Article

History of Abstract Interpretation

Roberto Giacobazzi
University of Verona, Italy

Francesco Ranzato
University of Padova, Italy

***Abstract*—We trace the roots of abstract interpretation and its role as a foundational principle to understand and design static program analysis and verification methods. Starting from the historical roots of formal methods and static program analysis, we show how abstract interpretation evolved and influenced the way we reason about program correctness in different programming languages and how this method shaped the literature and the practice in program analysis in the last 45 years.**

INTRODUCTION Abstract interpretation provides a universal principled method for designing static program analyzers. In this article we trace the mathematical and computational roots of abstract interpretation, from its origins as a theory for generalizing program analysis and verification methods to the following developments and applications, with a focus on the influence that abstract interpretation had in programming languages (PL).

As a major achievement in programming languages, abstract interpretation established a precise bridge between program semantic models and program analysis algorithms. The fundamental idea that analyzing programs corresponds precisely to approximate their semantics was in the air in the mid-late 1970s, but was lacking of a rigorous mathematical foundation. In this context, the fundamental early works delineating a mathematical theory of computation by Turing, von Neumann, McCarthy and the initial conceptualisations of the logical correctness of programs by Floyd, Hoare and Naur in the 1960-70s, set up an ideal background for the irruption of abstract interpretation as a breakthrough in program analysis and verification.

In this historical perspective, we describe how Patrick and Radhia Cousot envisaged the fundamental concepts of abstract interpretation starting from the very first embryonic ideas dating back to 1972 and culminating in the landmark POPL 1977 paper, that is widely acknowledged as the origin of abstract interpretation. We then discuss how these ideas influenced the way we analyze and, more in general, reason about programs. We show how abstract interpretation culminated in many successful industrializations in the 2000s, disseminating this methodology in a rich spectrum of fields in computer science, and making it a mature and lively research discipline.

Up to 1970: HISTORICAL ROOTS

The Many Computational Roots

Two main complications characterized the origins of programming languages as a distinctive discipline in computer science: (1) The call for program verification — i.e., methods and tools for verifying the logical correctness of programs — and (2) the need of efficient compile-time and run-time support to optimize code in order to overcome the inherent limitations of early computer architectures. The objective of designing computer programs by combining suitable annotations that express the intended semantics of program components can be rooted back to Charles Babbage's idea of *mechanical notation* as his grammar of a symbolic language [1]. This was meant to be a system of signs describing the semantic properties of composable mechanical entities in such a way that the resulting machine is correct by construction. A striking quote of Babbage about the importance of using a suitable notation in the analytical engine [2] is: "*By the aid of the Mechanical Notation, the Analytical Engine became a reality, for it became susceptible of demonstration*". Although the analytical machine can be considered a very first instance of a universal Turing Machine, a precise notion of program correctness appeared about a century later with the development of the first computers in the 1940s. Abstract flow diagrams describing the control and data flow of programs emerged from the pioneering visions of both Herman Goldstine and John von Neumann in their 1947 report [3], [4] and Alan Mathison Turing in his 1949 paper for the inaugural conference of the EDSAC computer at the Mathematical Laboratory of Cambridge [5]. It is worth remarking that, although not essential for its general formulation, flow diagrams played a notational role to model program semantics in the early stages of abstract interpretation.

Proving program correctness is way harder than syntactically composing denotations and signs [6]. It requires more advanced levels of abstraction to cope with both the linguistic complications of modern programming languages [7] and the inherent undecidability of semantic properties of programs. It is within this scope that abstract interpretation gave the most of its contribution. The 1960s saw the emergence of software bugs as harbingers of possible disastrous events, e.g., the Mariner I destruction [8]. As observed by Matti Tedre [8]: "*the 1960s were characterized by public disillusionment with computing technology. The larger the software project, the more certainly it was late, over budget, and bug ridden*". The vision of *programs as mathematical models* has been put forward by John McCarthy in his fundamental papers for the 1962 IFIP Congress [9] and the 1961 Western Joint Computer Conference [10]. A recent article [4] provides an extensive and thoughtful exploration of those fundamental years for the development of modern programming languages. As recognized by Patrick Cousot in his PhD thesis [11], the idea of formally proving the validity of some correctness specifications for programs is essentially due to Peter Naur [12], whose *general snapshots* are defined to be "*static conditions existing whenever the execution of the algorithm reaches particular points*". A widely attributed ancestor of abstract interpretation is indeed Naur's static checking of operand types for Algol programs [13], where "*the basic method is a pseudo-evaluation of the expressions of the program that proceeds like a run-time evaluation, but works with descriptions of the types and kinds of the operands instead of with values*".

