Frama-C and Why3: going way back — and forward, too

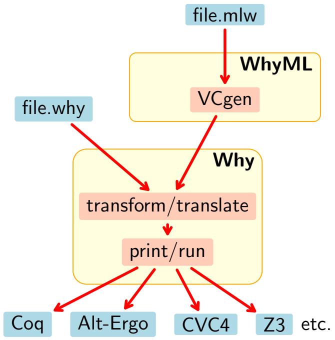
Andrei Paskevich

LRI, Université Paris-Sud — Toccata, Inria

Frama-C day, June 20, 2015

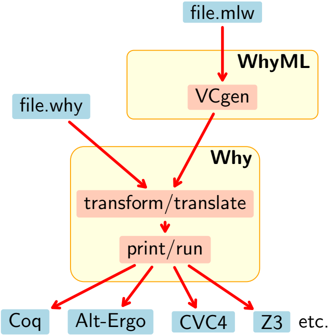
1. What is Why3?

Why3 in a nutshell



WhyML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants



Why3 in a nutshell

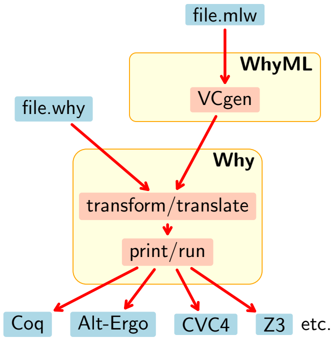
WhyML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WhyML, a specification language

- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)



WhyML, a programming language

- type polymorphism • variants
- limited support for higher order
- pattern matching • exceptions
- ghost code and ghost data (CAV 2014)
- mutable data with controlled aliasing
- contracts • loop and type invariants

WhyML, a specification language

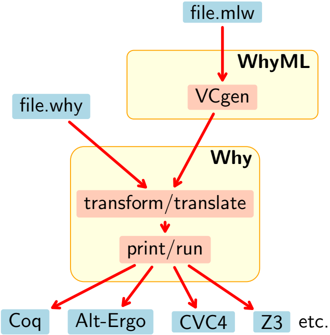
- polymorphic & algebraic types
- limited support for higher order
- inductive predicates

(FroCos 2011) (CADE 2013)

Why3, a program verification tool

- VC generation using WP or fast WP
- 70+ VC *transformations* ($\approx$ tactics)
- support for 25+ ATP and ITP systems

(Boogie 2011) (ESOP 2013) (VSTTE 2013)



Why3 out of a nutshell

three different ways of using Why3

as a logical language

- a front-end to many theorem provers: Frama-C/WP

as a programming language to prove algorithms

- 134 examples in our gallery
- AVL trees, binary heaps, a simple compiler, a tableaux-based theorem prover, etc.

as an intermediate verification language

- Java programs: Krakatoa (Marché Paulin Urbain)
- C programs: Frama-C/Jessie (Marché Moy)
- Ada programs: SPARK 2014 (AdaCore)
- probabilistic programs: EasyCrypt (Barthe et al.)

Example: maximum subarray problem

```
let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
```

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                     *)
(*  ..............................|### cur ####                     *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in




  for i = 0 to length a - 1 do





    cur += a[i];
    if !cur < 0 then       cur := 0;
    if !cur > !max then       max := !cur
  done;
  !max
```

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                     *)
(*  ..............................|### cur ####                     *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in


  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }


    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then       max := !cur
  done;
  !max
```

