Une barrière d'abstraction explicite pour la vérification de programmes

Paul Patault

encadré par Jean-Christophe Filliâtre et Andrei Paskevich

Novembre 2025 @ GT LVP

université PARIS-SACLAY

LMF Laboratoire Méthodes Formelles

Vérification déductive 101

```
method Maximum(values: seq<int>) returns (max: int)
{
  max := values[0];
  for idx := 0 to |values| {
    if max < values[idx] {
      max := values[idx];
    }
  }
}
```

Vérification déductive 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{
  max := values[0];
  for idx := 0 to |values| {
    if max < values[idx] {
      max := values[idx];
    }
  }
}
```

Vérification déductive 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{ ... }
```

→ on **suppose** les préconditions

→ on **prouve** les postconditions

Vérification déductive 101

```
method Maximum(values: seq<int>) returns (max: int)
  requires |values| > 0
  ensures  max in values
  ensures  forall i :: 0 ⩽ i < |values| ⟹ values[i] ⩽ max
{ ... }
```

→ on **suppose** les préconditions

→ on **prouve** les postconditions

implémentation

---

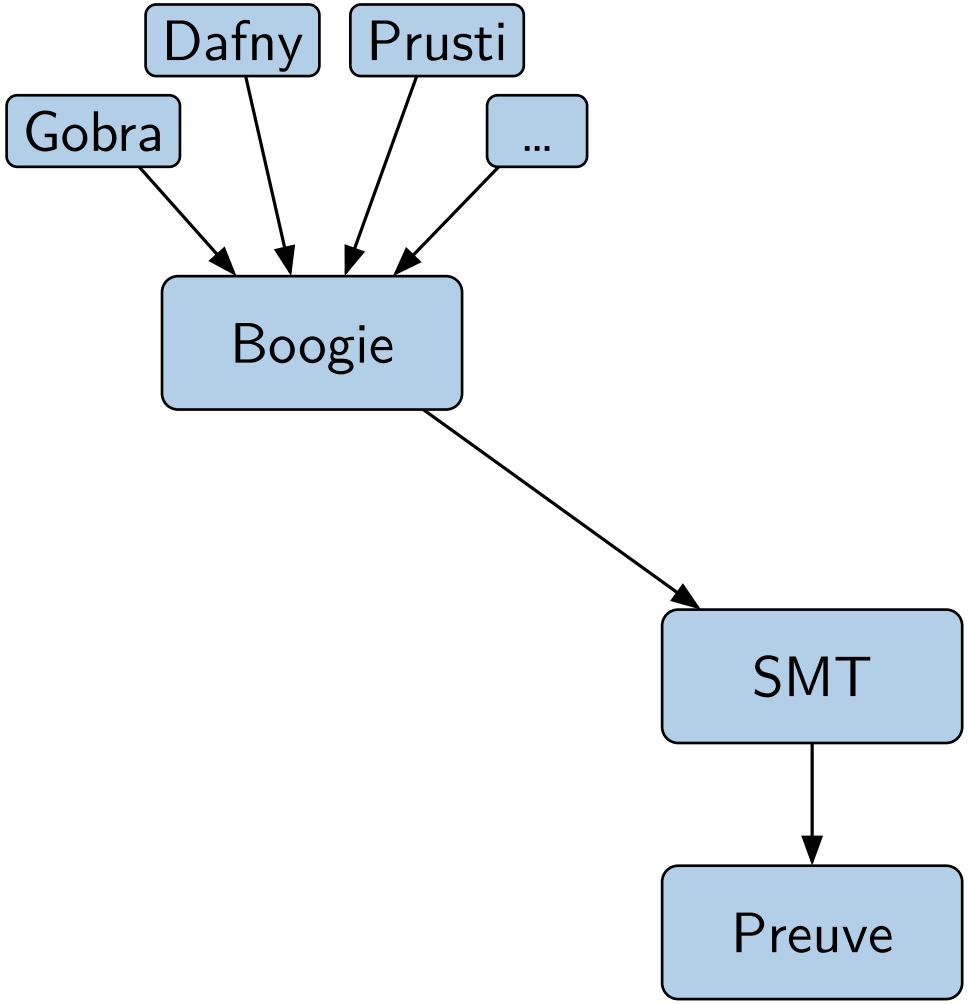
client

```
var m := Maximum(v);
...
```