The field of program semantics was established in 1967-1971 by several influential advances. On the one hand, we have the approaches based on logical semantics for proving programs correct, pioneered by Robert W. Floyd in 1967 [14], Sir Charles Antony Richard Hoare in 1969 [15] and Rodney Martineau "Rod" Burstall in 1969 [16], that are now commonly referred to as the Hoare (or Floyd-Hoare) program logic. On the other hand, a range of mathematical approaches based on algebraic ordered structures and fixed points that model the input/output program behaviour, initiated by Dana Stewart Scott and Christopher S. Strachey [17], [18], and afterwards by Zohar Manna, Stephen Ness and Jean Etienne Vuillemin [19], eventually called *à la* Scott (or Scott-Strachey) program semantics. In general, a fixed point is a solution to a recursive equation of the form $x = f(x)$. Whenever $f(x)$ represents the possible states (memory, registers, etc.) of a computational device after the exe-

cution of a finite sequence of instructions of a given program with input x, then a fixed point $x = f(x)$ represents a *program invariant*, namely a property of program states that is stable with respect to further applications of the same instructions. As observed by Floyd [14], invariants represent properties of recursive definitions as well as of iterative commands and, as pointed out by Edsger Wybe Dijkstra [20], finding strong enough invariants is the main challenge in proving program correctness, due to the undecidability of program termination. In their very essence, program analysis and program verification correspond precisely to automatically derive — i.e., through an algorithm — an invariant which is strong enough to let us prove statically — i.e., at compile-time or anyway before program execution — that some expected or desired property holds at run-time [21]. Major examples of program properties that can be statically inferred by automatically deriving program invariants are the absence of bugs — the so-called safety program properties — and the information on program variables used for code optimization in program compilation, such as liveness and constancy of variables as derived by classical data-flow analysis. This view that automatic program verification is feasible and practically useful, although only approximate information can be derived, is in line with the thesis supported by [22], that argued how the famous claim made in 1979 by De Millo, Lipton and Perlis [23] that "*formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics*" has been largely refuted by the scientific achievements that showed how "*formal verification is at present a concrete reality, permitting correctness proofs of complex software applications*" [22].

In program compilers, the origins of a theory for code optimization — and, in this perspective, of program analysis — can be rooted back to the works of Frances Elizabeth Allen [24], [25], and John Cocke and Raymond E. Miller [26], [27], although empirical ad hoc methods were in use long before then. Before the 1970s, *automatic coding* was highly machine-specific [7], [28], hence justifying the idea that optimizing code is mostly a machine-dependent task. The rise of so-called universal programming languages put forward the need of more abstract, yet automatic, methods for optimizing code. The necessity of statically extracting properties of the run-time behavior of programs was justified to simplify numerical expressions or to move invariant code out of program loops for optimization purposes [7], [29]. In this context, program analysis originally had a mostly algorithmic flavor, i.e., it was based on particular algorithmic solutions to specific optimization problems, as in the directed acyclic graph (DAG) representation of basic blocks in flow diagrams by Alfred Vaino Aho and Jeffrey David Ullman [30], or in program optimizations based on reducible graph properties by Allen [25]. These algorithms, although efficient, strongly relied on the intermediate code representations used in program compilation. For example, a basic control-flow analysis such as *reaching definitions* — devoted to detect which variable definitions (that is, left-hand sides of assignments) reach a given program point before this same variable is reassigned — was already conceived as an iterative fixed point solution of a system of equations derived from the program syntax [30], although no explicit relation between the intended semantics of the program and the program property to infer was mentioned. The lack of a clearly stated universal correspondence between a rigorous model of program semantics, e.g. the operational one given by a program interpreter, and the program property to infer or verify, reduced the possibility of bridging program analysis and program verification, as well as constrained the chances of systematically designing new analysis algorithms, beyond the specifics of some program optimization of interest. Monotone data flow frameworks appeared later in the mid 1970s [31], simultaneously with the publication of the first paper [32] introducing abstract interpretation to a major international research venue: The 1977 ACM Symposium on Principles of Programming Languages (POPL). Monotone data flow frameworks were conceived with the same purpose of abstract interpretation, namely to provide a generalization of the ad hoc nature of the program analysis methods introduced in the early 1970s. As recognized in the

aforementioned POPL 1977 paper [32], that cited a 1975 technical report on monotone data flow frameworks [33], this latter approach appeared to be an instance of the more general idea of abstract interpretation, for the specific case of bounded lattices, namely, when the fixed point solution of a system of equations can be obtained within a given bounded time which is known a priori.

The Mathematical Roots

The oldest papers cited by the PhD thesis [11] of Patrick Cousot are a number of mathematical articles of the 1940s on topological closure operators by António Aniceto Monteiro and Hugo Ribeiro [34], Oystein Ore [35], [36] and Morgan Ward [37], and on the celebrated 1928 and 1955 fixed point theorems in lattice theory by Bronislaw Knaster and Alfred Tarski [38], [39]. Moreover, Cousot cited a long array of articles on closure operators by the Portuguese mathematician José Morgado [40], very active from 1960 to 1966 in the algebraic study of this class of operators, most of them published in the Portugaliæ Mathematica journal [41]. A major achievement of abstract interpretation was to recognize that closure operators formalize in a simple and elegant way the notion of approximation, also called *abstraction*, of program properties. Interpreted on an ordered algebraic structure of properties of program states, a closure operator is:

1) *monotone*, i.e., the abstraction must preserve the relative precision of the input properties;
2) *extensive*, i.e., approximate properties are weaker (namely, larger for the ordering) than their input;
3) *idempotent*, i.e., all the loss of information in approximating an input property is achieved at once.

