# Interprétation abstraite

**Patrick Cousot**

*Département d'informatique, École normale supérieure,*
*45 rue d'Ulm,*
*75230 Paris cedex 05*
*Patrick.Cousot@ens.fr, http://www.di.ens.fr/~cousot*

RÉSUMÉ. *L'interprétation abstraite est une théorie de l'approximation discrète de sémantiques de systèmes informatiques principalement utilisée pour l'analyse et la vérification statique de logiciels.*

ABSTRACT. *Abstract interpretation is a theory of discrete approximation of the semantics of computer systems mainly applied to the static analysis and verification of software.*

MOTS-CLÉS : *Interprétation abstraite, sémantique, vérification, analyse statique.*
KEY WORDS : *Abstract interpretation, semantics, verification, static analysis.*

## 1. Introduction

Ces 25 dernières années, le matériel informatique a vu ses performances multipliées par $10^4$ à $10^6$. Il s'agit d'une révolution. Pour illustrer cet ordre de grandeur, ce rapport $10^6$ est celui qui existe entre la force de travail d'un esclave romain et la puissance d'une tranche d'une centrale nucléaire ou encore celui qui existe entre les distances Paris-Nice et Terre-Mars. La conséquence immédiate est que la taille des programmes exécutés par ces ordinateurs a cru dans des proportions similaires. Par exemple un éditeur de texte grand public moderne comprend plus de 1 700 000 lignes de C organisées en 20 000 procédures et 400 fichiers. Malheureusement, ni les capacités intellectuelles des programmeurs, ni les tailles des équipes de conception et de maintenance ne peuvent évoluer dans de telles proportions. La programmation des ordinateurs et les raisonnements sur les programmes sont difficiles. Les erreurs dans les logiciels qu'elles soient prévues (bogue de l'an 2000) ou imprévues (échec du vol 5.01 d'Ariane) sont fréquentes. Elle peuvent donc avoir des conséquences catastrophiques qui sont très coûteuses et quelquefois inadmissibles (comme dans le cas des logiciels embarqués dans les transports). Les erreurs sont d'autant plus difficiles à prévenir et retrouver que la taille des logiciels est grande voire énorme. Les méthodes manuelles classiques de validation des logiciels (revues de code, simulations,

Technique et science informatiques. Vol. 19 - n° 1/2000, pages 1 à 9

tests) ne passent pas à l'échelle. La production de logiciels fiables, leur maintenance et leur évolution au fil du temps (20 à 30 ans au minimum) est donc devenue la préoccupation fondamentale des informaticiens en ce début de millénaire. Pour faire face à leurs responsabilités, satisfaire aux probables réglementations futures et éviter que l'échec informatique devienne un problème de société important, les informaticiens doivent élargir la panoplie de méthodes et outils utilisés pour lutter contre les erreurs logicielles.

L'idée de base de l'analyse statique est d'utiliser l'ordinateur pour trouver les erreurs de programmation. La question de savoir comment programmer les ordinateurs pour qu'ils analysent le travail qu'on leur donne à faire, avant qu'ils ne le fassent effectivement est extrêmement difficile. Ceci tient à des questions d'indécidabilité et de complexité. Il faut donc se résoudre à un compromis qui consiste à ne considérer qu'une approximation des comportements possibles du programme à analyser. Par exemple le test consiste à explorer quelques exemples d'exécution que l'on espère bien choisis. La vérification de modèle (« model-checking » ) ne s'applique pleinement qu'aux systèmes finis et est généralement incomplète pour les systèmes infinis. Ces méthodes rudimentaires d'approximation sont des cas particuliers d'interprétations abstraites de sémantiques.

## 2. Modèles de calcul et sémantique

Un modèle de calcul ou d'exécution est une description mathématique formelle de la suite des opérations effectuées par un ordinateur exécutant un programme et de leurs effets internes à la machine (principalement sur la mémoire) et externes sur l'environnement (entrées/sorties par exemple), dans toutes les conditions possibles.

