Approches formelles pour les composants et outils système

Frédéric Loulergue

Journées AFADL, 18 Juin 2025

Équipe LMV : Langages, Modèles, Vérification
LIFO: Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans, France

Work in collaboration with

- Nikolai Kosmatov
- Allan Blanchard
- Yani Ziani
- Téo Bernier
- Julien Signoles
- Dara Ly
- Jérémy Damour
- Loïc Correnson
- Matthieu Lemerre
- Daniel Gracia Pérez
- Hélène Coullon
- Jolan Philippe
- Simon Robillard
- Ludovic Henrio
- Farid Arfi

SAET Meteor

- Line 14 metro in Paris (1998)
- B method
- Safety-critical software parts: onboard section, line
- Safety-critical software still in version 1.0

Florian Schütz, 2004

CompCert

- Optimizing C compiler in Coq
- `https://compcert.org`

- Study by Yang et al [1]:
  - 70 bugs in GCC
    25 release-blocking bugs
  - 202 bugs in CLANG
  - "*CompCert is the only compiler we have tested for which Cmith cannot find wrong-code errors.*"

Verified JavaCard Virtual Machine

- C implementation of a JavaCard VM
- Djoudi et al. [2]
- Verification of global security properties with Frama-C
- EAL7 certification

Outline

- Deductive Verification with Frama-C
  - An introduction to Frama-C
  - Verification of a Contiki Module with Frama-C
  - Memory Models for Verification with Frama-C
- Formalization of Configuration Languages

Deductive Verification with Frama-C

Deductive Verification with Frama-C

An Introduction to Frama-C

Frama-C Open-Source Distribution

Framework for Analysis of source code written in ISO C

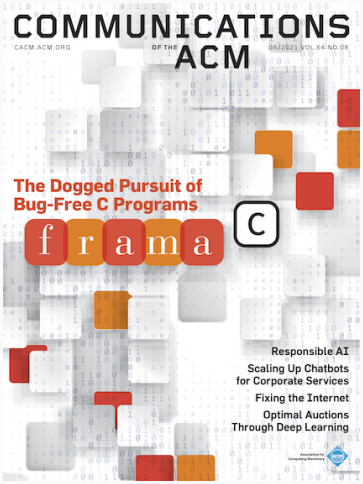
- analysis of C code extended with ACSL annotations
- ACSL Specification Language: *langua franca* of Frama-C analyzers
- mostly open-source (LGPL 2.1) `https://frama-c.com`
- also proprietary extensions and distributions
- targets both academic and industrial usage

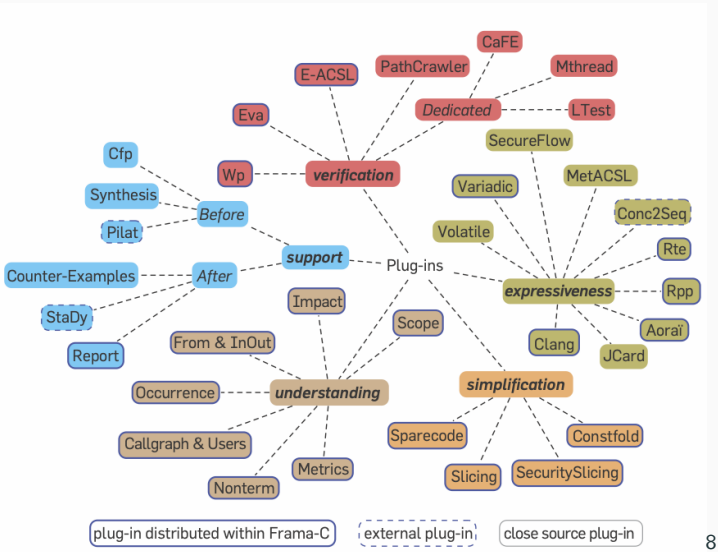
Frama-C: a Collection of Tools

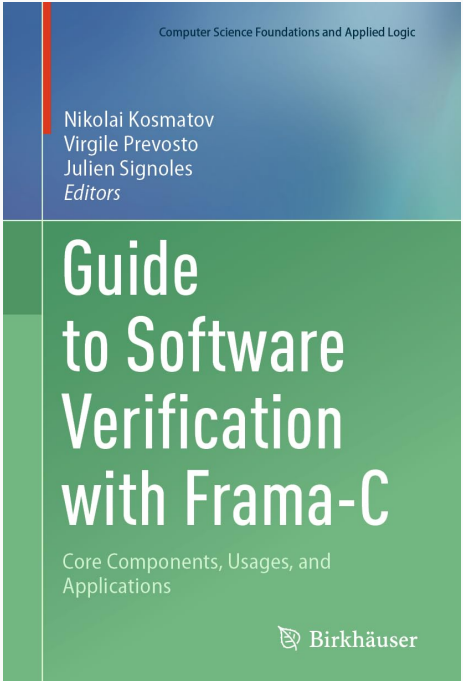
Several tools inside a single platform

- 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






Computer Science Foundations and Applied Logic
Nikolai Kosmatov
Virgile Prevosto
Julien Signoles
Editors
Guide to Software Verification with Frama-C
Core Components, Usages, and Applications
Birkhäuser

Focus of this introduction: deductive verification

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

Plugin for deductive verification

- WP
- Related documentation: WP User manual, ACSL language reference, ACSL language implementation.

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

Basic Components

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

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, CVC4, CVC5, ...
- 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, CVC4, CVC5, ...
- 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!

