Abstract Interpretation: Achievements and Perspectives

Patrick Cousot

D partement d'informatique, cole normale sup rieure
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr, http://www.di.ens.fr/~cousot

Abstract— **Computerized modern societies are highly fragile to software bugs. Traditional testing methods hardly scale up for large safety critical systems as found in avionics, automotive, healthcare, e-commerce and security industry. As a viable alternative, static analysis consists in determining and verifying statically dynamic properties of programs. This is completely automatic (since programs are not actually executed) and covers all possible cases (as opposed to testing). This approach has had significant success stories and its industrialization recently started. Since the program total verification problem is undecidable, the key idea is that of approximation, as formalized by the theory of abstract interpretation. The scope of application of abstract interpretation ranges from the theoretical design of hierarchies of the semantics of programming languages to the practical design of generic program static analyzers.**

Keywords— **Abstract interpretation, semantics, verification, program static analysis.**

I. Introduction

These 25 last years, the performances of the computer hardware have been multiplied by 10^4 to 10^6. This is a technical revolution. To illustrate this 10^6 order of magnitude, this is the factor between the working force of a roman slave and the power of a nuclear plant unit or that between the distance of Paris to Nice to that of Earth to Mars. The immediate consequence is that the size of the programs executed on these computers has grown in similar proportions. For example a text editor for the general public contains more than 1 700 000 lines of C organized in 20 000 procedures and 400 files. Unfortunately neither the intellectual capacities of the programmers nor the sizes of the design and maintenance teams can grow in similar proportions. The errors in software, whether anticipated and corrected in time (like the Y2K bug) or unforeseen (like the failure of the 5.01 Ariane launcher flight) are frequent. They can have catastrophic consequences which are very costly and sometimes inadmissible (which is the case of transportation embedded software). The difficulty to prevent and find errors grows faster than the size of programs which can now be really huge. Classical software verification methods (such as code reviews, simulations, tests, etc.) do not scale up. The production of reliable software, their maintenance and their evolution over long periods of time (20 to 30 years) has become a fundamental concern to computer scientists. Computer scientists must widen the set of methods and tools used to strive against software bugs. This is necessary to cope with their responsibilities, satisfy to future regulations which will inevitably be established and to avoid that the failure of computerized systems becomes an important societal problem,

The basic idea of *static program analysis* is to use the computer to discover programming errors. The problem of programming computers so as to analyze the work that they will be given to do, as described by a program, *before* executing effectively this program is extremely hard. This follows from undecidability and complexity problems. One must therefore resort to compromises which consists in considering only an approximation of the possible run-time behaviors of the program. For example, program debugging consists in exploring a few examples of possible executions which, hopefully are well-chosen enough to reveal bugs. Another example is model-checking which can be fully applied to finite models of programs only but are generally incomplete for infinite state systems. Such coarse approximation methods are particular cases of *abstract interpretations* of program semantics. Abstract interpretation provides the the right theory to develop, understand, relate, design and automate formal methods to reason about programs.

II. Computation models and semantics

A *computation or execution model* is a formal mathematical description of the operations executed in the course of time by a computer running a program, including their internal effects on the machine (mainly on the memory) and external effects in interaction with the environment, in all possible conditions.

A very simple computation model is that of maximal execution sequences. To each discrete time i of the computation, a state σ_i memorizes the instantaneous values of the characteristic components of the machine (variables, memories, registers, etc.) and of the environment (clocks, timers, captors, etc.). The program is therefore part of the state σ_i which also indicates which execution is standing in the program at time i. A possible computation of the program is modelled by a trace representing the evolution of the machine state as program execution progresses. A finite execution is modelled by a finite trace that is a finite sequence $\sigma = \sigma_0 \ldots \sigma_{n-1}$ of states (of length $|\sigma| = n$). An execution which does not terminate is modelled by an infinite trace that is an infinite sequence $\sigma = \sigma_0\sigma_1 \ldots$ of states (of length $|\sigma| = \infty$). In general, there are many possible computations (corresponding for example to all possible initial states or to all non-deterministic interactions with the environment such as input/output) so one must reason on sets of traces. One can for example consider the set of maximal traces (no finite trace being the prefix of a longer one).

