batch mode outputs:

```
...
Running Simplify on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
simplify/max_why.sx           : .? (1/0/1/0/0)
total   :   2
valid   :   1 ( 50%)
invalid :   0 (  0%)
unknown :   1 ( 50%)
timeout :   0 (  0%)
failure :   0 (  0%)
total wallclock time : 0.19 sec
total CPU time       : 0.16 sec
valid VCs:
    average CPU time : 0.08
    max CPU time     : 0.08
invalid VCs:
    average CPU time : nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : 0.08
    max CPU time     : 0.08
```

This says that Simplify could prove one VC and not the other. To see what these VC represent, we call now the Jessie plug-in in GUI mode. Each VC represents verification of the postcondition in a different context. The first VC represents the context where `i < j`:

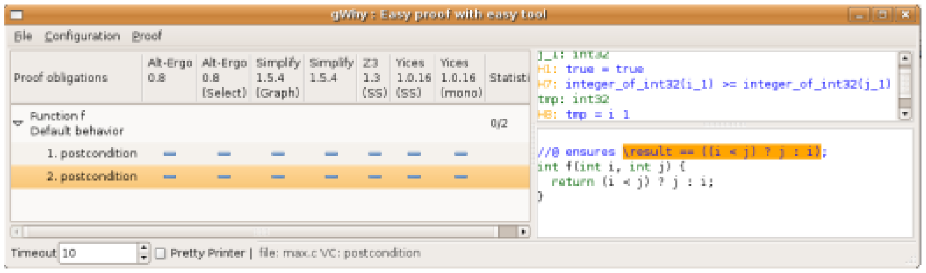

gWhy : Easy proof with easy tool
File Configuration Proof
Timeout 10
Pretty Printer | file: max.c VC: postcondition

Proof obligations	Alt-Ergo 0.8	Alt-Ergo 0.8 (Select)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 1.3 (SS)	Yices 1.0.16 (SS)	Yices 1.0.16 (mono)	Statisti
Function f Default behavior								0/2
1. postcondition	–	–	–	–	–	–	–	
2. postcondition	–	–	–	–	–	–	–	

```
i_1: int32
j_1: int32
H1: true = true
H4: integer_of_int32(i_1) < integer_of_int32(j_1)
tmp: int32
```

```
//@ ensures \result == ((i < j) ? j : i);
int f(int i, int j) {
  return (i < j) ? j : i;
}
```

The second VC represents the context where `i >= j`. Context can be seen in the upper right panel of GWhy, expressed in the intermediate language of Why. It is not yet possible to retrieve the equivalent C expressions or statements.

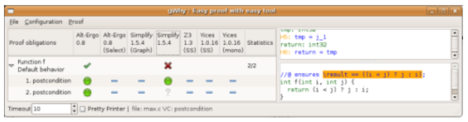

gWhy : Easy proof with easy tool
File Configuration Proof
Timeout 10
Pretty Printer | file: max.c VC: postcondition

Proof obligations	Alt-Ergo 0.8	Alt-Ergo 0.8 (Select)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 1.3 (SS)	Yices 1.0.16 (SS)	Yices 1.0.16 (mono)	Statisti
Function f Default behavior								0/2
1. postcondition	–	–	–	–	–	–	–	
2. postcondition	–	–	–	–	–	–	–	

```
j_1: int32
H1: true = true
H7: integer_of_int32(i_1) >= integer_of_int32(j_1)
tmp: int32
H8: tmp = i_1
```

```
//@ ensures \result == ((i < j) ? j : i);
int f(int i, int j) {
  return (i < j) ? j : i;
}
```

Running Simplify inside GWhy shows that the second VC is not proved by Simplify. However, it is proved by prover Alt-Ergo.


gWhy : Easy proof with easy tool
File Configuration Proof
Timeout 10
Pretty Printer | file: max.c VC: postcondition

Proof obligations	Alt-Ergo 0.8	Alt-Ergo 0.8 (Select)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 1.3 (SS)	Yices 1.0.16 (SS)	Yices 1.0.16 (mono)	Statistics
Function f Default behavior	✓			✗				2/2
1. postcondition	●	–	–	●	–	–	–	
2. postcondition	●	–	–	?	–	–	–	

```
tmp: int32
H5: tmp = j_1
return: int32
H6: return = tmp
```

```
//@ ensures \result == ((i < j) ? j : i);
int f(int i, int j) {
  return (i < j) ? j : i;
}
```

The Jessie plug-in can also be run in batch mode with prover Alt-Ergo instead of Simplify, with option `-jessie-atp alt-ergo`, which results in the following output:

```
...
Running Alt-Ergo on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
why/max_why.why                    : .. (2/0/0/0/0)
total   :  2
valid   :  2 (100%)
invalid :  0 (  0%)
unknown :  0 (  0%)
timeout :  0 (  0%)
```

```
failure :    0 (   0%)
total wallclock time : 0.35 sec
total CPU time       : 0.14 sec
valid VCs:
    average CPU time : 0.07
    max CPU time     : 0.08
invalid VCs:
    average CPU time : nan
    max CPU time     : 0.00
unknown VCs:
    average CPU time : nan
    max CPU time     : 0.00
```

1.3 Safety Checking vs. Functional Verification

In the simple `max` example, VC for the postcondition are grouped in the default behavior for function `max`. In general, VC for a function are grouped in more than one group:

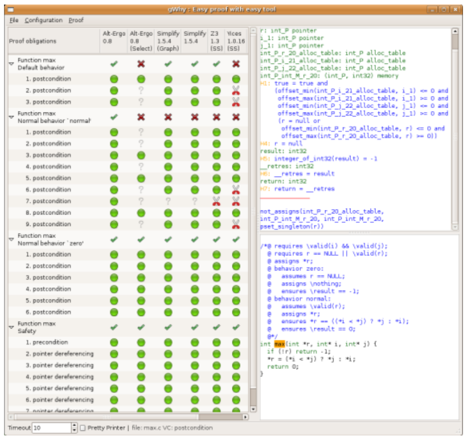
- *Safety*: VC this group guard against safety violations such as null-pointer dereferencing, buffer overflow, integer overflow, etc.
- *Default behavior*: VC in this group concern the verification of a function's default behavior, which includes verification of its postcondition, frame condition, loop invariants and intermediate assertions.
- *User-defined behavior*: VC in this group concern the verification of a function's user-defined behavior, which includes verification of its postcondition, frame condition, loop invariants and intermediate assertions for this specific behavior.