Un modèle simple de calcul est celui des traces maximales. À chaque pas ou instant $i$ du calcul, on note dans un état $\sigma_i$ les valeurs instantanées des composants caractéristiques de la machine (variables/mémoires, etc.) et de l'environnement (horloges, capteurs, etc.). Le programme fait donc partie de l'état $\sigma_i$ qui indique également où l'exécution est rendue dans le programme à l'instant $i$. Un calcul du programme est modélisé par une trace qui représente l'évolution des états de la machine au fur et à mesure de l'exécution du programme. Une exécution finie est modélisée par une trace finie soit une suite finie $\sigma = \sigma_0 \dots \sigma_{n-1}$ d'états (de longueur $|\sigma| = n$). Une exécution qui ne termine pas est modélisée par une trace infinie c'est-à-dire une suite infinie $\sigma = \sigma_0\sigma_1 \dots$ d'états (de longueur $|\sigma| = \infty$). En général il y a plusieurs calculs possibles (correspondant par exemple à tous les états initiaux possibles ou toutes les interactions possibles avec l'environnement) et l'on raisonne sur des ensembles de traces. On peut considérer par exemple des ensembles de traces maximales (aucune trace finie n'étant préfixe d'une trace plus longue).

La sémantique d'un programme $P$ est un modèle de calcul décrivant les exécutions *effectives* $[\![P]\!]$ de ce programme dans tous les environnements possibles. La sémantique d'un langage est donnée pour chaque programme syntaxiquement correct de ce langage. En général le logiciel est écrit dans un langage de programmation et la sémantique du logiciel est fournie par celle du langage.

Il existe de nombreuses méthodes de description de sémantiques (opérationnelle, dénotationnelle, axiomatique, etc.) qui sont équivalentes ou sont des approximations les unes des autres. Elles forment donc une hiérarchie, organisée selon la précision de la description des comportements des programmes à l'exécution, qui se comprend par interprétation abstraite [COU 00a].

Les sémantiques peuvent généralement se définir par point fixe. On peut en donner une intuition pour le cas de la sémantique de traces maximales (et par abstraction de nombreuses autres sémantiques) en observant que si l'ensemble $T$ des traces maximales engendrées par un système de transitions, c'est-à-dire une relation $t$ sur une ensemble d'états $\Sigma$, est :

— L'ensemble des traces finies $\sigma$ de longueur 1 réduites à un état final $\sigma_0$, sans transition possible ($\forall s : \langle \sigma_0, s\rangle \notin t$) ;

— L'ensemble des traces finies $\sigma$ de longueur $n > 1$ commençant par une transition $\langle \sigma_0, \sigma_1\rangle \in t$ prolongée par une trace finie $\sigma_1 \ldots \sigma_{n-1}$ de T ;

— L'ensemble des traces infinies $\sigma$ commençant par une transition $\langle \sigma_0, \sigma_1\rangle \in t$ et prolongée par une trace infinie $\sigma_1\sigma_2 \ldots$ de T ;

Formellement le point fixe $T = F(T)$ est :

$$\begin{aligned} T \quad = \quad & \{\sigma : |\sigma| = 1 \wedge \forall s : \langle \sigma_0, s\rangle \notin t\} \\ & \cup \{\sigma_0 \ldots \sigma_{n-1} : \langle \sigma_0, \sigma_1\rangle \in t \wedge \sigma_1 \ldots \sigma_{n-1} \in T\} \\ & \cup \{\sigma_0\sigma_1\sigma_2 \ldots : \langle \sigma_0, \sigma_1\rangle \in t \wedge \sigma_1\sigma_2 \ldots \in T\}\ . \end{aligned} \qquad [1]$$

En général il y a plusieurs points fixes (par exemple en ne prenant aucune trace infinie). On prend donc le plus petit point fixe pour l'ordre $X \sqsubseteq Y$ si et seulement si $X$ comporte plus de traces finies et moins de traces infinies que $Y$ (de sorte que dans la solution retenue il y a le moins de traces finies et le plus de traces infinies possibles).

## 3. Spécification et vérification

La spécification d'un logiciel est un modèle de calcul décrivant les exécutions *souhaitées* de ce logiciel dans tous les environnements possibles. La spécification minimale est l'absence d'erreurs à l'exécution (erreurs arithmétiques comme la division par zéro, dépassements de capacité comme l'accès mémoire en dehors des bornes de tableaux, etc). Des spécifications plus complexes peuvent être données par des langages de spécification comme les logiques temporelles. Ils ont également une sémantique, par exemple $\Box P$ spécifie l'ensemble des traces pour lesquelles $P$ est vrai à chaque instant et donc $[\![\Box P]\!] = \{\sigma : \forall i : 0 \leq i < |\sigma| : \sigma_i \ldots \in [\![P]\!]\}$.