The *semantics* of a program P is a computation model describing the *effective* executions $[\![P]\!]$ of the program in all possible environments. The semantics of a language is given for each syntactically correct program of this language. It follows that the semantics of a software is provided by the semantics of the programming language in which it is written. In particular this semantics specifies conditions under which no run-time error can appear during execution. The absence of run-time errors is a minimal specification of the software which can be checked and reveal programming errors (in general 10 to 40% of the program bugs can be found in that way).

There exists a great number of possible methods to describe programming language semantics (operational, denotational, axiomatic, etc.) which are all equivalent or are approximations of each other. They form a hierarchy [1], organized according to the precision of the description of the program behaviors during their execution, which can be understood by abstract interpretation.

In general semantics can be defined by fixpoints. One can give the underlying intuition for the maximal trace semantics (and by abstraction for a great number of other semantics) by observing that the set T of maximal traces generated by a transition system, that is a relation t on a set of states , is:

- the set of finite traces σ of length 1 reduced to a final state σ_0, without possible transition $(\forall s : \langle\sigma_0, s\rangle \notin t)$;
- the set of finite traces σ of length $n > 1$ starting by a transition $\langle\sigma_0, \sigma_1\rangle \in t$ followed by a finite trace $\sigma_1 \ldots \sigma_{n-1}$ of T;
- the set of infinite traces σ starting by a transition $\langle\sigma_0, \sigma_1\rangle \in t$ followed by a infinite trace $\sigma_1\sigma_2 \ldots$ of T ;

Formally the fixpoint $T = F(T)$ is :

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

In general there are many possible fixpoints (for example by taking no infinite trace). Therefore we consider the least fixpoint for the *computational partial ordering* $X \sqsubseteq Y$ if and only if X has more finite traces and less infinite traces than Y (so that in the $\sqsubseteq$-least fixpoint solution of (1) there are the less possible finite traces and the more possible infinite traces for the equation $T = F(T)$ to hold). It follows that the trace semantics is the $\sqsubseteq$-least fixpoint of F and the approximation of semantics amounts to the approximation of fixpoints.

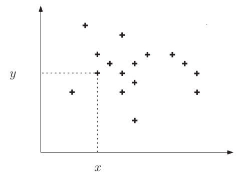
III. Specification and verification

The *specification* of a software is a computation model describing the *desirable* execution of this software in all possible environments. The minimal specification is the absence of runtime errors (arithmetic errors such as division by zero, bounded capacity overflows such as memory access by an array index outside of the array bounds, etc.). more complex specifications can be given using specification languages such as temporal logics. They also have a trace based semantics. For example $\Box P$ states that property P is always true in the future. More formally $\Box P$ specifies the set of traces such that P holds at each instant of time and so:

$$[\![\Box P]\!] \quad = \quad \{\sigma \mid \forall i : 0 \leq i < |\sigma| : \sigma_i \ldots \in [\![P]\!]\} \ .$$

The *verification* of a software P consists in proving that a semantics $[\![P]\!]$ of the software P satisfies a given specification S: $[\![P]\!] \subseteq [\![S]\!]$.

All interesting questions relative to the semantics of a non trivial program, such as its verification, are undecidable: no computer can always answer exactly these questions in a finite time for all possible programs. This results from the fact that the semantics of a program is not computable. For example, on can define mathematically the semantics of a program as the fixpoint of an equation but no computer can solve this equation (in particular because the semantics must take into account the possible existence of infinite program computations). A fundamental idea is therefore to consider *approximations*.



$$\{\ldots, \langle 5,\ 7\rangle, \ldots, \langle 13,\ 21\rangle, \ldots\}$$

Fig. 1. Set of points

IV. Abstract Interpretation

Abstract Interpretation [3], [4] is a theory of discrete approximation which can be applied to the semantics of (specification or programming) languages. Abstract Interpretation formalizes the idea that a semantics can be more or less precise according to the considered observation level [1].

