SMT en pratique: le démonstrateur Alt-Ergo

Collège de France
23 mars 2016

Sylvain Conchon

LRI (UMR 8623), Université Paris-Sud
Équipe Toccata, INRIA Saclay – Île-de-France

La révolution SMT

70's:	Stanford Pascal Verifier
1984:	Shostak algorithm
1992:	Simplify
1995:	SVC
2001:	ICS
2002:	CVC, haRVey
2004:	CVC Lite
2005:	Barcelogic, MathSAT
2005:	Yices
2006:	CVC3, Alt-Ergo
2007:	Z3, MathSAT4
2008:	Boolector, OpenSMT, Beaver,Yices2
2009:	STP, VeriT
2010:	MathSAT5, SONOLAR
2011:	STP2, SMTInterpol
2012:	CVC4

Alt-Ergo, un démonstrateur SMT dédié à la preuve de programme

Preuve de programme par vérification déductive

A. M. Turing. *Checking a Large Routine.* 1949

Preuve de programme par vérification déductive

A. M. Turing. *Checking a Large Routine.* 1949

Exemple avec l'outil Why3 [Filliâtre et al.]

$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do**
  $v \leftarrow u$
  for $s = 1$ **to** r **do**
    $u \leftarrow u + v$

Preuve de programme par vérification déductive

A. M. Turing. *Checking a Large Routine.* 1949

Exemple avec l'outil Why3 [Filliâtre et al.]

précondition $\{n \geq 0\}$
$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do**
  $v \leftarrow u$
  for $s = 1$ **to** r **do**
    $u \leftarrow u + v$
postcondition $\{u = fact(n)\}$

Preuve de programme par vérification déductive

A. M. Turing. *Checking a Large Routine.* 1949

Exemple avec l'outil Why3 [Filliâtre et al.]

précondition $\{n \geq 0\}$
$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do** *invariant* $\{u = fact(r)\}$
  $v \leftarrow u$
  for $s = 1$ **to** r **do** *invariant* $\{u = s \times fact(r)\}$
    $u \leftarrow u + v$
postcondition $\{u = fact(n)\}$

Obligations de preuve

```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                 u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
            (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
      (u = fact((n - 1) + 1) → u = fact(n))))
```

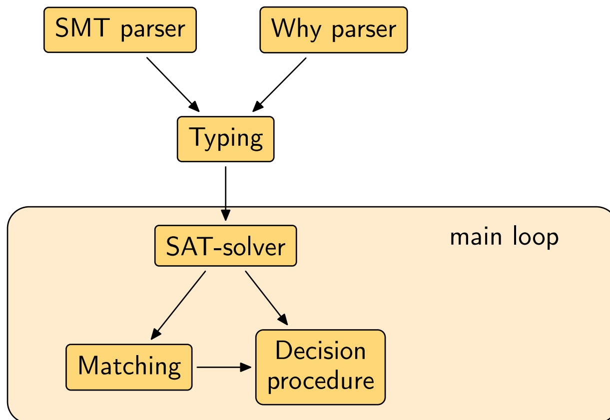
Point de départ

On identifie trois composants essentiels pour vérifier automatiquement de telles formules.

- Structure propositionnelle ($\wedge, \rightarrow$)
- Combinaison de quelques procédures de décision (égalité, arithmétique linéaire)
- Traitement des quantificateurs

$\rightarrow$ C'est le point de départ du développement d'Alt-Ergo

Architecture générale d'Alt-Ergo



- Front-end (parsing/typage)
- SAT solver
- Procédures de décision
- Un algorithme de filtrage (matching)

La boucle principale

Rappel : validité de F = insatisfiabilité de $\neg F$

Le SAT solveur traite la structure propositionnelle de la formule.

- il construit un modèle booléen (conjonction de littéraux) et il le vérifie en l'envoyant à la procédure de décision
- il conserve précisieusement les axiomes ($\forall x$: `int`. . . .) à part

Si le SAT a réussi à constuire un modèle satisfiable de la formule (modulo théories), on crée de nouvelles formules (instances) à partir des axiomes, en espérant trouver celles qui contrediront le modèle.

