Abstract Interpretation: Past, Present and Future

Patrick Cousot
CIMS *, NYU, USA
pcousot@cims.nyu.edu

Radhia Cousot
CNRS Emeritus, ENS **, France
rcousot@ens.fr

Abstract

Abstract interpretation is a theory of abstraction and constructive approximation of the mathematical structures used in the formal description of complex or infinite systems and the inference or verification of their combinatorial or undecidable properties. Developed in the late seventies, it has been since then used, implicitly or explicitly, to many aspects of computer science (such as static analysis and verification, contract inference, type inference, termination inference, model-checking, abstraction/refinement, program transformation (including watermarking, obfuscation, *etc*), combination of decision procedures, security, malware detection, database queries, *etc*) and more recently, to system biology and SAT/SMT solvers. Production-quality verification tools based on abstract interpretation are available and used in the advanced software, hardware, transportation, communication, and medical industries.

The talk will consist in an introduction to the basic notions of abstract interpretation and the induced methodology for the systematic development of sound abstract interpretation-based tools. Examples of abstractions will be provided, from semantics to typing, grammars to safety, reachability to potential/definite termination, numerical to protein-protein abstractions, as well as applications (including those in industrial use) to software, hardware and system biology.

This paper is a general discussion of abstract interpretation, with selected publications, which unfortunately are far from exhaustive both in the considered themes and the corresponding references.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]; D.3.1 [*Formal Definitions and Theory*]; F.3.1 [*Specifying and Verifying and Reasoning about Programs*].

General Terms Algorithms, Languages, Reliability, Security, Theory, Verification.

Keywords Abstract interpretation, Semantics, Proof, Verification, Static Analysis.

1. Abstraction

No reasoning on complex systems, including computer systems, can be done without abstracting the behavior, *i.e.* the semantics, of the system. Because reasoning on a system involves determining or proving its properties, the central concept is the abstraction of properties of the system, starting from the strongest one, as specified by the system semantics (and called the collecting semantics[1]). This is the purpose of abstract interpretation (where "interpretation" stands both for "meaning" and "execution"). A few gentle introductions to abstract interpretation [44, 79] can be consulted for a first approach, including some publicly available on the web (*e.g.* `web.mit.edu/16.399/www/`).

2. Scope

Abstract interpretation comprehends undecidable problems (hence is also applicable to decidable but complex ones). This implies that any tool (prover, checker, analyzer) designed by abstract interpretation will fail on infinitely many counter-examples. For example a finiteness or decidability hypothesis will only be applicable to a very restricted class of programs with finite behavior, hence will fail on infinitely many other ones. This is inherent to undecidable problems hence inescapable. By failure we understand being unsound/incorrect, non-terminating, using a human oracle to assist the computer, etc. Although abstract interpretation also applies to these cases[2], it is usually used for sound, terminating, and fully automatic program analysis/verification, including the inference of sound inductive arguments (like invariants) to deal with infinite recurrences for unbounded/non-terminating executions, which makes the problem particularly difficult, with a very high complexity.

3. Static analysis

The origin of abstract interpretation is in static program analysis [50, 51] where reachable states are abstracted by local interval numerical invariants understood as a generalization of type inference [49, 54]. The abstraction from the collecting semantics was formalized by a Galois insertion and convergence acceleration of the iterates by widening, later improved by narrowing. The main innovations at the time were to consider infinite non-Noetherian abstractions of infinite systems and to prove rigorously the correctness of the static analysis with respect to a formal semantics (see more details in footnote [6]).

* Work supported in part by the CMACS NSF award 0926166.

** Work supported in part by the European ARTEMIS project MBAT (grant agreement No. 269335).

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/2603088.2603165

[1] The collecting (or static in [52]) semantics is the strongest property of the standard semantics.

[2] For example, some commercial products do consider only two iterations in loops without widening, which is an under-approximation of an over-approximating abstraction of program executions which can be formalized by abstract interpretation theory. The theory also proves beyond doubt that the result cannot be claimed to be a sound over-approximation of the program behavior, a conclusion which is not always stated clearly enough for practitioners to have a precise understanding of the scope of commercial static analyzers.

4. Acceleration of fixpoint iteration

The next step was to consider arbitrary fixpoints in ordered structures, order duality for approximations from above and below, ascending and descending iterations, as well as fixpoint iteration with convergence acceleration from above and below with over- and under-approximation using the following extrapolation operators [39, Ch. 4], [52]

Extrapolation operators	iteration from below	iteration from above
over-approximation	widening	narrowing
under-approximation	dual narrowing	dual widening

Chaotic [53] and asynchronous iterations [38, 39] (including at higher-order [55, 149]) paved the way for demand-driven [203] and parallel implementations of static analyzers [86]. Moreover this approach includes alternative, seemingly different, presentations of static analysis (like constraint solving, deductive rules-based (type) systems, *etc* which nicely generalize in the abstract [68]). The extrapolation operators, which abstract the inference of an inductive hypothesis in iteration are the more powerful, and difficult to design, aspects of abstract interpretation. The design of powerful extrapolation operators pertinent to the degree of approximation of an abstraction is still an art which remains to be much better understood, formalized, and improved. Not enough work has be devoted to the systematic design of extrapolation operators and the fact that they cannot be increasing[3] (*a.o.* [8, 21, 36, 63, 131, 169]). Similarly non-iterative methods are just appearing for specific abstract domains and programs such as exact widenings as in "abstract acceleration" [145] and "policy iteration" [27, 117, 160, 212][4].

5. Semantics

Transition systems [39, 56] and recursive procedures [39, 55] (see also [67]) departed from the initial consideration of flowcharts understood as an intermediate program representation [49–51]. Backward and forward analyzes/proof methods become symmetric by inversion of the transition system [39, 40, Ch. 3], [56] so only one of them needs to be studied formally. Similarly for forward/backward transformers and fixpoint duality (see the figure on [52, page 241] prefiguring the μ-calculus). Which semantics is to be preferred, small-step operational [39, 56], denotational [148, 180], big-step natural [205], *etc* became subject to controversies [206]. The solution was ultimately to relate all of them by abstract interpretation.

Another alternative is to design a meta-language to define semantics and to design a general purpose abstract interpreter taking semantics as a parameter, either that of the program to be analyzed or that of a programming language [42, 152, 168].

6. Hierarchies of semantics

The choice of a standard (or collecting) semantics may look fundamental in the design of abstract interpretations. On one hand, the most concrete/collecting semantics dictates an initial specific abstraction specifying the most precise expressible properties. If the collecting semantics is too abstract, it has a limited expressive power which also limits the scope of the static analysis/verification methods that can be formally derived by abstract interpretation of this collecting semantics. For example, with the big-step natural semantics [205], it is possible to abstract into a partial correctness axiomatic semantics but not to reason on (non-)termination. On the other hand, if the collecting semantics is too precise, then further abstractions are more complex to express (in that they result in more complex formulae in the Galois connection calculus [82]).

