The Jessie plugin

for Deductive Verification in Frama-C

Tutorial and Reference Manual

Version 2.39

Claude Marché, Yannick Moy

August 23, 2017
INRIA Team *Toccata* http://toccata.lri.fr
INRIA Saclay - Île-de-France & LRI, CNRS UMR 8623
Batiment 650, Université Paris-Sud 91405 Orsay cedex, France

Contents

Chapter 1

Introduction

Jessie is a plugin for the Frama-C environment, aimed at performing deductive verification of C programs, annotated using the ACSL language [4], using the Why [1] tool for generating proof obligations.

This version 2.39 of Jessie is compatible with Frama-C version Nitrogen (and no other).

1.1 Important note for version 2.30

The use of the Why2 VC generator is now obsolete, and it is recommended to switch to the Why3 system for specification and VC generation. Why3 must be installed independently of Why2, please see the instructions given at `http://why3.lri.fr`.

The version of Why3 that is compatible with this version 2.39 of Jessie is the version 0.71. Please see `http://krakatoa.lri.fr/` for more details on compatibility between Frama-C, Why2/Jessie and Why3.

In this manual, it is assumed that the Why3 VC generator and IDE is in use. The old behavior using the Why2 VC generator and GUI remains possible, using option `-jessie-atp=gui`.

1.2 Basic Use

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 (VCs) are generated and a prover is called on these, as sketched in Figure 1.1.

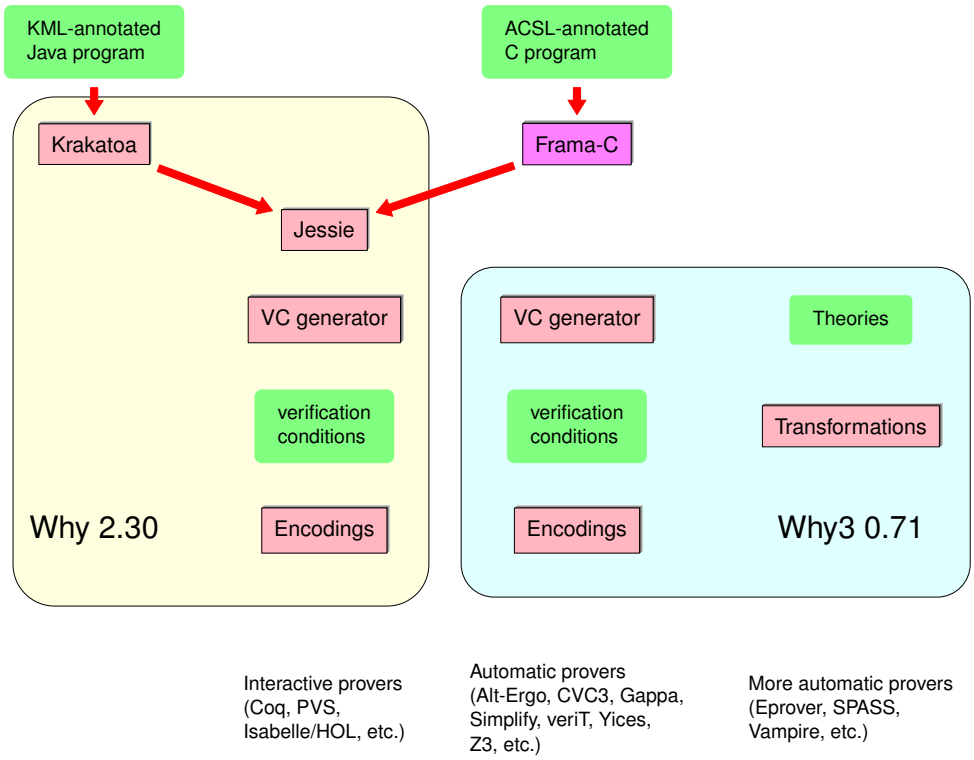
To prove the VCs generated, one needs to install external provers such as Alt-Ergo, CVC3 or Z3. Please see at URL `http://krakatoa.lri.fr/` how to get such provers. Once some of these are installed, you should run the auto-configuration tool by running command `why3config -detect` (equivalent to the former Why 2.xx command `why-config`)