For example an abstract model of the trace semantics T is the *transition semantics* t (or small-step operational semantics) which is the set of pairs $\langle\sigma_i, \sigma_{i+1}\rangle$ of states appearing along at least one trace of the trace semantics:

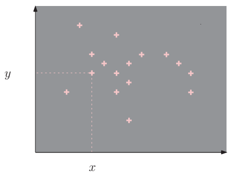
$$\begin{aligned} t \quad &= \quad \alpha_o(T) \\ &= \quad \{\langle\sigma_i, \sigma_{i+1}\rangle \mid \sigma_0 \ldots \sigma_i\sigma_{i+1} \ldots \in T\} \ . \end{aligned}$$

The set $\gamma_o(t)$ of maximal traces that can be rebuilt back from the transitions is the set of sequences of states linked by a transition and, in the case of finite sequences, terminated by a final state without possible transition:

$$\begin{aligned} \gamma_o(t) \quad = \quad & \{\sigma \mid \forall i : 0 < i + 1 < |\sigma| : (\langle\sigma_i, \sigma_{i+1}\rangle \in t) \wedge \\ & (|\sigma| = n \in \mathbb{N} \implies \forall s : \langle\sigma_{n-1}, s\rangle \notin t)\} \ . \end{aligned}$$

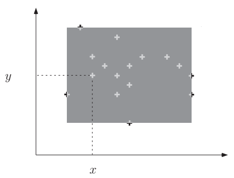
Another abstraction underlying *denotational semantics* consists in abstracting the finite traces by the pair of their initial and final states:

$$\alpha_d(\sigma_0\sigma_1 \ldots \sigma_{n-1}) \quad = \quad \langle\sigma_0, \sigma_{n-1}\rangle \ .$$



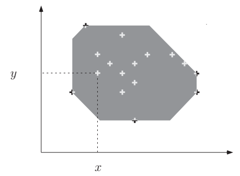
$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Fig. 2. Signs



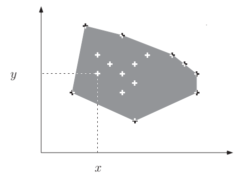
$$\begin{cases} x \in [3,\ 27] \\ y \in [4,\ 32] \end{cases}$$

Fig. 3. Intervals



$$\begin{cases} 3 \leq x \leq 27 \\ x + y \leq 88 \\ 4 \leq y \leq 32 \\ x - y \leq 61 \end{cases}$$

Fig. 4. Octagons



$$\begin{cases} 7x + 31y \leq 325 \\ 21x + 7y \geq 0 \end{cases}$$

Fig. 5. Polyhedra

The infinite traces are abstracted by the initial state followed by $\perp$ denoting non termination:

$$\alpha_d(\sigma_0\sigma_1\ldots) \quad = \quad \langle\sigma_0, \perp\rangle \ .$$

The abstraction of a set of traces is the set of abstractions of the individual traces in the set:

$$\alpha_d(X) \quad = \quad \{\alpha_d(\sigma) \mid \sigma \in X\} \ .$$

The last example, *natural semantics* simply ignores the infinite behaviors and so the extra abstraction is:

$$\alpha_n(X) \quad = \quad \{\langle s, s'\rangle \in X \mid s' \neq \perp\} \ .$$

Let us remark (1) that the l'*abstraction* α is monotone ($T \subseteq T' \Rightarrow \alpha(T) \subseteq \alpha(T')$) and (2) so is the *concretization* γ. The abstract semantics $t = \alpha(T)$ is an approximation or abstraction of the concrete semantics in that the concrete semantics $\gamma(t)$ that can be rebuilt from the abstract semantics t is in general larger than the initial concrete semantics T: (3) $T \subseteq (\gamma(\alpha(T))$. Finally the concrete semantics $\gamma(t)$ loose no information on the abstract semantics t in that $t = \alpha(\gamma(t))$ and more generally (4) $\alpha(\gamma(t)) \subseteq t$. Properties (1) to (4) characterize *Galois connections* which are used in the theory of abstract interpretation initiated by the seminal papers [3], [4], [5]. In practice one often use weaker hypotheses [6], in particular when there is no best approximation (for example there is no smallest convex polyhedron containing a disk).

