

# GOSPEL

an interoperable specification language for OCaml

Jean-Christophe Filliâtre

Clément Pascutto

Nov 16, 2020



Tarides

VOCaL = a Verified OCaml Library

on-going ANR-funded project (2015–2020)

aims at building a formally-verified general-purpose OCaml library of data structures and algorithms

contributors: Paris-Saclay, Inria, Verimag, TrustInSoft, OCamlPro

see <https://vocal.lri.fr>

OCaml is used in software where high assurance is welcome

e.g., Alt-Ergo, Astrée, Coq, Cubicle, EasyCrypt, Frama-C, Infer, ProVerif, TIS analyzer, Why3, etc.

- there are bugs in libraries  
OCaml makes no exception, e.g., OCamlGraph
- light adoption of formal methods:  
start small with verified basic blocks

one contribution of VOCaL is a behavioral interface specification language for OCaml, called GOSPEL

(for Generic OCaml SPECification Language)

1. a tour of GOSPEL
2. prototype tools using GOSPEL

a behavioral interface specification language for OCaml

design choices:

- written in regular OCaml interfaces, within comments
- syntax close to that of OCaml
- using first-order logic
- not necessarily executable

a semantics in terms of Separation Logic [FM'19]

```
val mjrty: string array -> string
```



```
val mjrty: string array -> string
```

```
(** [mjrty a] returns the element of [a] with absolute  
majority, if any, or raises [Not_found] otherwise *)
```

```
val mjrty: string array -> string
```

```
(** [mjrty a] returns the element of [a] with absolute  
majority, if any, or raises [Not_found] otherwise *)
```

```
(*@ r = mjrty a  
  requires 1 <= length a  
  ensures numof a r > length a / 2  
  raises   Not_found ->  
           forall c. numof a c <= length a / 2    *)
```

- model fields

```
type 'a t  
(*@ mutable model view: 'a seq *)
```

- model fields
- effectful functions

```
val push: 'a t -> 'a -> unit
(*@ push a x
    modifies a
    ...
```

- model fields
- effectful functions
- ghost code

```
type 'a elem
val make: 'a -> 'a elem

val find: 'a elem -> 'a elem

val union: 'a elem -> 'a elem -> unit
```

- model fields
- effectful functions
- ghost code

```
(*@ type 'a uf ... *)  
(*@ val create: unit -> 'a uf *)  
type 'a elem  
val make: 'a -> 'a elem  
(*@ e = make [uf] v ... *)  
val find: 'a elem -> 'a elem  
(*@ r = find [uf] e ... *)  
val union: 'a elem -> 'a elem -> unit  
(*@ union [uf] e1 e2 ... *)
```

- model fields
- effectful functions
- ghost code
- specs uses mathematical integers

```
val f: int -> int
(*@ y = f x
   ensures 3 * y > 2 * x *)
```

- model fields
- effectful functions
- ghost code
- specs uses mathematical integers
- runtime checks
- higher-order functions



```
val binary_search:
  ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(*@ r = binary_search cmp a fromi toi v
   requires 0 <= fromi <= toi <= length a
   requires is_pre_order cmp
   requires forall i j.
     fromi <= i <= j < toi -> cmp a[i] a[j] <= 0
   ...
```

no runtime assertion checking by default

preconditions are supposed to be verified at call site

```
val binary_search:
  ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(*@ r = binary_search cmp a fromi toi v
   checks    0 <= fromi <= toi <= length a
   requires  is_pre_order cmp
   requires  forall i j.
               fromi <= i <= j < toi -> cmp a[i] a[j] <= 0
   ...
```

**checks** means a runtime check  
(with an exception being raised if not met)

```
val binary_search:
  ('a -> 'a -> int) -> 'a array -> int -> int -> 'a -> int
(*@ r = binary_search cmp a fromi toi v
   checks    0 <= fromi <= toi <= length a
   requires  is_pre_order cmp
   requires  forall i j.
               fromi <= i <= j < toi -> cmp a[i] a[j] <= 0
   ...
```

this contract assumes function `cmp` to be **pure**

sometimes we can't make such an assumption, e.g.

```
val List.iter: ('a -> unit) -> 'a list -> unit
```

for such functions, GOSPEL features an **equivalent** clause, e.g.

```
val iter: ('a -> unit) -> 'a vector -> unit
(*@ iter f v
  equivalent
  "for i = 0 to length v - 1 do f (get v i) done" *)
```

this is a piece of code, not a logical formula!

