Ghosts for Lists: from Axiomatic to Executable Specifications

Frédéric Loulergue Allan Blanchard Nikolai Kosmatov – June 29, 2018 @ Tests & Proofs 2018

Contents

inventors for the digital world

01

Introduction

The Vessedia project

EU H2020 Vessedia project

- aims at making formal methods more usable in the context of the IoT
- comprises use-cases to evaluate the efficiency of the developed tools and methods
- `https://vessedia.eu`

Contiki-OS

- one of the use-cases targeted in Vessedia
- a lightweight OS for IoT

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.

Contiki and Formal Verification

- When started in 2003, no particular attention to security
- Later, communication security was added at different layers, via standard protocols such as IPsec or DTLS
- Security of the software itself did not receive much attention
- Continuous integration system does not include formal verification
  - and unit tests are under-represented

Today's talk: the list module of Contiki

- a critical component of the core part of Contiki
- many client modules in the whole OS
- verification performed with Frama-C

Frama-C at a glance

frama C

Software Analyzers

- A Framework for Modular Analysis of C code
- Developed at CEA List
- Released under LGPL license
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy
- `http://frama-c.com/` [Kirchner et al. FAC 2015]

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
  - Correctness of the specification is crucial
  - Attacks can exploit every single flaw ⇒ Complete proof is required!
- `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: `\valid(p)`, `\valid(p+0..2)`, `\separated(p+0..2,q+0..5)`, `\block_length(p)`

Plugin Frama-C/WP

WP: A plugin for deductive verification

- Based on Weakest Precondition calculus [Dijkstra, 1976]
- Goal: Prove that a given program respects its specification
- Requires formal specification
- Capable to formally prove that
  - each program function always respects its contract
  - each function call always respects the expected conditions on its inputs
  - each function call always gives enough guarantees to ensure the caller's contract
  - common security related errors (e.g. buffer overflows) can never occur

Plugin Frama-C/E-ACSL

E-ACSL: A plugin for dynamic verification

- Primary goal: runtime assertion checking
- Tranlate C + ACSL into C
- Violated assertion $\Rightarrow$ generated program fails at runtime
- Preserves the semantics if all assertions are satisfied
- A *executable* subset of ACSL:
  - bounded quantification
  - finite ranges and set comprehensions
  - no inductive predicate
  - no axiomatic definitions
  - Not yet supported:
    - predicate definition
    - logical function definition
    - ...

02

Ghosts for lists

The LIST module - Overview

Provides a generic 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

Some special features

- no dynamic allocation
- does not allow cycles
- maintain item unicity

The LIST module - 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 (list_init, list_length, list_head, list_tail, list_item_next)

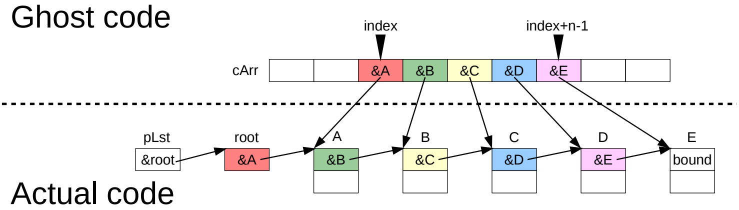
Update list beginning (list_pop, list_push)

Update list end (list_chop, list_add)

Update list anywhere (list_remove, list_insert)

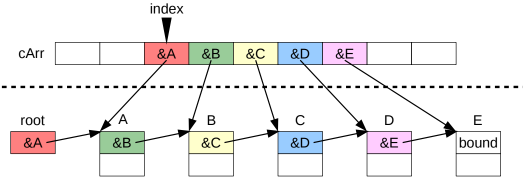
Formalization approach - Overview

We maintain a ghost array that stores the addresses of the different list elements.



