Dagstuhl Perspectives Workshop

Formal Methods — Just a Euro-Science?

Schloß Dagstuhl — Leibniz-Zentrum für Informatik

Abstract interpretation: from origin to perspectives

Patrick Cousot

cousot@di.ens.fr
di.ens.fr/~cousot

pcousot@cs.nyu.edu
cs.nyu.edu/~pcousot

Radhia Cousot

rcousot@di.ens.fr
di.ens.fr/~rcousot

November 30, 2010 — December 3, 2010

Abstract interpretation: origin (abridged)

Before starting (1972-73): formal syntax

- Radhia Rezig: works on precedence parsing (R.W. Floyd, N. Wirth and H. Weber, etc.) for Algol 68

  ➡ Pre-processing (by static analysis and transformation) of the grammar before building the *bottom-up* parser

- Patrick Cousot: works on context-free grammar parsing (J. Earley and F. De Remer)

  ➡ Pre-processing (by static analysis and transformation) of the grammar before building the *top-down* parser

---

- Radhia Rezig. *Application de la méthode de précédence totale à l'analyse d'Algol 68,* Master thesis, Université Joseph Fourier, Grenoble, France, September 1972.
- Patrick Cousot. *Un analyseur syntaxique pour grammaires hors contexte ascendant sélectif et général.* In Congrès AFCET 72, Brochure 1, pages 106-130, Grenoble, France, 6-9 November 1972.

Before starting (1972-73): formal semantics

- Patrick Cousot: works on the operational semantics of programming languages and the derivation of implementations from the formal definition

  ➡ Static analysis of the formal definition and transformation to get the implementation by "pre-evaluation" (similar to the more recent "partial evaluation")

• Patrick Cousot. *Définition interprétative et implantation de languages de programmation*. Thèse de Docteur Ingénieur en Informatique, Université Joseph Fourier, Grenoble, France, 14 Décembre 1974 (submitted in 1973 but defended after finishing military service with J.D. Ichbiah at CII).

Vision (1973):

Intervals →

Assertions →

Static analysis →

Les langages actuels ne sont pas faits pour l'optimisation. Entre autres, il y a certains faits sur un programme qui sont connus du programmeur et qui ne sont pas explicites dans le programme. On pourrait y remédier en incluant des assertions, tout comme on insère des déclarations de type pour les variables.

Exemple :

(1) - pour i de 0 à 10 faire a[i] := i ; fin ;

(2) - pour i de 11 à 10000 faire a[i] := 0 ; fin ;

(3) - a[(a[j] + 1) x a [j + 1]] := j ;

(4) - si a[j x j + 2 x j + 1] ≠ a[j] aller à étiquette ;

Pour un tel programme, il est important de savoir que $1 \leq j < 99$ (à charge éventuellement au système de le déduire à partir d'autres assertions), parce qu'on peut alors remplacer (4) par (4') :

(4') si j < 10 aller à étiquette ;

Cette insertion d'assertions peut donc servir de guide à une analyse automatique des programmes essentielle pour l'optimisation (mais également pour la mise au point, la documentation automatique, la décompilation, l'adaptation à un changement d'environnement d'exécution...). Dans tous les exemples que nous avons pris, (équivalence de définitions de données, équivalence de définition d'opérateurs) nous avons conduit cette analyse sémantique à la main.

La possibilité de son automation, nous semble conditionner les progrès dans le domaine de l'optimisation de l'implantation automatisée d'un langage étant donnée sa définition, aussi bien que dans celui de l'optimisation des programmes [41].

1973: Dijkstra's handmade proofs

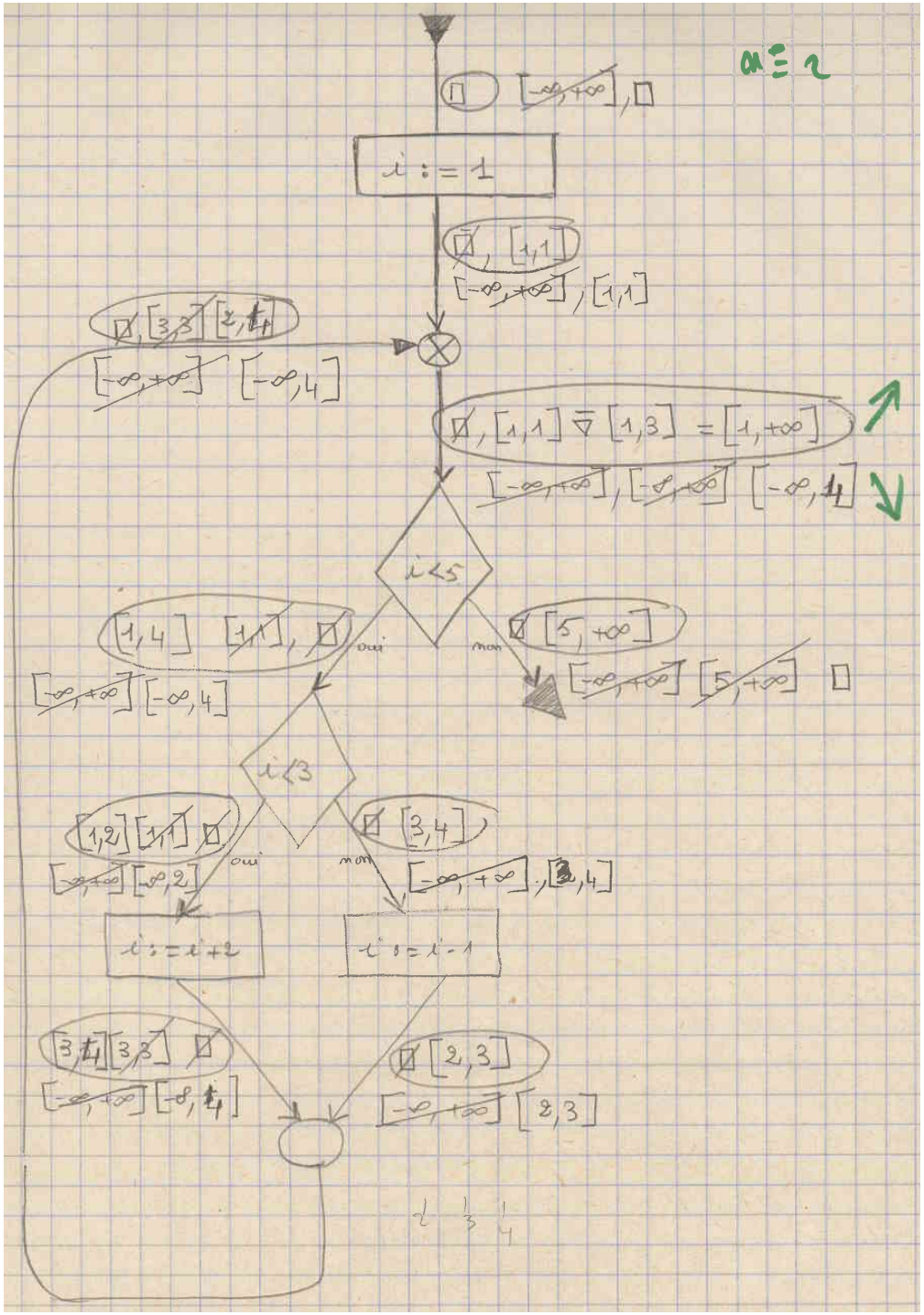
- Radhia Rezig: attends Marktoberdorf summer school, July 25–Aug. 4, 1973

  ➡ Dijkstra shows program proofs (*inventing* elegant backward invariants)

  ➡ Radhia has the idea of automatically *inferring* the invariants by a backward calculus to determine intervals

1974: origin

- Radhia Rezig shows her interval analysis ideas to Patrick Cousot
  - ➡ Patrick very critical on going backwards from $[-\infty, +\infty]$ and claims that going forward would be much better
  - ➡ Patrick also very skeptical on forward termination for loops
- Radhia comes back with the idea of extrapolating bounds to $\pm\infty$ for the forward analysis
- We discover widening = induction in the abstract and that the idea is very general



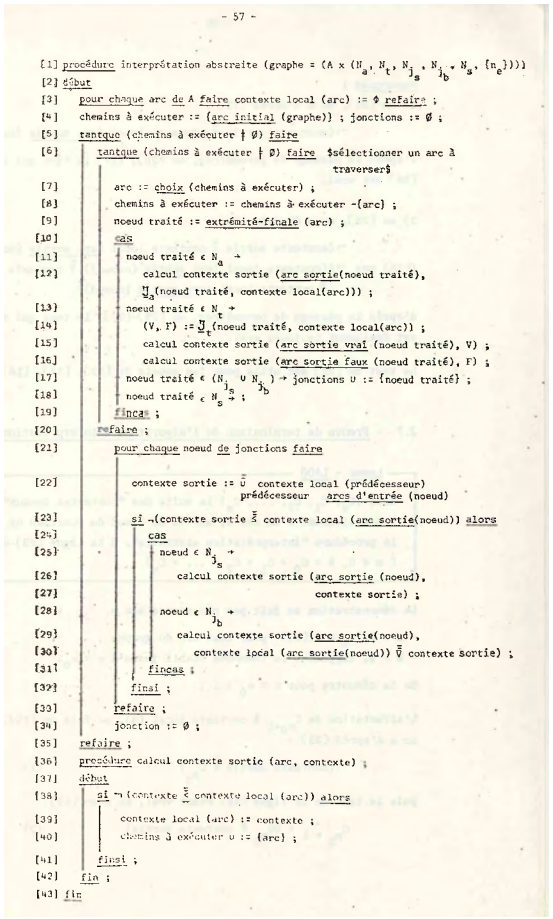
Notes of Radhia Rezig

on forward iteration from $\square = \bot$ [1] versus backward iteration from $[-\infty, +\infty]$ [2]

[1] i.e. forward least fixed point

[2] i.e. backward greatest fixed point

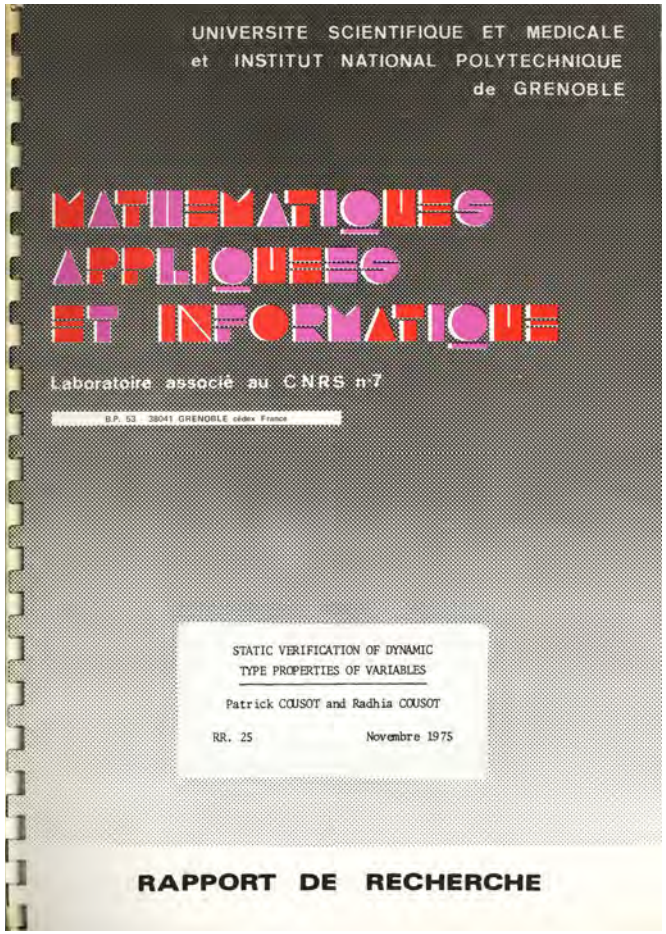
The first reports and publication(1975–76)