The result of applying a closure operator to all the properties of interest on the program behavior was called *abstract domain* and laid its foundations into this theory of closure operators and its mathematically equivalent notion provided by Galois connections [42]. This formal model of the notion of approximation on top of a mathematical formalization of the semantics of programs in terms of fixed point equations, provided the very first general model for approximating program properties, such as the range of variation of a numerical program variable and the (linear or non-linear) relationships between different program variables. This type of information was, and still is, crucial for optimizing program compilation and detecting software bugs. Closure operators, Galois connections and fixed points historically played the role of mathematical backstage of abstract interpretation, conceiving its very essential idea that:

program analysis is approximating a fixed point model of program semantics.

1972-76: THE ORIGINS

In the historical context of early 1970s described above, in 1972 Radhia Cousot — born Rezig and later married to Patrick Cousot — worked on precedence parsing for the Algol 68 programming language. Radhia Rezig's approach to parsing relied on a program pre-processing by static analysis and a grammar transformation before building the actual bottom-up parser. This work was presented in 1972 as Radhia Rezig's master thesis [43] at the Université Scientifique et Medicale Joseph Fourier of Grenoble, France. In 1972, at the same University in Grenoble, Patrick Cousot worked on context-free grammar parsing based on a pre-processing of the grammar by static analysis and transformation before defining a top-down parser, as described by his very first scientific paper [44], written and presented at the French AFCET conference, held in November 1972, while he was an undergraduate student in Grenoble.

In 1973 Patrick Cousot started to elaborate the vision that a formal definition of a programming language semantics could be used to derive a correct implementation of the language. In particular, Cousot studied some automatic optimizations of language definitions, that included the elimination of useless transformations, and a program pre-evaluation, which can be viewed as a forebear of the modern "partial evaluation" techniques. This work was done for his *Thèse de Docteur Ingénieur en Informatique*, submitted in 1973 to the Université Scientifique et Medicale Joseph Fourier of Grenoble and defended, after his military service [45], on December 14th, 1974 [46].

In 1974 Radhia Rezig showed her sketches of

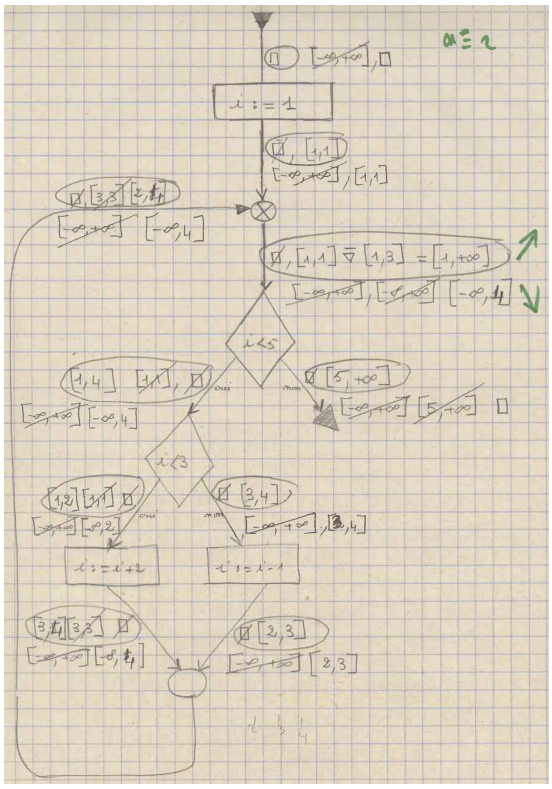

Figure 1. 1974 handwritten notes of Radhia Rezig on forward iteration from $\square = \bot$ (forward least fixed point) versus backward iteration from $[-\infty, +\infty]$ (backward greatest fixed point) on a flow diagram.

examples of interval analysis of numerical variables occurring in programs to Patrick Cousot. Intervals express properties of programs concerning the lower and upper bounds of numerical variables, and this class of properties is particularly useful for statically detecting run-time integer overflows. Patrick was critical on performing this analysis backwards from end program points towards the start point(s), and using an initial interval $x \in [-\infty, +\infty]$ that carries no information on a program variable x. Patrick claimed that a forward analysis would have been more effective and was also skeptical on achieving termination for programs with loops. Then, Radhia bounced back to him with the idea of extrapolating bounds of intervals to $\pm\infty$ when computing the forward analysis of the program [47]. This was an early discovery of the key notion of *widening*, which further extrapolates fixed point solutions when the abstract domain contains infinite ascending chains of larger and larger objects, as in the case of intervals $[1,1] \subseteq [1,3] \subseteq [1,5] \subseteq \cdots [1,i] \subseteq [1,i+2] \subseteq \cdots$. The handwritten notes by Radhia displayed in Fig. 1 provide a sketch of a forward vs. backward analysis using the interval domain for the following program:

$i := 1;$
while $i < 5$ **do**
  if $i < 3$ **then** $i := i + 2$ **else** $i := i - 1$ **endif**
endwhile

where we can already recognize the present-day notation for widening, denoted by $\bar{\nabla}$, that is able to extrapolate unstable bounds to stability, i.e. to $+\infty$, in

$$[1,1]\bar{\nabla}[1,3] = [1,+\infty].$$

The intuition here was that since the analysis infers that the upper (i.e., right) bound of the variable i is increasing, then, to avoid to follow the infinite chain of intervals

$$[1,1] \subseteq [1,3] \subseteq [1,5] \subseteq \cdots$$

that ineluctably leads to a nonterminating analysis, a widening operation should infer the "correct worst case" for the upper bound of i already at the first increment from $i \in [1,1]$ to $i \in [1,3]$ through the assignment $i := i+2$, i.e., the correct interval of variation for i should be widened to $[1,1]\bar{\nabla}[1,3] = [1,+\infty]$. This was an easy but nevertheless acute and influential observation, because it opened up the chance of performing a terminating static program analysis on *infinite* domains of program properties, which was not possible at that time for the aforementioned simple analyses used for optimizing program compilation.

