The Rôle of Abstract Interpretation in Formal Methods

Patrick Cousot
École normale supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr

Formal methods In computer science and software engineering, *formal methods* are mathematically-based techniques for the specification, development and verification of software and hardware systems. They therefore establish the satisfaction of a specification by a system semantics.

Abstract Interpretation Abstract interpretation [3, 8] is a theory of sound approximation of mathematical structures, in particular those involved in the description of the behavior of computer systems. It allows the systematic derivation of sound methods and algorithms for approximating undecidable or highly complex problems in various areas of computer science (semantics, verification and proof, model-checking, static analysis, program transformation and optimization, typing, software steganography, etc.). Its main current application is on the safety and security of complex hardware and software computer systems.

Semantics The *semantics* $\mathcal{S}[\![\mathbb{p}]\!]$ is a formal model of the execution of these software and hardware systems $\mathbb{p} \in \mathbb{P}$. A *semantic domain* $\mathcal{D}$ is a set of such formal models (so $\forall \mathbb{p} \in \mathbb{P} : \mathcal{S}[\![\mathbb{p}]\!] \in \mathcal{D}$).

An example is an operational semantics of a program describing all possible program executions as a set of maximal traces that is finite or infinite sequences of states in Σ, two successive states corresponding to an elementary program step. In that case $\mathcal{D} \triangleq \wp(\mathcal{T})$[1] that is a subset of the set $\mathcal{T} \triangleq \bigcup_{n=1}^{+\infty} \Sigma^n$ of all possible traces where $\Sigma^n \triangleq [0, n[\mapsto \Sigma$ is the set of traces of length n, $n = 0, 1, \ldots, +\infty$.

Properties and Specifications A specification is a required property of the semantics of the system. The interpretation of a property is therefore a set of semantic models that satisfy this property so the set of *properties* is $\mathcal{P} \triangleq \wp(\mathcal{D})$. The strongest property of a system $\mathbb{p} \in \mathbb{P}$ is its semantics $\{\mathcal{S}[\![\mathbb{p}]\!]\}$ (called the *collecting semantics* $\mathcal{C}[\![\mathbb{p}]\!] \triangleq \{\mathcal{S}[\![\mathbb{p}]\!]\}$).

Verification The *satisfaction* of a specification $P \in \mathcal{P}$ by a system $\mathbb{p}$ (more precisely by the system semantics $\mathcal{S}[\![\mathbb{p}]\!]$) is $\mathcal{S}[\![\mathbb{p}]\!] \in P$, which can equivalently be defined as the proof that $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$.

Abstraction To prove $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$ one can use a sound over-approximation of the collecting semantics $\mathcal{C}[\![\mathbb{p}]\!] \subseteq \mathcal{C}^\sharp[\![\mathbb{p}]\!]$ and a sound under-approximation of the property $P^\sharp \subseteq P$ and make the correctness proof in the abstract $\mathcal{C}^\sharp[\![\mathbb{p}]\!] \subseteq P^\sharp$.

[1] $\wp(S) \triangleq \{S' \mid S' \subseteq S\}$ is the powerset of the set S.

For automated proofs, $\mathcal{C}^\sharp[\![\mathbb{p}]\!]$ and $P^\sharp$ must be computer-representable and are not chosen in the concrete domain $\langle\mathcal{P}, \subseteq\rangle$ but in an *abstract domain* $\langle\mathcal{P}^\sharp, \sqsubseteq\rangle$. The correspondence is given by a *concretization function* $\gamma \in \mathcal{P}^\sharp \mapsto \mathcal{P}$ providing the meaning $\gamma(P^\sharp)$ of abstract properties $P^\sharp$ and preserving the abstract implication $\forall Q_1, Q_2 \in \mathcal{P}^\sharp : (Q_1 \sqsubseteq Q_2) \Longrightarrow (\gamma(Q_1) \subseteq \gamma(Q_2))$. Then $\mathcal{C}^\sharp[\![\mathbb{p}]\!] \sqsubseteq P^\sharp$ implies $\gamma(\mathcal{C}^\sharp[\![\mathbb{p}]\!]) \subseteq \gamma(P^\sharp)$ and by soundness $\mathcal{C}[\![\mathbb{p}]\!] \subseteq \gamma(\mathcal{C}^\sharp[\![\mathbb{p}]\!])$ and $\gamma(P^\sharp) \subseteq P$ we have proved correctness $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$.

Best Abstraction If we want to over-approximate a disk in two dimensions by a polyhedron there is no smallest one, as shown by Euclid. However if we want to over-approximate a disk by a rectangular parallelepiped which sides are parallel to the axes, then there is definitely a smallest (square) one. In such a case there is an abstraction function $\alpha \in \mathcal{P} \mapsto \mathcal{P}^\sharp$ such that for all $P \in \mathcal{P}$, $\alpha(P) \in \mathcal{P}^\sharp$ is an abstract over-approximation of P (so $P \subseteq \gamma(\alpha(P))$) and it is the most precise abstract over-approximation (so $\forall Q \in \mathcal{P}^\sharp : P \subseteq \gamma(Q) \Longrightarrow \alpha(P) \sqsubseteq Q$ whence $\gamma(\alpha(P)) \subseteq \gamma(Q)$ by monotony of γ). It follows in that case of existence of a best abstraction, that the pair $\langle\alpha, \gamma\rangle$ is a Galois connection [8].