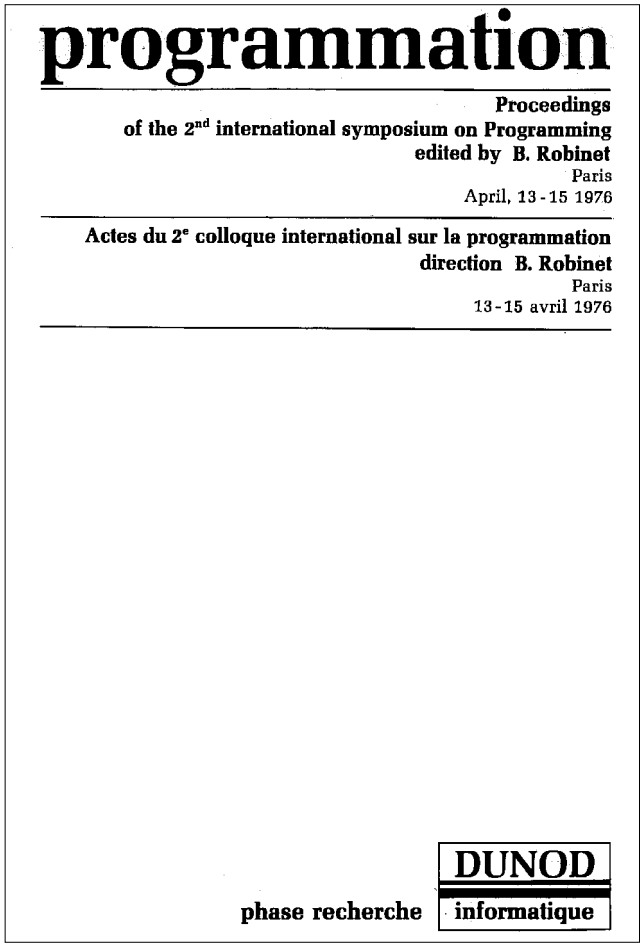
```
                                                              - 57 -

[1] procédure interprétation abstraite (graphe = (A x (N_a, N_t, N_js, N_jb, N_s), {n_e}))
[2] début
[3]   pour chaque arc de A faire contexte local (arc) := ∅ refaire ;
[4]   chemins à exécuter := {arc initial (graphe)} ; jonctions := ∅ ;
[5]   tantque (chemins à exécuter ≠ ∅) faire
[6]   |  tantque (chemins à exécuter ≠ ∅) faire  $sélectionner un arc à
                                                    traverser$
[7]   |  |  arc := choix (chemins à exécuter) ;
[8]   |  |  chemins à exécuter := chemins à exécuter -{arc} ;
[9]   |  |  noeud traité := extrémité-finale (arc) ;
[10]  |  |  cas
[11]  |  |  | noeud traité ∈ N_a →
[12]  |  |  |    calcul contexte sortie (arc sortie(noeud traité),
                  J_a(noeud traité, contexte local(arc))) ;
[13]  |  |  | noeud traité ∈ N_t →
[14]  |  |  |    (V, F) := J_t(noeud traité, contexte local(arc)) ;
[15]  |  |  |    calcul contexte sortie (arc sortie vrai (noeud traité), V) ;
[16]  |  |  |    calcul contexte sortie (arc sortie faux (noeud traité), F) ;
[17]  |  |  | noeud traité ∈ (N_js ∪ N_jb) → jonctions ∪ := {noeud traité} ;
[18]  |  |  | noeud traité ∈ N_s → ;
[19]  |  |  fincas ;
[20]  |  refaire ;
[21]  |  |  pour chaque noeud de jonctions faire
[22]  |  |  |  contexte sortie := ⊔ contexte local (prédécesseur)
                                  prédécesseur  arcs d'entrée (noeud)
[23]  |  |  |  si ¬(contexte sortie ⊑ contexte local (arc sortie(noeud)) alors
[24]  |  |  |  |  cas
[25]  |  |  |  |  | noeud ∈ N_js →
[26]  |  |  |  |  |    calcul contexte sortie (arc sortie (noeud),
[27]  |  |  |  |  |                                contexte sortie) ;
[28]  |  |  |  |  | noeud ∈ N_jb →
[29]  |  |  |  |  |    calcul contexte sortie (arc sortie(noeud),
[30]  |  |  |  |  |      contexte local (arc sortie(noeud)) ▽ contexte sortie) ;
[31]  |  |  |  |  fincas ;
[32]  |  |  |  finsi ;
[33]  |  |  refaire ;
[34]  |  |  jonction := ∅ ;
[35]  |  refaire ;
[36]  |  procédure calcul contexte sortie (arc, contexte) ;
[37]  |  début
[38]  |  |  si ¬(contexte ⊑ contexte local (arc)) alors
[39]  |  |  |  contexte local (arc) := contexte ;
[40]  |  |  |  chemins à exécuter ∪ := {arc} ;
[41]  |  |  finsi ;
[42]  |  fin ;
[43]  fin
```

The first abstract interpreter with widening (as of 23 Sep. 1975)



The first research report (Nov. 1975)



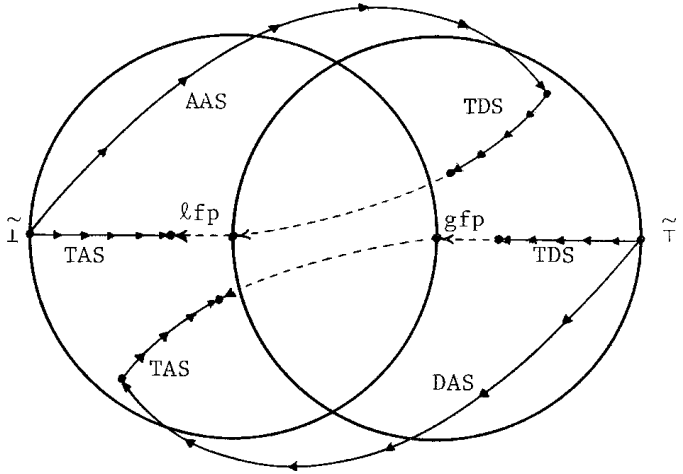
The first publication (ISOP II, Apr. 76)

The IRIA–SESORI contract (1975-76)

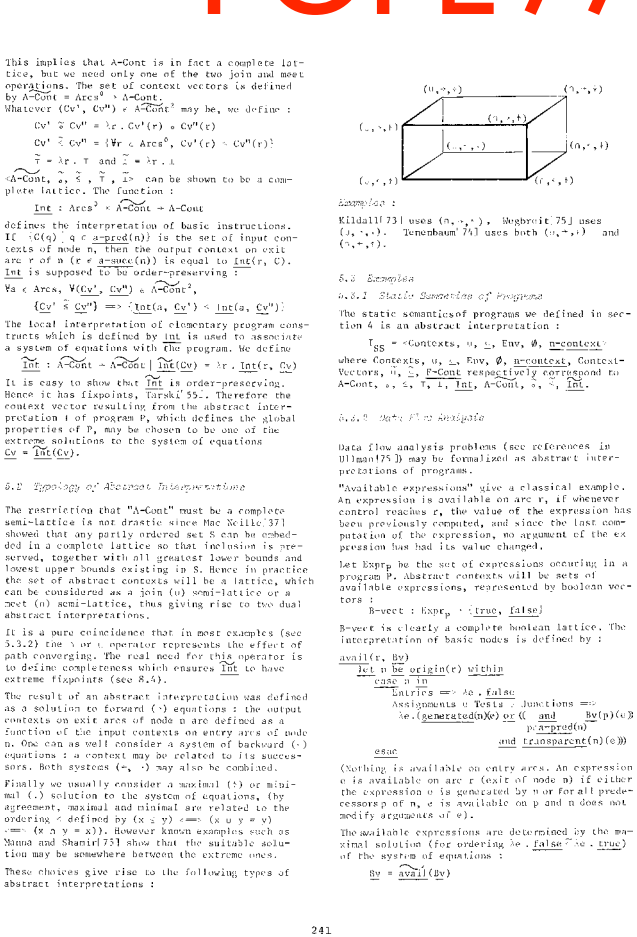
- The project evaluator points to us the literature on constant propagation in data flow analysis. It appears that it is related to some of ours ideas, but a.o.
  - We are not syntactic (as in boolean DFA)
  - We have no need of some hypotheses (e.g. distributivity not even satisfied by constant propagation!)
  - We have no restriction to finite lattices (or ACC)
  - We have no need of an a-posteriori proof of correctness (e.g. with respect to the MOP as in DFA)
  - ...
- New general ideas
  - The formal notions of abstraction/approximation
  - The formal notion of abstract induction (widening) to handle infiniteness and/or complexity
  - The systematic correct design with respect to a formal semantics
  - ...

Maturation (1976 – 77): from an *algorithmic* to an *algebraic* point of view

- Narrowing, duality
- Transition systems, traces
- Fixpoints, chaotic/asynchronous iterations, approximation
- Abstraction, formalized by Galois connections, closure operators, Moore families, ...;
- Numeric and symbolic abstract domains, combinations of abstract domains
- Recursive procedures, relational analyses, heap analysis
- etc.