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)

Example 1

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

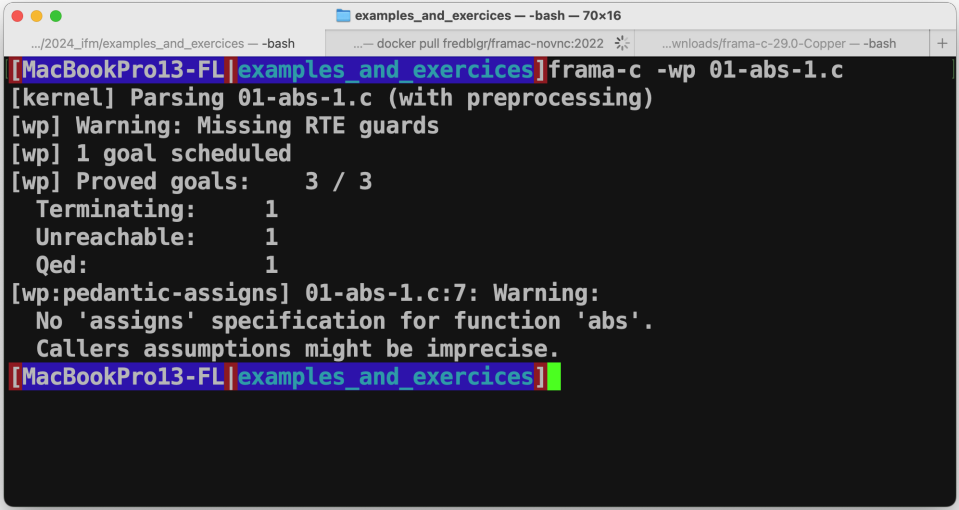
- CLI: `frama-c -wp file.c`
- GUI: `frama-c-gui -wp file.c`
- New GUI: `ivette -wp file.c`

Example 1 (continued)

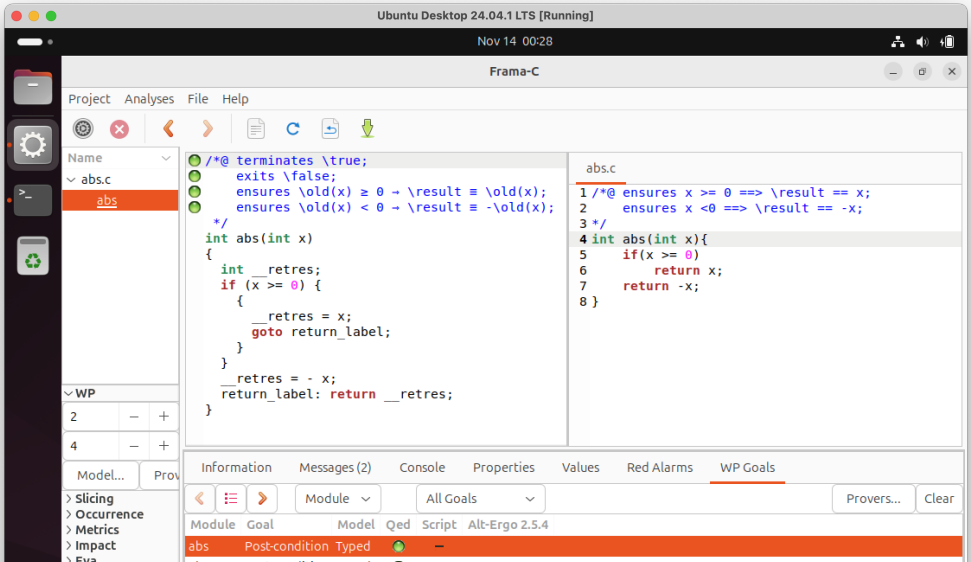
The basic proof succeeds for the following program:

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

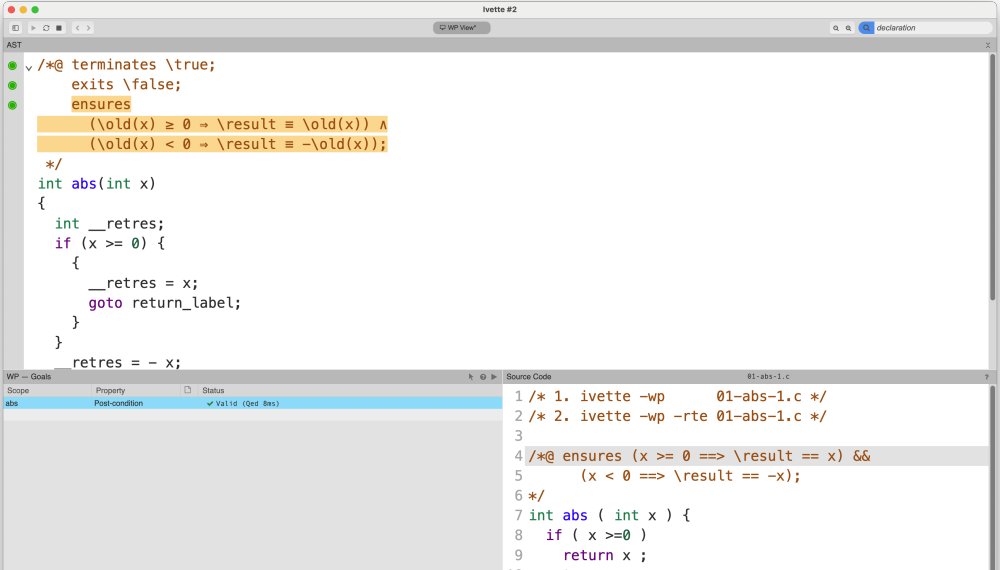
Example 1: CLI



Example 1: GUI



Example 1: New GUI



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 ;
}
```

There is a warning: Missing RTE guards

Safety warnings: arithmetic overflows

Absence of arithmetic overflows can be important to check

- A sad example: crash of Ariane 5 in 1996

WP can check the absence of runtime errors:

- To add RTE guards: use the RTE plugin
- 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

- Before the statement that contains −x:
  `/*@ assert rte: signed_overflow: -2147483647 <= x; */`
- Proof now fails...

Example 1 (continued): solution

This program is proved:

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

If **assigns \nothing** is omitted, the following program cannot be verified:

```
int answer = 42;
void main(){
  int x = abs(-1);
  /*@ assert x == 1; */
  /*@ assert answer == 42; */
}
```

Example 1 modified: problem!

All VC are proved! What's the problem?

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

- `frama-c -wp -rte -wp-smoke-tests 01-abs-3.c`
- Smoke tests fail

Example 1: variant with behaviors

```
/*@ 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 )
```

Safety warnings: invalid memory accesses

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

- WP can automatically generate VCs to check memory access validity: RTE
- 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 2

Specify and prove the following program:

```
#include<stddef.h>
void copy (size_t size, int * src, int * dst){
  for(size_t i=0; i < size; i++) dst[i] = src[i];
}
```

Example 2: first try for a contract

```
#include<stddef.h>
/*@ requires \valid(src + (0..size-1)) && \valid(dst + (0..size-1));
  @ ensures \forall integer k; 0 <= k < size ==> dst[k] == src[k];
  @ assigns dst[0..size-1];
*/
void copy (size_t size, int * src, int * dst){
  for(size_t i=0; i < size; i++) dst[i] = src[i];
}
```

When there are loops, the tool needs additional annotations: loop invariants

Example 2: adding loop annotations

```
/*@ requires \valid(src + (0..size-1)) && \valid(dst + (0..size-1));
  @ ensures \forall integer k; 0 <= k < size ==> dst[k] == src[k];
  @ assigns dst[0..size-1];
*/
void copy (size_t size, int * src, int * dst){
  /*@ loop invariant 0 <= i <= size;
      loop invariant \forall integer k; 0 <= k < i ==> dst[k] == src[k];
      loop assigns i, dst[0..size-1];
      loop variant size - i;
  */
  for(size_t i=0; i < size; i++) dst[i] = src[i];
}
```

Still not proved: why?

Example 2, solution: adding a separation requirement

```
/*@ requires \valid(src + (0..size-1)) && \valid(dst + (0..size-1));
  @ requires \separated(src + (0..size-1), dst + (0..size-1));
  @ ensures \forall integer k; 0 <= k < size ==> dst[k] == src[k];
  @ assigns dst[0..size-1];
*/
void copy (size_t size, int * src, int * dst){
  /*@ loop invariant 0 <= i <= size;
      loop invariant \forall integer k; 0 <= k < i ==> dst[k] == src[k];
      loop assigns i, dst[0..size-1];
      loop variant size - i;
  */
  for(size_t i=0; i < size; i++) dst[i] = src[i];
}
```

Deductive Verification with Frama-C

Verification of a Contiki Module with Frama-C

A lightweight OS for IoT

Contiki is a lightweight operating system for IoT

It provides a lot of features:

- (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.

Overview

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

The `list` module provides a generic list API for linked lists:

- about 176 LOC (excl. macros)
- required by 32 modules of Contiki
- more than 250 calls in the core part of Contiki

List: a rich API

```
struct list {
  struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int  list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

List: a rich API

Observers

```
struct list {
  struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int  list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

List: a rich API

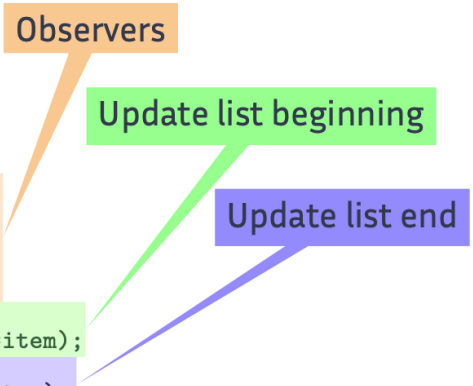
```
struct list {
  struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int  list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

Observers

Update list beginning

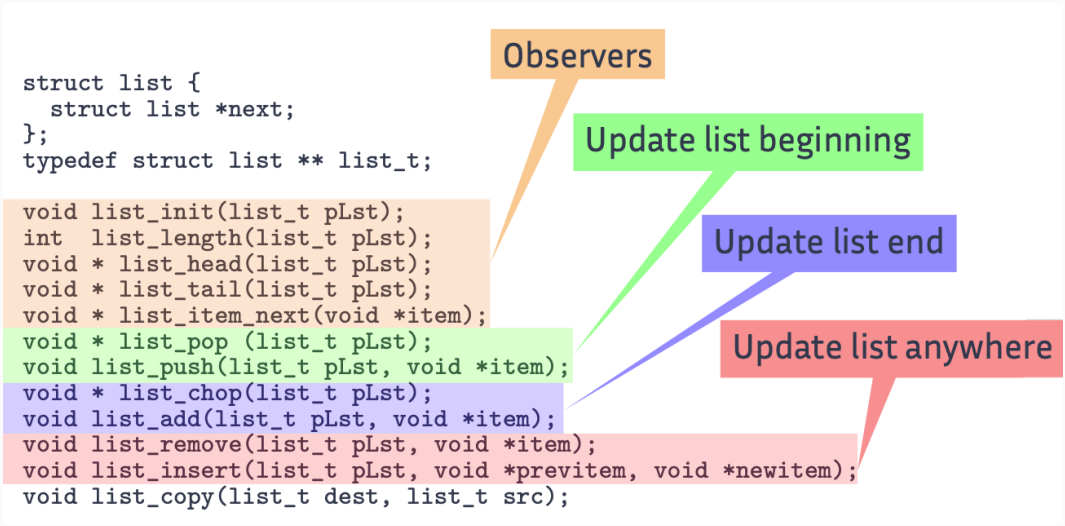
List: a rich API

```
struct list {
  struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int  list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

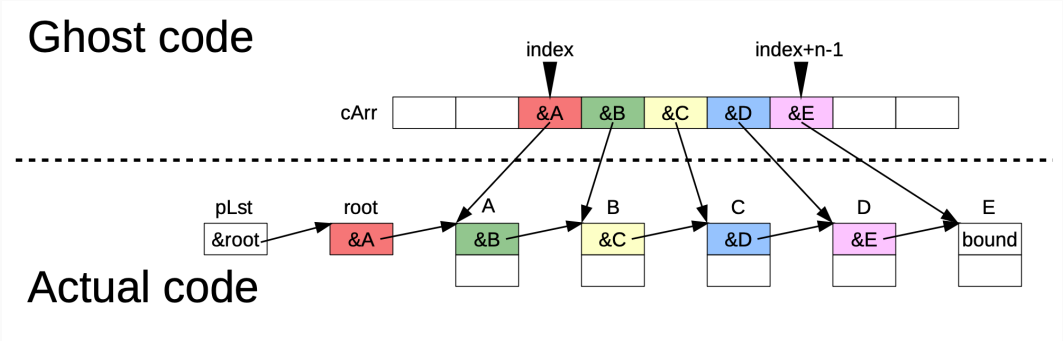


List: a rich API

