

Jessie Plugin Tutorial

Frama-C version: *Boron*
Jessie plugin version: 2.26

Claude Marché^{1,3}, Yannick Moy^{2,3},

June 13, 2010

¹ INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893

² France Télécom, Lannion, F-22307

³ LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Contents

1	Introduction	2
1.1	Batch Mode vs. GUI Mode	2
1.2	Basic Use	2
1.3	Safety Checking vs. Functional Verification	5
2	Safety Checking	8
2.1	Memory Safety	8
2.2	Integer Overflow Safety	9
2.3	Checking Termination	11
3	Functional Verification	13
3.1	Behaviors	13
3.1.1	Simple functional property	13
3.1.2	More advanced functional properties	13
3.2	Advanced Algebraic Modeling	15
4	Inference of Annotations	18
4.1	Postconditions and Loop Invariants	18
4.2	Preconditions and Loop Invariants	18
5	Separation of Memory Regions	20
6	Treatment of Unions and Casts	22
7	Reference Manual	24
7.1	General usage	24
7.2	Unsupported features	24
7.2.1	Unsupported C features	24
7.2.2	partially supported ACSL features	25
7.2.3	Unsupported ACSL features	25
7.3	Command-line options	25
7.4	Pragmas	26

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 in its GUI mode. To invoke this mode on a file `ex.c`, just type

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

The GUI of the Why Platform (a.k.a. GWhy) is called. It displays each VC in a row, with available provers arranged as columns.

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

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

runs prover Simplify on the generated VC. Valid identifiers for provers are documented in Why. The list includes `alt-ergo`, `cvc3`, `yices`, `z3`. See also the prover tricks page <http://why.lri.fr/provers.html>.

Finally, you can use the generic name `goals` to just ask for generation of VC 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) {  
    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
```

Frama-C

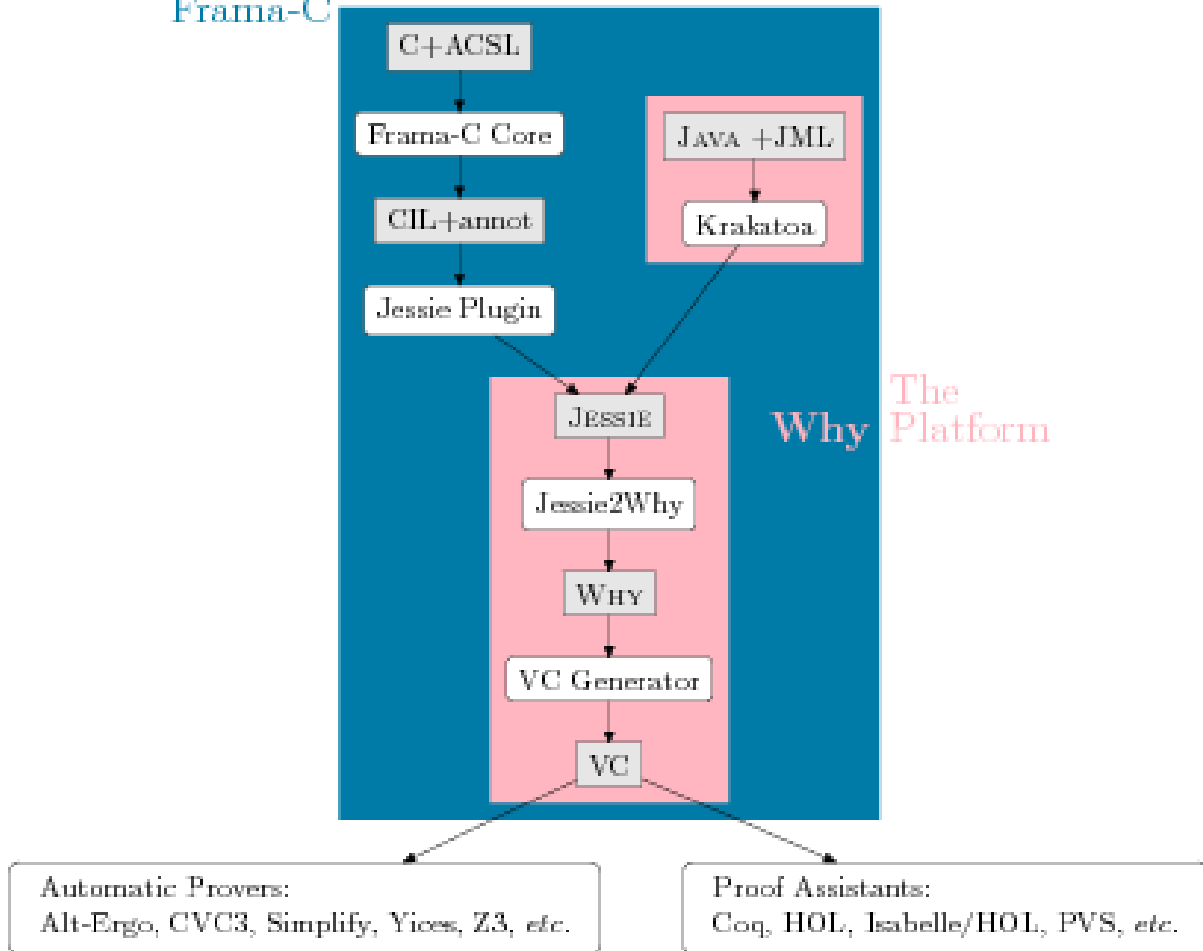


Figure 1.1: Frama-C and the Why Platform