Nevertheless, the importance of the choice of a universal standard/collecting semantics should be greatly relativized. The reason is that all known semantics are completely defined by an abstraction of the concrete operational trace semantics and form a hierarchy of semantics where all known semantics are recovered by abstractions [64] (some being Galois isomorphisms like between natural, angelic, or respectively demonic denotational, relational, predicate transformer, and axiomatic semantics). An abstraction based on any semantics in the hierarchy will be comparable to any other abstraction based on a different semantics in the hierarchy by considering the greatest lower bound in the hierarchy, which is the most abstract semantics more precise than these two semantics.

This hierarchical semantic approach is valid for transition systems [46] (including transfinite behaviors [120]), grammars [80], resolution-based languages like Prolog [61, 88], higher-order functional languages [67, 78], *etc*.

7. Combinations of semantics

Sometimes, the collecting semantics cannot be expressed as a single fixpoint but as a combination of fixpoints. An example in [39] is the intersection of forward and backward reachability. In that case, the best abstraction is not the trivial intersection of abstract forward and abstract backward reachability but an iteration of forward and backward abstract reachability analyzes [39, 61]. This generalizes to more complex combinations of fixpoints as found *e.g.* in temporal logic [94, 158, 159]. This point is not well understood when reasoning on (abstract) models rather than on properties and their different levels of abstraction.

8. Proof methods, verification and inference

Proof methods all consider an abstraction of the program semantics and so can be explained by abstract interpretation using basic induction principles such as fixpoint/post-fixpoint induction.

The essential difference between computer-aided verification and static analysis is that in verification an inductive argument is assumed to be provided by the end-user (*e.g.* loop invariants that must be expressed in some logical language) whereas in static analysis, this same inductive argument must be automatically inferred (and expressed in some combination of abstract domains). Notice that incompleteness in proof methods [41] follows from the abstraction of the invariant expressible in the collecting semantics which may be inexpressible in the abstract domain defined by the logical language [91]. Inference is of course more difficult than verification, for which the only problem is to prove implications, not to guess what to prove. Nevertheless, verification has rapidly difficulties to scale up when the size of the information to be provided by the end-user is exponential in the size of the program [219].

9. Over- and under-approximation

The theory of abstract interpretation is usually only presented for over-approximations. An over-approximation means that all executions (and maybe a few spurious ones) have to be considered. An example is automatic reachability analysis with invariant inference [52]. Right at the origin of the theory, under-approximation was considered order or complement dual so there is no need to rephrase

[3] Intuitively monotone/increasing operators would not be able to "jump" over fixpoints.

[4] Before [50], we tried to solve numerical interval equations by dichotomy, but rapidly understood that the method would not be generalizable *e.g.* to symbolic properties.

the dual theory [52][5]. An under-approximation means that some executions (and no spurious ones) have to be considered. However it is significantly more difficult in practice. An example is the automatic inference of necessary preconditions [83] (whereas an over-approximation would yield sufficient preconditions).

10. Abstract domains

In abstract interpretation, properties are sets of objects with that property [56]. For example, the collecting semantics is the singleton whose element is the standard semantics. So a concrete program property is the set of its possible semantics. If executions are traces, non-deterministic semantics are sets of traces, so properties are sets of sets of traces (later called "hyper-properties"). So concrete properties can expressed as sets of sets of objects formalizing the effect of an execution (*e.g.* traces). This point of view unifies various versions of abstract interpretation which were first thought to be different because they abstract different kind of semantics (operational, denotational, predicate transformers, axiomatic).

Abstract domains encode a subset of the possible properties of a program/system/*etc*. They are (pre-)ordered by a relation which encodes logical implication. Because of iteration/recursion, they must also encode some notion of limit (of computations, including unbounded or infinite non-terminating ones). Such limits are often expressed as least upper bounds of increasing chains for a computational order, which is often, but not always, the same as the logical order. Strictness analysis [180] was the first example were the logical and computational orders differ, but iterative fixpoint computation and acceleration results extend naturally by considering different ways of passing to the limit [66, 67].

It is quite remarkable that abstract domains are completely determined by the concrete properties and a Galois connection. This leads to the idea of a Galois connection calculus [82], which allows to specify the essence of static analysis/verification methods by defining the considered semantic domain and abstraction with a single formula of the calculus. Another consequence is that abstract domains are complete lattices since Galois connections preserve existing least upper bounds.

11. Refinement of abstract domains

The refinement of an abstract domain consists in adding the minimal new abstract properties to the domain so as to be able to express exactly in the refined domain some of the concrete properties which had to be strictly approximated in the original domain.

The first example was the disjunctive completion [56, Sect. 10.2], [116] of an abstract domain A: this is a refinement $\wp(A)$ of this abstract domain A adding to the disjunctive completion $\wp(A)$ all the concrete disjunctions of elements of A originally missing in A (hence which had to be strictly over-approximated in A). It is the most abstract domain that is exact on disjunctions of abstract properties in A. It was used in [56, Sect. 10.2] to prove that the merge-over-all paths data flow analysis can be expressed in fixpoint equational form [6]. Similarly, the complementation [34] adds complements that are missing in the abstract.

The transformer/fixpoint completion of an abstract domain is a program-dependent refinement which minimally enriches this abstract domain by adding to the abstract domain all the concrete properties that would make the abstract transformer/fixpoint imprecise (so that all counter-examples are considered altogether [124]) [127]. Various weaker notions of transformer/fixpoint completion can also be considered, *e.g.* ignoring termination [45], relative to an observation of program behaviors [6], or complete with respect to unification for resolution-based concrete semantics [128].

12. Combinations of abstract domains

The complex design of abstract domains/Galois connections is better performed compositionally, starting from basic abstract domains which are then refined (Sect. 11), combined (this Sect. 12), and composed to get very expressive abstractions [56, Sect. 10], [82]. This consists in defining a functor on abstract domains/Galois connections than takes existing domains as parameter and creates a new one.

The most popular abstract domain functor is certainly the reduced product [56, Sect. 10.1], and its numerous variants (such as the more economical [33, 85]) and scope extensions [114, 215]. The reduced product functor creates a new domain out of existing domains that essentially performs the conjunction of the information carried by each domain by transferring commonly expressible information between domains. It is the most abstract domain which is more precise than the given domains. A recent example is the understanding of Nelson-Oppen combination of logical theories with equality in SMT solvers as a reduced product [92, 96] where the commonly expressible information between domains are uninterpreted in/equalities between variables.

The reduced power functor [56, 125, Sect. 10.2] is used to express conditional abstract properties that rely upon abstract conditions, as *e.g.* in an analysis by cases. The reduced power functor has many more specific instances such as polyvariance for context-sensitive analyzes [130] or trace partitioning [137, 202] where a local invariant at a program point may depend upon an abstraction of the history of computations reaching that point.

13. Equational design of abstract domains

The refinement (Sect. 11) and combination (Sect. 12) functors on abstract domains can be used to recursively define abstract domains equationally [115, 123, 209]. This has been exploited for semantics [122] but remains to be exploited in static analysis *e.g.* in the automatic design of static analyzers, which is the objective of the Galois connection calculus [82].

