Specification and Proof of Programs with Frama-C
TAP 2013 Tutorial

Nikolai Kosmatov, Virgile Prevosto, Julien Signoles
firstname.lastname@cea.fr

CEA LIST

June 18, 2013
Motivation

Main objective:

Rigorous, mathematical proof of semantic properties of a program

- functional properties
- safety:
  - all memory accesses are valid,
  - no arithmetic overflow,
  - no division by zero, . . .
- termination
- . . .
Our goal

In this tutorial, we will see

- how to specify a C program
- how to prove it with an automatic tool
- how to understand and fix proof failures
Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
Introduction

Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
A brief history

- 90’s: CAVEAT, a Hoare logic-based tool for C programs
- 2000’s: CAVEAT used by Airbus during certification of the A380
- 2002: Why tool and its C front-end Caduceus
- 2006: Joint project to write a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2009: Hoare-logic based Frama-C plugin Jessie developed at INRIA
- 2012: New Hoare-logic based plugin WP developed at CEA LIST
- Frama-C today:
  - Most recent release: Frama-C Fluorine (v9)
  - Multiple projects around the platform
  - A growing community of users and plugin developers
Frama-C at a glance

- **FRAmework for Modular Analysis of C programs**
  - Various plugins: CFG, value analysis (abstract interpretation), impact analysis, dependency analysis, slicing, program proof, . . .

- Developed at CEA LIST and INRIA Saclay (Proval/Toccata team)

- Released under LGPL license

- Kernel based on CIL library [Necula et al. – Berkeley]

- Includes ACSL specification language

- Extensible platform
  - Adding specialized plugins is easy
  - Collaboration of analyses over the same code
  - Inter-plugin communication through ACSL formulas

Main Frama-C plugins

- Deductive Verification
- Specification Generation
- Dynamic Analysis
- Executable-ACSL
- PathCrawler
- SANTE
- Mthread
- Agen
- Aoraï
- WP
- Jessie
- Abstract Interpretation
- Value Analysis
- Code Transformation
- Semantic constant folding
- Formal Methods
- Spare code
- Slicing
- Browsing of unfamiliar code
- Metrics computation
- Variable occurrences
- Impact Analysis
- Scope & Data-flow browsing

included in main distribution
distributed externally
ACSL: ANSI/ISO C Specification Language

Presentation

- Based on the notion of contract, like in Eiffel
- Allows the users to specify functional properties of their programs
- Allows communication between various plugins
- Independent from a particular analysis
- ACSL manual at http://frama-c.com/acsl

Basic Components

- First-order logic
- Pure C expressions
- C types + \( \mathbb{Z} \) (integer) and \( \mathbb{R} \) (real)
- Built-in predicates and logic functions, particularly over pointers:
  \( \text{valid}(p) \), \( \text{valid}(p+0..2) \), \( \text{separated}(p+0..2,q+0..5) \), \( \text{block_length}(p) \)
Jessie plugin

- Hoare-logic based plugin, developed at INRIA Saclay
- Proof of functional properties of the program
- Modular verification (function per function)
- Input: a program and a specification in ACSL
- Jessie generates verification conditions (VCs)
- Use of Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Simplify, Z3, Yices, CVC3, ...
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?
- Limitations
  - Casts between pointers and integers
  - Limited support for union type
  - Aliasing requires some care
Jessie plugin

- Hoare-logic based plugin, developed at INRIA Saclay
- Proof of functional properties of the program
- Modular verification (function per function)
- Input: a program and a specification in ACSL
- Jessie generates verification conditions (VCs)
- Use of Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Simplify, Z3, Yices, CVC3, ... 
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?
  - If the specification is wrong, the program can be wrong
- Limitations
  - Casts between pointers and integers
  - Limited support for union type
  - Aliasing requires some care
In this tutorial

In this tutorial we use

- Frama-C Carbon
- Jessie and Why 2.29
- Alt-Ergo 0.93

To run Jessie on a C program file.c

- frama-c -jessie file.c

All examples were also tested with

- Frama-C Nitrogen
- Jessie and Why 2.31
- Why3 0.73
- Alt-Ergo 0.95
Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
Contracts

- **Goal:** specification of imperative functions
- **Approach:** give assertions (i.e. properties) about the functions
  - **Precondition** is supposed to be true on entry (ensured by callers of the function)
  - **Postcondition** must be true on exit (ensured by the function if it terminates)
- Nothing is guaranteed when the precondition is not satisfied
- **Termination** may or may not be guaranteed (total or partial correctness)