Under the French IRIA-SESORI contract No. 75-035, Patrick and Radhia Cousot wrote in French a first research internal report of the "Grenoble Laboratoire IMAG associè au CNRS no. 7", dated back September 23th 1975 [48] with title "*Vérification statique de la cohérence dynamique des programmes*". This document described in detail the original idea of abstract interpretation and already included an explicit full specification of an abstract interpreter with widening called "*interpétation abstraite*". As illustrative example, Cousot and Cousot considered

an interval analysis of the following program in PASCAL:

```
var j : integer;
j := 1;
while j ≤ 10 do  j := j + 1;
```

and showed how their "interpétation abstraite" procedure was able to infer the loop invariant $j \in [1, +\infty]$ by relying on the widening step $[1,1]\bar{\nabla}[1,2] = [1,+\infty]$, and the invariant $j \in [11, +\infty]$ at the exit point obtained by logically combining the inferred loop invariant $j \in [1, +\infty]$ with the exit condition. All the key ideas of abstract interpretation were clearly elucidated already in this research report [48], describing all the concepts outlined above added with the join operation between abstract values at the junction nodes of the program graph for branching and loop constructs. With all its own rights, historically speaking, this was the very first publication introducing the abstract interpretation methodology with all its foundational constituents, so we can reasonably argue that abstract interpretation was born on September 23th, 1975 in Grenoble, France. This publication was soon followed in November 1975 by an internal research report written in English [49] devoted to a static type analysis of program variables designed as an abstract program evaluation that computes, by successive approximations, an abstract context at every program point.

The first formal and widely disseminated research publication authored by Patrick and Radhia Cousot was "*Static determination of dynamic properties of programs*", appeared in the Proceedings of the 2nd International Symposium on Programming, held in Paris, April 13th-15th, 1976 [50]. This was a follow-up written in English of the aforementioned first research report [48] and its introduction clearly identified the goals of abstract interpretation: "*The static analysis of programs we do consists of an abstract evaluation of these programs, similar to those used by Peter Naur for verifying the type of expressions in Algol 60, by Michel Sintzoff for verifying that a module corresponds to its logical specification, by Gary Kildall for global program optimisation, by Ben Wegbreit for extracting properties of programs, by Michael Karr for finding affine relationships among variables of a program, by Jacob T. Schwartz for automatic data structure choice in SETL, etc.*". This article contained a consolidation of the ideas of the first research report [48] and presented a so-called "*abstract interpretor*" for the static analysis of imperative programs with widenings. The correctness of the program analysis was explicitly stated with respect to a collecting program semantics — while for the proof of correctness they cited the first research report [48] — and the applications were interval and pointer analyses.

1977-79: ACM POPL Conferences

For the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1977, Patrick and Radhia submitted on August 12, 1976 the hard copies of a four-hands written manuscript of about 100 pages [45]. The suggestion to submit their work to the ACM POPL conference came from Stephen Warshall [45], already well-known for his transitive closure algorithm and visiting Grenoble in 1976. The POPL 1977 manuscript contained all the building blocks of the abstract interpretation method in their full maturity: concrete and abstract semantic transformers; backward and forward reachability; iterative fixed point computation; chaotic and asynchronous fixed point iterations; fixed point abstraction under commutativity with abstraction; widening and narrowing; conjugate and inversion of least with greatest fixed points [32]. The paper was eventually accepted (25 accepted papers out of 107 submissions) and presented in Los Angeles, CA, 17-19th January 1977, and it is still nowadays one of the most cited papers in PL and, according to data from the ACM Digital Library, the most cited POPL paper ever. It is worth remarking that this landmark paper was never published in an archival journal.

We claim that the abstract interpretation method relies on precisely *four key principles* that maturated from the first two papers [48], [50] discussed above and have been clearly elucidated in the POPL 1977 paper [32]:

1) *Program analysis means approximating program semantics*, as illustrated by the abstract model of program semantics in Section 4 that ignores the sequencing of control flow;

2) *Approximation is encoded by abstract domains*, that formalized the approximation process of program semantics and were defined and studied in Sections 5-6-7;
3) *Abstract interpreters are compositionally designed*, as described by the abstract evaluation of programs in Section 8 that composes elementary abstract interpreters for atomic commands such as variable assignments and Boolean tests;
4) *Static program analysis boils down to executing the abstract interpreter*, and, as shown in Section 9, this corresponds to approximating fixed point solutions of recursive equations in abstract domains by exploiting the widening operations discussed above.

