

# VÉRIFICATION DE PROGRAMMES PAR INTERPRÉTATION ABSTRAITE

Patrick COUSOT

École normale supérieure

cousot@ens.fr

<http://www.di.ens.fr/~cousot>

## PLAN DE L'EXPOSÉ

- Un tour d'horizon rapide des méthodes formelles :
  - méthodes déductives
  - méthodes exhaustives (model-checking)
  - analyse statique
- Exemple : preuves d'invariance par analyse statique
- Quelques idées formalisées par l'interprétation abstraite
- Catégories d'analyseurs statiques
- quelques thèmes de recherche en interprétation abstraite
- Références bibliographiques

UN TOUR D'HORIZON RAPIDE  
DES MÉTHODES FORMELLES

## VERIFICATIONS DE LOGICIELS

Démontrer :

- qu'une propriété (d'une certaine classe)
- d'un programme (d'une certaine famille)

est vraie

- pour tous les comportements possibles du programme à l'exécution
- dans tous les environnements possibles de cette exécution

→ INDÉCIDABLE

## MODÉLISATION DU PROBLÈME

1 Quelle famille de programmes d'intérêt ?

- langage de programmation
- type d'application

2 Comment spécifier les propriétés à démontrer (dans une classe d'intérêt donnée) :

- langage / logique de spécification
- prédefinies (en fonction du programme)

3 Comment spécifier les environnements possibles des exécutions

- explicitement (langage / logique / modèle)
- implicitement (prédefini, paramétrisation)

4 Comment déterminer les comportements possibles du programme s'exécutant dans un environnement possible quelconque

- sémantique
- modèle (fourni manuellement)

5 Comment faire la preuve ?

- interactivement
- automatiquement

# CONSÉQUENCES DE L'INDÉCIDABILITÉ / COMPLEXITÉ

## - INDÉCIDABILITÉ :

Impossible de faire un outil automatique universel sans limites

## - CONSÉQUENCES :

- Degré de généralité de l'outil :

- Outils universels : toujours utilisables, mais échec possible :

- faute de ressources (non terminaison, dépassement des capacités de la machine)
    - faute de précision (fausses alarmes)

- Outils spécialisés : utilisables dans certains cas seulement (décidabilité).

- Degré d'automatique de l'outil :

- Outils interactifs : aide indispensable de l'utilisateur
  - Outils adaptatifs : possibilité d'adaptation de l'outil au problème (par le concepteur voire l'utilisateur).

# GRANDES CATEGORIES D'OUTILS

## (1) MÉTHODES DÉDUCTIVES

- Programmes : sous-ensemble d'un langage de programmation
- Propriétés : spécification par une logique (ex: logique de Moore & prédictats du 1<sup>er</sup> ordre)
- Environnements : spécification par une logique (ex: préconditions)
- Comportement du programme : conditions de vérification déterminées par le concepteur à partir de la sémantique du langage de programmation
- Comment faire la preuve : démonstrateur de théorème + aide de l'utilisateur pour les preuves inducitives
- Généralité : outil universel (dans la mesure où l'utilisateur peut fournir l'assistance nécessaire à la preuve)
- Automatisme : outil interactif (assistance utilisateur indispensable, peut être très difficile).

## GRANDES CATÉGORIES D'OUTILS

### (2) MÉTHODES EXHAUSTIVES (MODEL-CHECKING)

- Programmes : langage d'entrée de l'outil (ex: Promela pour SPIN permettant de décrire des systèmes concurrents communiquant par messages synchrones/asynchrones sur des canaux)
- Propriétés : logique temporelle ou pré définies (blocages, familles, terminaison, ...)
- Environnements : description explicite dans le langage d'entrée de l'outil
- Comportement du programme : fourni (par l'utilisateur) par traduction dans le langage d'entrée de l'outil
- Comment faire la preuve : par énumération ( vérification d'un nombre fini de cas, pas d'induction) éventuellement partielle (SAT).
- Généralité : outil spécialisé (pour une classe de modèles et de propriétés)
- Automatique : outil automatique (échec faute de temps ou de ressources), pas d'interaction possible sauf à changer le modèle de comportement du programme.

# GRANDES CATEGORIES D'OUTILS

## (3) ANALYSE STATIQUE

- Programmes : (sous-ensemble) d'un langage de programmation
- Propriétés : prédefinies ou spécifiées par l'utilisateur
- Environnements : prédefinis ou spécifiés par l'utilisateur
- Comportements du programme : déterminés par le concepteur à partir d'une sémantique du langage de programmation
- Comment faire la preuve : calcul effectif par l'outil d'une approximation des comportements du programme puis preuve de satisfaction des spécifications
- Généralité : outil spécialisé (au langage, à une classe de propriétés d'intérêt, presque toujours indécidables)
- Automatique : outil automatique (fausses alertes), adaptatif (par le concepteur en améliorant la précision de l'approximation ou l'utilisateur en fournissant des indications sur les comportements souhaités du programme).

EXEMPLE : PREUVES D'INVARIANCE  
PAR ANALYSE STATIQUE

## EXEMPLE : SÉMANTIQUE

- Programme

1:

while  $x > 0$  do

2:       $x := x - 1$

3:      od;

4:

- États:  $s = \langle e, x \rangle \in \Sigma$

$e$ : étiquette

$e \in \{1, 2, 3, 4\}$

$x \in \mathbb{Z}$ , valeur de  $X$

ou  $x \in [\min, \max] \cup \{-\infty\}$

- Booléens:  $\mathbb{B} = \{\text{tt}, \text{ff}\}$

- États d'entrée:

$\varepsilon \in \Sigma \rightarrow \mathbb{B}$

$\varepsilon(\langle e, x \rangle) = [e=1]$

ou  $[e=1 \wedge x \in [\min, \max]]$

### Relation de transition

- $t \in \Sigma \times \Sigma \rightarrow \mathbb{B}$

- $t(\langle e, x \rangle, \langle e', x' \rangle) = \text{tt}$  s'écrit  $\langle e, x \rangle \xrightarrow{t} \langle e', x' \rangle$