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
```

A program does not need to be complete to be analyzed with the Jessie plug-in. As a first example, take program `max`:

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


KML-annotated
Java program
ACSL-annotated
C program
Krakatoa
Frama-C
Jessie
VC generator
verification
conditions
Encodings
Why 2.30
VC generator
verification
conditions
Encodings
Theories
Transformations
Why3 0.71
Interactive provers
(Coq, PVS,
Isabelle/HOL, etc.)
Automatic provers
(Alt-Ergo, CVC3, Gappa,
Simplify, veriT, Yices,
Z3, etc.)
More automatic provers
(Eprover, SPASS,
Vampire, etc.)

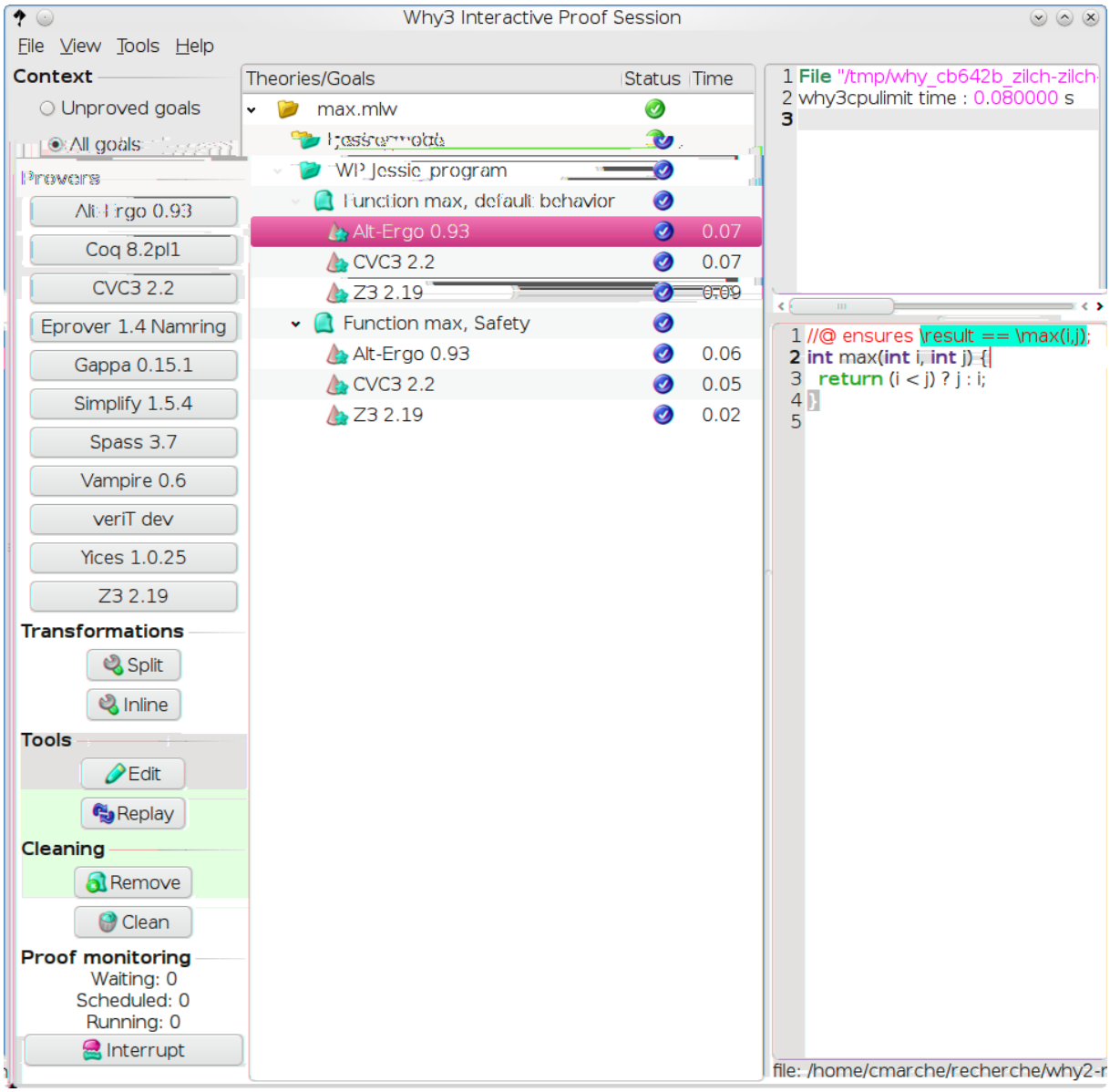


Figure 1.2: Why3 IDE for `max` function