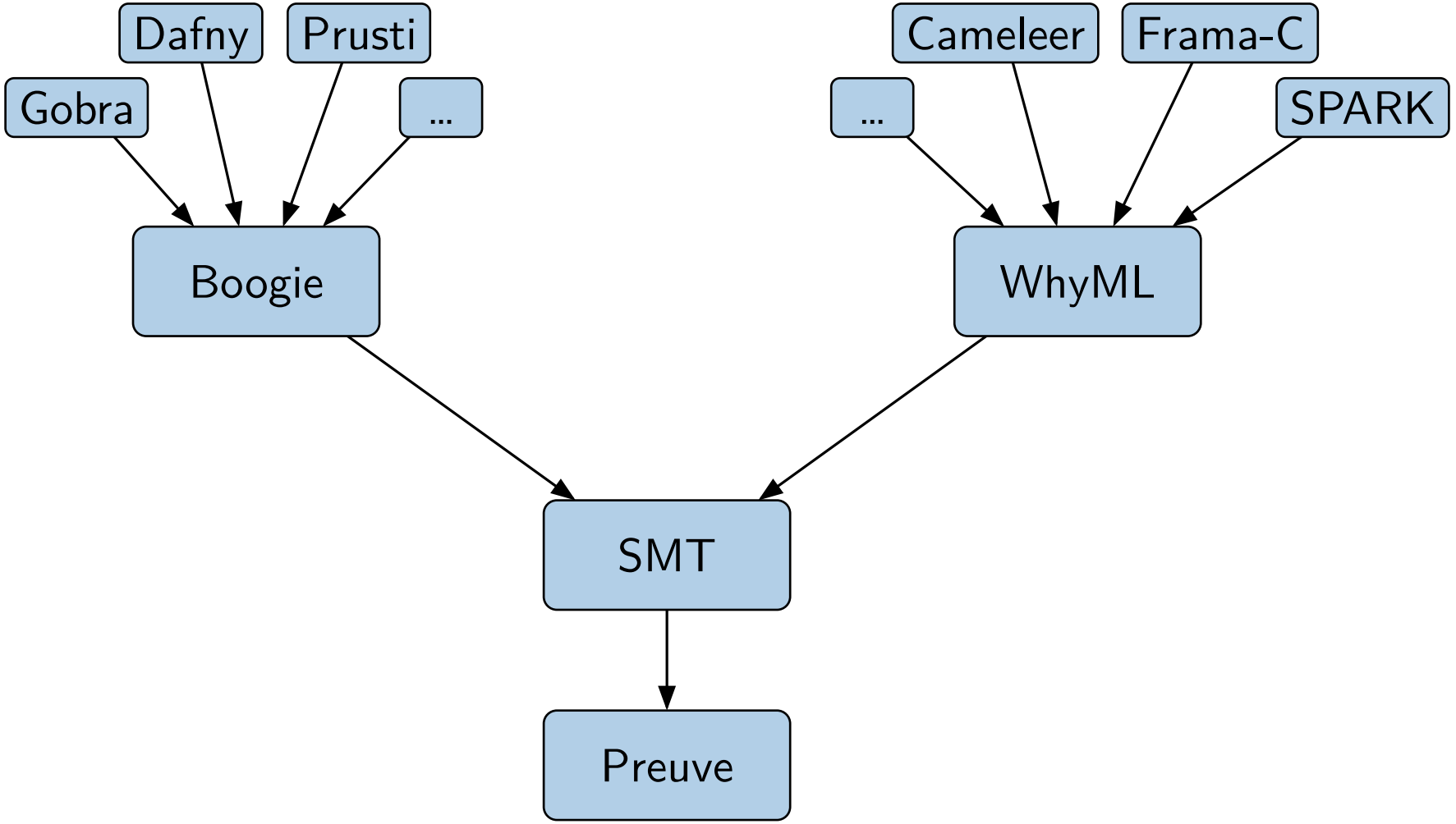
→ on **prouve** les préconditions

→ on **suppose** les postconditions

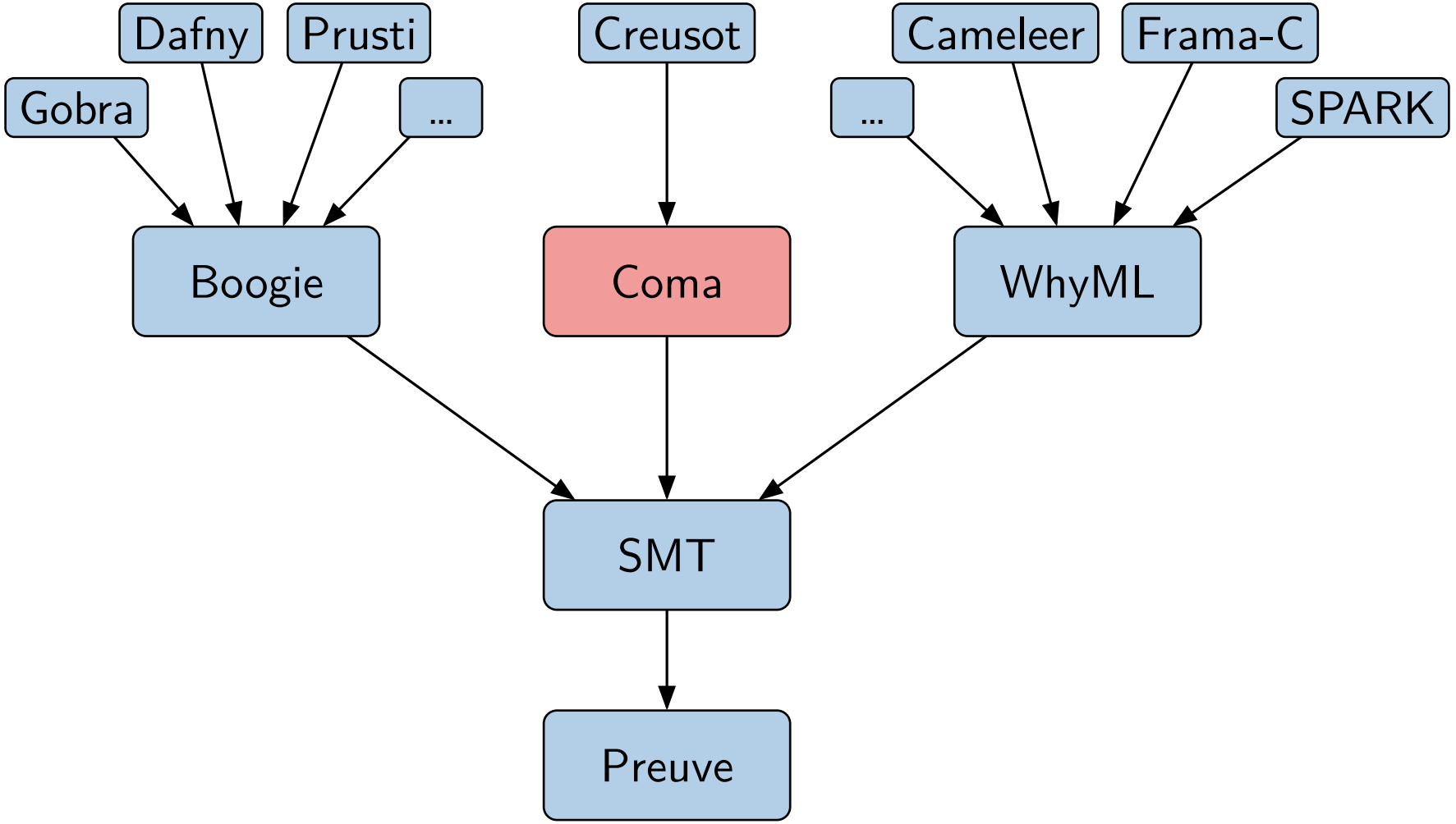
Langages intermédiaires (VCgen)



Langages intermédiaires (VCgen)



Langages intermédiaires (VCgen)



Coma *[Paskevich, Patault, Filliâtre]*

- « minimal »
  - définition et applications de fonctions
  - allocation de références
  - assertions logiques