14. Galois connections for best abstraction

If the concept of Galois connection is central to abstract interpretation, it is equivalent to many other formalisms including closure operators [39, 56], Moore families and principal ideals, [56], soundness/logical relations [182], topologies [55, 208], *etc* which

[5] In [39, Ch. 4.1] both over- and under-approximations are explicitly considered to sandwich a fixpoint and it is stated that after Ch. 4.2 included, dual results are to be left implicit.

[6] At the time in data flow analysis, the merge-over-all paths, propagating abstract properties along execution paths, was used as the concrete domain to "justify" the abstract fixpoint data flow equations. This abstraction does not loose information if and only if it is closed under disjunctive completion [56, Sect. 10.2]. The correctness proof is therefore purely syntactic, not related to the program semantics, hence essentially circular and unsound. Using temporal logic to specify the data flow analysis does not help either since it does link the data flow to the program syntax (flowchart) not to its semantics [207]. For example, liveness is misinterpreted in [207] as noted in [71]. Moreover the finiteness hypothesis to compute the solution of the data flow equations using a fixpoint checker eliminate trivial infinite but Noetherian analyses such as constant propagation. The correctness argument in [151] is based upon a distributivity hypothesis not satisfied by the main example of the paper which is constant propagation.
Other precursors [183, 184, 211] understood (non-relational) static analysis as a specification calculus on non-standard values or (relational) static analysis as recursive definitions [199] *i.e.* sound by definition. For example the sign analysis in [211] is unsound in that negative $\ominus$ is interpreted as < 0 and positive $\oplus$ as $\geqslant 0$ so that $-1 \times 0 = 0$ is a counter-example to the classical rule $\ominus \times \oplus = \ominus$. 0 has no best abstraction and is handled specifically in mathematics by simplification rules. Intuition is not always right.

can express the concept of best abstraction: any concrete property has a most precise approximation in the abstract [56]. An interesting consequence is that the analysis/verification in the abstract is completely determined by the collecting semantics and the abstraction (nothing else has to be invented apart from convergence acceleration operators). This leads to the idea of calculational (even automatic) design [44, 210], or at least automatic certification of analyzers/verifiers (such as [44] and its extensions which has been formally checked in Coq [18]).

15. In absence of best abstraction

It was recognized early that in many cases there is no best abstraction. For example in [95], there is no best/smallest, even no minimal polyhedron enclosing a disk. Then only half of the Galois connection can be used [62], the most popular one being a concretization function only [95]. Because Galois connection preserves arbitrary joins the power set structure of concrete properties is preserved as complete lattices in abstract domains. This is no longer the case in absence of best abstraction, in which case joins/meets, even finite ones may not exist (so *e.g.* [151, 221] are no longer applicable). This can be compensated by using widening/narrowing[7]. Another example is the abstraction of languages by grammars [69] or of properties by a logical theory [91] where there is no best abstraction. In that case properties and fixpoint transformers may have infinitely many sound abstractions without a best/most precise one and so arbitrary choices have to be made in the abstract.

16. Abstraction of syntax

The syntax of languages as described by grammars provide varied examples of abstractions. Grammars can be given a fixpoint operational semantics of derivations which abstractions yield *e.g.* the Chomsky-Schützenberger fixpoint characterization of the language generated by the grammar, as well as top-down and bottom-up parsers and numerous algorithms used in compilation [75, 77, 80]. This generalizes to resolution-based languages [88] where essentially, text substitution is replaced by unification.

17. Syntactic abstractions

By syntactic abstraction we mean an abstraction of a collecting semantics to a meta-formalism describing languages, such as regular expressions, context-free grammars [69], or even theories in logic [93]. Typically, for syntactic abstractions, unless a property is exactly describable in the meta-formalism, there is no best abstraction. For example, if an over approximation of non-regular language of finite unbounded strings on a finite alphabet is described by a regular expression, there exists a sentence, of say length n, deriving from the regular expression which is not in the non-regular language to be over-approximated. A better description of the non-regular language consists in the regular expression enumerating the finitely many sentences of length less than or equal to n, and over approximating only those of greater length. There is no best and even minimal abstraction since, at least in theory, n can be indefinitely increased to get better and better abstractions. Of course the description of the concrete collecting semantics must not be subject to such expressivity restrictions which explains that abstract interpretation describe concrete domains using set/model theory rather than proof theory for a given first-order logic. Category theory would also be a mathematically reasonable choice, although it might not help in practice and would considerably restrict the readership [1].

[7] which is the first use of widening/narrowing in [39, Ch. 4], the convergence condition being later added [39, Ch. 5] to ensure termination of fixpoint iteration.

18. Abstraction of programs versus languages, and the power of extrapolation operators

That verification/analysis formal method have to be designed for programming languages and not for checking a given program (or a given model) makes a significant difference by adding a universal dependence on all programs of the language. So an abstract domain collects the part of the abstraction which is common to all programs and must be instantiated when analyzing a specific program.

A first example is set-based analysis which is often presented as acting on an infinite domain, which is true for the language, whereas a finite subset only is used for any given program. A consequence is that the set-based constraint-solving methods turned out to be mere finite fixpoint iterations in a finite domain [69].

A second example is on the use of Notherian/finite or non-Notherian/infinite domains. The case of finite abstractions is easy since always equivalent to predicate abstraction and reciprocally [47]. On the contrary, abstract interpretation stresses infinite non-Noetherian abstractions of infinitary properties. For a given program, a complete finite abstraction does exist for proving any desired property of this particular program [45]. In fact finding this abstraction or the proof is mathematically equivalent (and undecidable except *e.g.* for finite states). This is no longer the case for programming languages, for which any finite abstraction, or finite refinement [94], will fail or not terminate on infinitely many programs for which an infinite abstraction would have succeeded [45, 60]. This means that the relevant abstraction can only be determined during the analysis, which is the rôle of extrapolation operators.

Moreover, in practice, a finitude hypothesis is not of much help, since it does not prevent the combinatorial explosion of the set of cases to be considered, so that extrapolation operators must be used (or replaced by specific fixpoint computation methods when they exist). Of course trivial extrapolation operators (like any execution is to be over-approximated by chaos after a given number of steps) lead to very poor results, which are inconclusive but of a finite number of cases. Others like Milner's idea that all instances of a recursive procedure should be the same polymorphic type, which is a widening [43, 177], is quite satisfactory in practice.

19. Temporal abstraction

Although initially designed for reachability analysis [51], abstract interpretation applies *sensu stricto* to temporal properties, including time-symmetric trace-based ones referring both to the past, present, and future [71] (for which incompleteness results can be proved even for finite systems [126]).

20. Languages

Almost all families of programming languages (imperative languages [52], functional languages [180], Prolog [13, 26, 104, 142, 178, 179], constraint logic programs [119, 142], constraint solvers [189], database query languages [35], object-oriented languages [154], multithreaded programs [112], byte code [12], machine code [108, 109, 153, 222], *etc*), but also specification languages [129], grammars [175], algebraic polynomial systems [70], graph rewriting systems [138], logics [5], games [141], synchronous languages [135], continuous systems [37], hybrid automata [140], quantum computing [190], *etc* have been subject to static analysis by abstract interpretation.

21. Control-flow analysis

Besides data, static analysis by abstract interpretation should take into account the control structure of programs.