```
  @ 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 == \max(*i,*j);
  @   ensures \result == 0;
  @*/
int max(int *r, int* i, int* j) {
```

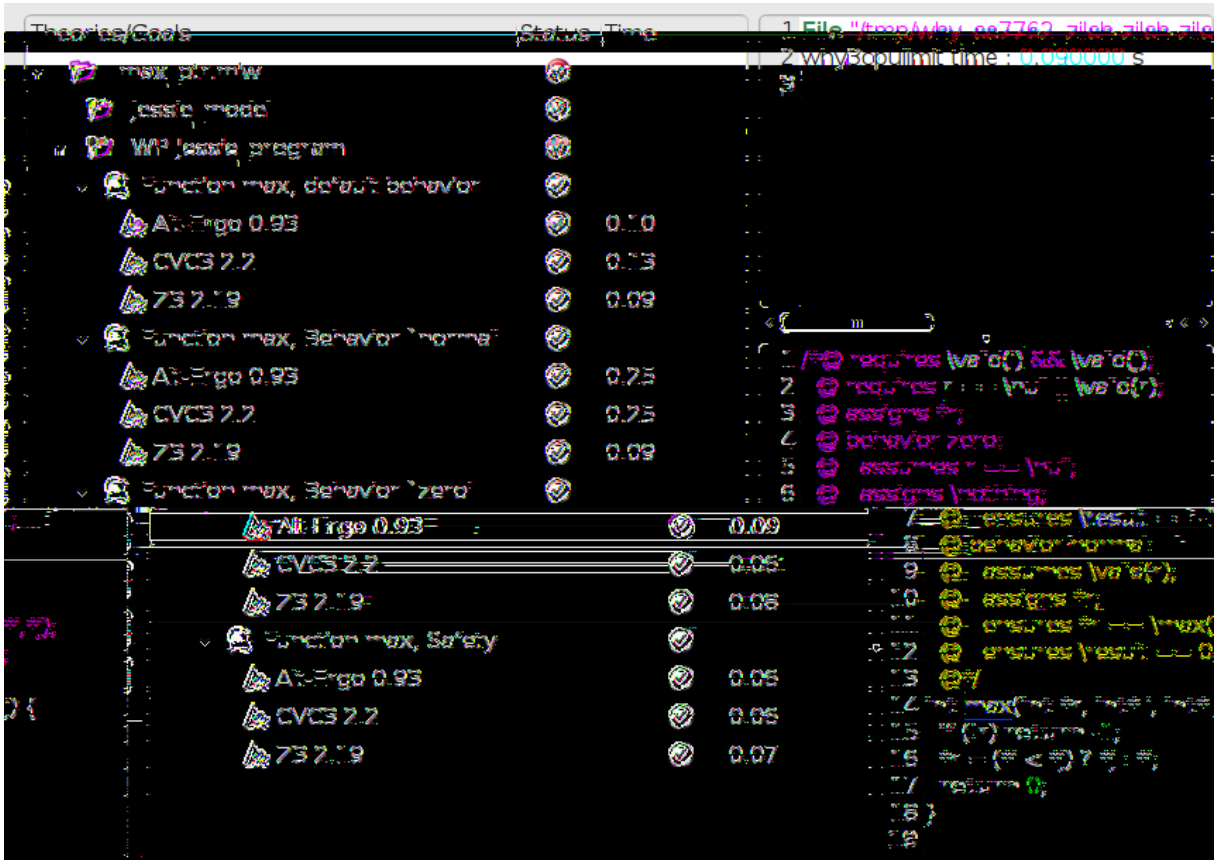


Figure 1.3: Why3 IDE for `max_ptr` function