The abstraction formalizes a loss of information, which does not allow to answer all possible questions about the program semantics hence its possible executions. All answers given by the abstract semantics are always correct for the concrete semantics. However, in general, some concrete questions cannot be answered exactly when considering the abstract semantics only.

For example the trace, denotational and natural semantics allow to answer question (1): "Can the program execution starting from state x terminate in state y?". Only the trace and denotational semantics can answer the question (2) "Does any execution starting from state x always terminate?". The only valid answer with the natural semantics is "I do not know" since all infinite behaviors are ignored. Finally the trace semantics can answer the question (3) "can state x be immediately followed by state y during a program execution" while the denotational and natural semantics do not allow this question to be answered directly since all intermediate states in the computation are forgotten. This shows that the more precise/concrete semantics can answer more questions while the abstract/approximate semantics are more simple. Two semantics may not be comparable. This is the case of the natural semantics (which can answer question (1) but not question (3)) and of the transition semantics (which can answer question (3) but not question (1)). More generally, the semantics can be ordered in a complete lattice according to their relative precision [5], [1]. In practice one must find in this infinite lattice the interesting semantics.

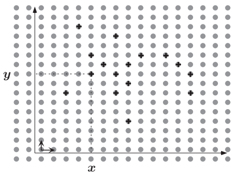
V. Static Analysis

If the approximation is coarse enough, the abstraction of a semantics provides a version of this semantics which is less precise (and therefore less questions can be answered) but which is computable (so that the questions can be answered by a computer). By effective computation of the abstract semantics, the computer can analyze the behavior of programs and software *before executing them*. This can be used to discover programming errors before they lead to catastrophes [7], which is essential for computer-based critical systems (for example: planes, launchers, nuclear plants, etc.).

To give the intuition of the approximations which are used in practice, let us consider a *safety analysis*. Safety properties (such as absence of run-time errors) specify that something bad cannot happen or equivalently that program execution must remain in good or healthy states. This notion can be formalized by the composition of abstractions as explained below. The first abstraction is that of a trace by the set of states appearing along this trace:

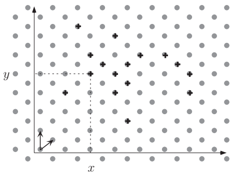
$$\alpha_s(\sigma) \quad = \quad \{\sigma_k \mid k \in [0, |\sigma|[\} \ .$$

The approximation of a set X of traces is then the set of states appearing at least once on at least one trace of the



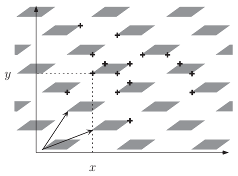
$$\begin{cases} x = 19 \bmod 88 \\ y = 19 \bmod 99 \end{cases}$$

Fig. 6. Simple congruences



$$\begin{cases} 3x + 5y = 8 \\ \qquad \bmod 27 \\ 2x - 9y = 23 \\ \qquad \bmod 53 \end{cases}$$

Fig. 7. Relational congruences



$$\begin{cases} 3x + 7y \in [2, 31] \\ \qquad \bmod 10 \\ 2x - 5y \in [0, 9] \\ \qquad \bmod 11 \end{cases}$$

Fig. 8. Trapezoidal congruences

set:

$$\alpha_s(X) \quad = \quad \bigcup_{\sigma \in X} \alpha_s(\sigma) \ .$$

$\alpha_s(X)$ is an *invariant* which means that, understood as a property, all states which are reachable from the initial states during program execution must satisfy that property.