$\langle 1, x \rangle \xrightarrow{t} \langle e, x \rangle$  si  $x > 0$

$\langle 1, x \rangle \xrightarrow{t} \langle 4, x \rangle$  si  $x \leq 0$

$\langle e, x \rangle \xrightarrow{t} \langle 3, x-1 \rangle$

$\langle 3, x \rangle \xrightarrow{t} \langle e, x \rangle$  si  $x > 0$

$\langle 3, x \rangle \xrightarrow{t} \langle 4, x \rangle$  si  $x \leq 0$

ou  $\langle e, x \rangle \xrightarrow{t} \langle 3, x-1 \rangle$

si  $x-1 \in [\min, \max]$

$\langle e, x \rangle \xrightarrow{t} \langle 3, x \rangle$

si  $x-1 \notin [\min, \max]$

### Fermeture transitive réflexive:

- $t^* \in \Sigma \times \Sigma \rightarrow \mathbb{B}$

$\langle e, x \rangle \xrightarrow{t^*} \langle e', x' \rangle$

ssi  $\langle e, x \rangle = \langle e_0, x_0 \rangle \xrightarrow{t} \langle e_1, x_1 \rangle \xrightarrow{t} \dots \xrightarrow{t} \langle e_n, x_n \rangle = \langle e', x' \rangle$   
 $n \geq 0$ .

### États accessibles:

$\{s' \mid \exists a \in \Sigma : \varepsilon(a) \wedge t^*(s, s')\}$



## EXEMPLE : SPÉCIFICATION

- Propriété des états :

$$\rho : \Sigma \rightarrow \text{IB}$$

- Propriété INVARIANTE

Toujours vraie au cours d'une exécution quelconque commençant par un état initial :

$$\forall s, s' \in \Sigma : \varepsilon(s) \wedge t^*(s, s') \Rightarrow \rho(s')$$

- Exemples :

- Absence d'erreurs à l'exécution (RTE) :

$$\rho(\langle e, x \rangle) = [x \neq -2]$$

(avec la sémantique d'erreurs)

- Intervalle de variation :

$$\begin{aligned} \rho(\langle e, x \rangle) = & (\ell = 2 \quad ? \quad x > 0 \\ & | \ell = 3 \quad ? \quad x \geq 0 \\ & | \ell = 4 \quad ? \quad x \leq 0) \end{aligned}$$

## EXEMPLE : PREUVE D'INVARIANCE

- Problème :  $\Sigma$  états du programme  
 $t \in \Sigma \times \Sigma \rightarrow \mathbb{B}$ ,  $\mathbb{B} = \{\text{tt}, \text{ff}\}$  transition entre un état et ses successeurs possibles

$\varepsilon \in \Sigma \rightarrow \mathbb{B}$ , états d'entrée

$P \in \Sigma \rightarrow \mathbb{B}$ , propriété d'invariance

Démontrer :

$$\forall s, s' \in \Sigma : \varepsilon(s) \wedge t^*(s, s') \Rightarrow P(s')$$

- Principe de la preuve :

Trouver  $I \in \Sigma \rightarrow \mathbb{B}$ , tel que

$$(i) \quad \forall s \in \Sigma : \varepsilon(s) \Rightarrow I(s)$$

$$(ii) \quad \forall s, s' \in \Sigma : I(s) \wedge t(s, s') \Rightarrow I(s')$$

$$(iii) \quad \forall s \in \Sigma : I(s) \Rightarrow P(s)$$

- Méthodes déductives : - demander  $I$  à l'utilisateur  
- vérifier (i), (ii), (iii), possibilité d'échec.

- Vérification exhaustive : - calculer  $I(s') = [\exists s \in \Sigma : \varepsilon(s) \wedge t^*(s, s')]$   
(en supposant  $\Sigma$  fini, possibilité d'échec)  
- vérifier (iii)

- Analyse statique : - calculer  $I(s') \Leftarrow [\exists s \in \Sigma : \varepsilon(s) \wedge t^*(s, s')]$   
par des méthodes effectives d'approximation  
- vérifier (iii), possibilité de fausse alarme.

## EXEMPLE : APPROXIMATION PAR INTERVALLES

1: while  $x > 0$  do  
2:      $x := x - 1$   
3: od;  
4:

$P : \mathbb{Z} \rightarrow \mathbb{B}$       ou     $\Sigma = \{1, \dots, 4\} \times \mathbb{Z}$

- $P \longmapsto \langle P_1, P_2, P_3, P_4 \rangle$

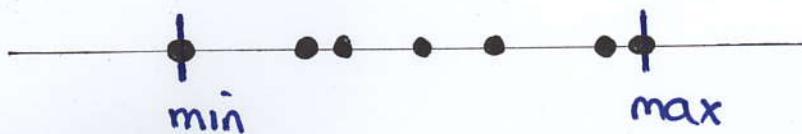
$$P_e = \{x \mid P(\langle e, x \rangle)\}$$

Ensemble des valeurs de  $X$  au point  $e$  quand  $P$  est vrai

- $\langle P_1, P_2, P_3, P_4 \rangle \longmapsto \langle X_1, X_2, X_3, X_4 \rangle$

$$X_e = [\underline{\min} P_e, \underline{\max} P_e]$$

Valeurs minimale et maximale de  $X$  au point  $e$  quand  $P$  est vrai.



## EXEMPLE : CONTRAINTES D'INTERVALLES

1:    while  $x > 0$  do  
 2:         $x := x - 1$   
 3:  
 4:        od ;

$$\begin{aligned} X \in [a_1, b_1] \\ X \in [a_2, b_2] \\ X \in [a_3, b_3] \\ X \in [a_4, b_4] \end{aligned}$$

Approximation des états accessibles par un intervalle de valeurs possibles en chaque point du programme.

