Interprétation Abstraite

Patrick COUSOT
École Normale Supérieure
45 rue d'Ulm, 75230 Paris cedex 05, France
mailto:Patrick.Cousot@ens.fr
http://www.di.ens.fr/~cousot

Journées ASPROM sur la Sûreté des Logiciels
Paris, 24–25 Oct. 2000

Motivations introductives

Références

- Cet exposé reprend en partie le texte introductif sur l'interprétation abstraite :

  P. Cousot. *Interprétation abstraite*, TSI 19(1-2-3), pp. 155–164, Hermès, Jan. 2000.
- Les compléments sur la comparaison des méthodes formelles sont à paraître dans :

  P. Cousot. *Progress on Abstract Interpretation Based Formal Methods and Future Challenges*, Schloß Dagstuhl's 10^{th} anniversary conference « Informatics – 10 Years Back, 10 Years Ahead », R. Wilhelm (Ed.), Saarbrücken, August 28-31, 2000. Lecture Notes in Computer Science n° 2000, Springer-Verlag.

Le problème de la sûreté du logiciel

- L'évolution de la puissance du matériel d'un facteur de 10^6 ces 25 dernières années a conduit à une explosion de la taille des programmes ;
- Le champ d'application des très grands logiciels va certainement se développer rapidement dans l'avenir ;
- Le champ d'application des très grands logiciels va certainement se développer rapidement dans l'avenir ;
- La taille et l'efficacité des équipes de programmation et de maintenance en charge de leur conception et de leur suivi ne peut croître dans des proportions similaires ;

Le problème de la sûreté du logiciel (suite)

- Avec un taux fréquent (et souvent optimiste) d'une bogue par millier de lignes de tels programmes énormes pourraient rapidement devenir ingérables en particulier pour les systèmes critiques ;
- Par conséquent dans les dix prochaines années, le *problème de sûreté du logiciel* va vraisemblablement devenir un problème majeur pour les sociétés modernes hautement informatisées.

Que peut-on y faire ?

- Utiliser notre cerveau (outils intellectuels) ;
- Utiliser notre ordinateur (outils mécaniques).

Vérification de programme assistée par ordinateur

- Méthodes de vérification empiriques (exécuter/simuler le programme dans suffisamment d'environnements représentatifs) :
  - Test,
  - Simulation;
- Méthodes de vérification formelles (prouver mécaniquement que les exécutions du programme sont correctes dans tous les environnements spécifiés) :
  - Méthodes déductives,
  - Vérification de modèle (« Model checking »),
  - Typage,
  - Analyse statique.

Indécidabilité et Approximation

- Comme la vérification de programmes est indécidable, les méthodes de vérification assistées par ordinateur sont toutes partielles/incomplètes ;
- Toutes impliquent une forme d'approximation :
  - limitations pratiques de complexité,
  - interaction requise avec l'utilisateur,
  - semi-algorithmes ou hypothèses de finitude,
  - restrictions sur les spécifications ou les programmes ;
- Ces approximations peuvent être formalisées par l'interprétation abstraite.

Plan

L'interprétation abstraite (en quelques mots)

- L'interprétation abstraite est une théorie ensembliste de l'approximation (du comportement de systèmes dynamiques discrets comme la sémantique des programmes) ;
- Comme ces comportements peuvent être caractérisés par des points fixes, une part essentielle de la théorie concerne les méthodes constructives effectives d'approximation et de test de points fixes par abstraction.

Référence de base

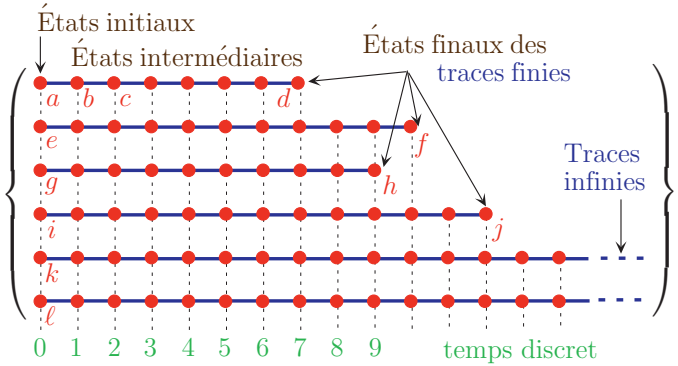
– P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'77*, Los Angeles, CA, 1977. ACM Press, pp. 238–252.