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Modèle : $\{ j = 0, 3i = j - 1 \}$

Axiomes : $\{ \forall x{:}int.\ f(x) > 0 \}$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Modèle : $\{ j = 0, 3i = j - 1 \} \rightarrow$ unsat

Axiomes : $\{ \forall x{:}int.\ f(x) > 0 \}$

Exemple

i,j :int

$j = 0 \wedge (\ 3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Modèle : $\{\ j = 0,\ f(i\text{-}1) \leq 2j\ \}$

Axiomes : $\{\ \forall x{:}int.\ f(x) > 0\ \}$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Modèle : $\{ j = 0, f(i\text{-}1) \leq 2j \} \rightarrow$ sat

Axiomes : $\{ \forall x{:}int.\ f(x) > 0 \}$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

Modèle : $\{ j = 0, f(i\text{-}1) \leq 2j \} \rightarrow$ sat

Axiomes : $\{ \forall x{:}int.\ f(x) > 0 \} \rightarrow$ (instance) $f(i\text{-}1) > 0$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

$\wedge\ f(i\text{-}1) > 0$

Modèle : $\{ j = 0,\ f(i\text{-}1) \leq 2j \}$

Axiomes : $\{ \forall x{:}int.\ f(x) > 0 \}$

Exemple

i,j :int

$j = 0 \wedge (3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

$\wedge\ f(i-1) > 0$

Modèle : $\{\ j = 0,\ f(i-1) \leq 2j,\ f(i-1) > 0\ \}$

Axiomes : $\{\ \forall x{:}int.\ f(x) > 0\ \}$

Exemple

i,j :int

$j = 0 \wedge ($ $3i = j - 1 \vee f(i - 1) \leq 2j) \wedge (\forall x{:}int.\ f(x) > 0)$

$\wedge\ f(i-1) > 0$

Modèle : $\{\ j = 0,\ f(i-1) \leq 2j,\ f(i-1) > 0\ \} \rightarrow$ unsat

Axiomes : $\{\ \forall x{:}int.\ f(x) > 0\ \}$

Un peu plus de détails...

Plan

Nous allons présenter les composants suivants d'Alt-Ergo :

- Un de ses mécanismes de combinaison de procédures de décision
- Sa gestion des formules avec quantificateurs
- Son traitement du polymorphisme
- Sa gestion des symboles AC

Traitement de l'égalité : l'algorithme de Shostak

Le cœur de la (combinaison de) procédure(s) de décision d'Alt-Ergo est l'algorithme de Shostak:

R. Shostak [1984]: Deciding combination of theories

Cet algorithme permet de déterminer si une conjonction d'égalités $\bigwedge_i u_i = v_i$ avec des symboles non-interprétés est satisfiable modulo une théorie X (dite de Shostak)

Théories de Shostak

Une théorie de Shostak est équipée de deux fonctions:

- un canonizer $\texttt{canon}_X$ qui met les termes en forme normale
- un solver $\texttt{solve}_X$ qui résout des équations et donne une solution sous la forme d'une substitution

Exemple: la théorie de l'arithmétique linéaire sur les entiers

$$\texttt{canon}_{LA}\ (a + 0 + b - (a + 1) + b) \quad = \quad 2 * b - 1$$

$$\texttt{solve}_{LA}\ (2 * a + 3 * b + 3 = 0) \quad = \quad \begin{cases} a \mapsto -3 * k \\ b \mapsto 2 * k - 1 \end{cases}$$

L'algorithme de Shostak

On maintient une partition des termes dans un dictionnaire (un terme est associé au représentant de sa classe d'équivalence)

Pour cela, on résout chaque égalité à l'aide du solver $\texttt{solve}_X$ et les termes sont maintenus en forme normale à l'aide de $\texttt{canon}_X$

Pour chaque égalité $u = v$:

1. canonizer u et v : $u' = \texttt{canon}_X(u)$ et $v' = \texttt{canon}_X(v)$
2. appeler $\texttt{solve}_X(u',v')$: renvoie une substitution σ
3. appliquer σ aux représentants et les canonizer

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a))=b$$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	f(a)
2a - f(a)	2a - f(a)
f(2a - f(a))	f(2a - f(a))

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On traite $f(a) - a = 0$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	f(a)
2a - f(a)	2a - f(a)
f(2a - f(a))	f(2a - f(a))

Un exemple ...

$$\text{f(a) - a} = 0 \wedge \text{f(2a - f(a))}=\text{b}$$

$$\texttt{solve}(\text{f(a) -a, 0}) = [\text{f(a)} \mapsto \text{a}]$$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	f(a)
2a - f(a)	2a - f(a)
f(2a - f(a))	f(2a - f(a))

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On applique la substitution

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	2a - a
f(2a - f(a))	f(2a - a)

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On canonize les représentants

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	f(a)

Un exemple ...