```
  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 depends of the proprecessor 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 VCs: Safety, default behavior, normal behavior 'normal' and normal behavior 'zero' for the two user-defined behaviors.

In the next chapters, we detail how to prove each kind of VC. We also recommend to look at the collection of verified programs at URL http://toccata.lri.fr/gallery/jessieplugin.en.html to get more hints on how to specify and prove programs with Jessie.

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 `long`s. 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 plug-in to ignore termination issues.

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

```
> frama-c -jessie binary-search.c
```

Figure 2.1 shows the result after

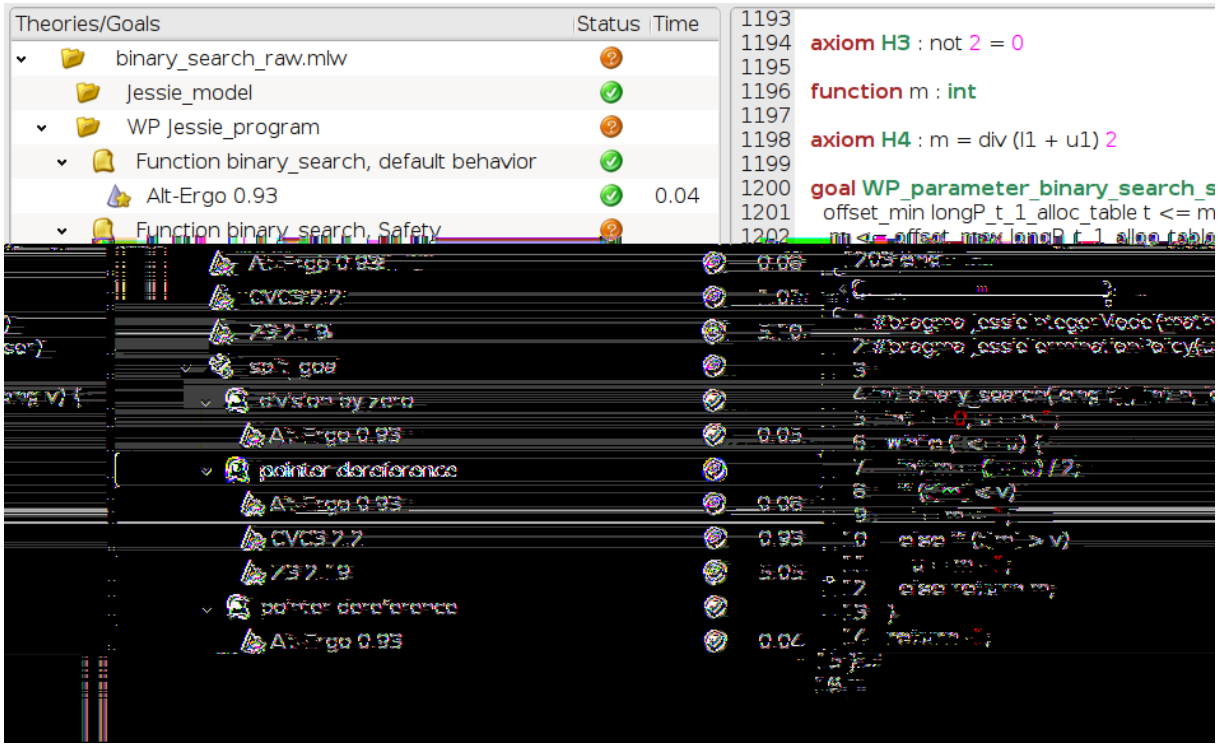


Figure 2.1: Memory safety with no annotations

- clicking on Alt-Ergo button to launch Alt-Ergo on both Safety and Normal behavior. The normal behavior is proved valid but not the safety.
- selecting the raw "safety" and clicking on the button "split" to split the VC in parts. 3 sub-goals are obtained: one that states the divisor `2` is not null, and two more that state the array access `t[m]` should be within bounds.
- clicking again on Alt-Ergo, then CVC3 then Z3