Control-flow (closure, escape, binding-time, *etc*) analysis are an abstract interpretation [165–167, 187].

22. Parallelism

Parallelism was considered early as a challenge to scalability [57, 58, 97]. One important aspect was which collecting semantics should be considered, otherwise stated which semantic model and proof methods should be abstracted. It turned out that the proof methods known at the time were just different abstractions of the same interleaving semantics [97]. Unfortunately the interest of funding studies of parallelism faded in the late eighties because the speed of sequential processors was progressing spectacularly. But any exponential process must stop by exhaustion of resources, which is the case nowadays, so the interest in parallelism is revived, by necessity. Fortunately, principles are immutable, so that more recent proof methods such as assume-guarantee, are also, like all proof methods, abstract interpretations, which can serve as an effective basis for designing scalable analyzers of parallel programs [172, 173]. Other aspects of parallelism are scheduling [132], parallelization of sequential or non-deterministic programs, where abstract interpretation can be used to take semantic aspects into account [25, 217].

23. Types

Type theory that developed around operational semantics, subject reduction, type inference by unification and typing rules, long appeared hardly comparable to abstract interpretation. This is not the case since [43] showed that polymorphic types and type inference are abstract interpretations with widening (to exclude recursive calls of functions with infinitely many different principle types). The approach was based on a denotational semantics, which was intentionally unusual but not surprising in the hierarchy of semantics [46]. The same approach applies to other unification-based analyzes [3].

24. Binary abstraction and hardware analysis

Binary abstractions abstract infinite sets of sequences with finite elements, sometimes of bounded length hence finite but still very large which requires widening for scalability [161, 171]. A trivial abstraction where "0 or 1" cases are abstracted into $\top$ is used for hardware analysis [223] and is one of the rare formal methods that scales on complex hardware. A more sophisticated abstract domain is in Astrée used to analyze integers and floats in their machine representation [171].

25. Numerical abstractions

Numerical abstractions abstract infinite sets of numerical vectors of finite dimension. There are numerous examples, including convex abstractions such as the non-relational intervals [52], and the relational linear inequalities [95], linear interval inequalities [30, 31], zonotopes [134] and a few non-convex abstractions such as linear absolute value inequalities [32]. Very good compromises such as octagons [170], pentagons [156], or parallelotopes [7] provide both scalability and enough precision for most common situations (like buffer-overrun checking in case the array bounds are symbolic).

26. Symbolic abstractions

Symbolic abstractions abstract infinite sets of in/finite functions, graphs, *etc* as found in most programming languages such as arrays [89], trees [162], infinitary relations [163], and heaps (like non-sharing [54, 55] or shape analysis [29, 198]). This research area where directly reusable mathematics essentially does not exist is quite important and still in infancy (except in a few easier particular cases, such as linear lists).

27. Simulations

(Bi-)simulations are abstract interpretations [194]. This new understanding has lead to more efficient simulation [196] and partitioning [195] algorithms.

28. Probabilistic abstractions

Probabilistic abstraction is another relatively underdeveloped application domain of abstract interpretation [2, 96, 99, 176, 204]. The reasons are that probabilistic behaviors are often only known with imprecision, that they mix with non-determinism on data, and that inference for infinite systems is much more complex than enumeration in the finite case since very subtle extrapolations are requested for precision.

29. Program transformation

The main applications of abstract interpretation beyond static analysis (*e.g.* to automatically infer loop invariants), are program transformations such as partial evaluation [146], abstract debugging [22, 40] abstract slicing [201], parallelization (see Sect. 22), compilation [200], *etc*.

Program transformations can be formalized as an abstraction, where a syntactic transformation of a source program is understood as first performed on the source program semantics to get a generally more efficient although equivalent transformed semantics which is then abstracted into the object program [74]. This is another example of syntactic abstraction (of the transformed semantics into the object program). More examples include partial evaluation [74], dead-end elimination [72], and refactoring [90].

30. Termination

Termination is an abstract interpretation [81] where the transfinite variant functions abstract in each program state what remains to be executed before potential or definite termination. This termination abstraction yields known proof methods [59, 65, 97] while further abstractions with widenings (*e.g.* piecewise linear [218] or ordinal-valued [218]) or policy iteration ([160]) yield static analyzers automatically inferring abstractions of variant functions. Another aspect of time abstraction is the determination of bounds on the worst-case execution times of programs [108, 109, 222].

31. Modularity

Static analyzers are complex programs, certainly much more complex than compilers, so they must be modular and extensible. The design by combination of abstractions is fundamental for scalability and extensibility [85]. Another aspect is the modular analysis of programs by parts [73]. The sound analyzers can be classified into the ones that perform a global program analysis (like Astrée [16, 17, 84]) which are precise but maybe costly and an analysis by part (like cccheck [107]) where cheap analyzes can be performed at compile time, but may require user-interaction to design code contracts (an activity which may also be automatized [83]), but not loop invariants which in all cases are inferred automatically.

32. Generalist versus domain-aware static analyzers

Sound generalist analyzers like Polyspace Code Prover™ [8], C Global Surveyor [220], Checkmate [113], CodeSonar for machine code [197], or Julia [188] produce good results for a large variety of codes in a given programming language. Most often they lack extreme precision. Using heuristics to sort out proliferating

[8] www.mathworks.com/products/polyspace-code-prover/

false alarms may be a fake commercial argument but does not have the rigor necessary to verify very large safety or security critical software.

This leads to the idea of analyzers aware of an application domain such as the precision of floating-point computations for Fluctuat [102, 133], control-command for Astrée [16, 17, 84]. In addition to generalist abstractions, filters, integrators, relation of variables to the clock, *etc* must be handled with greater precision thanks to specifically adapted abstract domains [110, 111]. Initially designed for avionic software, Astrée [16, 17, 84] had to be extended with a non-linear abstract domain to analyze precisely quaternions used for satellite positioning [20].

33. Industrial applications

The use of a principled approach to static analysis based on abstract interpretation allow for the rigorous and incremental design of precise and scalable static analyzers, with very few false alarms.

Astrée [14, 16, 17, 48, 84, 85, 87, 150] is a static analyzer commercialized by AbsInt and used in the medical, transportation, and communications industry for analyzing the absence of runtime errors in control command software, *e.g.* to control medical monitoring equipment, planes or satellites. This is certainly one of the first completely automatic tools based on formal methods that did scale up with enough precision to allow, after avoiding a few potentially catastrophic software failures, to become voluntarily mandatory in the design of *e.g.* avionic safety critical systems (see [103, 213] for reports on preliminary experiences).

cccheck [107, 155] is an example of general purpose, modular, precise, and efficient analyzer relying on the use of abstract code contracts. Being based on an intermediate language used by compilers it is applicable to several different programming languages that compile to this intermediate language. This is a possible approach to cope with the proliferation of programming languages. Other complementary approaches include the implementation of reusable libraries of abstract domains, like APRON [144] or the Parma library [9] for numerical properties.

