Towards Secure Things, or How to Verify IoT Software with Frama-C

Tutorial at ZINC 2018

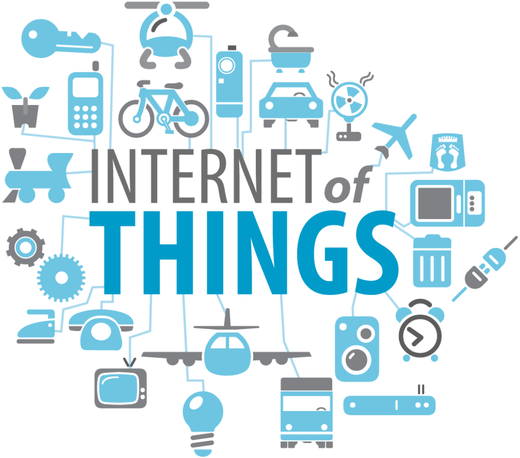
Allan Blanchard, Nikolai Kosmatov, Frédéric Loulergue
some slides authored by Julien Signoles

Email: allan.blanchard@inria.fr, nikolai.kosmatov@cea.fr, frederic.loulergue@nau.edu

Novi Sad, May 30th, 2018

Outline

Internet of Things



(c) Internet Security Buzz

- connect all devices and services
- 46 billions devices by 2021
- transport huge amounts of data

And Security?

And Security?

By Waqas on October 22, 2016 Email @hackread CYBER ATTACKS MALWARE SECURITY

And Security?

By Waqas on October 22, 2016 Email @hackread CYBER ATTACKS MALWARE SECURITY

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

And Security?

By Waqas on October 22, 2016 Email @hackread CYBER ATTACKS MALWARE SECURITY

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

by Tom Spring August 26, 2016 , 2:55 pm

And Security?

By Waqas on October 22, 2016 Email @hackread CYBER ATTACKS MALWARE SECURITY

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

by Tom Spring August 26, 2016 , 2:55 pm

Hacking a computer-aided sniper rifle

Elizabeth Weise | USATODAY
Published 5:56 p.m. UTC Aug 7, 2015

Outline

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)

Frama-C Historical Context

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2004: start of Frama-C project as a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- 2012: WP: Weakest-precondition based plugin
- 2012: E-ACSL: Runtime Verification plugin
- 2013: CEA Spin-off TrustInSoft
- 2016: Eva: Evolved Value Analysis
- 2016: Frama-Clang: C++ extension
- Today: Frama-C Sulfur (v.16)
- Upcoming: Frama-C Chlorine (v.17, expected in June)

Frama-C Open-Source Distribution

Framework for Analysis of source code written in ISO 99 C

[Kirchner et al, FAC'15]

- analysis of C code extended with ACSL annotations
- ACSL Specification Language
  - *langua franca* of Frama-C analyzers
- mostly open-source (LGPL 2.1)

http://frama-c.com

- also proprietary extensions and distributions
- targets both academic and industrial usage