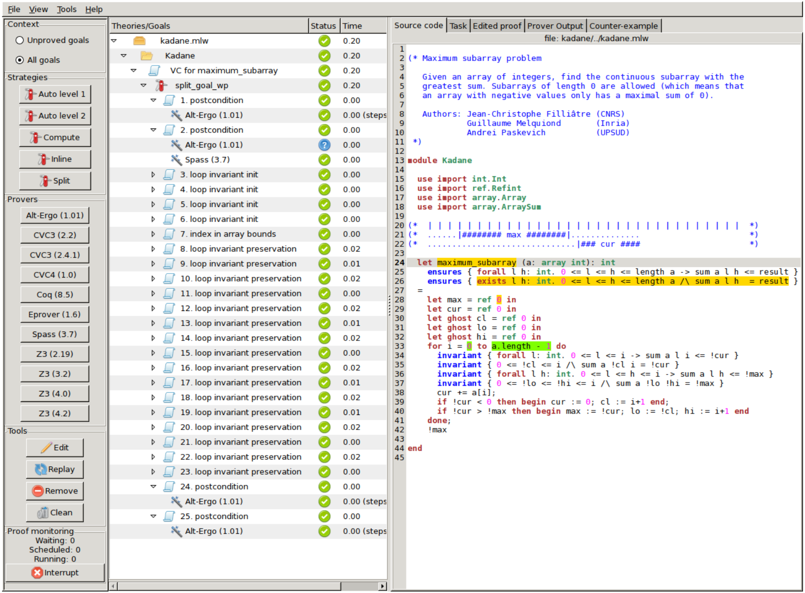
Kadane's algorithm

```
(*  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  *)
(*  ......|######## max ########|..............                      *)
(*  ...............................|### cur ####                     *)

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Kadane's algorithm

```
use import ref.Refint
use import array.Array
use import array.ArraySum

let maximum_subarray (a: array int): int
  ensures { forall l h: int. 0 <= l <= h <= length a -> sum a l h <= result }
  ensures { exists l h: int. 0 <= l <= h <= length a /\ sum a l h  = result }
=
  let max = ref 0 in
  let cur = ref 0 in
  let ghost cl = ref 0 in
  let ghost lo = ref 0 in
  let ghost hi = ref 0 in
  for i = 0 to length a - 1 do
    invariant { forall l: int. 0 <= l <= i -> sum a l i <= !cur }
    invariant { 0 <= !cl <= i /\ sum a !cl i = !cur }
    invariant { forall l h: int. 0 <= l <= h <= i -> sum a l h <= !max }
    invariant { 0 <= !lo <= !hi <= i /\ sum a !lo !hi = !max }
    cur += a[i];
    if !cur < 0 then begin cur := 0; cl := i+1 end;
    if !cur > !max then begin max := !cur; lo := !cl; hi := i+1 end
  done;
  !max
```

Why3 proof session



A long time ago at CEA Saclay

CAVEAT a static analyzer for C (Baudin Pacalet Raguideau Schoen Williams, DSN 2002)

- Hoare-style memory model (no aliases)
- custom theorem prover

A long time ago at CEA Saclay

CAVEAT a static analyzer for C (Baudin Pacalet Raguideau Schoen Williams, DSN 2002)

- Hoare-style memory model (no aliases)
- custom theorem prover

Meanwhile at LRI / Inria Futurs

Why a tool for deductive program verification (Filliâtre, 2002)

- handles a subset of ML and C, also Java (via Krakatoa)
- Hoare-style memory model (no aliases)
- multiple third-party theorem provers: Coq, PVS, Simplify, etc.

Family tree

A long time ago at CEA Saclay

CAVEAT a static analyzer for C (Baudin Pacalet Raguideau Schoen Williams, DSN 2002)

- Hoare-style memory model (no aliases)
- custom theorem prover

Meanwhile at LRI / Inria Futurs

Why a tool for deductive program verification (Filliâtre, 2002)

- handles a subset of ML and C, also Java (via Krakatoa)
- Hoare-style memory model (no aliases)
- multiple third-party theorem provers: Coq, PVS, Simplify, etc.

Caduceus a multi-prover verifier for C programs (Filliâtre Marché, ICFEM 2004)

- component-as-array memory model (no pointer cast)
- Why for VC generation
- Coq and Simplify as the back-end provers

Family tree, cont.

And then they have met

CAT ANR project, 2006–2009

- led by CEA List (B. Monate)
- academic partners: ProVal (Inria/LRI) and Lande (Inria)
- industrial partners: Airbus France, Dassault Aviation, Siemens

And then they have met

CAT ANR project, 2006–2009

- led by CEA List (B. Monate)
- academic partners: ProVal (Inria/LRI) and Lande (Inria)
- industrial partners: Airbus France, Dassault Aviation, Siemens

Frama-C a C Analysis Toolbox (2008)

- ACSL specification language (Baudin Filliâtre Marché Monate Moy Prevosto)
  - inspired by Caduceus
  - first-order logic with total functions and unbounded quantification
- various analyzers implemented as plug-ins

Value abstract interpretation
Jessie deductive verification via Why
WP deductive verification with dedicated VCgen (2010)

Family tree, cont.

And then they have met

CAT ANR project, 2006–2009

- led by CEA List (B. Monate)
- academic partners: ProVal (Inria/LRI) and Lande (Inria)
- industrial partners: Airbus France, Dassault Aviation, Siemens

Frama-C a C Analysis Toolbox (2008)

- ACSL specification language (Baudin Filliâtre Marché Monate Moy Prevosto)
  - inspired by Caduceus
  - first-order logic with total functions and unbounded quantification
- various analyzers implemented as plug-ins

Value	abstract interpretation
Jessie	deductive verification via Why
WP	deductive verification with dedicated VCgen (2010)

Meanwhile at LRI / Inria Saclay

Why3 full redesign of Why (Bobot Filliâtre Kanig Marché Melquiond Paskevich, 2010)

2. A case for a rich(er) specification language

Logical foundations

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

Admits many useful extensions without sacrificing tractability

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

Admits many useful extensions without sacrificing tractability

- polymorphic types