Formalization approach - Induction

Ghost code

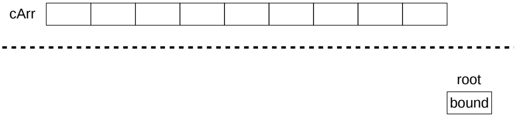


Actual code

```
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);
}
```

Formalization approach - Base case

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);
// ...
}
```


inventors for the digital world

Formalization approach - Advantages

- As long as we maintain the `linked_n` invariant,
  we can easily reason about the content of the list:

```
predicate unchanged{L1,L2}(struct list **array, int idx, int sz)=
  \forall integer i ; idx <= i < idx+sz ==>
    \at(array[i]->next, L1) == \at(array[i]->next, L2);
```

- While we have to update the array accordingly when the list is modified

- Set of lemmas (proved in Coq) to leverage automated verification

inventors for the digital world

Results

- Written specification and ghost code
  - > 46 lines for ghost functions
  - > 500 lines for contracts
  - > 240 lines for logic definitions and lemmas
  - > 650 lines of other annotations

- It generates 798 proof obligations
  - > 772 (96.7%) are automatically discharged by SMT solvers
  - > 24 are lemmas proved with Coq
  - > 2 assertions proved with Coq
  - > 2 assertions proved using TIP

- Bug found
- More details: NFM'18 doi:10.1007/978-3-319-77935-5_3
- Problem: not executable

03

Executable Specifications

Main Idea

- Remove ACSL features that are not supported by E-ACSL
- Replace it with semantically equivalent (in principle) supported ones
- Workaround for features not completely supported

Constraints for Execution

E-ACSL subset of ACSL

- No inductive predicates: `linked_N` is inductive
- No axiomatic function: `index_of` is axiomatic

E-ACSL subset not supported and workarounds

- Non inductive predicates: inlining is a workaround
- Functions:
  - inlining is Ok for non recursive functions
  - C assertions added in the code for recursive functions as a workaround

Main Idea in Practice

- Replace inductive predicate by:
  - a non inductive predicate
  - using a recursive logical function
- Replace axiomatic functions by logical functions
- Write a non logical C function expected to be equivalent
- Prove the equivalence with non logical C functions
- Inline the non inductive predicate
- hand coded calls to the C functions

Example: linked_N

Executable predicate and recursive function:

```
logic boolean array_view(struct list *root, struct list **cArr,
                         ℤ idx, ℤ sz, struct list *bound) =
  (sz==0)? root==bound : (root==cArr[idx] ∧
    array_view(root->next, cArr, idx+1, sz-1, bound));

predicate linked_exec{L}(struct list *root,
                         struct list **cArr, ℤ idx,
                         ℤ sz, struct list *bound) =
  0 ≤ sz ∧ 0 ≤ idx ∧ idx + sz ≤ MAX_SZ ∧
  (∀ ℤ k; idx ≤ k < idx + sz ⇒ \valid(cArr[k])) ∧
  array_view(root, cArr, idx, sz, bound) == \true;
```

Equivalence lemma:

```
∀ struct list *root, struct list **a, struct list *b, ℤ idx, ℤ sz;
linked_n(root, a, idx, sz, b)⟺linked_exec(root, a, idx, sz, b);
```

04

Conclusion

Let's sum up!

We presented how to work with axiomatic and executable specifications in the context of the verification of the list module of Contiki

Deductive verification

- based on a companion ghost array that tracks the status of the list
- comprises a set of lemmas that reduces the need for interactive proof
- allowed us to find and fix a bug in the module

Dynamic verification

- doable
- predicates should be inlined (in the specificaiton)
- runtime assertion checking calls by hand if un-inlinable functions are used

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
  - there are things to do inside the WP plugin
- separation was really hard to handle
  - (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract or more concrete specification
  - (Ongoing) using ACSL logic lists
  - (Ongoing) using an observation function
  - (Future) using contiguous partial functions

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
  - there are things to do inside the WP plugin
- separation was really hard to handle
  - (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract or more concrete specification
  - (Ongoing) using ACSL logic lists
  - (Ongoing) using an observation function
  - (Future) using contiguous partial functions

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
  - there are things to do inside the WP plugin
- separation was really hard to handle
  - (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract or more concrete specification
  - (Ongoing) using ACSL logic lists
  - (Ongoing) using an observation function
  - (Future) using contiguous partial functions

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
  - there are things to do inside the WP plugin
- separation was really hard to handle
  - (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract or more concrete specification
  - (Ongoing) using ACSL logic lists
  - (Ongoing) using an observation function
  - (Future) using contiguous partial functions

Ongoing & Future work

The verification of the list module can be improved

- assigns clauses are not precise enough
  - there are things to do inside the WP plugin
- separation was really hard to handle
  - (Ongoing) we should not have to deal with ghost separation
- one would prefer a more abstract or more concrete specification
  - (Ongoing) using ACSL logic lists
  - (Ongoing) using an observation function
  - (Future) using contiguous partial functions

Thank you!

Thank you!