Example: a C Program Annotated in ACSL

```
/*@ 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 assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

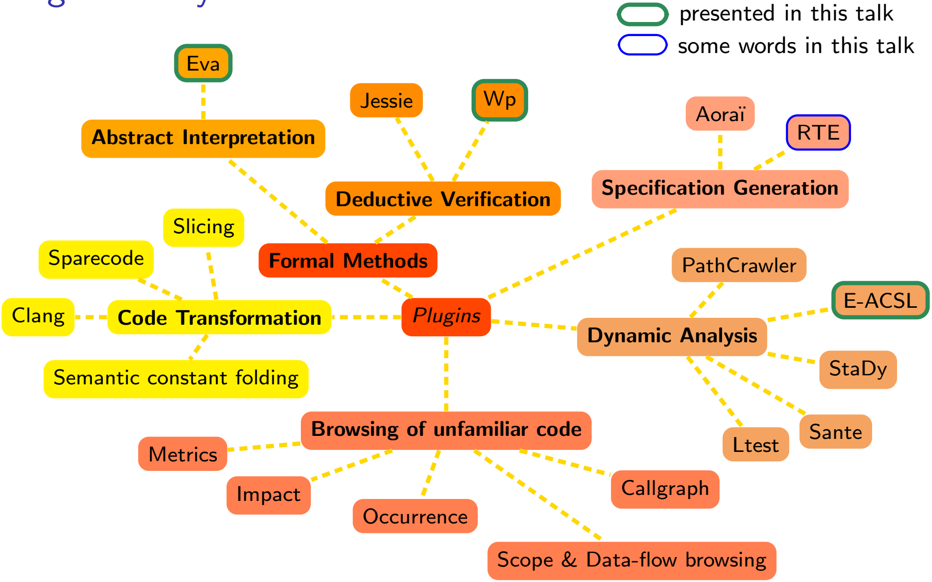
Can be proven with Frama-C/WP

Frama-C, a Collection of Tools

Several tools inside a single platform

- plugin architecture like in Eclipse
- tools provided as plugins
  - over 20 plugins in the open-source distribution
  - close-source plugins, either at CEA (about 20) or outside
- a common kernel
  - provides a uniform setting
  - provides general services
  - synthesizes useful information

Plugin Gallery



Frama-C, a Development Platform

- mostly developed in OCaml ($\approx$ 180 kloc in the open-source distribution, $\approx$ 300 kloc with proprietary extensions)
- initially based on Cil [Necula et al, CC'02]
- library dedicated to analysis of C code

development of plugins by third party

- dedicated plugins for specific task (verifying your coding rules)
- dedicated plugins for fine-grained parameterization
- extensions of existing analysers

Outline

A lightweight OS for IoT

Contiki is a lightweight operating system for IoT

It provides a lot of features (for a micro-kernel):

- (rudimentary) memory and process management
- networking stack and cryptographic functions
- ...

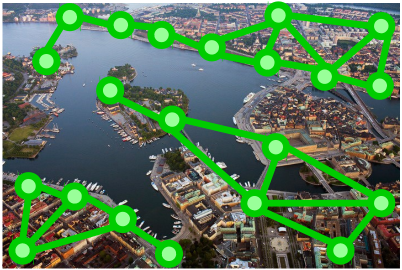
Typical hardware platform:

- 8, 16, or 32-bit MCU (little or big-endian),
- low-power radio, some sensors and actuators, ...

Note for security: there is *no* memory protection unit.

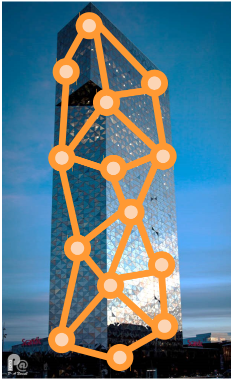
Contiki: Typical Applications

- IoT scenarios: smart cities, building automation, ...
- Multiple hops to cover large areas
- Low-power for battery-powered scenarios
- Nodes are interoperable and addressable (IP)



Traffic lights
Parking spots
Public transport
Street lights
Smart metering
...

Light bulbs
Thermostat
Power sockets
CO2 sensors
Door locks
Smoke detectors
...



Outline

Value Analysis Overview

Compute possible values of variables at each program point

- an automatic analysis
- based on abstract interpretation
- produces a correct over-approximation
- reports alarms for potentially invalid operations
- reports alarms for potentially invalid ACSL annotations
- can prove the absence of runtime errors
- graphical interface: displays the domains of each variable

Domains of Value Analysis

- Historical domains
  - small sets of integers, e.g. $\{5, 18, 42\}$
  - reduced product of intervals: quick to compute, e.g. $[1..41]$
  - modulo: pretty good for arrays of structures, e.g. $[1..41], 1\%2$
  - precise representation of pointers, e.g. *32-bit aligned offset from* $\&t[0]$
  - initialization information
- Eva, Evolved Value Analysis
  - more generic and extensible domains
  - possible to add new, or combine domains

Outline

Example 1

Run Eva: `frama-c-gui div1.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 0;
  }else{
    x = 5; y = 5;
  }
  sum = x + y;      // sum can be 0
  result = 10/sum; // risk of division by 0
  return result;
}
```

Example 1

Run Eva: `frama-c-gui div1.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 0;
  }else{
    x = 5; y = 5;
  }
  sum = x + y;      // sum can be 0
  result = 10/sum; // risk of division by 0
  return result;
}
```

Risk of division by 0 is detected, it is real.

Example 2

Run Eva: `frama-c-gui div2.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // sum cannot be 0
  result = 10/sum; // no div. by 0
  return result;
}
```

Example 2

Run Eva: `frama-c-gui div2.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // sum cannot be 0
  result = 10/sum; // no div. by 0
  return result;
}
```

Risk of division by 0 is detected, but it is a false alarm.

Eva Parameterization

- Eva is automatic, but can be imprecise due to overapproximation
- a fine-tuned parameterization for a trade-off precision / efficiency
- One useful option: slevel *n*
  - keep up to *n* states in parallel during the analysis
  - different slevel's can be set for specific functions or loops

Example 2, cont'd

Run Eva: `frama-c-gui div2.c -val -main=f -slevel 2`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // sum cannot be 0
  result = 10/sum; // no div. by 0
  return result;
}
```