```
type seq 'a

function length (s: seq 'a) : int
function get (s: seq 'a) (i: int) : 'a
function cons (v: 'a) (s: seq 'a) : seq 'a

axiom cons_length : forall v: 'a, s: seq 'a.
     length (cons v s) = 1 + length s

axiom cons_get : forall v: 'a, s: seq 'a, i: int.
     0 <= i <= length s ->
        get (cons x s) i = if i = 0 then x else get s (i-1)
```

  - supported natively in Alt-Ergo, support in CVC4 may come soon
  - requires non-trivial encoding for many-sorted / one-sorted provers

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

Admits many useful extensions without sacrificing tractability

- polymorphic types
- algebraic types and pattern matching

```
type tree 'a = Empty | Node (tree 'a) 'a (tree 'a)

function height (t: tree 'a) : int =
  match t with
  | Empty -> 0
  | Node l _ r -> 1 + max (height l) (height r)
  end
```

  - type definitions supported in Alt-Ergo (non-recursive), Z3, CVC4
  - pattern matching can be easily encoded

Logical foundations

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

Admits many useful extensions without sacrificing tractability

- polymorphic types
- algebraic types and pattern matching
- inductive predicates

```
type vertex
predicate edge vertex vertex

inductive path vertex (list vertex) vertex =
  | Path_empty: forall x: vertex. path x Nil x
  | Path_cons: forall x y z: vertex, l: list vertex.
      edge x y -> path y l z -> path x (Cons x l) z
```

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

Admits many useful extensions without sacrificing tractability

- polymorphic types
- algebraic types and pattern matching
- inductive predicates
- higher-order constructions

```
function create (len: int) (f: int -> 'a) : seq 'a

axiom create_length: forall len: int, f: int -> 'a.
  0 <= len -> length (create len f) = len

axiom create_get: forall len: int, f: int -> 'a, i: int.
  0 <= i < len -> get (create len f) i = f i