Assume that a state s consists of a control state designating a program point $\ell \in \mathcal{C}$ and of variables $\mathtt{X_1}, \ldots, \mathtt{X_n}$ taking integer values $\langle x_1, \ldots, x_n \rangle \in \mathbb{Z}$. The *global invariant* $\alpha_s(X)$ is isomorphic to the vector of *local invariants* associating with each program point the set of tuples of possible values of the variables at that point:

$$\alpha_l(Y) \quad = \quad \prod_{\ell \in \mathcal{C}} \{ \langle x_1, \ldots, x_n \rangle \mid \langle \ell, \langle x_1, \ldots, x_n \rangle \rangle \in Y \} \ .$$

Intuitively the global invariant always holds during program execution while the local invariant attached to a program point holds whenever execution reaches that control point.

In this way the computation of the program set of traces is approximated by a set of points of $\mathbb{Z}^n$ which, in practice is infinite or large enough not to be computable. So further approximations must be applied. For $n = 2$, figures 2 to 8 provide examples of approximations of the set of vectors of integers given in figure 1 which are frequently used in static program analysis. The first group of approximations in figures 1 to 5 consist in considering envelops of the points [5], [8], [9], [10] while the second group of approximations in figures 6 to 8 is based on the idea of congruence [11], [12], [13].

The approximations involved in figures 2, 3 and 6 are *non relational.* Such non relational approximations ignore the relationships between the values of the program variables:

$$\alpha_r(X) \quad = \quad \prod_{i=1}^{n} \{ x_i \mid \exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n : \\ \langle x_1, \ldots, x_i, \ldots, x_n \rangle \in X \} \ .$$

In the case of such non relational approximations, it remains to abstract a set of integers, for example by the interval of its minimal and maximal values $\alpha_i(Z) = [\min Z, \max Z]$. Finally, the abstraction of a set of traces by intervals of values of the integer variables attached to each program control point is obtained by composition of abstractions, $\alpha(T) = \alpha_i(\alpha_r(\alpha_l(\alpha_s(T))))$.

The theory of abstract interpretation shows that the abstraction $\alpha(T)$ is solution of a fixpoint equation designed by abstraction of the equation (1) defining the trace semantics T. let us consider a very simple example [8]:

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

The corresponding fixpoint system of equations is the following:

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

Equation (4) expresses the fact that the set of values of `x` at program point 3 is the set of values of `x` at program point 2 augmented by the value 1. Equation (3) expresses the fact that the set of values of `x` at program point 2 is the set of values of `x` at program point 1 (entry in the loop) or at program point 3 (following iterations) satisfying the test `x < 10000`. Hence these equations are an abstract or simplified version of the Floyd/Naur/Hoare invariance proof method [14].

The most common method to solve these equations is iterative, starting from the infimum and using any chaotic or asynchronous iteration strategy [3]. The increasing iteration for equations (2)–(5) is given in figure 9.

Intuitively the chosen iteration strategy consists in following simultaneously all possible execution paths, without omitting any one, replacing sets of concrete values by interval abstract values. In general the increasing iteration will not be convergent. A *widening* operation must be used to speed up convergence. For example the na ve widening for intervals which is used in figure 10 consists in extrapolating unstable bounds to infinity. One obtain an over-approximation of the fixpoint, which can be improved by a decreasing iteration (which finite convergence must, in general, be enforced by a *narrowing* operation). The iteration with convergence speed-up for equations (2)–(5) is given in figure 10.

After extension to recursive procedures [15], the interval analysis can be used for imperative languages such as [16], C or Java.

tattooing of software, etc.

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

Fig. 9. Increasing iteration for equations [2]–[5]

The minimal specification is that there should be no overflows. This can be easily checked by the analysis:

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

In general such run-time errors will be signaled as *certain*, *impossible* or *potential* (when the analysis is not precise enough to conclude), but the cover of the specification is always complete.