- style par passage de continuations (CPS)
  - facilite l'encodage de structures de contrôle
  - factorise le calcul de VC

- **barrière d'abstraction explicite**

Le langage

$$
\begin{aligned}
e \;::=\;& f \\
\mid\;& \texttt{fun}\ \pi \longrightarrow e \\
\mid\;& e\ e \\
\mid\;& e\ t \\
\mid\;& \texttt{let rec}^{?}\ f\ \pi = e\ \texttt{in}\ e \\
\mid\;& \texttt{assert}\ \{\ \phi\ \}\ e \\
\mid\;& \texttt{hide}\ e
\end{aligned}
$$

$$
\pi \;::=\; (x : \tau)^{*}\ (f : \pi \longrightarrow \bot)^{*}
$$

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= if n < 0 then fail () else
  if n < 2 then out n   else
  fib (n-2) (fun x ⟶
  fib (n-1) (fun y ⟶
  out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= hide if n < 0 then fail () else
       if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= assert { n ⩾ 0 }
  hide if n < 0 then fail () else
       if n < 2 then out n else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  assert { n ⩾ 0 }
  hide if n < 0 then fail () else
       if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  assert { n ⩾ 0 }
  if n < 0 then fail () else
  hide if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  hide if n < 2 then out n   else
       fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

implémentation

---

client

```
fib 42 (fun r ⟶ assert { r > 10^8 } halt ())
```

Barrière d'abstraction

```
let rec fib (n: int) (out: int → ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x →
       fib (n-1) (fun y →
       out (x+y) ))
```

implémentation

---

client

```
fib 42 (fun r → assert { r > 10^8 } halt ())
```

$$
\begin{aligned}
&(42 < 0 \longrightarrow \mathsf{false}) \wedge \\
&(0 \leqslant 42 < 2 \longrightarrow 42 = F(42)) \wedge \\
&(\forall r.\ r = F(42) \longrightarrow r > 10^8)
\end{aligned}
$$

Barrière d'abstraction

```
let rec fib (n: int) (out: int ⟶ ⊥): ⊥
= let out r = assert { r = F(n) } hide out r in
  if n < 0 then fail () else
  if n < 2 then out n   else
  hide fib (n-2) (fun x ⟶
       fib (n-1) (fun y ⟶
       out (x+y) ))
```

implémentation

VC de la fonction

```
∀n. not n < 0 ⟶ not n < 2 ⟶
  (n-2 < 0 ⟶ false) ∧
  (0 ⩽ n-2 < 2 ⟶ n-2 = F(n-2)) ∧
  (2 ⩽ n-2 ⟶ ∀x. x = F(n-2) ⟶
   (n-1 < 0 ⟶ false) ∧
   (0 ⩽ n-1 < 2 ⟶ n-1 = F(n-1)) ∧
   (2 ⩽ n-1 ⟶ ∀y. y = F(n-1) ⟶
    x + y = F(n)))
```

Générateur modal de conditions de vérification

$\mathcal{A}$: vérification d'un appel

$\mathcal{D}$: vérification d'une définition

$$\mathcal{A}(\texttt{assert } \{ \ \phi \ \} \ \mathsf{e}) \triangleq \varphi \wedge \mathcal{A}(\mathsf{e})$$