La vérification d'un logiciel $P$ consiste à prouver qu'une sémantique du logiciel satisfait une spécification $S$ donnée : $[\![P]\!] \subseteq [\![S]\!]$.

Toutes les questions intéressantes relatives à la sémantique d'un programme non trivial, comme la vérification, sont indécidables : aucun ordinateur ne peut toujours y répondre exactement en un temps fini. Ceci résulte du fait qu'en général la sémantique d'un programme n'est pas calculable. Par exemple on peut définir mathématiquement la sémantique d'un programme comme solution d'une équation de point fixe mais aucun ordinateur ne peut résoudre exactement les équations en un temps fini (en particulier à cause du fait que la sémantique doit prendre en compte l'existence éventuelle de comportements infinis du programme). Une idée fondamentale est donc de considérer des *approximations*.

## 4. Interprétation abstraite

L'interprétation abstraite [COU 78, COU 77a] est une théorie de l'approximation de sémantiques de langages (de programmation ou de spécification). L'interprétation abstraite permet de formaliser l'idée qu'une sémantique est plus ou moins précise selon le niveau d'observation auquel on se place [COU 00a].

Par exemple un modèle abstrait de la sémantique de traces $T$ est la sémantique de transitions $t$ (ou sémantique opérationnelle de petits pas) qui est l'ensemble des paires $\langle\sigma_i, \sigma_{i+1}\rangle$ d'états figurant sur au moins une trace de la sémantique de traces : $t = \alpha_o(T) = \{\langle\sigma_i, \sigma_{i+1}\rangle : \sigma_0 \ldots \sigma_i\sigma_{i+1} \ldots \in T\}$. L'ensemble $\gamma_o(t)$ des traces maximales que l'on peut reconstituer à partir des transitions est l'ensemble des suites d'états reliés par une transition et dans le cas de suites finies, se terminant par un état sans transition possible : $\gamma_o(t) = \{\sigma : \forall i : 0 < i + 1 < |\sigma| : \langle\sigma_i, \sigma_{i+1}\rangle \in t \wedge |\sigma| = n \in \mathbb{N} \implies \forall s : \langle\sigma_{n-1}, s\rangle \notin t\}$.

Une autre abstraction à la base de la sémantique dénotationnelle consiste à abstraire une trace finie par la paire des états initiaux et finaux $\alpha_d(\sigma_0\sigma_1 \ldots \sigma_{n-1}) = \langle\sigma_0, \sigma_{n-1}\rangle$ tandis que les traces infinies sont abstraites par l'état initial en notant $\perp$ la non terminaison : $\alpha_d(\sigma_0\sigma_1 \ldots) = \langle\sigma_0, \perp\rangle$. L'abstraction d'un ensemble de traces est l'ensemble des abstractions des traces de cet ensemble $\alpha_d(X) = \{\alpha_d(\sigma) : \sigma \in X\}$.

Comme dernier exemple, la sémantique naturelle ignore tout simplement les comportements infinis et donc l'abstraction supplémentaire est $\alpha_n(X) = \{\langle s, s'\rangle \in X \mid s' \neq \perp\}$.

On remarque (1) que l'abstraction $\alpha$ est croissante ($T \subseteq T' \Rightarrow \alpha(T) \subseteq \alpha(T')$) et (2) que la concrétisation l'est également. La sémantique abstraite $t = \alpha(T)$ est une approximation ou abstraction de la sémantique concrète en ce sens que la sémantique concrète $\gamma(t)$ que l'on peut reconstituer à partir de la sémantique abstraite est en général plus grande que la sémantique concrète initiale $T$ : (3) $T \subseteq (\gamma(\alpha(T))$. Enfin la sémantique concrète $\gamma(t)$ ne perd aucune information sur la sémantique abstraite $t$ en ce sens que $t = \alpha(\gamma(t))$ et plus généralement (4) $\alpha(\gamma(t)) \subseteq t$. Les propriétés (1) à (4) sont caractéristiques des correspondances de Galois qui sont à la base de la théorie de l'interprétation abstraite initiée par [COU 78, COU 77a, COU 79]. En pratique on utilise souvent des hypothèses plus faibles [COU 92].