34. Security

Security is typical of computer science lack of responsibility with respect to customers. One can leave a door open and take no responsibility at all when a burglar comes in, whereas this would be excluded by a restriction clause in any theft home insurance. This situation cannot go on for ever, and the computer industry will have to take responsibility for its provable errors. Such a proof of error can be given by a counter-example. Better, a proof of absence of error can be given by a tool proving the absence of error, provided the tool is sound and precise (otherwise it could be qualified of time-consuming and too restrictive, as for security types). Fully automatic tools are already able to catch such security bugs by taking values and objects into account (*e.g.* [216]), at least the common ones, but this is not yet a current practice. Many security properties cannot be checked at runtime so static analysis and verification tools have an outstanding domain of application.

Other areas in security where the semantic-based approach of abstract interpretation is much more powerful than syntactic matching or type inference is abstract non-interference [121], steganography [76], obfuscation [191, 192], and malware detection [193].

35. Unexpected applications

35.1 Biology

We anticipated in [39, Ch. 3] that basing abstract interpretation collecting semantics on transition systems "is obviously very general. It applies not only to computer systems but also to economic or biological systems, provided that the model of the system to study evolves according to discrete time steps.". In biology, it took some time before abstract interpretation was applied in practice in the context of rule-based modeling of protein-protein interaction networks [28, 100, 101]. The widening for economy was a bit too optimistic!

35.2 SAT and SMT solvers

The fact that SAT and SMT solvers are abstract interpretations of the semantics of logics came more as an unanticipated surprise [23, 24, 105, 106, 214]. This understanding yields new decision algorithms [136, 214].

36. Conclusion

Any semantic based-program manipulation, analysis, and verification in the scope of abstract interpretation theory can also be designed in the form of a specific paradigm/method/algorithm, illustrated by examples, and justified by an informal reasoning. This is the usual problem-specific approach of computer science which leads to a proliferation of disparate results lacking a unifying point of view. The consequence is that only one or two paradigms/methods/algorithms/examples are taught, remembered, and later extended, which completely inhibits the creative power. This empirical approach will certainly disappear over a long period of time in favor of more abstract foundations. This took centuries in mathematics were, by combining fundamental ideas, one reaches a definite result. This is indispensable to make computer science teachable beyond tricky know-how rapidly becoming outdated in the short term. In particular, it is not possible to assimilate, understand, and compare thousand of approaches to verification or inference in static analysis without a reduction to a few abstract concepts, explaining concisely how they can be (re-)designed. This is the objective of abstract interpretation, and requires a perpetual evolution, renewal, and rethinking of the foundations and practice, with very few basic principles remaining stable. Principled computer science is the only path to cost-effective safety and security.

Acknowledgments

Our acknowledgments go to the many contributors to abstract interpretation including those who are not referenced here by obvious lack of exhaustivity. They also go to Roberto Giacobazzi and Francesco Ranzato for their comments.

References

[1] S. Abramsky. Abstract interpretation, logical relations and kan extensions. *J. Log. Comput.*, 1(1):5–40, 1990.

[2] A. Adjé, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. E. Cohen and A. Rybalchenko, editors, *VSTTE*, volume 8164 of *Lecture Notes in Computer Science*, 22–47. 2013.

[3] A. Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[4] M. Alpuente and G. Vidal, editors. *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*. 2008.

[5] G. Amato and G. Levi. Abstract interpretation based semantics of sequent calculi. Palsberg [186], 38–57.

[6] G. Amato and F. Scozzari. Observational completeness on abstract interpretation. *Fundam. Inform.*, 106(2-4):149–173, 2011.

[7] G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electr. Notes Theor. Comput. Sci.*, 287:17–28, 2012.

[8] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.

[9] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[10] H. E. Bal, editor. *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*. IEEE Computer Society, 1994.

[11] A. Banerjee, O. Danvy, K.-G. Doh, and J. Hatcliff, editors. *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013*, volume 129 of *EPTCS*, 2013.

[12] R. Barbuti, N. D. Francesco, and L. Tesei. An abstract interpretation approach for enhancing the Java bytecode verifier. *Comput. J.*, 53(6): 679–700, 2010.

[13] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 15(1):133–181, 1993.

[14] 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, 2011.

[15] D. Bjørner, M. Broy, and I. V. Pottosin, editors. *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*, volume 735 of *Lecture Notes in Computer Science*. 1993.

[16] 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. T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, 85–108. 2002.

[17] 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. R. Cytron and R. Gupta, editors, *PLDI*, 196–207. 2003.

[18] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. Logozzo and Fähndrich [157], 324–344.

[19] H.-J. Boehm and C. Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 2013.

[20] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, volume SP-669, 1–7, Istambul, Turkey, May 2009. ESA.

[21] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.

[22] F. Bourdoncle. Semantic analysis of interval congruences. Bjørner et al. [15], 128–141.

[23] M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Interpolation-based verification of floating-point programs with abstract CDCL. Logozzo and Fähndrich [157], 412–432.

[24] M. Brain, V. D'Silva, L. Haller, A. Griggio, and D. Kroening. An abstract interpretation of DPLL(T). Giacobazzi et al. [118], 455–475.

[25] F. Bueno, M. J. G. de la Banda, and M. V. Hermenegildo. Effectivness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Trans. Program. Lang. Syst.*, 21(2):189–239, 1999.

[26] F. Bueno, P. López-García, and M. V. Hermenegildo. Multivariant non-failure analysis via standard abstract interpretation. Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, 100–116. 2004.

[27] T. Bultan and P.-A. Hsiung, editors. *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*. 2011.

[28] F. Camporesi, J. Feret, and J. Hayman. Context-sensitive flow analyses: A hierarchy of model reductions. A. Gupta and T. A. Henzinger, editors, *CMSB*, volume 8130 of *Lecture Notes in Computer Science*, 220–233. 2013.

[29] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. G. C. Necula and P. Wadler, editors, *POPL*, 247–260. 2008.

[30] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. J. Palsberg and Z. Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, 309–325. 2009.

[31] L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, 112–128. 2010.

[32] L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, 156–175. 2011.

[33] A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. Banerjee et al. [11], 325–336.

[34] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. Mycroft [181], 100–117.

[35] A. Cortesi and R. Halder. Abstract interpretation of recursive queries. C. Hota and P. K. Srimani, editors, *ICDCIT*, volume 7753 of *Lecture Notes in Computer Science*, 157–170. 2013.

[36] A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[37] G. Costantini, P. Ferrara, and A. Cortesi. Linear approximation of continuous systems with trapezoid step functions. R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *Lecture Notes in Computer Science*, 98–114. 2012.

[38] 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.

[39] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French)*. Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France, 21 March 1978.

[40] P. Cousot. Semantic foundations of program analysis. S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[41] P. Cousot. Methods and logics for proving programs. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 841–994. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[42] P. Cousot. Abstract interpretation based static analysis parameterized by semantics. Hentenryck [139], 388–394.

[43] P. Cousot. Types as abstract interpretations. P. Lee, F. Henglein, and N. D. Jones, editors, *POPL*, pages 316–331. ACM Press, 1997.

[44] P. Cousot. The calculational design of a generic abstract interpreter. M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[45] P. Cousot. Partial completeness of abstract fixpoint checking. B. Y. Choueiry and T. Walsh, editors, *SARA*, volume 1864 of *Lecture Notes in Computer Science*, 1–25. 2000.