Here is a more complex variant of function `max` which takes pointer parameters and returns 0 on success and -1 on failure.

```
/*@ requires \valid(i) && \valid(j);
  @ requires r == NULL || \valid(r);
  @ assigns *r;
  @ behavior zero:
  @   assumes r == NULL;
  @   assigns \nothing;
  @   ensures \result == -1;
  @ behavior normal:
  @   assumes \valid(r);
  @   assigns *r;
  @   ensures *r == ((*i < *j) ? *j : *i);
  @   ensures \result == 0;
  @*/
```

```
int max(int *r, int* i, int* j) {
  if (!r) return -1;
  *r = (*i < *j) ? *j : *i;
  return 0;
}
```

Running the Jessie plug-in in GUI mode results in 4 groups of VC: Safety, Default behavior, Normal behavior ‘normal‘ and Normal behavior ‘zero‘ for the two user-defined behaviors.


gWhy : Easy proof with easy tool
File Configuration Proof
Timeout 10
Pretty Printer | file: max.c VC: postcondition

Proof obligations	Alt-Ergo 0.8	Alt-Ergo 0.8 (Select)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 1.3 (SS)	Yices 1.0.16 (SS)
Function max Default behavior	✓	✗	✓	✓	✓	✗
1. postcondition	●	●	●	●	●	●
2. postcondition	●	?	●	●	●	✂
3. postcondition	●	?	●	●	●	✂
Function max Normal behavior `normal'	✓	✗	✗	✗	✗	✗
1. postcondition	●	?	●	●	●	●
2. postcondition	●	?	●	●	●	●
3. postcondition	●	●	●	●	●	●
4. postcondition	●	?	●	●	●	●
5. postcondition	●	●	●	●	●	●
6. postcondition	●	?	●	●	●	✂
7. postcondition	●	?	?	?	✂	✂
8. postcondition	●	●	●	●	●	●
9. postcondition	●	?	●	●	●	✂
Function max Normal behavior `zero'	✓	✓	✓	✓	✓	✓
1. postcondition	●	●	●	●	●	●
2. postcondition	●	●	●	●	●	●
3. postcondition	●	●	●	●	●	●
4. postcondition	●	●	●	●	●	●
5. postcondition	●	●	●	●	●	●
6. postcondition	●	●	●	●	●	●
Function max Safety	✓	✓	✓	✓	✓	✓
1. precondition	●	●	●	●	●	●
2. pointer dereferencing	●	●	●	●	●	●
3. pointer dereferencing	●	●	●	●	●	●
4. pointer dereferencing	●	●	●	●	●	●
5. pointer dereferencing	●	●	●	●	●	●
6. pointer dereferencing	●	●	●	●	●	●
7. pointer dereferencing	●	●	●	●	●	●

```
r: int_P pointer
i_1: int_P pointer
j_1: int_P pointer
int_P_r_20_alloc_table: int_P alloc_table
int_P_i_21_alloc_table: int_P alloc_table
int_P_j_22_alloc_table: int_P alloc_table
int_P_int_M_r_20: (int_P, int32) memory
H1: true = true and
    (offset_min(int_P_i_21_alloc_table, i_1) <= 0 and
     offset_max(int_P_i_21_alloc_table, i_1) >= 0 and
     offset_min(int_P_j_22_alloc_table, j_1) <= 0 and
     offset_max(int_P_j_22_alloc_table, j_1) >= 0 and
     (r = null or
      offset_min(int_P_r_20_alloc_table, r) <= 0 and
      offset_max(int_P_r_20_alloc_table, r) >= 0))
H4: r = null
result: int32
H5: integer_of_int32(result) = -1
__retres: int32
H6: __retres = result
return: int32
H7: return = __retres


not_assigns(int_P_r_20_alloc_table,
int_P_int_M_r_20, int_P_int_M_r_20,
pset_singleton(r))
```

```
/*@ requires \valid(i) && \valid(j);
  @ requires r == NULL || \valid(r);
  @ assigns *r;
  @ behavior zero:
  @   assumes r == NULL;
  @   assigns \nothing;
  @   ensures \result == -1;
  @ behavior normal:
  @   assumes \valid(r);
  @   assigns *r;
  @   ensures *r == ((*i < *j) ? *j : *i);
  @   ensures \result == 0;
  @*/
int max(int *r, int* i, int* j) {
  if (!r) return -1;
  *r = (*i < *j) ? *j : *i;
  return 0;
}
```

VC that are proved in one group can be available to prove VC in other groups. No circularity paradox is possible here, since the proof of a VC can only rely on other VC higher in the control-flow graph of the function. We made the following choices:

- To prove a VC in *Safety*, one can rely on VC in *Default behavior*. Typically, one can rely on preconditions or loop invariants to prove safety.
- To prove a VC in *Default behavior*, one can rely on VC in *Safety*. Typically, one can rely on ranges of values implied by safety to prove loop invariants and postconditions.
- To prove a VC in a *Normal behavior*, one can rely on VC in both *Safety* and *Default behavior*.

Next, we detail how to prove each group of VC.

Chapter 2

Safety Checking

A preliminary to any verification task using the Jessie plug-in is to verify the safety of functions. Safety has two components: memory safety and integer safety. Memory safety deals with validity of memory accesses to allocated memory. Integer safety deals with absence of integer overflows and validity of operations on integers, such as the absence of division by zero.