L'abstraction formalise une perte d'information, qui ne permet pas de répondre à toutes les questions. Toutes les réponses données par la sémantique abstraite sont toujours justes pour la sémantique concrète. Cependant en général certaines questions concrètes ne peuvent pas trouver de réponse exacte par la sémantique abstraite. Par exemple les sémantiques de traces, dénotationnelle et naturelle permettent de répondre à la question (1) : « L'exécution du programme partant de l'état $x$ peut-elle terminer dans l'état $y$ ? ». Seules les sémantiques de traces et dénotationnelle permettent de répondre à la question (2) « L'exécution du programme partant de l'état $x$ se termine-t-elle toujours ? ». La seule réponse valide avec la sémantique naturelle est « on ne sait pas » puisque les comportements infinis sont ignorés. Enfin la sémantique de traces permet de répondre à la question (3) « L'état $x$ peut-il être immédiatement suivi de l'état $y$ pendant l'exécution du programme » alors que les sémantiques dénotationnelles et naturelles ne permettent pas d'y répondre directement puisque les états intermédiaires du calcul sont oubliés. Ceci montre que les sémantiques les plus concrètes/précises permettent de répondre à plus de questions et que les sémantiques abstraites/approchées sont plus simples. Deux sémantiques peuvent être non directement comparables. C'est le cas de la sémantique naturelle (qui permet de répondre à la question (1) mais pas à la question (3)) et de la sémantique de transitions (qui permet de répondre à la question (3) mais pas à la question (1)). De manière générale, les sémantiques s'ordonnent en un treillis complet selon la relation de précision [COU 79, COU 00a]. En pratique, il s'agit de trouver dans ce treillis infini les sémantiques d'intérêt.

## 5. Analyse statique

Si l'approximation est suffisamment grossière, l'abstraction d'une sémantique peut permettre d'en donner une version moins précise mais calculable par ordinateur. Par calcul effectif de la sémantique abstraite, l'ordinateur est capable d'analyser le comportement de programmes et de logiciels avant même de les exécuter. Ceci permet de découvrir les erreurs de programmation avant qu'elles ne produisent des catastrophes [LAC 98], ce qui est essentiel pour les systèmes informatiques critiques (par exemple : avions, fusées, centrales nucléaires, etc.).

Pour donner l'intuition des approximations utilisées, considérons une analyse d'invariance (« safety analysis »). L'abstraction d'une trace est l'ensemble des états sur cette trace $\alpha_s(\sigma) = \{\sigma_k : k \in [0, |\sigma|[\}$. L'approximation d'un ensemble $X$ de traces est alors l'ensemble des états figurant sur au moins une trace de l'ensemble $\alpha_s(X) = \bigcup_{\sigma \in X} \alpha_s(\sigma)$. Supposons par exemple qu'un état $s$ soit composé d'un état de contrôle désignant un point $\ell \in \mathcal{C}$ du programme et de variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ à valeurs entières $\langle x_1, \ldots, x_n \rangle \in \mathbb{Z}$. L'invariant global $\alpha_s(X)$ est isomorphe au vecteur des invariants locaux associant à chaque point du programme l'ensemble des tuples des valeurs possibles des variables $\alpha_l(Y) = \prod_{\ell \in \mathcal{C}} \{\langle x_1, \ldots, x_n \rangle : \langle \ell, \langle x_1, \ldots, x_n \rangle \rangle \in Y\}$. On se ramène donc à l'approximation calculable d'un ensemble (en pratique infini) de points de $\mathbb{Z}^n$. Pour $n = 2$, la figure 1 donne des approximations fréquemment utilisées pour l'analyse statique de programmes. Une approximation non relationnelle permet d'ignorer les relations entre les valeurs des variables $\alpha_r(X) = \prod_{i=1}^{n} \{x_i : \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n : \langle x_1, \ldots, x_i, \ldots, x_n \rangle \in X\}$. Il reste alors à abstraire un