[46] 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.

[47] P. Cousot. Verification by abstract interpretation. N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268, Taormina, Italy, June 29 – July 4 2003. © Springer-Verlag, Berlin, Germany.

[48] P. Cousot. Formal verification by abstract interpretation. A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, 3–7. 2012.

[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 p.

[50] 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 p.

[51] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. *Proceedings of the Second International Symposium on Programming*, 106–130. Dunod, Paris, France, 1976.

[52] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. R. M. Graham, M. A. Harrison, and R. Sethi, editors, *POPL*, 238–252. 1977.

[53] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, Aug. 1977.

[54] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. *ACM Symposium on Language Design for Reliable Software*, Raleigh, North Calorina, ACM SIGPLAN Notices 12(3):77–94, 1977.

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

[56] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *POPL*, 269–282. ACM Press, 1979.

[57] P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes (shortened version). J. W. de Bakker and J. van

Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, 119–133. 1980.

[58] P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, 243–271. Macmillan, New York, New York, United States, 1984.

[59] 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 Inf.*, 24(1):1–31, 1987.

[60] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *JTASPEFT/WSA*, 107–110, 1991.

[61] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[62] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[63] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. M. Bruynooghe and M. Wirsing, editors, *PLILP*, volume 631 of *Lecture Notes in Computer Science*, 269–295. 1992.

[64] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. R. Sethi, editor, *POPL*, 83–94. ACM Press, 1992.

[65] P. Cousot and R. Cousot. "a la Burstall" intermittent assertions induction principles for proving inevitable ability properties of programs. *Theor. Comput. Sci.*, 120(1):123–155, 1993.

[66] P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis (invited paper). Bjørner et al. [15], 98–127.

[67] P. Cousot and R. Cousot. Invited talk: Higher order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection, and PER analysis). Bal [10], 95–112.

[68] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. P. Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, 293–308. 1995.

[69] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. *FPCA*, 170–181, 1995.

[70] P. Cousot and R. Cousot. Abstract interpretation of algebraic polynomial systems (extended abstract). M. Johnson, editor, *AMAST*, volume 1349 of *Lecture Notes in Computer Science*, 138–154. 1997.

[71] P. Cousot and R. Cousot. Temporal abstract interpretation. M. N. Wegman and T. W. Reps, editors, *POPL*, 12–25. 2000.

[72] P. Cousot and R. Cousot. A case study in abstract interpretation based program transformation: Blocking command elimination. *Electr. Notes Theor. Comput. Sci.*, 45:41–64, 2001.

[73] P. Cousot and R. Cousot. Modular static program analysis. R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, 159–178. 2002.

[74] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. J. Launchbury and J. C. Mitchell, editors, *POPL*, pages 178–190. 2002.

[75] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theor. Comput. Sci.*, 290(1):531–544, 2003.

[76] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. Jones and Leroy [147], 173–185.

[77] P. Cousot and R. Cousot. Grammar analysis and parsing by abstract interpretation. T. W. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation*, volume 4444 of *Lecture Notes in Computer Science*, 175–200. 2006.

[78] P. Cousot and R. Cousot. Bi-inductive structural semantics. *Inf. Comput.*, 207(2):258–283, 2009.

[79] P. Cousot and R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. J. Esparza, B. Spanfelner, and O. Grumberg, editors, *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, 1–29. IOS Press, 2010.

[80] P. Cousot and R. Cousot. Grammar semantics, analysis and parsing by abstract interpretation. *Theor. Comput. Sci.*, 412(44):6135–6192, 2011.

[81] P. Cousot and R. Cousot. An abstract interpretation framework for termination. J. Field and M. Hicks, editors, *POPL*, 245–258. 2012.

[82] P. Cousot and R. Cousot. A Galois connection calculus for abstract interpretation. Jagannathan and Sewell [143], 3–4.

[83] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic inference of necessary preconditions. Giacobazzi et al. [118], 128–148.

[84] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. S. Sagiv, editor, *ESOP*, volume 3444 of *Lecture Notes in Computer Science*, 21–30. 2005.

[85] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. M. Okada and I. Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, 272–300. 2006.

[86] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3): 229–264, 2009.

[87] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée. *TASE*, 3–20. IEEE Computer Society, 2007.

[88] P. Cousot, R. Cousot, and R. Giacobazzi. Abstract interpretation of resolution-based semantics. *Theor. Comput. Sci.*, 410(46):4724–4746, 2009.

[89] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. T. Ball and M. Sagiv, editors, *POPL*, 105–118. 2011.

[90] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. G. T. Leavens and M. B. Dwyer, editors, *OOPSLA*, pages 213–232. 2012.

[91] P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. S. Nanz, editor, *The Future of Software Engineering*, pages 48–71. 2010.

[92] P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. M. Hofmann, editor, *FOSSACS*, volume 6604 of *Lecture Notes in Computer Science*, 456–472. 2011.

[93] P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *J. ACM*, 59(6):31, 2012.

[94] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. Nielson and Filé [185], 333–348.

[95] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *POPL*, 84–96. ACM Press, 1978.

[96] P. Cousot and M. Monerau. Probabilistic abstract interpretation. H. Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, 169–193. 2012.

[97] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles (in French)*. Thèse d'État ès sciences mathématiques, Nancy, Institut national polytechnique de Lorraine, 15 November 1985.

[98] R. Cousot, editor. *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. 2003.

[99] S. Crafa and F. Ranzato. Bisimulation and simulation algorithms on probabilistic transition systems by abstract interpretation. *Formal Methods in System Design*, 40(3):356–376, 2012.

[100] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Abstracting the differential semantics of rule-based models: Exact and automated model reduction. *LICS*, 362–381. IEEE Computer Society, 2010.

[101] V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract interpretation of cellular signalling networks. F. Logozzo, D. Peled, and L. D. Zuck, editors, *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, 83–97. 2008.

[102] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. M. Alpuente, B. Cook, and C. Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, 53–69. 2009.

[103] D. Delmas and J. Souyris. Astrée from research to industry. Nielson and Filé [185], 437–451.

[104] G. Delzanno, R. Giacobazzi, and F. Ranzato. Static analysis, abstract interpretation and verification in (constraint logic) programming. A. Dovier and E. Pontelli, editors, *25 Years GULP*, volume 6125 of *Lecture Notes in Computer Science*, 136–158. 2010.

[105] V. D'Silva, L. Haller, and D. Kroening. Abstract conflict driven learning. R. Giacobazzi and R. Cousot, editors, *POPL*, 143–154. 2013.

[106] V. D'Silva, L. Haller, and D. Kroening. Abstract satisfaction. Jagannathan and Sewell [143], 139–150.

[107] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. B. Beckert and C. Marché, editors, *FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, 10–30. 2010.

[108] C. Ferdinand, R. Heckmann, and R. Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. M. Broy, I. H. Krüger, and M. Meisinger, editors, *ASWSD*, volume 4147 of *Lecture Notes in Computer Science*, 1–14. 2004.

[109] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2): 163–189, 1999.

[110] J. Feret. Static analysis of digital filters. D. A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, 33–48. 2004.