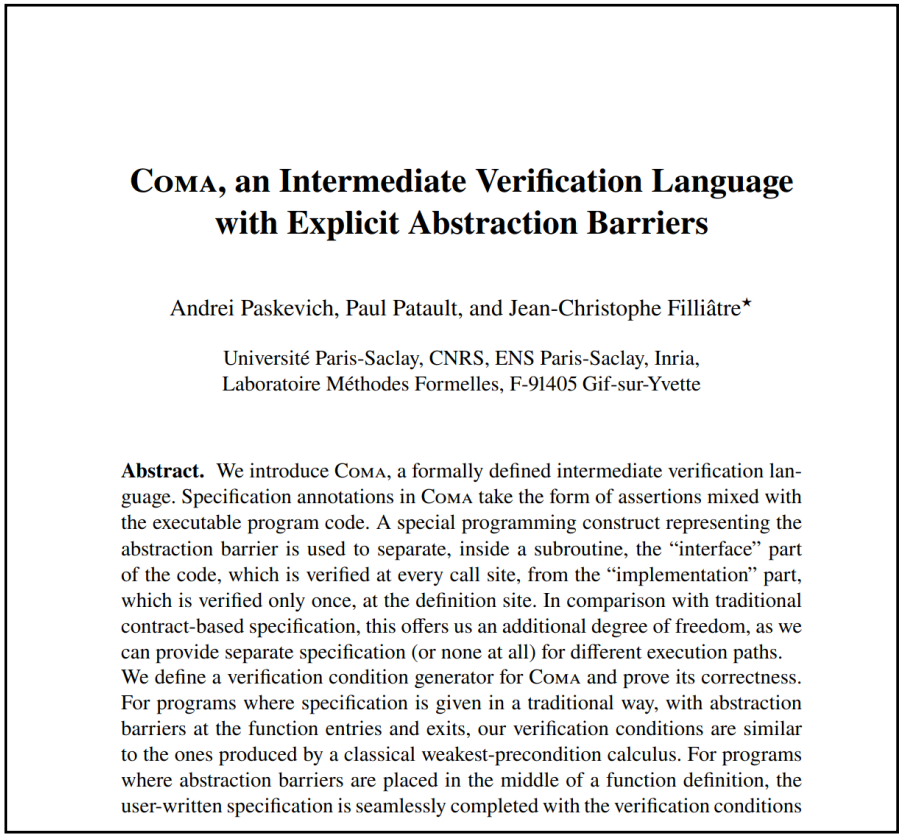
$$\mathcal{D}(\texttt{assert } \{ \ \phi \ \} \ \mathsf{e}) \triangleq \varphi \longrightarrow \mathcal{D}(\mathsf{e})$$

$$\mathcal{A}(\texttt{hide } \mathsf{e}) \triangleq \top$$

$$\mathcal{D}(\texttt{hide } \mathsf{e}) \triangleq \mathcal{A}(\mathsf{e}) \wedge \mathcal{D}(\mathsf{e})$$

Pour plus de détails

- langage Coma
- logique de « *recipes* »
- calcul de VC
- le tout prouvé en LaTeX



Coma, an Intermediate Verification Language with Explicit Abstraction Barriers

Andrei Paskevich, Paul Patault, and Jean-Christophe Filliâtre*

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria,
Laboratoire Méthodes Formelles, F-91405 Gif-sur-Yvette

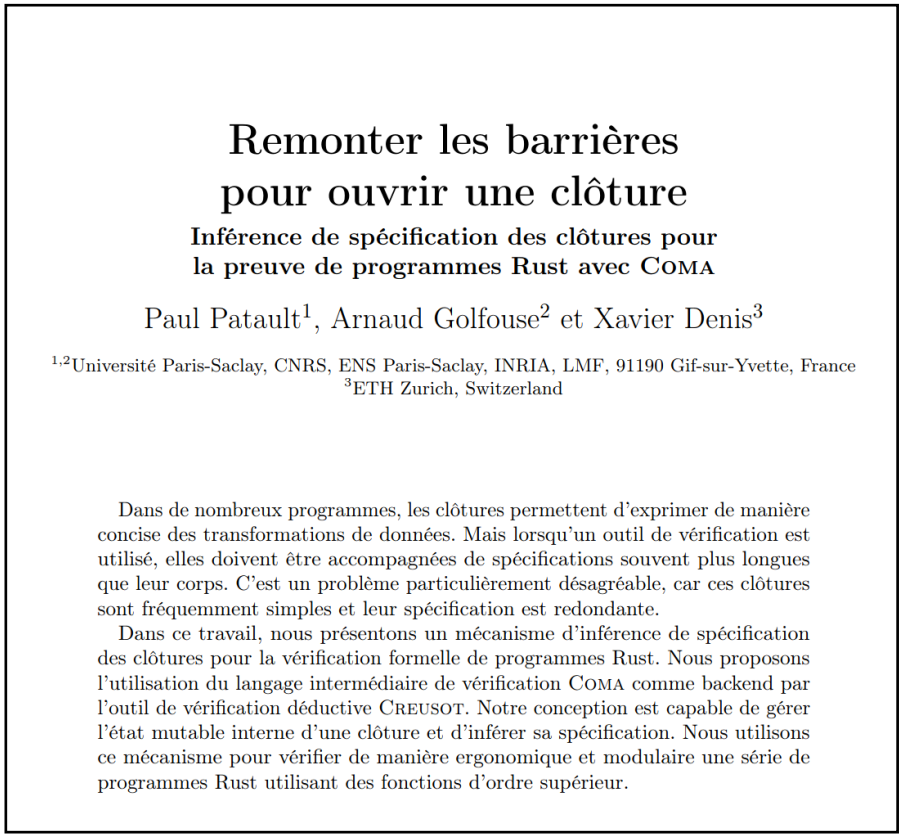
Abstract. We introduce Coma, a formally defined intermediate verification language. Specification annotations in Coma take the form of assertions mixed with the executable program code. A special programming construct representing the abstraction barrier is used to separate, inside a subroutine, the "interface" part of the code, which is verified at every call site, from the "implementation" part, which is verified only once, at the definition site. In comparison with traditional contract-based specification, this offers us an additional degree of freedom, as we can provide separate specification (or none at all) for different execution paths. We define a verification condition generator for Coma and prove its correctness. For programs where specification is given in a traditional way, with abstraction barriers at the function entries and exits, our verification conditions are similar to the ones produced by a classical weakest-precondition calculus. For programs where abstraction barriers are placed in the middle of a function definition, the user-written specification is seamlessly completed with the verification conditions