2

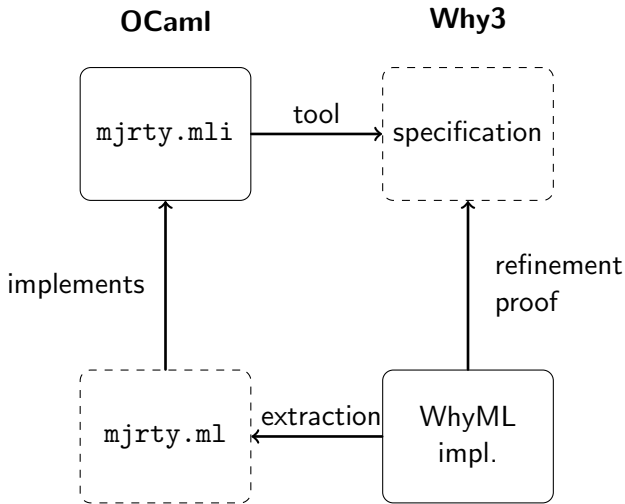
---

prototype tools using GOSPEL

- deductive program verification
  - ▶ a Why3 plugin that reads GOSPEL specs
  - ▶ Cameleer (Mário Pereira, Univ. NOVA de Lisboa)
- runtime assertion checking
  - ▶ PhD of Clément

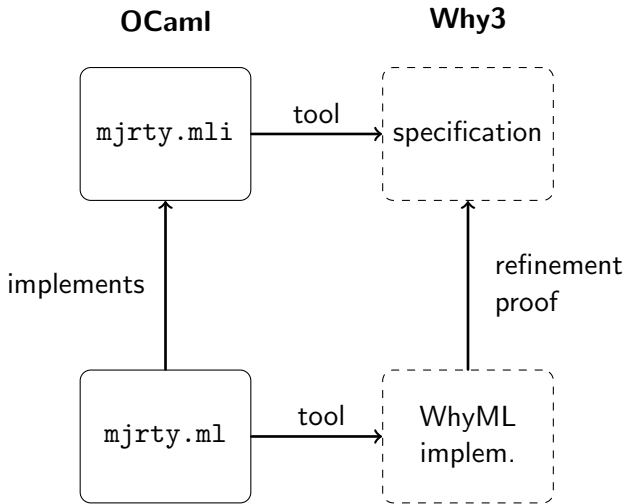
all this is work in progress

## workflow when using the Why3 plugin

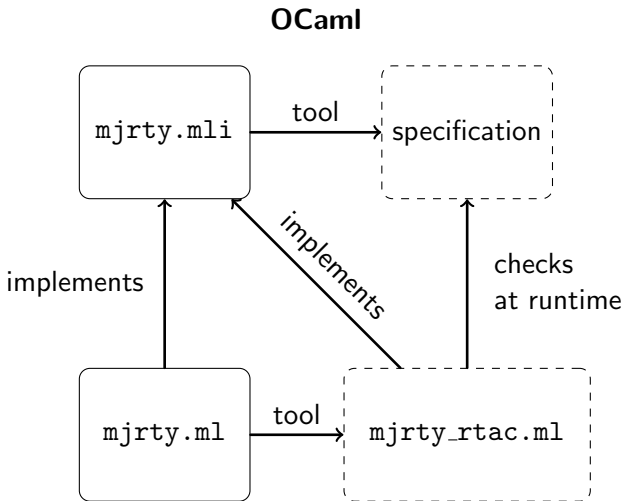




## workflow when using Cameleer



# workflow when using Runtime Assertion Checking



## VOCaL — a Verified OCaml Library

module	loc	tool
HashTable	150	CFML
UnionFind	60	Why3,CFML
Vector	150	Why3
PriorityQueue	81	Why3
PairingHeap	42	Why3
ZipperList	58	Why3
Lists	50	Coq
Arrays	63	Why3
Mjrtty	26	Why3
RingBuffer	34	Why3
CountingSort	24	Why3

including iterators  
amortized complexity

used in Why3 code

includes OCaml's mergesort

## some contributions of the project

- how to avoid proving the absence of integer overflows  
[F, Paskevich, with Clochard, VSTTE'15]
- time credits and amortized complexity  
[Guéneau, Charguéraud, Pottier, ITP'15, ESOP'18]
- a modular way to reason about iteration  
[F, Pereira, NFM'16] [Pottier, CPP'17]
- temporary read-only permissions for separation logic  
[Pottier, Charguéraud, ESOP'17]
- simulating induction-recursion for partial algorithms  
[Monin, with Larchey-Wendling, TYPES'18]

<https://vocal.lri.fr/>

- publications

<https://github.com/vocal-project/vocal/>

- GOSPEL implementation
- GOSPEL to Why3 plug-in
- the VOCaL library