```
struct list {
  struct list *next;
};
typedef struct list ** list_t;

void list_init(list_t pLst);
int  list_length(list_t pLst);
void * list_head(list_t pLst);
void * list_tail(list_t pLst);
void * list_item_next(void *item);
void * list_pop (list_t pLst);
void list_push(list_t pLst, void *item);
void * list_chop(list_t pLst);
void list_add(list_t pLst, void *item);
void list_remove(list_t pLst, void *item);
void list_insert(list_t pLst, void *previtem, void *newitem);
void list_copy(list_t dest, list_t src);
```

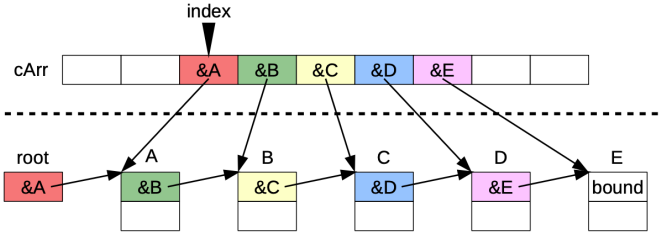


Idea: a ghost array that stores the addresses of the list elements



First formalization approach: inductive representation predicate

Ghost code

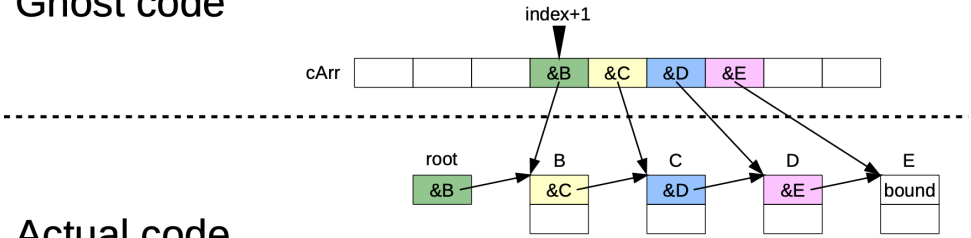


Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
    linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
      linked_n(root, cArr, index, n, bound);
}
```

First formalization approach: inductive representation predicate

Ghost code

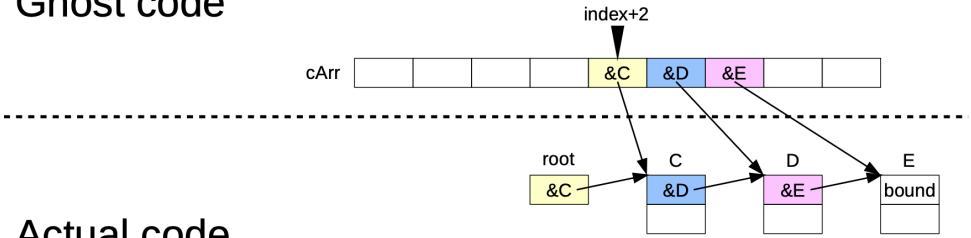


Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
    linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
      linked_n(root, cArr, index, n, bound);
}
```

First formalization approach: inductive representation predicate

Ghost code

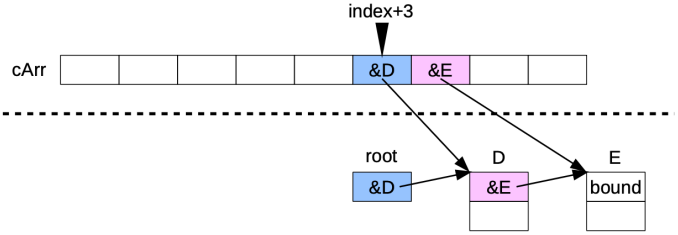


Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
    linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
      linked_n(root, cArr, index, n, bound);
}
```

First formalization approach: inductive representation predicate

Ghost code

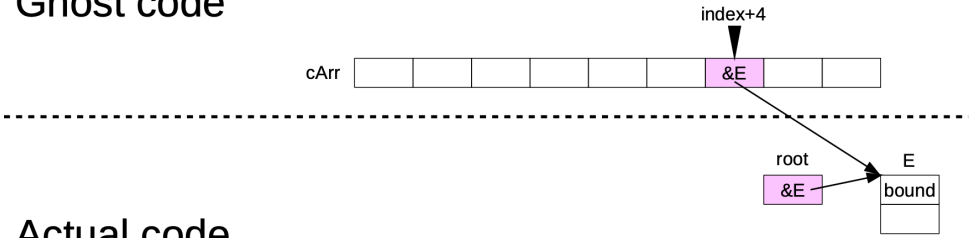


Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
    linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
      linked_n(root, cArr, index, n, bound);
}
```

First formalization approach: inductive representation predicate

Ghost code

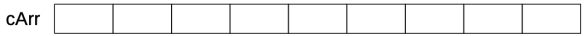


Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
// ...
case linked_n_cons{L}:
  \forall struct list *root, **cArr, *bound, integer index, n;
    /*indexes properties*/ ==> \valid(root) ==> root == cArr[index] ==>
    linked_n(root->next, cArr, index + 1, n - 1, bound) ==>
      linked_n(root, cArr, index, n, bound);
}
```

First formalization approach: inductive representation predicate

Ghost code



Actual code

```
inductive linked_n{L}(struct list *root, struct list **cArr,
                      integer index, integer n, struct list *bound) {
case linked_n_bound{L}:
  \forall struct list **cArr, *bound, integer index;
    0 <= index <= MAX_SIZE ==> linked_n(bound, cArr, index, 0, bound);
// ...
}
```

First formalization approach: ghost code

To specify and verify the API

- When `linked_n` holds it's easy to reason about the content of the list
- When the list is modified, the ghost array should be modified accordingly
- Separation hypotheses needed

Lemmas are required

```
/*@ lemma linked_split_segment:
  \forall struct list *root, **cArr, *bound, *AddrC,
    integer i, n, k;
  n > 0 ==> k >= 0 ==>
  AddrC == cArr[i + n - 1]->next ==>
  linked_n(root, cArr, i, n + k, bound) ==>
  (linked_n(root, cArr, i, n, AddrC) &&
   linked_n(AddrC, cArr, i + n, k, bound)); */
```

