A Personal Historical Perspective on Abstract Interpretation

Patrick Cousot

Abstract Following an historical perspective, and restricted to my work with Radhia Cousot, I discuss the origin and evolution of concepts in abstract interpretation applied to semantics, verification, static and dynamic analysis, and algorithm design.

Abstract interpretation is a unifying theory of formal methods that proposes a general methodology for proving the correctness of computing systems, by sound (and sometimes complete) approximation on their semantics.

§ 1. Origin Radhia Cousot attended the Marktoberdorf summer school in July 25 to August 4, 1973 where Edsger Dijkstra showed program proofs by inventing backward invariants which shortly after became the weakest precondition calculus [147]. Instead of inventing invariants, Radhia had the idea of calculating these invariants by a backward analysis. To make the computation feasible she thought about starting with intervals (from her numerical analysis and operational research background).

Back to Grenoble, she showed me her ideas and was a bit disappointed that I disagreed with going backward (with an abductive reasoning) claiming that going forward (with a deductive reasoning) would be more precise. But, not discouraged, she made it forward and was convinced by an example showing better results with a forward analysis (which we later understood as incomparable in general!).

§ 2. Widening The next problem was loops for which the analysis was going on for ever. We first tried dichotomy, recurrence equations, etc. with moderate success and then had the idea to push unstable bounds to infinity, which we called widening since it made intervals larger. Definitely this process could not go on for ever with finitely many variables. Widening captures mathematical induction by forcing the convergence to an inductive program property.

§ 3. Example of widening (and narrowing) Consider the problem of inferring an invariant for the program `i=1; while (i<m) i = i+1;` where `m` is some (possibly

Patrick Cousot
CIMS, NYU, New York, NY, USA, e-mail: pcousot@cims.nyu.edu

large) integer constant[1]. A mathematical reasoning would be that before the first loop iteration, we have $\mathtt{i} \in [1,1]$. Before the second loop iteration, we have either $\mathtt{i} \in [1,1]$ on loop entry or $\mathtt{i} \in [2,2]$ after the incrementation so $\mathtt{i} \in [1,2]$ is invariant at that point of the computation. Making the induction hypothesis that $\mathtt{i} \in [1,n]$ is invariant before the n^{th} iteration, $n < \mathtt{m}$, the invariant would be $\mathtt{i} \in [1,1] \cup [2,n+1] = [1,n+1]$ before the $n+1^{\text{th}}$ iteration. Therefore, by recurrence on n, it follows that $\mathtt{i} \in [1,n]$ is invariant before the n^{th} iteration. This implies that $i = m$ on loop exit, assuming $1 \leqslant \mathtt{m} \leqslant \mathtt{max\_int}$ (and that the program terminates). Inferring automatically the induction hypothesis and then proving it correct is not a simple task for computers on large non-trivial programs[2].

A widening will help, by enforcing convergence at the cost of a loss of precision. We have $\mathtt{i} \in [1,1]$ on loop entry, $\mathtt{i} \in [1,2]$ after one iteration. At this point no widening occurs, since the extrapolation should be between consecutive loop iterations. One more iteration yields $\mathtt{i} \in [1,3]$. Since the upper bound is unstable the widening extrapolates to $\mathtt{i} \in [1,\mathtt{max\_int}]$. Therefore, we have $\mathtt{i} \in [\mathtt{m},\mathtt{max\_int}]$ on loop exit, which is sound but imprecise.

Anticipating on Section § 9., the next iteration with test `i<m` tells us that $\mathtt{i} \in [1,\mathtt{m}-1]$ before any iteration. It follows after one more iteration that $\mathtt{i} = \mathtt{m}$ on loop exit.

Observe that after an increasing iteration with larger and larger intervals $[1,1]$, $[1,2]$, $[1,3]$, widened to $[1,\mathtt{max\_int}]$, we have a decreasing one from $[1,\mathtt{max\_int}]$ to $[1,\mathtt{m}]$. This decreasing sequence might be infinite or slowly converging so that a narrowing will enforce finite convergence by interpolation.

The resulting invariant will always be sound but, by Rice theorem [183, 7], it may be imprecise. General alternatives to widening[3] consist in considering finitary cases or asking end-users to help in guessing the invariant and proving it correct, which hardly scale up to dozens of millions lines.

Radhia had not abandoned the idea of going both forward and backward as shown by her notes in fig. 1[4]. Their iterated forward-backwark reduction appeared later [31].

§ 4. From flowcharts to structural induction Of course the analysis was for flowcharts since we were taught at that time that a good programmer first draws a flowchart and then translate it into a program). Fortunately, under the influence of Edsger Dijkstra [146], this methodology was abandoned, although flowchart/control-flow graphs still persist in compilers. Most modern abstract interpretation-based

[1] A typical example is the loop of a synchronous control/command program to be executed every clock tick.

[2] Typically the body of the external loop of a synchronous control/command program has millions of lines.

[3] Other methods such as policy iteration and acceleration are domain-specific.

[4] In modern terms the forward analysis was computing a least fixpoint (also fixed point) by iteration from the infimum (denoted $\bigsqcap$ now $\bot$) while the backward analysis was computing a greatest fixpoint by iterating from the supremum $[-\infty,+\infty]$ so as to include non-terminating executions. Moreover being Cartesian, the forward and backward analysis where incomparable so no one is better than the other.

static analyzers work by structural induction that is induction on the program syntax (as found in Hoare's logic and denotational semantics).

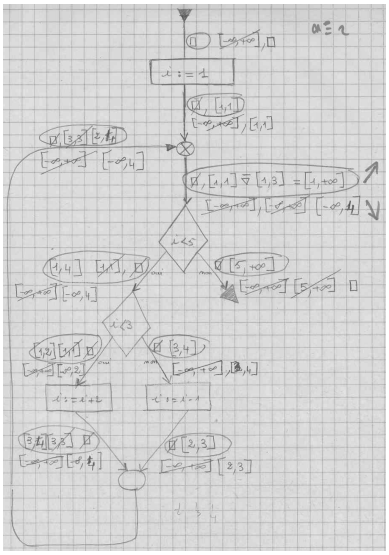


Fig. 1 Early notes of Radhia Cousot on static interval analysis

§ 5. Bibliographic input Started in complete isolation, the work went on in the context of a contract IRIA-SESORI (now INRIA) and we were evaluated at midterm by Hervé Gallaire and Bernard Lorho who were very encouraging. Bernard Lohro came with the 1970 paper of Fran Allen [5] and the thesis of Gary Kildall [164] (later published as [165]). Michel Sintzoff, visiting Grenoble, gave us his 1972 paper [190] and the 1973 paper of Ben Wegbreit [196] (later published as [197])[5]. Stephen Schuman introduced us to the work of Mike Karr [158] (later published as [159]) and James King [166] (later published as [167]). None of these references nor their citations were available in our computer science or mathematics libraries.

At first, we were somewhat discouraged thinking that these guys had invented everything before us! But none had widenings and so none could handle intervals. Moreover, we found errors in these papers (for example Kildall's constant propagation is not distributive, Sintzoff's rule of signs is wrong) while we had a general proof of correctness of our static analysis algorithm, including for intervals as a particular case. Our background in mathematics let us think that proofs do matter.

§ 6. Soundness and termination proof An important question was how to prove the correctness of static analysis algorithms rather than postulating that they work

[5] Bibliographic search and typesetting was entirely manual at the time, with delays from one to three weeks, if ever, when requesting preprints!

correctly by definition. The key idea is to refer to a formal definition of program execution, that is an operational semantics (a concept that we ignored at the time).

The proof appeared in the final report of the SESORI report [72]. Properties $\langle E, \emptyset, \cup, \nabla \rangle$ of concrete values are axiomatized using a set E of properties equipped with a binary join operation $\cup$ and infimum $\emptyset$ (which is shown to induce a partial order) plus a widening (satisfying $\forall x, y \in E \;.\; x \cup y \leq x \,\nabla\, y$ and a convergence hypothesis) for which basic mathematical properties are proved [6]. Abstract properties of each variable also form a join-semilattice of abstract values with infimum $\langle \overline{E}, \overline{\emptyset}, \overline{\cup}, \overline{\nabla} \rangle$ and widening $\overline{\nabla}$. As a typical example for integer variables, $\overline{E}$ is the set of intervals with $\overline{\emptyset}$ denoting the empty interval and the widening moving unstable bounds to infinity. Abstract properties (called abstract contexts) which assign abstract values to finitely many variables/identifiers are shown to have the same structure $\langle \overline{\overline{E}}, \Phi, \overline{\overline{\cup}}, \overline{\overline{\nabla}} \rangle$, pointwise. This Cartesian abstraction[7] was later extended to the notion of abstract domain to include forward and backward transformers on relational properties.

The correspondence between sets of concrete and abstract values is established through an abstraction (denoted @, now α) assumed to preserve finite joins and a concretization (denoted γ) where $\gamma \circ \alpha$ is extensive and $\alpha \circ \gamma$ is the identity[8]. It is shown, in great detail, that these abstract contexts are join semi-lattices with widening. Then program graphs are formalized together with paths, cycles, partitions, etc. and Claude Berge (hyper)graphs theory [9] is used to determine loop heads for widening. The interpretation of side-effect free assignment, test, junction, and loop head graph nodes is defined as a map f from concrete environments before execution to concrete environments after execution of the node (that would nowadays be understood as forward transformers for a small-step operational semantics).

Then the abstract interpretation $\overline{F}$ of these graph nodes is defined and proved to be increasing and locally sound in that $\dot{\alpha}(\{f(\rho) \mid \rho \in \dot{\gamma}(\mathcal{I})\}) \mathrel{\dot{\leq}} \overline{F}(\mathcal{I})$ for any abstract contexts $\mathcal{I}$ (where the dot notation means the pointwise extension from abstract values to abstract environments). So the concept of collecting semantics is left implicit.