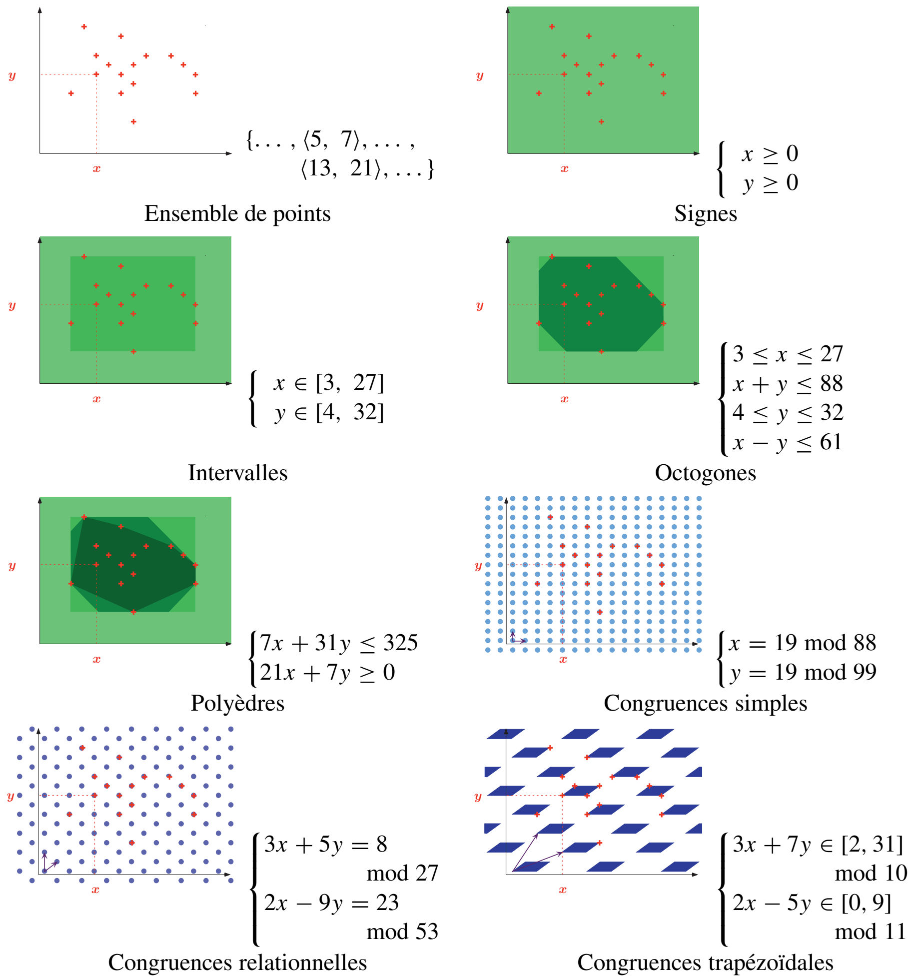


**Figure 1.** *Approximations d'un ensemble de points*

ensemble d'entiers, par exemple par l'intervalle des valeurs minimales et maximales $\alpha_i(Z) = [\min Z, \max Z]$. Finalement l'abstraction d'un ensemble de traces par des intervalles de valeurs des variables entières associées à chaque point de contrôle d'un programme [COU 76] est obtenue par composition $\alpha(T) = \alpha_i(\alpha_r(\alpha_l(\alpha_s(T))))$.

La théorie de l'interprétation abstraite montre que l'abstraction $\alpha(T)$ est solution d'une équation de point fixe, déduite par abstraction de l'équation [1] définissant la sémantique de traces $T$. Considérons un exemple très simple [COU 76] :

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

$$\begin{cases} X_1 = [1,1] & [2] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] & [3] \\ X_3 = X_2 \oplus [1,1] & [4] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] & [5] \end{cases}$$

L'équation [4 ] exprime que l'ensemble des valeurs de `x` au point 3 est l'ensemble des valeurs de `x` en 2 augmentées de 1. L'équation [3 ] exprime que l'ensemble des valeurs de `x` au point 2 est l'ensemble des valeurs de `x` en 1 (entrée dans la boucle) ou en 3 (itérations suivantes) satisfaisant le test `x < 10000`. Ces équations sont donc une version abstraite ou simplifiée des conditions de vérification de Floyd/Naur/Hoare pour la preuve d'invariance [COU 90].