Sémantique

Sémantique : Intuition

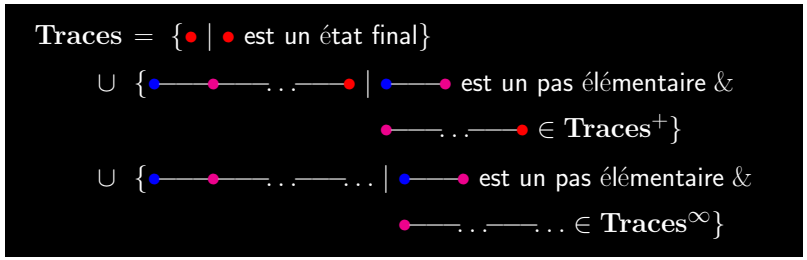
- La sémantique d'un langage spécifie la sémantique des programmes du langage ;
- La sémantique d'un programme est un modèle mathématique formel des comportements d'un ordinateur exécutant ce programme (en interaction avec n'importe quel environnement possible) ;
- Toute sémantique d'un programme peut être définie comme solution d'une équation de point fixe ;
- Toutes les sémantiques d'un programme peuvent être organisées en une hiérarchie par abstraction.

Exemple: Sémantique de traces [7 , 9]



Points fixes

Plus petit point fixe : Intuition [7 , 9]



- En général, l'équation a plusieurs solutions.
- Choisir la plus petite pour l'ordre partiel :

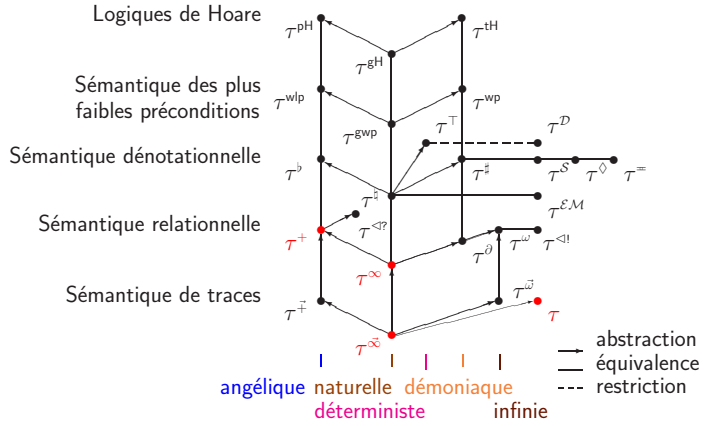
  « *plus de traces finies & moins de traces infinies* ».

Abstraction

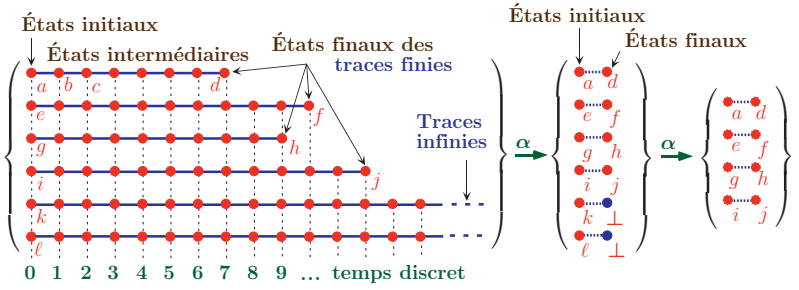
Abstraction: Intuition

- L'interprétation abstraite formalise l'idée intuitive que la sémantique est plus ou moins précise selon le niveau d'observation des exécutions du programme ;
- La théorie de l'interprétation abstraite formalise la notion d'approximation ou d'abstraction dans un cadre mathématique qui est indépendant des applications particulières.

Treillis des sémantiques [9]

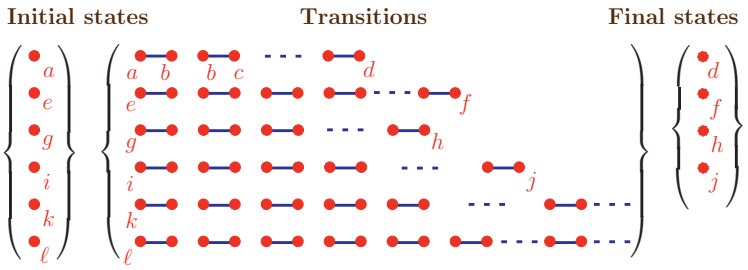


Exemple 1 d'abstraction [1]



[1] P. Cousot. *Constructive design of a hierarchy of sémantique of a transition system by abstract interpretation.* To appear in TCS (2000).

Exemple 2 d'abstraction [2]

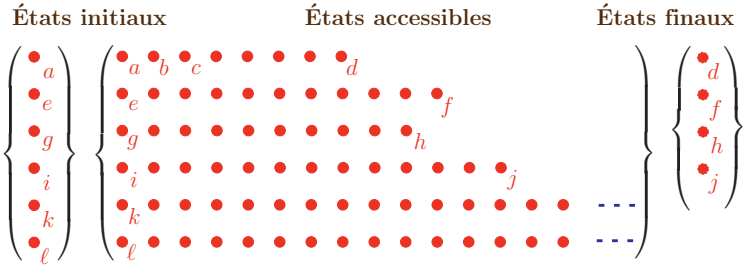


Sémantique opérationnelle (des petits pas)

[2] P. Cousot. *Constructive design of a hierarchy of sémantique of a transition system by abstract interpretation.* To appear in TCS (2000).