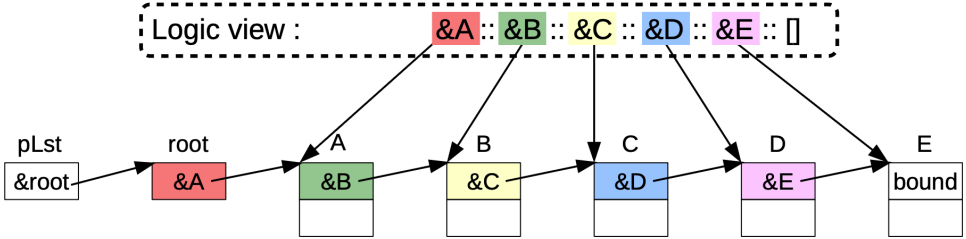
Spec. and ghost code

- 46 lines for ghost functions
- 500 lines for contracts
- 240 lines for logic def and lemmas
- 650 lines of other annotations

798 proof obligations

- 96.7% are automatically discharged
- 24 are lemmas proved with Coq
- 2 assertions proved with Coq

Second formalization approach: logics lists




```
/*@
  inductive linked_ll{L}(list *bl, list *el, \list<list*> ll) {
  case linked_ll_nil{L}: ∀ list *el;
    linked_ll{L}(el, el, \Nil);
  case linked_ll_cons{L}: ∀ list *bl, *el, \list<list*> tail;
    \separated(bl, el) ⇒ \valid(bl) ⇒
    linked_ll{L}(bl->next, el, tail) ⇒
    separated_from_list(bl, tail) ⇒
      linked_ll{L}(bl, el, \Cons(bl, tail));
  }
```

Second formalization approach: logics lists

First attempt, we would like to write:

```
/*@ ∃ list* hd, \list<list*> l ;
  @
  @ ...
  @
  @ behavior not_empty:
  @   assumes *list ≠ NULL ;
  @   requires linked_ll(*list, NULL, \Cons(hd, l));
  @   assigns *list ;
  @   ensures \result == hd == \old(*list) ;
  @   ensures linked_ll(*list, NULL, l);
  @*/
list * list_pop(list_t list);
```

...but it's not possible (existential scope)

Second formalization approach: logics lists

Second attempt, let's write a logic function that builds the logic list:

```
/*@ axiomatic To_ll {
    logic \list<list*> to_ll{L}(list* bl, list* el)
      reads { e->next
            | list* e ; \valid(e) ∧ in_list(e, to_ll(bl, el)) } ;

    axiom to_ll_nil{L}: ∀ list *el ;
      to_ll{L}(el, el) == \Nil ;

    axiom to_ll_cons{L}: ∀ list *bl, *el ;
      \separated(bl, el) ⇒
      \valid(bl) ⇒
      ptr_separated_from_list(bl, to_ll{L}(bl->next, el)) ⇒
        to_ll{L}(bl, el) == (\Cons(bl, to_ll{L}(bl->next, el))) ;
  }
*/
```

Second formalization approach: logics lists

and let's use it:

```
/*@ requires linked_ll(*list, NULL, to_ll(*list, NULL));
  @ assigns *list ;
  @
  @ ensures linked_ll(*list, NULL, to_ll(*list, NULL));
  @
  @ behavior empty:
  @   assumes *list == NULL ;
  @   ensures \result == NULL ;
  @
  @ behavior not_empty:
  @   assumes *list ≠ NULL ;
  @   ensures \result == \old(*list) ;
  @   ensures to_ll{Pre}(\at(*list, Pre), NULL) ==
  @           ([| \at(*list, Pre) |] ^ to_ll(*list, NULL));
  @
  @ complete behaviors; disjoint behaviors;
  @*/
list * list_pop(list_t list);
```

Second formalization approach: logics lists

as in the first approach, we need lemmas, for e.g.:

```
/*@
  lemma linked_ll_split{L}:
    ∀ struct list *bl, *el, \list<struct list*> l1, l2;
      linked_ll(bl, el, l1 ^ l2) ⇒ l1 ≠ \Nil ⇒
        \let bound = \nth(l1, \length(l1) - 1)->next ;
        linked_ll(bl, bound, l1) ∧ linked_ll(bound, el, l2) ;
*/

/*@
  lemma to_logic_list_split{L}:
    ∀ struct list *bl, *el, *sep, \list<struct list*> ll;
      ll ≠ \Nil ⇒ linked_ll(bl, el, ll) ⇒
      ll == to_logic_list(bl, el) ⇒
      in_list(sep, ll) ⇒
        ll == (to_logic_list(bl, sep) ^ to_logic_list(sep, el));
*/
```

Second formalization approach: logics lists

Specifications and annotations

- 410 lines for contracts
- 270 lines for logic definitions and lemmas
- 1020 lines for guiding annotations

757 (500) proof obligations

- 33 are lemmas proved with Coq
- 2 assertions proved with Coq
- all others automatically discharged

Lemmas: replacing Coq proofs by “proofs" in Frama-C

- Use Coq mostly when proofs by induction are needed
- The Coq formalization is not documented
- It changes quite often
- Proving Frama-C lemmas is not for the casual Coq user

Lemmas: replacing Coq proofs by “proofs" in Frama-C

- Use Coq mostly when proofs by induction are needed
- The Coq formalization is not documented
- It changes quite often
- Proving Frama-C lemmas is not for the casual Coq user

Can we get rid of Coq?

Lemmas: replacing Coq proofs by “proofs" in Frama-C

- Use Coq mostly when proofs by induction are needed
- The Coq formalization is not documented
- It changes quite often
- Proving Frama-C lemmas is not for the casual Coq user

Can we get rid of Coq?

Yes (mostly)

Proofs by Induction without Coq?