POPL'77 FDPC'77 POPL'79



This implies that A-Cont is in fact a complete lattice, but we need only one of the two join and meet operations. The set of context vectors is defined by $\widetilde{A\text{-Cont}} = Arcs^0 \to A\text{-Cont}$.
Whatever $(\widetilde{Cv'}, \widetilde{Cv''}) \in \widetilde{A\text{-Cont}}^2$ may be, we define :

$$\widetilde{Cv'} \widetilde{\leq} \widetilde{Cv''} = \lambda r.\ Cv'(r) \leq Cv''(r)$$

$$\widetilde{Cv'} \widetilde{\leq} \widetilde{Cv''} = \{\forall r \in Arcs^0,\ Cv'(r) \leq Cv''(r)\}$$

$\widetilde{\top} = \lambda r.\ \top$ and $\widetilde{\bot} = \lambda r.\ \bot$

$\langle \widetilde{A\text{-Cont}}, \widetilde{\wedge}, \widetilde{\vee}, \widetilde{\top}, \widetilde{\bot} \rangle$ can be shown to be a complete lattice. The function :

$$\widetilde{Int} : Arcs^2 \to \widetilde{A\text{-Cont}} \to A\text{-Cont}$$

defines the interpretation of basic instructions. If $\widetilde{Cv}(q) \mid q \in a\text{-pred}(n)\}$ is the set of input contexts of node n, then the output context on exit arc r of n $(r \in a\text{-succ}(n))$ is equal to $\widetilde{Int}(r, \widetilde{Cv})$. $\widetilde{Int}$ is supposed to be order-preserving :

$$\forall a \in Arcs,\ \forall \{\widetilde{Cv'}, \widetilde{Cv''}\} \in \widetilde{A\text{-Cont}}^2,$$

$$\{\widetilde{Cv'} \widetilde{\leq} \widetilde{Cv''}\} \Rightarrow \{\widetilde{Int}(a, \widetilde{Cv'}) \leq \widetilde{Int}(a, \widetilde{Cv''})\}$$

The local interpretation of elementary program constructs which is defined by $\widetilde{Int}$ is used to associate a system of equations with the program. We define

$$\widetilde{\widetilde{Int}} : \widetilde{A\text{-Cont}} \to \widetilde{A\text{-Cont}} \mid \widetilde{\widetilde{Int}}(\widetilde{Cv}) = \lambda r.\ \widetilde{Int}(r, \widetilde{Cv})$$

It is easy to show that $\widetilde{\widetilde{Int}}$ is order-preserving. Hence it has fixpoints, Tarski'55]. Therefore the context vector resulting from the abstract interpretation I of program P, which defines the global properties of P, may be chosen to be one of the extreme solutions to the system of equations $\widetilde{Cv} = \widetilde{\widetilde{Int}}(\widetilde{Cv})$.

6.2 Topology of Abstract Interpretations

The restriction that "A-Cont" must be a complete semi-lattice is not drastic since Mac Neille 37] showed that any partly ordered set S can be embedded in a complete lattice so that inclusion is preserved, together with all greatest lower bounds and lowest upper bounds existing in S. Hence in practice the set of abstract contexts will be a lattice, which can be considered as a join (u) semi-lattice or a meet (n) semi-lattice, thus giving rise to two dual abstract interpretations.

It is a pure coincidence that in most examples (see 5.3.2) the $\widetilde{\vee}$ or $\widetilde{\wedge}$ operator represents the effort of path converging. The real need for this operator is to define completeness which ensures $\widetilde{Int}$ to have extreme fixpoints (see 8.4).

The result of an abstract interpretation was defined as a solution to forward (·) equations : the output contexts on exit arcs of node n are defined as a function of the input contexts on entry arcs of node n. One can as well consider a system of backward (·) equations : a context may be related to its successors. Both systems (·, ·) may also be combined.

Finally we usually consider a maximal (↑) or minimal (↓) solution to the system of equations, (by agreement, maximal and minimal are related to the ordering ≤ defined by $(x \leq y) \Leftrightarrow (x \cup y = y) \Leftrightarrow (x \cap y = x)$). However known examples such as Manna and Shamir 73] show that the suitable solution may be somewhere between the extreme ones.

These choices give rise to the following types of abstract interpretations :

Examples :

Kildall 73] uses $(\cup, \wedge, \leq)$, Wegbreit 75] uses $(\cup, \wedge, \cdot)$. Tenenbaum 74] uses both $(\cup, \wedge, \cdot)$ and $(\cap, \vee, \cdot)$.

6.3 Examples

6.3.1 Static Semantics of Programs

The static semantics of programs we defined in section 4 is an abstract interpretation :

$$I_{SS} = \langle Contexts, \cup, \subseteq, Env, \emptyset, n\text{-context} \rangle$$

where Contexts, ∪, ⊆, Env, ∅, n-context, Context-Vectors, $\widetilde{\cup}$, $\widetilde{\subseteq}$, $\widetilde{A\text{-Cont}}$ respectively correspond to A-Cont, ∧, ≤, ∨, ⊥, Int, A-Cont, ∧, ≤, $\widetilde{Int}$.

6.3.2 Data Flow Analysis

Data flow analysis problems (see references in Ullman 75]) may be formalized as abstract interpretations of programs.

"Available expressions" give a classical example. An expression is available on arc r, if whenever control reaches r, the value of the expression has been previously computed, and since the last computation of the expression, no argument of the expression has had its value changed.

Let $Expr_P$ be the set of expressions occuring in a program P. Abstract contexts will be sets of available expressions, represented by boolean vectors :

B-vect : $Expr_P \to \{true, false\}$

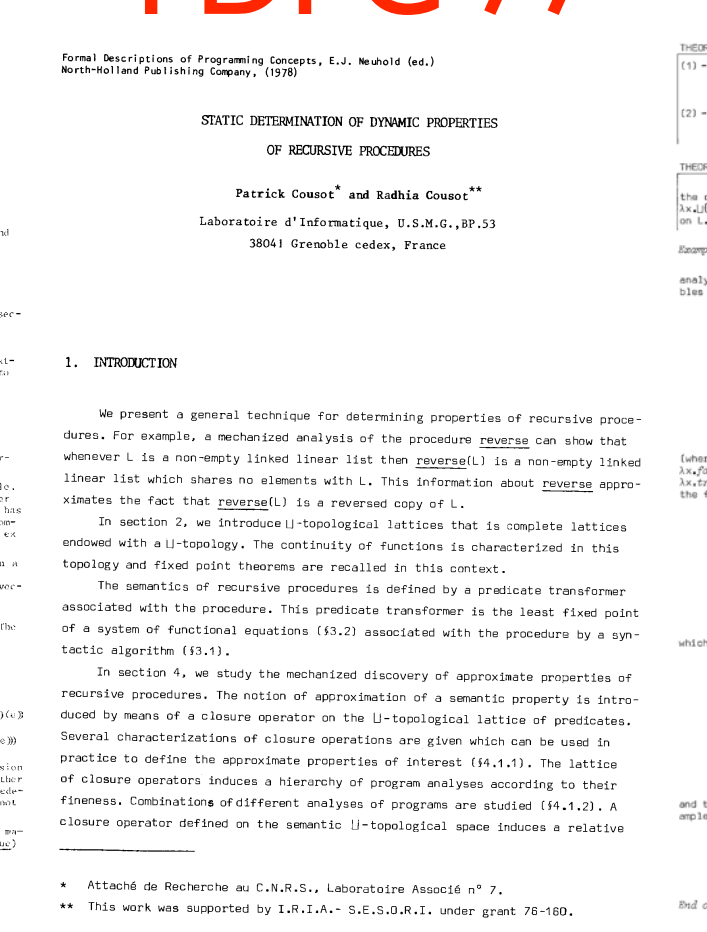
B-vect is clearly a complete boolean lattice. The interpretation of basic nodes is defined by :

```
avail(r, Bv)
  let n be origin(r) within
    case n in
      Entries => λe . false
      Assignments & Tests & Functions =>
        λe.(generated(n)(e) or ((  and   Bv(p)(e))
                                p∈a-pred(n)
                            and transparent(n)(e)))
    esac
```

(Nothing is available on entry arcs. An expression e is available on arc r (exit of node n) if either the expression e is generated by n or for all predecessors p of n, e is available on p and n does not modify arguments of e).

The available expressions are determined by the maximal solution (for ordering $\lambda e . false \widetilde{\leq} \lambda e . true$) of the system of equations :

$$\widetilde{Bv} = \widetilde{avail}(Bv)$$



Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.)
North-Holland Publishing Company, (1978)

STATIC DETERMINATION OF DYNAMIC PROPERTIES

OF RECURSIVE PROCEDURES

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP.53

38041 Grenoble cedex, France

1. INTRODUCTION

We present a general technique for determining properties of recursive procedures. For example, a mechanized analysis of the procedure reverse can show that whenever L is a non-empty linked linear list then reverse(L) is a non-empty linked linear list which shares no elements with L. This information about reverse approximates the fact that reverse(L) is a reversed copy of L.

In section 2, we introduce ⊔-topological lattices that is complete lattices endowed with a ⊔-topology. The continuity of functions is characterized in this topology and fixed point theorems are recalled in this context.

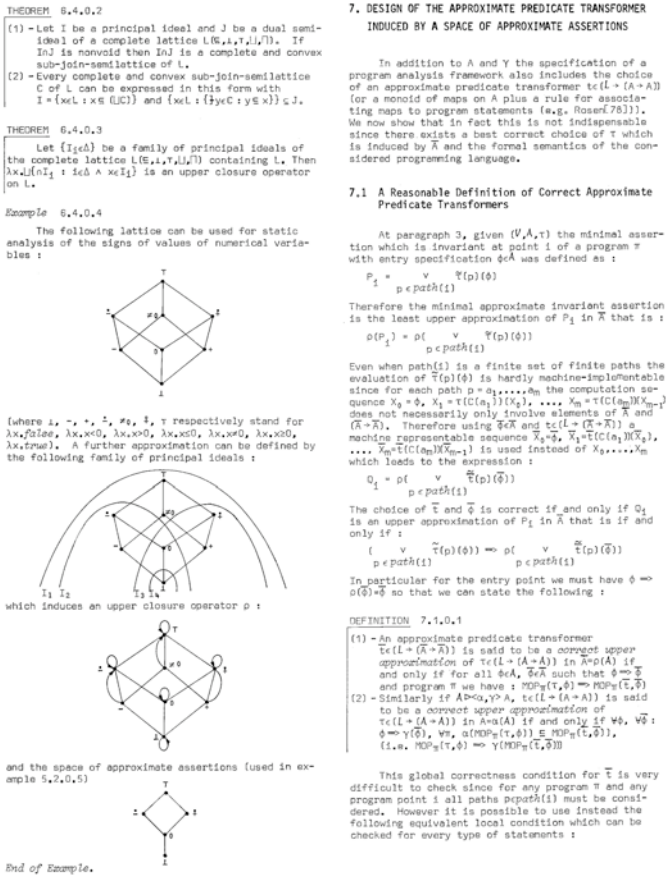
The semantics of recursive procedures is defined by a predicate transformer associated with the procedure. This predicate transformer is the least fixed point of a system of functional equations (§3.2) associated with the procedure by a syntactic algorithm (§3.1).

In section 4, we study the mechanized discovery of approximate properties of recursive procedures. The notion of approximation of a semantic property is introduced by means of a closure operator on the ⊔-topological lattice of predicates. Several characterizations of closure operations are given which can be used in practice to define the approximate properties of interest (§4.1.1). The lattice of closure operators induces a hierarchy of program analyses according to their fineness. Combinations of different analyses of programs are studied (§4.1.2). A closure operator defined on the semantic ⊔-topological space induces a relative

---

* Attaché de Recherche au C.N.R.S., Laboratoire Associé n° 7.

** This work was supported by I.R.I.A.- S.E.S.O.R.I. under grant 76-160.



THEOREM 6.4.0.2

(1) - Let I be a principal ideal and J be a dual semi-ideal of a complete lattice $L(\subseteq, \perp, \top, \sqcup, \sqcap)$. If $I \cap J$ is nonvoid then $I \cap J$ is a complete and convex sub-join-semilattice of L.
(2) - Every complete and convex sub-join-semilattice C of L can be expressed in this form with $I = \{x \in L : x \leq \bigsqcup C\}$ and $\{x \in L : (\exists y \in C : y \leq x)\} \subseteq J$.