Abstractions effectives

Exemple 3 d'abstraction [3]



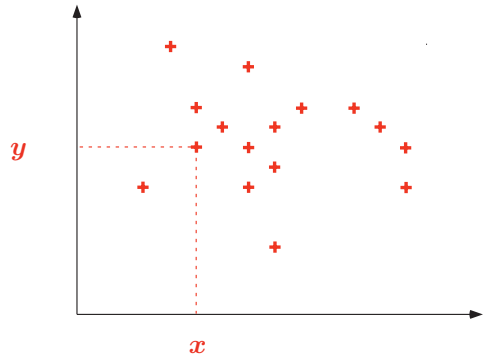
Correction partielle / Sémantique de l'Invariance

[3] P. Cousot. *Constructive design of a hierarchy of sémantique of a transition system by abstract interpretation.* To appear in TCS (2000).

Abstractions effectives

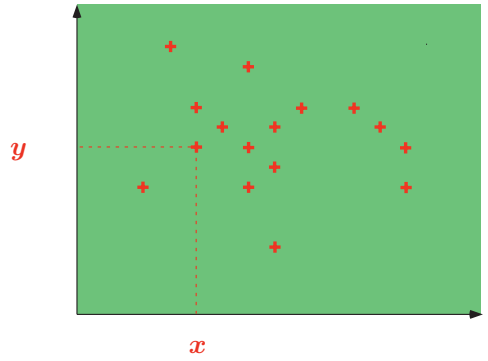
- Si l'approximation est suffisamment grossière, l'abstraction de la sémantique peut conduire à une version moins précise mais qui est effectivement calculable par ordinateur ;
- Le calcul de cette sémantique abstraite revient à une résolution itérative effective d'équations de point fixe ;
- Par calcul effectif de la sémantique abstraite, l'ordinateur est capable d'analyser le comportement des programmes et des logiciels avant toute exécution. [10].

Exemples d'abstractions effectives d'un ensemble [in]fini de points ;



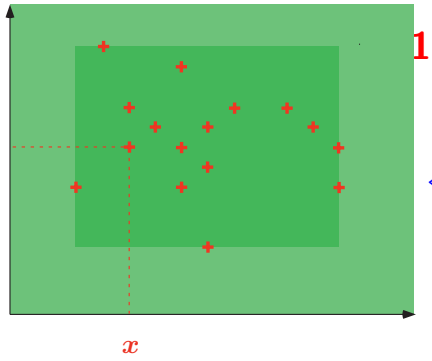
$$\{\ldots, \langle 19,\ 88\rangle, \ldots, \langle 19,\ 99\rangle, \ldots\}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 1 : signes [12]



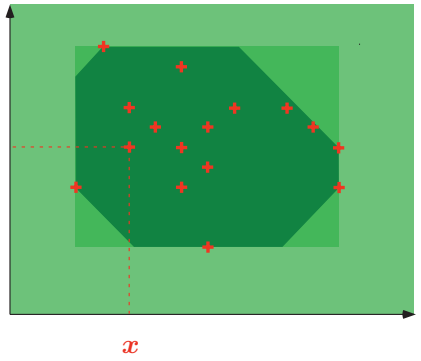
$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 2 : intervalles [10, 11]



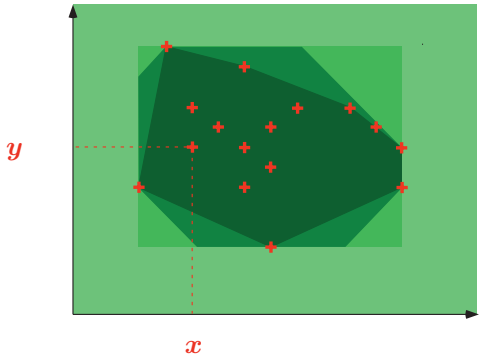
$$\begin{cases} x \in [19,\ 88] \\ y \in [19,\ 99] \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 3 : octogones



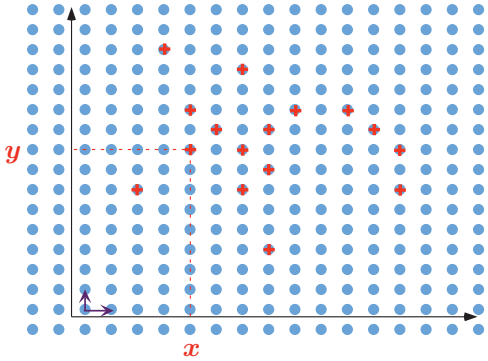
$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 88 \\ 1 \leq y \leq 9 \\ x - y \leq 99 \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 4 : polyèdres [15]



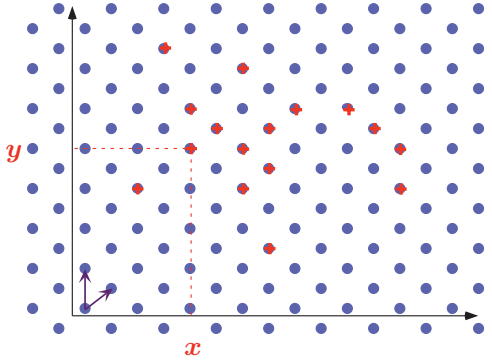
$$\begin{cases} 19x + 88y \leq 2000 \\ 19x + 99y \geq 0 \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 5 : congruences simples [17]



