UNIVERSIDADE
BEIRA INTERIOR

Certified Programming in the heavy presence of pointers

The case for Union-Find

Jean-Christophe Filliâtre, Mário Pereira, Simão Melo de Sousa

A contextual Introduction

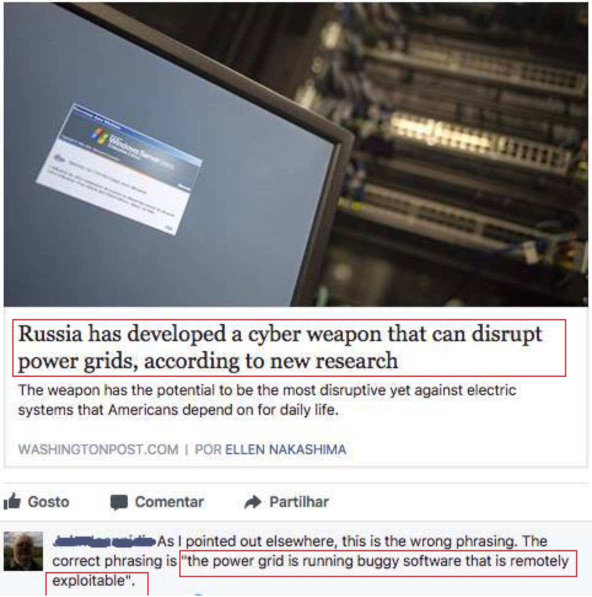
a bug is a disaster waiting to happen

security perspective: attack entry, security breach, information leakage, etc.

critical systems perspective: failure, dammage, (mission, life, etc.) loss, etc.

business perspective: disturbance, costs, loss of trust/business, etc.

claim.... as seen on a *reliable* social medium



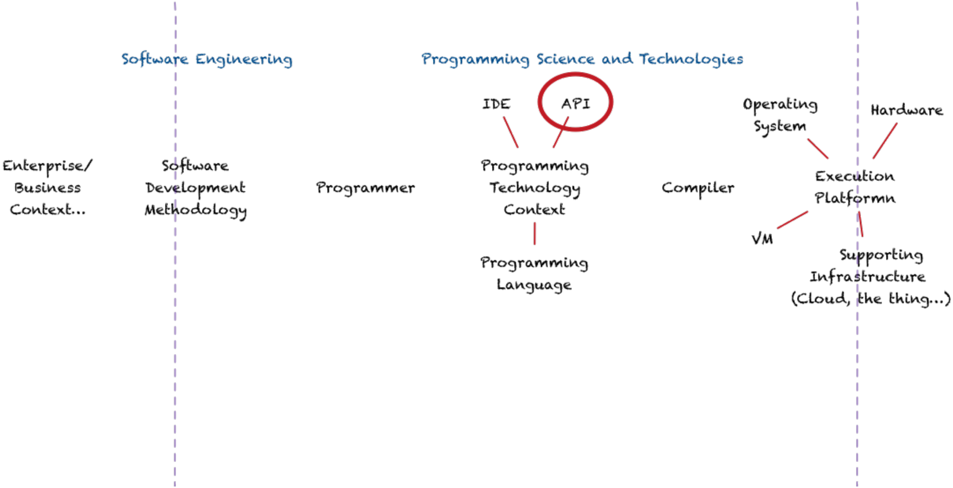
Russia has developed a cyber weapon that can disrupt power grids, according to new research

The weapon has the potential to be the most disruptive yet against electric systems that Americans depend on for daily life.

WASHINGTONPOST.COM | POR ELLEN NAKASHIMA

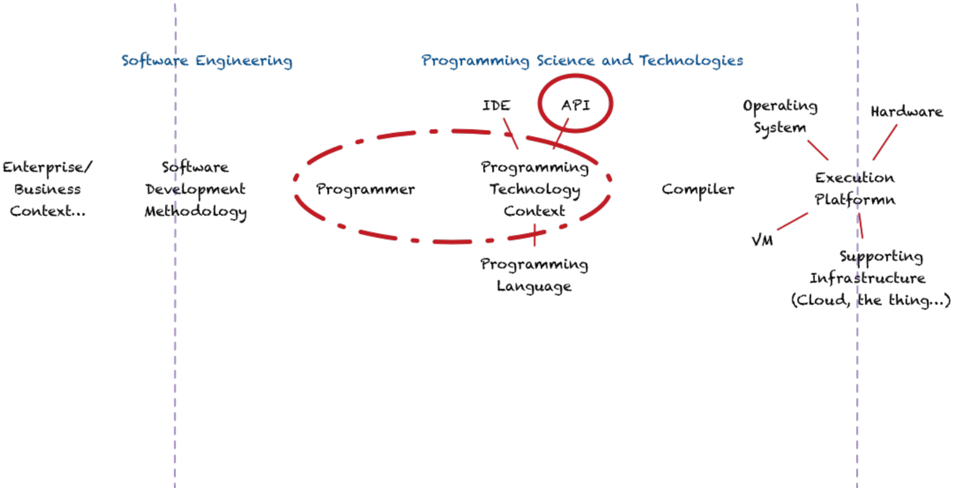
[redacted] As I pointed out elsewhere, this is the wrong phrasing. The correct phrasing is "the power grid is running buggy software that is remotely exploitable".