Contraintes :

$$\left\{ \begin{array}{l} [a_1, b_1] = [\underline{\min}, \underline{\max}] \\ [a_2, b_2] = ([a_1, b_1] \cup [a_3, b_3]) \cap [1, \underline{\max}] \\ [a_3, b_3] = [a_2 - 1, b_2 - 1] \cap [\underline{\min}, \underline{\max}] \\ [a_4, b_4] = ([a_1, b_1] \cup [a_3, b_3]) \cap [\underline{\min}, 0] \end{array} \right. \quad \text{états initiaux}$$

où

$$[a, b] \cup [a', b'] = [\underline{\min}(a, a'), \underline{\max}(b, b')]$$

$$\begin{aligned} [a, b] \cap [a', b'] &= [\underline{\max}(a, a'), \underline{\min}(b, b')] \quad \text{si } \underline{\max}(a, a') \leq \underline{\min}(b, b') \\ &= \emptyset \quad \text{si } \underline{\max}(a, a') \geq \underline{\min}(b, b') \end{aligned}$$

L'analyse statique se termine en résolvant les équations

# EXEMPLE : RÉSOLUTION ITÉRATIVE DES ÉQUATIONS

## Principe :

- Système d'équations :

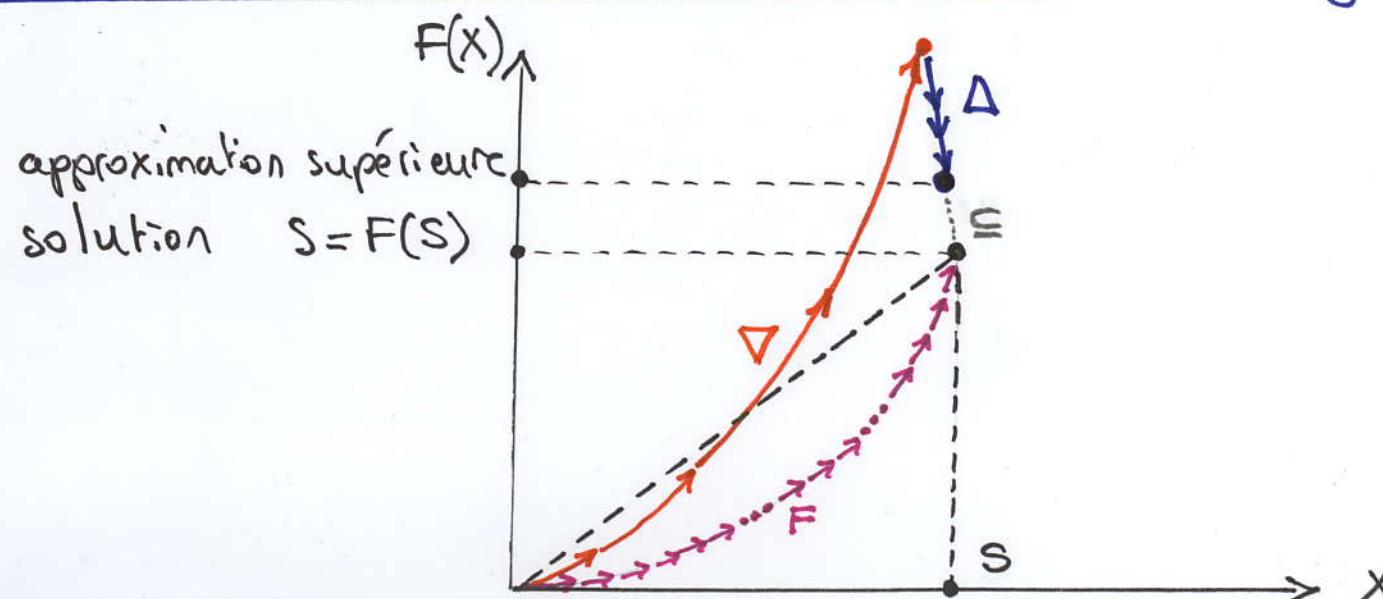
$$X = F(X) \quad \text{ou} \quad \begin{cases} X_l = f_l(x_1, \dots, x_n) \\ l = 1, \dots, m \end{cases}$$

- Résolution itérative :

$$x^0 = \emptyset, \quad x^1 = F(x^0), \quad \dots, \quad x^{k+1} = F(x^k), \quad \dots \quad s = \lim_{k \rightarrow \infty} x^k \quad (\text{limite})$$

- En général non convergent.

## Résolution itérative avec accélération de la convergence par extrapolation :



# RÉSULTAT DE L'ANALYSE STATIQUE D'INTERVALLES

Visualizer

quit    clocks    trees    octagons    filters    help

Search string: / Next Previous First Last Goto line: /

decrement.c

```
typedef signed int INTEGER;
volatile INTEGER INTEGERinput;
INTEGER X;
void main()
{
    X = INTEGERinput;
    while (X >= 0) {
        @X = X - 1;/*
    }
}
```

info

```
/* Analyzer launched at 2003/ 5/25 11:02:27
Command line was "/Volumes/PB_Cousot/Projet/absinthe2/analyzer.opt --unroll 0 --no-clock --no-octagon --trace --exec-fn main --export-invariant-stat decrement-int.invariant.bin decrement.c"
Launched by "cousot" on "<unknown>"
Executable "analyzer.opt" created at 2003/ 5/24 21:03:28 */
```

context: main 8:15  
variables: X (1)  
invariant:  
[<<X in [0, 2147483646]>; <>;  
]  
main