La méthode la plus commune pour résoudre les équations est itérative, partant de l'infimum et utilisant n'importe quelle stratégie chaotique ou asynchrone [COU 78]. L'itération croissante est la suivante :

$$\begin{array}{lcllllllllll} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \dots \\ \left\{\begin{array}{l} X_1 \\ X_2 \\ X_3 \\ X_4 \end{array}\right. & \begin{array}{c} = \\ = \\ = \\ = \end{array} & \begin{array}{c} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ \emptyset \\ \emptyset \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,1] \\ \emptyset \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,1] \\ [2,2] \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,2] \\ [2,2] \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,2] \\ [2,3] \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,3] \\ [2,3] \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,3] \\ [2,4] \\ \emptyset \end{array} & \begin{array}{c} [1,1] \\ [1,4] \\ [2,4] \\ \emptyset \end{array} & \begin{array}{c} \dots \\ \dots \\ \dots \\ \emptyset \end{array} \end{array}$$

Intuitivement la stratégie d'itération choisie consiste à suivre simultanément toutes les exécutions possibles, sans en omettre aucune, en remplaçant les ensembles de valeurs concrètes par des valeurs d'intervalle abstraites. En général l'itération croissante n'est pas convergente. Une opération d'élargissement doit être utilisée pour accélérer la convergence. On obtient une surestimation du point fixe, qui peut être améliorée par une itération décroissante (dont il faut en général contrôler la convergence finie par une opération de rétrécissement) :

$$\begin{array}{lcl} & & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array}\ \overbrace{\begin{array}{cc} 4 & 5 \end{array}}^{\text{élargissement}}\ \begin{array}{cc} 6 & 7 \end{array} \\ \left\{\begin{array}{l} X_1 \\ X_2 \\ X_3 \\ X_4 \end{array}\right. & \begin{array}{c} = \\ = \\ = \\ = \end{array} & \underbrace{\begin{array}{ccccccc} \emptyset & [1,1] & [1,1] & [1,1] & [1,1] & [1,1] & [1,1] \\ \emptyset & \emptyset & [1,1] & [1,1] & [1,2] & [1,+\infty] & [1,+\infty] \\ \emptyset & \emptyset & \emptyset & [2,2] & [2,2] & [2,2] & [2,10000] \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \end{array}}_{\text{itération croissante}} \underbrace{\begin{array}{c} [1,1] \\ [1,9999] \\ [2,10000] \\ [10000,10000] \end{array}}_{\text{itération décroissante}} \end{array}$$

Après extension aux procédures récursives [COU 77b], l'analyse peut être utilisée pour des langages impératifs comme Pascal [BOU 93] ou Java.

La spécification minimale est qu'il ne doit pas y avoir de dépassements de capacité.

Le résultat de l'analyse permet de le vérifier aisément :

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

En général on signalera qu'une erreur à l'exécution est certaine, impossible ou potentielle (quand l'analyse est trop approchée pour conclure), mais la couverture de la spécification est toujours complète.

La plupart des structures manipulées par les programmes ne sont pas numériques. C'est le cas par exemple des structures de calcul (graphes d'appel, arbres de récursivité), de données (arbres de recherche), de communication (programmes distribués), de transfert d'information (programmes mobiles), etc. Il est très difficile de trouver des représentations informatiques compactes et précises de tels ensembles d'objets (langages, automates, arbres, graphes, etc.) telles que les diverses opérations ensemblistes sont implantées efficacement, la taille mémoire n'explose pas combinatoirement pour des ensembles complexes ou irréguliers et les approximations sont précises. De très nombreux travaux sont en cours sur ce thème.

Ce type d'analyse par interprétation abstraite s'étend à des assertions intermittentes aussi bien qu'invariantes [COU 81] d'où l'idée de test abstrait [BOU 93] qui généralise la vérification de modèle (« model-checking » ) [COU 00b]. En pratique, de telles analyses statiques ont été utilisées pour l'analyse statique des logiciels du programme de vol et de la centrale inertielle du lanceur Ariane 5, ce qui fût une réussite pour les vols 502, 503 et l'ARD [LAC 98].

De très nombreux langages ont été considérés comme la programmation logique et plus récemment le code mobile. Quelques autres applications récentes de l'analyse statique par interprétation abstraite concernent l'inférence de types (pour les systèmes indécidables) [COU 97b], la vérification de modèle de systèmes infinis « abstract model-checking » [COU 99], la transformation et l'optimisation de programmes, la différenciation automatique, l'analyse de protocoles cryptographiques, le tatouage sémantique de logiciels, etc.