Then a forward abstract interpretation algorithm is given to traverse the graph from an initial abstract context on some structure $\langle \overline{E}, \overline{\emptyset}, \overline{\cup}, \overline{\nabla} \rangle$, the abstract context of all other nodes being the infimum Φ, until stabilization. Each iteration first propagates the information at entry, junction, or loop head nodes to the next junction or loop head nodes. Then the join at junction nodes and widening at loop heads is computed. This is repeated until stabilization at loop heads.

---

[6] Having not yet discovered Garrett Birkhoff's lattice theory [15] at the time, they were not named "join-semilattice with infimum and widening"!

[7] Cartesian abstractions disgard any relation between variables, program points, etc.

[8] Thanks to a discussion with Claude Benzaken pointing to Garrett Birkhoff's lattice theory book [15], we discovered complete lattices [15, Ch. V], Galois connections [15, Ch. V, §8], and Tarski's fixpoint theorem [15, Ch. V, Th. 11], [192]. It took us some time to understand that our pair $\langle \alpha, \gamma \rangle$ was (almost) a Galois connection, since they are defined semi-dually in [15, Ch. V, §8]. They also preserve arbitrary joins (not only finite ones) of posets and are not necessarily surjective, that is, $\alpha \circ \gamma$ is reductive.

For the termination proof, the successive abstract contexts at each node are shown to be increasing, hence would be strictly increasing at loop heads in case of nontermination. This would be in contradiction with the definition of the widening $\overline{\overline{\nabla}}$ on abstract contexts (previously derived pointwise for the widening $\overline{\nabla}$ an abstract values).

For soundness, it is shown that the algorithm stops with stable contexts, meaning that one more iterations imply the same result[9]. The main theorem shows, by recurrence , that at any point in any execution (understood as a sequence of concrete contexts assigning values to variables) of the graph reaching a node will have values of variables in the concretisation of the abstract context of that node.

Since the forward abstract interpretation algorithm involves only abstract contexts, its termination proof involves only abstract contexts, and the soundness proof is relative to concrete executions , the algorithm and its soundness proof apply equally well to relational analyzes such as [158] which is (based on a poset satisfying the ascending chain condition, so $\overline{\overline{\nabla}} = \overline{\overline{\cup}}$).

Several examples are given (intervals and parity for integers, nullity of pointers).

§ 7. Initial publication The main innovative contributions were the introduction of the widening, of soundness with respect to a concrete semantics using abstraction and concretization functions, the original interval example, together with its correctness proofs. The paper was rather long (126 p.), with a lot of very similar cases due to the use of flowcharts with 6 kinds of nodes, and written in French, so had no impact at all (6 citations on Google Scholar[10], including 2 by ourself). The paper was translated in English (18 p., without proofs). In addition, it included the idea of dual abstract interpretation[11] [71] (13 citations on Google Scholar, including 2 by ourself). Submitted for publication and accepted at the ISOP conference (now ESOP) [73], we had in between discovered and cited the pioneer work of Peter Naur on pseudo evaluation [178, 177]. Fortunately Peter Naur was our session chair and thanked us for building so beautifully upon his work[12]. He also stopped Wilhelm De Roever, trying to find counter-examples to termination, "there is a proof, which means you cannot find a counter-example". On Google Scholar, the paper is cited by 751.

§ 8. Development of the theory from an algorithmic to an algebraic framework The year 1975-79 were very productive[13] and saw the development of basic ideas in the theory of abstract interpretation. Although the work was more or less simultaneous depending on the inspiration of the day, we use a mix of chronologic and thematic presentation of these ideas.

---

[9] In modern terms, the iterations stop at a postfixpoint, not necessarily a fixpoint.

[10] Citation counts were collected on Nov. 30, 2022.

[11] In modern terms using greatest instead of least fixpoint.

[12] We did not dare tell him that the citation was a posteriori!

[13] Up to abandoning hiking, climbing, and cross-country skiing in Grenoble, at least during working days.

§ 9. Narrowing We observed that the abstract interpreter could go on iterating after finding a solution and this improved the solution (see Example § 3.). This improvement might be slow to converge (e.g. for intervals), so we had to enforce rapid convergence. Since dual widening was not working[14], we invented the narrowing, as well as dual widening and narrowing, which first appeared in [74], see [59] for examples.

§ 10. Fixpoints We also rapidly understood that the abstract interpreter was solving a (system of) fixpoint equation(s), and that, by Tarski's fixpoint theorem, there was a least solution (under function increasingness and complete lattice hypotheses), and moreover, least fixpoints can be computed iteratively for additive/join preserving functions. We also observed on simple mathematical examples that, for increasing functions not satisfying join preservation, iterating beyond infinity also yields fixpoint solutions. Having learned set theory in the book [175] of James Donald Monk (a student of Alfred Tarski), in particular the John von Neumann encoding of ordinals [179], we where ready to consider transfinite iterations that lead to the constructive version [82] of Tarski's fixpoint theorem[15].

§ 11. Verification Bibliographic search on fixpoints and semantics lead us to the work of Zohar Manna with Stephen Ness and Jean Vuillemin [170]; with Shmuel Katz [163] [16], pointing to Dana Scott [187, 188]. Together with Edsger Dijkstra [148], this inspired us to look for a link between fixpoints, program semantics, and verification[17]. This work is reported in [75] (175 citations on Google Scholar).

Program properties are first-order predicates attached to program points relating initial and current values of variables (nowadays we use set theory to cope with inexpressivity problems of logics). By pattern matching of the program syntax, a system of equations is built using forward strongest postcondition transformers (to get relational reachability)[18] [75, Ch. 3] [19]. By Tarski's fixpoint theorem, these equations have a least fixpoint which we called "optimal invariants" (after [171], which is unfortunate since "strongest" would have been much better). This strongest invariant provides a proof of total correctness (including termination) [75, Ch. 4]. By considering inequations instead of equations, we get, by Tarski's fixpoint theorem, an

[14] Starting from a postfixpoint over approximating the least fixpoint, a decreasing sequence will converge above the least fixpoint under Tarski's fixpoint theorem hypotheses. A dual widening would extrapolate with under approximations which may lead to an under approximation of the least fixpoint, i.e. an incorrect invariant. By contrast, a narrowing will interpolate by over approximation and so will converge above the least fixpoint.

[15] The review was so knowledgeable, helpful, and generous that we have always thought it might have been by Alfred Tarski himself.

[16] At the time, the C.A.C.M., available in our library, was our main source of information.

[17] Although [148, Ch. 5] has no explicit fixpoints, we recognized them as being implicitly defined iteratively. This was later corrected in [151] and [150, Th. 8].

[18] We later understood that structural induction as found in Scott-Strachey denotational semantics [189] and Gordon Plotkin's structural operational semantics [182] is much more readable, although mostly equivalent, "mostly" since patterns can be ambiguous while induction on the program syntax is not.

[19] Nowadays rule-based deductive systems are used but the structural induction idea is the same.

overapproximation suitable for partial correctness [75, Ch. 5]. We show in [75, Ch. 6] that this yields symbolic execution [166]. Using difference equations, it is proposed to build the symbolic execution tree by chaotic iteration (see the forthcoming Sect. § 14.), inferring an inductive term for the tree, and passing to the limit by hand [75, Ch. 7]. However as, we said at the end of this Ch. 7, automating the inference of the inductive argument is difficult (and still not solved in symbolic execution which considers a few prefixes of a few executions paths, a trivial abstract interpretation by under approximation). In [75, Ch. 8] we showed that invariants can be discovered by strengthening chaotic iterations, a widening in disguise with no convergence hypothesis. The conclusion is a bit optimistic but recognizes that the "semi-automation resolution of the semantics equation is a terrific task"[20]. We were enthusiastic about understanding that semantics and verification can be explained in the abstract using a few concepts of fixpoints, successive approximations, chaotic iterations, etc. We wrote several papers to explain that in more details [79, 30, 76], all rejected! Nevertheless, the paper [75] has 175 citations according to Google Scholar, not so bad (compared to the average 1.75 citations per paper in computer science). See also [61] for a recent discussion of verification by abstract interpretation. A very singular point in this verification work is [53] where we used semidefinite programming methods to solve equations. We abandoned this promising venue for non-linear properties because of the imprecision of the solvers providing no formal correctness guarantee.

§ 12. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints Although it came after more than three years of work, the 1977 POPL paper [74] is often cited as the foundational paper in abstract interpretation. It is the most cited paper of the POPL conference (with 5615 citations but 8725 citations on Google Scholar, 8366 on Microsoft Academic, 6557 on Semantic Scholar, 6035 on Research Gate, 4040 on the ACM Digital library which shows that the automated collection of references is incomplete and the number of citations is definitely proportional to the size of communities. Recent papers in machine learning typically have 10-50 thousands citations. In contrast Robert Floyd's fundamental paper on "Assigning meanings to programs " [152] has 4067 citations on Google Scholar. Citation counts are certainly both unreliable and meaningless when taken out of context. Moreover, I have always suspected that most authors cite the paper by mimicry, without even reading it.).

Ch. 3 defines the syntax and semantics of flowcharts (built by structural induction, that's the limit!).

Ch. 4 defines their equational collecting reachability semantics (called static semantics) as a least fixpoint and illustrates different alternatives shown in Fig. 2

[20] As an example of unfortunate linguistic calque in our papers, we use "terrific" with its French meaning of "terrifiant" that is causing terror which is archaic in English.