The remaing VC 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 where `n` is positive and array `t` is valid between indices `0` and `n-1` included. Since function `binary_search` accesses array `t` inside a loop, providing a precondition is not enough to make the generated VC provable. One must also provide a *loop invariant*. A loop invariant is a property that remains true at each iteration of the loop. It is often necessary for the user to provide these properties to help Jessie reason about loops. Assuming that the right property has been provided, Jessie is then left with the easier task of generating and verifying VCs that ensure that the property indeed holds at the beginning of each iteration.

In this example, it is necessary to provide an 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;
```

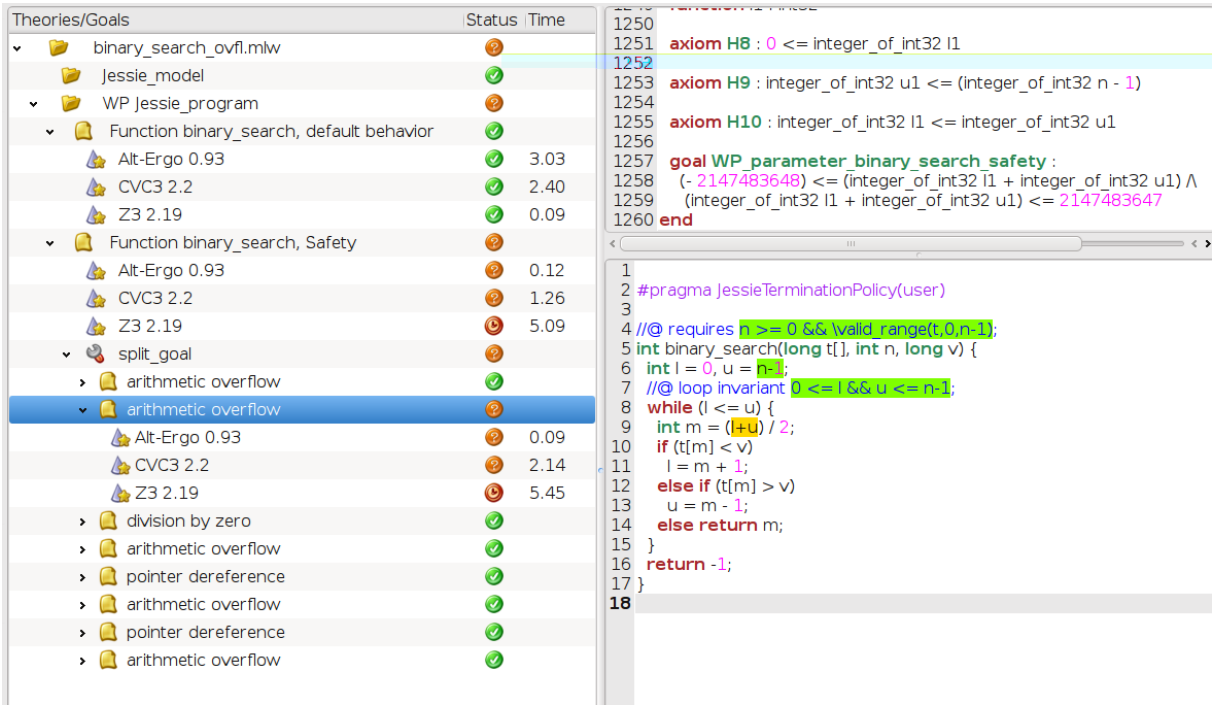


Figure 2.2: Memory safety + integer overflow safety

```
while (l <= u ) {
  ...
```

Launching again the IDE after adding these annotations, you should obtain that all VCs are now proved.

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

The result can be seen in Figure 2.2. There are more subgoals to Safety VC, to check that integer operations return a result within bounds, only one of which is not proved. With this exception, the results are nearly the same as with exact 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

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

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

This is our choice here. All VCs 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 that requires to prove the property $0 > 0$. This VC is false, so our first step should be to help Jessie generate a more provable VC. The VC $0 > 0$ is generated 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:

```
//@ 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−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;
}
```

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.