VI. Other Achievements and Perspectives

Most data structures manipulated by programs are not numerical. This is the case of the control structures (call graphs, recursion trees), of the data structures (such as search tress), communication structures (programs distributed on networks), information transfer structures (mobile code), etc. It is very difficult to find appropriate compact and precise computer representations of sets of suchobjects (languages, automata, trees, graphs, etc.) such that the various set theoretic operations which are used in the abstract equations can be implemented efficiently, so that memory size does not explode combinatorially and the approximations remain precise for complex and irregular sets. There is a lot of research done on this problem, see [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32] for a few solutions proposed in our teams.

This type of analysis by abstract interpretation can be extended to intermittent as well as invariant assertions [33] whence the idea of *abstract testing* [16] which generalizes model checking [34]. In practice such static analyzes have been successfully used for the static analysis of the flight software and the inertial central of the Ariane 5 launcher, which was a success for flights 502, 503 et l'ARD [7].

Numerous languages have been considered such as logic programming (see references in [35], [36]) and more recently mobile code [37], [38], [39], [40]. Some recent applications of static analysis by abstract interpretation concerns type inference (for undecidable systems) [41], abstract infinite state model checking [42], [34], the transformation and optimization of programs, automatic differentiation, the analysis of cryptographic protocols [43], the semantic

A lot remains to be done on fundamental problems such as:

- the analysis of complex control structures (such as higher-order recursive, parallel, distributed or mobile programs);
- the analysis of complex data structures (such as floating point numbers, non-linear approximations of sets of integers, dynamic allocation of data structures);
- the modularization and compositional design of analyzers;
- the analysis of complex properties of programs (such as probabilistic analyses [44], liveness properties with fairness hypothesis);
- etc.

Abstract interpretation has been recently industrialized by start-up enterprises ("AbsInt Angewandte Informatik GmbH" `http://www.absint.de` (Germany) in 1998 and "Polyspace Technologies" `http://www.polyspace.com` (France) in 1999. These enterprises have developed efficient and powerful static analyzers which are commercially available.

References

[1] P. Cousot, "Constructive design of a hierarchy of semantics of a transition system by abstract interpretation," *Theoret. Comput. Sci.*, To appear (preliminary version in [2]).

[2] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages.

[3] P. Cousot, *M thodes it ratives de construction et d'approximation de points fixes d'op rateurs monotones sur un treillis, analyse s mantique de programmes.* Th se d' tat s sciences math matiques, Universit scientifique et m dicale de Grenoble, Grenoble, 21 Mar. 1978.

[4] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in 4^{th} *POPL*, Los Angeles, pp. 238–252, ACM Press, 1977.

[5] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in 6^{th} *POPL*, San Antonio, pp. 269–282, ACM Press, 1979.

[6] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *J. Logic and Comp.*, vol. 2, pp. 511–547, Aug. 1992.

[7] P. Lacan, J. 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, ESA Publications, SP-422, 25–28 May 1998.

[8] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proc.* 2^{nd} *Int. Symp. on Programming*, pp. 106–130, Dunod, Paris, 1976.

[9] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *ACM SIGPLAN '89 PLDI*, Portland, pp. 41–53, 1989.

[10] P. Cousot and N. Halbwachs, "Automatic discovery of linear

$$
\begin{array}{lcllllllll}
 & & 0 & 1 & 2 & 3 & \overbrace{4 \quad\quad\quad 5}^{\text{widening}} & & 6 & 7 \\
\left\{\begin{array}{l} X_1 \\ X_2 \\ X_3 \\ X_4 \end{array}\right. &
\begin{array}{c} = \\ = \\ = \\ = \end{array} &
\begin{array}{c} \emptyset \\ \emptyset \\ \emptyset \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ \emptyset \\ \emptyset \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,1] \\ \emptyset \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,1] \\ [2,2] \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,2] \\ [2,2] \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,+\infty] \\ [2,2] \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,+\infty] \\ [2,10000] \\ \emptyset \end{array} &
\begin{array}{c} [1,1] \\ [1,9999] \\ [2,10000] \\ [10000,10000] \end{array}
\end{array}
$$

$$\underbrace{\quad 0 \ldots 6 \quad}_{\text{increasing iteration}} \quad \underbrace{\quad 6 \ldots 7 \quad}_{\text{decreasing iteration}}$$

Fig. 10. Iteration for equations [2]–[5] with widening/narrowing

restraints among variables of a program," in 5^{th} *POPL*, Tucson, pp. 84–97, ACM Press, 1978.