Role of contracts

- Main input of the verification process
- Must reflect the informal specification
- Should not be modified just to suit the verification tasks
Example 1

Specify and prove the following program:

```c
// returns the absolute value of x
int abs ( int x ) {
    if ( x >=0 )
        return x ;
    return -x ;
}
```
Example 1 (Continued) - Explain the proof failure

Explain the proof failure for the following program:

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

For $x = \text{INT\_MIN}$, $-x$ cannot be represented by an int and overflows.
Example 1 (Continued) - Explain the proof failure

Explain the proof failure for the following program:

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

- For \( x=\text{INT\_MIN} \), \(-x\) cannot be represented by an \text{int} and overflows
- Example: on 32-bit, \( \text{INT\_MIN} = -2^{31} \) while \( \text{INT\_MAX} = 2^{31} - 1 \)
Safety warnings: arithmetic overflows

Absence of arithmetic overflows can be important to check

- A sad example: crash of Ariane 5 in 1996
- Jessie automatically generates VCs to check absence of overflows
- They ensure that arithmetic operations do not overflow
- If not proved, an overflow may occur. Is it intended?
Example 1 (Continued) - Solution

This is the completely specified program:

```csharp
#include <limits.h>
/*@ requires x > INT_MIN;
   ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
   assigns \nothing;
*/
int abs ( int x ) {
   if ( x >=0 )
      return x ;
   return -x ;
}
```
Example 2

Specify and prove the following program:

```c
// returns the maximum of x and y
int max ( int x, int y ) {
    if ( x >=y )
        return x ;
    return y ;
}
```
Example 2 (Continued) - Find the error

The following program is proved. Do you see any error?

/*@ ensures \result >= x && \result >= y; */
int max ( int x, int y ) {
    if ( x >=y )
        return x ;
    return y ;
}
Example 2 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```c
#include <limits.h>
/*@ ensures \result >= x && \result >= y; */
int max ( int x, int y ) {
    return INT_MAX ;
}
```

Our specification is incomplete. Should say that the returned value is one of the arguments.
Example 2 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```c
#include <limits.h>
/*@ ensures \result >= x && \result >= y;
*/
int max ( int x, int y ) {
    return INT_MAX ;
}
```

- Our specification is incomplete
- Should say that the returned value is one of the arguments
Example 2 (Continued) - Solution

This is the completely specified program:

```c
/*@ ensures \result >= x && \result >= y; 
ensures \result == x || \result == y; 
assigns \nothing; 
*/
int max ( int x, int y ) {
    if ( x >=y )
        return x ;
    return y ;
}
```
Example 3

Specify and prove the following program:

```c
// returns the maximum of *p and *q
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```
Example 3 (Continued) - Explain the proof failure

Explain the proof failure for the following program:

```c
/*@ ensures \result >= *p && \result >= *q;
ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```
Example 3 (Continued) - Explain the proof failure

Explain the proof failure for the following program:

```c
/*@ ensures \result >= *p && \result >= *q;
   ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```

- Nothing ensures that pointers \( p, q \) are valid
- It must be ensured either by the function, or by its precondition
Safety warnings: invalid memory accesses

An invalid pointer or array access may result in a segmentation fault or memory corruption.

- Jessie automatically generates VCs to check memory access validity
- They ensure that each pointer (array) access has a valid offset (index)
- If the function assumes that an input pointer is valid, it must be stated in its precondition, e.g.
  - \( \text{\texttt{valid}}(p) \) for one pointer \( p \)
  - \( \text{\texttt{valid}}(p+0..2) \) for a range of offsets \( p, p+1, p+2 \)
Example 3 (Continued) - Find the error

The following program is proved. Do you see any error?

/*@ requires \valid(p) && \valid(q);
ensures \result >= *p && \result >= *q;
ensures \result == *p || \result == *q;
*/

int max_ptr ( int *p, int *q ) {
  if ( *p >= *q )
    return *p ;
  return *q ;
}
Example 3 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