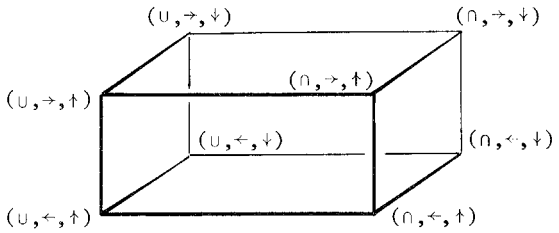


Fig. 2

that explains lattice duality (∪/∩), forward reachability (⟶) versus backward accessibility (⟵), and greatest (↑) versus least (↓) fixpoint.

Ch. 4 introduces the abstract interpretation framework using Galois retractions (surjection/insertion/embedding). At the time we had not reached Ch. V, §8 on Galois connections of Garrett Birkhoff's book [15] so the connection was not yet made! To justify our need for a formal semantics, we explain that data flow analysis problems (such as available expressions), although purely syntactical, do fit in this framework but, in general, a semantic-based approach is needed, citing Gary Kildall's constant propagation [165]. Several years later, I showed that the syntactic and semantic points of view lead to incomparable definitions of live variables [65], not a good news for compiler correctness! A first example is to abstract sets of states into predicates (more precisely isomorphic infinite disjunctive/conjonctive normal form of atomic predicates which are assignments of values to variables), and show that Floyd's methods can be justified as a postfixpoint[21]. Of course using first-order predicates instead would be without best abstraction and incomplete.

Ch. 6 is about "consistent abstract interpretations" (that is soundness through commutation of transformers with abstraction and concretization) summarized by Fig. 3

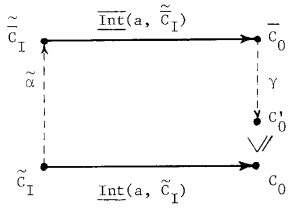


Fig. 3

where an abstract transformer $\overline{\underline{\mathtt{Int}}}(\mathtt{a},\widetilde{\overline{\mathtt{C}}}_{\mathtt{I}})$ for a statement a overapproximates the concrete transformer $\underline{\mathtt{Int}}(\mathtt{a},\widetilde{\mathtt{C}}_{\mathtt{I}})$ up to the correspondance $\langle\widetilde{\alpha},\ \widetilde{\gamma}\rangle$ between concrete and abstract properties.

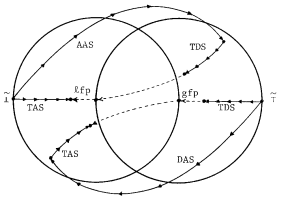
The next Ch. 7 is on the lattice of abstract interpretations, based on the idea that given a semantics $\boldsymbol{\lambda} P \bullet S[\![P]\!]$ of programs P, the abstractions of the strongest prop-

[21] At the POPL 77 conference, Ed. Clarke asked us to cite his thesis where he said to have invented fixpoint characterizations of program correctness (which, as noticed e.g. in [84] can be traces to David Park [181]) . We asked him to sent us his thesis, but we never received it. In good faith, I cited him in my thesis. Years later, on April 12, 2012, I gave the Gaschnig/Oakley Memorial Lecture (SCS Distinguished Lecture Series) at Carnegie Mellon University. He publicly claimed to have invented abstract interpretation in his thesis, now online `http://www.cs.cmu.edu/~emc/hoare/thesis.pdf` hence checkable. A funny widening!

erty, i.e. it's collecting semantics $\boldsymbol{\lambda} P \bullet \{S[\![P]\!]\}$ of each such program P by a Galois connection $\langle \alpha, \gamma \rangle$ can be organized, up to equivalence, in a complete lattice (isomorphic to the lattice of all closure operators $\gamma \circ \alpha$). Several examples are given (collecting semantics, intervals, constancy, signs, constants with signs, etc.). Designing a semantics, proof method, or static analysis essentially consists in characterizing a point in this lattice[22]. This idea yielded to the hierarchy of semantics of section § 27.

Ch. 8 discusses the correctness and termination of the resolution of abstract equations by iteration referring to various possible hypotheses to ensure finiteness. It provides an example of infinite iterates (with an example on probabilistic analysis of the performance of programs. Probabilistic static analysis came much later [141] and still lacks effective widenings [23]).

Ch. 9 is on iterative least fixpoint approximation methods, the instantiation of this idea with widenings to get an overapproximation from below to reach a postfixpoint (and no longer a fixpoint as in [72], which is less precise) and then a truncated decreasing sequence or with narrowing to reach a fixpoint, as well as there duals for greatest fixpoints.



AAS : ascending approximation sequence
DAS : descending approximation sequence
TAS : truncated ascending sequence
TDS : truncated descending sequence

Fig. 4

The conclusion in Ch. 10 states that abstract interpretation is certainly the weakest mathematical model to data flow analysis, in optimizing compilers type verification, type discovery (inference), program testing, symbolic evaluation, program performance analysis, formalization of semantics, verification of program correctness, discovery of inductive invariants, proofs of program termination, program transformation. This was an ambitious research program which took us decades to fulfill in part.

Whereas the call for papers required a five page summary of the work, our submission dated August 12, 1976 was 101 manuscript pages, sent in maybe a dozen copies, one for each program committee member. Since the submission was too long to fit on 15 pages, we had to eliminate a few subjects in the final version, such as type checking and inference, backward analysis, live variables, trace semantics, symbolic execution, pointers and sharing, etc. The program chairman, Ravi Sethi, decided to anonymize the submissions and erased all citations, so told us that he had to white-out all references by hand in all of our copies. During the conference, Zohar Manna

[22] The definition of equivalence of two abstract domains by existence of two Galois retractions between them is not antisymmetric [21]. Therefore, we later changed the definition in [83, Th. 8.0.1] of the equivalence of Galois connections $\langle \alpha_i, \gamma_i \rangle$ to be the isomorphism between their closures $\alpha_i \circ \gamma_i$. This is isomorphic to the closure operators on a complete lattice (i.e. the collecting abstract domain), which, by Ward theorem [195, Th. 5.3], is a complete lattice.

[23] Incidentally, the paper was rejected at POPL with C's and D's. After shortening by elimination of background matter and reformatting, it was accepted at ESOP with three A's. TEX and competent reviewers do matter!

took us apart to tell us "never do that again". Decades later, Zohar Manna told us that he had hard discussions during the program committee meeting to have the paper accepted. His motivation was the diagram shown in Fig. 4 where iterations reach fixpoints which are not extremal (neither least nor greatest). He thought we had solved the optimal fixpoint problem of [171]! The picture made it!

§ 13. Recursion and modularity We also worked on recursive procedures [78] (to answer critiques that flowcharts do not account for recursivity). The input-output semantics is relational and defined by a system of recursive equations which solution is an exact procedure summary. Two widenings are used both to avoid infinitely many calls with different parameters and infinitely many results. The approximation is handled from a topological point of view using closure operators inspired by [187]. Chaotic iterations are extended to higher order, with a bug[24] later corrected by Mads Rosendahl (with additional hypotheses [185]). Examples include intervals and heap analysis (to answer critiques that abstract interpretation can only cope with numerical domains). 243 citations on Google Scholar. A computer science paper with topology is probably too much [25].

We came back to the subject two decades later in the form of modular static analysis [109] later published in [111], with no more success (228 citations on Google Scholar)!

§ 14. Chaotic and asynchronous iterations Having observed that different iteration strategies may lead to different results (due to the widening not being increasing in its first parameter), we looked for mathematical results on iteration, mainly in numerical analysis. We found the notion of chaotic iteration in the work [184] of François Robert (a professor of numerical analysis in Grenoble) and their generalization by asynchronous iterations, including with memory, in the work of Gérard Baudet [8]. We discovered a bit later that these chaotic iteration ideas originated from Dan Chazan and Willard Miranker [23]. After an erroneous attempt in [75] for chaotic iterations, we got it right for asynchronous iterations with memory in [29]: solving increasing equations on a complete lattice by asynchonous iterations with memory always yields the least fixpoint (thus excluding widening which is not increasing!) [29]. Having seen the report [29], François Robert pointed to a similar result by Jean-Claude Miellou [173], three months earlier. But we had no continuity hypotheses (we used transfinite iterations instead) and, more importantly, we had no hypothesis that the iterates are increasing (which requires a costly synchronization between parallel processes). François Robert encouraged us to publish the results but the submission was rejected under the claim that this result was part of the computer science folklore [168] and moreover transfinite iterations are useless in Computer Science as proved by Dana Scott's continuous semantics. Nevertheless the result holds for any partial order (not only the specific Dana Scott's order which happens

[24] The sequence of abstract properties of procedure actual parameters along recursive calls is not necessarily increasing, although this was enforced when using widenings.

[25] Anecdotically, Ugo Montanari told us that no one understand abstract interpretation theory because it is not phrased in category theory. We never went to this extreme, but sheaves looks to be perfect candidates. See also the categorical construction of [194].

to be continuous [186]) and unbounded nondeterminism yields increasing but non-continuous transformers [151] (despite rejection the paper has 62 citations on Google Scholar). For a few decades now the preferred iteration strategy with widening is that of François Bourdoncle [20] which is empirically adequate, but the best compromise between efficiency and precision is still to be found and proved optimal.