In this perspective, abstract interpretation can be seen from its very beginning as a *constructive method* [28] for assisting (and to help mechanizing) the proof of correctness of programs.

In those years 1977-1978, Patrick and Radhia Cousot also published some companion articles describing more technical results on the mathematical structures used in the POPL 1977 paper: a constructive version of Tarski's fixed point theorem [51], published in 1979 on the Pacific Journal of Mathematics (and submitted in December 1977), the same journal where Alfred Tarski published his celebrated fixed point theorem; a characterization of topological closure operators [52], published in 1979 on Portugaliæ Mathematica, the same journal where José Morgado published most of his papers on closure operators. These works represented the transposition of the ideas of abstract interpretation into mathematical lattice theory, and produced new results in the fields of fixed points and closure operators. Moreover, on March 21st 1978, Patrick Cousot defended his PhD thesis [11] (the French academic degree was "Docteur d'Etat ès Sciences Mathematiques") at the Université Scientifique et Médicale de Grenoble, that included an expanded version of the results in the POPL 1977 paper.

Cousot and Cousot's POPL 1977 paper has been followed by two fundamental papers at the end of the 1970s, appeared respectively in POPL 1978 and POPL 1979. In POPL 1978 [53], abstract interpretation was applied to the convex polyhedra abstract domain. This allows to automatically infer complex linear inequalities between program variables such as $ax + by \leq k$, namely, the most helpful relations for finding program bugs. Since then, convex polyhedra probably achieved the status of chief abstract domain used in static program analysis, which has been implemented in several libraries and APIs. In POPL 1979 [54] abstract interpretation is viewed from a more theoretical perspective: abstract domains can be specified by closure operators; existence of the best correct approximations of semantic transformers; notion of precision (also known as completeness) in abstract interpretation; how abstractions can be built and enriched by combining simpler abstractions.

THE INFLUENCE IN PROGRAMMING LANGUAGES

1980-90s

Abstract interpretation had an initial difficulty in spreading within the PL community, in particular in the decades 1980-90s. We single out some reasons behind this trouble. In imperative programming, the prevalence of the early frameworks for data-flow static analysis (*à la* Kildall [55] and Kam-Ullman [31]) firmly oriented to be used in compile-time program optimizations, limited the use of abstract interpretation in compilers. The structure of the intermediate code commonly used in compilers in those years and the efficiency of already known and used algorithms based on data-flow analysis, constrained the use of abstract interpretation to specific optimization problems in program parallelization [56]. In concurrency, the scientific debate was mainly focussed on finding the "right" model and calculus to be used for modeling concurrent processes, and its static analysis saw a dominance of type inference systems coming from the functional programming paradigm, while synchronization properties were considered too hard to be a subject of some static analysis. In functional programming, the dominance of type-based verification *à la* Milner [57], [58] with his well-known claim that "*well-typed programs cannot go wrong*", and binding-time analysis *à la* Jones-Muchnick [59], reduced program verification to type-checking. Strictness analysis —

i.e., statically determining whether a functional program preserves undefinedness or not — was basically the only genuinely true example of abstract interpretation of functional programs, as pioneered by Alan Mycroft in 1980 [60].

On the other hand, abstract interpretation was quite successful in the logic programming paradigm, in particular for Prolog, and had a large community of researchers and practitioners in the 1980s, boosted by the Association for Logic Programming that was founded in 1986. In 1986-87 some early articles of abstract interpretation of logic programs include the works by Chris Mellish [61], Harald Søndergaard [62] and Maurice Bruynooghe et al. [63]. We isolate some reasons for the success of abstract interpretation in the 1980s for analyzing logic programs:

1) Logic programs had since the beginning a clean and simple fixed point semantics [64];
2) *À la* Milner type systems for logic programs were difficult to design and assess due to the nature of logical relations, where an unspecified input/output structure in predicates appears disruptive if compared to the clean and elegant notion of type in functional programming;
3) The synchronization mechanism was simple, just based on unification of terms, and no type system was used;
4) The complicated control posed heavy problems in optimization and the need of AND/OR parallelism strongly called for new static program analyses beyond types.

Abstract interpretation was therefore for logic programming the right theory at the right time, providing effective solutions for optimization [65] and parallelism [66]. For about a decade in the 1990s, the largest amount of papers published in abstract interpretation were indeed in the field of logic programming.

The XXI Century

The true widespread success of abstract interpretation in PL arrived at the beginning of the XXI century. We identify two main reasons for this seeming rediscovery:

1) The inclination and ability of scientists and researchers to design programming language semantics and compilers from a mathematical description of their behaviour, even for complex programming constructs and features. This ability is rooted in the early work on formal program semantics by Peter David Mosses [67] in the mid 1970s, but reached a full maturity just in the mid-late 1990s. These efforts gave to an arbitrary programming language, even including low-level languages [68], the full mathematical structure to make abstract interpretation naturally applicable, thus going beyond the simplicity of logic programming.
2) The successful industrialization of abstract interpretation tools, that proved that this theory can lead to effective and practical applications, in particular of large industrial size. The need for an industrial use-case was felt already in the 1990s with the emergence of model checking. The model checking technique is born in 1983 [69], but it became popular — beyond the initial verification of digital circuits — in the 1990s [70], where it has been viewed as an actual verification method alternative to abstract interpretation for proving program correctness. Abstract model checking [71] is an example of the scientific debate of those years on *model checking vs. abstract interpretation* [72], [73]. This discussion was a quest for the community of abstract interpretation of an industrial use-case characterized by a complexity and size which were not affordable by model checking methods and, at the same time, of large practical impact.