```c
/*@ requires \valid(p) && \valid(q);
   ensures \result >= *p && \result >= *q;
   ensures \result == *p || \result == *q;
*/
int max_ptr ( int *p, int *q ) {
   *p = 0;
   *q = 0;
   return 0;
}
```

Our specification is incomplete. Should say that the function cannot modify \*p and \*q.
Example 3 (Continued) - a wrong version

This is a wrong implementation that is also proved. Why?

/*@ requires \valid(p) && \valid(q);
    ensures \result >= *p && \result >= *q;
    ensures \result == *p || \result == *q;
*/

int max_ptr ( int *p, int *q ) {
    *p = 0;
    *q = 0;
    return 0 ;
}

▶ Our specification is incomplete
▶ Should say that the function cannot modify *p and *q
Frame rule

The clause `assigns v1, v2, ..., vN;`

- **Part of the postcondition**
- Specifies which (non local) variables can be modified by the function
- No need to specify local variable modifications in the postcondition
  - a function is allowed to change local variables
  - a postcondition cannot talk about them anyway, they do not exist after the function call
- Avoids to state that for any unchanged global variable `v`, we have `ensures \text{old}(v) == v`
- Avoids to forget one of them: explicit permission is required
- If nothing can be modified, specify `assigns \nothing`
This is the completely specified program:

```c
/*@ requires \valid(p) && \valid(q);
   ensures \result >= *p && \result >= *q;
   ensures \result == *p || \result == *q;
   assigns \nothing;
*/

int max_ptr ( int *p, int *q ) {
    if ( *p >= *q )
        return *p ;
    return *q ;
}
```
Behaviors

Specification by cases

- Global precondition (requires) applies to all cases
- Global postcondition (ensures, assigns) applies to all cases
- Behaviors define contracts (refine global contract) in particular cases
- For each case (each behavior)
  - the subdomain is defined by assumes clause
  - the behavior’s precondition is defined by requires clauses
    - it is supposed to be true whenever assumes condition is true
  - the behavior’s postcondition is defined by ensures, assigns clauses
    - it must be ensured whenever assumes condition is true
- complete behaviors states that given behaviors cover all cases
- disjoint behaviors states that given behaviors do not overlap
Example 4

Specify using behaviors and prove the function abs:

```c
// returns the absolute value of x
int abs ( int x ) {
    if ( x >=0 )
        return x ;
    return -x ;
}
```
Example 4 (Continued) - Explain the proof failure for

```c
#include <limits.h>
/*@ requires x > INT_MIN;
assigns \nothing;
behavior pos:
  assumes x > 0;
  ensures \result == x;
behavior neg:
  assumes x < 0;
  ensures \result == -x;
complete behaviors;
disjoint behaviors;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

The behaviors are not complete
The case `x==0` is missing. A wrong value could be returned.
Example 4 (Continued) - Explain the proof failure for

```c
#include <limits.h>
/*@ requires x > INT_MIN; 
  assigns \nothing;
behavior pos:
  assumes x > 0;
  ensures \result == x;
behavior neg:
  assumes x < 0;
  ensures \result == -x;
complete behaviors;
disjoint behaviors;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

- The behaviors are not complete
- The case $x=0$ is missing. A wrong value could be returned.
Example 4 (Continued) - Explain another proof failure for

```c
#include <limits.h>
/*@ requires x > INT_MIN;
 assigns \nothing;
 behavior pos:
   assumes x >= 0;
   ensures \result == x;
 behavior neg:
   assumes x <= 0;
   ensures \result == -x;
 complete behaviors;
 disjoint behaviors;
*/
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

▶ The behaviors are not disjoint
▶ The case x==0 is covered by both behaviors. Is it intended?
Example 4 (Continued) - Explain another proof failure for

```c
#include <limits.h>
/*@ requires x > INT_MIN;
 assigns nothing;
 behavior pos:
    assumes x >= 0;
    ensures \result == x;
 behavior neg:
    assumes x <= 0;
    ensures \result == -x;
 complete behaviors;
 disjoint behaviors;
*/
int abs ( int x ) {
    if ( x >= 0 )
        return x ;
    return -x ;
}
```

- The behaviors are not disjoint
- The case x==0 is covered by both behaviors. Is it intended?
# include <limits.h>

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

behavior pos:
  assumes x >= 0;
  ensures \result == x;

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

complete behaviors;

disjoint behaviors;

*/

int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
Function contracts are handled as follows:

- Suppose function $g$ contains a call to a function $f$
- Suppose we try to prove the caller $g$
- Before the call to $f$ in $g$, the precondition of $f$ must be ensured by $g$
  - VC is generated to prove that the precondition of $f$ is respected
- After the call to $f$ in $g$, the postcondition of $f$ is supposed to be true
  - the postcondition of $f$ is assumed in the proof below
  - modular verification: the code of $f$ is not checked at this point
  - only a contract and a declaration of the callee $f$ are required