§ 15. Types In 1977, we published a paper [77] explaining that types could be value dependent if formalized by abstract interpretation (which is nowadays well-recognized through liquid types, dependent types, gradual types, etc). Twenty years later, invited by Neil Jones at POPL 1997, and, at the time, POPL being essentially about types, I showed that type inference in higher order languages are abstract interpretations with widenings [41]. I think this answered the criticism that program analysis is unpredictable because of widening whereas that is not the case for type systems with well-defined inference rules. The point is just that the widening is hidden in the rules and I showed that changing the hidden widening changes the precision of the type system. Another point was to show that abstract interpretation applies to functional languages with denotational semantics. The paper has had very little success (259 citations on Google Scholar) since the proof of soundness of type systems are most often based on preservation and progress with respect to an operational semantics. I recently showed that this is also an abstract interpretation [69] that could be useful in applications outside typing.

§ 16. Galois connections, closures, Moore families, etc The bibliographic search and study of Galois connections, led us to closure operators, Moore families, etc. The understanding that it was possible to project the abstract domain in the concrete by the upper closure $\gamma \circ \alpha$ which made, at least theoretically, the study abstraction independent of a specific implementation. Nevertheless, abstraction and concretizations pairs $\langle \alpha, \gamma \rangle$ are still useful to describe property encodings and implementations. We also studied weaker forms of closures [81] since best approximations are often a too strong hypothesis.

§ 17. Transition systems For conciseness of concepts and proofs, Claude Pair advised us in 1978 to use transition systems (i.e. a relation on states, which we called "discrete dynamical system" by reference to dynamical systems in mathematics and physics), which I did in [31], then submitting [32], which was rejected, but the concepts were reused in [33]. This considerably densified the presentation of basic concepts (without the need to go through all the details of a programming language syntax and semantics) and drastically shortened the proofs, but also narrowed the scope of application, e.g. to higher-order functional language without going to operational details.

§ 18. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes At that point, I had enough content for my thesis [31].

Ch. 2 includes complete lattices, closures, fixpoint theorems, continuity, iterations, elimination methods, and the new asynchronous iterations with memory.

Ch. 3, introduces transition systems and their inverse (to cope with backward analysis). The basic idea is that all results on program verification and analysis can be concisely formulated using transition systems: (small-step) operational semantics, exact semantic analysis (using pre and post conditions, static partitioning to cope with local invariants, fixpoint characterization of forward reachable and backward accessible states), and applications to invariance verification (by Peter Naur, Robert Floyd, Tony Hoare, and Edsger Dijkstra methods), termination, incorrect termination, nontermination, combination (intersection) of forward reachable and backward accessible states.

Ch. 4, on constructive methods of fixpoint approximation for increasing functions on complete lattices. The first method is to simplify the equations based on closure operators and their equivalents. A series of theorems shows that complete lattices are preserved by this abstraction and that all possible closures form a complete lattice (i.e. the so-called lattice of abstract interpretations). Numerous theorems are recalled if classic or prove to characterize the properties of closures.

The second method is convergence acceleration by widening, narrowing, and their dual convergence operators elaborating on [74]. A distinction is made between convergence operators to over approximate solutions and those with the additional property of ensuring finite convergence, which are two orthogonal problems. This is often misunderstood. For example Graig interpolation is a dual narrowing not enforcing convergence [59].

Ch. 5, applies the previous results to static analysis, giving the examples of parity, signs, their combination by a reduced product, live variables and available expressions handled syntactically, constant propagation, pointer nullity, equivalence classes of pointers that may reach the same memory cell, simple typing, all with lattices satisfying the ascending chain, as well as intervals and polyhedra for infinitary analysis with convergence acceleration, and finally iterated combination of forward reachability/backward accessibility of analyses. Finally these abstract interpretations are ordered in a lattice which supremum is an analysis of the connexity of the program flow graph.

Ch. 6 on the analysis of recursive procedures is an improvement over [78].

Ch. 7 concludes that much remains to be done (including e.g. equations with complements which are not increasing, still a difficulty!).

§ 19. Automatic discovery of linear restraints among variables of a program

The polyhedral analysis to infer invariants of the form $AX <= B$ where A is a matrix, B is a constant column, and X is column of values of variables [140], was designed and implemented by Nicolas Halbwachs (using the double description method and reinventing many polyhedron manipulation algorithms). This was certainly the first implemented static analyzer based on the theory of abstract interpretation. The widening eliminated unstable constraints. Its precision was improved by Nicolas Halbwachs in his thesis [155] by keeping some redundant constraints. The narrowing was a few more decreasing iterations. The analyzes were surprisingly precise, in particular because of the inference of affine relations between variables that never ap-

pear in the same statement, see for example the Heapsort algorithm, [140, p. 95][26]. The research on the polyhedral analysis efficient implementation, refinement, and extensions [25, 26, 27, 28] has been very active since four decades (2114 citations on Google Scholar).

§ 20. Systematic Design of Program Analysis Frameworks Many of the new results obtained in my thesis were published in [83]. The program semantics of a program graph is defined by an abstract transition system, that is a complete lattice of abstract properties and abstract transformers for assignments and tests nodes (so that transitions are on abstract properties and backward analysis is nothing but forward analysis for the inversed transformers of the inversed graph). The abstract property attached to each program points can be calculated in two ways. One is the least fixpoint of a system of equation defined using the abstract transformers. The other is the "merge over all paths" (MOP). It consists in propagating the abstract property along each program prefix path using the transformers and then merging (by a join) the abstract properties collected at each program point along any program path. This is to refute an argument of the time that fixpoint are imprecise.

A "very reasonable hypothesis" introduces the notion of best/most precise overapproximation of a concrete by an abstract property. Formalizations by upper closures, Moore families, complete join congruences, family of principle ideals, and Galois connections are shown to be equivalent. The emphasis is on best transformers (left implicit in Fig. 3 of [74]) and the composing abstractions.

Given a concrete domain, increasing concrete transformers, and an abstract domain, it is shown how to design sound (and complete) abstract transformers for both MOP and fixpoint abstract semantics. Examples are given such as the prefix trace semantics, the reachable states, and available expressions for which MOP and fixpoint solutions coincide. This shows that constructing abstract semantics proves correctness, which is better that postulating the data flow equations, as common at the time (and still today). This also shows the necessity of reasoning in traces rather than invariants, since available expressions are definitely not an abstraction of reachable states.

If the abstract transformers do not preserve arbitrary joints then the MOP and fixpoints definitions of program properties may be different, the MOP solution being always more precise. So fixpoint-based program analysis was generally considered inherently imprecise. We showed that the problem is not with fixpoints but with the abstract domain. First we observed that when the abstract property are sets of reachable states, the two solutions are the same. More generally, this is the case when the abstract transformers preserve arbitrary joins (previous results assumed the abstract domain to be ACC/Noetherian[27]). Finally, it is shown that any non-distributive abstract domain can be refined by a powerset construction into a distributive one, which is more precise and for which MOP and fixpoint definitions of abstract properties exactly coincide. However, the powerset abstract domains might be infinite

[26] Incidently the Heapsort program starts with a division, which was not implemented, and was erroneously handled by hand. Fortunately, this analysis after this initialization error is correct!

[27] ACC stands for ascending chain condition, i.e. any strictly increasing chain is finite.

which requires using widenings which can always be chosen to be more precise than the original analysis. (For example the ACC constant propagation lattice becomes the powerset of values with infinite chains. A possible widening will extrapolate sets other than empty and singleton sets to all values. Doing that only for sets of cardinality larger that $n \geqslant 1$ provides an infinite chain of more and more precise widening where constant propagation is $n = 1$.) Thus shows that the concept of MOP is superfluous since in can be exactly expressed using fixpoints, but not conversely.

The last chapter of [83] is on the combination of abstract interpretations. The general idea is to construct complex abstract domains from simpler ones. The first is the reduced (cardinal) product which concretization is a logical conjunction. It can be understood as a Cartesian product with a reduction propagating information from components to components (for example odd in parity and $\geqslant 0$ in signs reduces the sign to > 0). The reduced product is also the greatest lower bound in the lattice of abstract interpretation. In overapproximate form, this is the basis for organizing abstract interpreters. In a later detailed study [133, 135, 136], we showed that the idea is also used in SMT solvers using Open-Nelson algorithm (where the reduction is only for equalities and inequalities). The reduced cardinal power allows for case analysis (which applied recursively yields e.g. BDDs or choice-based arborescent domains [24, 134]). A full version with proofs was submitted and rejected since the POPL submission was not marked "Extended Abstract", discouraging, isn't it?

§ 21. Parallelism We applied the abstract interpretation theory to the verification and static analysis of shared memory and synchronously communicating parallel processes (with an interleaving semantics).

The main idea is that proof methods by Edward Ashcroft (1975), Robert M. Keller (1976), Suzan Owikci and David Gries (1976), Leslie Lamport (1977), etc. are all applications of fixpoint induction, with different abstractions formalized by Galois connections [84]. The paper was submitted, rejected (despite a signed and laudative review by Leslie Lamport), and recycled, in part, in [88].

For static analysis, the interleaving semantics leads to a costly analysis of interference. Hoping for reduced interference costs, we considered Tony Hoare's communicating sequential processes (CSP) [85]. The static analysis repeats the local analysis of each individual processes between initialization and communication points followed by an analysis of the synchronous communications. The cost remains high because a global relational invariant is needed to capture the information flow through all processes.

Our research on parallelism stopped by lack of funding, the short-sighted view being that, by Moore's law doubling transistor counts every two years and Dennard scaling enabling increases in clock frequency, parallel machines would be surpassed two years later by sequential ones! May be also by lack of interest, [85] has 60 and [88] has 48 citations of Google Scholar.