```

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)
  
```

Simplify is an automatic theorem prover distributed separately from Why and Jessie. Among the existing automatic provers, it is one that produces good results for software verification. Another automatic prover that produces good results is Alt-ergo.

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 occurred. 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 any functional property.

Consider now adding a postcondition to function `max`:

```

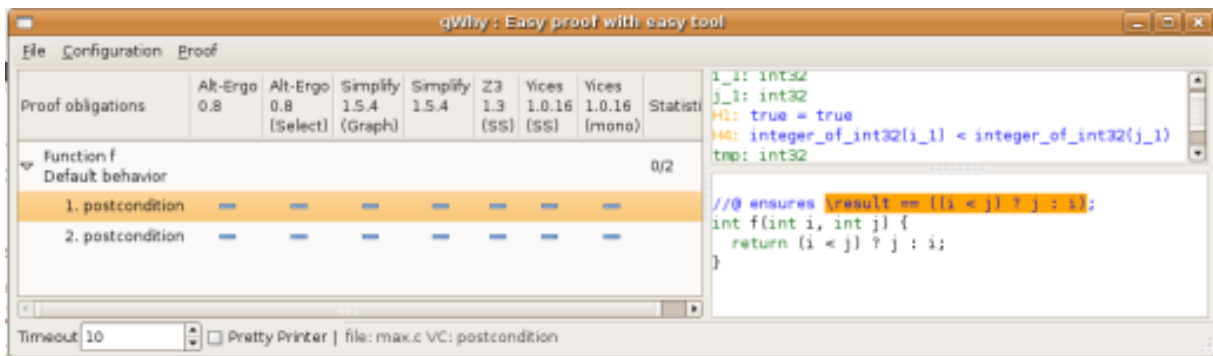
//@ ensures \result == ((i < j) ? j : i);
int max(int i, int j) {
    return (i < j) ? j : i;
}
  
```

```
}
```

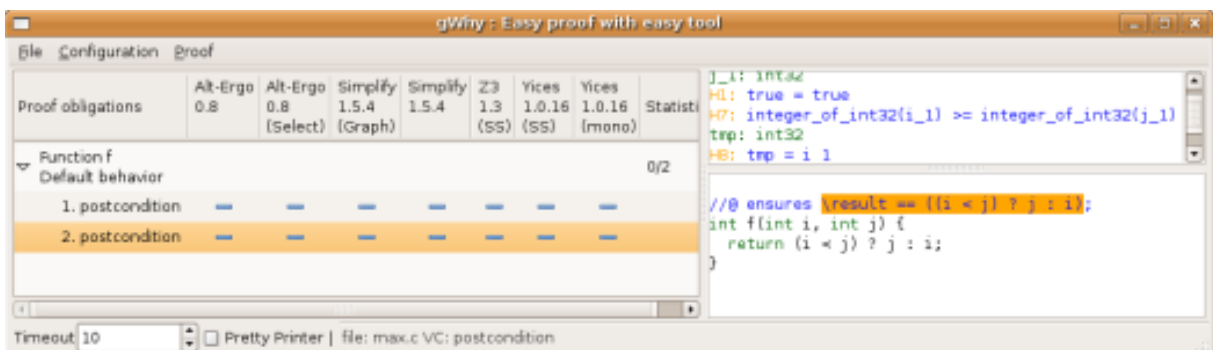
This ACSL annotation expresses the fact function `max` returns the maximum of its parameters `i` and `j`. Now, running the Jessie plug-in in batch mode outputs:

```
...
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
```

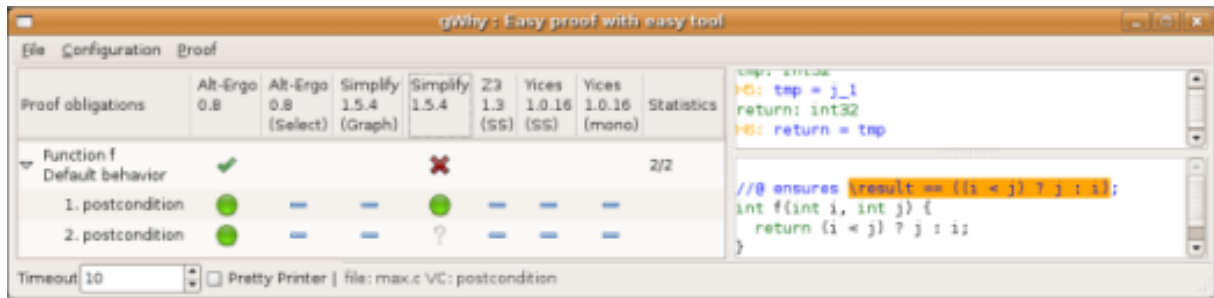
This says that Simplify could prove one VC and not the other. To see what these VC represent, we call now the Jessie plug-in in GUI mode. Each VC represents verification of the postcondition in a different context. The first VC represents the context where $i < j$:



The second VC represents the context where $i \geq j$. 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.



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



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 in 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;
}

```

Notice that the annotations refer to the null pointer using ACSL syntax `\null`. It would be possible to use also the C macro `NULL`, but in that case we would have to ask Frama-C preprocessor phase to process the annotations too, since it does not by default. This is done by option `-pp-annot` of Frama-C. However, this alternative is not recommended since it is depended of the proprocessor in use (see http://bts.frama-c.com/dokuwiki/doku.php?id=mantis:frama-c:start#faq_tips_and_tricks)

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.

The screenshot shows the gWhy tool interface. On the left, a table lists proof obligations and their verification status across six solvers: Alt-Ergo 0.8, Alt-Ergo 0.8 (Select), Simplify 1.5.4 (Graph), Simplify 1.5.4, Z3 1.3 (SS), and Yices 1.0.16 (SS). The obligations are grouped into four categories: Function max Default behavior, Function max Normal behavior 'normal', Function max Normal behavior 'zero', and Function max Safety. The right side shows a code editor with C code and annotations for a function named 'max'.

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 the verification of functional properties using the Jessie plug-in is to verify the safety of functions. Safety has several components: memory safety, integer safety, termination. 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. Termination amounts to check whether loops are always terminating, as well as recursive or mutually recursive functions.

2.1 Memory Safety

Our running example will be the famous `binary_search` function, which searches for a `long` in an ordered array of `longs`. On success, it returns the index at which the element appears in the array. On failure, it returns `-1`.

```
#pragma JessieIntegerModel(math)
#pragma JessieTerminationPolicy(user)

int binary_search(long t[], int n, long v) {
    int l = 0, 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 return m;
    }
    return -1;
}
```

To concentrate first on memory safety only, we declare two pragmas as above. The first pragma dictates that integers in C programs behave as infinite-precision mathematical integers, without overflows. The second pragma instructs the plugin to ignore termination issues.

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

```
> frama-c -jessie 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.

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 which requires `n` to be positive and array `t` to be valid between

Proof obligations	Alt-Ergo 0.9	Simplify 1.5.4	Z3 2.2 (SS)	Yices 1.0.24 (SS)	CVC3 2.1 (SS)	Statistics
Function binary_search	✘	✘	✘	✘	✘	1/3
Safety						
1. check division by zero	✔	✔	✔	✔	✔	
2. pointer dereferencing	?	?	✘	✘	✘	
3. pointer dereferencing	?	?	✘	✘	✘	

Figure 2.1: Memory safety with no annotations

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. 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(long t[], int n, long v) {
    int l = 0, u = n-1;
    //@ loop invariant 0 <= l && u <= n-1;
    while (l <= u) {
        ...
    }
```

There are now 7 VC generated: 2 to guarantee the loop invariant is initially established (because the conjunction 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` in the loop, that is difficult to take into account automatically.

We solve this problem by adding an assertion to help automatic provers, providing some form of hint in the proof. This can be done by inserting assertions in the code, or to add a *global lemma*, that should be proved using available axioms, and used as an axiom in proving the VC for safety. This works for our example.