```
/*@ 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;
}
```

All VCs 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

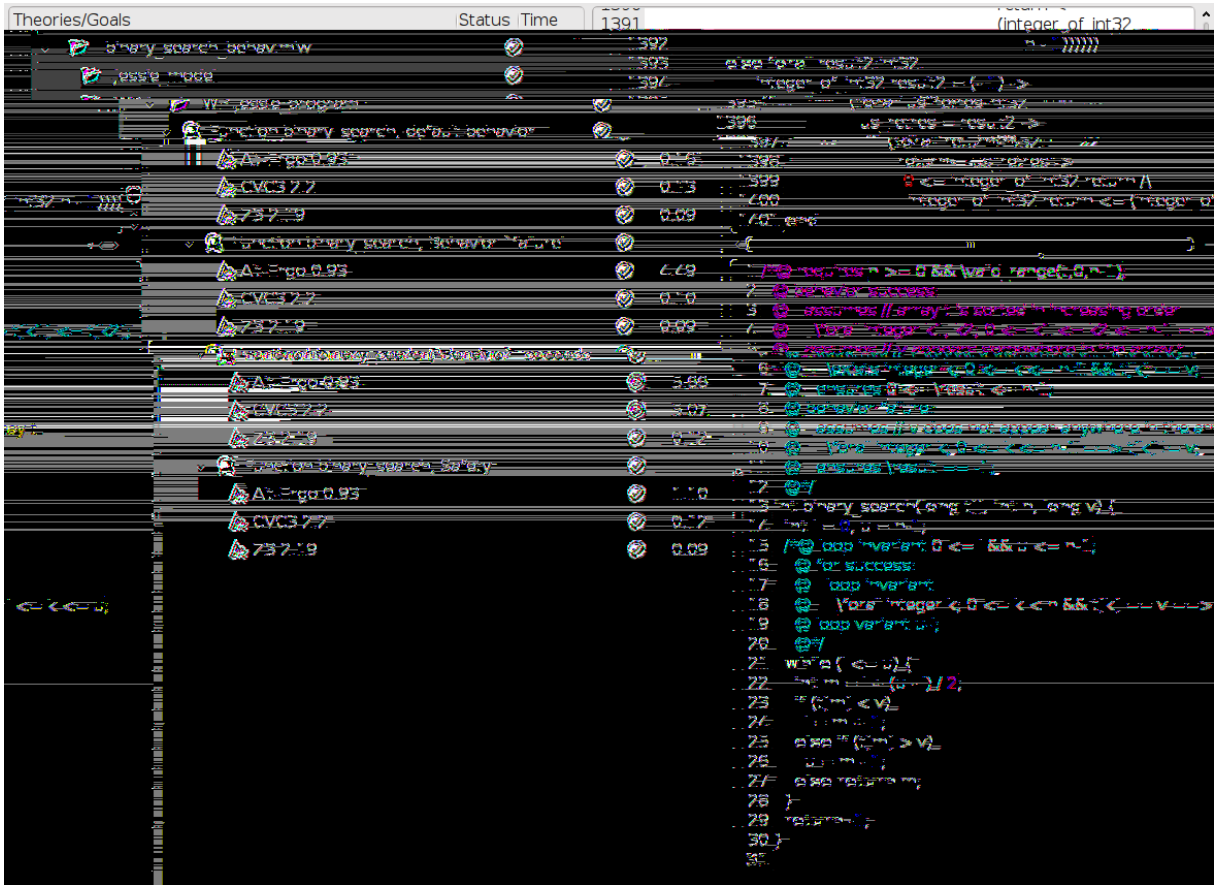


Figure 3.1: Postconditions in behaviors

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.