§ 22. Proof methods and induction principles Having discussed at lot with Rod Burstall while he was visiting Grenoble[28], we understood that static analysis algo-

[28] Being the PhD adviser of Alan Mycroft, he ensured that abstract interpretation crossed the channel!

rithms are nothing but abstractions of proof methods. We concentrated for nearly a decade on studying program verification, trying to be independent of specific languages thanks to transition systems (i.e. assuming a small-step operational semantics) and formulating proof methods as induction principles (and their duals as in Fig. 2 plus negation ¬) [86] so as, e.g. to allow for a backward contrapositive method). This was extended to parallel programs [90] (in French, cited by 1 on Google Scholar, this one being ourselves :).

We were particularly interested in Rod Burstall's method involving symbolic execution and induction [22] which worked very well on recursive procedures and for total correctness. Nevertheless, we had difficulties to compare it to invariance and termination proofs by variant functions (à la Naur/Floyd/Hoare). Since the two formalisms looks different, we introduced an extension [34] of Hoare's logic for a simple non-deterministic imperative iteration programs with random assignment to cope with Rod Burstall's method hand simulation by unrolling loops plus a structural induction rule. The proof outline can be presented using local assertions as in Hoare's logic, up to the unrolling of loops to cope with induction.

The next question was whether the two methods are equivalent, that is, given a proof by one method is it possible to rewrite that proof to be a proof by the other method? Comparing logics is impractical since it is language dependent and many cases have to be considered for all language constructs. So we used transition systems, both for [Turing/]Naur/Floyd/Hoare's method [89] (for partial and total correctness including for parallelism) and Burstall's method [87]. Given an ordinal (built out of the program structure and well-founded relations), the inductive invariant relates program initial states to intermediate and final states for each such ordinal. Its proof considers a transition ("hand simulation") and the use of induction hypotheses at lower ranks ("little induction"). The same idea is behind "segments" in [122].

The equivalence proof [91] (which was published eight years later, the manuscript having been lost in the editorial process) essentially consists in recursion elimination in Burstall's invariant for each ordinal using a transfinite stack to get [Turing]/Naur/Floyd/Hoare's invariant relating initial and intermediate states.

This proof explains why Burstall's method works so well for recursion, why it is more powerful than [Turing/]Naur/Floyd/Hoare's method. It was implicitly used in [78] and in Astrée in particular in the form of unrolling ("hand simulation") where the "little induction" is by widening. So automatic proof of the soundness of static analyzers based on [Turing/]Naur/Floyd/Hoare's logic (such as [157, 6]) cannot directly account for Astrée which, when unrolling loops and recursions, uses intermittent assertions [22, 172, 91] which are not invariants.

To conclude this section let us mention an attempt to present computer logics independently of a specific programming language [92]; the abstraction of induction principles for transition systems to fixpoint induction principles [46]; more recently induction principles for proving safety and termination properties [122]; and induction principles generalizing Dana Scott's induction for proving the total correctness of programs with denotational semantics [64]. A sound and complete proof method for *general* liveness is still an open problem.

A criticism of our work on the design of sequential or parallel program proof methods by abstract interpretation was that our only examples were existing methods, so that there was nothing new but theory, at best variants of existing methods. We had an opportunity to show that theory can help practice when formally designing an invariance proof method for shared variables parallelism with weak memory model [3] . The construction is by abstraction of a formal semantics of the maximally parallel programming language [2] restricted by a specification of its specific memory model [4]. The proof method is sound and complete by calculational design.

§ 23. Back to static analysis At the beginning of the 90's, our work moved back to semantics and static analysis, after the renewed interest in England and Scotland around Alan Mycroft's pioneer work on strictness analysis of lazy functional languages. At the time, we were also aware of similar interest in Denmark by Neil Jones and Flemming Nielson, but not in Germany by Reinhard Wilhelm, in Italy by Giorgio Levi, in Spain by Manuel Hermenegildo, and neither, somewhat later in the US by Thomas Reps and David Schmidt.

§ 24. A variety of abstract interpretation models At that point, we had used Galois connections and, in absence of best abstraction, the concretization function only, as in [140] for polyhedra. As more examples without best abstraction emerged (like abstractions to languages in [102]), we proposed several possible models of abstraction beyond a concretization only [96].

As first observed by Alan Mycroft in his thesis [176], the static analysis of functional languages with denotational semantics involves two partial orders: Scott's computational order $\sqsubseteq$ and logical implication $\subseteq$. The basic fixpoint abstraction theorems can be extended to cope with that case [99]. Another rare example of the need for a computational and a different approximation algorithm appears in the formalization of dynamic analysis by abstract interpretation [142]. It more often happens that in the abstract these two orders both collapse to an abstract logical implication $\subseteq$, which may look confusing with two orders in the concrete and one in the abstract. Another difficulty is that general properties of functions in $A \longrightarrow B$ are in $\wp(A \longrightarrow B)$ and not in its abstraction $\wp(A) \longrightarrow \wp(B)$ (as in typing and not expressive enough for comportment analysis). We explained and solve these difficulties in several papers showing various examples [94, 99, 100].

Although stritness analysis is Boolean (must not terminate/may terminate), it is subject to combinatorial explosions (with parameters and recursion). The finite domain is huge, and we proposed the use of widening. Widening was understood as heuristic compared to Galois connection, and not, as we do, as a mechanized induction. Moreover, claims that some analyses are in infinite domains while requiring no widening (as in set-based analysis, see [102]) are due to the misunderstanding of what is an abstract domain. In [93, 97], we showed that (1) abstract domains hence widenings are relative to a program, in general not to a programming language; (2) all that can be done with a Galois connection can be done with an appropriate widening; (3) infinite abstract domains with widening are more powerful that finite domains; and (4) using the most specific framework is beneficial (see e.g. [103]).

§ 25. Language independence It is difficult to explain methods for reasoning about programs without referring to a specific programming language or even to a specific semantics [38]. Starting with transition systems in my thesis [31], through [92], and culminating in [124][29] with a Galois connection calculus covering the abstraction (of properties of the semantics of) af all constructs found in programming languages. Of coarse this list of constructs is incomplete (and maybe unbounded)!

§ 26. Bi-inductive definition of maximal traces Until [89], we had considered prefix closed finite traces. But since [89], our concrete semantics has been (properties of) finite or infinite maximal traces. We looked for an inductive definition of such maximal trace semantics. We first generalized Peter Aczel's inductive definitions [1] to an arbitrary order [101]. Then we investigated an order combining induction (for finite traces) and coinduction (for infinite traces) [98]. A question/opinion at the 1992 POPL conference was that bi-induction applies only to imperative programs, not functional ones. Fifteen years later, we prove that this is not the case and applies equally well to the λ-calculus [118, 119].

§ 27. Hierarchies of semantics Understanding that program semantics are abstractions of one another, we have constructed a hierarchy of semantics including all classical semantics and program verification methods [40, 39, 50]. The idea is to understand any semantics as an abstraction of the maximal trace semantics defined by bi-induction. This also solves the problem of which semantics to choose for proving the soundness of static analyses, by refining in the hierarchy, if necessary.

§ 28. Calculational Design The idea of calculational design is that given a formal definition of a concrete semantics/mathematical structure and an abstraction, it should be possible to design the exact (e.g. section § 27.) or approximate (e.g. section § 20.) abstract semantics/mathematical structure by calculus using induction and the composition of elementary abstractions for sets of values, functions, relations, products, fixpoints, etc. If the abstract semantics is given this becomes a refinement. If both the concrete and abstract semantics are given this is soundness verification, with a proof that can be mechanically checked.

§ 29. Calculational design of abstract semantics The idea, long left implicit, clearly appeared in the 1998 Marktoberdorf summer school [42] where an analyzer for a simple imperative language is designed formally, by calculus. A recent example is [68]. Although the a posteriori verification of the design from [157] to [153] tends to more automation, an interactive mechanized tool able to automate part of the calculations involved in the design is still to be invented.

§ 30. Calculational design of algorithms The exact or approximate abstraction ideas formalized by abstract interpretation can be used to design algorithm by abstraction of (usually infinite) mathematical structures. This is the case for example of parsers derived by abstraction of languages [121, 117, 114], of symbolic terms

[29] After being awarded the ACM SIGPLAN Programming Languages Achievement Award in 2013, we where generously offered 2 pages in the POPL 2014 proceedings. This resulted in the densest paper we ever wrote. Add a single TeX point after the title and it goes to 3 almost full pages!

(and unification) by abstraction of sets of ground terms (and union) [66], of weighted graphs path algorithms such as the Roy-Floyd-Warshall shortest path algorithm [67], of software model checking [68], etc.

§ 31. Static analysis is harder than verification Program verification and static analysis are performed by induction (on data, on program steps, etc.). The difference is that in verification the inductive argument is given while in static analysis it must be discovered, then verified and maybe strengthened. This makes analysis harder than verification, in a sense made precise by [138].

§ 32. The origin of ASTRÉE The first industrial application of abstract interpretation started with Alain Deutsch who developed an abstract interpretation-based static analyzer at INRIA after the failure of the Ariane 501 launcher on June 4, 1996 due to an arithmetic overflow. He commercialized the Polyspace analyzer with the help of Daniel Pilaud in January 1999. Polyspace was sold to MathWorks in 2012.

Meanwhile, Wladimir Mercouroff, head of the Foundation of the École normale supérieure in Paris, regularly asked me to propose an industrial seminar on abstract interpretation, which I always postponed amicably. But he convinced Radhia, and the seminar was finally organized on January 28th, 1999 with Alain Deutsch, Éric Goubault, Nicolas Halbwachs, and Arnault Venet. The numerous participants from industry looked to enjoy. At the end of the day, Radhia spoke with Famantanantsoa Randimbivololona from Airbus who asked her if we could have more advanced discussions. Essentially, the Polyspace analyzer was used at Airbus, but produced too many false alarms. After many meetings to identify the needs, we arrived at the conclusion that we should gather top-level researchers in a European project to make proposals for improving Polyspace and solve other urgent problems that we had identified. This was the Daedalus IST FP5 European project (1999-2003). Some problems advanced very well (like the WCET analysis [199]) but Polyspace was reluctant to integrate the proposals done by the group.