QUELQUES IDÉES FORMALISÉES  
PAR L'INTERPRÉTATION ABSTRAITE

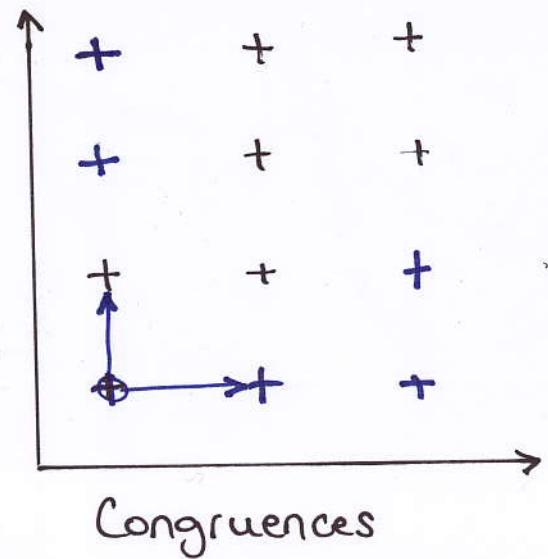
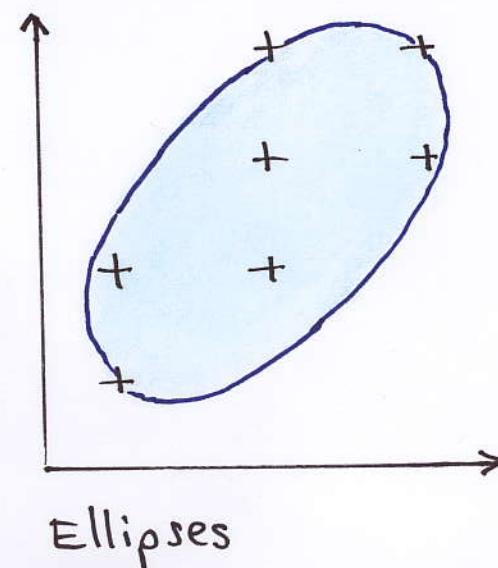
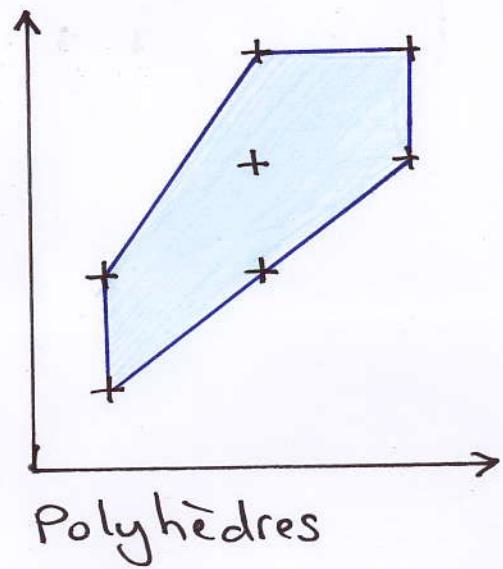
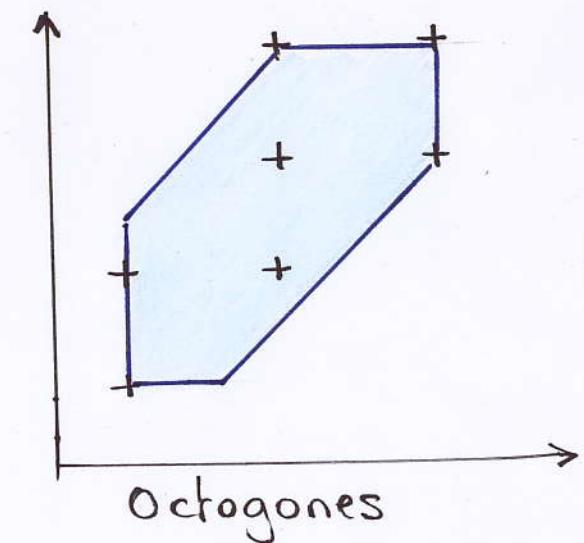
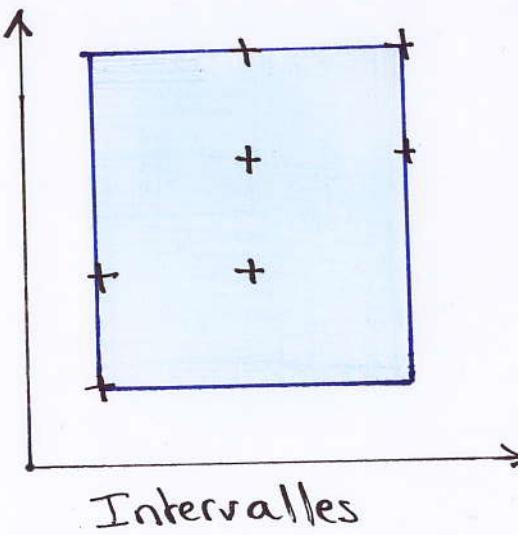
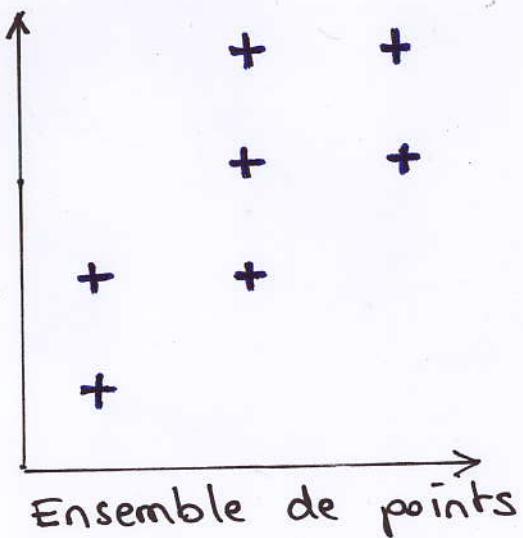
## APPROXIMATION DES PROPRIÉTÉS

Toute preuve de programme implique de connaître une approximation des propriétés de la sémantique du programme :

- fournie sous forme d'un invariant induitif pour les méthodes déductives
- fournie sous forme de modèle du programme pour les méthodes exhaustives
- calculée explicitement par l'analyseur statique

L'interprétation abstraite formalise essentiellement cette idée d'approximation de la sémantique (i.e. des comportements à l'exécution dans tous les environnements possibles) des programmes d'un langage de programmation.

## EXEMPLES D'APPROXIMATION DE PROPRIÉTÉS



# REPRÉSENTATION DES PROPRIÉTÉS : DOMAINES ABSTRAITS

Représentations universelles des propriétés :

L'utilisation d'une seule représentation universelle des propriétés des programmes est :

- soit trop limitée (BDDs à tout faire en vérification exhaustive)
- soit algorithmiquement complexe (Prédicats du premier ordre pour les méthodes déductives)

Domaines abstraits :

- Représentation machine efficace de classes de propriétés (ex : intervalles, octogones, ...), algorithmique spécialisée très efficace
- Possibilités de composition pour exprimer des propriétés complexes (ex : produit réduit).

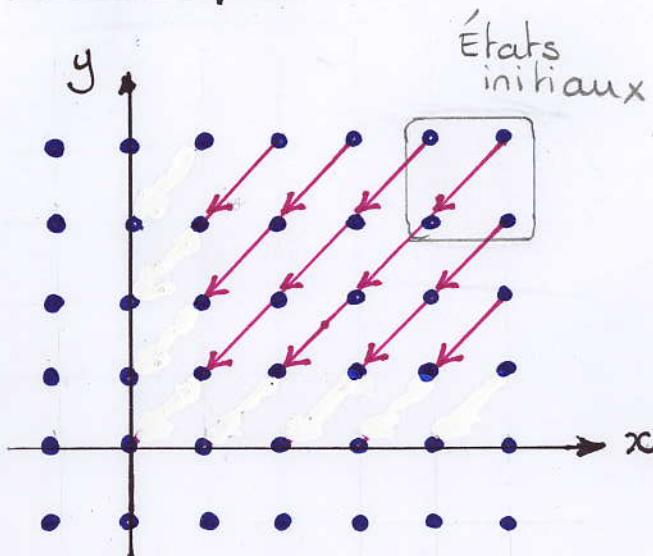
C'est l'objet de la théorie des domaines abstraits, implantation sous forme de bibliothèques.