THEOREM 6.4.0.3

Let $\{I_\xi \mid \Delta\}$ be a family of principal ideals of the complete lattice $L(\subseteq, \perp, \top, \sqcup, \sqcap)$ containing L. Then $\lambda x. \bigsqcap \{I_\xi \mid \xi \in \Delta \wedge x \in I_\xi\}$ is an upper closure operator on L.

Example 6.4.0.4

The following lattice can be used for static analysis of the signs of values of numerical variables :

(where $\perp, -, +, \pm, \neq 0, \sharp, \top$ respectively stand for $\lambda x.false$, $\lambda x. x<0$, $\lambda x. x>0$, $\lambda x. x \leq 0$, $\lambda x. x \neq 0$, $\lambda x. x \geq 0$, $\lambda x.true$). A further approximation can be defined by the following family of principal ideals :

which induces an upper closure operator ρ :

and the space of approximate assertions (used in example 5.2.0.5)

End of Example.

7. DESIGN OF THE APPROXIMATE PREDICATE TRANSFORMER INDUCED BY A SPACE OF APPROXIMATE ASSERTIONS

In addition to A and Y the specification of a program analysis framework also includes the choice of an approximate predicate transformer $\tau c(L \to (A \to A))$ (or a monoid of maps on A plus a rule for associating maps to program statements (e.g. Roser 78])). We now show that in fact this is not indispensable since there exists a best correct choice of τ which is induced by $\overline{A}$ and the formal semantics of the considered programming language.

7.1 A Reasonable Definition of Correct Approximate Predicate Transformers

At paragraph 3, given (V, A, τ) the minimal assertion which is invariant at point i of a program π with entry specification $\phi \in A$ was defined as :

$$P_i = \bigvee_{p \in path(i)} \tau(p)(\phi)$$

Therefore the minimal approximate invariant assertion is the least upper approximation of P_i in $\overline{A}$ that is :

$$\rho(P_i) = \rho\Big(\bigvee_{p \in path(i)} \overline{\tau}(p)(\overline{\phi})\Big)$$

Even when path(i) is a finite set of finite paths the evaluation of $\overline{\tau}(p)(\overline{\phi})$ is hardly machine-implementable since for each path $p = a_1, \ldots, a_m$ the computation sequence $X_0 = \overline{\phi}$, $X_1 = \overline{\tau}(C(a_1))(X_0)$, ..., $X_m = \overline{\tau}(C(a_m))(X_{m-1})$ does not necessarily only involve elements of $\overline{A}$ and $(\overline{A} \to \overline{A})$. Therefore using $\overline{\gamma} \in \overline{A}$ and $\tau c(L \to (\overline{A} \to \overline{A}))$ a machine representable sequence $\overline{X}_0 = \overline{\phi}$, $\overline{X}_1 = \overline{\tau}(C(a_1))(\overline{X}_0)$, ..., $\overline{X}_m = \overline{\tau}(C(a_m))(\overline{X}_{m-1})$ is used instead of $X_0, \ldots, X_m$ which leads to the expression :

$$Q_i = \rho\Big(\bigvee_{p \in path(i)} \overline{\tau}(p)(\overline{\phi})\Big)$$

The choice of $\overline{\tau}$ and $\overline{\phi}$ is correct if and only if Q_i is an upper approximation of P_i in $\overline{A}$ that is if and only if :

$$\Big(\bigvee_{p \in path(i)} \overline{\tau}(p)(\phi)\Big) \Rightarrow \rho\Big(\bigvee_{p \in path(i)} \overline{\tau}(p)(\overline{\phi})\Big)$$

In particular for the entry point we must have $\phi \Rightarrow \rho(\overline{\phi}) = \overline{\phi}$ so that we can state the following :

DEFINITION 7.1.0.1

(1) - An approximate predicate transformer $\tau c(L \to (\overline{A} \to \overline{A}))$ is said to be a *correct upper approximation* of $\tau c(L \to (A \to A))$ in $\overline{A} \wp (A)$ if and only if for all $\phi \in A$, $\overline{\phi} \in \overline{A}$ such that $\phi \Rightarrow \overline{\phi}$ and program π we have : $MOP_\pi(\tau, \phi) \Rightarrow MOP_\pi(\overline{\tau}, \overline{\phi})$
(2) - Similarly if $A \wp \langle \alpha, \gamma \rangle \wp A$, $\tau c(L \to (A \to A))$ is said to be a *correct upper approximation* of $\tau c(L \to (A \to A))$ in $A \wp \alpha(A)$ if and only if $\forall \phi, \forall \overline{\phi} : \phi \Rightarrow \gamma(\overline{\phi}), \forall \pi, \alpha(MOP_\pi(\tau, \phi)) \in MOP_\pi(\overline{\tau}, \overline{\phi})$, (i.e. $MOP_\pi(\tau, \phi) \Rightarrow \gamma(MOP_\pi(\overline{\tau}, \overline{\phi}))$)

This global correctness condition for $\overline{\tau}$ is very difficult to check since for any program π and any program point i all paths $p \in path(i)$ must be considered. However it is possible to use instead the following equivalent local condition which can be checked for every type of statements :

On this page: dual, conjugate and inversion:lfp/gfp wp/sp (i.e. pre/post) $\widetilde{wp}/\widetilde{sp}$)

Topology, higher-order fixpoints, operational/ summary/... analysis

Galois connections, closure operators, Moore families, ideals,...

Cited by 3926

Cited by 148

Cited by 1033

Google scholar

And a bit of Mathematics...

PACIFIC JOURNAL OF MATHEMATICS
Vol. 82, No. 1, 1979

CONSTRUCTIVE VERSIONS OF TARSKI'S FIXED POINT THEOREMS

PATRICK COUSOT AND RADHIA COUSOT

Let F be a monotone operator on the complete lattice L into itself. Tarski's lattice theoretical fixed point theorem states that the set of fixed points of F is a nonempty complete lattice for the ordering of L. We give a constructive proof of this theorem showing that the set of fixed points of F is the image of L by a lower and an upper preclosure operator. These preclosure operators are the composition of lower and upper closure operators which are defined by means of limits of stationary transfinite iteration sequences for F. In the same way we give a constructive characterization of the set of common fixed points of a family of commuting operators. Finally we examine some consequences of additional semi-continuity hypotheses.

1. Introduction. Let $L(\subseteq, \perp, \top, \cup, \cap)$ be a nonempty *complete lattice* with *partial ordering* $\subseteq$, *least upper bound* $\cup$, *greatest lower bound* $\cap$. The *infimum* $\perp$ of L is $\cap L$, the *supremum* $\top$ of L is $\cup L$. (Birkhoff's standard reference book [3] provides the necessary background material.) Set inclusion, union and intersection are respectively denoted by $\subseteq$, $\cup$ and $\cap$.

Let F be a *monotone* operator on $L(\subseteq, \perp, \top, \cup, \cap)$ into itself (i.e., $\forall X, Y \in L, \{X \subseteq Y\} \Rightarrow \{F(X) \subseteq F(Y)\}$).

The fundamental theorem of Tarski [19] states that the set $fp(F)$ of *fixed points* of F (i.e., $fp(F) = \{X \in L: X = F(X)\}$) is a nonempty complete lattice with ordering $\subseteq$. The proof of this theorem is based on the definition of the least fixed point $lfp(F)$ of F by $lfp(F) = \cap\{X \in L: F(X) \subseteq X\}$. The least upper bound of $S \subseteq fp(F)$ in $fp(F)$ is the least fixed point of the restriction of F to the complete lattice $\{X \in L: (\cup S) \subseteq X\}$. An application of the duality principle completes the proof.

This definition is not constructive and many applications of Tarski's theorem (specially in computer science (Cousot [5]) and numerical analysis (Amann [2])) use the alternative characterization of $lfp(F)$ as $\cup\{F^n(\perp): i \in N\}$. This iteration scheme which originates from Kleene [10]'s first recursion theorem and which was used by Tarski [19] for complete morphisms, has the drawback to require the additional assumption that F is *semi-continuous* $(F(\cup S) = \cup F(S)$ for every *increasing* nonempty *chain* S, see e.g., Kolodner [11]).

PORTUGALIAE MATHEMATICA
Vol. 38 Fasc. 1-2 — 1979

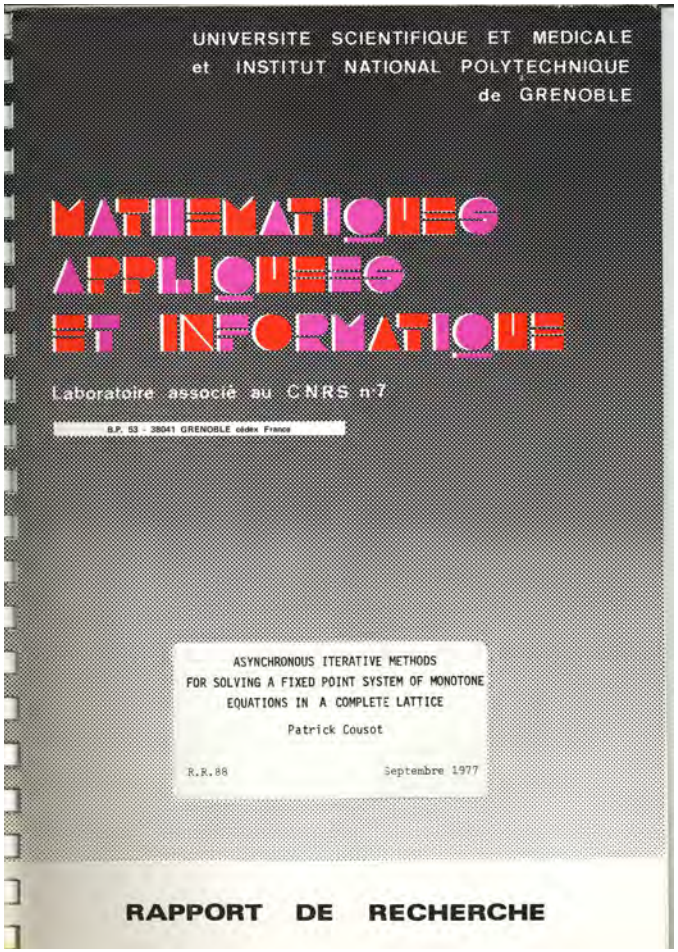
A CONSTRUCTIVE CHARACTERIZATION OF THE LATTICES OF ALL RETRACTIONS, PRECLOSURE, QUASI-CLOSURE AND CLOSURE OPERATORS ON A COMPLETE LATTICE

BY
PATRICK COUSOT AND RADHIA COUSOT *
Université de Metz
Faculté des Sciences
Ile du Saulcy
5700 Metz — France