The proof of a loop invariant is a proof by induction

- establishment:
  - prove the invariant holds when the loop starts
- preservation:
  - assume the invariant holds before the loop body
  - prove that the execution of the body implies the invariant

Main Idea: use lemma functions

- State lemmas as contracts of additional C functions (called lemma functions)
- Write a loop and loop annotations to prove the contract

Example: the split lemma – statement

```
/*@
  requires Linked : linked_n (root, array, index, n + k, bound);
  requires ValidArray : \valid( array + (0 .. MAX_SIZE - 1) );
  requires 0 <= index ;
  requires 0 < n && k >= 0 ;
  requires b0 == array[index + n - 1]->next ;

  assigns \nothing ;

  ensures linked_n(root, array, index, n, b0) ;
  ensures linked_n(b0, array, index + n, k, bound);
*/
void linked_n_split_segment (struct list * root, struct list * bound,
                             struct list ** array, int index, int n, struct list * b0, int k);
```

Example: the split lemma – proof

```
void linked_n_split_segment (struct list * root, struct list * bound,
                             struct list ** array, int index, int n, struct list * b0, int k){
  //@ ghost linked_n_valid_range(root, bound, array, index, n+k);
  //@ ghost linked_n_next_of_all_indexes(root, bound, array, index, n+k);

  struct list * sep = array[index + n + k - 1] ;

  /*@
    loop invariant n <= i <= n + k ;
    loop invariant sep == array [ index + i - 1 ] ;
    loop invariant linked_n(root, array, index, i, sep->next) ;
    loop invariant linked_n(sep->next, array, index + i, (n + k) - i, bound) ;
    loop assigns i, sep ;
    loop variant i ;
  */
  for(int i = n + k ; i > n ; --i){
    //@ ghost linked_n_before_last(root, sep->next, array, index, i) ;
    //@ ghost linked_n_next_of_all_indexes(root, sep, array, index, i-1);
    //@ assert linked_n(root, array, index, i - 1, sep) ;
```

Case studies

	Lemmas, incl. lemma functions & lemma macros	Gen. goals	Proved Coq	Lines of code		Execution time
				lemmas, incl. l.fun./macros	guiding annotations	
Case study (1). The memory management module `MEMB`						
Classic	15	134	15	33	20	47 sec.
Auto-active	3	217	1	25	25	19 sec.
Case study (2). The linked list module						
Classic	24	805	19	163	708	24 min.
Auto-active	17	1631	1	366	629	21 min.
Case study (3). *ACSL by Example*, v. 17.2.0						
Classic	87	1398	40	594	485	92 min.
Auto-active	53	1790	0	670	611	78 min.

CoMeMoV: a Collaborative Research Project funded by ANR

CoMeMoV = Collaborative Memory Models for formal Verification

https://comemov.github.io

Memory model?

In the WP plugin

- Translation to VC written in a first-order logic $\mathcal{L}$
- Memory model: maps any heap C-value at a given program point to some variable or term in the logical $\mathcal{L}$ language.

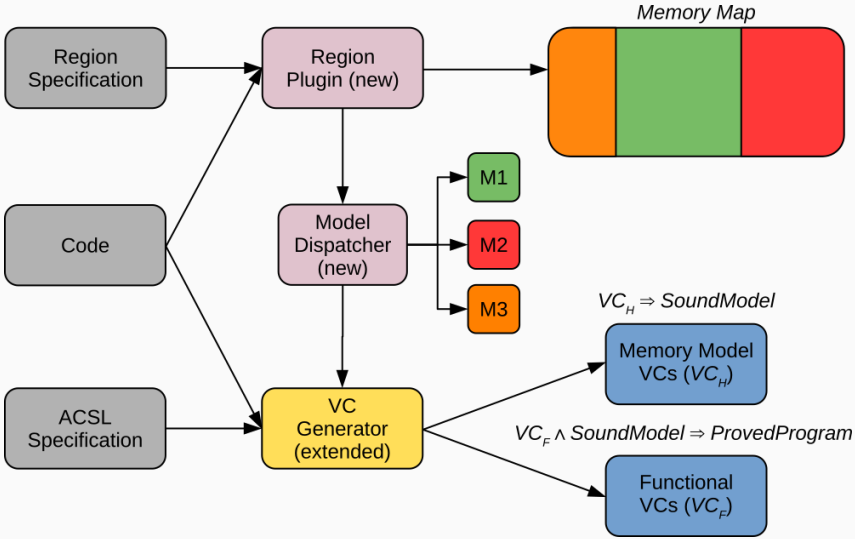
The Hoare memory model

- Local or global C variable $\rightarrow$ $\mathcal{L}$ variable(s)
- Memory reads or writes through pointer values: not handled

The typed memory model

- Heap represented by three memory variables which hold arrays of values indexed by addresses: holding respectively arrays of integers, floats and addresses
- Addresses represented by native records (base, offset): array theory and record theory work well together in SMT solvers
- Limitations, for e.g. heterogeneous casts, union types, ...
  $\Rightarrow$ byte model

CoMeMoV Objective



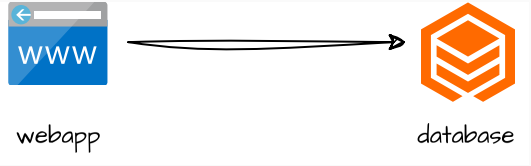
Formalization of Configuration Languages

For-CoaLa: a Collaborative Research Project funded by ANR

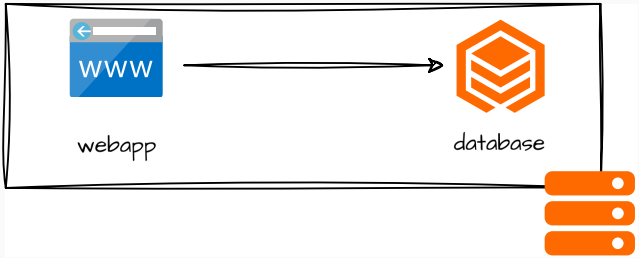
Building formally verified configuration languages!