[111] J. Feret. The arithmetic-geometric progression abstract domain. R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, 42–58. 2005.

[112] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. B. Beckert and R. Hähnle, editors, *TAP*, volume 4966 of *Lecture Notes in Computer Science*, 116–133. 2008.

[113] P. Ferrara. Checkmate: A generic static analyzer of Java multithreaded programs. D. V. Hung and P. Krishnan, editors, *SEFM*, 169–178. IEEE Computer Society, 2009.

[114] P. Ferrara. Generic combination of heap and value analyses in abstract interpretation. McMillan and Rival [164], 302–321.

[115] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Comput. Surv.*, 28(2):333–336, 1996.

[116] G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.

[117] T. M. Gawlitza, H. Seidl, A. Adjé, S. Gaubert, and E. Goubault. Abstract interpretation meets convex optimization. *J. Symb. Comput.*, 47(12):1416–1446, 2012.

[118] R. Giacobazzi, J. Berdine, and I. Mastroeni, editors. *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*. 2013.

[119] R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *J. Log. Program.*, 25(3):191–247, 1995.

[120] R. Giacobazzi and I. Mastroeni. Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003.

[121] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. Jones and Leroy [147], 186–197.

[122] R. Giacobazzi and I. Mastroeni. Transforming semantics by abstract interpretation. *Theor. Comput. Sci.*, 337(1-3):1–50, 2005.

[123] R. Giacobazzi and I. Mastroeni. Transforming abstract interpretations by abstract interpretation. Alpuente and Vidal [4], 1–17.

[124] R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. P. Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, 356–373. 2001.

[125] R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theor. Comput. Sci.*, 216(1-2):159–211, 1999.

[126] R. Giacobazzi and F. Ranzato. Incompleteness of states w.r.t. traces in model checking. *Inf. Comput.*, 204(3):376–407, 2006.

[127] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[128] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract domains condensing. *ACM Trans. Comput. Log.*, 6(1):33–60, 2005.

[129] F. Giannotti and D. Latella. Gate splitting in LOTOS specifications using abstract interpretation. *Sci. Comput. Program.*, 23(2-3):127–149, 1994.

[130] T. Gilray and M. Might. A survey of polyvariance in abstract interpretations. J. McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, 134–148. 2013.

[131] D. Gopan and T. W. Reps. Lookahead widening. T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, 452–466. 2006.

[132] E. Goubault. Schedulers as abstract interpreter of higher dimensional automata. N. D. Jones, editor, *PEPM*, 134–145. ACM Press, 1995.

[133] E. Goubault and S. Putot. Static analysis of numerical algorithms. K. Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, 18–34. 2006.

[134] E. Goubault, S. Putot, and F. Védrine. Modular static analysis with zonotopes. Miné and Schmidt [174], 24–40.

[135] N. Halbwachs. About synchronous programming and abstract interpretation. *Sci. Comput. Program.*, 31(1):75–89, 1998.

[136] L. Haller, A. Griggio, M. Brain, and D. Kroening. Deciding floating-point logic with systematic abstraction. G. Cabodi and S. Singh, editors, *FMCAD*, 131–140. IEEE, 2012.

[137] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. G. Levi, editor, *SAS*, volume 1503 of *Lecture Notes in Computer Science*, 200–214. 1998.

[138] C. Hankin. Graph rewriting systems and abstract interpretation. G. L. Burn, S. J. Gay, and M. Ryan, editors, *Theory and Formal Methods*, Workshops in Computing, 27–36. 1993.

[139] P. V. Hentenryck, editor. *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings*, volume 1302 of *Lecture Notes in Computer Science*. 1997.

[140] T. A. Henzinger and P.-H. Ho. A note on abstract interpretation strategies for hybrid automata. P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems*, volume 999 of *Lecture Notes in Computer Science*, pages 252–264. 1994.

[141] T. A. Henzinger, R. Majumdar, F. Y. C. Mang, and J.-F. Raskin. Abstract interpretation of game properties. Palsberg [186], 220–239.

[142] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.

[143] S. Jagannathan and P. Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 2014.

[144] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, 661–667. 2009.

[145] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. Jagannathan and Sewell [143], 529–540.

[146] N. D. Jones. Combining abstract interpretation and partial evaluation (brief overview). Hentenryck [139], 396–405.

[147] N. D. Jones and X. Leroy, editors. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. 2004.

[148] N. D. Jones and F. Nielson. *Abstract interpretation: a semantics-based tool for program analysis*, 527–636. Oxford University Press, 1995.

[149] N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. *Journal of Functional and Logic Programming*, 1997(2), 1997.

[150] D. Kästner, C. Ferdinand, S. Wilhelm, S. Nenova, O. Honcharova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, and É.-J. Sims. Astrée: Nachweis der Abwesenheit von Laufzeit. *Softwaretechnik-Trends*, 29(3), 2009.

[151] G. A. Kildall. A unified approach to global program optimization. P. C. Fischer and J. D. Ullman, editors, *POPL*, pages 194–206. ACM Press, 1973.

[152] T. Lev-Ami, R. Manevich, and S. Sagiv. TVLA: A system for generating abstract interpreters. R. Jacquart, editor, *IFIP Congress Topical Sessions*, pages 367–376. Kluwer, 2004.

[153] J. Lim and T. W. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Trans. Program. Lang. Syst.*, 35(1):4, 2013.

[154] F. Logozzo. Class-level modular analysis for object oriented languages. Cousot [98], 37–54.

[155] F. Logozzo, M. Barnett, M. Fähndrich, P. Cousot, and R. Cousot. A semantic integrated development environment. G. T. Leavens, editor, *SPLASH*, 15–16. 2012.

[156] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[157] F. Logozzo and M. Fähndrich, editors. *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*. 2013.

[158] D. Massé. Combining forward and backward analyses of temporal properties. O. Danvy and A. Filinski, editors, *PADO*, volume 2053 of *Lecture Notes in Computer Science*, 103–116. 2001.

[159] D. Massé. Abstract domains for property checking driven analysis of temporal properties. C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, 349–363. 2004.

[160] D. Massé. Policy iteration-based conditional termination and ranking functions. McMillan and Rival [164], 453–471.

[161] L. Mauborgne. Abstract interpretation using typed decision graphs. *Sci. Comput. Program.*, 31(1):91–112, 1998.

[162] L. Mauborgne. An incremental unique representation for regular trees. *Nord. J. Comput.*, 7(4):290–311, 2000.

[163] L. Mauborgne. Infinitary relations and their representation. *Sci. Comput. Program.*, 47(2-3):121–144, 2003.

[164] K. L. McMillan and X. Rival, editors. *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*. 2014.

[165] J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10, 2012.

[166] J. Midtgaard, M. D. Adams, and M. Might. A structural soundness proof for shivers's escape technique - a case for Galois connections. Miné and Schmidt [174], 352–369.

[167] J. Midtgaard and T. P. Jensen. Control-flow analysis of function calls

and returns by abstract interpretation. *Inf. Comput.*, 211:49–76, 2012.