[11] P. Granger, "Static analysis of arithmetical congruences," *Int. J. Comput. Math.*, vol. 30, pp. 165–190, 1989.

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

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

[14] P. Cousot, "Methods and logics for proving programs," in *Formal Models and Semantics* (J. van Leeuwen, ed.), vol. B of *Handbook of Theoretical Computer Science*, ch. 15, pp. 843–993, Elsevier, 1990.

[15] P. Cousot and R. Cousot, "Static determination of dynamic properties of recursive procedures," in *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA* (E. Neuhold, ed.), pp. 237–277, North-Holland, 1977.

[16] F. Bourdoncle, "Abstract debugging of higher-order imperative languages," in *Proc. PLDI*, pp. 46–55, ACM Press, 1993.

[17] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," in *ACM Symposium on Language Design for Reliable Software*, Raleigh, ACM SIGPLAN Not. 12(3):77–94), 1977.

[18] P. Cousot and R. Cousot, "Formal language, grammar and set-constraint-based program analysis by abstract interpretation," in *Proc. 7^{th} FPCA*, La Jolla, pp. 170–181, ACM Press, 25–28 June 1995.

[19] P. Cousot and R. Cousot, "Abstract interpretation of algebraic polynomial systems," in *Proc. 6^{th} Int. Conf. AMAST '97* (M. Johnson, ed.), Sydney, LNCS 1349, pp. 138–154, Springer-Verlag, 13–18 Dec. 1997.

[20] R. Cridlig and . Goubault, "Semantics and analysis of Linda-based languages," in *Proc. 3^{rd} Int. Work. WSA '93* (P. Cousot, M. Falaschi, G. Fil , and A. Rauzy, eds.), Padova, LNCS 724, pp. 72–86, Springer-Verlag, 22–24 Sep. 1993.

[21] R. Cridlig, "Semantic analysis of shared-memory concurrent languages using abstract model-checking," in *Proc. PEPM '95*, La Jolla, ACM Press, 21–23 June 1995.

[22] A. Deutsch, "On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications," in *17^{th} POPL*, San Fransisco, pp. 157–168, ACM Press, Jan. 1990.

[23] A. Deutsch, "A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations," in *Proc. 1992 ICCL*, Oakland, pp. 2–13, IEEE Comp. Soc. Press, 20–23 Apr. 1992.

[24] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proc. PLDI*, Orlando, pp. 230–241, ACM Press, June 1994.

[25] A. Deutsch, "Semantic models and abstract interpretation techniques for inductive data structures and pointers, invited paper," in *Proc. PEPM '95*, La Jolla, pp. 226–229, ACM Press, 21–23 June 1995.

[26] L. Mauborgne, "Abstract interpretation using TDGs," in *Proc. 1^{st} Int. Symp. SAS '94* (B. Le Charlier, ed.), Namur, 20–22 Sep. 1994, LNCS 864, pp. 363–379, Springer-Verlag, 1994.

[27] L. Mauborgne, "Abstract interpretation using typed decision graphs," *Sci. Comput. Programming*, vol. 31, pp. 91–112, May 1998.

[28] L. Mauborgne, "Binary decision graphs," in *Proc. 6^{th} Int. Symp. SAS '99* (A. Cortesi and G. Fil , eds.), Venice, IT, 22–24 Sep. 1999, LNCS 1694, pp. 101–116, Springer-Verlag, 1999.