# L'APPROXIMATION COMPLÈTEMENT DES PROPRIÉTÉS DÉTERMINE L'APPROXIMATION DE LA SÉMANTIQUE

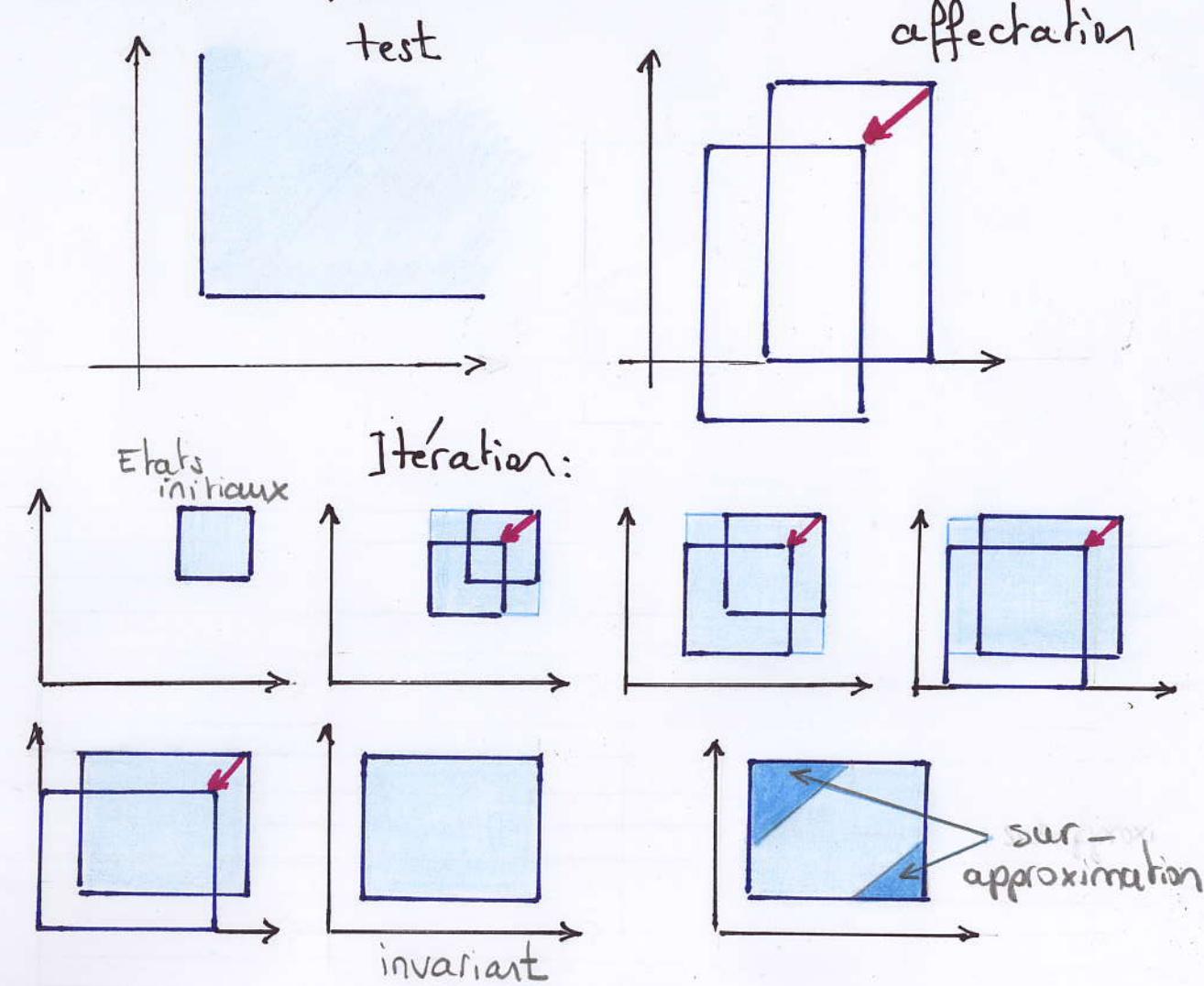
Programme :

```
while (x>1 & y>1) {
    x=x-1; y=y-1;f
```

Sémantique :



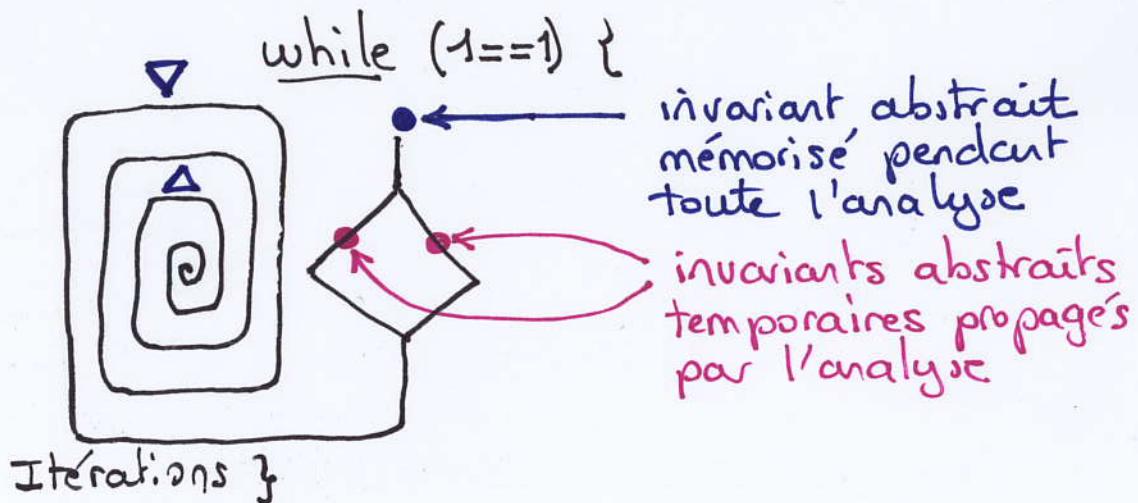
Sémantique approchée :



# PROBLÈMES DE COÛT

## - Coût mémoire :

- Éviter de mémoriser des informations inutilisées
- Itérations chaotiques.