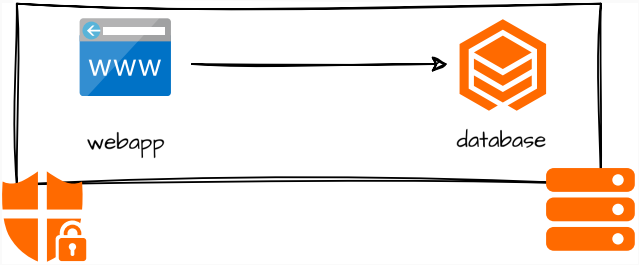
PI: Hélène Coullon

https://for-coala.github.io/about/

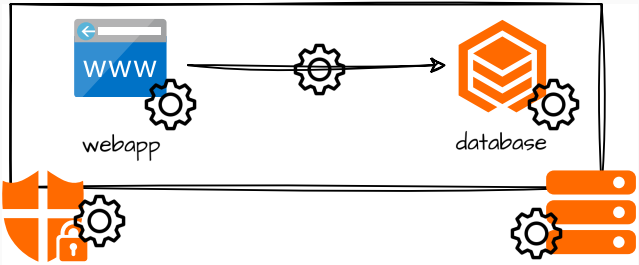

www
webapp
database

For-CoaLa: Infrastructure-as-Code & Configuration Management

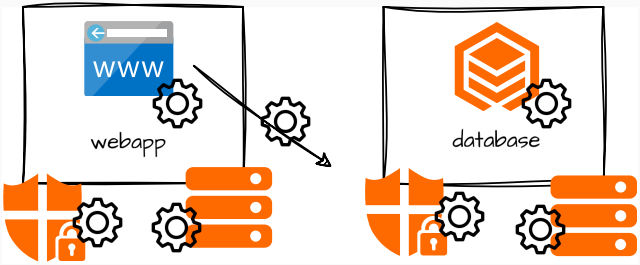


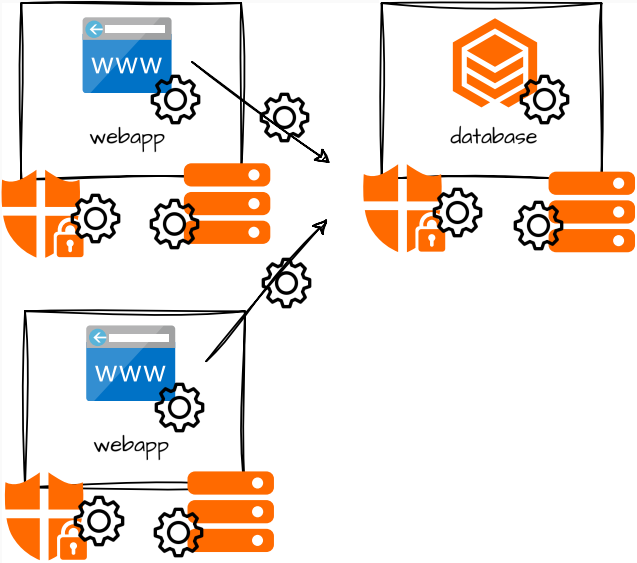

www
webapp
database

For-CoaLa: Infrastructure-as-Code & Configuration Management

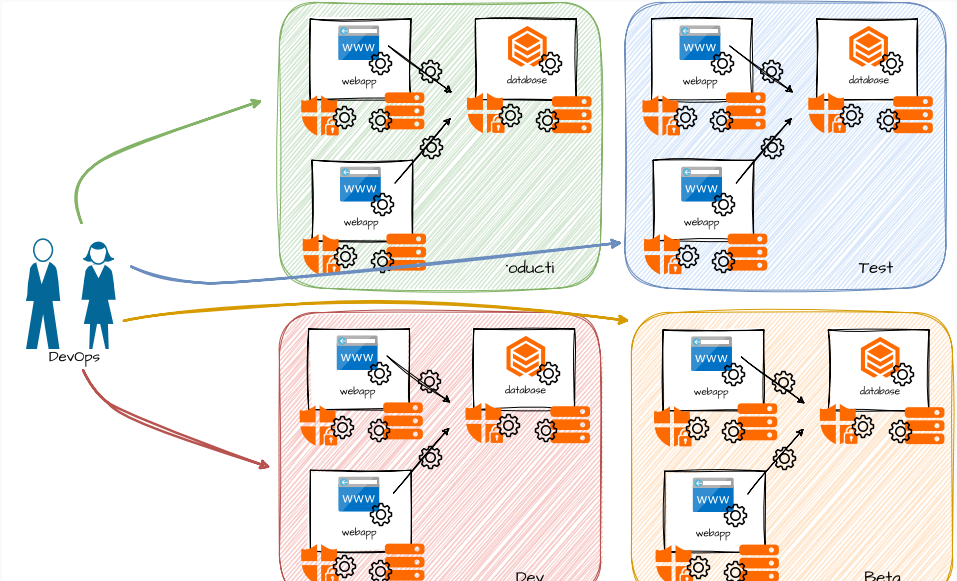


For-CoaLa: Infrastructure-as-Code & Configuration Management

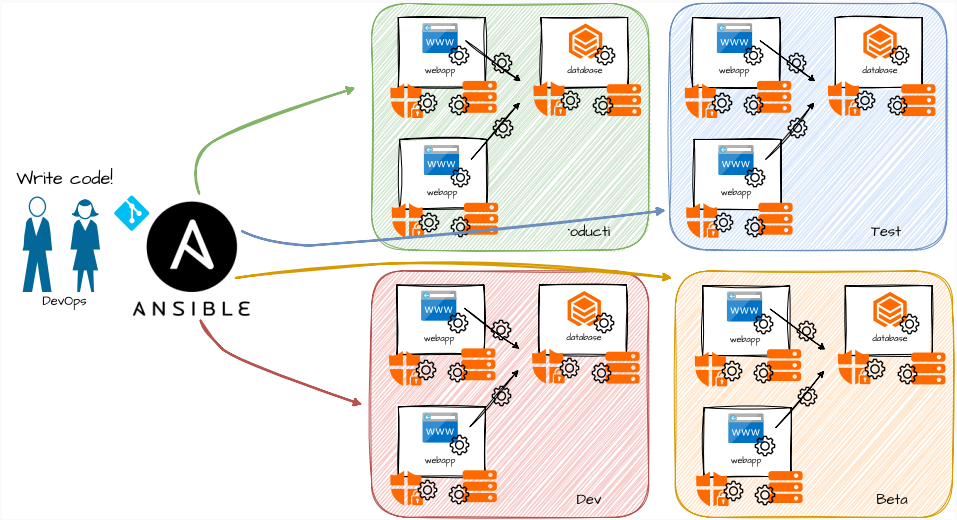



www
webapp
database
www
webapp

For-CoaLa: Infrastructure-as-Code & Configuration Management

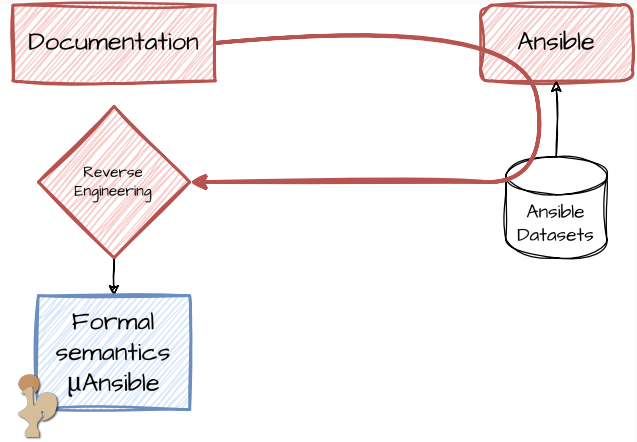


For-CoaLa: Infrastructure-as-Code & Configuration Management

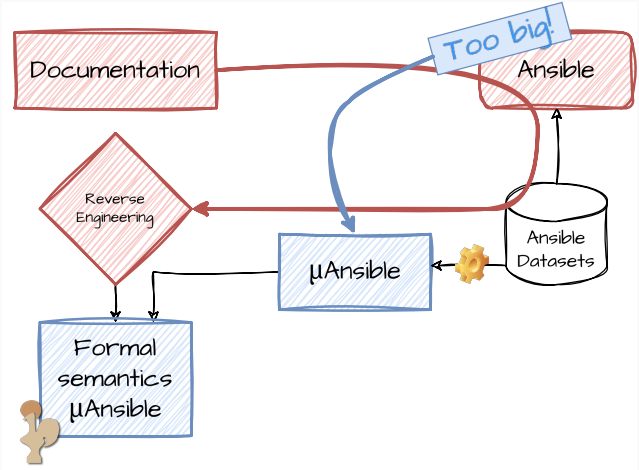


Configuration management is a critical operation for which languages have to be safe!

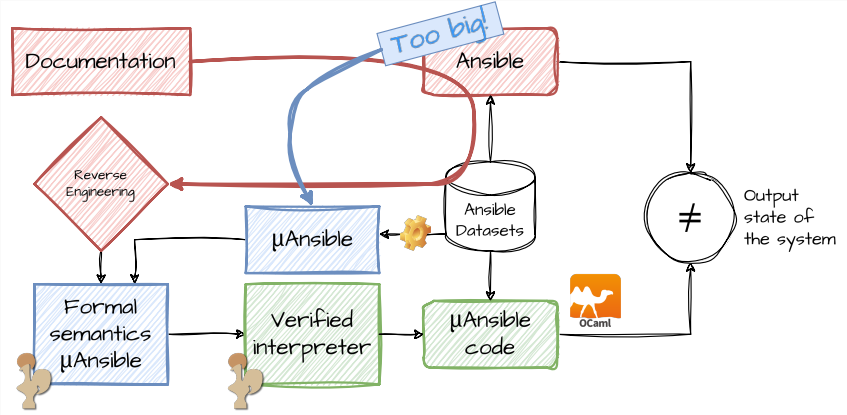
For-CoaLa: Main Objective



For-CoaLa: Main Objective



For-CoaLa: Main Objective



Thank You!

References

[1] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993532.

[2] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal Verification of a JavaCard Virtual Machine with Frama-C. In *Formal Methods (FM 2021)*, volume 13047 of *LNCS*, pages 427–444, Cham, 2021. Springer. ISBN 978-3-030-90870-6. doi:10.1007/978-3-030-90870-6_23.

[3] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27 (3):573–609, 2015. doi:10.1007/s00165-014-0326-7.

[4] Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. Towards Full Proof Automation in Frama-C using Auto-Active Verification. In *Proc. of the 11th NASA Formal Methods Symposium (NFM 2019)*, volume 11460 of *LNCS*, pages 88–105. Springer, 2019. doi:10.1007/978-3-030-20652-9_6.