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

//@ requires n >= 0 && \valid_range(t, 0, n-1);
int binary_search(long t[], int n, long v) {
    ...
}
```

The results are shown on Figure 2.2, where all VCs, are proved by some prover. This guarantees the memory safety of function `binary_search`. The lemma itself is proved by Alt-Ergo, which has a little knowledge of the division operator. Given the lemma, other VCs are fully proved by Simplify and Z3, and partly by Yices and CVC3.

2.2 Integer Overflow Safety

Let us now consider machine integers instead of idealized mathematical integers. This is obtained by removing the pragma `JessieIntegerModel`. Without this pragma, integer types are now interpreted as bounded machine

Proof obligations	Alt-Ergo 0.9	Simplify 1.5.4	Z3 2.2 (SS)	Yices 1.0.24 (SS)	CVC3 2.1 (SS)	Statistics
▼ User goals	✓	✗	✗	✗	✗	1/1
Lemma mean	✓	?	?	?	?	
▼ Function binary_search						
Default behavior	✓	✓	✓	✗	✗	4/4
1. loop invariant initially holds	✓	✓	✓	✓	✓	
2. loop invariant initially holds	✓	✓	✓	✓	✓	
3. loop invariant preserved	✓	✓	✓	?	?	
4. loop invariant preserved	✓	✓	✓	?	?	
▼ Function binary_search						
Safety	✓	✓	✓	✗	✗	3/3
1. check division by zero	✓	✓	✓	✓	✓	
2. pointer dereferencing	✓	✓	✓	?	?	
3. pointer dereferencing	✓	✓	✓	?	?	

Figure 2.2: Memory safety with precondition and loop invariant

The screenshot shows the gWhy verification conditions viewer. The left pane displays a table of proof obligations, and the right pane shows the corresponding C code with assertions.

Proof obligations	Alt-Ergo 0.9	Simplify 1.5.4	Z3 2.2 (SS)	Yices 1.0.24 (SS)	CVC3 2.1 (SS)	Statistics
▶ User goals	✓	✗	✗	✗	✗	1/1
▶ Function binary_search						
Default behavior	✓	✓	✗	✗	✗	4/4
▼ Function binary_search						
Safety	✗	✗	✗	✗	✗	12/13
1. check arithmetic overflow	✓	?	✓	✓	✓	
2. check arithmetic overflow	✓	✓	✓	✓	✓	
3. check arithmetic overflow	✓	✓	✓	✓	✓	
4. check arithmetic overflow	?	?	?	?	?	
5. check division by zero	✓	✓	✓	✓	✓	
6. check arithmetic overflow	✓	✓	?	?	?	
7. check arithmetic overflow	✓	✓	?	?	?	
8. pointer dereferencing	✓	✓	?	?	?	
9. pointer dereferencing	✓	✓	?	?	?	
10. check arithmetic overflow	✓	✓	✓	✓	✓	
11. check arithmetic overflow	✓	✓	?	?	?	
12. check arithmetic overflow	✓	?	?	?	?	
13. check arithmetic overflow	✓	✓	✓	✓	✓	

```

H10: Integer_of_int32(result0) = Integer_of_int32(n) - 1
u: int32
H11: u = result0
l0: int32
u0: int32
H12: true
H13: (0 <= integer_of_int32(l0) and
integer_of_int32(u0) <= integer_of_int32(n) - 1)
H14: integer_of_int32(l0) <= integer_of_int32(u0)