Having correctness proofs for real-life programs, beyond small code fragments and tiny examples, was always considered crucial and shaped the debate between enthusiasts and skeptics of formal methods from the 1970s to the end of the XX century [28]. The complexity of a proof of correctness, often way beyond the length of the code which is supposed to prove correct, is at the heart of this debate, that raised the belief that, although fundamental for the progress of computer science as a discipline, formal program verification was essentially unpractical [6], [8]. Andrew Stuart Tanenbaum [74] wrote: "*Imagine*

[...] a fully automated air traffic control system whose software was all shown to be correct by formal proof, but never tested even once. Would you be willing to be a passenger on the first actual flight using the system?". Having a killer application successfully working on real software systems, in particular in safety-critical situations, was indeed the holy grail of formal methods until the end of the XX century. Abstract interpretation broke in this debate, providing effective solutions for proving the correctness of large programs employed in safety- and life-critical systems.

Historically, the earliest industrial application of abstract interpretation was the interval analysis implemented in the AdaWorld compiler for IBM PC 80286 by J.D. Ichbiah and his French Alsys SA company team in 1980-87 [75]. In the 1990s, Alain Deutsch was a PhD student of Christian Quiennec at the former Université Paris VI, advised by Patrick Cousot. In 1992, Deutsch's PhD thesis introduced abstract interpretation for dynamic data structures with pointers [76]. In 1996, after the crash of the Ariane 5 flight 501 due to a software failure [77], Deutsch was a member of the French INRIA team in charge of understanding this code bug, and used a prototype of static analyzer, called IABC, for Ada source code based on abstract interpretation, developed by Deutsch after his PhD thesis [78]. This tool demonstrated the effectiveness of static program analysis on industrial size and safety-critical applications. After this success, French INRIA, CNES and Aerospatiale pushed towards the creation of a company in charge of the industrialization of this prototype static analyzer IABC. Eventually, in 1999 Alain Deutsch and Daniel Pilaud founded PolySpace Technologies in Grenoble, which had an impetuous and continuous growth in 1999-2006 and in 2007 was eventually acquired by The MathWorks software corporation.

A key step forward was the development of Astrée (*Analyseur Statique de logiciels Temps-RÉel Embarqués*, namely, real-time embedded software static analyzer) representing an authentic progress in the industrialization of abstract interpretation. Started from scratch in November 2001 as an initiative of Patrick and Radhia Cousot at the Laboratoire d'Informatique of the École Normale Supérieure in Paris, the Astrée project was initially supported by the French CNRS and INRIA [79]. The first development of Astrée took two years of work for a team of about ten people including PhD students and software engineers. The main applications of Astrée appeared two years later with Airbus Industries [80] and since then Astrée achieved unprecedented results on the static analysis of C programs, notably for: array index out of bounds, integer divisions by zero, invalid pointer dereferences, arithmetic overflows and wrap-arounds, floating point overflows and invalid operations, IEEE floating values Inf and NaN, user-defined assertions, unreachable code, uninitialised variables, elimination of false alarms by local refinements. In December 2009, Astrée was acquired and soon later commercialized by the German company AbsInt Angewandte Informatik, extending the tool to dynamic memory allocation, recursion and C++ [81]. From this success story, a number of program analysis and verification systems have been developed, all having abstract interpretation in their technological heart. These include the most recent and widely known Infer and Zoncolan tools, developed by Facebook in the 2015-17 [82]: Infer detects memory safety and concurrency bugs in Java/C/C++/Objective-C code; Zoncolan finds security and privacy violations in Facebook's Hack codebase. Both Infer and Zoncolan are fully based on abstract interpretation and are routinely used by Facebook software engineers on millions of lines of C++ and Hack code.

CONCLUSION

We traced the origins and the evolution of the abstract interpretation method in computer science, from the historical computational and mathematical background where the earliest research documents of the 1970s introduced its embryonic ideas, passing through an analysis of the difficulties of the 1980-90s in spreading within the programming languages research community, and culminating to its expansion in the software industry of the 2000s. The way abstract interpretation emerged and spread in computer science and industry is one more example of how beneficial is to find the right abstraction level in a formal description of computations. Stop and go alternated in correspondence with, respectively, the inability and ability to describe the semantics of programs in a simple and con-

cise way. In this perspective, the advance of abstract interpretation has grown in parallel with the success and practice of formally specifying programming languages and their tools. We are convinced that this condensed history of abstract interpretation provides yet another example that scientific successes of groundbreaking research ideas pave the way for impactful innovation in industry and, ultimately, in human progress.

ACKNOWLEDGMENTS

This article follows a talk [83] given at the workshop on the History of Formal Methods (HFM 2019), held in Porto, Portugal, on 11th October 2019, as part of the 2019 Formal Method World Congress. We are much indebted to and have been inspired from Patrick Cousot's talk "Abstract Interpretation: 40 years back + some years ahead" at the *Next 40 years of Abstract Interpretation (N40AI) Workshop* held in Paris on January 21st, 2017, as part of the POPL 2017 week. We thank the anonymous reviewers of this paper for their helpful comments and suggestions.