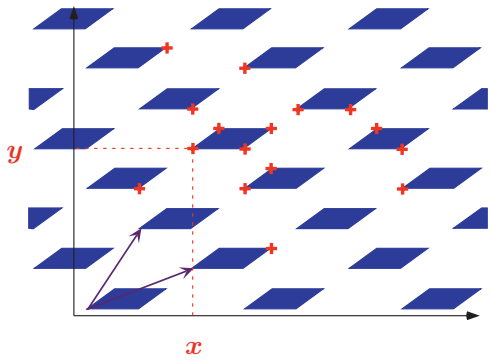
$$\begin{cases} x = 19 \bmod 88 \\ y = 19 \bmod 99 \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 6 : congruences linéaires [18]



$$\begin{cases} 1x + 9y = 8 \bmod 8 \\ 1x - 9y = 9 \bmod 9 \end{cases}$$

Exemples d'abstractions effectives d'un ensemble [in]fini de points ; Exemple 6 : congruences trapézoïdales linéaires [20, 21]



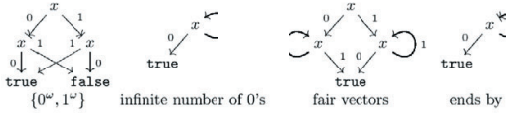
$$\begin{cases} 1x + 9y \in [0, 88] \bmod 10 \\ 1x - 9y \in [0, 99] \bmod 11 \end{cases}$$

Abstractions effectives de structures symboliques

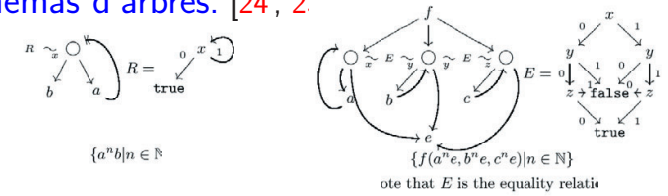
- La plupart des structures manipulées par les programmes sont des *structures symboliques* telles que structures de contrôle (graphes d'appel), structures de données (arbres de recherche), structures de communication (programmes distribués & mobiles), etc;
- Il est très difficile de trouver des abstractions compactes et expressives de tels ensembles d'objets (langages, automates, arbres, graphes, etc.).

Exemples d'abstractions d'ensembles infinis d'arbres infinis

Graphes de décision binaires : [22]



Schémas d'arbres: [24, 2?]



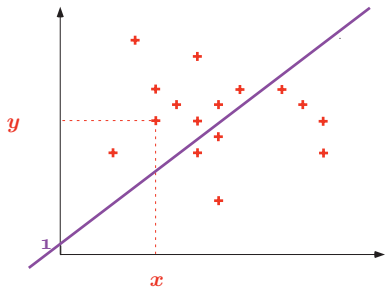
Perte d'information

Perte d'information

- Toutes les réponses données par la sémantique abstraite sont toujours correctes par rapport à la sémantique concrète ;
- À cause de la perte d'information, toutes les questions ne peuvent recevoir de réponse définitive en utilisant la sémantique abstraite ;
- Les sémantiques les plus concrètes peuvent répondre à plus de questions ;
- Les sémantiques les plus abstraites sont plus simples.

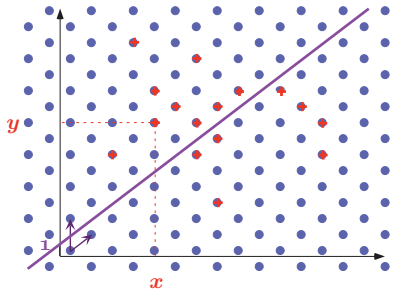
Exemple de perte d'information

- L'opération `1/(x+1-y)` est-t-elle bien définie à l'exécution?
- Sémantique concrète : **oui !**



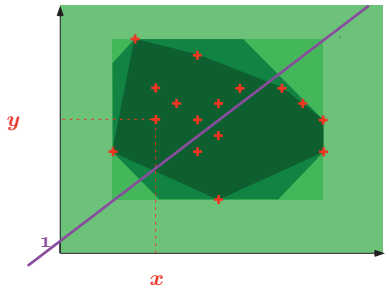
Exemple de perte d'information

- L'opération `1/(x+1-y)` est-t-elle bien définie à l'exécution?
- Sémantique abstraite 2 : **oui !**



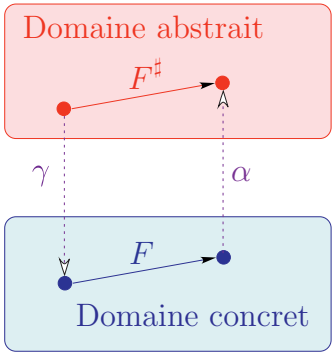
Exemple de perte d'information

- L'opération `1/(x+1-y)` est-t-elle bien définie à l'exécution?
- Sémantique abstraite 1 : **je ne sais pas !**