[29] N. Mercouroff, "An algorithm for analyzing communicating processes," in *Proc. 7^{th} Int. Conf. on Mathematical Foundations of Programming Semantics* (S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, eds.), Pittsburgh, pp. 312–325, Springer-Verlag, 25–28 Mar. 1991.

[30] J. Stransky, "A lattice for abstract interpretation of dynamic (LISP-like) structures," *Inform. and Comput.*, vol. 101, pp. 70–102, 1992.

[31] F. V drine, "Binding-time analysis and strictness analysis by abstract interpretation," in *Proc. 2^{nd} Int. Symp. SAS '95* (A. Mycroft, ed.), Glasgow, 25–27 Sep. 1995, LNCS 983, pp. 400–417, Springer-Verlag, 1995.

[32] A. Venet, "Abstract cofibred domains: Application to the alias analysis of untyped programs," in *Proc. 3^{rd} Int. Symp. SAS '96* (R. Cousot and D. Schmidt, eds.), Aachen, 20–22 Sep. 1996, LNCS 1145, pp. 368–382, Springer-Verlag, 1996.

[33] P. Cousot, "Semantic foundations of program analysis," in *Program Flow Analysis: Theory and Applications* (S. Muchnick and N. Jones, eds.), ch. 10, pp. 303–342, Prentice-Hall, 1981.

[34] P. Cousot and R. Cousot, "Temporal abstract interpretation," in *27^{th} POPL*, Boston, pp. 12–25, ACM Press, Jan. 2000.

[35] P. Cousot and R. Cousot, "Abstract interpretation and application to logic programs" [1], *J. Logic Programming*, vol. 13, no. 2–3, pp. 103–179, 1992.

[36] S. Debray, "Formal bases for dataflow analysis of logic programs," in *Advances in Logic Programming Theory* (G. Levi, ed.), Int. Schools for Computer Scientists, section 3, pp. 115–182, Clarendon Press, 1994.

[37] J. Feret, "Confidentiality analysis for mobiles systems," in *Proc. 7^{th} Int. Symp. SAS '2000* (J. Palsberg, ed.), Santa Barbara, LNCS, Springer-Verlag, 29 June – 1 Jul. 2000. To appear.

[38] A. Venet, "Abstract interpretation of the π-calculus," in *Analysis and Verification of Multiple-Agent Languages, LOMAPS Workshop* (M. Dam, ed.), Stockhlom, 24–26 June 1996, LNCS 1192, pp. 51–75, Springer-Verlag, 1996.

[39] A. Venet, "Automatic determination of communication topologies in mobile systems," in *Proc. 5^{th} Int. Symp. SAS '98* (G. Levi, ed.), Pisa, 14–16 Sep. 1998, LNCS 1503, pp. 152–167, Springer-Verlag, 1998.

[40] A. Venet, "Automatic analysis of pointer aliasing for untyped programs," *Sci. Comput. Programming, Special Issue on SAS'96*, vol. 35, pp. 223–248, Sept. 1999.

[41] P. Cousot, "Types as abstract interpretations, invited paper," in *24^{th} POPL*, Paris, pp. 316–331, ACM Press, Jan. 1997.

[42] P. Cousot and R. Cousot, "Refining model checking by abstract interpretation," *Aut . Soft . Eng.*, vol. 6, pp. 69–95, 1999.

[43] D. Monniaux, "Abstracting cryptographic protocols with tree automata," in *Proc. 6^{th} Int. Symp. SAS '99* (A. Cortesi and G. Fil , eds.), Venice, 22–24 Sep. 1999, LNCS 1694, pp. 149–163, Springer-Verlag, 1999.

[44] D. Monniaux, "Abstract interpretation of probabilistic semantics," in *Proc. 7^{th} Int. Symp. SAS '2000* (J. Palsberg, ed.), Santa Barbara, LNCS 1824, pp. 322–339, Springer-Verlag, 29 June – 1 Jul. 2000.

[1] The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.

Published in the *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, CD Rom paper 224, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.