Example 2, cont'd

Run Eva: `frama-c-gui div2.c -val -main=f -slevel 2`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // sum cannot be 0
  result = 10/sum; // no div. by 0
  return result;
}
```

Absence of division by 0 is proved, no false alarm.

Example 3

Run Eva: `frama-c-gui div3.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; //y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // y can be non-initialized
  result = 10/sum;
  return result;
}
```

Example 3

Run Eva: `frama-c-gui div3.c -val -main=f`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; //y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // y can be non-initialized
  result = 10/sum;
  return result;
}
```

Alarm on initialization of y is reported.

Example 3, cont'd

Run Eva: `frama-c-gui div3.c -val -main=f -slevel 2`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; //y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // y can be non-initialized
  result = 10/sum;
  return result;
}
```

Example 3, cont'd

Run Eva: `frama-c-gui div3.c -val -main=f -slevel 2`

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; //y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;  // y can be non-initialized
  result = 10/sum;
  return result;
}
```

Alarm on initialization of y is reported, even with a bigger slevel

Example 4

Run Eva: `frama-c-gui sqrt.c -val`

```
#include "__fc_builtin.h"
int A, B;
int root(int N){
  int R = 0;
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}

void main(void)
{
  A = Frama_C_interval(0,64);
  B = root(A);
}
```

Example 4

Run Eva: `frama-c-gui sqrt.c -val`

```
#include "__fc_builtin.h"
int A, B;
int root(int N){
  int R = 0;
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}

void main(void)
{
  A = Frama_C_interval(0,64);
  B = root(A);
}
```

Risk of arithmetic overflows is reported

Example 4, cont'd

Run Eva: `frama-c-gui sqrt.c -val -slevel 8`

```
#include "__fc_builtin.h"
int A, B;
int root(int N){
  int R = 0;
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}

void main(void)
{
  A = Frama_C_interval(0,64);
  B = root(A);
}
```

Example 4, cont'd

Run Eva: `frama-c-gui sqrt.c -val -slevel 8`

```
#include "__fc_builtin.h"
int A, B;
int root(int N){
  int R = 0;
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}

void main(void)
{
  A = Frama_C_interval(0,64);
  B = root(A);
}
```

Absence of overflows is proved with a bigger slevel

Example 5

Run Eva: `frama-c-gui pointer1.c -val`

```
#include "stdlib.h"

int main(void){
  int *p;
  if( p )
    *p = 10;
  return 0;
}
```

Example 5

Run Eva: `frama-c-gui pointer1.c -val`

```
#include "stdlib.h"

int main(void){
  int *p;
  if( p )
    *p = 10;
  return 0;
}
```

Alarm on initialization of p is reported

Example 6

Run Eva: `frama-c-gui pointer2.c -val`

```
#include "stdlib.h"

int main(void){
  int * p = (int*)malloc(sizeof(int));
  *p = 10;
  return 0;
}
```

Example 6

Run Eva: `frama-c-gui pointer2.c -val`

```
#include "stdlib.h"

int main(void){
  int * p = (int*)malloc(sizeof(int));
  *p = 10;
  return 0;
}
```

Alarm on validity of p is reported

Example 7

Run Eva: `frama-c-gui pointer3.c -val`

```
#include "stdlib.h"

int main(void){
  int * p = (int*)malloc(sizeof(int));
  if( p )
    *p = 10;
  return 0;
}
```

Example 7

Run Eva: `frama-c-gui pointer3.c -val`

```
#include "stdlib.h"