2.1 Memory Safety

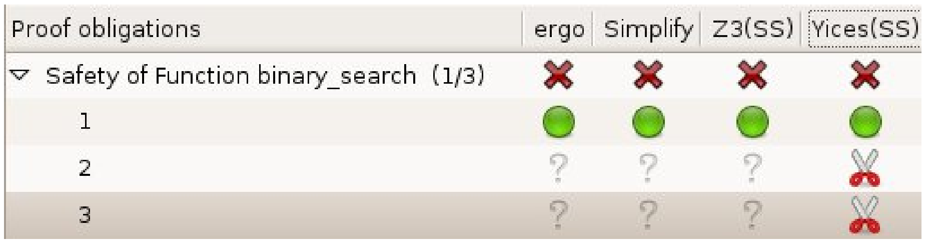
To concentrate first on memory safety only, we call the Jessie plug-in with option `-jessie-int-model exact`. This option dictates that integers in C programs behave as infinite-precision mathematical integers, without overflows. Our running example will be the famous `binary_search` function, which searches an element in an ordered array of such elements. On success, it returns the index at which the element appears in the array. On failure, it returns `-1`.

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  while (l <= u) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

Let's call Frama-C with the Jessie plug-in on this program:

```
> frama-c -jessie -jessie-int-model exact binary-search.c
```

As seen on Figure 2.1, we get 3 VC, an obvious one that states the divisor `2` is not null, and two more that state the array access `t[m]` should be within bounds. This is due to the memory model used, that decomposes any access check into two: one that states the access is above the minimal bound allowed, and one that states the access is below the maximal bound allowed.



Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)
▽ Safety of Function binary_search (1/3)	✗	✗	✗	✗
1	●	●	●	●
2	?	?	?	✂
3	?	?	?	✂

Figure 2.1: Memory safety with no annotations

The obvious VC is trivially proved by all provers, while the two VC for memory safety cannot be proved. Indeed, it is false that, in any context, function `binary_search` is memory safe. To ensure memory safety, `binary_search` must be called in a context in which requires `n` to be positive and array `t` to be valid between indices `0` and `n-1` included. Since function `binary_search` accesses array `t` inside a loop, it is not enough to give a function precondition to make the generated VC provable. Classically, one must add a loop invariant that states the guarantees provided on the array index, despite its changing value. It states that the value of index `l` stays within the bounds of the array `t`.

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  //@ loop invariant 0 <= l && u <= n-1;
  while (l <= u ) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

There are now 7 VC generated: 2 to guarantee the loop invariant is initially established (because the conjunct is split), 2 to guarantee the same loop invariant is preserved through the loop, and the 3 VC seen previously. Not all VC generated are proved automatically with these annotations. Of the 3 VC seen previously, the maximal bound check is still not proved. And the preservation of the loop invariant that deals with an upper bound on u is not proved either. It comes from the non-linear expression assigned to min the loop, that is difficult to take into account automatically. A first attempt at solving this problem is to add an assertion to help automatic provers, providing some form of cut or hint in the proof. This is not sufficient here. Another possibility is to add a *lemma*, that should be proved using available axioms, and used as an axiom in proving the VC for safety. This is working on our example.

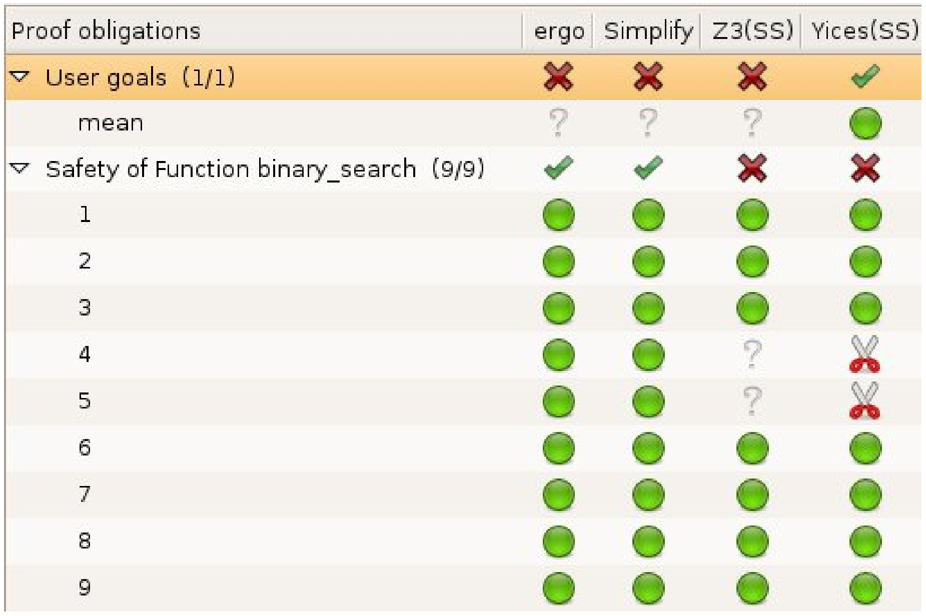
```
/*@ lemma mean : \forall integer x, y; x <= y ==> x <= (x+y)/2 <= y; */

//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  //@ loop invariant 0 <= l && u <= n-1;
  while (l <= u) {
    int m = (l + u) / 2;
    //@ assert l <= m <= u;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

As shown in Figure 2.2, no single automatic prover proves all VC, but altogether, all VC are proved by some prover. This guarantees the memory safety of function binary_search. Notice the assertion adds 2 VC, which totals 9 VC for function binary_search, plus the lemma.

In the following, we remove the assertion, which is not useful.

2.2 Integer Overflow Safety

Let's now consider machine integers instead of idealized mathematical integers. Notice our default target machine architecture states that plain int are 32-bit integers.



Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)
▽ User goals (1/1)	✗	✗	✗	✓
mean	?	?	?	●
▽ Safety of Function binary_search (9/9)	✓	✓	✗	✗
1	●	●	●	●
2	●	●	●	●
3	●	●	●	●
4	●	●	?	✂
5	●	●	?	✂
6	●	●	●	●
7	●	●	●	●
8	●	●	●	●
9	●	●	●	●

Figure 2.2: Memory safety with precondition and loop invariant

Since we are dealing with signed integers, a reasonable choice is to prevent overflows altogether. Thus, we call Frama-C with option:

```
> framac -jessie -jessie-int-model bounded binary-search.c
```

The result can be seen in Figure 2.3.There are 10 more VC to check integer operations return a result within bounds, one of which only is not proved. Except that, the result is nearly the same as with exact integers, except proving the lemma takes more time, due to the added layer of encoding for bounded integers.

The only VC not proved checks that `l+u` does not overflow a machine integer. Nothing prevents this from happening with our current precondition for function `binary_search`. There are two possibilities here. The easiest one is to strengthen the precondition by requiring that `n` is no more than half the maximal signed integer `INT_MAX`. The best one is to change the source of `binary_search` to prevent overflows even in presence of large integers. It consists in changing the buggy line

```
int m = (l + u) / 2;
```

into

```
int m = l + (u - l) / 2;
```

This is our choice here. As shown in Figure 2.4, all VC are now proved automatically.

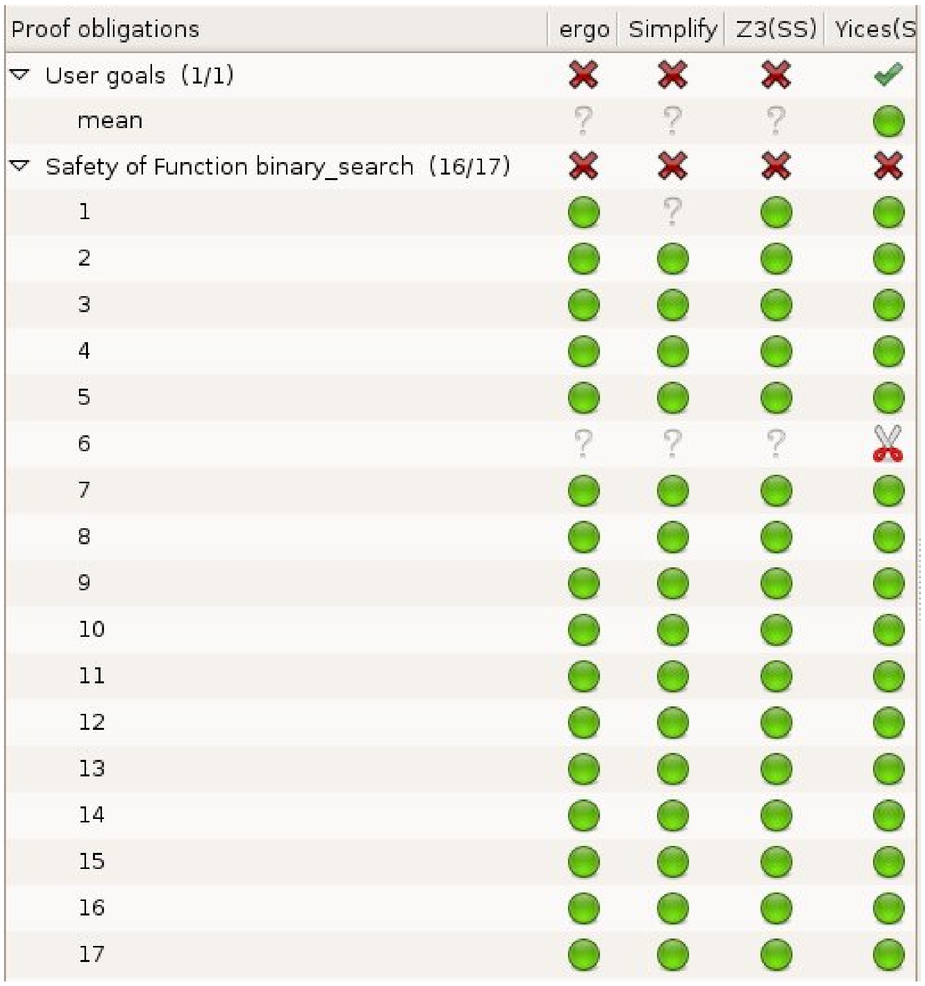
Another choice is to set the integer mode to modulo, which means overflows are allowed, with modulo semantics. To reach that result, we call Frama-C with options:

```
> framac -jessie -jessie-int-model modulo binary-search.c
```

This leads to the same choices as above to guarantee this time that the memory access `t[m]` is within bounds. Otherwise, `m` might become strictly negative, very likely accessing beyond `t` bounds.

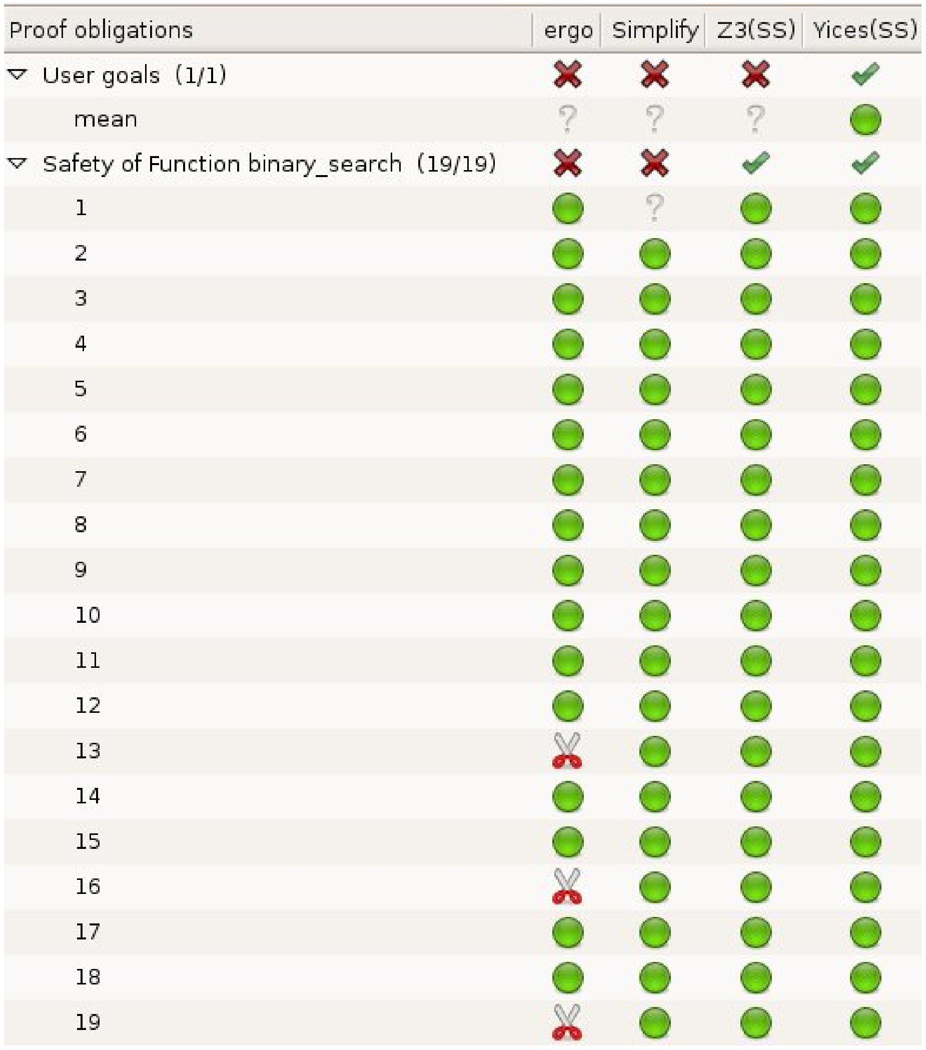
2.3 Combining with Value analysis plug-in

TODO



Proof obligations	ergo	Simplify	Z3(SS)	Yices(S
▽ User goals (1/1)	✗	✗	✗	✓
mean	?	?	?	●
▽ Safety of Function binary_search (16/17)	✗	✗	✗	✗
1	●	?	●	●
2	●	●	●	●
3	●	●	●	●
4	●	●	●	●
5	●	●	●	●
6	?	?	?	✂
7	●	●	●	●
8	●	●	●	●
9	●	●	●	●
10	●	●	●	●
11	●	●	●	●
12	●	●	●	●
13	●	●	●	●
14	●	●	●	●
15	●	●	●	●
16	●	●	●	●
17	●	●	●	●

Figure 2.3: Memory safety + integer overflow safety



Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)
▽ User goals (1/1)	✗	✗	✗	✓
mean	?	?	?	●
▽ Safety of Function binary_search (19/19)	✗	✗	✓	✓
1	●	?	●	●
2	●	●	●	●
3	●	●	●	●
4	●	●	●	●
5	●	●	●	●
6	●	●	●	●
7	●	●	●	●
8	●	●	●	●
9	●	●	●	●
10	●	●	●	●
11	●	●	●	●
12	●	●	●	●
13	✂	●	●	●
14	●	●	●	●
15	●	●	●	●
16	✂	●	●	●
17	●	●	●	●
18	●	●	●	●
19	✂	●	●	●

Figure 2.4: Safety for patched program

Chapter 3

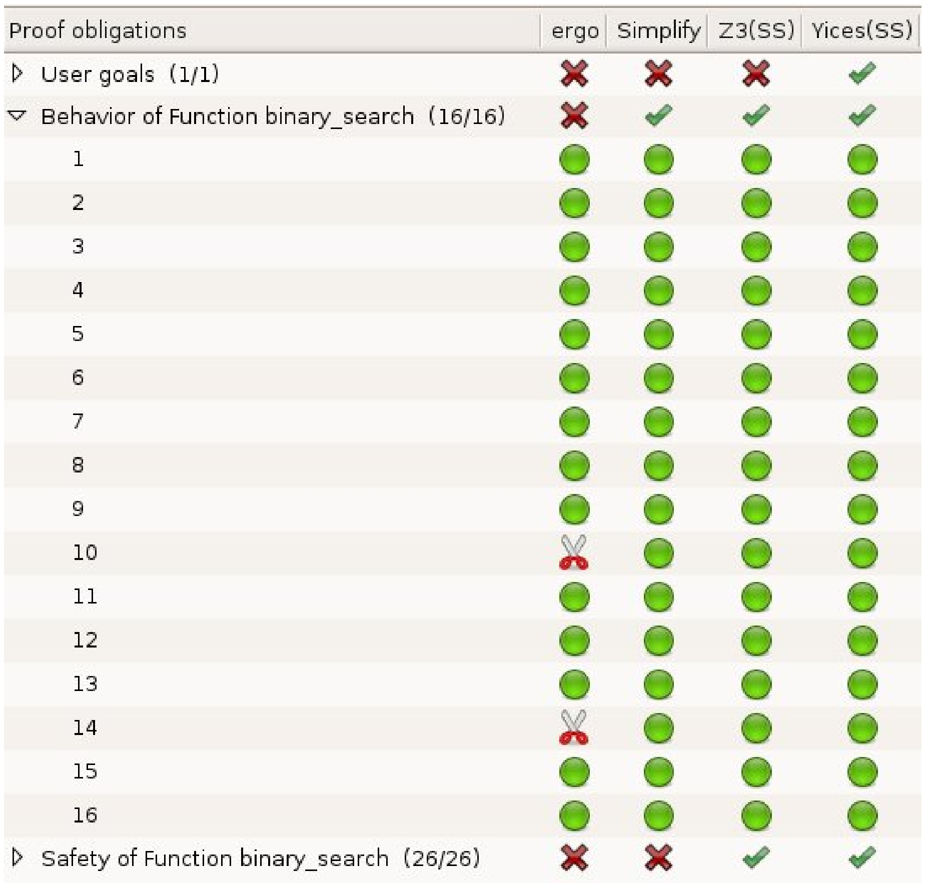
Functional Verification

3.1 Behaviors

Now that the safety of function `binary_search` is proved, one can attempt the verification of functional properties, like the input-output behavior of function `binary_search`. At the simplest, one can add a postcondition that `binary_search` should respect upon returning to its caller. Here, we add bounds on the value returned by `binary_search`. To prove this postcondition, strengthening the loop invariant is necessary.

```
/*@ lemma mean : \forall integer x, y; x <= y ==> x <= (x+y)/2 <= y; */

/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ ensures -1 <= \result <= n-1;
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  /*@ loop invariant
    @   0 <= l && u <= n-1 && -1 <= p <= n-1;
    @*/
  while (l <= u) {
    int m = l + (u - l) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

As shown in Figure 3.1, all VC are proved automatically here.