§ 33. ASTRÉE So I proposed to make a small prototype that would show that the proposals were worthwhile. After a discussion of the pros and cons followed by a secret vote of the team members, Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival, which turned unanimous, the static analyzer project was launched. From small it grew to big and was later named ASTRÉE (Analyseur Statique Temps RÉel Embarqué). After ten years of academic research, including the successful application to the proof of absence of runtime errors in the control-command code of the A380 before its maiden flight in January 2005, the CNRS licensed ASTRÉE to AbsInt, now the main developer and unique distributor.

The success of ASTRÉE is based on its modular, extensible, and specializable structure, soundness, and scalability [17, 18, 126, 12, 13, 14]. Based on universal abstract domains (like intervals and octagons), it is organized as a hierarchical reduced product (with partial and ordered reduction) [127], allowing for the extension with specialized domains (such as choice trees for data case analysis, filters, and dozen others). Control includes loop unrolling, trace partitioning for control case analysis,

which together with widening allows for handling inductive properties much more refined than mere invariants. Beyond soundness, precision, and usability, the main difficulty was scalability [128] for which Astrée overwhelmingly outperforms other static analyzers [129] [30].

The original program was about 60.000 lines of OCaml, now more than 265.000 lines, not counting the much larger human interface[31]. The size of programs that can be analyzed with precision has scale to over 10.000.000 lines of C/C++ in one hour (which, is a 5.000 factor in size compared to model-checking of automotive codes [198], itself maybe a factor 100 compared to the benchmarks used in academic competitions for evaluating software verification systems). The licensed users are in the thousands. The general development by AbsInt is towards generality (starting with dynamic memory allocation, recursion, non-forward gotos, etc. initially prohibited) while preserving soundness, and improving precision and efficiency. In 2020, Astrée was the most precise of the only two static analyzers satisfying the NIST's Ockham criteria of the Software Assurance Metrics And Tool Evaluation project, both abstract interpretation-based. The criteria include precision and soundness. Unexpected bugs where found in the benchmarks [16].

A number of papers report on early applications of Astrée [55, 19, 160, 162, 11, 10, 174, 161] or suggest potential ones [52], including by exceptionally experienced end-users such as [191]. These papers have a handful of citations[32].

§ 34. Applications We have studied a number of subproblems and applications of abstract interpretation such as

- model-checking refined by abstract interpretation [104, 105], by expressive temporal logics [107] (so that model-checking is undecidable even for finite transition systems), by refined refinement [137] (COUGAR is better the CEGAR);
- program transformation (including online and offline partial evaluation) [108, 48, 113];
- abstract testing [106, 200];
- software watermarking [115];
- predicate abstraction [51];
- logic programming languages [130];
- array analysis [131];

[30] Some anonymous reviewers of [129] asked us to suppress the name of the other static analyzers surpassed by Astrée, but we refused. One author of [198] told us that they had similar difficulties. Reports on failures on hard industrial problems do not seem well accepted by part of the academic community whose successes are confined to academic benchmarks only.

[31] The human interface providing explanations of the analysis results is essential since, for example, many testers are not programmers and have difficulties to understand formal concepts like invariant, undecidability, approximation, false alarm, etc.

[32] It may be that most academics in formal methods are more interested by the social process of recognition by their peers than by the effective industrial application of their ideas. One should also distinguish between experimental industrial applications such as bug-finding versus mandatory static analyses in the software product development cycle to provide strong guarantees for correctness. The level of guarantees and difficulties in the two cases are not really comparable [149].

- necessary preconditions, in theory [125] and in practice [125];
- abstract interpretation-based tools in a programming environment [169];
- code refactoring [132];
- security analysis of web applications [193];
- dependency analysis [62] (a property comparing any two pairs of executions);
- responsibility analysis [145, 144] (a property comparing any execution to all others and involving both over and under approximations);
- dynamic analysis [70] (one of the few cases with strictness analysis requiring both a computational and an approximation ordering);
- [bi]simulations [69] and their hybrid version [143] for discrete computations interacting with continuous processes

including the application of abstract interpretation to itself to understand the iteration process as in [139].

§ 35. Introductions, surveys, reports, and prospective discussion texts Over the years, I have been invited to write a number of introductions [36, 116, 120, 63], surveys [35, 95, 45, 112, 56, 156, 58], reports [80, 180, 49, 51], and prospective discussion texts [37, 43, 44, 110, 47, 54, 123, 60] on abstract interpretation, with an average of about 100 citations on Google Scholar, so one may have doubts on their usefulness[33].

§ 36. Conclusion Abstract interpretation is a unifying conceptual tool to formalize reasonings in semantics, verification, and dynamic and static analysis of sequential or parallel programs.

Of course this paper refers only to our own work and initial inspirations and superbly ignores the contributions of many other scientists and engineers to abstract interpretation [154]. The task of reporting on all contributions looked insurmountable to me! Even the book [57] with 2000 citations only accounts for a very small fraction of the work on abstract interpretation.

Because automatic program verification is undecidable, all the proposed methods for program analysis have to abandon some desirable features like full automation, termination, scalability, soundness, etc., all guaranteed by abstract interpretation-based static analysis. Not surprisingly, we abandoned completeness only, that is an answer may be "yes", "no, or "I don't know" (a sound approximation sometimes confused with unreliableness!). Of course, precision can be improved indefinitely. So the practical problem is to find a useful balance between expressivity, cost, precision, and usability.

Although abstract interpretation has certainly been influential, its academic recognition is modest (as shown by our reference counts on Google Scholar). Nevertheless, it is certainly the only fully automatic, sound, and scalable verification formal method widely adopted in safety-critical industry. So our initial goal has been achieved. More rigorous lawful requirements on the use of certified tools for software verification

[33] The most cited survey [95] with 891 citations on Google Scholar is curiously the one in which the editor published the galley proofs where typographers introduced hundreds of errors, making the text unreadable, but nevertheless cited!

would certainly contribute to a wider adoption in those software industries where safety and security are nowadays of secondary or even no interest because of high costs.

§ 37. Acknowledgements I thank Bertrand Meyer for inviting me to contribute to this volume and the three anonymous reviews for their careful reading, corrections, and nice constructive comments.

References

1. P. Aczel. An introduction to inductive definitions. In John Barwise, editor, *Handbook of Mathematical Logic*, chapter 7, pages 739–782. North–Holland, Amsterdam, 1977.
2. J. Alglave and P. Cousot. Syntax and analytic semantics of LISA. *CoRR*, abs/1608.06583, 2016.
3. J. Alglave and P. Cousot. Ogre and pythia: an invariance proof method for weak consistency models. In *POPL*, pages 3–18. ACM, 2017.
4. J. Alglave, P. Cousot, and L. Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.
5. F.E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19. ACM, 1970.
6. A.W. Appel. *Program Logics – for Certified Compilers*. Cambridge University Press, 2014.
7. Paolo Baldan, Francesco Ranzato, and Linpeng Zhang. A Rice's theorem for abstract semantics. In *ICALP*, volume 112 of *Leibniz International Proceedings in Informatics*, pages 112:1–112:19. Dagstuhl Publishing, Germany, 2021.
8. G. Beaudet. Asychronous iterative methods for multiprocessors. Technical report, Carnegie Mellon University, Pittsburgh, PA, November 1976.
9. C. Berge. *Graphes et hypergraphes*. Dunod Université, Dunod, Paris, 1973.
10. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. L'analyseur statique Astrée (in french). In J.-L. Boulanger, editor, *Utilisations industrielles des techniques formelles : interprétation abstraite*, pages 67–114. Hermès Science, Paris, France, June 2011.
11. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, January 2011.
12. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*, Atlanta, Georgia, 20–22 April 2010. American Institute of Aeronautics and Astronautics.
13. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. In *Third IEEE International workshop UML and Formal Methods*, Shanghai, China, 16 November 2010. IEEE.
14. Julien Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. *Found. Trends Program. Lang.*, 2(2-3):71–190, 2015.
15. G. Birkhoff. *Lattice Theory*. American Mathematical Society, Colloquium Publications, Volume XXV, 3 edition, 1973.
16. P.E. Black and K. Singh Walia. SATE VI Ockham sound analysis criteria. Technical Report Intern. Rep. 8304, National Institute of Standards and Technology, May 2020.
17. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.
18. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.
19. O. Bouissou, É. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, É. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, volume SP-669, pages 1–7, Istambul, Turkey, May 2009. ESA.
20. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.
21. J. Burghardt, F. Kammüller, and J.W. Sanders. On the antisymmetry of Galois embeddings. *Inf. Process. Lett.*, 79(2):57–63, 2001.
22. R.M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 308–312. North-Holland, 1974.
23. D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2:199–222, 1969.
24. J. Chen and P. Cousot. A binary decision tree abstract domain functor. In *SAS*, volume 9291 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2015.
25. L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