We feel grateful to Patrick and Radhia Cousot for having invented abstract interpretation. This work is dedicated to the memory of Radhia Cousot.

REFERENCES

1. Charles Babbage. On a method of expressing by signs the action of machinery. *Philosophical Transactions of the Royal Society of London*, 116(1/3):250–265, 1826.
2. Charles Babbage. *Passages from the Life of a Philosopher*. Longman, London, 1864.
3. Hermann H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. *Report of U.S. Ord. Dept., Institute for Advanced Study, Princeton*, 1947.
4. Simone Martini. The standard model for programming languages: The birth of a mathematical theory of computation. In *Recent Developments in the Design and Implementation of Programming Languages, Gabbrielli's Festschrift, Bologna, Italy*, volume 86 of *OASIcs*, pages 8:1–8:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
5. Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines, University of Cambridge Mathematical Laboratory, Cambridge, UK,* http://www.turingarchive.org/browse.php/b/8, pages 67–69, 1949.
6. James H. Fetzer. Philosophical aspects of program verification. *Minds Mach.*, 1(2):197–216, 1991.
7. David Nofre, Mark Priestley, and Gerard Alberts. When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. *Technology and Culture*, 55(1):40–75, 2014.
8. Matti Tedre. *The science of computing – Shaping a discipline*. CRC Press Taylor & Francis Group, 2015.
9. John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
10. John McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, volume 35 of *Studies in Logic and the Foundations of Mathematics (This paper is a corrected version of the paper of the same title given at the Western Joint Computer Conference, May 1961.)*, pages 33 – 70. Elsevier, 1963.
11. P. Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes*. PhD thesis, Université Scientifique et Médicale de Grenoble, March 21st 1978.
12. Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.
13. Peter Naur. Checking of operand types in Algol compilers. *BIT*, 5:151–163, 1965.
14. Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
16. R. M. Burstall. Proving Properties of Programs by Structural Induction. *The Computer Journal*, 12(1):41–48, 02 1969.
17. D. S. Scott. An outline of mathematical theory of computation. In *Proc. of the Fourth Annual Princeton Conference on Information Science and System*, pages 169–176, 1970.
18. Dana S. Scott and C.S. Strachey. Towards a mathematical semantics for computer languages. *Proceedings of the Symposium on Computers and Automata*, 21, 01 1971.
19. Zohar Manna, Stephen Nes, and Jean Vuillemin. Inductive methods for proving properties of programs. *Commun. ACM*, 16(8):491–502, 1973.

20. E. W. Dijkstra. *A discipline of programming*. Series in automatic computation. Prentice-Hall, 1976.
21. Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. Program analysis is harder than verification: A computability perspective. In *Proceedings of the 30th International Conference on Computer Aided Verification CAV 2018*, volume 10982 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2018.
22. Andrea Asperti, Herman Geuvers, and Raja Natarajan. Social processes, program verification and all that. *Math. Struct. Comput. Sci.*, 19(5):877–896, 2009.
23. Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.
24. Frances E. Allen. Program optimization. *Annual Review of Automatic Programming*, 5:239–307, 1969.
25. Frances E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.
26. John Cocke and Raymond E. Miller. Some analysis techniques for optimizing computer programs. In *Proc. 2nd Hawaii Int. Conf. on Systems Sciences*, pages 143–146, 1969.
27. John Cocke and Raymond E. Miller. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, 1970.
28. Mark Priestley. *A Science of Operations – Machines, Logic and the Invention of Programming*. Springer, 2011.
29. A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.
30. A. V. Aho and J. D. Ullman. Optimization of straight line code. *SIAM J. of Computing*, 1(1):1–19, 1972.
31. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
32. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
33. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. Technical Report 169, C.S. Lab., Princeton University, 1975.
34. A. Monteiro and H. Ribeiro. L'opération de fermeture et ses invariants dans les systèmes partiellement ordonnés. *Portugaliæ Mathematica*, 3(3):171–184, 1942.
35. O. Ore. Some studies on closure relations. *Duke Mathematical Journal*, 10(4):761–785, 1943.
36. O. Ore. Combinations of closure relations. *Ann. Math.*, 44(3):514–533, 1943.
37. M. Ward. The closures operators of a lattice. *Ann. of Math.*, 43(2):191–196, 1942.
38. B. Knaster and A. Tarski. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.
39. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
40. Jaime Carvalho e Silva. *José Morgado (1921-2003)*, 2003.
41. J. Morgado. Some results on the closure operators of partially ordered sets. *Portug. Math.*, 19(2):101–139, 1960.
42. O. Ore. Galois connexions. *Transactions of the American Mathematical Society*, 55(3):493–513, 1944.
43. R. Rezig. *Application de la méthode de précédence totale à l'analyse d'Algol 68 (in French)*. Master thesis, Université Joseph Fourier, Grenoble, France, September 1972.
44. P. Cousot. Un analyseur syntaxique pour grammaires hors-contexte ascendant sélectif et général. In *Les techniques de l'informatique*, Congrès AFCET 72, Brochure 1, pages 106–130, Grenoble, France, November 1972.
45. Patrick Cousot. Personal communication, 2017.
46. P. Cousot. *Définition interprétative et implantation de languages de programmation (in French)*. Thèse de docteur ingénieur en informatique, Université Joseph Fourier, Grenoble, France, 14 December 1974.
47. Patrick Cousot. Personal communication, 2019.
48. P. Cousot and R. Cousot. Vérification statique de la cohérence dynamique des programmes. Res. rep. Rapport du contrat IRIA SESORI No 75-035, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, 23 Sep. 1975. 125 pages.
49. 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 pages.
50. 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.
51. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.
52. 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.
53. P. Cousot and N. Halbwachs. Automatic discovery