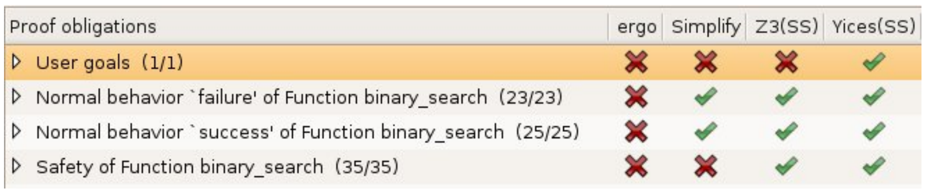
Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)
▷ User goals (1/1)	✗	✗	✗	✓
▽ Behavior of Function binary_search (16/16)	✗	✓	✓	✓
1	●	●	●	●
2	●	●	●	●
3	●	●	●	●
4	●	●	●	●
5	●	●	●	●
6	●	●	●	●
7	●	●	●	●
8	●	●	●	●
9	●	●	●	●
10	✂	●	●	●
11	●	●	●	●
12	●	●	●	●
13	●	●	●	●
14	✂	●	●	●
15	●	●	●	●
16	●	●	●	●
▷ Safety of Function binary_search (26/26)	✗	✗	✓	✓

Figure 3.1: General postcondition

One can be more precise and separate the postcondition according to different behaviors. The *assumes* clause of a behavior gives precisely the context in which a behavior applies. Here, we state that function `binary_search` has two modes: a success mode and a failure mode. This directly relies on array `t` to be sorted, thus we add this as a general requirement. The success mode states that whenever the calling context is such that value `v` is in the range of `t` searched, then the value returned is a valid index. The failure mode states that whenever the calling context is such that value `v` is not in the range of `t` searched, then function `binary_search` returns `-1`. Again, it is necessary to strengthen the loop invariant to prove the VC generated.

```
/*@ lemma mean : \forall int x, y; x <= y ==> x <= (x+y)/2 <= y; */
/*@ requires
  @   n >= 0 && \valid_range(t,0,n−1) &&
  @   \forall int k1, int k2; 0 <= k1 <= k2 <= n−1 ==> t[k1] <= t[k2];
  @ behavior success:
  @   assumes \exists int k; 0 <= k <= n−1 && t[k] == v;
  @   ensures 0 <= \result <= n−1;
  @ behavior failure:
  @   assumes \forall int k; 0 <= k <= n−1 ==> t[k] < v || t[k] > v;
  @   ensures \result == −1;
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n−1, p = −1;
  /*@ loop invariant
    @   0 <= l && u <= n−1 && −1 <= p <= n−1
    @   && (p == −1 ==> \forall int k; 0 <= k < n ==> t[k] == v ==> l <= k <= u)
    @   && (p >= 0 ==> t[p]==v);
    @*/
  while (l <= u) {
    int m = l + (u − l) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m − 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

Figure 3.2 summarizes the results obtained in that case, for each behavior.



Proof obligations	ergo	Simplify	Z3(SS)	Yices(SS)
▷ User goals (1/1)	✗	✗	✗	✓
▷ Normal behavior `failure' of Function binary_search (23/23)	✗	✓	✓	✓
▷ Normal behavior `success' of Function binary_search (25/25)	✗	✓	✓	✓
▷ Safety of Function binary_search (35/35)	✗	✗	✓	✓

Figure 3.2: Postconditions in behaviors

3.2 Advanced Algebraic Modeling

The following example introduces use of algebraic specification. The goal is the verify a simple sorting algorithm (by extraction of the minimum).

The first step is to introduce logical predicates to define the meanings for an array to be sorted in increasing order, to be a permutation of another. This is done as follows, in a separate file say `sorting.h`

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

/*@ inductive Permut{L1,L2}(int a[], integer l, integer h) {
  @  case Permut_refl{L}:
  @   \forall int a[], integer l, h; Permut{L,L}(a, l, h) ;
  @  case Permut_sym{L1,L2}:
  @    \forall int a[], integer l, h;
  @      Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
  @  case Permut_trans{L1,L2,L3}:
  @    \forall int a[], integer l, h;
  @      Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
  @        Permut{L1,L3}(a, l, h) ;
  @  case Permut_swap{L1,L2}:
  @    \forall int a[], integer l, h, i, j;
  @       l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==>
  @     Permut{L1,L2}(a, l, h) ;
  @ }
  @*/

/*@ predicate Sorted{L}(int a[], integer l, integer h) =
  @   \forall integer i; l <= i < h ==> a[i] <= a[i+1] ;
  @*/
```

The code is then annotated using these predicates as follows

```
#pragma JessieIntegerModel(math)

#include "sorting.h"

/*@ requires \valid(t+i) && \valid(t+j);
  @ assigns t[i],t[j];
  @ ensures Swap{Old,Here}(t,i,j);
  @*/
void swap(int t[], int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}

/*@ requires \valid_range(t,0,n-1);
  @ behavior sorted:
  @   ensures Sorted(t,0,n-1);
  @ behavior permutation:
  @   ensures Permut{Old,Here}(t,0,n-1);
  @*/
void min_sort(int t[], int n) {
  int i,j;
  int mi,mv;
  if (n <= 0) return;
  /*@ loop invariant 0 <= i < n;
    @ for sorted:
    @  loop invariant
    @   Sorted(t,0,i) &&
    @   (\forall integer k1, k2 ;
    @      0 <= k1 < i <= k2 < n ==> t[k1] <= t[k2]) ;
    @ for permutation:
    @  loop invariant Permut{Pre,Here}(t,0,n-1);
    @ loop variant n-i;
    @*/
  for (i=0; i<n-1; i++) {
    // look for minimum value among t[i..n-1]
    mv = t[i]; mi = i;
    /*@ loop invariant i < j && i <= mi < n;
      @ for sorted:
      @  loop invariant
      @    mv == t[mi] &&
      @    (\forall integer k; i <= k < j ==> t[k] >= mv);
      @ for permutation:
      @  loop invariant
      @   Permut{Pre,Here}(t,0,n-1);
      @ loop variant n-j;
      @*/
    for (j=i+1; j < n; j++) {
      if (t[j] < mv) {
        mi = j ; mv = t[j];
      }
    }
    swap(t,i,mi);
  }
}
```

Chapter 4

Inference of Annotations

4.1 Postconditions and Loop Invariants

To alleviate the annotation burden, it is possible to ask the Jessie plug-in to infer some of them, through a combination of abstract interpretation and weakest preconditions. This requires that APRON library for abstract interpretation is installed and Frama-C configuration recognized it. Then, one can call

```
 > framac -jessie -jessie-infer-annot inv max.c
```

to perform abstract interpretation on program `max.c`, which computes necessary loop invariants and postconditions (meaning an overapproximation of the real ones).

```
int max(int *r, int* i, int* j) {
  if (!r) return -1;
  *r = (*i < *j) ? *j : *i;
  return 0;
}
```

On our unannotated `max.c` program, this produces postcondition `true` for the first return and `\valid(r) && \valid(i) && \valid(j)` for the second return.