26. L. Chen, A. Miné, Ji Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2009.
27. L. Chen, A. Miné, Ji Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2010.
28. L. Chen, A. Miné, Ji Wang, and P. Cousot. Linear absolute value relation analysis. In *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2011.
29. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Sep. 1977. 15 p.
30. P. Cousot. An introduction to a mathematical theory of global program analysis. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, 19 p., Mar. 1977.
31. P. Cousot. *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 de Grenoble Alpes, March 1978.
32. P. Cousot. Analysis of the behavior of dynamic discrete systems, part i: deterministic systems. Res. rep. R.R. 161, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Jan. 1979. 34 p.
33. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
34. P. Cousot. A hoare-style axiomatization of Burstall's intermittent assertion method for non-deterministic programs. Technical report, University Paul Verlaine, Metz, France, September 1983.
35. P. Cousot. Methods and logics for proving programs. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 841–993. Elsevier and MIT Press, 1990.
36. P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.
37. P. Cousot. Program analysis: The abstract interpretation perspective. *ACM Comput. Surv.*, 28(4es):165, 1996.
38. P. Cousot. Abstract interpretation based static analysis parameterized by semantics. In *SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 388–394. Springer, 1997.
39. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *MFPS*, volume 6 of *Electronic Notes in Theoretical Computer Science*, pages 77–102. Elsevier, 1997.
40. P. Cousot. Design of semantics by abstract interpretation, invited address. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, March 23–26 1997.
41. P. Cousot. Types as abstract interpretations. In *POPL*, pages 316–331. ACM Press, 1997.
42. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
43. P. Cousot. Directions for research in approximate system analysis. *ACM Comput. Surv.*, 31(3es):6, 1999.
44. P. Cousot. Abstract interpretation: Achievements and perspectives. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, L'Aquila, Italy, July 31 – August 6 2000.
45. P. Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2-3):155–164, January 2000.
46. P. Cousot. Partial completeness of abstract fixpoint checking. In *SARA*, volume 1864 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.
47. P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer, 2001.
48. P. Cousot. Design of syntactic program transformations by abstract interpretation of semantic transformations. In *ICLP*, volume 2237 of *Lecture Notes in Computer Science*, pages 4–5. Springer, 2001.
49. P. Cousot. Abstract interpretation: Theory and practice. In *SPIN*, volume 2318 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2002.
50. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
51. P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 243–268. Springer, 2003.
52. P. Cousot. Integrating physical systems in the static analysis of embedded control software. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 135–138. Springer, 2005.
53. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2005.
54. P. Cousot. The verification grand challenge and abstract interpretation. In *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2005.
55. P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *EMSOFT*, pages 7–9. ACM, 2007.
56. P. Cousot. The rôle of abstract interpretation in formal methods. In *SEFM*, pages 135–140. IEEE Computer Society, 2007.
57. P. Cousot. *Principles of Abstract Interpretation*. MIT Press, 21 September 2011.
58. P. Cousot. Formal verification by abstract interpretation. In *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2012.
59. P. Cousot. Abstracting induction by extrapolation and interpolation. In *VMCAI*, volume 8931 of *Lecture Notes in Computer Science*, pages 19–42. Springer, 2015.
60. P. Cousot. On various abstract understandings of abstract interpretation. In *TASE*, pages 2–3. IEEE Computer Society, 2015.
61. P. Cousot. Verification by abstract interpretation, soundness and abstract induction. In *PPDP*, pages 1–4. ACM, 2015.

62. P. Cousot. Abstract semantic dependency. In *SAS*, volume 11822 of *Lecture Notes in Computer Science*, pages 389–410. Springer, 2019.
63. P. Cousot. A formal introduction to abstract interpretation. In Alexander Pretschner, P. Müller, and P. Stöckle, editors, *Engineering Secure and Dependable Software Systems*. NATO SPS, Series D, Vol. 53. IOS Press, Amsterdam, 2019.
64. P. Cousot. On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics. In *LOPSTR*, volume 12042 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2019.
65. P. Cousot. Syntactic and semantic soundness of structural dataflow analysis. In *SAS*, volume 11822 of *Lecture Notes in Computer Science*, pages 96–117. Springer, 2019.
66. P. Cousot. The symbolic term abstract domain. *TASE, Hangzhou, China*, December 2020.
67. P. Cousot. Abstract interpretation of graphs. In John P. Gallagher, R. Giacobazzi, and Pedro López-García, editors, *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems (AVERTIS)*, 2021. to appear.
68. P. Cousot. Calculational design of a regular model checker by abstract interpretation. *Theor. Comput. Sci.*, 869:62–84, 2021.
69. P. Cousot. Correspondences between concrete and abstract semantics: Homomorphisms, [bi]simulations, refinements, preservation, logical relations, galois connections, closures, and approximations. refused for publication at POPL'22 with one A and three incomprehensibility comments., July 2021.
70. P. Cousot. Dynamic interval analysis by abstract interpretation. In *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2021, Rhodes, Greece*, 2021. to appear.
71. P. Cousot and R. Cousot. Static verification of dynamic type properties of variables. Res. rep. R.R. 25, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Nov. 1975. 18 p.
72. P. Cousot and R. Cousot. Vérification statique de la cohérence dynamique des programmes. Res. rep., Rapport du contrat IRIA SESORI N° 75-035, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, 23 Sep. 1975. 125 p.
73. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
74. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
75. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGART Newsl.*, 64:1–12, 1977.
76. P. Cousot and R. Cousot. Fixed point approach to the approximate semantic analysis of programs. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, 48 p., June 1977.
77. P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94. ACM, 1977.
78. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Description of Programming Concepts*, pages 237–278. North-Holland, 1977.
79. P. Cousot and R. Cousot. Towards a universal model for static analysis of programs. Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, 90 p., Jan. 1977.
80. P. Cousot and R. Cousot. Exemples d'analyse sémantique automatique de programmes. In *Actes des journées d'études sésori, « Synthèse, manipulation et transformation de programmes », Saint-Rémy de Provence, France*, pages 111–130. Publication IRIA, 10–12 May 1978.
81. P. Cousot and R. Cousot. A constructive characterization of the lattices of all retractions, pre–closure, quasi–closure and closure operators on a complete lattice. *Portugaliæ Mathematica*, 38(2):185–198, 1979.
82. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.
83. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM Press, 1979.
84. P. Cousot and R. Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980.
85. P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes (shortened version). In *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1980.
86. P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools & Notions for Program Construction: an Advanced Course*, pages 75–119. Cambridge University Press, Cambridge, UK, August 1982.
87. P. Cousot and R. Cousot. "à la Burstall"induction principles for proving inevitability properties of programs. Res. rep. LRIM-83-08, University Paul Verlaine, Metz, France, November 1983.
88. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, New York, United States, 1984.
89. P. Cousot and R. Cousot. 'a la Floyd' induction principles for proving inevitability properties of programs. In M. Nivat and J. Reynolds, editors, *Algebraic methods in semantics*, pages 277–312. Cambridge University Press, Cambridge, UK, December 1985.
90. P. Cousot and R. Cousot. Principe des méthodes de preuve de propriétés d'invariance et de fatalité des programmes parallèles. In J.-P. Verjus and G. Roucairol, editors, *Parallélisme, communication et synchronisation*, pages 129–149. Éditions du CNRS, Paris, 1985, ISBN 2-222-03672-0.
91. P. Cousot and R. Cousot. Sometime ≡ always + recursion ≡ always on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs. *Acta Informatica*, 24(1):1–31, 1987.
92. P. Cousot and R. Cousot. A language independent proof of the soundness and completeness of generalized hoare logic. *Inf. Comput.*, 80(2):165–191, 1989.

93. P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.
94. P. Cousot and R. Cousot. Relational abstract interpretation of higher-order functional programs. JTASPEFL '91, Bordeaux. *BIGRE*, 74:33–36, October 1991.
95. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.
96. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
97. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, 1992.
98. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *POPL*, pages 83–94. ACM Press, 1992.
99. P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis (invited paper). In *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 98–127. Springer, 1993.
100. P. Cousot and R. Cousot. Invited talk: Higher order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection, and PER analysis. In *ICCL*, pages 95–112. IEEE Computer Society, 1994.
101. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 1995.
102. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181. ACM, 1995.
103. P. Cousot and R. Cousot. Abstract interpretation of algebraic polynomial systems (extended abstract). In *AMAST*, volume 1349 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 1997.
104. P. Cousot and R. Cousot. Parallel combination of abstract interpretation and model-based automatic analysis of software. In R. Cleaveland and D. Jackson, editors, *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, AAS'97*, pages 91–98, Paris, France, January 1997. ACM Press.
105. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6(1):69–95, 1999.
106. P. Cousot and R. Cousot. Abstract interpretation based program testing. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248 and electronic proceedings `http://www.ssgrr.it/en/ssgrr2000/proceedings.htm`, L'Aquila, Italy, July 31 – August 6 2000. Scuola Superiore G. Reiss Romoli.
107. P. Cousot and R. Cousot. Temporal abstract interpretation. In *POPL*, pages 12–25. ACM, 2000.
108. P. Cousot and R. Cousot. A case study in abstract interpretation based program transformation: Blocking command elimination. In *MFPS*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 41–64. Elsevier, 2001.
109. P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR 2001*, Compact disk, L'Aquila, Italy, 6–12 August, 2001 2001. Scuola Superiore G. Reiss Romoli.
110. P. Cousot and R. Cousot. Verification of embedded software: Problems and perspectives. In *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2001.
111. P. Cousot and R. Cousot. Modular static program analysis. In *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.
112. P. Cousot and R. Cousot. On abstraction in software verification. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2002.
113. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, pages 178–190. ACM, 2002.
114. P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theor. Comput. Sci.*, 290(1):531–544, 2003.
115. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *POPL*, pages 173–185. ACM, 2004.
116. P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In *IFIP Congress Topical Sessions*, volume 156 of *IFIP*, pages 359–366. Kluwer/Springer, 2004.
117. P. Cousot and R. Cousot. Grammar analysis and parsing by abstract interpretation. In *Program Analysis and Compilation*, volume 4444 of *Lecture Notes in Computer Science*, pages 175–200. Springer, 2006.
118. P. Cousot and R. Cousot. Bi-inductive structural semantics: (extended abstract). *Electron. Notes Theor. Comput. Sci.*, 192(1):29–44, 2007.
119. P. Cousot and R. Cousot. Bi-inductive structural semantics. *Inf. Comput.*, 207(2):258–283, 2009.
120. P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–29. IOS Press, 2010.
121. P. Cousot and R. Cousot. Grammar semantics, analysis, and parsing by abstract interpretation. *Theor. Comput. Sci.*, 412(44):6135–6192, 2011.
122. P. Cousot and R. Cousot. An abstract interpretation framework for termination. In *POPL*, pages 245–258. ACM, 2012.
123. P. Cousot and R. Cousot. Abstract interpretation: past, present and future. In *CSL-LICS*, pages 2:1–2:10. ACM, 2014.
124. P. Cousot and R. Cousot. A Galois connection calculus for abstract interpretation. In *POPL*, pages 3–4. ACM, 2014.
125. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 2013.
126. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

127. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.
128. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods Syst. Des.*, 35(3):229–264, 2009.
129. P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée. In *TASE*, pages 3–20. IEEE Computer Society, 2007.
130. P. Cousot, R. Cousot, and R. Giacobazzi. Abstract interpretation of resolution-based semantics. *Theor. Comput. Sci.*, 410(46):4724–4746, 2009.
131. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.
132. P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA*, pages 213–232. ACM, 2012.
133. P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. In *The Future of Software Engineering*, pages 48–71. Springer, 2010.
134. P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 72–95. Springer, 2010.
135. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FoSSaCS*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer, 2011.
136. P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *J. ACM*, 59(6):31:1–31:56, 2012.
137. P. Cousot, P. Ganty, and J-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2007.
138. P. Cousot, R. Giacobazzi, and F. Ranzato. Program analysis is harder than verification: A computability perspective. In *CAV (2)*, volume 10982 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2018.
139. P. Cousot, R. Giacobazzi, and F. Ranzato. A^2i: abstract^2 interpretation. *Proc. ACM Program. Lang.*, 3(POPL):42:1–42:31, 2019.
140. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM Press, 1978.
141. P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012.
142. Patrick Cousot. Dynamic interval analysis by abstract interpretation. In *Formal Methods in Outer Space*, volume 13065 of *Lecture Notes in Computer Science*, pages 61–86. Springer, 2021.
143. Patrick Cousot. Asynchronous correspondences between hybrid trajectory semantics. *CoRR*, abs/2209.14945, 2022.
144. C. Deng and P. Cousot. Responsibility analysis by abstract interpretation. In *SAS*, volume 11822 of *Lecture Notes in Computer Science*, pages 368–388. Springer, 2019.
145. Chaoqiang Deng and Patrick Cousot. The systematic design of responsibility analysis by abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 44(1):3:1–3:90, 2022.
146. E.W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.
147. E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
148. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
149. E.W. Dijkstra. On the reliability of programs. circulated privately, n.d.
150. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990.
151. E.W. Dijkstra and A.J.M. van Gasteren. A simple fixpoint argument without the restriction to continuity. *Acta Informatica*, 23(1):1–7, 1986.
152. R.W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proc. Symp. in Applied Math.*, volume 19, pages 19–32. Amer. Math. Soc., 1967.
153. L. Franceschino, D. Pichardie, and J.n-P. Talpin. Verified functional programming of an abstract interpreter. *CoRR*, abs/2107.09472, 2021.
154. R. Giacobazzi and F. Ranzato. History of abstract interpretation. *IEEE Annals of the History of Computing*, To appear.
155. N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse de 3ème cycle informatique, Université de Grenoble Alpes, Grenoble, France, March 1979.
156. M. Hinchey, M. Jackson, P. Cousot, B. Cook, J.P. Bowen, and T. Margaria. Software engineering and formal methods. *Commun. ACM*, 51(9):54–59, 2008.
157. J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *POPL*, pages 247–259. ACM, 2015.
158. M. Karr. On affine relationships among variables of a program. Technical report, CA-7402-2811, Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, Mass. 01880, U.S.A., February 1974.
159. M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
160. D. Kästner, C. Ferdinand, S. Wilhelm, S. Nevona, O. Honcharova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, and É.-J. Sims. Astrée: Nachweis der abwesenheit von laufzeitfehlern. In *Workshop "Entwicklung zuverlässiger Software-Systeme"*, Regensburg, Germany, 18 June 2009.
161. D. Kästner, A. Miné, S. Wilhelm, X. Rival, A. Schmidt, J. Feret, P. Cousot, and C. Ferdinand. Finding all potential run-time errors and data races in automotive software. In *WCX 17: SAE World Congress Experience, April 4-6, 2017 Detroit, Michigan, USA SAE Technical Paper 2017-01-0054*, March 2017.
162. D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, and X. Rival. Astrée: Proving the absence of runtime errors. In *Embedded Real Time Software and Systems - ERTSS 2010*, 2010.

163. S. Katz and Z. Manna. Logical analysis of programs. *Commun. ACM*, 19(4):188–206, 1976.
164. G.A. Kildall. *Global expression optimization during compilation*. Phd, University of Washington, Computer Science Group, TR 72-06-02, 1972.
165. G.A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206. ACM Press, 1973.
166. J.C. King. On affine relationships among variables of a program. IBM Reserach Report RC5082, T. J. Watson Reserach Center, Yorktown Heights, N. Y,, October 1974.
167. J.C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
168. J.-L. Lassez, V.L. Nguyen, and L. Sonenberg. Fixed point theorems and semantics: A folk tale. *Inf. Process. Lett.*, 14(3):112–116, 1982.
169. F. Logozzo, M. Barnett, M. Fähndrich, P. Cousot, and R. Cousot. A semantic integrated development environment. In *SPLASH*, pages 15–16. ACM, 2012.
170. Z. Manna, S. Ness, and J. Vuillemin. Inductive methods for proving properties of programs. *Commun. ACM*, 16(8):491–502, 1973.
171. Z. Manna and A. Shamir. The optimal fixedpoint of recursive programs. In *STOC*, pages 194–206. ACM, 1975.
172. Zohar Manna and Richard J. Waldinger. Is "sometime" sometimes better than "always"? (intermittent assertions in proving program correctness). *Commun. ACM*, 21(2):159–172, 1978.
173. J.-C. Miellou. Algorithmes de relaxation : propriétés de convergence monotone. Séminaire d'Analyse Numérique n° 278, Université scientifique et médicale de Grenoble, Grenoble, France, June 1977.
174. A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking static analysis to the next level: Proving the absence of run-time errors and data races with Astrée. In *8th European Congress on Embedded Real-Time Software and Systems, Toulouse, France*, January 2016.
175. J.D. Monk. *Introduction to Set Theory*. McGraw–Hill, 1969.
176. A. Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, University of Edinburgh, UK, 1982.
177. P. Naur. The design of the GIER ALGOL compiler. *BIT Numerical Mathematics*, 3:124–140 and 145–166, June 1963.
178. P. Naur. Checking of operand types in ALGOL compilers. *BIT Numerical Mathematics*, 5:151–163, 09 1965.
179. J. Von Neumann. Zur Einführung der transfiniten Zahlen. *Acta Scientiarum Mathematicarum (University of Szeged)*, 1(4):199–208, 1923.
180. F. Nielson, P. Cousot, M. Dam, P. Degano, P. Jouvelot, A. Mycroft, and B. Thomsen. Logical and operational methods in the analysis of programs and systems. In *LOMAPS*, volume 1192 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 1996.
181. D.M.R. Park. Fixpoint induction and proofs of program properties. *Machine Intelligence.*, 5:59–78, 1969.
182. G.D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 1972–01:17–139, 2004.
183. Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Trans.Amer.Math.Soc.*, 74(1):358–366, 1953.
184. F. Robert. Convergence locale d'itérations chaotiques non linéaires. Technical Report n° 58, L.A. 7, Université scientifique et médicale de Grenoble, Grenoble, France, Dec. 1976.
185. M. Rosendahl. Higher-order chaotic iteration sequences. In *PLILP*, volume 714 of *Lecture Notes in Computer Science*, pages 332–345. Springer, 1993.
186. D.S. Scott. The lattice of flow diagrams. In *Symposium on Semantics of Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 311–366. Springer, 1971.
187. D.S. Scott. Continuous lattices. In F.W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic. Dalhousie University, Halifax, January 16–19, 1971*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer, 1972.
188. D.S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.
189. D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computer Laboratory, August 1971.
190. M. Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM Conference on Proving Assertions About Programs*, pages 203–207. ACM, 1972.
191. J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. In *SAFECOMP*, volume 4680 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2007.
192. A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific J. of Math.*, 5:285–310, 1955.
193. O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2013.
194. Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382. Springer, 1996.
195. M. Ward. The closure operators of a lattice. *Annals of Mathematics*, 43(2):191–196, April 1942.
196. B. Wegbreit. Property extraction in well-found property sets. Technical report, Center for Research in Computing Technology, Harward University, Cambridge, Massachusetts, February 1973.
197. B. Wegbreit. Property extraction in well–founded property sets. *IEEE Trans. Software Eng.*, 1(3):270–285, 1975.
198. L. Westhofen, Ph. erger, and J.P. Katoen. Benchmarking software model checkers on automotive code. *CoRR*, abs/2003.11689, 2020.
199. R. Wilhelm. Real time spent on real time. *Commun. ACM*, 63(10):54–60, November 2020.
200. B. Yin, L. Chen, J. Liu, Ji Wang, and P. Cousot. Verifying numerical programs via iterative abstract testing. In *SAS*, volume 11822 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2019.