[168] M. Might. Abstract interpreters for free. R. Cousot and M. Martel, editors, *SAS*, volume 6337 of *Lecture Notes in Computer Science*, 407–421. 2010.

[169] B. Mihaila, A. Sepp, and A. Simon. Widening as abstract domain. G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 170–184. 2013.

[170] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[171] A. Miné. Abstract domains for bit-level machine integer and floating-point operations. J. D. Fleuriot, P. Höfner, A. McIver, and A. Smaill, editors, *ATx/WInG@IJCAR*, volume 17 of *EPiC Series*, 55–70. EasyChair, 2012.

[172] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012.

[173] A. Miné. Relational thread-modular static value analysis by abstract interpretation. McMillan and Rival [164], 39–58.

[174] A. Miné and D. Schmidt, editors. *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*. 2012.

[175] U. Möncke and R. Wilhelm. Grammar flow analysis. H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, 151–186. 1991.

[176] D. Monniaux. Abstract interpretation of probabilistic semantics. Palsberg [186], 322–339.

[177] B. Monsuez. Polymorphic typing by abstract interpretation. R. K. Shyamasundar, editor, *FSTTCS*, volume 652 of *Lecture Notes in Computer Science*, 217–228. 1992.

[178] S. Muñoz-Hernández, J. J. Moreno-Navarro, and M. V. Hermenegildo. Efficient negation using abstract interpretation. R. Nieuwenhuis and A. Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, 485–494. 2001.

[179] K. Muthukumar and M. V. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. K. Furukawa, editor, *ICLP*, 49–63. MIT Press, 1991.

[180] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. B. Robinet, editor, *Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, 269–281. 1980.

[181] A. Mycroft, editor. *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 25-27, 1995, Proceedings*, volume 983 of *Lecture Notes in Computer Science*. 1995.

[182] A. Mycroft and N. D. Jones. A relational framework for abstract interpretation. H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 156–171. 1985.

[183] P. Naur. The design of the GIER ALGOL compiler. *BIT*, 3:124–140 and 145–166, 1963.

[184] P. Naur. Checking of operand types in ALGOL compilers. *BIT*, 5: 151–163, 1965.

[185] H. R. Nielson and G. Filé, editors. *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*. 2007.

[186] J. Palsberg, editor. *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings*, volume 1824 of *Lecture Notes in Computer Science*. 2000.

[187] J. Palsberg and M. I. Schwartzbach. Binding-time analysis: Abstract interpretation versus type inference. Bal [10], 277–288.

[188] É. Payet and F. Spoto. Static analysis of Android programs. *Information & Software Technology*, 54(11):1192–1201, 2012.

[189] M. Pelleau, A. Miné, C. Truchet, and F. Benhamou. A constraint solver based on abstract domains. Giacobazzi et al. [118], 434–454.

[190] S. Perdrix. Quantum entanglement analysis based on abstract interpretation. Alpuente and Vidal [4], 270–282.

[191] M. D. Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security*, 17(6):855–908, 2009.

[192] M. D. Preda, I. Mastroeni, and R. Giacobazzi. A formal framework for property-driven obfuscation strategies. L. Gasieniec and F. Wolter, editors, *FCT*, volume 8070 of *Lecture Notes in Computer Science*, 133–144. 2013.

[193] M. D. Preda, I. Mastroeni, and R. Giacobazzi. Analyzing program dependencies for malware detection. S. Jagannathan and P. Sewell, editors, *PPREW@POPL*, page 6. 2014.

[194] F. Ranzato and F. Tapparo. Generalized strong preservation by abstract interpretation. *J. Log. Comput.*, 17(1):157–197, 2007.

[195] F. Ranzato and F. Tapparo. Generalizing the Paige-Tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206(5):620–651, 2008.

[196] F. Ranzato and F. Tapparo. An efficient simulation algorithm based on abstract interpretation. *Inf. Comput.*, 208(1):1–22, 2010.

[197] T. W. Reps, J. Lim, A. V. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, 41–56. 2010.

[198] T. W. Reps, M. Sagiv, and R. Wilhelm. Shape analysis and applications. *The Compiler Design Handbook, 2nd ed.*, page 12. CRC Press, 2007.

[199] J. C. Reynolds. Automatic computation of data set definitions. *IFIP Congress (1)*, 456–461, 1968.

[200] X. Rival. Certification of compiled assembly code by invariant translation. *STTT*, 6(1):15–37, 2004.

[201] X. Rival. Abstract dependences for alarm diagnosis. K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, 347–363. 2005.

[202] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[203] M. Rosendahl. Abstract interpretation as a programming language. Banerjee et al. [11], 84–104.

[204] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. Boehm and Flanagan [19], 447–458.

[205] D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). Mycroft [181], 1–18.

[206] D. A. Schmidt. On the need for a popular formal semantics. *ACM Comput. Surv.*, 28(4es):175, 1996.

[207] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. D. B. MacQueen and L. Cardelli, editors, *POPL*, 38–48. 1998.

[208] D. A. Schmidt. Inverse-limit and topological aspects of abstract interpretation. *Theor. Comput. Sci.*, 430:23–42, 2012.

[209] F. Scozzari. *Domain theory in abstract interpretation: equations, completeness and logic*. PhD thesis, Dipartimento di Matematica, Univ. di Siena, via del Capitano 15, I-53100 Siena, Italy, February 1999.

[210] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. Boehm and Flanagan [19], 399–410.

[211] M. Sintzoff. Calculating properties of programs by valuations on specific models. *Proceedings of an ACM Conference on Proving Assertions about Programs*, Las Cruses, ACM SIGPLAN Notices 7(1):203–207, 6–7 January 1972.

[212] P. Sotin, B. Jeannet, F. Védrine, and E. Goubault. Policy iteration within logico-numerical abstract domains. Bultan and Hsiung [27], 290–305.

[213] J. Souyris and D. Delmas. Experimental assessment of Astrée on safety-critical avionics software. F. Saglietti and N. Oster, editors, *SAFECOMP*, volume 4680 of *Lecture Notes in Computer Science*, 479–490. 2007.

[214] A. V. Thakur and T. W. Reps. A generalization of Stålmarck's method. Miné and Schmidt [174], 334–351.

[215] A. Toubhans, B.-Y. E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. Giacobazzi et al. [118], 375–395.

[216] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. V. Cortellessa and D. Varró, editors, *FASE*, volume 7793 of *Lecture Notes in Computer Science*, 210–225. 2013.

[217] S. Tzolovski. Data dependence as abstract interpretations. Hentenryck [139], page 366.

[218] C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. Z. Shao, editor, *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, 412–431. 2014.

[219] M. Y. Vardi. Branching vs. linear time: Final showdown. T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, 1–22. 2001.

[220] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded C programs. W. Pugh and C. Chambers, editors, *PLDI*, 231–242. 2004.

[221] B. Wegbreit. Property extraction in well-founded property sets. *IEEE Trans. Software Eng.*, 1(3):270–285, 1975.

[222] R. Wilhelm and B. Wachter. Abstract interpretation with applications to timing validation. A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, 22–36. 2008.

[223] J. Yang and C.-J. H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *Lecture Notes in Computer Science*, 70–87. 2002.