$$f(a) - a = 0 \land f(2a - f(a)) = b$$

On applique de nouveau la substitution

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	a

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On traite $f(2a - f(a)) = b$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	a

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On canonize les termes de l'équation : $a = b$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	a

Un exemple ...

$$\text{f(a) - a} = 0 \wedge \text{f(2a - f(a))}=\text{b}$$

$$\texttt{solve}(\text{a, b}) = [\text{b} \mapsto \text{a}]$$

termes	représentants
0	0
b	b
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	a

Un exemple ...

$$f(a) - a = 0 \wedge f(2a - f(a)) = b$$

On applique la substitution

termes	représentants
0	0
b	a
a	a
2a	2a
f(a)	a
2a - f(a)	a
f(2a - f(a))	a

Traitement des axiomes

Considérons les trois axiomes suivants (dans la syntaxe d'Alt-Ergo) qui définissent une relation d'ordre le

```
logic le: int,int → prop
axiom refl: forall x:int. le(x,x)
axiom trans:
   forall x,y,z:int. le(x,y) and le(y,z) → le(x,z)
axiom antisym:
   forall x,y:int. le(x,y) and le(y,x) → x = y
```

Traitement des axiomes

Considérons les trois axiomes suivants (dans la syntaxe d'Alt-Ergo) qui définissent une relation d'ordre le

```
logic le: int,int → prop
axiom refl: forall x:int. le(x,x)
axiom trans:
   forall x,y,z:int. le(x,y) and le(y,z) → le(x,z)
axiom antisym:
   forall x,y:int. le(x,y) and le(y,x) → x = y
```

et quelques buts que nous voudrions prouver:

```
goal g1: le(2,5) and le(5,10) → le(2,10)
goal g2: forall a:int. le(a,5) and le(5,8) and le(8,a) → a=5
```

Guider la génération d'instances

Durant la phase de recherche de preuve, Alt-Ergo cherche à créer de nouvelles instances des axiomes qu'il connait.

Guider la génération d'instances

Durant la phase de recherche de preuve, Alt-Ergo cherche à créer de nouvelles instances des axiomes qu'il connait.

Questions:

- Comment trouver les bonnes instances pour prouver un but ?
- Comment limiter le nombre d'instances (potentiellement prohibitif) ?

Guider la génération d'instances

Durant la phase de recherche de preuve, Alt-Ergo cherche à créer de nouvelles instances des axiomes qu'il connait.

Questions:

- Comment trouver les bonnes instances pour prouver un but ?
- Comment limiter le nombre d'instances (potentiellement prohibitif) ?

Réponse: en utilisant des heuristiques !

Guider la génération d'instances

Durant la phase de recherche de preuve, Alt-Ergo cherche à créer de nouvelles instances des axiomes qu'il connait.

Questions:

- Comment trouver les bonnes instances pour prouver un but ?
- Comment limiter le nombre d'instances (potentiellement prohibitif) ?

Réponse: en utilisant des heuristiques !

Les deux ingrédients principaux qui guident ou restreignent la création de nouvelles instances sont :

- des (listes de) motifs (triggers)
- une base de données de termes clos connus

Motifs

Considérons l'axiome suivant:

logic P,Q,R: int $\rightarrow$ prop

axiom ax1: forall x:int. (P(x) or Q(x)) $\rightarrow$ R(x)

Pour ne pas créer toutes les instances de ax1, pour tous les entiers x de l'univers, on peut restreindre son utilisation, par exemple au motif P(x)

Cette restriction a pour effet d'autoriser la création de nouvelles instances, si et seulement si un terme de la forme P(a) est "connu" par Alt-Ergo.

"Connu" veut dire que ce terme apparait dans le modèle SAT

Effet des motifs sur la preuve

```
logic P,Q,R: int → prop
axiom ax1: forall x:int. (P(x) or Q(x)) → R(x)
```

Ainsi, avec le motif P(x), on pourra prouver le but suivant :

goal g3: P(1) → R(1)

Par contre, on ne pourra pas prouver :

goal g4: Q(2) → R(2)

Pour cela, il faudrait utiliser le motif Q(x) pour ax1.

Heureusement, Alt-Ergo permet d'utiliser une liste de motifs. Il dipose également d'un algorithme qui tente de calculer les meilleurs motifs pour chaque axiome.