Various domains from APRON library are available with option `-jessie-abstract-domain`:

- *box* - domain of intervals, where an integer variables is bounded by constants.
- *oct* - domain of octagons, where the sum and difference of two integer variables are bounded by constants.
- *poly* - domain of polyhedrons, computing linear relations over integer variables.

4.2 Preconditions and Loop Invariants

Preconditions can also be computed by calling

```
> framac -jessie -jessie-infer-annot pre max.c
```

which attemps to compute a sufficient precondition to guard against safety violations and prove functional properties. In case it computes `false` as sufficient precondition, which occurs e.g. each time the property is beyond the capabilities of our method, it simply ignores it. Still, our method can compute a stronger precondition than necessary. E.g., on function `max`, it computes precondition `\valid(r) && \valid(i) && \valid(j)`, while a more precise precondition would allow `r` to be null. Still, the generated precondition is indeed sufficient to prove the safety of function `max`:

```
Running Simplify on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
simplify/max_why.sx           : ......... (9/0/0/0/0)
```

To improve on the precision of the generated precondition, various methods have been implemented:

- *Quantifier elimination* - This method computes an invariant `I` at the program point where check `C` should hold, forms the quantified formula `\forall x,y... ; I ==> C` over local variables `x,y...`, and eliminates quantifiers from this formula, resulting in a suffcient precondition. This is the method called with option `-jessie-infer-annot pre`.
- *Weakest preconditions with quantifier elimination* - This method improves on direct quantifier elimination by propagating formula `I ==> C` backward in the control-flow graph of the function before quantifying over local variables and eliminating quantifiers. This is the method called with option `-jessie-infer-annot wpre`.
- *Modified weakest preconditions with quantifier elimination* - This method strengthen the formula obtained by weakest preconditions with quantifier elimination, by only considering tests and assignments which deal with variables in the formula being propagated. Thus, it may result in a stronger precondition (i.e. a precondition less precise) but at a smaller computational cost. In particular, it may be applicable to programs where weakest preconditions with quantifier elimination is too costly. This is the method called with option `-jessie-infer-annot spre`.

Chapter 5

Separation of Memory Regions

By default, the Jessie plug-in assumes different pointers point into different memory *regions*. E.g., the following postcondition can be proved on function `max`, because parameters `r`, `i` and `j` are assumed to point into different regions.

```
/*@ requires \valid(i) && \valid(j);
  @ requires r == NULL || \valid(r);
  @ ensures *i == \old(*i) && *j == \old(*j);
  @*/
int max(int *r, int* i, int* j) {
  if (!r) return -1;
  *r = (*i < *j) ? *j : *i;
  return 0;
}
```

To change this default behavior, call instead

```
> frama-c -jessie -jessie-no-regions max.c
```

In this setting, the postcondition cannot be proved:

```
Running Simplify on proof obligations
(. = valid * = invalid ? = unknown # = timeout ! = failure)
simplify/max_why.sx           : ?..?........ (10/0/2/0/0)
```

Now, function `max` should only be called in a context where parameters `r`, `i` and `j` indeed point into different regions, like the following:

```
int main(int a, int b) {
  int c;
  max(&c,&a,&b);
  return c;
}
```

In this context, all VC are proved.

In fact, regions that are only read, like the regions pointed to by `i` and `j`, need not be disjoint. Since nothing is written in these regions, it is still correct to prove their contract in a context where they are assumed disjoint, whereas they may not be disjoint in reality. It is the case in the following context:

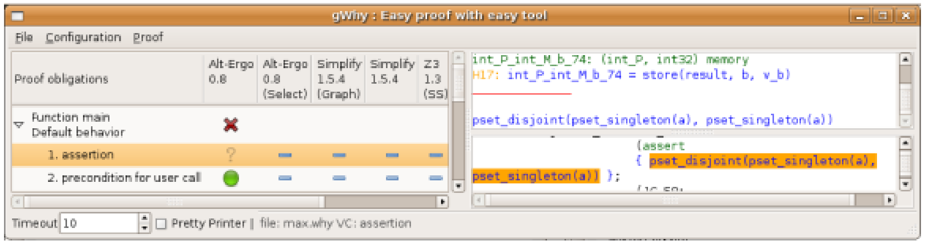
```
int main(int a, int b) {
  int c;
  max(&c,&a,&a);
  return c;
}
```

In this context too, all VC are proved.

Finally, let's consider the following case of a context in which a region that is read and a function that is written are not disjoint:

```
int main(int a, int b) {
  int c;
  max(&a,&a,&b);
  return c;
}
```

The proof that regions are indeed disjoint boils down to proving that set of pointers `{&a}` and `{&a}` are disjoint (because function `max` only writes and reads `*r` and `*i`), which is obviously false.