```
/*@ 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 3.1 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) &&
  @   \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,j; l <= i <= j < h ==> a[i] <= a[j] ;
```

```
  @*/
```

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);
  @ behavior permutation:
  @   ensures Permut{Old,Here}(t,0,n-1);
  @*/
void sel_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];
      }
    }
  L:
    swap(t,i,mi);
    //@ assert Permut{L,Here}(t,0,n-1);
  }
}
```

Each VC is proved by at least by Z3. The behavior “sorted” is the most diificult one. Indeed, if you split this one into subgolas, then all subgoals are proved by Alt-Ergo too.

Chapter 4

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 anymore. 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 VCs 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 VCs 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 5

Reference Manual

5.1 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 using Why3 VC generator and displayed in the Why3IDE interface. Using `-jessie-atp=gui` option will use the Why2 VC generator and display in the Why GUI interface, as it was the default in version 2.29 and before (deprecated).

The `-jessie-atp=<p>` option allows to run VCs in batch, using the given theorem prover `<p>`. It uses the Why2 VC generator only, and is thus deprecated. Running prover in batch mode after using the Why3 VC generator can be done using the Why3 tool `why3 replay`, see Why3 manual for details.

5.2 Unsupported features

5.2.1 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. In some cases, Frama-C's specialization plug-in can be used 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.

5.2.2 partially supported ACSL features

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

Axiomatic declarations
supported (experimental)

5.2.3 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

5.3 Command-line options

-jessie activates the plug-in, to perform C to Jessie translation

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

-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-why3=<command> alternative command used to call Why3 (default is "ide")

5.4 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 [2, 3].
- `full`: models the full IEEE-754 standard, including infinite values and NaNs. This model is the *full* model discussed in [2, 3].

- `multirounding`: models floating-point arithmetics so as to support combinations of compilerd and architectures that do not strictly follows IEEE-754 standard (e.g. double roundings, 80-bits extended formats, compilation usinf fused-multiply-add instructions). This is based on paper [5, 6].

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.

5.5 Troubleshooting

Here is a set of common error messages and possible fix or workaround.

unsupported "cannot handle this lvalue" this message may appear in the following situations:

- use an array as a parameter of a logic function. You should use a pointer instead.

unsupported "this kind of memory access is not currently supported" this message may appear in the following situations:

- equality on structures in the logic. The workaround is to check equality field by field. Tip: define field-by-field equality as a logic predicate.

unsupported "Jessie plugin does not support struct or union as parameter to logic functions." This is already quite explicit: jessie does not support structures or unions as a parameter of logic functions or predicates. You can circumvent this limitation by using an indirection via a pointer.

unsupported "cannot take address of a function" The jessie plugin does not support functions as parameters to other functions. There is no simple workaround. One thing you can try is to remove the function parameter and use a fixed abstract function (i.e. with a contract but no body), and then prove that all the functions that might be passed as parameters respect this contract.

unsupported "Type builtin_va_list not allowed" Jessie does not handle varyadic functions. The same trick as above could be attempted.

unsupported "Casting from type <..> to type <..> not allowed" Jessie does not support this cast, typically between pointer and integer. There is no simple workaround. One way of proving such kind of code is to replace the casts by an abstract function, whose post-condition explicitly explains how the conversion is made.

failure: cannot interpreted this lvalue This may happen if

- using a structure in an assigns clause. You need to expand ans say which field are assigned.

Bibliography

[1] The Why verification tool. http://why.lri.fr/.

[2] Ali Ayad and Claude Marché. Behavioral properties of floating-point programs. Hisseo publications, 2009. http://hisseo.saclay.inria.fr/ayad09.pdf.

[3] 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.

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[5] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. Hisseo publications, 2009. http://hisseo.saclay.inria.fr/tuyen09.pdf.

[6] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Mu noz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.

[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.