integer_of_int32(l0) + integer_of_int32(u0) <= 2147483647

/*@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int l = 0, u = n-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 return m;
  }
  return -1;
}

```

Timeout: 10 Pretty Printer file: binary_search_ovfl.c VC: check arithmetic overflow

Figure 2.3: Memory safety + integer overflow safety

integers. However, the default is a *defensive* interpretation, which forbids the arithmetic operations to overflow.¹

The result can be seen in Figure 2.3. There are ten more VC to check that integer operations return a result

¹In a context where it is intended for the operations to overflow, and thus operations are intentionally done modulo, the same pragma should be set to the value `modulo`, see Jessie manual.

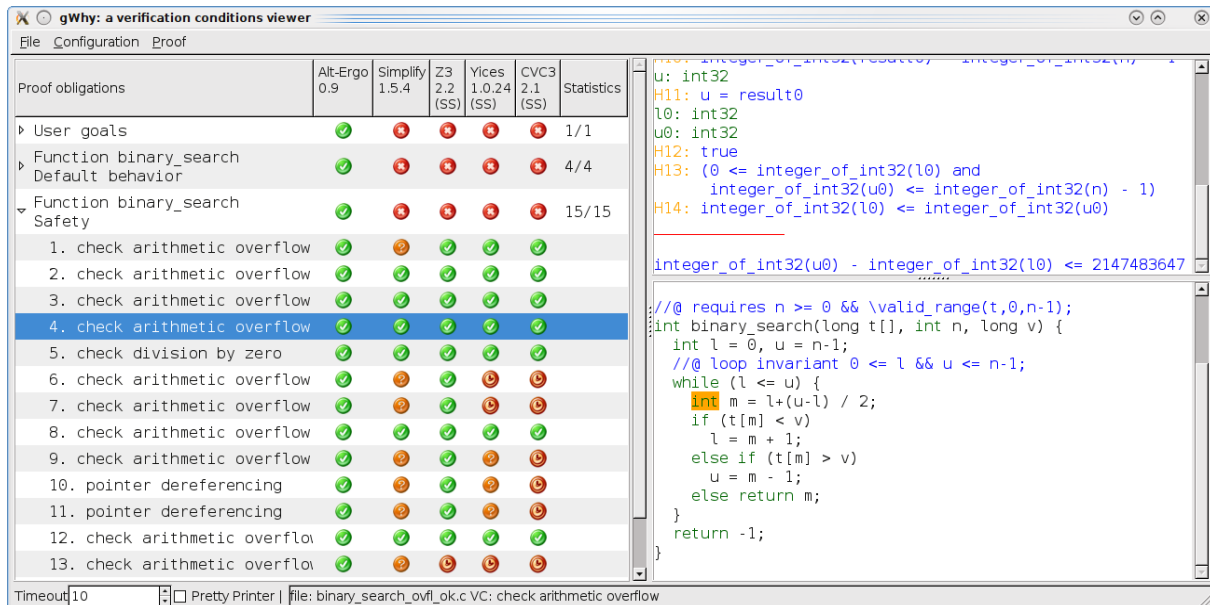


Figure 2.4: Safety for patched program

within bounds, only one of which is not proved. With this exception, the results are nearly the same as with exact integers (proving the lemma takes more time, due to the additional encoding for bounded integers).

The only unproved VC expresses that $l+u$ does not overflow. Nothing prevents this from happening with our current precondition for function `binary_search` [7]. There are two possibilities here. The easiest is to strengthen the precondition by requiring that n is no more than half the maximal signed integer `INT_MAX`. The best way 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.

2.3 Checking Termination

The last kind of safety property we want is termination. To check it, we first remove the pragma `JessieTerminationPolicy`. If we run the VC generation again, we get an additional VC which requires to prove the property $0 > 0$, which is false. This is because we did not provide any *loop variant* for the `while` loop. A loop variant is a quantity which must decrease strictly at each loop iteration, while provably remaining non-negative for as long as the loop runs. In this example, a proper variant is $u - l$. So our annotated program now looks as follows:

```

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

//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;
  //@ loop invariant 0 <= l && u <= n-1;

```

```

    @ loop variant u-1;
    @*/
while (l <= u) {
  int m = 1+(u-1) / 2;
  if (t[m] < v)
    l = m + 1;
  else if (t[m] > v)
    u = m - 1;
  else return m;
}
return -1;
}

```

The additional VC is now proved.

Termination of recursive functions can be dealt with similarly by adding a `decreases` clause to the function's contract. It is also possible to prove termination by using variants over any datatype d equipped with a well-founded relation. See the ACSL documentation for details.

Chapter 3

Functional Verification

3.1 Behaviors

3.1.1 Simple functional property

Now that the safety of function `binary_search` has been established, 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;
    /*@ loop invariant 0 <= l && u <= n-1;
       @ loop variant u-l;
       @*/
    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 return m;
    }
    return -1;
}
```

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

3.1.2 More advanced functional properties

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.