1. Introduction

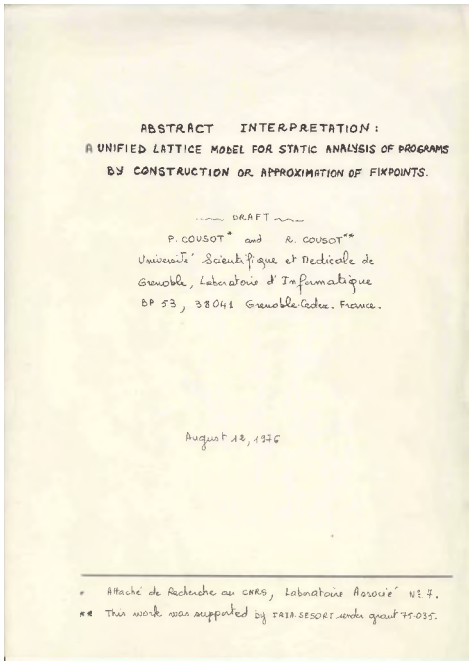
We give a constructive characterization of the complete lattices of all retractions, preclosure, quasi-closure and closure operators on a complete lattice. Our general approach is the following: in order to study the structure of the set $\Gamma \subseteq (L \to L)$ of operators ρ on a complete lattice L satisfying a given axiom A, we show that ρ has property A if and only if it is a fixed point of some monotone operator F on the complete lattice $(L \to L)$ proving that Γ is the set of fixed points of F. Then using Cousot & Cousot's constructive version of Tarski's lattice theoretical fixed point theorem, we constructively characterize the infimum, supremum, union and intersection of the complete lattice Γ which are defined by means of limits of stationary transfinite iteration sequences for F. Variants of this argument are used when F is a closure operator (in which case the constructive version of Tarski's theorem amounts to Ward's theorem) or when the operators with property A are the postfixed points of F or the common fixed points of two functionals. The reasoning is repeated when Γ is characterized by means of more than one axiom.

This work was supported by CNRS, Laboratoire Associé n.° 7.
(*) Attaché de Recherche au CNRS, CRIN-LA. 262.
Reçu Décembre 21, 1978.



On submitting...

- For POPL'77, we submit (on Aug. 12, 1976) copies of a two-hands written manuscript of 100 pages. The paper is accepted !



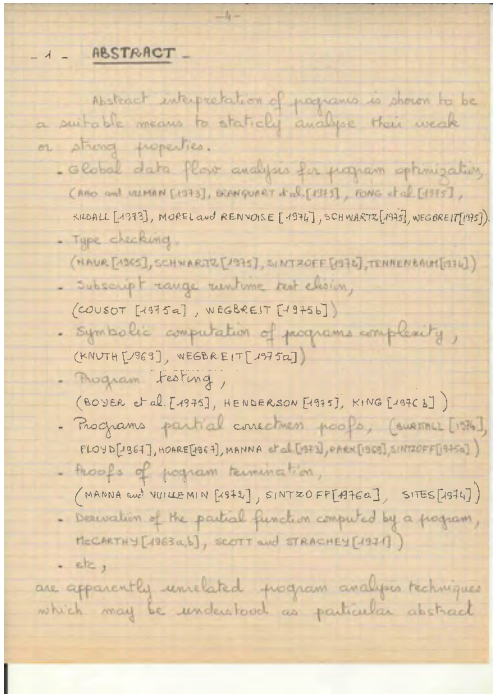
ABSTRACT INTERPRETATION :
A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS.

DRAFT

P. COUSOT* and R. COUSOT**
Université Scientifique et Médicale de Grenoble, Laboratoire d'Informatique
BP 53, 38041 Grenoble Cedex. France.

August 12, 1976

* Attaché de Recherche au CNRS, Laboratoire Associé N° 7.
** This work was supported by IRIA-SESORT under grant 75-035.

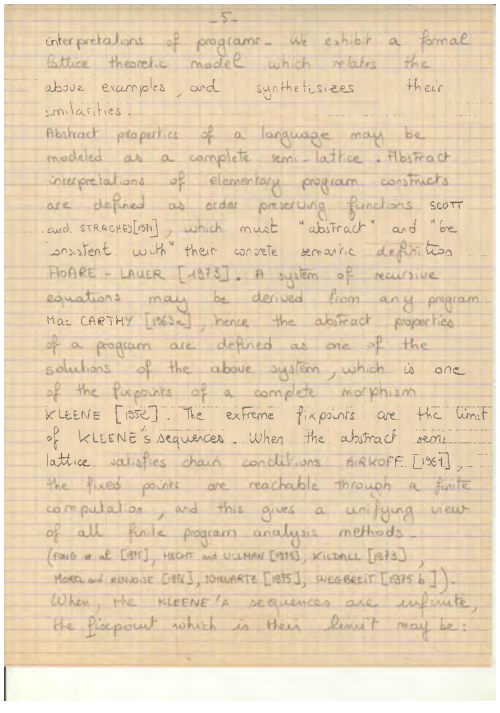


-1- ABSTRACT -

Abstract interpretation of programs is shown to be a suitable means to statically analyse their weak or strong properties.

- Global data flow analysis for program optimization, (AHO and ULMAN [1973], BRANQUART et al. [1975], KING et al [1975], KILDALL [1973], MOREL and RENVOISE [1976], SCHWARTZ [1975], WEGBREIT [1975]).
- Type checking, (NAUR [1965], SCHWARTZ [1975], SINTZOFF [1972], TENNENBAUM [1976])
- Subscript range runtime test elision, (COUSOT [1975a], WEGBREIT [1975b])
- Symbolic computation of programs complexity, (KNUTH [1963], WEGBREIT [1975a])
- Program testing, (BOYER et al. [1975], HENDERSON [1975], KING [1976b])
- Programs partial correctness proofs, (BURSTALL [1974], FLOYD [1967], HOARE [1967], MANNA et al [1973], PARK [1969], SINTZOFF [1972a])
- Proofs of program termination, (MANNA and WALDINGER [1974], SINTZOFF [1976a], SITES [1974])
- Derivation of the partial function computed by a program, McCARTHY [1963a,b], SCOTT and STRACHEY [1971])
- etc,

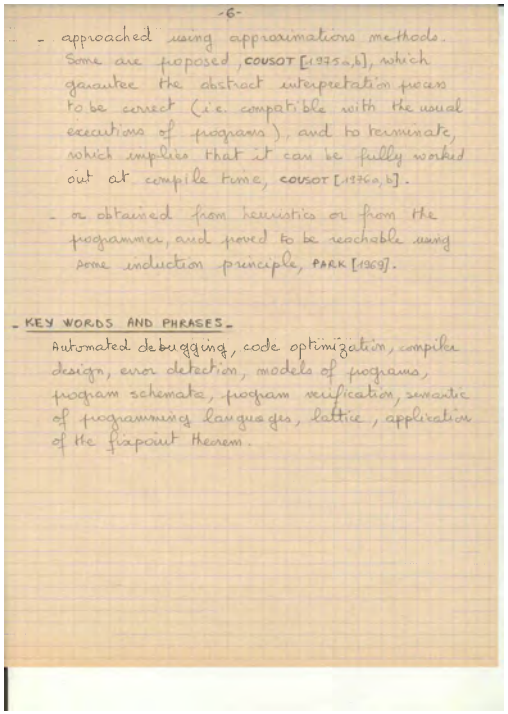
are apparently unrelated program analyses techniques which may be understood as particular abstract



-5-

interpretations of programs. We exhibit a formal lattice theoretic model which relates the above examples, and systhetisizes their similarities.

Abstract properties of a language may be modeled as a complete semi-lattice. Abstract interpretations of elementary program constructs are defined as order preserving functions SCOTT and STRACHEY [1971], which must "abstract" and "be consistent with" their concrete semantic definition HOARE - LAUER [1973]. A system of recursive equations may be derived from any program Mac CARTHY [1963a], hence the abstract properties of a program are defined as one of the solutions of the above system, which is one of the fixpoints of a complete morphism KLEENE [1952]. The extreme fixpoints are the limit of KLEENE's sequences. When the abstract semi-lattice satisfies chain conditions BIRKOFF [1967], the fixed points are reachable through a finite computation, and this gives a unifying view of all finite program analysis methods. (KING et al [1976], HECHT and ULLMAN [1975], KILDALL [1973], MOREL and RENVOISE [1976], SCHWARTZ [1975], WEGBREIT [1975b]). When the KLEENE's sequences are infinite, the fixpoint which is their limit may be:



-6-

- approached using approximations methods. Some are proposed, COUSOT [1975a,b], which garantee the abstract interpretation proves to be correct (i.e. compatible with the usual executions of programs), and to terminate, which implies that it can be fully worked out at compile time, COUSOT [1976a,b].
- or obtained from heuristics or from the programmer, and proved to be reachable using some induction principle, PARK [1969].

- KEY WORDS AND PHRASES -

Automated debugging, code optimization, compiler design, error detection, models of programs, program schemata, program verification, semantic of programming languages, lattice, application of the fixpoint theorem.

On convincing ...

- During PC's thesis defense, it was suggested that abstraction/approximation is useless since computers are finite and executions are timed-out (consequently, the second part of the thesis on fixpoint approximation/widening/narrowing/... is superfluous!)
- On the contrary, in 1978, during a seminar at Harvard [1], G. Birkhoff appears interested, according to his questions & feedback, in the effective computational aspects of lattice fixpoint theory

[1] invited by Ed. Clarke.

Deductive semantics is a property preserving abstraction of the relational semantics (in PC's thesis, 21 march 1978 also § 3 of POPL'79)

3.1.3 L'approche du point fixe à l'étude du comportement d'un système dynamique discret

DEFINITION 3.1.3.0.1

$wp \in (((S\times S)\to B)\to((S\to B)\to(S\to B)))$

$= \lambda\theta.\{\lambda\beta.[\lambda e_1.(\exists e_2 \in S : \theta(e_1,e_2) \text{ et } \beta(e_2))]\}$

i.e. pre

DEFINITION 3.1.3.0.2

$sp \in (((S\times S)\to B)\to((S\to B)\to(S\to B)))$

$= \lambda\theta.\{\lambda\beta.[\lambda e_2.(\exists e_1 \in S : \beta(e_1) \text{ et } \theta(e_1,e_2))]\}$

i.e. post transformer

Partant du fait que $\tau^* = eq \text{ ou } \tau^*\circ\tau = eq \text{ ou } \tau\circ\tau^*$, nous obtiendrons $wp(\tau^*)$ et $sp(\tau^*)$ comme points fixes d'une équation.

THEOREME 3.1.3.0.3

(a) - $((S\times S)\to B)(\Longrightarrow, \lambda(e_1,e_2).\text{faux}, \lambda(e_1,e_2).\text{vrai}, \text{OU}, \text{ET}, \text{non})$ est un treillis booléen complet,

(b) - Soient $a, b \in ((S\times S)\to B)$ alors $\lambda\alpha.[a \text{ ou } b\circ\alpha]$ et $\lambda\alpha.[a \text{ ou } \alpha\circ b]$ sont des morphismes complets pour la disjonction,

(c) - Soient $\tau \in ((S\times S)\to B)$ et eq la relation d'égalité alors $\tau^* = lfp(\lambda\alpha.[eq \text{ ou } \alpha\circ\tau]) = lfp(\lambda\alpha.[eq \text{ ou } \tau\circ\alpha])$.

fixpoint reflexive transitive closure

THEOREME 3.1.3.0.6.