in this talk



the software lifecycle is complex

but first:


Software Engineering
Programming Science and Technologies
IDE
API
Operating System
Hardware
Enterprise/ Business Context...
Software Development Methodology
Programmer
Programming Technology Context
Compiler
Execution Platformn
VM
Programming Language
Supporting Infrastructure (Cloud, the thing...)

Some facts about programming activities

programming environments are always the **result of design choices** (made by its creators)

there is no known silver bullet, then it's always a (maybe outdated) **compromise** (security, efficiency, high level/low level, automatic/manual memory management etc.)

Some facts about programming activities

- **code reuse** (central to the OO paradigm success), boilerplates/ code recipes, *stackoverflow* style of programming, the `import everything` paradigm, etc.

- relation between the programmer and its programming environment:

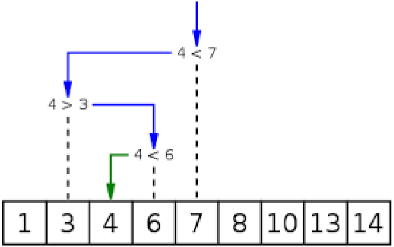
  from the programming environment perspective, it is **assumed** that the programmer **knows** the programming environment, **its limitations, its compromises** and subsequent coding practices, and **masters what he's doing**

  from the programmers perspective, it is **assumed** that the offered programming facilities **do exactly** what they're supposed to do, as efficient as possible, as simple as possible and do not introduce unspecified behavior

how do we know that these assumptions hold?

binary search: first publication in 1946 ...

... first **correct** publication in 1962



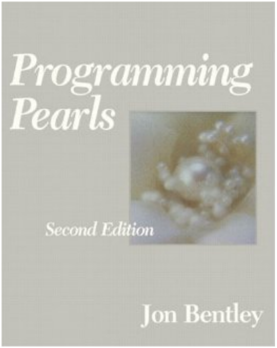
old but gold

Jon Bentley - Programming Pearls. 1986 (2^{nd} ed. 2000)

(column 4) - writing correct programs
The challenge of binary search

(as seen p.37)

Warning
Boring material ahead
skip to section 4,4
when drowsiness strikes



concise and crystal clear explanation of what is going on...

and yet...

old but gold

in 2006, an embarrassing bug was found in the standard library of Java... in the (binary) search methods

Joshua Bloch, Google Research Blog
"Nearly All Binary Searches and Mergesorts are Broken"

... been there for more than nine years!

old but gold

the bug :

```
...
  int mid = (low + high) / 2;
  int midVal = a[mid];
...
```

may surpass the `int` type range: *integer overflow*

and then cause an *array out of bound* error

possible solution:

```
   int mid = low + (high - low) / 2
```

lessons?

it's a good idea to have API/Standard Libraries done right...

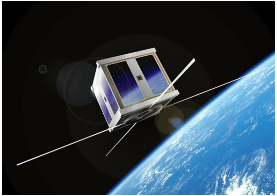
The Vocal Project

VOCAL – a **V**erified **OCA**ml **L**ibrary

a library of **correct-by-construction**, **efficient general-purpose** data structures and algorithms

- priority queues
- hash tables
- sequences
- sets/maps
- resizable arrays
- heap
- ...

- graph algorithms
- sorting
- searching
- string processing
- union-find
- ...



software that could benefit from such a library: Coq, Frama-C, Astrée, SPARK, Infer, Alt-Ergo, Cubicle, EasyCrypt, ProVerif, etc.

which, in turn, are used in avionics, defense, aerospacial, finance, security, hardware, etc.