Pre/post of the caller and of the callee have dual roles in the caller’s proof

- Pre of the caller is supposed, Post of the caller must be ensured
- Pre of the callee must be ensured, Post of the callee is supposed
Example 5

Specify and prove the function `max_abs`

```c
int abs ( int x );
int max ( int x, int y );

// returns maximum of absolute values of x and y
int max_abs ( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}
```
Example 5 (Continued) - Explain the proof failure for

```c
#include <limits.h>
/
/* @ requires x > INT_MIN;
   ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
   assigns \nothing; */
int abs ( int x );
/
/*@ ensures \result >= x && \result >= y;
   ensures \result == x || \result == y;
   assigns \nothing; */
int max ( int x, int y );
/
/*@ ensures \result >= x && \result >= -x &&
          \result >= y && \result >= -y;
   ensures \result == x || \result == -x ||
          \result == y || \result == -y;
   assigns \nothing; */
int max_abs( int x, int y ) {
    x=abs(x);
    y=abs(y);
    return max(x,y);
}
```
Example 5 (Continued) - Explain the proof failure for

```c
#include <limits.h>
/*@ requires x > INT_MIN;
  ensures (x >= 0 ==> \result == x) &&
            (x < 0 ==> \result == -x);
  assigns \nothing; */
int abs ( int x );

/*@ ensures \result >= x && \result >= y;
  assigns \nothing; */
int max ( int x, int y );

/*@ requires x > INT_MIN;
  requires y > INT_MIN;
  ensures \result >= x && \result >= -x &&
                 \result >= y && \result >= -y;
  ensures \result == x || \result == -x ||
                 \result == y || \result == -y;
  assigns \nothing; */
int max_abs( int x, int y ) {
  x=abs(x);
  y=abs(y);
  return max(x,y);
}
```
Example 5 (Continued) - Solution

```c
#include <limits.h>

/*@ requires x > INT_MIN;
 ensures (x >= 0 ==> result == x) &&
        (x < 0 ==> result == -x);
 ensures result >= x && result >= y;
 ensures result == x || result == y;
 assigns nothing; */
int abs ( int x ) ;

/*@ ensures result >= x && result >= y;
 ensures result == x || result == y;
 assigns nothing; */
int max ( int x , int y ) ;

/*@ requires x > INT_MIN;
 requires y > INT_MIN;
 ensures result >= x && result >= -x &&
        result >= y && result >= -y;
 ensures result == x || result == -x ||
        result == y || result == -y;
 assigns nothing; */
int max_abs( int x , int y ) {
 x=abs(x);
 y=abs(y);
 return max(x , y);
}
```
Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
Loops and automatic proof

- What is the issue with loops? Unknown, variable number of iterations
- The only possible way to handle loops: proof by induction
- Induction needs a suitable inductive property, that is proved to be
  - satisfied just before the loop, and
  - satisfied after $k + 1$ iterations whenever it is satisfied after $k \geq 0$ iterations
- Such inductive property is called loop invariant
- The verification conditions for a loop invariant include two parts
  - loop invariant initially holds
  - loop invariant is preserved by any iteration
Loop invariants - some hints

How to find a suitable loop invariant? Consider two aspects:

- **identify variables modified in the loop**
  - variable number of iterations prevents from deducing their values (relationships with other variables)
  - define their possible value intervals (relationships) after \( k \) iterations
  - use `loop assigns` clause to list variables that (might) have been assigned so far after \( k \) iterations

- **identify realized actions, or properties already ensured by the loop**
  - what part of the job already realized after \( k \) iterations?
  - what part of the expected loop results already ensured after \( k \) iterations?
  - why the next iteration can proceed as it does? ...