Il reste beaucoup à faire sur le plan fondamental en ce qui concerne l'ordre supérieur, la modularité, les flottants, les analyses probabilistes, les propriétés de vivacité avec équité, etc.

L'interprétation abstraite a été très récemment industrialisée (« AbsInt Angewandte Informatik GmbH » (Allemagne) en 1998 et « Polyspace Technologies » (France) en 1999. Ces sociétés développent des analyseurs statiques de logiciels qui sont disponibles commercialement.

## 6. Bibliographie

[BOU 90] BOURDONCLE F., « Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity ». In DERANSART P. et MAŁUSZYŃSKI J., Eds., *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '90*, Linköping, Suède, 20–22 août 1990, Lecture Notes in Computer Science 456, p. 307–323. Springer-Verlag, Berlin, Allemagne, 1990.

[BOU 93] BOURDONCLE F., « Abstract Debugging of Higher-Order Imperative Languages ». In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, p. 46–55. ACM Press, New York, USA, 1993.

[COU 76] COUSOT P. et COUSOT R., « Static determination of dynamic properties of programs ». In *Proceedings of the Second International Symposium on Programming*, p. 106–130. Dunod, Paris, 1976.

[COU 77a] COUSOT P. et COUSOT R., « Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints ». In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 238–252, Los Angeles, Californie, 1977. ACM Press, New York, N. Y., USA.

[COU 77b] COUSOT P. et COUSOT R., « Static determination of dynamic properties of recursive procedures ». In NEUHOLD E., Ed., *IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B., Canada*, p. 237–277. North-Holland Pub. Co., Amsterdam, Pays-Bas, 1977.

[COU 78] COUSOT P., « *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes* ». Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, 21 mars 1978.

[COU 79] COUSOT P. et COUSOT R., « Systematic design of program analysis frameworks ». In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 269–282, San Antonio, Texas, 1979. ACM Press, New York, N. Y., USA.

[COU 81] COUSOT P., Semantic Foundations of Program Analysis. In MUCHNICK S. et JONES N., Eds., *Program Flow Analysis: Theory and Applications*, Chapitre 10, p. 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1981.

[COU 90] COUSOT P., Methods and Logics for Proving Programs. In VAN LEEUWEN J., Ed., *Formal Models and Semantics*, vol. B de *Handbook of Theoretical Computer Science*, Chapitre 15, p. 843–993. Elsevier Science Publishers B.V., Amsterdam, Pays-Bas, 1990.

[COU 92] COUSOT P. et COUSOT R., « Abstract Interpretation Frameworks ». *Journal of Logic and Computation*, vol. 2, n° 4, p. 511–547, août 1992.

[COU 97a] COUSOT P., « Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation ». *Electronic Notes in Theoretical Computer Science*, vol. 6, 1997. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages.

[COU 97b] COUSOT P., « Types as Abstract Interpretations, papier invité ». In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 316–331, Paris, janvier 1997. ACM Press, New York, USA.

[COU 99] COUSOT P. et COUSOT R., « Refining Model Checking by Abstract Interpretation ». *Automated Software Engineering*, vol. 6, p. 69–95, 1999.

[COU 00a] COUSOT P., « Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation ». *Theoretical Computer Science*, À paraître (Version préliminaire dans [COU 97a]).

[COU 00b] COUSOT P. et COUSOT R., « Temporal Abstract Interpretation ». In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, 2000. ACM Press, New York, USA.

[LAC 98] LACAN P., MONFORT J., RIBAL L. V. Q., DEUTSCH A. et GONTHIER G., « The software reliability verification process: The ARIANE 5 Example ». In *Proceedings DASIA 98 – DAta Systems IN Aerospace*, Athènes, Grèce. ESA Publications, SP-422, 25–28 mai 1998.

**Patrick Cousot** est ingénieur civil des mines. Il a soutenu une thèse de docteur-ingénieur puis une thèse de doctorat d'État en informatique à Grenoble. Après avoir été attaché et chargé de recherche au CNRS à l'IMAG, professeur d'informatique à l'Université de Metz puis à l'École polytechnique, il actuellement professeur et directeur des études en informatique à l'École normale supérieure. Ses travaux de recherche concernent l'interprétation abstraite qu'il a introduite et son application à l'analyse statique et à la sémantique. Ses travaux scientifiques ont été récompensés en 1999 par la médaille d'argent du CNRS.