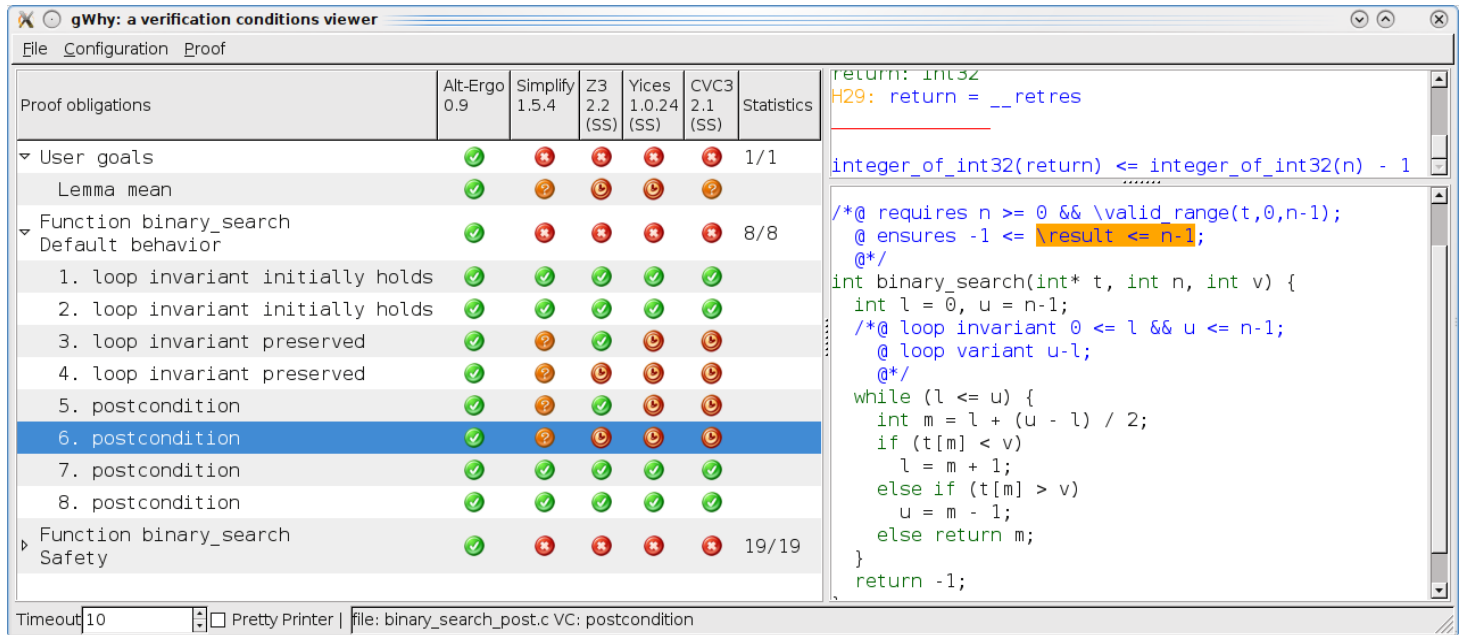


Figure 3.1: General postcondition

```

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

/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ behavior success:
  @   assumes // array t is sorted in increasing order
  @   \forall integer k1, k2; 0 <= k1 <= k2 <= n-1 ==> t[k1] <= t[k2];
  @   assumes // v appears somewhere in the array t
  @   \exists integer k; 0 <= k <= n-1 && t[k] == v;
  @   ensures 0 <= \result <= n-1;
  @ behavior failure:
  @   assumes // v does not appear anywhere in the array t
  @   \forall integer k; 0 <= k <= n-1 ==> t[k] != v;
  @   ensures \result == -1;
  @*/

int binary_search(long t[], int n, long v) {
  int l = 0, u = n-1;
  /*@ loop invariant 0 <= l && u <= n-1;
    @ for success:
    @   loop invariant
    @   \forall integer k; 0 <= k < n && t[k] == v ==> l <= k <= u;
    @ loop variant u-l;
    @*/
  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 return m;
  }
  return -1;
}