Quels que soient $a, b \in ((S\times S)\to B)$ et $\beta \in (S\to B)$ nous avons:

- $wp(lfp(\lambda\alpha.[a \text{ ou } b\circ\alpha]))(\beta)$
  $= lfp(\lambda\alpha.[wp(a)(\beta) \text{ ou } wp(b)(\alpha)])$
  $= \underset{n\in\omega}{\text{OU}}\, wp(b^n)(wp(a)(\beta))$
- $sp(lfp(\lambda\alpha.[a \text{ ou } \alpha\circ b]))(\beta)$
  $= lfp(\lambda\alpha.[sp(a)(\beta) \text{ ou } sp(b)(\alpha)])$
  $= \underset{n\in\omega}{\text{OU}}\, sp(b^n)(sp(a)(\beta))$

fixpoint backward reachability

fixpoint forward reachability

iterative fixpoint computation

abstract transformer

concrete transformer

Preuve: Posons $h=\lambda\theta.[wp(\theta)(\beta)]$, $f=\lambda\alpha.[a \text{ ou } b\circ\alpha]$ et $g = \lambda\alpha.[wp(a)(\beta) \text{ ou } wp(b)(\alpha)]$ et montrons que $h\circ f = g\circ h$.

...

Fixpoint abstraction under commutativity with abstraction h

The principles (1977–79) are lasting !

- Define the semantics (operational, denotational, axiomatic, ...) of the programming language (as a ... / trace semantics / transition system / transformers / ...)
- Define the strongest property of interest (also called the *collecting semantics*)
- Express this collecting semantics in fixpoint form
- Define the abstraction/concretization compositionally (by composition of elementary abstractions and abstraction constructors/functors)
- Design the abstract proof / analysis semantics by calculus using [structural] abstraction i.e. abstract domain + abstract fixpoint
- Define the widening/narrowing
- Implement the abstract domain and fixpoint computation by elimination or iteration with convergence acceleration, if needed

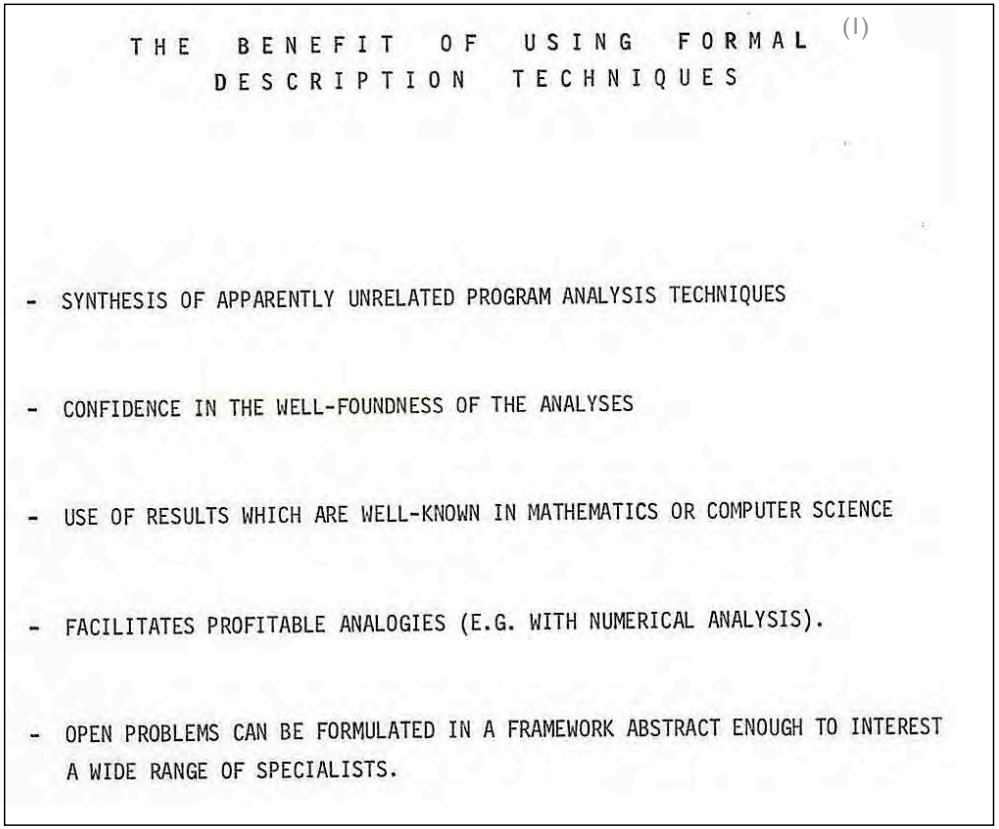
Very first industrial implementation

- Interval analysis was implemented in the AdaWorld compiler for IBM PC 80286 by J.D. Ichbiah and his Alsys SA corporation team in 1980–87.

Relevance of formal methods is an old question!

Panels from the IFIP congress 77:

- Mathematical theory of data-flow analysis, J.D. Ullman (P. Cousot, K. Kennedy, B. Rosen, R. Tarjan)
- The use and benefit of formal description techniques, E. Neuhold (E. Blum, B. Boehm, P. Cousot, J. De Bakker, S. Igarashi, M. Nivat, S. Owicki, R. Tennent)



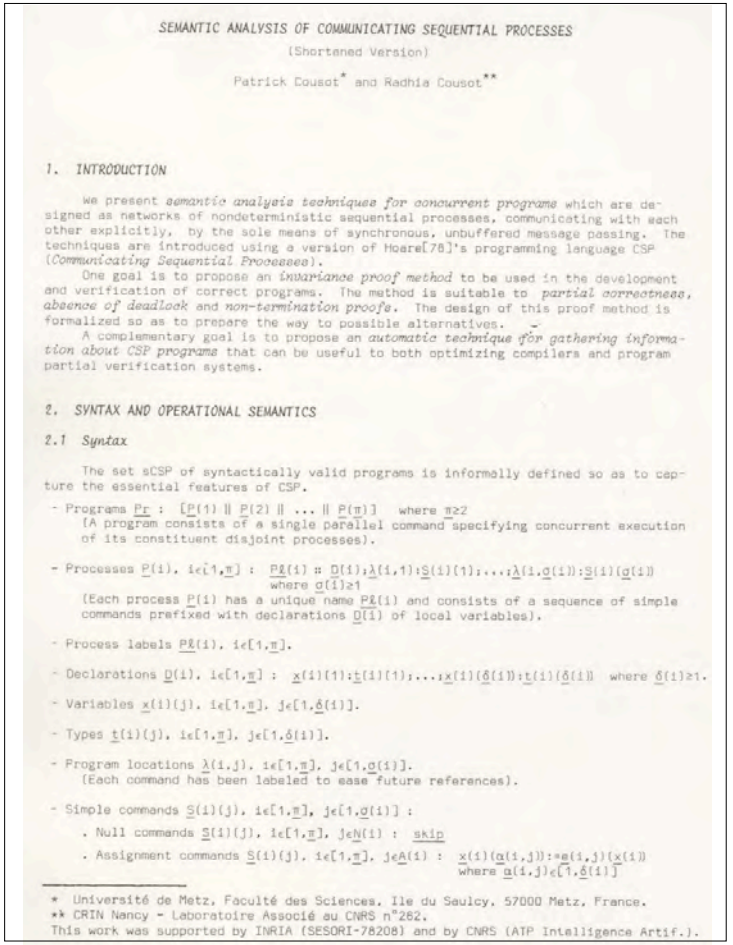
THE BENEFIT OF USING FORMAL DESCRIPTION TECHNIQUES [1]

- SYNTHESIS OF APPARENTLY UNRELATED PROGRAM ANALYSIS TECHNIQUES
- CONFIDENCE IN THE WELL-FOUNDNESS OF THE ANALYSES
- USE OF RESULTS WHICH ARE WELL-KNOWN IN MATHEMATICS OR COMPUTER SCIENCE
- FACILITATES PROFITABLE ANALOGIES (E.G. WITH NUMERICAL ANALYSIS).
- OPEN PROBLEMS CAN BE FORMULATED IN A FRAMEWORK ABSTRACT ENOUGH TO INTEREST A WIDE RANGE OF SPECIALISTS.

[1] ... a bit optimistic :-)

Progress is slow... e.g. parallelism

CSP



SEMANTIC ANALYSIS OF COMMUNICATING SEQUENTIAL PROCESSES

(Shortened Version)

Patrick Cousot* and Radhia Cousot**

1. INTRODUCTION

We present *semantic analysis techniques for concurrent programs* which are designed as networks of nondeterministic sequential processes, communicating with each other explicitly, by the sole means of synchronous, unbuffered message passing. The techniques are introduced using a version of Hoare[78]'s programming language CSP (*Communicating Sequential Processes*).

One goal is to propose an *invariance proof method* to be used in the development and verification of correct programs. The method is suitable to *partial correctness*, *absence of deadlock* and *non-termination proofs*. The design of this proof method is formalized so as to prepare the way to possible alternatives.

A complementary goal is to propose an *automatic technique for gathering information about CSP programs* that can be useful to both optimizing compilers and program partial verification systems.

2. SYNTAX AND OPERATIONAL SEMANTICS

2.1 Syntax

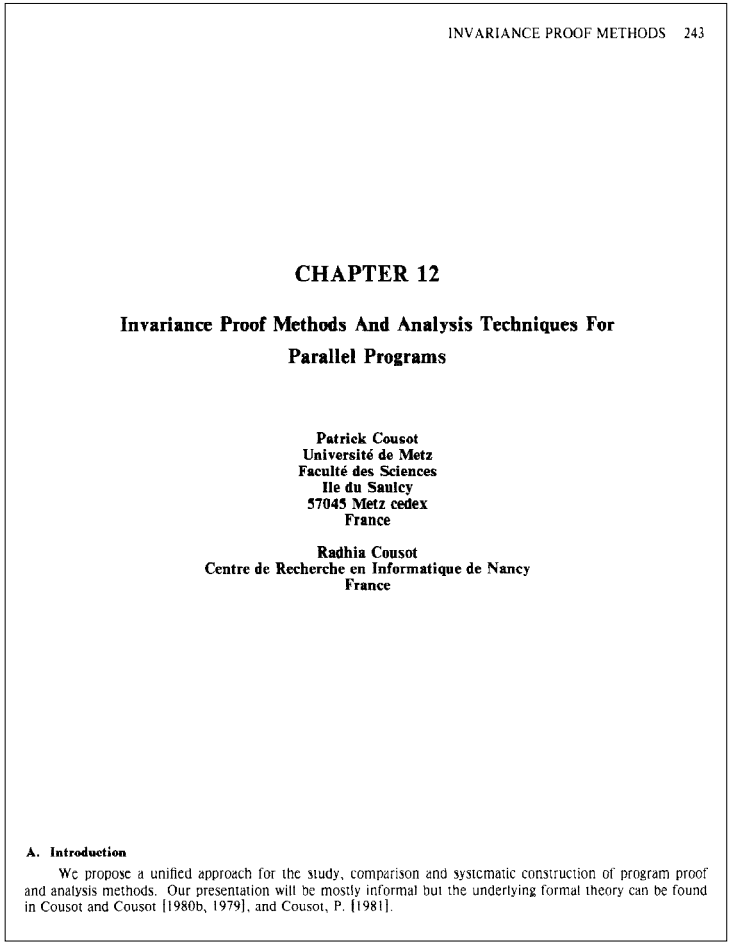
The set sCSP of syntactically valid programs is informally defined so as to capture the essential features of CSP.