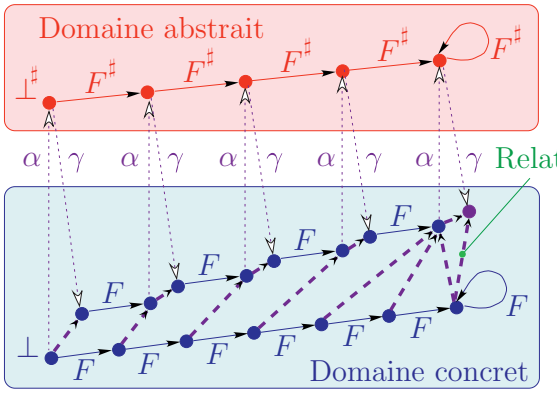
Abstraction de point fixe

Abstraction de fonction



$$F^\sharp = \alpha \circ F \circ \gamma$$

Abstraction de point fixe



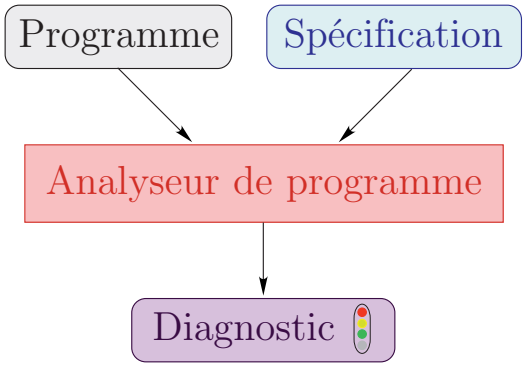
$$lfp\, F \sqsubseteq \gamma(lfp\, F^\sharp)$$

Analyse statique de programmes

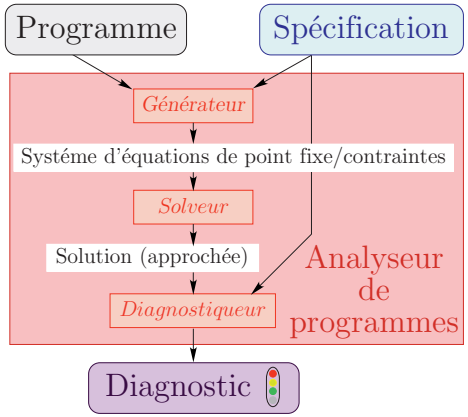
Objectif de l'analyse de programmes

- L'analyse statique de programme est la détermination statique automatique de propriétés dynamiques de l'exécution des programmes;
- Le principe est de calculer une sémantique approchée du programme pour tester une spécification donnée ;
- L' *interprétation abstraite* est utilisée pour dériver, à partir d'une sémantique concrète, une sémantique abstraite approchée calculable;
- Cette dérivation formelle n'est pas (complètement) mécanisable.

Objectif de l'analyse de programmes



Principe de l'analyse de programmes



Exemple : analyse d'intervalles (1975)[4]

Programme à analyser :

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Générateur

Équations (interprétation abstraite de la sémantique) :

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

L'analyseur lit le texte du programme et engendre une représentation des équations puis les résout itérativement. Les équations sont une abstraction de la sémantique de traces du programme. La dérivation formelle de l'algorithme produisant les équations par interprétation abstraite de la sémantique de traces est (essentiellement) manuelle.

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Générateur

Itération chaotique croissante, initialisation:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = \emptyset \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante, itération:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante, itération:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante, itération:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante, itération:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante, itération:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ?**

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ??**

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ???**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ?????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ??????**

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique croissante : **convergence ???????**

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Accélération de la convergence par extrapolation :

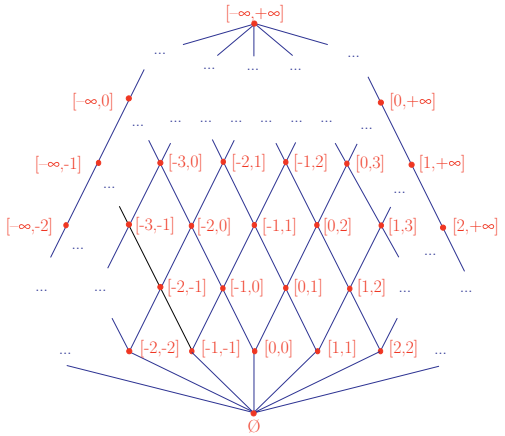
```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,+\infty] \Leftarrow \text{élargissement} \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Élargissement



Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique décroissante :

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,+\infty] \\ X_3 = [2,+\infty] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique décroissante :

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, +\infty] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique décroissante :

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = \emptyset \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Itération chaotique décroissante, solution finale:

```
   x := 1;
1:
   while x < 10000 do
2:
        x := x + 1
3:
   od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = [10000, 10000] \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Solveur