## - Coût calcul :

- Abstraction infinie : possibilité de divergence des itérés
- Accélération de la convergence par extrapolation (approximation supérieure par élargissement  $\nabla$  améliorée par rétrécissement  $\Delta$ )
- La théorie de l'interprétation abstraite montre que l'usage de domaines abstraits infinis avec accélération de la convergence est strictement plus puissante que des abstractions finies.

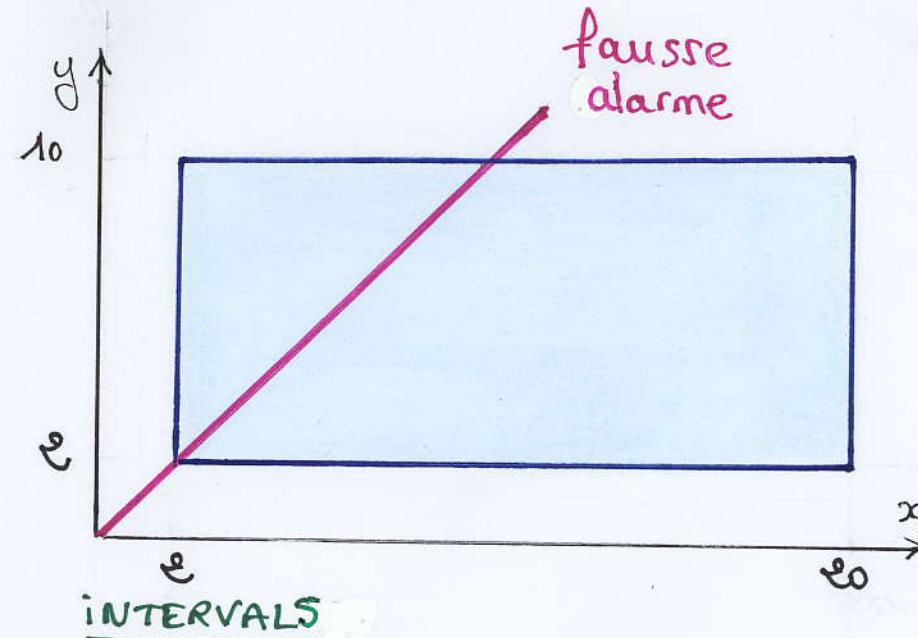
## PROBLÈME DE PRÉCISION

- Si l'analyseur statique n'a pas de domaine abstrait exprimant les propriétés de la sémantique du programme strictement nécessaires pour faire la preuve:
  - La théorie de l'interprétation abstraite montre que, dans ces conditions, la preuve est impossible
    - fausse alarme
- Si la preuve est faisable, la théorie de l'interprétation abstraite montre qu'il existe une abstraction qui convient (assez précise, calculable):
  - choix d'un domaine abstrait convenable par l'utilisateur (dans une palette offerte par l'analyseur)
  - raffinement de l'analyseur (par le concepteur qui connaît un domaine abstrait plus précis).

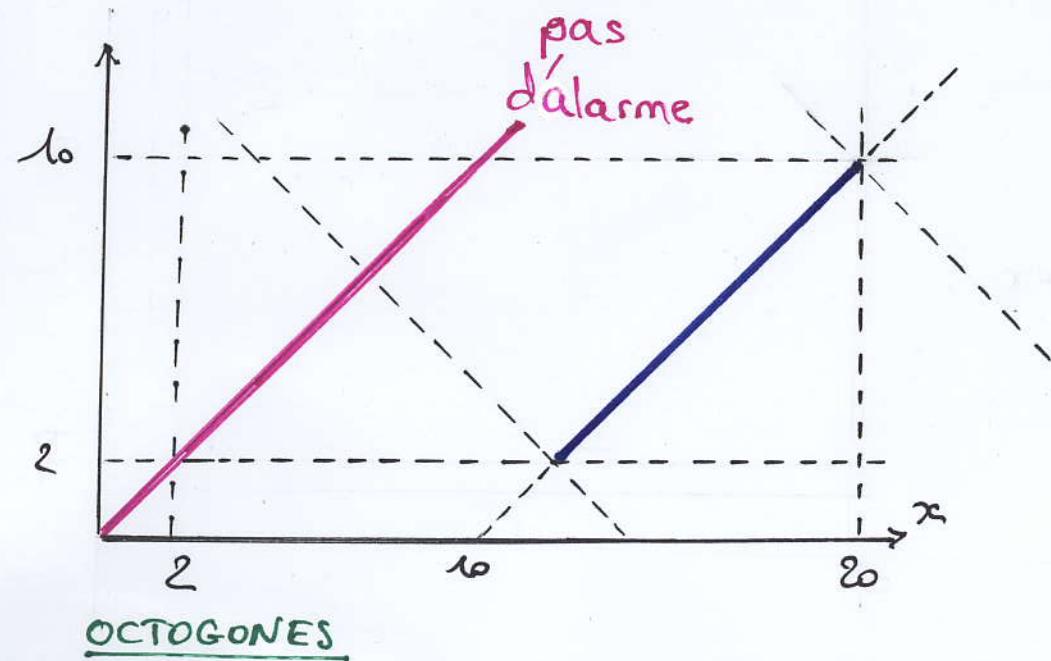
# EXEMPLE DE CHOIX POSSIBLE DE DOMAINE ABSTRAIT

```
typedef signed int INTEGER;  
INTEGER x, y;  
void main()  
{  
    x = 20; y = 10;  
    while ((x > 1) && (y > 1)) {  
        assert (x != y);  
        x = x - 1;  
        y = y - 1;  
    };  
}
```