- Programs Pr : [P(1) || P(2) || ... || P(π)] where π≥2
  (A program consists of a single parallel command specifying concurrent execution of its constituent disjoint processes).
- Processes P(i), i∈[1,π] : P&(i) :: D(i);λ(i,1):S(i)(1);...;λ(i,σ(i)):S(i)(σ(i)) where σ(i)≥1
  (Each process P(i) has a unique name P&(i) and consists of a sequence of simple commands prefixed with declarations D(i) of local variables).
- Process labels P&(i), i∈[1,π].
- Declarations D(i), i∈[1,π] : x(i)(1):t(i)(1);...;x(i)(δ(i)):t(i)(δ(i)) where δ(i)≥1.
- Variables x(i)(j), i∈[1,π], j∈[1,δ(i)].
- Types t(i)(j), i∈[1,π], j∈[1,δ(i)].
- Program locations λ(i,j), i∈[1,π], j∈[1,σ(i)].
  (Each command has been labeled to ease future references).
- Simple commands S(i)(j), i∈[1,π], j∈[1,σ(i)] :
  . Null commands S(i)(j), i∈[1,π], j∈N(i) : skip
  . Assignment commands S(i)(j), i∈[1,π], j∈A(i) : x(i)(α(i,j)):=e(i,j)(x(i)) where α(i,j)∈[1,δ(i)]

* Université de Metz, Faculté des Sciences, Ile du Saulcy, 57000 Metz, France.
** CRIN Nancy - Laboratoire Associé au CNRS n°262.
This work was supported by INRIA (SESORI-78208) and by CNRS (ATP Intelligence Artif.).

Cited by 33

Shared memory



CHAPTER 12

Invariance Proof Methods And Analysis Techniques For Parallel Programs

Patrick Cousot
Université de Metz
Faculté des Sciences
Ile du Saulcy
57045 Metz cedex
France

Radhia Cousot
Centre de Recherche en Informatique de Nancy
France

A. Introduction

We propose a unified approach for the study, comparison and systematic construction of program proof and analysis methods. Our presentation will be mostly informal but the underlying formal theory can be found in Cousot and Cousot [1980b, 1979], and Cousot, P. [1981].

Cited by 21

Google scholar

- a.o. THÉSÉE is in progress

Abstract interpretation: Past

Diffusion...

- from http://30yai.di.univr.it/ :

... popularization ...

jonny goes to england

Abstract Interpretation

with 2 comments

(source ilachinski.com)

I realized I don't write a lot about what I'm actually doing day in day out for work. This is partly because it is thought as "hard to describe" to non-computer science people. Today, I want to describe a method which is widely used in the software verification community in an easy way.

The technique is called *Abstract Interpretation* (AI from now on) and has been first described by Cousot in 1976/77 [1]. Now, to describe AI I'll directly start with an example also partly taken from [1]:

• • •

[1] Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California, January 17 – 19, 1977). POPL '77. ACM, New York, NY, 238-252. DOI= http://doi.acm.org/10.1145/512950.512973

http://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml

2 Responses

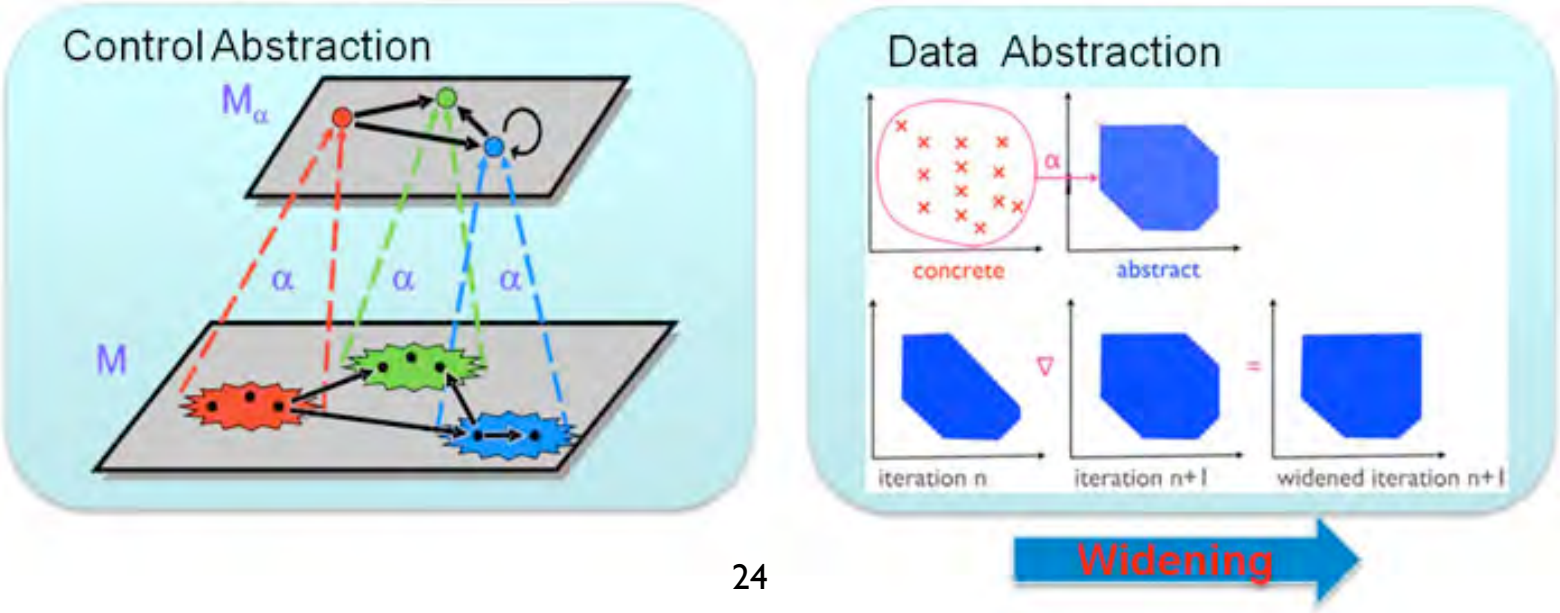
1. Wow, the paper is from 1977.
2. This was a very, very good post. Now I grokked what you really are up to! Please keep updating as time goes by.

   The idea is very beautiful, almost mythological in the core: one does abstractions and more abstractions in the level of ideas, then creates a rule to collapse the whole structure to the concrete world – to make it "real", one utters the words which do the linking from the divine realm of angels, infinity and brightness to the domain of humans, limitedness and dirt. (Yeah I guess I've read too much Joseph Campbell lately… 😃

From http://jonnyengland.wordpress.com/2007/12/30/abstract-interpretation/

... and perduring misunderstandings

- State-based versus [abstract] property-based reasoning
- Logical versus algebraic formalization
- Empirical versus calculational design
- Finite versus infinite abstraction
- Checking versus inference
- Bug finding versus verification



Abstract interpretation: Present

Domains of applications

- Syntax
- Semantics
- Proofs
- Static analysis
- Data-flow analysis
- Model-checking
- Control-flow analysis
- Types
- Data structure/heap analysis
- Abstract domains (numerical & symbolic)
- Predicate abstraction
- Refinement
- Strong Preservation
- Program transformation
- Program optimization
- Parallelization
- WCET (cache, pipeline)
- Watermarking
- Information hiding
- Code obfuscation
- Malware detection
- Termination
- Computer security
- Computational biology
- ...

Industrial diffusion

The ASTRÉE Static Analyzer

Centre National de la Recherche Scientifique — École Normale Supérieure — INRIA (since Sep. 2007)

Participants:

Patrick Cousot (project leader), Radhia Cousot, Jérôme Feret, Antoine Miné, Xavier Rival

Former participants:

Bruno Blanchet (Nov. 2001 — Nov. 2003), David Monniaux (Nov. 2001 — Aug. 2007), Laurent Mauborgne (Nov. 2001 — Aug. 2010).

Contact(‡): ostree@ens.fr — http://www.astree.ens.fr/

ASTRÉE stands for *Analyseur statique de logiciels temps-réel embarqués* (real-time embedded software static analyzer). The development of ASTRÉE started from scratch in Nov. 2001 at the Laboratoire d'Informatique of the École Normale Supérieure (LIENS), initially supported by the ASTRÉE project, the Centre National de la Recherche Scientifique, the École Normale Supérieure and, since September 2007, by INRIA (Paris—Rocquencourt).

Objectives of ASTRÉE

ASTRÉE is a static program analyzer aiming at *proving* the absence of *Run Time Errors* (RTE) in programs written in the C programming language. On personal computers, such errors, commonly found in programs, usually result in unpleasant error messages and the termination of the application, and sometimes in a system crash. In embedded applications, such errors may have graver consequences.

ASTRÉE analyzes structured C programs, with complex memory usages, but without dynamic memory allocation and recursion. This encompasses many embedded programs as found in earth transportation, nuclear energy, medical instrumentation, aeronautic, and aerospace applications, in particular synchronous control/command such as electric flight control [30], [31] or space vessels maneuvers [32].

Industrial Applications of ASTRÉE

The main applications of ASTRÉE appeared two years after starting the project. Since then, ASTRÉE has achieved the following unprecedented results on the static analysis of synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language:

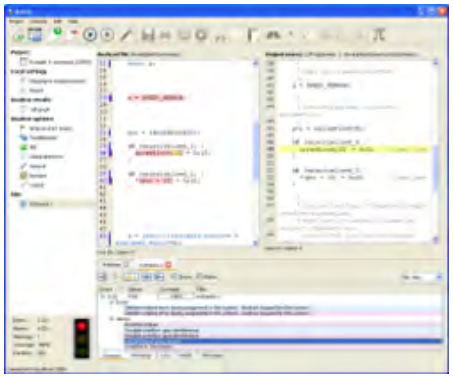
- In Nov. 2003, ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1^{h}20 on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon™ 64 using 580 Mb of memory).

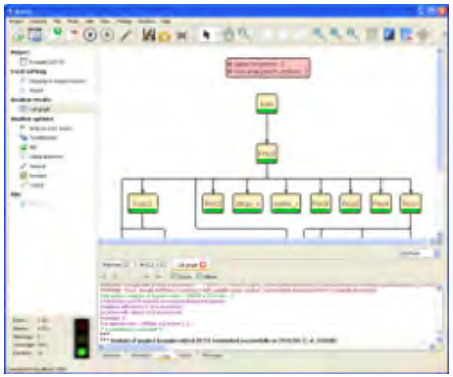
- From Jan. 2004 on, ASTRÉE was extended to analyze the electric flight control codes then in development and test for the A380 series. The operational application by Airbus France at the end of 2004 was just in time before the A380 maiden flight on Wednesday, 27 April, 2005.