Motifs explicites

Au cas où les motifs trouvés par Alt-Ergo ne soient pas les meilleurs, l'utilisateur peut en ajouter lors de la définition d'un axiome

Par exemple, on peut ajouter la liste de motifs explicites [f(x), Q(y)] à l'axiome ax2 suivant:

```
logic P,Q,R: int → prop
logic f: int → int
axiom ax2: forall x,y:int [f(x), Q(y)]. P(f(x)) and Q(y) → R(x)
```

Algorithme de filtrage

Alt-Ergo utilise un algorithme de filtrage (matching) pour créer de nouvelles instances de formules universellement quantifiées.

Étant donné un terme t et un motif P, l'algorithme renvoie un ensemble de substitutions S sur les variables de P, tel que

$$\mathsf{t} = \sigma(\mathsf{P}) \quad \text{pour toute substitution} \quad \sigma \in \mathsf{S}$$

Limitations du filtrage

Ce filtrage purement syntaxique est très limité!

Considérons par exemple le problème suivant:

logic P,R : int $\rightarrow$ prop
logic f : int $\rightarrow$ int

axiom ax : forall x:int [P(f(x))]. P(f(x)) $\rightarrow$ R(x)

goal g : forall a:int. P(a) $\rightarrow$ a = f(2) $\rightarrow$ R(2)

Le motif P(f(x)) ne permet pas le création de l'instance de l'axiome ax qui permettrait de prouver g car il n'y a aucun terme connu de la forme P(f(_)) dans notre problème

Pour prouver ce but, il faut étendre l'algorithme de filtrage afin de trouver des substitutions modulo les égalités connues (ici, a = f(2))

E-filtrage

Étant donné un ensemble d'équations E, un terme t et un motif P, l'algorithme de e-filtrage renvoie un ensemble de substitutions S sur les variables de P, tel que

$$E \models t = \sigma(P) \quad \text{pour toute substitution} \quad \sigma \in S$$

Sur l'exemple précédent:

```
logic P,R : int → prop
logic f : int → int

axiom ax : forall x:int [P(f(x))]. P(f(x)) → R(x)

goal g1 : forall a:int. P(a) → a = f(2) → R(2)
```

l'algorithme de e-filtrage utilise l'égalité $a = f(2)$ pour renvoyer la substitution $\sigma = \{x \mapsto 2\}$ qui permet de créer $P(f(2)) \to R(2)$

Base de données des termes connus

La base de données des termes connus est formée des termes extraits des littéraux supposés ou impliqués par le SAT solveur

Le mécanisme de création d'instances est donc fortement impacté par le nombre et la pertinence de ces termes connus

- Plus il y a de termes connus, plus il y a d'instances de créées
- Des termes connus inutiles vont créer des instances inutiles

Termes connus et mise en CNF

La forme des formules à prouver, et en particulier le processus de transformation en CNF, a un impact très fort sur le nombre de termes connus

Considérons par exemple la formule suivante:

$$A \lor (B \land C)$$

Quand A est supposé vrai, les termes de A deviennent connus et le reste des termes de la sous-formule $(B \land C)$ peuvent être ignorés

Termes connus et mise en CNF

La forme des formules à prouver, et en particulier le processus de transformation en CNF, a un impact très fort sur le nombre de termes connus

Considérons par exemple la formule suivante:

$$A \lor (B \land C)$$

Quand A est supposé vrai, les termes de A deviennent connus et le reste des termes de la sous-formule $(B \land C)$ peuvent être ignorés

Cependant, la conversion en CNF (de Tseitin) donne

$$(A \lor X) \land (X \leftrightarrow (B \land C))$$

et le solveur SAT va devoir affecter une valeur à X (même si A est vrai) et les termes de B et C seront ajoutés à la base des termes.

Mise en CNF paresseuse

Alt-Ergo utilise un SAT solveur couplé à un algorithme de mise en CNF qui transforme paresseusement la formule en entrée.

Par exemple, plutôt que d'envoyer la (CNF de la) formule $A \vee (B \wedge C)$ au SAT, celui-ci reçoit uniquement la formule $A \vee X$, où X est un littéral particulier, appelé proxy.

Puis, le SAT "ouvre" dynamiquement ces proxys quand il les ajoute à son modèle.

Ainsi, c'est seulement quand X est supposé (vrai ou faux) par le SAT que l'on ajoute:

- les deux littéraux B et C, si X est supposé vrai
- la clause $\neg B \vee \neg C$, si X est supposé faux

Ainsi, on maîtrise mieux la base des termes connus lors des phases de filtrage.