Proof obligations	Alt-Ergo 0.8	Alt-Ergo 0.8 (Select)	Simplify 1.5.4 (Graph)	Simplify 1.5.4	Z3 1.3 (SS)
Function main Default behavior	✗				
1. assertion	?	−	−	−	−
2. precondition for user call	●	−	−	−	−

```
int_P_int_M_b_74: (int_P, int32) memory
H17: int_P_int_M_b_74 = store(result, b, v_b)

pset_disjoint(pset_singleton(a), pset_singleton(a))
```

```
                    (assert
                    { pset_disjoint(pset_singleton(a),
pset_singleton(a)) };
```

Timeout 10 Pretty Printer | file: max.why VC: assertion

Chapter 6

Treatment of Unions and Casts

Unions without pointer fields are translated to bitvectors, so that access in these unions are translated to low-level accesses. Thus, the following code can be analyzed, but we do not yet provide a way to prove the resulting assertions, by asserting that any subset of bits from the bitvector representation of 0 is 0:

```
union U {
  int i;
  struct { short s1; short s2; } s;
};

//@ requires \valid(x);
void zero(union U* x) {
  x->i = 0;
  //@ assert x->s.s1 == 0;
  //@ assert x->s.s2 == 0;
}
```

Unions with pointer fields (either direct fields or sub-fields of structure fields) are translated differently, because we treat pointers differently than other types, to allow an automatic analysis of separation of memory blocks. Thus, we treat unions with pointer fields as discriminated unions, so that writing in a field erases all information on other fields. This allows to verify the following program:

```
union U {
  int i;
  int* p;
};

//@ requires \valid(x);
void zero(union U* x) {
  x->i = 0;
  //@ assert x->i == 0;
  x->p = (int*)malloc(sizeof(int));
  *x->p = 1;
  //@ assert *x->p == 1;
}
```

Finally, casts between pointer types are allowed, with the corresponding accesses to memory treated as low-level accesses to some bitvector. This allows to verify the safety of the following program:

```
//@ requires \valid(x);
void zero(int* x) {
  char *c = (char*)x;
  *c = 0;
  c++;
  *c = 0;
  c++;
  *c = 0;
  c++;
  *c = 0;
}
```

Notice that unions are allowed in logical annotations, but not pointer casts yet.

Chapter 7

Reference Manual

7.1 General usage

The Jessie plug-in is activated by passing option `-jessie` to `frama-c`. Running the Jessie plug-in on a file f.jc produces the following files:

- f.jessie: sub-directory where every generated files go
- f.jessie/f.jc: translation of source file into intermediate Jessie language
- f.jessie/f.cloc: trace file for source locations

The plug-in will then automatically call the Jessie tool of the Why platform to analyze the generated file f.jc above. By default, VCs are generated for the Simplify theorem prover, and this prover is run over these VCs, the result being displayed on standard output. The `-jessie-atp` option allows to specify an alternate theorem prover.

If the option `-framac -jessie` is given, then instead of running a prover, the VCs will be displayed inside the GWhy interface.

7.2 Unsupported features

7.2.1 Unsupported C features

Arbitrary gotos only forward gotos, not jumping into nested blocks, are allowed. There is no plan to support arbitrary gotos in the future.

Function pointers

Arbitrary cast
- from integers to pointers, from pointer to integers: no support
- between pointers: experimental support, only for casts in code, not logic

Note: casts between integer types are supported

Union types experimental support, both in code and annotations

Variadic C functions unsupported

Floating point computations By default, float and double numbers are interpreted as reals. Consequently, it is not possible to check properties relaed to rounding errors or overflows. Experimental supports for other models are described below, see description of the FloatModel pragma

7.2.2 partially supported ACSL features

Inductive predicates supported, but must follow the positive Horn clauses style presented in the ACSL documentation.

Axiomatic declarations supported (experimental)

7.2.3 Unsupported ACSL features

Contract clauses

- loop assigns
- complete, disjoint behaviors
- terminates clauses
- general code invariants (only loop invariants are supported)

Logic specifications

- polymorphic types in logic declarations
- model variables and fields
- global invariants and type invariants
- higher-order constructs `\lambda`, `\sum`, `\prod` ...
- `\let` construct
- array and structure field functional modifiers
- `volatile` declarations
- `\initialized` and `\specified` predicates

Ghost code

- it is not checked whether ghost code does not interfere with program code.
- ghost structure fields are not supported

7.3 Command-line options

-jessie activates the plugin, to perform C to Jessie translation

-jessie-project-name <s> specify project name for Jessie analysis

-jessie-cpu-limit <i> set the time limit in sec. for the analysis

-jessie-atp <s> do not launch the GUI but run specified automated theorem prover in batch (among `alt-ergo`, `cvc3`, `simplify`, `yices`, `z3`), or just generate the verification conditions (`goals`)

-jessie-int-model <s> Warning: will become obselete, use pragma JessieIntegerModel instead

set the model for integer arithmetic (exact, bounded or modulo)

-jessie-behavior <s> restrict verification to the given behavior (safety, default or a user-defined behavior)

-jessie-std-stubs use annotated standard headers

-jessie-hint-level <i> level of hints, i.e. assertions to help the proof (e.g. for string usage)

-jessie-infer-annot <s> infer function annotations (inv, pre, spre, wpre)

-jessie-abstract-domain <s> use specified abstract domain (box, oct or poly)

-jc-opt <s> give an option to the jessie tool (e.g., -trust-ai)

-why-opt <s> give an option to Why (e.g., -fast-wp)

7.4 Pragmas

Integer model

`# pragma JessieIntegerModel`(value)

Possible values: `exact`, `math`, `strict`, `modulo`

- `exact` or `math`: all int types are modeled by mathematical unbounded integers
- `strict`: int types are modeled by integers with appropriate bounds, and for each arithmetic operations, it is mandatory to show that no overflow occur
- `modulo`: models exactly machine integer arithmetics, allowing overflow, that is results must be taken modulo 2^n for the appropriate n for each type.

Floating point model

`# pragma FloatModel`(value)

Possible values: `real`, `math`, `strict`, `full`

- `real` or `math`: all float types are modeled by mathematical unbounded real numbers

- `strict`: float types are modeled by real numbers with appropriate bounds are roundings, and for each floating-point arithmetic operations, it is mandatory to show that no overflow occur. This model is based on [1].
- `full`: models exactly floating-point arithmetics, allowing infinite values or nans

Separation policy

`# pragma SeparationPolicy`(value)

Possible values: `none`, `regions`

Invariant policy

`# pragma InvariantPolicy`(value)

Possible values: `none`, `arguments`, `ownership`

Bibliography

[1] Sylvie Boldo and Jean-Christophe Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE International Symposium on Computer Arithmetic*, pages 187–194, Montpellier, France, June 2007.