Le choix systématique  
d'abstractions très  
précises serait trop  
coûteux

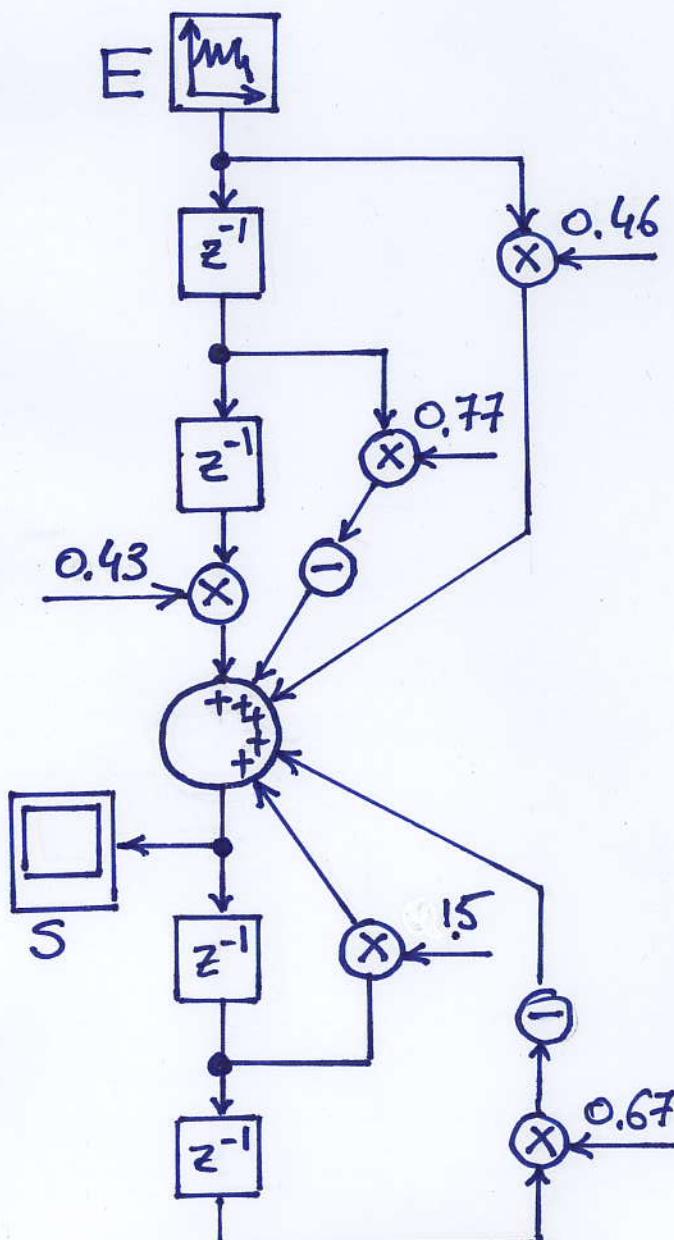


INTERVALS



OCTOGONES

# EXEMPLE DE RAFFINEMENT NÉCESSAIRE



Filtre de traitement du signal en SIMULINK.

```

typedef enum {FALSE = 0, TRUE = 1} BOO;
typedef float NUM;
typedef struct {NUM E; NUM S;} T;

static volatile NUM NUM_input;

static NUM E;
static NUM S;
static volatile BOO B;

void main()
{
    while ((1 == 1))
    {
        /* Simulate the global invariant on E */
        E = NUM_input;
        assert (-15.5 <= E && E <= 15.5);
        /* E in [-15.5, 15.5] */
        {
            static T R[2];
            if (!B)
            {
                R[0].S = E;
                S = E;
                R[0].E = E;
            }
            else
            {
                S = (((((0.46 * E) - (R[0].E * 0.77))
                        + (R[1].E * 0.43)) + (R[0].S * 1.5))
                      - (R[1].S * 0.67));
            }
            R[1].E = R[0].E;
            R[0].E = E;
            R[1].S = R[0].S;
            R[0].S = S;
        }
    }
}

```

Programme synchrone en C.

## DIFFICULTÉS :

### - Méthodes déductives :

- Pas de démonstrateurs de théorèmes pour les flottants (tenant compte des arrondis)

### - Méthodes exhaustives :

- Système d'états infinis (ou très grands)
- Tout modèle fini n'est valable que pour un jeu de coefficients particuliers (et donc à refaire à chaque fois)

### - Analyseurs statiques :

- Les domaines abstraits généraux sont trop imprécis  
⇒ Possibilité de concevoir une abstraction spécialisée pour les filtres, qui soit générique (instanciable pour des coefficients quelconques), donc nécessairement infini.

**Visualizer**

quit | clocks | trees | octagons | filters | help

Search string: / Next Previous First Last Goto line: 29 /

filtre.c

```

@E = NUM_input;
@assert (-15.5 <= E && E <= 15.5);
/* E in [-15.5, 15.5] */
*{
    static T R[2];
    @if (!B)
    {
        @R[0].S = E;
        @S = E;
        @R[0].E = E;@
    }
    else
    {
        @S = (((((0.46 * E) - (R[0].E * 0.77))
                  + (R[1].E * 0.43)) + (R[0].S * 1.5))
                  - (R[1].S * 0.67));@
    }
    @if (1) E = R[0].E;
}

```

info warning

main 17:13-17:45  
assert failure:

invariant:  
[<s in [-16.2981, 16.2981]>]

FILTRE:

```

Var_entree_1      :R[0].E
Var_entree_2      :R[1].E
Var_sortie        :S
Var_sortie_pred   :R[1].S
coef_e1           :[0.46 ; 0.46]
coef_e2           :[-0.77 ; -0.77]
coef_e3           :[0.43 ; 0.43]
coef_a            :[1.5 ; 1.5]
coef_b            :[-0.67 ; -0.67]
Egalite des entrees a l'origine!!
Nb de deroulement : 41
plus_grande_entree : <= 15.5
erreur_en_entree  : <= 2.59140556478e-05
gain_1ieres_sorties : <= 1.05070118277
gain_last_entrees : <= 1.05045071664
gain autres entrees : <= 0.00113947520253
erreur_sortie     : <= 0.000356692013453
sortie_max         : <= 16.3000046656

```

$$E \in [-15.5, 15.5]$$



$$S \in [-16.2981, 16.29]$$

CATÉGORIES  
D'ANALYSEURS STATIQUES

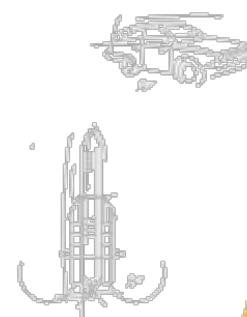
## ANALYSEURS GÉNÉRAUX

- Dédiés à un langage de programmation général
- Pas d'hypothèses particulières sur les programmes à analyser, ni sur les environnements d'exécution
- Propriétés à vérifier généralistes, ayant un grand nombre d'applications potentielles (RTE, WCET)
- Abstractions généralistes, choisies par le concepteur de l'analyseur, peu de facilités pour ajuster le ratio coût / précision
- Enièrement automatiques, temps d'exécution souvent grand
- Le problème principal est l'imprécision :
  - Peut être acceptable, voire voulue, si l'est possible de conclure avec une information imparfaite (Ex: WCET)
  - Peut être inacceptable pour une preuve de correction (Ex: typiquement 5 à 15 % de cas d'incertitude pour les RTE).
- Rentabilité commerciale par diffusion massive.