- In April 2008, ASTRÉE was able to prove completely automatically the absence of any RTE in a C version of the automatic docking software of the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station [32].

Astrée Run-Time Error Analyzer

Astrée is a static program analyzer that proves the absence of run-time errors (RTE) in safety-critical embedded applications written or automatically generated in C.





Astrée analyzes structured C programs with complex memory usages, but without recursion or dynamic memory allocation. This targets embedded applications as found in earth transportation, nuclear energy, medical instrumentation, aeronautics and space flight, in particular synchronous control/command such as electric flight control.

Which run-time properties are analyzed by Astrée?

Astrée analyses whether the C programming language is used correctly and whether there can be any run-time errors during any execution in any environment. This covers:

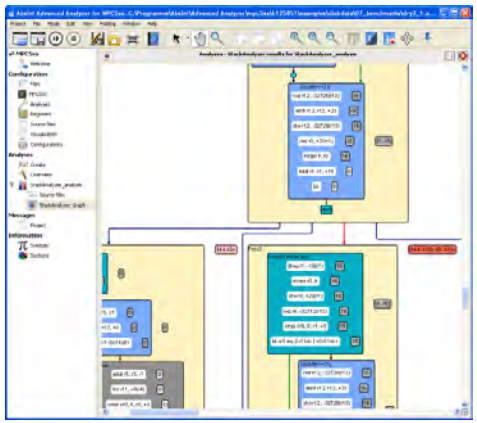
- Any use of C that has undefined behavior according to ISO/IEC 9899:1999, the international norm governing the C programming language. Examples include division by zero or out-of-bounds array indexing.
- Any use of C that violates hardware-specific aspects as defined by ISO/IEC 9899:1999, e.g. the size of integers and arithmetic overflow.
- Any potentially harmful or incorrect use of C that violates user-defined programming guidelines, such as no modular arithmetic for integers (even if this might be the hardware choice).
- Any violation of optional user-defined assertions to prove additional run-time properties (similar to `assert` diagnostics).

In addition to that, Astrée reports code that is guaranteed to be unreachable for all possible inputs and each program execution under any circumstances.

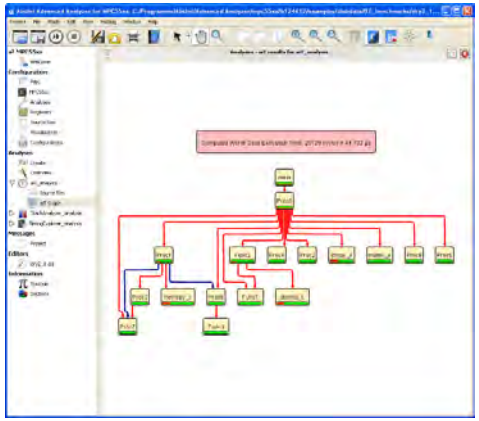
Astrée can be customized and integrated into established tool-chains.

... and many other abstract interpreters

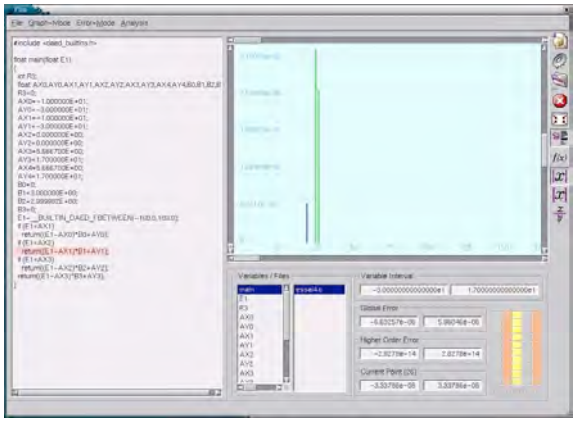
StackAnalyzer



AiT

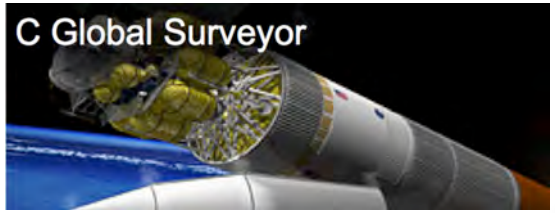


Fluctuat



SmallFoot

Julia



Cao

TERMINATOR

proof tools for termination and liveness

TVLA

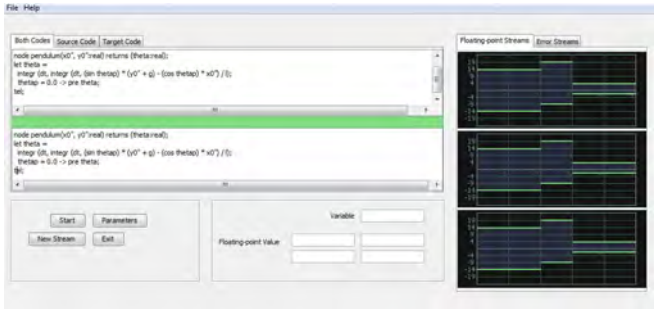
3-Valued Logic Analysis Engine

Sparrow

The Early Bird

Space invador

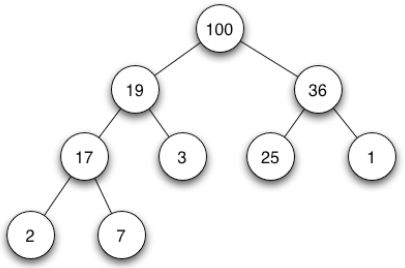
Sardana



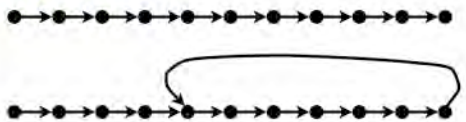
Clousot

...and more theoretical work: safety is OK, liveness is in progress and other properties are still unexplored

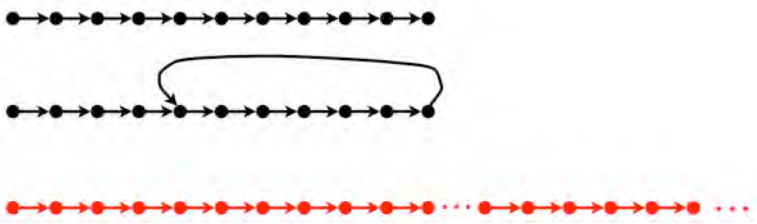
- Discrete symbolic data structures (e.g. heap)



- Under-approximation
- Liveness:
  - Finite systems:

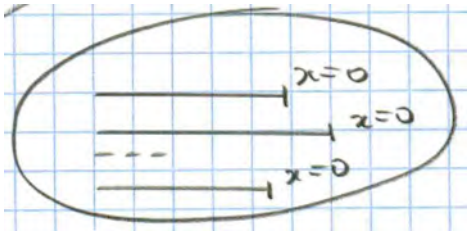


  - Infinite systems:

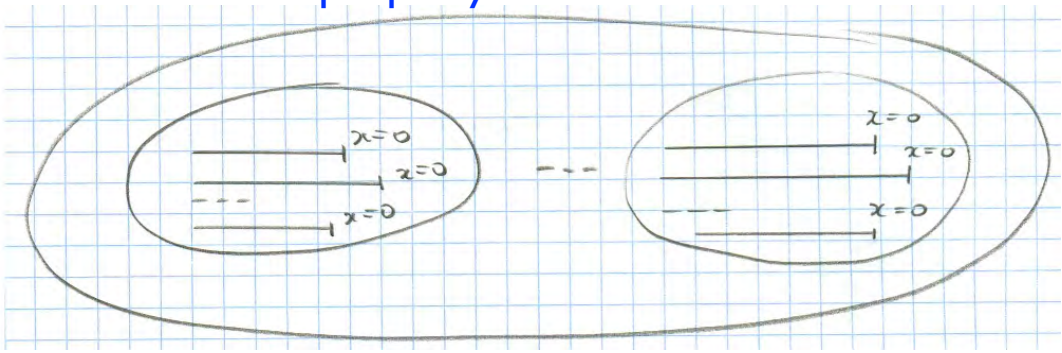


- Non-safety/liveness properties:

Trace semantics:



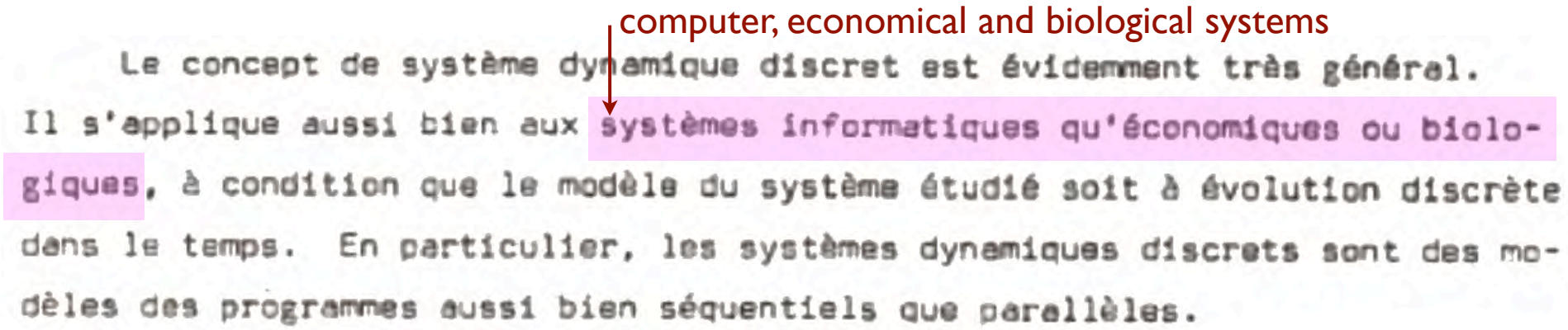
Trace semantics property:



Abstract interpretation: Future

Prospective ideas...

- The future is very hard to predict, e.g. 1978:



- More properties:
  - Security (not dynamically checkable)
  - ...
- More systems and tools:
  - Parallel and distributed systems,
  - Cyber-physical (continuous+discrete)
  - Biological, financial, ...
- Better practices:
  - Verification from design to implementation

Conclusion

Formal methods – just a Euro-science?

- For abstract interpretation, may be at the beginning
- Immediately recognized and welcomed in the US compared to Grenoble (e.g. invitations in 1977 at IBM by W. Miranker (fixpoints) and P. Goldberg (program analysis), and IFIP Congress by E. Neuhold (semantics) and J.D. Ullman (DFA), etc.)
- Nevertheless, early take off was mainly European (C. Hankin, M. Hermenegildo, J. Hughes, N. Jones, J. Launchbury, G. Levi, A. Mycroft, R. Wilhelm, ...)
- *Formal* is often understood as opposed to *experimental* and *practical*, both in Europe and the US
- This will not necessarily be the case everywhere in the world where mathematical skills and training is considered helpful for computer science (China, India, ...)