(LRI/CNRS + Inria + Verimag + TrustInSoft + OCamlPro)

a specification language for OCaml

regular .mli files with

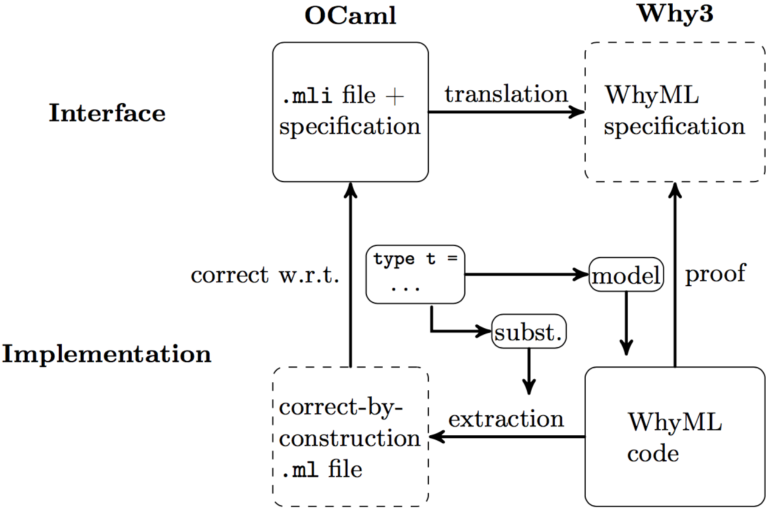
- formal specification in special comments (*à la* JML / ACSL)
- informal comments
- users can ignore formal specs
- simple, mostly first-order logic

regular .ml files

- no spec

three design workflows: why3, COQ, CFML+COQ

Vocal/Why3 workflow



OCaml

Why3

Interface

`.mli` file + specification

translation

WhyML specification

correct w.r.t.

`type t =`
`...`

model

proof

subst.

Implementation

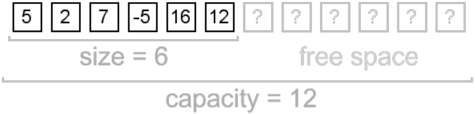
correct-by-construction `.ml` file

extraction

WhyML code

```
(** resizable arrays *)

type 'a t
(*@ ephemeral *)
(*@ field mutable view: 'a seq *)
(*@ invariant length view ≤ Sys.max_array_length *)
```

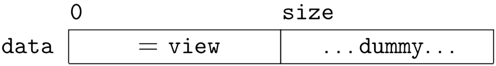


example — Vector.mli

```
val resize: 'a t → int → unit
(** [resize a n] sets the length of vector [a] to [n].
   The elements that are no longer part of the vector, if any,
   are internally replaced by the dummy value of vector [a],
   so that they can be garbage collected when possible.
   Raise [Invalid_argument]
            if [n < 0] or [n > Sys.max_array_length] *)
(*@ resize a n
      checks   0 ≤ n ≤  Sys.max_array_length
      modifies a
      ensures  length a.view = n
      ensures  forall i. 0 ≤ i < min (length (old a.view)) n →
                         a.view[i] = (old a.view)[i]  *)
```

WhyML code for `Vector`

```
type t 'a = {
               dummy: 'a;
        mutable size: int63;
        mutable data: array 'a;
  ghost mutable view: seq 'a;
}
invariant { length view = size}
invariant { forall i. 0 ≤ i < size → view[i] = data[i] }
invariant { 0 ≤ size ≤ length data ≤ max_array_length }
invariant { forall i. size ≤ i < length data → data[i] = dummy }
```



0

size

data

= view

...dummy...

WhyML code for Vector

```
type t 'a = {
               dummy: 'a;
        mutable size: int63;
        mutable data: array 'a;
  ghost mutable view: seq 'a;
}
invariant { length view = size}
invariant { forall i. 0 ≤ i < size → view[i] = data[i] }
invariant { 0 ≤ size ≤ length data ≤ max_array_length }
invariant { forall i. size ≤ i < length data → data[i] = dummy }

let resize (a: t 'a) (n: int63) : unit
  writes   { a.data, a.size, a.data.elts, a.view }
  ensures  { n = a.size }
  ensures  { forall i. 0 ≤ i < MinMax.min ((old a).size) n →
               a.view[i] = (old a).view[i] }
  raises   { Invalid_argument → not (0 ≤ n ≤ max_array_length) }
= if not (zero≤ n ≤max_array_length) then raise Invalid_argument;
  unsafe_resize a n
```

extracted code from Vector

```
type 'a t = {
  dummy: 'a;
  mutable size: int;
  mutable data: ('a array);
}

let resize (a: 'a t) (n: int) : unit =
  begin
    if not (0 <= n && n <= Sys.max_array_length)
     then raise (Invalid_argument );
    unsafe_resize a n
  end
```

driver to ocaml

```
module mach.int.Int63
  syntax type int63 "int"
  syntax val  ( + ) "%1 + %2"
  ...
end

module mach.array.Array63
  syntax type array "(%1 array)"
  syntax val  ([])   "Array.get %1 %2"
  ...
end
```