# PolySpace

TECHNOLOGIES

- [!\[\]\(4d9c6fd0f6d4550c5067cf48debcceaa\_img.jpg\) PRODUCT DATASHEETS](#)
- [!\[\]\(757fbee3f843dd9f37076018ae289d1b\_img.jpg\) DOWNLOADS](#)
- [!\[\]\(fd8c39a335a6d79ebbb8980182a2ea4d\_img.jpg\) SAVINGS](#)
- [!\[\]\(951ba56395c53fba3db2838aca5ded0c\_img.jpg\) REFERENCES](#)
- [!\[\]\(5afb28a9bb741ebee85df068181dc94f\_img.jpg\) FAQ](#)



## NEXT GENERATION DEVELOPMENT TOOLS FOR EMBEDDED APPLICATIONS

The first solution to automatically detect run-time errors before compilation without testing

Based on Abstract Interpretation, PolySpace Technologies provides the earliest detection of run-time errors, dramatically reducing testing and debugging costs with :

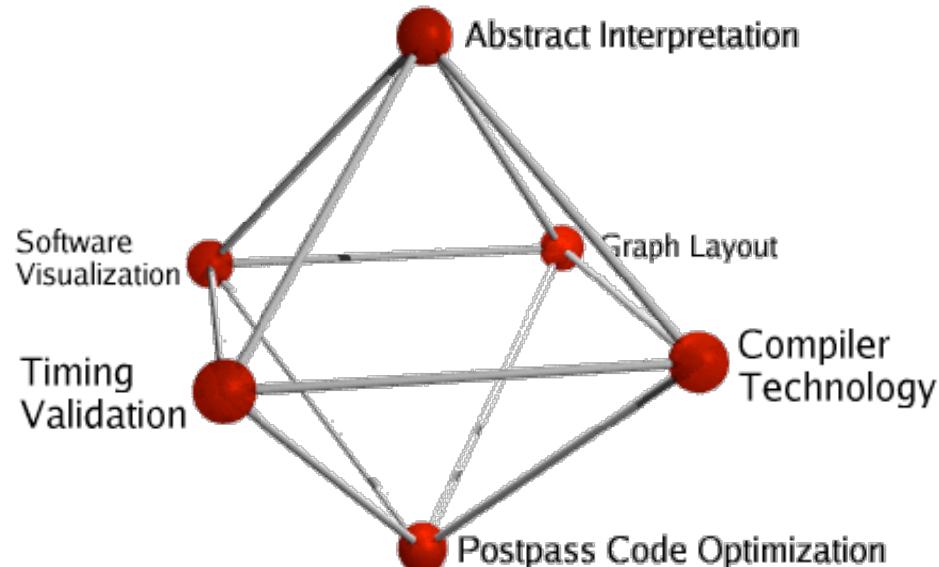
- No Test Cases to Write
- No Code Instrumentation
- No Change to your Development Process
- No Execution of your Application

[LEARN MORE](#)

April 23 2003

PolySpace releases PolySpace Developer Edition

[Learn More](#)



**NEW:** StackAnalyzer  
for PowerPC and HC12



**NEW Product Line**  
aiT WCET Analyzers

Abstract Interpretation - Timing Validation  
Compiler Technology - Software Visualization  
Graph Layout - Postpass Code Optimization

[Company](#) - [Products](#) - [Sitemap/Search](#)

## ANALYSEURS SPÉCIALISÉS

- Dédiés à un type d'application particulier
- Prise en compte des particularités des programmes à analyser et de leur environnement d'exécution
- Dédiés à un type particulier de propriétés à vérifier (RTE / WCET) dans le contexte d'utilisation (ex: prise en compte des règles de programmation de l'utilisateur)
- Abstractions généralistes raffinées par le concepteur de l'analyseur pour s'adapter au meilleur rapport coût / précision pour la famille de programmes considérée (avec possibilité de paramétrage par l'utilisateur pour prendre en compte les variations possibles entre membres de la famille)
- Analyse automatique, paramétrisation possible, précis et efficace
- Le problème principal est la compétence technique requise pour développer les analyseurs spécialisés
- Artisanat de très haute technologie qui ne serait rentable que pour des applications extrêmement critiques.

QUELQUES THÈMES DE RECHERCHE  
EN INTERPRETATION ABSTRAITE

# QUELQUES THÈMES DE RECHERCHE EN INTERPRÉTATION ABSTRAITE

## - Fondements :

- Sémantique des langages de programmation
- Analyse d'aspects complexes des langages de programmation (récursivité d'ordre supérieur, héritage dans les langages objets, mobilité)
- Techniques de résolution de contraintes / équations

## - Propriétés abstraites :

- Propriétés de sûreté (safety) complexes (histoire des calculs)
- Propriétés de vivacité (liveness)
- Propriétés probabilistes
- Combinaison (automatique) de domaines abstraits
- Décomposition optimale de propriétés complexes
- Conception de domaines abstraits (numériques, symboliques)

### — Interaction :

- Interaction avec l'utilisateur pour le raffinement d'analyses
- Détermination automatique des sources de perte de précision
- Aide à la détection de l'origine des fausses alertes

### — Conception d'analyseurs :

- Analyseurs modulaires, adaptables par paramétrisation
- Analyses modulaires séparées (modules de programmes, modules de bibliothèques, décomposition de grands programmes en parties analysées séparément)
- Automatisation de la preuve de correction de l'analyseur par référence à une sémantique

— ...

## RÉFÉRENCES

## RÉFÉRENCES

P. COUSOT

Abstract Interpretation Based Formal Methods and Future Challenges

In « Informatics, 10 years back — 10 years ahead »

R. Wilhelm (Ed).

Lecture notes in Computer Science 2000, pp. 138-156.

Springer Verlag, 2001

<http://www.di.ens.fr/~cousot/papers/LNCS2000-01.shtml>

Voir également :

<http://www.di.ens.fr/~cousot>