A stronger property on each iteration may be required to prove the final result of the loop

Some experience may be necessary to find appropriate loop invariants
Loop invariants - more hints

Remember: a loop invariant must be true
- before (the first iteration of) the loop, even if no iteration is possible
- after any complete iteration even if no more iterations are possible
- in other words, any time right before the loop condition check

In particular, a **for** loop

```c
for ( i = 0; i < n; i ++ ) { /* body */ }
```

should be seen as

```c
i = 0; // action before the first iteration
while ( i < n ) // an iteration starts by the condition check
{
    /* body */
    i ++; // last action in an iteration
}
```
Example 6

Specify and prove the function `find_min`:

```c
// returns the index of the minimal element
// of the given array `a` of size `length`
int find_min(int* a, int length) {
    int min, min_idx;
    min_idx = 0;
    min = a[0];
    for (int i = 1; i < length; i++) {
        if (a[i] < min) {
            min_idx = i;
            min = a[i];
        }
    }
    return min_idx;
}
```
Loop termination

- Program termination is undecidable
- A tool cannot deduce neither the exact number of iterations, nor even an upper bound
- If an upper bound is given, a tool can check it by induction
- An upper bound on the number of remaining loop iterations is the key idea behind the loop variant

Terminology

- Partial correctness: if the function terminates, it respects its specification
- Total correctness: the function terminates, and it respects its specification
Loop variants - some hints

- Unlike an invariant, a loop variant is an integer expression, not a predicate
- Loop variant is not unique: if $V$ works, $V + 1$ works as well
- No need to find a precise bound, any working loop variant is OK
- To find a variant, look at the loop condition
  - For the loop `while(exp1 > exp2 )`, try `loop variant exp1-exp2`;
- In more complex cases: ask yourself why the loop terminates, and try to give an integer upper bound on the number of remaining loop iterations
Example 6 (Continued) - Solution

```c
/*@ requires length > 0 && valid(a+(0..length-1));
   assigns nothing;
   ensures 0<=\result<length && 
           ( \forall integer j; 0<=j<length \implies a[\result]<=a[j] ); */
int find_min(int* a, int length) {
    int min, min_idx;
    min_idx = 0;
    min = a[0];
   /*@ loop invariant 0<=i<=length && 0<=min_idx<length;
   loop invariant \forall integer j; 0<=j<i \implies min<=a[j];
   loop invariant a[min_idx]==min;
   loop assigns min, min_idx, i;
   loop variant length - i; */
    for (int i = 1; i<length; i++) {
        if (a[i] < min) {
            min_idx = i;
            min = a[i];
        }
    }
    return min_idx;
}
```
Example 7

Specify and prove the function all_zeros:

```c
// returns a non-zero value iff all elements
// in a given array t of n integers are zeros
int all_zeros(int t[], int n) {
    int k;
    for(k = 0; k < n; k++)
        if (t[k] != 0)
            return 0;
    return 1;
}
```
Example 7 (Continued) - Find the errors

/*@ requires n\geq 0 && \valid(t+(0..n-1));
  ensures \result \neq 0 \\
  (\forall \text{integer } j; 0 \leq j < n \implies t[j] = 0); */
int all_zeros(int t[], int n) {
  int k;
 /*@ loop invariant 0 \leq k < n; */
  loop variant n-k;
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
Example 7 (Continued) - Solution

/*@ requires n>=0 && !valid(t+(0..n-1));
  assigns nothing;
  ensures result != 0 <==>
    (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
 /*@ loop invariant 0 <= k <= n;
  loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
  loop variant n-k;
*/
  for (k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}

N.Kosmatov, V.Prevosto, J.Signoles (CEA LIST)  Proof of Programs with Frama-C  June 18, 2013  49 / 66
\textbf{forall and \ exists - hints and examples}

- Do not confuse $\&\&$ and $\implies$ inside \texttt{forall} and \texttt{exists}

- Some common patterns:
  - \texttt{forall integer j; 0 <= j $\&\&$ j < n $\implies$ t[j] == 0;}
  - \texttt{exists integer j; 0 <= j $\&\&$ j < n $\&\&$ t[j] != 0;}
  - Each one here is negation of the other

- A shorter form:
  - \texttt{forall integer j; 0 <= j < n $\implies$ t[j] == 0;}
  - \texttt{exists integer j; 0 <= j < n $\&\&$ t[j] != 0;}

- With several variables:
  - \texttt{forall integer i,j; 0 <= i <= j < length $\implies$ a[i] <= a[j];}
  - \texttt{exists integer i,j; 0 <= i <= j < length $\&\&$ a[i] > a[j]
Example 8

Specify and prove the function binary_search:

```c
/* takes as input a sorted array a, its length, and a value key to search, returns the index of a cell which contains key, returns -1 iff key is not present in the array */

int binary_search(int * a, int length, int key) {
    int low = 0, high = length - 1;
    while (low <= high) {
        int mid = (low + high)/2;
        if (a[mid] == key) return mid;
        if (a[mid] < key) { low = mid + 1; }
        else { high = mid - 1; }
    }
    return -1;
}
```
Example 8 (Continued) - Solution (1/2)