Deductive program verification in a picture

the state of the verified OCaml modules with Vocal/Why3

module	spec	code	#VCs	
`UnionFind`	74	176	135	union-find
`PairingHeap`	41	245	52	persistent priority queues
`ZipperList`	66	180	87	zipper data structure for lists
`Arrays`	37	121	77	binary search and binary sort
`Queue`	54	185	119	mutable queues
`Vector`	149	309	142	resizable arrays
`HashSet`	21	34	12	sets using hash tables
`MergeSort`	12	401	630	in-place mergesort of lists
`Dfs`	-	58	5	depth-first graph marking
`Schorr-Waite`	-	184	172	in-place graph marking

all the VCs were proved automatically!

… and with Vocal/COQ/CFML

module	tool	loc	Coq
`Listmap`	Coq	50	170
`HashTable`	CFML	150	750
`UnionFind` *	CFML	60	800
`IntervalMap`	CFML	WiP	WiP

(∗) including (amortized) computational complexity

Union-Find

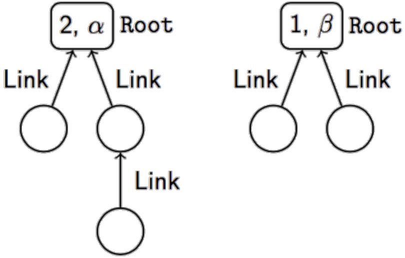
---

correct-by-construction implementation using Vocal/Why3

union-find data-structure

```
type 'a content =
  | Link of 'a content ref
  | Root of int * 'a

type 'a elem = 'a content ref
```



union-find in Why3 : first attempt

```
type content 'a =
  | Link (ref (content 'a))
  | Root int 'a
```

```
Error:
This field has non-pure type, it cannot be used in a recursive
type definition
```

the Why3 types and effect system:

mutability of **bounded depth** $\Longrightarrow$ **all** aliases must be known **statically**

solution : built-in memory model

embed a custom memory model into the why3's logic :

- a type for memory pointers
- operations for pointer allocation, read and write
- an association table from pointers to their values

examples : Frama-C, Dafny, VeriFast, VCC, CFML

here we use the ***component-as-array*** memory model design technique
[Burstall, 1972]

memory model for union-find (1/2)

```
type loc 'a

type content 'a =
  | Link (loc 'a)
  | Root int 'a

type memory 'a = {
  ghost mutable refs: loc 'a → option (content 'a);
}
```

here `None`/`Some` mean non-allocated/allocated

memory model for union-find (2/2)

```
val alloc (ghost mem: memory 'a) (v: content 'a) : loc 'a
  writes   { mem }
  ensures  { (old mem).refs result = None }
  ensures  { mem.refs = (old mem.refs)[result ← Some v] }

val set_ref (ghost mem: memory 'a) (l: loc 'a) (v: content 'a)
  requires { mem.refs l ≠ None }
  writes   { mem }
  ensures  { mem.refs = (old mem.refs)[l ← Some v] }
...
```

implementation: union-find data structure

```
type uf  'a = {
         memo: memory 'a;
 mutable dom : set (loc 'a);    (* which "pointers" are involved *)
 mutable rep : loc 'a → loc 'a;(* representative element *)
 mutable img : loc 'a → 'a;    (* representative element value *)
 mutable dst : loc 'a → int;   (* distance *)
 mutable maxd: int;            (* max value for dst *)
  }
  invariant { forall x. mem x dom → img x = img (rep x) }
  invariant { forall x. mem x dom → rep (rep x) = rep x }
  invariant { forall x. mem x dom → mem (rep x) dom }
  invariant { forall x y. mem x dom → mem y dom →
                  rep x = rep y → img x = img y }
  invariant { forall x y. mem x dom → rep x = y → mem y dom }
  invariant { forall x. mem x dom ↔ allocated memo x }
  ...
```