int main(void){
  int * p = (int*)malloc(sizeof(int));
  if( p )
    *p = 10;
  return 0;
}
```

Absence of runtime errors is proved

Outline

Overview of the `aes-ccm` Modules

- Critical! – Used for communication security
  - end-to-end confidentiality and integrity
- Advanced Encryption Standard (AES): a symmetric encryption algo.
  - AES replaced in 2002 Data Encryption Standard (DES)
- Modular API – independent from the OS
- Two modules:
  - AES-128
  - AES-CCM* block cypher mode
  - A few hundreds of LoC
- High complexity crypto code
  - Intensive integer arithmetics
  - Intricate indexing
  - based on multiplication over finite field $\mathrm{GF}(2^8)$

Examples 8, 9, 10

Analyze three versions of a part of the aes module

Explore and explain the results

Ex.8. Run Eva: `frama-c-gui aes1.c -val`

Ex.9. Run Eva: `frama-c-gui aes2.c -val`

Ex.10. Run Eva: `frama-c-gui aes3.c -val`

Examples 11, 12, 13, 14

Analyze three versions of a part of the `ccm` module

Explore and explain the results

Ex.11. Run Eva: `frama-c-gui ccm1.c -val`

Ex.12. Run Eva: `frama-c-gui ccm1.c -val -slevel 50`

Ex.13. Run Eva: `frama-c-gui ccm2.c -val -slevel 50`

Ex.14. Run Eva: `frama-c-gui ccm3.c -val -slevel 50`

Outline

Objectives of Deductive Verification

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

Outline

ACSL: ANSI/ISO C Specification Language

Presentation

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

Basic Components

- Typed first-order logic
- Pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-ins predicates and logic functions, particularly over pointers: `\valid(p)`, `\valid(p+0..2)`, `\separated(p+0..2,q+0..5)`, `\block_length(p)`

WP plugin

- Hoare-logic based plugin, developed at CEA List
- Proof of semantic properties of the program
- Modular verification (function by function)
- Input: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Z3, CVC3, CVC4, Yices, Simplify . . .
- WP manual at `http://frama-c.com/wp.html`
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?

WP plugin

- Hoare-logic based plugin, developed at CEA List
- Proof of semantic properties of the program
- Modular verification (function by function)
- Input: a program and its specification in ACSL
- WP generates verification conditions (VCs)
- Relies on Automatic Theorem Provers to discharge the VCs
  - Alt-Ergo, Z3, CVC3, CVC4, Yices, Simplify ...
- WP manual at `http://frama-c.com/wp.html`
- If all VCs are proved, the program respects the given specification
  - Does it mean that the program is correct?
  - NO! If the specification is wrong, the program can be wrong!

Outline

Introduction

Verification of absence of runtime errors using EVA

Deductive verification using WP

- Overview of ACSL and WP
- Function contracts
- Programs with loops
- An application to Contiki
- My proof fails... What to do?

Runtime Verification using E-ACSL

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 the caller)
  - Postcondition must be true on exit (ensured by the function)
- Nothing is guaranteed when the precondition is not satisfied
- Termination may be guaranteed or not (total or partial correctness)

Primary role of contracts

- Must reflect the informal specification
- Should not be modified just to suit the verification tasks

Example 1

Specify and prove the following program:

```
// returns the absolute value of x
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

Try to prove with Frama-C/WP using the basic command

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

Example 1 (Continued)

The basic proof succeeds for the following program:

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

- The returned value is not always as expected.

Example 1 (Continued)

The basic proof succeeds for the following program:

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

- The returned value is not always as expected.
- For `x=INT_MIN`, `-x` cannot be represented by an `int` and overflows
- Example: on 32-bit, `INT_MIN`$= -2^{31}$ while `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

WP can automatically check the absence of runtime errors

- Use the command `frama-c-gui -wp -wp-rte file.c`
- It generates VCs to ensure that runtime errors do not occur
  - in particular, arithmetic operations do not overflow
- If not proved, an error may occur.

Example 1 (Continued) - Solution

This is the completely specified program:

```
#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:

```
// returns the maximum of a and b
int max ( int a, int b ) {
  if ( a > b )
    return a ;
  return b ;
}
```

Example 2 (Continued) - Find the error

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

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

Example 2 (Continued) - a wrong version

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

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

Example 2 (Continued) - a wrong version

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

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

- Our specification is incomplete
- Should say that the returned value is one of the arguments

Example 2 (Continued) - Find another error

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

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

Example 2 (Continued) - a wrong version

With this specification, we cannot prove the following program. Why?

```
/*@ ensures \result >= a && \result >= b ;
    ensures \result == a || \result == b ; */
int max(int a, int b);

extern int v ;

int main(){
  v = 3;
  int r = max(4,2);
  //@ assert v == 3 ;
}
```

Example 2 (Continued) - a wrong version

With this specification, we cannot prove the following program. Why?