Résultat de l'analyse d'intervalles :

```
   x := 1;
1: {x = 1}
   while x < 10000 do
2: {x ∈ [1,9999]}
        x := x + 1
3: {x ∈ [2,10000]}
   od;
4: {x = 10000}
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, 9999] \\ X_3 = [2, 10000] \\ X_4 = [10000, 10000] \end{cases}$$

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Exemple : analyse d'intervalles (1975)[4] — Diagnostiqueur

Exploitation du résultat de l'analyse d'intervalles :

```
   x := 1;
1:
   while x < 10000 do
2:
       x := x + 1   ⟵ pas de débordement
3:
   od;
4:
```

[4] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Quelques applications ...

- « Data flow analysis», « set-based analysis », etc. pour l'optimisation et la transformation de programmes (y compris l'évaluation partielle) [12, 14] ;
- L'inférence de type (y compris pour les systèmes indécidables/« soft typing » [8]) ;
- « Abstract model-checking » de systèmes infinis [13, 14] ;
- Test & déboguage abstrait [5, 2];
- Analyses probabilistes [26] ; .../...

Quelques applications ... (suite)

- Analyse de la topologie de communication pour le code distribué/mobile [28] ;
- Différentiation automatique de programmes numériques ;
- Tatouage sémantique de logiciel ;
- ... ;

Une application impressionnante (1996/97)

- L'*interprétation abstraite* a été utilisée pour l'analyse statique du code ADA embarqué du lanceur Ariane 5 [5]; [19]
- Détection automatique de toutes les erreurs à l'exécution certaines ●, potentielles ●, impossibles ● ou inaccessibles ● [6];
- Découverte automatique de l'erreur de programmation du vol 501 ;
- Succès pour les vols 502 & 503 et l'ARD [7].

[4] Logiciel de vol (60.000 lignes de code ADA) et Unité de Mesure Inertielle (30.000 lignes de code ADA).
[5] comme les débordements scalaires et flottants, les erreurs d'indexation des tableaux, les exceptions arithmétiques comme les divisions par zéro, les variables non initialisées, l'exclusion mutuelle sur les données partagées, etc.
[6] Atmospheric Reentry Demonstrator : module retournant sur terre.

Industrialisation de l'analyse statique par interprétation abstraite

- Connected Components Corporation (U.S.A.), L. Harrison, 1993;
- AbsInt Angewandte Informatik GmbH (Allemagne), R. Wilhelm & C. Ferdinand, 1998;
- Polyspace Technologies (Grenoble), A. Deutsch & D. Pilaud, 1999.

Méthodes formelles abstraites

Le problème de la vérification ultime

- Trouver la dernière erreur dans un logiciel ;
- Les méthodes formelles abstraites peuvent-t-elle résoudre le problème de la vérification ultime

Analyse de programme : inconvénients

- Peut analyser de grands programmes (220 000 lignes de C) sans aucune intervention du programmeur mais les spécifications sont simples ;
- La sémantique des langages de programmation est très complexe et donc son abstraction l'est également ;
- L'abstraction donc la conception de l'analyseur est manuelle (et au delà des capacités du programmeur moyen) ;
- Les analyses possibles doivent être choisie dans une palette prédéfinie, l'extension de l'analyseur étant réservée à des spécialistes ;

.../...

Analyse de programme : inconvénients (suite)

- Les erreurs peuvent être expliquées par des contre-exemples abstraits (et plus difficilement par des contre-exemples concrets) ;
- Les 5 à 10 % de cas d'incertitude doivent être traités par d'autres méthodes empiriques ou formelles (y compris des abstractions plus fines).

Typage : fondements [16, 25]

- Analyses décidables, par restriction à la fois sur les spécifications (types autorisés) et sur les programmes ;
- Présentation claire de l'analyse de type (algorithme d'inférence) grâce à un système formel logique équivalent (vérification de type) ;
- Étendu aux structures de données complexes (références), au polymorphisme, aux exceptions et aux modules séparés d'une façon élégante qui passe à l'échelle pour les très grands programmes ; .../...

Typage : fondements [16, 25] (suite)

- Le typage étant intégré dans le compilateur, la certification peut être conduite jusqu'au niveau du code engendré (« proof-carrying code », « certified compiler », etc);

Typage : inconvénients

- Les systèmes de types (e.g. avec un sous-typage subtile) peut être très complexe à comprendre pour l'utilisateur moyen ;
- Compositionnel mais pas complètement abstrait (le même code polymorphe a des types différents dans des contextes différents) ;
- Interaction simpliste avec le programmeur (le programmeur ne reçoit aucune indication sur l'impossibilité de typer des programmes incorrects (ou corrects!) et ne peut fournir d'indications pour aider l'algorithme de typage) ; .../...

Typage : inconvénients (suite)

- Les programmes considérés sont à la fois complexes (ordre supérieur) et trop restreints (principalement les langages fonctionnels voire objet) ;