Polymorphisme

Alt-Ergo est le seul solveur SMT avec un langage d'entrée polymorphe

type α list
logic cons:α, α list $\rightarrow$ α list
logic hd:α list $\rightarrow$ α
axiom hdcons : forall x:α. forall l:α list. hd(cons(x,l)) $=$ x

- C'est un polymorphisme prenexe à la ML

Traitement du polymorphisme par monomorphization

On peut penser à traiter les formules polymorphes en calculant statiquement :

- les instances des types polymorphes nécessaires pour prouver un but
- en répliquant les symboles et axiomes pour chacune de ces instances de type

Monomorphization : exemple

Considérons les axiomes polymorphes de la théorie des tableaux:

type (α,β) array
logic select : (α,β) array, $\alpha \rightarrow \beta$
logic store : (α,β) array, α, $\beta \rightarrow (\alpha,\beta)$ array
axiom a1 : $\forall$a:(α,β) array. $\forall$i:α. $\forall$e:β. select((store(a,i,e),i) = e
axiom a2 : $\forall$a:(α,β) array.$\forall$i,j:α.$\forall$e:β. i$\neq$j $\rightarrow$ select(store(a,i,e),j) = select(a,j)
axiom a3 : $\forall$a,b:(α,β) array. ($\forall$ i:α. select(a,i)=select(b,i)) $\rightarrow$ a=b

Monomorphization : exemple

Considérons les axiomes polymorphes de la théorie des tableaux:

type (α,β) array
logic select : (α,β) array, $\alpha \rightarrow \beta$
logic store : (α,β) array, α, $\beta \rightarrow (\alpha,\beta)$ array
axiom a1 : $\forall$a:(α,β) array. $\forall$i:α. $\forall$e:β. select((store(a,i,e),i) = e
axiom a2 : $\forall$a:(α,β) array.$\forall$i,j:α.$\forall$e:β. i$\neq$j $\rightarrow$ select(store(a,i,e),j) = select(a,j)
axiom a3 : $\forall$a,b:(α,β) array. ($\forall$ i:α. select(a,i)=select(b,i)) $\rightarrow$ a=b

Ainsi, pour prouver le but suivant :

goal g:
$\forall$ i,j:int.
$\forall$ m:(int,(int,int) array) array.
$\forall$ r:(int,int) array. r = select(m,i) $\rightarrow$ store(m, i, store(r, j, select(r, j))) = m

il faut générer les instances des axiomes avec les types (int,(int,int) array) array et (int,int) array.

Limite de la monomorphization

Malheureusement, le fait de déterminer les instances de types (et donc d'axiomes) nécessaires pour prouver une formule monomorphe n'est pas décidable !

[Bobot & Paskevich, Frocos 2011]

Traitement du polymorphisme dans Alt-Ergo

L'extension de la boucle principale d'Alt-Ergo (vue précédemment) au traitement des types et formules polymorphes revient uniquement à :

- modifier le front-end (système de type)
- étendre l'algorithme de filtraqe pour renvoyer des substitutions de type

Le reste du système est inchangé !

Traitement des symboles AC

Le traitement des symboles associatif-commutatif (AC) est difficile à automatiser :

(A) $\forall$ x, y, z. u(u(x , y) , z) = u(x , u(y , z))

(C) $\forall x, y.$ u (x , y) = u(y , x)

Alt-Ergo étend l'algorithme de Shostak pour traiter les symboles AC définis par l'utilisateur :

```
logic ac u : int, int → int

goal g :
   forall x, y, z, a, b, c1, c2, d, e1, e2:int.
   u(z-y,b) - a = x and u(b,y+1)=a and z=2*y+1 → x=0
```

Cela est réalisé en combinant des techniques de reécriture modulo AC (complétion close modulo AC) avec les solvers/canonizers des théories de Shostak.

Pour résumer

Alt-Ergo en un clin d'oeil

- Solveur SMT purement fonctionnel écrit en OCaml
- Principaux algorithmes certifiés avec Coq