```
/*@ ensures \result >= a && \result >= b ;
    ensures \result == a || \result == b ; */
int max(int a, int b);

extern int v ;

int main(){
  v = 3;
  int r = max(4,2);
  //@ assert v == 3 ;
}
```

- Again, our specification is incomplete
- Should say that we do not modify any memory location

Assigns clause

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

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- If nothing can be modified, specify `assigns \nothing`

Example 2 (Continued) - Solution

This is the completely specified program:

```
/*@ ensures \result >= a && \result >= b;
    ensures \result == a || \result == b;
    assigns \nothing;
*/
int max ( int a, int b ) {
  if ( a >= b )
    return a ;
  return b ;
}
```

Example 3

Specify and prove the following program:

```
// 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 with the option -wp-rte for the program:

```
/*@ 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 with the option `-wp-rte` for the program:

```
/*@ 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.

- WP can automatically generate VCs to check memory access validity
  - use the command `frama-c-gui -wp -wp-rte file.c`
- 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.
  - `\valid(p)` for one pointer `p`
  - `\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?

```
/*@ 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 ;
}
```

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`

Assigns clause

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

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- If nothing can be modified, specify `assigns \nothing`

Assigns clause

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

- Part of the postcondition
- Specifies which (non local) variables can be modified by the function
- If nothing can be modified, specify `assigns \nothing`
- Avoids to state for all unchanged global variables `v`:
  `ensures \old(v) == v;`
- Avoids to forget one of them: explicit permission is required

Example 3 (Continued) - Solution

This is the completely specified program:

```
/*@ 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 ;
}
```

Example 4

Specify and prove the following program:

```
/* swaps two pointed values */
void swap(int *a, int *b){
  int tmp = *a ; *a = *b ; *b = tmp ;
}
```

Example 4 - Solution

This is the completely specified program:

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

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 5

Specify using behaviors and prove the function abs:

```
// returns the absolute value of x
int abs ( int x ) {
  if ( x >=0 )
    return x ;
  return -x ;
}
```

Example 5 (Continued) - Solution

```
#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 ;
}
```

Contracts and function calls

// Pre_f assumed
`f(<args>){`
`  code1;`
// Pre_g to be proved
`  g(<args>);`
// $Post_g$ assumed
`  code2;`
`}`
// $Post_f$ to be proved

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

- Pre of the caller is assumed, Post of the caller must be ensured
- Pre of the callee must be ensured, Post of the callee is assumed

Example 6

Specify and prove the function `max_abs`

```
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 6 (Continued) - Explain the proof failure for

```
#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 6 (Continued) - Explain the proof failure for

```
#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 6 (Continued) - Solution

```
#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 );

/*@ 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

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 before the loop condition check

In particular, a for loop

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

should be seen as

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

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 7

Specify and prove the function `reset_array`:

```
// writes 0 in each cell of the
// array a of len integers
void reset_array(int* a, int len){
  for(int i = 0 ; i < len ; ++i){
    a[i] = 0 ;
  }
}
```

Example 7 (Continued) - Solution

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

Example 8

Specify and prove the function all_zeros:

```
// 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 8 (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 assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Example 9

Specify and prove the function sqrt:

```
/* takes as input an integer and returns
   its (integer) square root */
int root(int N){
  int R = 0;
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}
```

Example 9 (Continued) - Solution

```
/*@
  requires 0 <= N <= 1000000000;
  assigns \nothing;
  ensures \result * \result <= N ;
  ensures N < (\result+1) * (\result+1);
*/
int root(int N){
  int R = 0;
  /*@
    loop invariant 0 <= R * R <= N;
    loop assigns R;
    loop variant N-R;
  */
  while(((R+1)*(R+1)) <= N) {
    R = R + 1;
  }
  return R;
}
```

Outline

Overview of the `memb` Module

- No dynamic allocation in Contiki
  - to avoid fragmentation of memory in long-lasting systems
- Memory is pre-allocated (in arrays of blocks) and attributed on demand
- The management of such blocks is realized by the `memb` module

The `memb` module API allows the user to

- initialize a `memb` store (i.e. pre-allocate an array of blocks),
- allocate or free a block,
- check if a pointer refers to a block inside the store
- count the number of allocated blocks

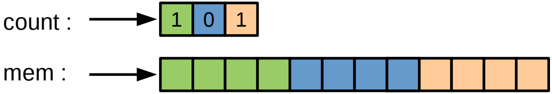
memb Data structure

```
struct memb {
  unsigned short size;
  unsigned short num;
  char *count;
  void *mem;
};
```

For example:

size = 4

num = 3



memb allocation function