constant square_seq : seq int = create 42 (fun i -> i * i)
```

Logical foundations

First-order logic offers a good compromise

- expressive enough to describe abstract models of our code
- tractable enough to allow for proof search automation

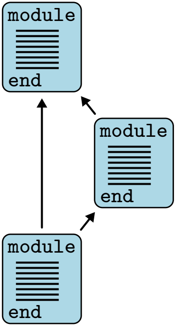
Admits many useful extensions without sacrificing tractability

- polymorphic types
- algebraic types and pattern matching
- inductive predicates
- higher-order constructions

All these extensions are supported in ACSL and WhyML

Modularity considered useful

WhyML declarations are organized in modules

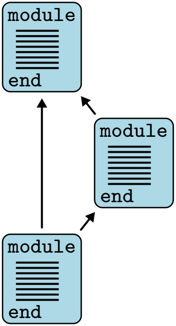


Modularity considered useful

WhyML declarations are organized in modules

a module M_1 can be

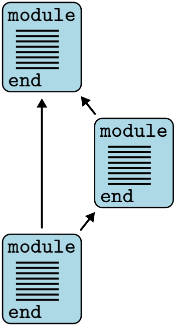
- used (`use`) in a module M_2
  - symbols of M_1 are shared
  - axioms of M_1 remain axioms
  - lemmas of M_1 become axioms
  - goals of M_1 are ignored



WhyML declarations are organized in modules

a module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2
  - declarations of M_1 are copied or instantiated
  - axioms of M_1 remain axioms or become lemmas
  - lemmas of M_1 become axioms
  - goals of M_1 are ignored



Modularity considered useful

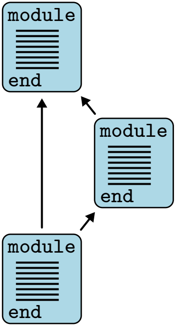
WhyML declarations are organized in modules

a module M_1 can be

- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2

cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function



Modularity considered useful

WhyML declarations are organized in modules

a module M_1 can be

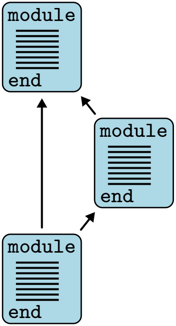
- used (`use`) in a module M_2
- cloned (`clone`) in a module M_2

cloning can instantiate

- an abstract type with a defined type
- an uninterpreted function with a defined function

one missing piece to come soon to Why3:

- instantiate a used module with another module



Module instantiation

```
module SortedList
  use import List

  type t
  predicate le t t
  clone relations.PartialOrder with type t = t, predicate rel = le

  inductive sorted (l: list t) =
    | Sorted_Nil: sorted Nil
    | Sorted_One: forall x: t. sorted (Cons x Nil)
    | Sorted_Two: forall x y: t, l: list t.
        le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end

module SortedListInt
  clone export SortedList with type t = int, predicate le = (<=)
end
```

- a functor-style alternative to polymorphism, often more convenient
- used to describe algebraic structures, parametrized theories, etc.
- a possible addition to ACSL specification modules?

Do you trust your axioms?

You should not.

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Solution: exhibit a model of your axiomatics.

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Solution: exhibit a model of your axiomatics. In Coq.

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Solution: exhibit a model of your axiomatics. In Coq. Or Isabelle, or PVS.

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Solution: exhibit a model of your axiomatics. In Coq. Or Isabelle, or PVS.

Caveat: the type system of Coq is different

- we must ensure that every type we consider
  - is inhabited (otherwise $\exists x : \tau . \top$ may be false)
  - has decidable equality

Do you trust your axioms?

You should not.

Why3 library, seq.Seq module, 21.03.15 — 12.04.15

```
axiom create_length: forall len: int, f: int -> 'a.
  length (create len f) = len
```

Solution: exhibit a model of your axiomatics. In Coq. Or Isabelle, or PVS.

Caveat: the type system of Coq is different

- we must ensure that every type we consider
  - is inhabited (otherwise $\exists x : \tau . \top$ may be false)
  - has decidable equality

- Coq type classes come to rescue

```
Class WhyType T := {
  why_inhabitant : T;
  why_decidable_eq : forall x y : T, { x = y } + { x <> y } }.
```

Finite sequences in Why3

```
type seq 'a
function length (s: seq 'a) : int
function create (len: int) (f: int -> 'a) : seq 'a
axiom create_length: forall len: int, f: int -> 'a.
  0 <= len -> length (create len f) = len
```

realized in Coq

```
Definition seq : forall (a:Type), Type.
    intro a. exact (list a). Defined.
Global Instance seq_WhyType :
  forall (a:Type) {a_WT:WhyType a}, WhyType (seq a). ... Qed.
Definition length:
  forall {a:Type} {a_WT:WhyType a}, (seq a) → Z. ... Defined.
Definition create:
  forall {a:Type} {a_WT:WhyType a}, Z → (Z → a) → (seq a). ... Defined.
Lemma create_length :
  forall {a:Type} {a_WT:WhyType a}, forall (len:Z) (f:(Z → a)),
    (0%Z <= len)%Z → ((length (create len f)) = len). ... Qed.
```

A common idea with multiple facets

A common idea with multiple facets

1. executable specifications — JML, E-ACSL, SPARK

    runtime checking, test generation

Programs as specification

A common idea with multiple facets

1. executable specifications — JML, E-ACSL, SPARK

   runtime checking, test generation

2. ghost code: local variables, function parameters, datatype fields

   invaluable for specification: property witnesses, data models, etc.

A common idea with multiple facets

1. executable specifications — JML, E-ACSL, SPARK

   runtime checking, test generation

2. ghost code: local variables, function parameters, datatype fields

   invaluable for specification: property witnesses, data models, etc.