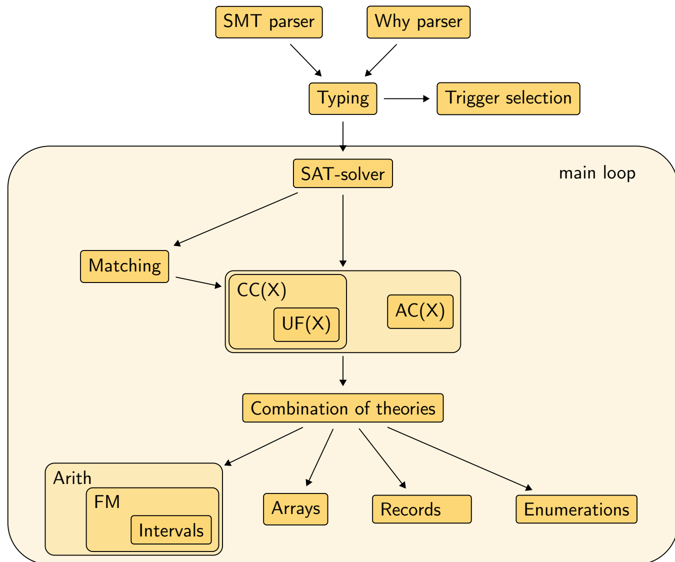
Alt-Ergo en un clin d'oeil

- Solveur SMT purement fonctionnel écrit en OCaml
- Principaux algorithmes certifiés avec Coq

Quelques particularités d'Alt-Ergo :

- Combinaison de procédures de décision à la Shostak
- CNF paresseuse
- Logique polymorphe
- Traitement des symboles AC

Architecture complète d'Alt-Ergo

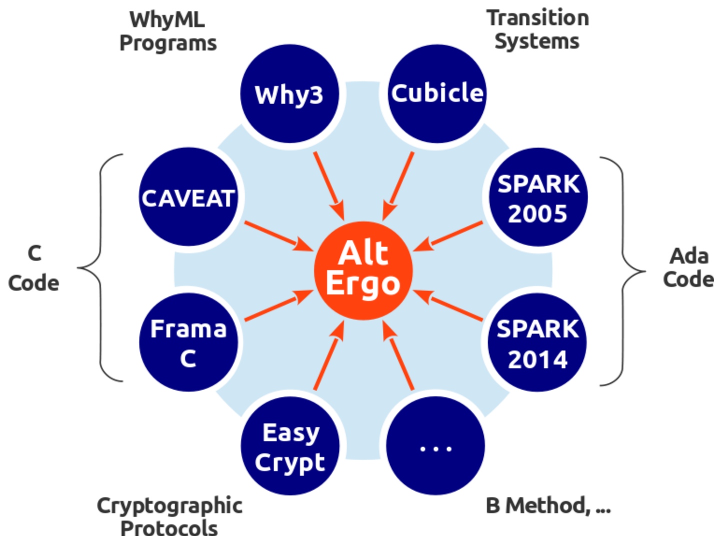


Chronologie de la construction d'Alt-Ergo

- 10/2006 : SAT naïf, égalité, arithmétique linéaire
- 02/2007 : polymorphisme, CC(X)
- 07/2008 : backjumping, stratégies de preuve
- 07/2009 : symboles AC
- 05/2010 : théorie des tableaux, arithmétique non-linéaire
- 04/2011 : types éumérés, interface graphique, explications
- 12/2011 : théorie des enregistrements,
- 01/2013 : production de modèles
- 09/2013 : OCamlPro (vers un démonstrateur plus efficace)
- 12/2014 : architecture à plugins
- 01/2015 : nouvelles structures de données, CDCL
- 11/2015 : améliorations (arithmétique, analyse par cas)
- 02/2016 : améliorations (filtrage, SAT)

Alt-Ergo dans la pratique

Utilisations d'Alt-Ergo



Quelques résultats

	Alt-Ergo (1.20)	CVC4 (1.4)	Z3 (4.4.0)
Atelier-B (12928 PO)	97,34% (7328s)	93,97% (5860s)	85,88% (2668s)
Gallerie Why3 (9752 PO)	89.20% (6808s)	85.39% (3069s)	79.83% (5456s)
SPARK 2014 (15994 PO)	78.44% (3116s)	75.92% (612s)	78.10% (1279s)

Qualification DO-178C

Alt-Ergo est le seul solveur SMT qualifié au sens de la norme DO-178C (utilisé par Airbus Industrie dans le cadre de son projet A350).

Contributeurs

Les principaux contributeurs à Alt-Ergo sont :

François Bobot (CEA)

Évelyne Contejean (LRI)

Denis Cousineau (Prove & Run)

Claire Dross (Adacore)

Mohamed Iguernlala (OcamlPro)

Johanes Kanig (Adacore)

Stéphane Lescuyer (Prove & Run)

Alain Mebsout (Iowa University)

Cody Roux (Carnegie Mellon University)

MERCI !

Venez visiter le site web d'Alt-Ergo:

https://alt-ergo.ocamlpro.com

An SMT Solver For Software Verification