```

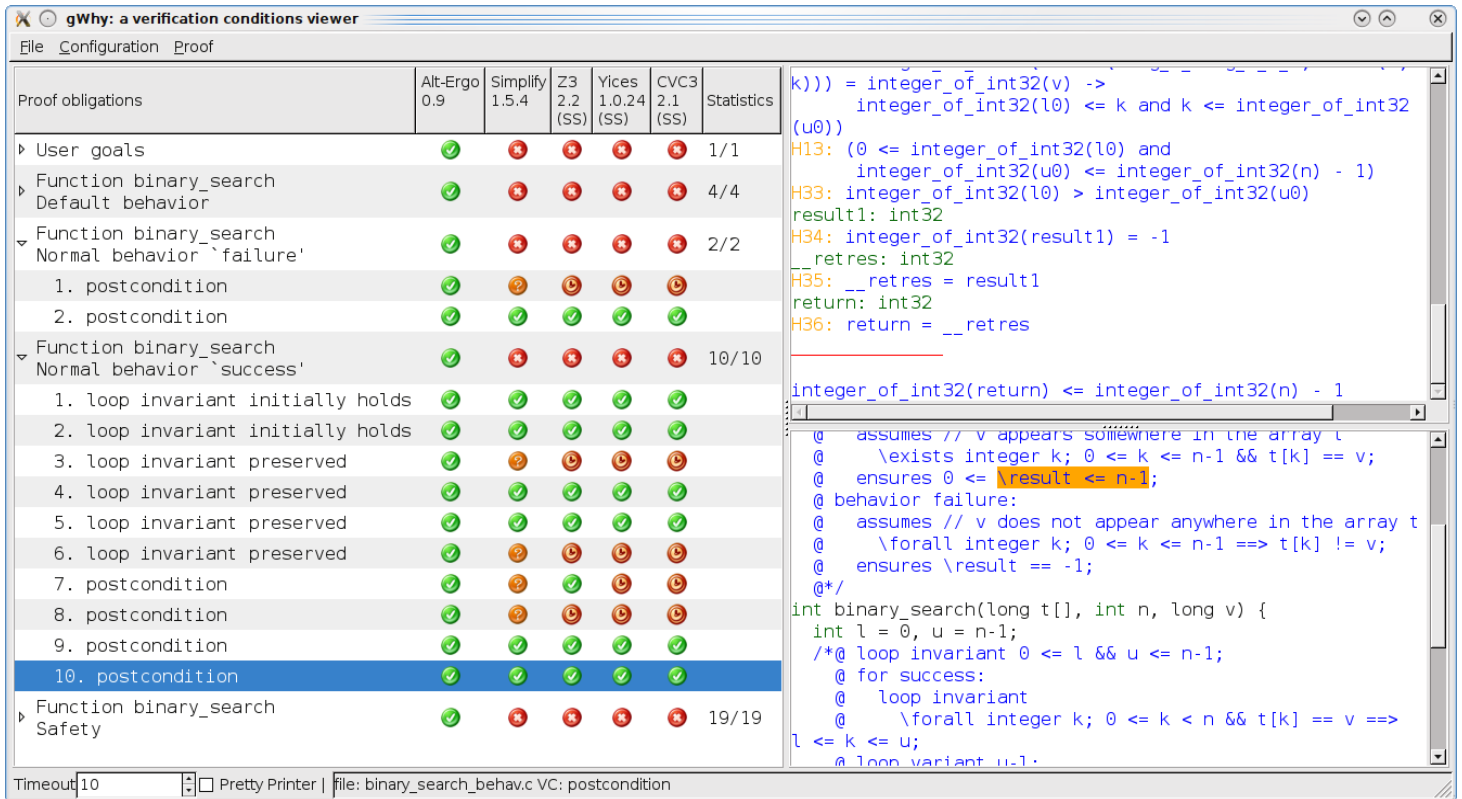


Figure 3.2: Postconditions in behaviors

}

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

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) &&
@   \forallall 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}:
@     \forallall int a[], integer l, h; Permut{L,L}(a, l, h) ;
@   case Permut_sym{L1,L2}:
@     \forallall 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:

```

The screenshot shows the gWhy verification conditions viewer. The left pane displays a table of proof obligations, and the right pane shows the corresponding C code with annotations.

Proof obligations	Alt-Ergo 0.9	Simplify 1.5.4	Z3 2.2 (SS)	Yices 1.0.24 (SS)	CVC3 2.1 (SS)	Statistics
Function min_sort Default behavior	✓	✓	✓	✗	✗	13/13
Function min_sort Normal behavior `permutation`	✓	✗	✗	✗	✗	3/3
1. postcondition	✓	✓	✓	✓	✓	
2. loop invariant initially holds	✓	✓	✓	✓	✓	
3. loop invariant preserved	✓	⚠	⚠	⚠	⚠	
Function min_sort Normal behavior `sorted`	✓	✗	✗	✗	✗	11/11
1. postcondition	✓	✓	✓	✓	✓	
2. loop invariant initially holds	✓	✓	✓	✓	✓	
3. loop invariant initially holds	✓	✓	✓	✓	✓	
4. loop invariant initially holds	✓	✓	✓	⚠	✓	
5. loop invariant initially holds	✓	✓	✓	⚠	✓	
6. loop invariant preserved	✓	✓	✓	⚠	✓	
7. loop invariant preserved	✓	✓	⚠	⚠	✓	
8. loop invariant preserved	✓	✓	✓	⚠	✓	
9. loop invariant preserved	✓	✓	✓	⚠	⚠	
10. loop invariant preserved	✓	✓	✓	⚠	⚠	
11. postcondition	✓	✓	✓	✓	✓	
Function min_sort Safety	✗	✓	✓	✗	✓	13/13
Function swap Default behavior	✗	✓	✓	✗	✓	2/2
1. postcondition	✓	✓	✓	⚠	✓	
2. postcondition	⚠	✓	✓	⚠	✓	
Function swap Safety	✓	✓	✓	✓	✓	3/3

The code editor on the right shows the C code for a minimum sort function with annotations. The highlighted line in the code is: `Permut{Pre,Here}(t,0,n-1);`

Figure 3.3: VCs for minimum sort

```

@ 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);
}
}

```

Each VC is proved by at least one prover. Figure 3.3 displays the results in GWhy, with emphasis on the VC for preservation of the loop invariant for permutation behavior, which is the most difficult one, only proved by Alt-Ergo.

Chapter 4

Inference of Annotations

Inference of annotations is an experimental feature of the Jessie plugin, whose theoretical bases are described in [5, 6].

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-atp=simplify -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-atp=simplify -jessie-infer-annot pre max.c
```

which attempts 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 $\forall x, y \dots ; I \implies C$ over local variables $x, y \dots$, and eliminates quantifiers from this formula, resulting in a sufficient 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 $I \implies C$ 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 strengthens 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, add the following line at the top of the file:

```
# pragma SeparationPolicy(none)
```

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:

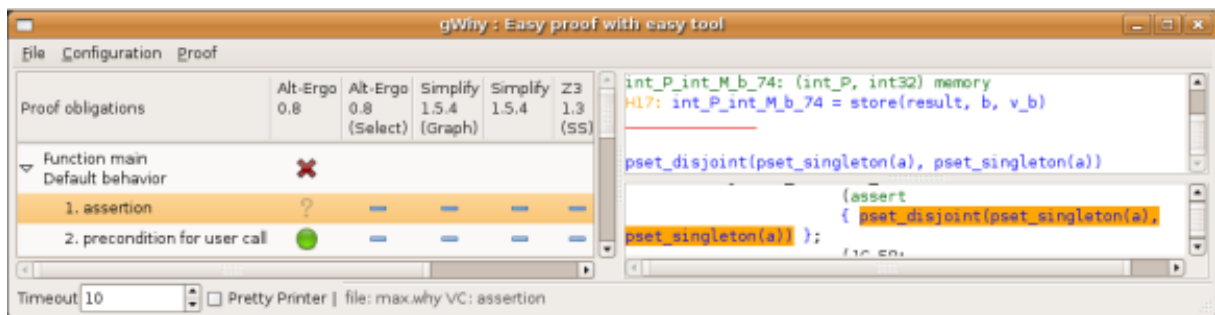
```
int main(int a, int b) {
  int c;
  max(&c, &a, &a);
  return c;
}
```

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.



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 and displayed in the GWhy interface. The `-jessie-atp=<p>` option allows to run VCs in batch, using the given theorem prover `<p>`.

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 a near future.

Function pointers

There is no plan to support them in a near future. One should consider using the specialization plugin to remove 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

`volatile` declaration modifier

not supported

`const` declaration modifier

accepted but not taken into account, that is treated as non-const.

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

Logic language

- direct equality on structures is not supported. Equality of each field should be used instead (e.g. by introducing an adequate predicate). Similarly, direct equality of arrays is not supported, and equality of each cells should be used instead.
- array and structure field functional modifiers are not supported
- higher-order constructs `\lambda`, `\sum`, `\prod`, etc. are not supported

Logic specifications

- model variables and fields
- global invariants and type invariants
- `volatile` declarations
- `\initialized` and `\specified` predicates

Contract clauses

- `terminates` clause
- abrupt termination clauses
- general code invariants (only loop invariants are supported)

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-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-cpu-limit=<i>` set the time limit in sec. for proving each VC. Only works when `-jessie-atp` is set.

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

`-jessie-std-stubs` (obsolete) 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`)

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

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

7.4 Pragmas

Integer model

```
# pragma JessieIntegerModel(value)
```

Possible values: `math`, `defensive`, `modulo`.

Default value: `defensive`

- `math`: all int types are modeled by mathematical, unbounded integers ;
- `defensive`: 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 JessieFloatModel(value)
```

Possible values: `math`, `defensive`, `full`, `multirounding`.

Default value: `defensive`.

- `math`: all float types are modeled by mathematical unbounded real numbers
- `defensive`: float types are modeled by real numbers with appropriate bounds and roundings, and for each floating-point arithmetic operations, it is mandatory to show that no overflow occur. This model follows the IEEE-754 standard, under its *strict* form, as explained in [1, 2].
- `full`: models the full IEEE-754 standard, including infinite values and NaNs. This model is the *full* model discussed in [1, 2].
- `multirounding`: models floating-point arithmetics so as to support combinations of compiler and architectures that do not strictly follows IEEE-754 standard (e.g. double roundings, 80-bits extended formats, compilation using fused-multiply-add instructions). This is based on paper [3, 4].

Floating point rounding mode

```
# pragma JessieFloatRoundingMode(value)
```

Possible values: `nearesteven`, `down`, `up`, `tozero`, `nearestaway`.

Default value: `nearesteven`.

Separation policy

```
# pragma SeparationPolicy(value)
```

Possible values: `none`, `regions`

Invariant policy

```
# pragma InvariantPolicy(value)
```

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

Termination policy

```
# pragma JessieTerminationPolicy(value)
```

Possible values: `always`, `never`, `user`

Default: `always`

- `always` means that every loop and every recursive function should be proved terminating. If they are not annotated by variants, then an unprovable VC ($0 < 0$) is generated.
- `user` means that VCs for termination are generated for each case where a loop or function variant is given. Otherwise no VC is generated.
- `never` means no VC for termination are ever generated, even for annotated loop or recursive function.

Bibliography

- [1] Ali Ayad and Claude Marché. Behavioral properties of floating-point programs. Hisseo publications, 2009. <http://hisseo.saclay.inria.fr/ayad09.pdf>.
- [2] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, Edinburgh, Scotland, July 2010. Springer.
- [3] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. Hisseo publications, 2009. <http://hisseo.saclay.inria.fr/tuyen09.pdf>.
- [4] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.
- [5] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.
- [6] Yannick Moy and Claude Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 2010. to appear.
- [7] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, Nice, France, January 2007.