```
void * memb_alloc(struct memb *m)
{
  for(int i = 0; i < m->num; ++i) {
    if(m->count[i] == 0) {
      ++(m->count[i]);
      int offset = i * m->size ;
      return (void *)((char *)m->mem + offset);
    }
  }
  return NULL;
}
```

Two behaviors:

- if a block is available, it is marked as busy, and its address is returned
- if no block is available, the function returns NULL

memb allocation function

In the specification that is provided, there are missing parts.
Hints:

- `requires`: the precondition of this function is some kind of validity
- `assumes`: we need to express that a free block exists
- `ensures`: the `_memb_numfree` expresses the number of free blocks
- `loop invariant`: we already expressed this kind of invariant

Outline

Proof failures

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

- incorrect implementation (→ check your code)
- incorrect annotation (→ check your spec)
- missing or erroneous (previous) annotation (→ check your spec)
- insufficient timeout (→ 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
  - ...

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)

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

Outline

Objectives of E-ACSL

- Frama-C initially designed as a static analysis platform
- Extended with plugins for dynamic analysis
- E-ACSL: runtime assertion checking tool
  - detect runtime errors
  - detect annotation failures
  - treat a concrete program run (i.e. concrete inputs)

E-ACSL plugin at a Glance

http://frama-c.com/eacsl.html

- convert E-ACSL annotations into C code
- implemented as a Frama-C plugin

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

$\xrightarrow{\text{E-ACSL}}$

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}
```

E-ACSL plugin at a Glance

http://frama-c.com/eacsl.html

- convert E-ACSL annotations into C code
- implemented as a Frama-C plugin

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  return x / (y-1);
}
```

$\xrightarrow{\text{E-ACSL}}$

```
int div(int x, int y) {
  /*@ assert y-1 != 0; */
  e_acsl_assert(y-1 != 0);
  return x / (y-1);
}
```

- the general translation is more complex than it may look

Outline

Example 1

Consider file `01-main1.c`:

```c
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 0;
  }else{
    x = 5; y = 5;
  }
  sum = x + y;
  //@ assert sum != 0;
  result = 10 / sum;
  return result;
}
```

```c
int main(void){
  f(42);
  f(0);
  return 0;
}
```

Example 1

Consider file `01-main1.c`:

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 0;
  }else{
    x = 5; y = 5;
  }
  sum = x + y;
  //@ assert sum != 0;
  result = 10 / sum;
  return result;
}
```

```
int main(void){
  f(42);
  f(0);
  return 0;
}
```

```
frama-c -e-acsl <main.c> -then-last \
    -print -ocode monitored_main.c
```

Example 1

Consider file 01-main1.c:

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 0;
  }else{
    x = 5; y = 5;
  }
  sum = x + y;
  //@ assert sum != 0;
  result = 10 / sum;
  return result;
}
```

```
int main(void){
  f(42);
  f(0);
  return 0;
}
```

```
frama-c -e-acsl <main.c> -then-last \
    -print -ocode monitored_main.c
```

generates monitored_main.c that contains:

```
e_acsl_assert(sum != 0, "Assertion", "f", "sum != 0", 10);
```

Example 1

- Compiling `monitored_main.c` requires several libraries
- The E-ACSL plugin provides a convenient script to instrument and compile the program: `e-acsl-gcc.sh`

Example 1

- Compiling `monitored_main.c` requires several libraries
- The E-ACSL plugin provides a convenient script to instrument and compile the program: `e-acsl-gcc.sh`

```
e-acsl-gcc.sh <main.c> -c -O monitored_main
```

- `monitored_main`: the executable without runtime monitoring
- `monitored_main.eacsl`: the executable with runtime monitoring

Example 1

- Compiling `monitored_main.c` requires several libraries
- The E-ACSL plugin provides a convenient script to instrument and compile the program: `e-acsl-gcc.sh`

```
e-acsl-gcc.sh <main.c> -c -O monitored_main
```

- `monitored_main`: the executable without runtime monitoring
- `monitored_main.eacsl`: the executable with runtime monitoring

```
./monitored_main.eacsl
```

```
Assertion failed at line 10 in function f.
The failing predicate is:
sum != 0.
Aborted (core dumped)
```

Example 1, part 2

Consider file `01-main2.c`:

```
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;
  //@ assert sum != 0;
  result = 10 / sum;
  return result;
}
```

```
int main(void){
  f(42);
  f(0);
  return 0;
}
```

Example 1, part 2

Consider file `01-main2.c`:

```c
int f ( int a ) {
  int x, y;
  int sum, result;
  if(a == 0){
    x = 0; y = 5;
  }else{
    x = 5; y = 0;
  }
  sum = x + y;
  //@ assert sum != 0;
  result = 10 / sum;
  return result;
}
```