ESOP (mai 2025)

actuellement → preuve de correction du VCgen en Rocq

Pour plus de détails

- Creusot $\longrightarrow$ Coma
- utilisation de la barrière
- inférence de spécification pour les fermetures



Remonter les barrières pour ouvrir une clôture

Inférence de spécification des clôtures pour la preuve de programmes Rust avec Coma

Paul Patault[1], Arnaud Golfouse[2] et Xavier Denis[3]

[1,2]Université Paris-Saclay, CNRS, ENS Paris-Saclay, INRIA, LMF, 91190 Gif-sur-Yvette, France
[3]ETH Zurich, Switzerland

Dans de nombreux programmes, les clôtures permettent d'exprimer de manière concise des transformations de données. Mais lorsqu'un outil de vérification est utilisé, elles doivent être accompagnées de spécifications souvent plus longues que leur corps. C'est un problème particulièrement désagréable, car ces clôtures sont fréquemment simples et leur spécification est redondante.

Dans ce travail, nous présentons un mécanisme d'inférence de spécification des clôtures pour la vérification formelle de programmes Rust. Nous proposons l'utilisation du langage intermédiaire de vérification Coma comme backend par l'outil de vérification déductive Creusot. Notre conception est capable de gérer l'état mutable interne d'une clôture et d'inférer sa spécification. Nous utilisons ce mécanisme pour vérifier de manière ergonomique et modulaire une série de programmes Rust utilisant des fonctions d'ordre supérieur.

JFLA (janvier 2025)

Creusot *[Denis, Jourdan, Marché, Golfouse]*

```
let o = Some(42);

let a = o.map(
  #[requires(x@ + 1 ⩽ i32::MAX@)]
  #[ensures(result@ == x@ + 1)]
  |x| x + 1,
);
let b = o.map(
  #[requires(2 * x@ ⩾ i32::MIN@)]
  #[requires(2 * x@ ⩽ i32::MAX@)]
  #[ensures(result.0@ == 2 * x@)]
  #[ensures(result.1  == x)]
  |x| (2 * x, x),
);
```

Creusot *[Denis, Jourdan, Marché, Golfouse]*

```
let o = Some(42);

let a = o.map(
  #[requires(x@ + 1 ⩽ i32::MAX@)]
  #[ensures(result@ == x@ + 1)]
  |x| x + 1,
);
let b = o.map(
  #[requires(2 * x@ ⩾ i32::MIN@)]
  #[requires(2 * x@ ⩽ i32::MAX@)]
  #[ensures(result.0@ == 2 * x@)]
  #[ensures(result.1  == x)]
  |x| (2 * x, x),
);
```

```
let o = Some(42);

let a = o.map(|x| x + 1);


let b = o.map(|x| (2 * x, x));
```