- Les restrictions sur les propriétés considérées sont sévères (les erreurs arithmétiques, de débordement, de déréférençage de pointeurs nuls, ... sont testées à l'exécution, toutes les propriétés de vivacité sont simplement ignorées);
- le codage des types comme des termes/formules et l'approximation des points fixes au bout d'une itération rendent la généralisation à des propriétés expressives très difficile voire impossible.

Typage : inconvénients (fin)

- la spécification logique du système de type est souvent inexistante dans le manuel de référence, non équivalente à l'algorithme d'inférence ou tellement inextricable qu'elle est inutile à la fois pour le programmeur et l'écrivain du compilateur.

« Model checking » : fondements [3, 4, 27]

- Utilise un modèle du programme (c'est-à-dire une abstraction conçue manuellement de la sémantique du programme);
- Utilise une spécification du programme fournie par le programmeur (dans une logique temporelle très expressive) ;
- Test la spécification par recherche/exploration exhaustive de l'espace d'états du modèle ;
- Grand succès grâce à la conception de structures de données (par exemple les BDDs) et d'algorithmes (par exemple SAT) très astucieux pour représenter de très grands ensembles de vecteurs de booléens et leurs transformations.

« Model checking » : inconvénients

- Ne passe pas à l'échelle (a gagné un facteur 100 en 10 ans dont 10 grâce au matériel) ;
- L'abstraction de la sémantique du programme en un modèle est souvent triviale, manuelle et/ou non vérifiée ;
- Le modèle doit être ultimement fini (pour permettre une recherche exhaustive) ;
- L'abstraction manuelle est complète mais l'spécifique pour un programme et donc non réutilisable ;
- Utilisé le plus souvent comme un outil de mise au point plutôt que de vérification.

Méthodes déductives : fondements

- Utilise (une abstraction conçue manuellement) de la sémantique du programme pour obtenir des conditions de vérification minimales pour démontrer la correction du programme ;
- Utilise un démonstrateur de théorème ou un assistant de preuve pour démontrer que les conditions de vérification sont satisfaites.

Méthodes déductives : inconvénients

- Un argument inductif (par exemple un invariant, une fonction de terminaison, etc.) doit être découvert, généralement par l'utilisateur ;
- Seule la vérification d'une preuve existante peut être (partiellement) automatisée (difficile de trouver la preuve automatiquement) ;
- Les conditions de vérification sont parfois incorrectes, essentiellement pour simplifier le vérificateur (par exemple alias, arithmétique modulaire, etc.) ; .../...

Méthodes déductives : inconvénients (suite)

- La taille de la preuve est souvent exponentielle en la taille du programme ;
- La mise au point d'une preuve infructueuse est complexe au moins autant que la mise au point du programme ;
- L'interaction avec le prouveur est difficile sinon désespérante ;
- Les démonstrateurs de programmes sont instables dans le temps (par exemple les changement de stratégies de preuves remettent en cause les preuves anciennes) ; .../...

Méthodes déductives : inconvénients (fin)

- Le codage des propriétés par des termes ou formules syntaxiques est uniforme (de sorte que par exemple les BDDs sont difficilement encodables efficacement) ;
- Inadéquats pour le calcul de points fixes (seulement pour le test de points fixes) ;
- Pas d'outil pour mécaniser l'abstraction.

Aucune méthode formelle ne peut à elle seule résoudre le problème de la vérification ultime.

Tendance actuelle : combinaison de méthodes

- **Abstraction conçue par l'utilisateur:** dériver un modèle abstrait fini de la sémantique du programme par interprétation abstraite, démontrer la correction de l'abstraction par des méthodes déductives et vérifier ensuite le modèle abstrait par « model-checking »;
- **Limitation fondamentale [1]:** 1°) découverte de l'abstraction 2°) dérivation de la sémantique abstraite sont logiquement équivalents à faire la preuve ! (resp. 1°) découverte de l'invariant & 2°) vérification de l'invariant)

Référence

[1] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, eds, *Proc. 4th Int. Symp. on Abstraction, Reformulations and Approximation, SARA '2000*, Horseshoe Bay, TX, USA, LNAI 1864, pp. 1–25. Springer-Verlag, 26–29 July 2000.

Aucune combinaison de méthodes formelles ne peut résoudre le problème de la vérification ultime.

Une alternative possible : combiner les méthodes formelles et empiriques

Test abstrait de programmes

Test usuel	Test abstrait
Exécuter le programme sur un jeu de données	Calculer la sémantique abstraite avec une abstraction prédéfinie
Vérifier la vraisemblance	Vérifier les assertions abstraites fournies par l'utilisateur
Fournir plus de tests jusqu'à couverture suffisante	Choisir des abstractions plus fines jusqu'à la preuve des assertions ou plus d'autre abstraction.

Un petit exemple