```c
int main(void){
  f(42);
  f(0);
  return 0;
}
```

```
./monitored_main.eacsl
```

- No output
- Both calls to f are error-free

Example 2

```
#include "stdlib.h"

struct list {
  struct list *next;
  int value;
};

/*@
  requires \valid(list);
  assigns *list;
*/
void list_init(struct list ** list) {
  *list = NULL;
}

int main(void){
  struct list ** l = malloc(sizeof(void *));
  list_init(l);
  free(l);
  list_init(l);
}
```

Example 2

Two features of the E-ACSL plugin:

- Function contract checking
- Runtime error detection

Example 2

Two features of the E-ACSL plugin:

- Function contract checking
- Runtime error detection

In the example (file `02-list1.c`):

- At each call to list_init the contract is checked

Example 2

Two features of the E-ACSL plugin:

- Function contract checking
- Runtime error detection

In the example (file `02-list1.c`):

- At each call to list_init the contract is checked

```
./monitored_list.eacsl
```

```
Precondition failed at line 8 in function list_init.
The failing predicate is:
\valid(list).
Aborted (core dumped)
```

Example 2

Two features of the E-ACSL plugin:

- Function contract checking
- Runtime error detection

In the example (file `02-list1.c`):

- At each call to list_init the contract is checked

```
./monitored_list.eacsl
```

```
Precondition failed at line 8 in function list_init.
The failing predicate is:
\valid(list).
Aborted (core dumped)
```

Monitoring memory related constructs requires:

- keeping track of the program memory at runtime
- using a dedicated memory runtime library

Outline

From ACSL to E-ACSL

- ACSL was designed for static analysis tools only
- based on logic and mathematics
- cannot execute any term/predicate (e.g. unbounded quantification)
- cannot be used by dynamic analysis tools (e.g. testing or monitoring)
- E-ACSL: executable subset of ACSL [Delahaye et al., RV'13]
  - few restrictions
  - one compatible semantics change

E-ACSL Restrictions

- quantifications must be guarded

```
\forall τ1 x1,..., τn xn;
  a1 <= x1 <= b1 && ... && an <= xn <= bn
  ==> p

\exists τ1 x1,..., τn xn;
  a1 <= x1 <= b1 && ... && an <= xn <= bn
  && p
```

- sets must be finite
- no lemmas nor axiomatics
- no way to express termination properties

Outline

An Application to Contiki: Example 3

Example `list_chop` (started):

```
struct list
{
  struct list *next;
  int value;
};

/*@
  requires \valid(list);
  requires 0 <= length(*list);
*/
struct list * list_chop(struct list ** list){
  // removes the last element of the list
}
```

An Application to Contiki: Example 3

Example `list_chop` (cont'd):

```
int main(void){
  struct list node;
  node.value = 1;
  node.next  = &node;

  struct list * l = &node;

  l = list_chop(&l);
}
```

- List l is cyclic, that can be detected by length
  - length should not be positive for a cyclic list
- Our goal: verify the contract of `list_chop` and detect that `l` is cyclic

An Application to Contiki: Example 3

Example `list_chop` (cont'd):

```
int main(void){
  struct list node;
  node.value = 1;
  node.next  = &node;

  struct list * l = &node;

  l = list_chop(&l);
}
```

- List l is cyclic, that can be detected by length
  - length should not be positive for a cyclic list
- Our goal: verify the contract of `list_chop` and detect that `l` is cyclic
- Contiki API: `int list_length(struct list **);`

⇒ the length of a list should be at most `INT_MAX`

An Application to Contiki: Example 3

```
/*@
  logic int length_aux{L}(struct list * l,
                                        int n)=
    n < (int)0 ? ((int)-1) :
      l == NULL ? n :
        n < INT_MAX ?
          length_aux(l->next, (int)(1+n)) :
          ((int)-1);

  logic int length{L}(struct list * l) =
    length_aux(l, (int)0);
*/
```

An Application to Contiki: Example 3

```
/*@
  logic int length_aux{L}(struct list * l,
                                        int n)=
    n < (int)0 ? ((int)-1) :
      l == NULL ? n :
        n < INT_MAX ?
          length_aux(l->next, (int)(1+n)) :
          ((int)-1);

  logic int length{L}(struct list * l) =
    length_aux(l, (int)0);
*/
```

- The E-ACSL specification language supports logical functions
- The E-ACSL plugin does not yet

An Application to Contiki: Example 3