/*@
predicate sorted{L}(int* a, int length) = 
  \forall integer i,j; 0\leq i\leq j < length \implies a[i] \leq a[j];
*/
/*@
requires \valid(a+(0..length -1));
requires sorted(a, length);
requires length \geq 0;

assigns \nothing;

behavior exists:
  assumes \exists integer i; 0\leq i < length \&\& a[i] == key;
  ensures 0\leq result < length \&\& a[result] == key;

behavior not_exists:
  assumes \forall integer i; 0\leq i < length \implies a[i] != key;
  ensures result == -1;

complete behaviors;
disjoint behaviors;
*/
Example 8 (Continued) - Solution (2/2)

```c
int binary_search(int* a, int length, int key) {
    int low = 0, high = length - 1;
    /*@ loop invariant 0<=low<=high+1;
    loop invariant high<length;
    loop assigns low,high;
    loop invariant forall integer k; 0<=k<low ==> a[k] < key;
    loop invariant forall integer k; high<k<length ==> a[k] > key;
    loop variant high-low; */
    while (low<=high) {
        int mid = low+(high-low)/2;
        if (a[mid] == key) return mid;
        if (a[mid] < key) { low = mid+1; }
        else { high = mid - 1; }
    }
    return -1;
}
```
Example 9

Specify and prove the function `sort`:

```c
// sorts given array `a` of size `length` > 0
void sort (int* a, int length) {
    int current;
    for (current = 0; current < length - 1; current++) {
        int min_idx = current;
        int min = a[current];
        for (int i = current + 1; i < length; i++) {
            if (a[i] < min) {
                min = a[i];
                min_idx = i;
            }
        }
        if (min_idx != current) {
            L: a[min_idx] = a[current];
            a[current] = min;
        }
    }
}
```
Referring to another state

- Specification may require *values at different program points*
- Use $\texttt{\textbackslash at(e,L)}$ to refer to the value of expression $e$ at label $L$
- Some predefined labels:
  - $\texttt{\textbackslash at(e,Here)}$ refers to the current state
  - $\texttt{\textbackslash at(e,Old)}$ refers to the pre-state
  - $\texttt{\textbackslash at(e,Post)}$ refers to the post-state
- $\texttt{\textbackslash old(e)}$ is equivalent to $\texttt{\textbackslash at(e,Old)}$
Example 9 (Continued) - Solution (1/3)

/*@ predicate sorted{L}(int* a, integer length) =
     \forall integer i,j; 0<=i<=j<length \implies a[i]<=a[j];*/

/*@ predicate swap{L1,L2}(int* a, integer i, integer j, integer length)=
     0<i<j<length
     && \at(a[i],L1) == \at(a[j],L2)
     && \at(a[i],L2) == \at(a[j],L1)
     && \forall integer k; 0<=k<length && k!=i && k!=j \implies
                     \at(a[k],L1) == \at(a[k],L2);*/

/*@ inductive same_elements{L1,L2}(int*a, integer length) {
     case refl{L}:
         \forall int*a, integer length; same_elements{L,L}(a,length);
     case swap{L1,L2}:
         \forall int*a, integer i,j,length;
         swap{L1,L2}(a,i,j,length) \implies same_elements{L1,L2}(a,length);
     case trans{L1,L2,L3}:
         \forall int*a, integer length;
         same_elements{L1,L2}(a,length)
         \implies same_elements{L2,L3}(a,length)
         \implies same_elements{L1,L3}(a,length);
}
*/
Example 9 (Continued) - Solution (2/3)

/*@ requires valid(a+(0..length−1));
 requires length > 0;
 assigns a[0..length−1];
 behavior sorted:
  ensures sorted(a,length);
 behavior same_elements:
  ensures same_elements{Pre,Here}(a,length);
*/