```
0: { n:[−∞,+∞]?; f:[−∞,+∞]? }
  read(n);
1: { n:[0,+∞]; f:[−∞,+∞]? }
  f := 1;
2: { n:[0,+∞]; f:[−∞,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[−∞,+∞] }
      f := (f * n);
    4: { n:[1,+∞]; f:[−∞,+∞] }
      n := (n - 1)
    5: { n:[0,+∞]; f:[−∞,+∞] }
  od;
6: { n:[−∞,+∞]?; f:[−∞,+∞] }
sometime true;;
```

- inféré par l'analyseur statique
- erreur certaine
- pas d'erreur
- erreur potentielle
- programme de l'utilisateur
- spécification du programmeur

Un petit exemple (suite)

```
0: { n:⊥; f:⊥ }
  initial (n < 0);
1: { n:[−∞,-1]; f:Ω }
  f := 1;
2: { n:[−∞,-1]; f:[−∞,1] }
  while (n <> 0) do
    3: { n:[−∞,-1]; f:[−∞,1] }
      f := (f * n);
    4: { n:[−∞,-1]; f:[−∞,0] }
      n := (n - 1)
    5: { n:[−∞,-2]; f:[−∞,0] }
  od
6: { n:⊥; f:⊥ }
```

- inféré par l'analyseur statique
- spécification du programmeur
- programme de l'utilisateur
- pas d'erreur
- erreur potentielle
- erreur potentielle
- ⊥ code inaccessible

Conclusions

Conclusions

- La vérification des programmes par les méthodes formelles complètes (« model checking »/méthodes déductives) est très coûteuse (interaction avec l'utilisateur difficile à grande échelle) ;
- L'abstraction est impérative pour la vérification de programmes mais difficile (peu automatisable, délicate pour un programmeur moyen) ;
- L'analyse de programme basée sur des abstractions réutilisables n'exige pas d'intervention du programmeur donc financièrement rentable [8];

[7] Moins d'un franc par ligne de programme dont la production coûte de 250 à 500 F.

Conclusion (suite)

- La vérification complète de grands programmes complexes par des méthodes formelles n'est pas viable à coût raisonnable ;
- Le test de programme est encore la principale méthode de "vérification" de programmes dans l'industrie du logiciel ;
- Par similarité, l'analyse statique par interprétation abstraite peut être étendue au test abstrait de programmes ;
- L'interprétation abstraite offre des techniques puissantes qui, grâce à l'approximation, peuvent être des alternatives à la fois à la recherche exhaustive du « model-checking » et aux méthodes d'exploration partielle du test classique.

Brève bibliographie

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. PLDI*, pages 46–55. ACM Press, 1993.

[3] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, LNCS 131. Springer-Verlag, May 1981.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10^{th} POPL*, pages 117–126. ACM Press, Jan. 1983.

[5] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[6] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997. URL: http://www.elsevier.nl/locate/entcs/volume6.html, 25 pages.

[7] P. Cousot. Design of semantics by abstract interpretation, invited address. In *Mathematical Foundations of Programming Semantics, 30th Annual Conf. (MFPS XIII)*, Carnegie Mellon University, Pittsburgh, PA, US, 23–26 Mar. 1997.

[8] P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, FR, Jan. 1997. ACM Press.

[9] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, To appear (Preliminary version in [6]).

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[13] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Aut . Soft . Eng.*, 6:69–95, 1999.

[14] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

[16] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th POPL*, pages 207–212, Albuquerque, NM, Jan. 1982. ACM Press.

[17] P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.

[18] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. Int. J. Conf. TAPSOFT '91, Volume 1 (CAAP '91)*, Brighton, GB, LNCS 493, pages 169–192. Springer-Verlag, 1991.

[19] P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The ARIANE 5 example. In *Proceedings DASIA 98 – DAta Systems In Aerospace*, Athens, GR. ESA Publications, SP-422, 25–28 May 1998.

[20] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proc. ACM Int. Conf. on Supercomputing, ICS '92*, pages 226–235, Washington D.C., Jul. 1992.

[21] F. Masdupuy. Semantic analysis of interval congruences. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Academgorodok, Novosibirsk, RU, LNCS 735, pages 142–155. Springer-Verlag, 28 June – 2 Jul. 1993.

[22] L. Mauborgne. Binary decision graphs. In A. Cortesi and G. Filé, editors, *Proc. 6th Int. Symp. SAS '99*, Venice, IT, 22–24 Sep. 1999, LNCS 1694, pages 101–116. Springer-Verlag, 1999.

[23] L. Mauborgne. Tree schemata and fair termination. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 302–321. Springer-Verlag, 29 June – 1 Jul. 2000.

[24] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. In G. Smolka, editor, *Programming Languages and Systems, Proc. 9th ESOP '2000*, Berlin, DE, LNCS 1782, pages 275–289. Springer-Verlag, Mar. – Apr. 2000.

[25] R. Milner. A theory of polymorphism in programming. *J. Comput. System Sci.*, 17(3):348–375, Dec. 1978.

[26] D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 322–339. Springer-Verlag, 29 June – 1 Jul. 2000.

[27] J.-P. Queille and J. Sifakis. Verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.

[28] A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proc. 5th Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 152–167. Springer-Verlag, 1998.

FIN