```
/*@
  logic int length_aux{L}(struct list * l,
                                            int n)=
    n < (int)0 ? ((int)-1) :
      l == NULL ? n :
        n < INT_MAX ?
          length_aux(l->next, (int)(1+n)) :
          ((int)-1);

  logic int length{L}(struct list * l) =
    length_aux(l, (int)0);
*/
```

- The E-ACSL specification language supports logical functions
- The E-ACSL plugin does not yet

$\Rightarrow$ let us implement C function equivalent to length and use it to verify `0 <= length(l)` (that is, l is non cyclic) at runtime

An Application to Contiki: Example 3 – part 1 (WP)

Prove the equivalence of the logical and the recursive C functions, file `03-wp_list_1.c`:

```
/*@ ensures \result == length_aux(l, n);
  @ assigns \nothing; */
int length_aux(struct list * l, int n){
  if (n < 0)
    return -1;
  else if (l == NULL)
    return n;
  else if (n < INT_MAX)
    return length_aux(l->next, n+1);
  else
    return -1;
}
/*@ ensures \result == length(l);
  @ assigns \nothing; */
int length(struct list * l){
  return length_aux(l, 0);
}
```

An Application to Contiki: Example 3 – part 2 (WP)

Prove the equivalence of the logical and the iterative C functions (additional annotations will be needed), file `03-wp_list_2.c`:

```
/*@ ensures \result == length(list);
  @ assigns \nothing; */
int length(struct list * list){
  int len = 0;
  struct list * l = list;

  while(l != NULL && len < INT_MAX){
    l = l->next;
    len ++;
  }
  if(l!=NULL){
    return -1;
  }
  else
    return len;
}
```

An Application to Contiki: Example 3 – part 3 (E-ACSL)

Now with one of the C versions of length:

- We generate the annotated C code
- In function __gen_e_acsl_list_chop we add:

```
  __e_acsl_assert(0<=length(*list),
                  (char*)"Precondition",
                  (char *)"list_chop",
                  (char*)"0<=length(l)",
                  60);
```

- option -C considers that the C file is already instrumented
- Exercise: compile the modified instrumented file `03-list_3.c`: and run it

Outline

Possible Usage in Combination with Other Tools

- check unproved properties of static analyzers (e.g. Value, WP)
- check the absence of runtime error in combination with RTE
- check memory consumption and violations (use-after-free)
- help testing tools by checking properties which are not easy to observe
- complement program transformation tools
  - temporal properties (Aoraï)
  - information flow properties (SecureFlow)

Outline

Conclusion

We have presented how to:

- verify the absence of runtime errors with Eva
- formally specify functional properties with ACSL
- prove a programs respects its specification with WP
- verify annotations at runtime or detect runtime errors with E-ACSL

All of these and much more inside Frama-C

May be used for:

- teaching
- academic prototyping
- industrial applications

http://frama-c.com

Further reading

User manuals:

- user manuals for Frama-C and its different analyzers, on the website:
  `http://frama-c.com`

About the use of WP:

- Introduction to C program proof using Frama-C and its WP plugin
  Allan Blanchard
  `https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf`
- ACSL by Example
  Jochen Burghardt, Jens Gerlach
  `https://github.com/fraunhoferfokus/acsl-by-example`

Further reading

Other tutorial papers:

- on deductive verification:
  N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C (TAP 2013)
- on runtime verification:
  - N. Kosmatov and J. Signoles. A lesson on runtime assertion checking with Frama-C (RV 2013)
  - N. Kosmatov and J. Signoles. Runtime assertion checking and its combinations with static and dynamic analyses (TAP 2014)
- on test generation:
  N. Kosmatov, N. Williams, B. Botella, M. Roger, and O. Chebaro. A lesson on structural testing with PathCrawler-online.com (TAP 2012)
- on analysis combinations:
  N. Kosmatov and J. Signoles. Frama-C, A collaborative framework for C code verification: Tutorial synopsis (RV 2016)

Further reading

More details on the verification of Contiki:

- on the MEMB module:
  F. Mangano, S. Duquennoy, and N. Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study (CRiSIS 2016)
- on the AES-CCM* module:
  A. Peyrard, S. Duquennoy, N. Kosmatov, and S. Raza. Towards formal verification of Contik: Analysis of the AES–CCM* modules with Frama-C (RED-IoT 2017)
- on the LIST module:
  - A. Blanchard, N. Kosmatov, and F. Loulergue. Ghosts for lists: A critical module of contiki verified in Frama-C (NFM 2018)
  - F. Loulergue, A. Blanchard, and N. Kosmatov. Ghosts for lists: from axiomatic to executable specifications (TAP 2018)