void sort (int* a, int length) {
  int current;
  /*@ loop invariant 0<=current<=length;
  loop assigns a[0..length−1],current;
  for sorted: loop invariant sorted(a,current);
  for sorted: loop invariant
    forall integer i,j; 0<=i<=current<=j<=length ==> a[i] <= a[j];
  for same_elements: loop invariant
    same_elements{Pre,Here}(a,length);
  loop variant length−current;
  */
Example 9 (Continued) - Solution (3/3)

```c
for (current = 0; current < length - 1; current++) {
    int min_idx = current;
    int min = a[current];
    /*@ loop invariant current+1<=i<=length; */
    loop assigns i, min, min_idx;
    loop invariant current<=min_idx<i;
    loop invariant a[min_idx] == min;
    for sorted: loop invariant
        \forall all integer j; current<=j<i ==> min <= a[j];
    loop variant length -i;
    */
    for (int i = current + 1; i < length; i++) {
        if (a[i] < min) {
            min = a[i];
            min_idx = i;
        }
    }
    if (min_idx != current) {
        L: a[min_idx] = a[current];
        a[current] = min;
        /*@ for same_elements: assert swap{L, Here}{a, current, min_idx, length}; */
    }
}
```
Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
Proof failures

A proof of a VC for some annotation can fail for various reasons:

- incorrect implementation  
  \(\rightarrow\) check your code
- incorrect annotation  
  \(\rightarrow\) check your spec
- missing or erroneous (previous) annotation  
  \(\rightarrow\) check your spec
- insufficient timeout  
  \(\rightarrow\) try longer timeout
- complex property that automatic provers cannot handle.
Analysis of proof failures

When a proof failure is due to the specification, the erroneous annotation may be not obvious to find. For example:

- proof of a “loop invariant preserved” may fail in case of
  - incorrect loop invariant
  - incorrect loop invariant in a previous, or inner, or outer loop
  - missing assumes or loop assumes clause
  - too weak precondition
  - ...

- proof of a postcondition may fail in case of
  - incorrect loop invariant (too weak, too strong, or inappropriate)
  - missing assumes or loop assumes clause
  - inappropriate postcondition in a called function
  - too weak precondition
  - ...

My proof fails... What to do? Proof failures
Analysis of proof failures (Continued)

- Additional statements (assert, lemma, ...) may help the prover
  - They can be provable by the same (or another) prover or checked elsewhere
- Separating independent properties (e.g. in separate, non disjoint behaviors) may help
  - The prover may get lost with a bigger set of hypotheses (some of which are irrelevant)
- Use other Frama-C analyzers...

When nothing else helps to finish the proof:
- an interactive proof assistant can be used
- Coq, Isabelle, PVS, are not that scary: we may need only a small portion of the underlying theory
Combine different Frama-C analyzers

- **Value analysis (Value plugin)** [Cuoq et al, SEFM 2013]
  - based on abstract interpretation, computes possible values of variables
  - may prove some annotations

- **Runtime assertion checking (E-ACSL plugin)** [Delahaye et al, SAC’13]
  - treats E-ACSL, a large subset of ACSL: executable specification
  - detects erroneous annotations at runtime

- **Testing (PathCrawler plugin)** [Botella et al, AST 2009]
  - ensures rigorous path coverage of the program (DSE testing tool)
  - combined with E-ACSL, detects errors in annotations
  - in some cases, may prove that the annotation is verified

- **Program slicing (Slicing plugin)** [Chebaro et al, ASEJ 2013]
  - simplify your code preserving desired behaviors
  - use other analyzers (e.g. testing) on the simplified program

- ...
Combination with E-ACSL and PathCrawler (unpublished)

- (1) Initial C program specified in ACSL
- (2) Translation of the specification into C using E-ACSL plugin
- (3) Test case generated by PathCrawler violating the annotation
- (4) Annotation status reported in the Frama-C GUI as false

```c
int x2 (int i)
{ int k = 2 * i ;
  /*@ assert k > 0 ; */
  return k ; }

int x2 (int i)
{ int k = 2 * i ;
  e_acsl_assert(k > 0) ;
  return k ; }

void main()
{ int i = -35 ;
  x2 (i) ; }
```
Conclusion

Outline

Introduction
  Frama-C tool
  ACSL specification language
  Jessie plugin

Function contracts
  Pre- and postconditions
  Specification with behaviors
  Contracts and function calls

Programs with loops
  Loop invariants
  Loop termination
  More exercises

My proof fails... What to do?
  Proof failures
  Combination of analyses

Conclusion
We learned how to specify and prove a C program with Frama-C. Hoare-logic based tools provide a powerful way to formally verify programs. The program is proved with respect to the given specification, so absence of proof failures is not sufficient. The specification must be correct. The proof is automatic, but analysis of proof failures is manual. Proof failures help to complete the specification or find bugs. Interactive proof tools may be necessary to finish the proof for complex properties that cannot be proved automatically.