```
type t 'a = { mutable size: int;   (* total number of elements *)
              mutable data: array (list (key, 'a)); (* buckets *)
        ghost mutable view: map key (option 'a); (* pure model *) }
  invariant { 0 < length data }
  invariant { forall i: int. 0 <= i < length data -> good_hash data i }
  invariant { forall k: key, v: 'a. good_data k v view data }
```

A common idea with multiple facets

1. executable specifications — JML, E-ACSL, SPARK

   runtime checking, test generation

2. ghost code: local variables, function parameters, datatype fields

   invaluable for specification: property witnesses, data models, etc.

```
type t 'a = { mutable size: int;   (* total number of elements *)
              mutable data: array (list (key, 'a)); (* buckets *)
        ghost mutable view: map key (option 'a); (* pure model *) }
  invariant { 0 < length data }
  invariant { forall i: int. 0 <= i < length data -> good_hash data i }
  invariant { forall k: key, v: 'a. good_data k v view data }
```

3. pure methods — why write the same code twice?

   in Why3, all logic types can be used in programs

Automated provers usually do not handle proofs by induction

```
let rec function fib (n: int) : int
  requires { n >= 0 }
  variant  { n }
= if n = 0 then 0 else
  if n = 1 then 1 else
  fib (n-1) + fib (n-2)

lemma fib_nonneg: forall n: int. 0 <= n -> 0 <= fib n
```

Automated provers usually do not handle proofs by induction

```
let rec function fib (n: int) : int
  requires { n >= 0 }
  variant  { n }
= if n = 0 then 0 else
  if n = 1 then 1 else
  fib (n-1) + fib (n-2)

lemma fib_nonneg: forall n: int. 0 <= n -> 0 <= fib n
```

Lemma functions — let us take advantage of the VC generator!

```
let rec lemma fib_nonneg (n: int) : unit
  requires { 0 <= n }
  ensures  { 0 <= fib n }
  variant  { n }
= if n > 1 then begin fib_nonneg (n-2); fib_nonneg (n-1) end
```

Easier than running Coq and uses familiar program constructions.

Conclusion

- A rich specification language saves a lot of time for users
  - ACSL and WhyML agree

Conclusion

- A rich specification language saves a lot of time for users
    - ACSL and WhyML agree
- Some features of ACSL are not yet fully implemented in Frama-C
    - Hope to see them sooner rather than later!

Conclusion

- A rich specification language saves a lot of time for users
  - ACSL and WhyML agree
- Some features of ACSL are not yet fully implemented in Frama-C
  - Hope to see them sooner rather than later!
- (Un)surprisingly, ACSL and WhyML have a lot in common
  - polymorphism, pattern matching, modules, ghost code, etc.
  - Frama-C/WP can refer to existing WhyML libraries using “drivers”

Conclusion

- A rich specification language saves a lot of time for users
  - ACSL and WhyML agree
- Some features of ACSL are not yet fully implemented in Frama-C
  - Hope to see them sooner rather than later!
- (Un)surprisingly, ACSL and WhyML have a lot in common
  - polymorphism, pattern matching, modules, ghost code, etc.
  - Frama-C/WP can refer to existing WhyML libraries using “drivers”
  - could that be a direct transparent interface instead?

Conclusion

- A rich specification language saves a lot of time for users
  - ACSL and WhyML agree
- Some features of ACSL are not yet fully implemented in Frama-C
  - Hope to see them sooner rather than later!
- (Un)surprisingly, ACSL and WhyML have a lot in common
  - polymorphism, pattern matching, modules, ghost code, etc.
  - Frama-C/WP can refer to existing WhyML libraries using “drivers”
  - could that be a direct transparent interface instead?

    Why3 plugin for Frama-C?

    ACSL parser for Why3?

    Let’s discuss it.