implementation: union-find data structure

```
type uf  'a = {
         memo: memory 'a;
 mutable dom : set (loc 'a);    (* which "pointers" are involved *)
 mutable rep : loc 'a → loc 'a;(* representative element *)
 mutable img : loc 'a → 'a;    (* representative element value *)
 mutable dst : loc 'a → int;   (* distance *)
 mutable maxd: int;            (* max value for dst *)
  } ...
  invariant { forall x. match memo.refs x with
               | Some (Link y)   → x ≠ y ∧ allocated memo y ∧
                                    rep x = rep y ∧ dst x < dst y
               | Some (Root r v) → img x = v ∧ rep x = x
               | None → true end }
  invariant { 0 ≤ maxd }
  invariant { forall x. mem x dom → dst x ≤ maxd }
  invariant { forall x. mem x dom → match memo.refs (rep x) with
               | Some (Root r _) → true
               | _               → false end }
```

implementation: union-find data operations

```
(* with path compression *)
let rec find (ghost uf: uf 'a) (x: loc 'a) : loc 'a
  requires { mem x uf.dom }
  writes   { uf.memo }
  variant  { uf.maxd - uf.dst x }
  ensures  { result = uf.rep x }
  ensures  { uf.dst result ≥ uf.dst x }
= match get_ref uf.memo x with
| Root _ _ → x
| Link y → let rx = find uf y in
            set_ref uf.memo x (Link rx);
            rx end
```

implementation: union-find data operations

```
let link (ghost uf: uf 'a)  (x y: loc 'a) : ghost loc 'a
 requires { mem x uf.dom }
 requires { x = uf.rep x }
 requires { mem y uf.dom }
 requires { y = uf.rep y }
 ensures  { (result = old (rep uf x)) || (result = old (rep uf y))}
 ensures  { forall z. mem z uf.dom →
     rep uf z = if old (equiv uf z x || equiv uf z y)
                then result
                else old (rep uf z) }
 ensures  { forall z. mem z uf.dom →
     img uf z = if old (equiv uf z x || equiv uf z y)
                then img uf result
                else old (img uf z) }
```

implementation: union-find data operations

```
let union (ghost uf: uf 'a) (x y: loc 'a) : ghost loc 'a
  requires { mem x uf.dom }
  requires { mem y uf.dom }
  ensures  { result = old (rep uf x) || result = old (rep uf y) }
  ensures  { forall z. mem z uf.dom →
    rep uf z = if old (equiv uf z x || equiv uf z y)
               then result
               else old (rep uf z) }
   ensures  { forall z. mem z uf.dom →
     img uf z = if old (equiv uf z x || equiv uf z y)
                then img uf result
                else old (img uf z) }
= let a = find uf x in
  let b = find uf y in
  link uf a b
```

OCaml driver for union-find

```
(* custom driver for UnionFind_impl, to map the custom memory
   model  to OCaml references. *)

module UnionFind_impl.Mem
  syntax type loc      "(%1 content) ref"
  syntax function Link "Link %1"
  syntax function Root "Root (%1, %2)"
  syntax val  (==)     "%1 == %2"
  syntax val  (!=)     "%1 != %2"
  syntax val  alloc    "ref %1"
  syntax val  get_ref  "!%1"
  syntax val  set_ref  "%1 := %2"
end

module UnionFind_impl.Impl
 prelude "type 'a content = Link of 'a content ref
                          | Root of int * 'a"
end
```

Overall result

module	spec	code	#VCs
`UnionFind`	74	176	135

all the VCs were proved automatically

conclusion

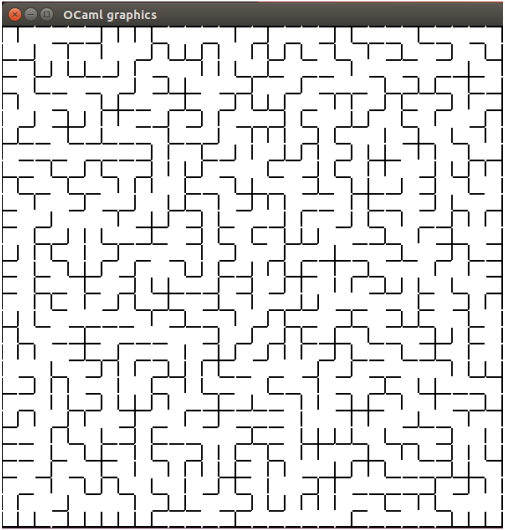
it works like a charm

how to generate a perfect maze of size $N \times N$?

use **union-find**!

it works like a charm

how to generate a perfect maze of size $N \times N$?



future work

obviously:

- more correct-by-construction data-structures and algorithms
- integration into critical software / client code

in the Vocal/CFML/COQ, the proof process is not automatic but it is possible to prove very subtle or complex properties, for instance about computational complexity

our plan to achieve this level of proof power can be divided in two points:

1. separation logic support in why3 (as a library), for a better systematic memory region / frame reasonning
2. integrate time credits techniques for checking computational complexity