[5] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for Lists: A Critical Module of Contiki Verified in Frama-C. In *Proc. of the 10th NASA Formal Methods Symposium (NFM 2018)*, volume 10811 of *LNCS*, pages 37–53. Springer, 2018.

[6] Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Logic against ghosts: Comparison of two proof approaches for a list module. In *Proc. of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2019)*, pages 2186–2195. ACM, 2019. doi:10.1145/3297280.3297495.

[7] Frédéric Loulergue, Allan Blanchard, and Nikolai Kosmatov. Ghosts for lists: from axiomatic to executable specifications. In *Proc. of the 12th International Conference on Tests and Proofs (TAP 2018)*, volume 10889 of *LNCS*, pages 177–184. Springer, 2018. doi:10.1007/978-3-319-92994-1_11.

[8] Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. A memory allocation module of Contiki formally verified with Frama-C. A case study. In *Proc. of the 11th International Conference on Risks and Security of Internet and Systems (CRiSIS 2016)*, volume 10158 of *LNCS*, pages 114–120. Springer, 2016.

[9] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

[10] Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. Component-based distributed software reconfiguration:a verification-oriented survey. *ACM Comput. Surv.*, 56(1):2:1–2:37, 2024. doi:10.1145/3595376.

[11] Farid Arfi, Hélène Coullon, Frédéric Loulergue, Jolan Philippe, and Simon Robillard. An overview of the decentralized reconfiguration language concerto-d through its maude formalization. In *7th Interaction and Concurrency Experience (ICE)*, volume 414 of *Electronic Proceedings in Theoretical Computer Science*, page 21–38. Open Publishing Association, December 2024. doi:10.4204/eptcs.414.2.