of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, POPL 1978*, pages 84–96. ACM Press, 1978.

54. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*), pages 269–282. ACM Press, 1979.
55. G.A. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages (POPL '73)*, pages 194–206. ACM Press, 1973.
56. Zahira Ammarguellat and Williams Ludwell Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1990)*, pages 283–295. ACM, 1990.
57. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
58. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1982*, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.
59. Neil D. Jones and Steven S. Muchnick. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages, POPL 1976*, pages 77–94. ACM Press, 1976.
60. Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*, page 269–281, Berlin, Heidelberg, 1980. Springer-Verlag.
61. Chris Mellish. Abstract interpretation of prolog programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming, ICLP 1986*, volume 225 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 1986.
62. Harald Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of the European Symposium on Programming, ESOP 1986*, volume 213 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 1986.
63. Maurice Bruynooghe, Gerda Janssens, Alain Callebaut, and Bart Demoen. Abstract interpretation: Towards the global optimization of prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming, San Francisco, California, USA, August 31 - September 4, 1987*, pages 192–204. IEEE-CS, 1987.
64. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, October 1976.
65. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Program. Lang. Syst.*, 2(1-4):181–196, 1993.
66. Kalyan Muthukumar and Manuel V. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *J. Log. Program.*, 13(2&3):315–347, 1992.
67. Peter David Mosses. *Mathematical semantics and compiler generation*. PhD thesis, University of Oxford, UK, 1975.
68. Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 379–391. ACM, 2009.
69. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 117–126, New York, NY, USA, 1983. Association for Computing Machinery.
70. Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV 1996*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–427. Springer, 1996.
71. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1992, pages 343–354, New York, NY, USA, 1992. Association for Computing Machinery.
72. Annalisa Bossi, Dennis Dams, Gilberto Filé, and Elena Marchiori. Verification, model checking and abstract interpretation (workshop overview). In *Proceedings of the 1997 International Symposium on Logic Programming*, pages 421–422. MIT Press, 1997.
73. David A. Schmidt. Data flow analysis is model checking

of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998*, pages 38–48. ACM, 1998.

74. Andrew S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *ACM SIGPLAN Notices*, 11(5):64–68, 1976.
75. Troy Pearse, Steve Furgason, Youfeng Wu, and Naresh Gupta. Software reviews: Wonderful world of Alsys Ada. *IEEE Software*, 4(04):90–91, jul 1987.
76. A. Deutsch. *Models Operationnels de Langage de Programmation et Representations de Relations Sur des Langages Rationnels avec Application a la Determination Statique de Proprietes de Partages Dynamiques de Donnees*. PhD thesis, Universitè Paris VI, France, 1992.
77. Gérard Le Lann. An analysis of the Ariane 5 Flight 501 failure – A system engineering perspective. In *Proceedings of the 1997 International Conference on Engineering of Computer-Based Systems*, ECBS'97, page 339–346, USA, 1997. IEEE Computer Society.
78. Ph. Lacan, J. N. Monfort, L. V. Q. Ribal, A. Deutsch, and G. Gonthier. ARIANE 5 - The Software Reliability Verification Process. In B. Kaldeich-Schürmann, editor, *Proceedings of DASIA 98 - Data Systems in Aerospace*, volume 422 of *ESA Special Publication*, page 201, July 1998.
79. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astreé analyzer. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.
80. Laurent Mauborgne. Astrée: verification of absence of run-time error. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions*, volume 156 of *IFIP*, pages 385–392. Kluwer/Springer, 2004.
81. Daniel Kästner, Christian Ferdinand, Stephan Wilhelm, Stefana Nenova, Olha Honcharova, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival, and Élodie-Jane Sims. Astrée: Nachweis der abwesenheit von laufzeit (in German). *Softwaretechnik-Trends*, 29(3), 2009.
82. Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.
83. Roberto Giacobazzi and Francesco Ranzato. History of abstract interpretation. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal*, volume 12233 of *Lecture Notes in Computer Science*, page 13. Springer, 2019.

Roberto Giacobazzi is currently professor at the Department of Computer Science, University of Verona, Italy. His research interests are centered around abstract interpretation since 1989. Contact him at roberto.giacobazzi@univr.it.

Francesco Ranzato is currently professor at the Department of Mathematics "Tullio Levi-Civita", University of Padova, Italy. He works on abstract interpretation principles and applications since 1994. Contact him at francesco.ranzato@unipd.it.