Abstraction is very often implicit, as shown by the following classical examples.

Aggregation Abstraction In the operational trace semantics example $\mathcal{D} \triangleq \wp(\mathcal{T})$ so properties are $\mathcal{P} \triangleq \wp(\wp(\mathcal{T}))$ where $\mathcal{T}$ is the set of traces. An example is $P_{01} \triangleq \{\{\sigma 0 \mid \sigma \in \mathcal{T}\}, \{\sigma 1 \mid \sigma \in \mathcal{T}\}\} \in \mathcal{P}$ specifying that executions of the system always terminate with 0 or always terminate with 1. This cannot be expressed in the traditional view of program properties as set of traces [1, 21]. This traditional understanding of a program property is given by the *aggregation abstraction* $\alpha_\cup \in \wp(\wp(\mathcal{T})) \mapsto \wp(\mathcal{T})$, $\alpha_\cup(P) \triangleq \bigcup P$ with concretization $\gamma_\cup \in \wp(\mathcal{T}) \mapsto \wp(\wp(\mathcal{T}))$, $\gamma_\cup(Q) \triangleq \wp(Q)$. An example is $\alpha_\cup(P_{01}) = \{\{\sigma 0, \sigma 1 \mid \sigma \in \mathcal{T}\}\}$ specifying that execution always terminate, either with 0 or with 1.

Transition Abstraction The transition abstraction $\alpha_\tau \in \wp(\mathcal{T}) \mapsto \wp(\Sigma \times \Sigma)$ collects transitions along traces. $\alpha_\tau(\sigma_0 \ldots \sigma_n) \triangleq \{\sigma_i \to \sigma_{i+1} \mid 0 \leqslant i < n\}$, $\alpha_\tau(\sigma_0 \ldots \sigma_i \ldots) \triangleq \{\sigma_i \to \sigma_{i+1} \mid i \geqslant 0\}$, and $\alpha_\tau(T) \triangleq \bigcup\{\alpha(\sigma) \mid \sigma \in T\}$. The concretization $\gamma_\tau \in \wp(\Sigma \times \Sigma) \mapsto \wp(\mathcal{T})$ is $\gamma_\tau(\tau) \triangleq \bigcup_{n=1}^{+\infty}\{\sigma \in [0, n[\mapsto \Sigma \mid \forall i < n : \langle \sigma_i, \sigma_{i+1}\rangle \in \tau\}$. The abstraction may also collect initial states $\alpha_\iota(T) \triangleq \{\sigma_0 \mid \sigma \in T\}$ so $\alpha_{\iota\tau}(T) \triangleq \langle \alpha_\iota(T), \alpha_\tau(T)\rangle$. We let $\gamma_{\iota\tau} \triangleq \gamma_\iota(\iota) \cap \gamma_\tau(\tau)$ where $\gamma_\iota(\iota) \triangleq \{\sigma \in \mathcal{T} \mid \sigma_0 \in \iota\}$ ($\langle \alpha_{\iota\tau}, \gamma_{\iota\tau}\rangle$ is a Galois connection).

This abstraction into a transition system [3] underlies small-step operational semantics. This is an approximation since traces can express properties not expressible by a transition system (like fairness of parallel processes).

Input-Output Abstraction The input-output abstraction $\alpha_{io} \in \wp(\mathcal{T}) \mapsto \wp(\Sigma \times (\Sigma \cup \{\bot\}))$ collects initial and final states of traces (and maybe $\bot$ for infinite traces to track nontermination). $\alpha_{io}(\sigma_0 \ldots \sigma_n) = \langle \sigma_0, \sigma_n\rangle$, $\alpha_{io}(\sigma_0 \ldots \sigma_i \ldots) = \langle \sigma_0, \bot\rangle$, and $\alpha_{io}(T) = \{\alpha_{io}(\sigma) \mid \sigma \in T\}$. The input-output abstraction α_{io} underlies denotational semantics, as well as big-step operational, predicate transformer and axiomatic semantics extended to nontermination [5] and interprocedural static analysis using relational procedure summaries [3, 7, 12].

Reachability Abstraction and Invariants The reachability abstraction $\alpha_r \in \wp(\mathcal{T}) \mapsto \wp(\Sigma)$ collects states along traces. $\alpha_r(T) \triangleq \{\sigma_i \mid \exists n \in [0, +\infty] : \sigma \in \Sigma^n \cap T \wedge i \in [0, n[\} = \{s' \in \Sigma \mid \exists s \in \iota : \langle s, s'\rangle \in \tau^\star\}$ where $\alpha_{\iota\tau}(T) = \langle \iota, \tau\rangle$ is the transition abstraction and $\tau^\star$ is the reflexive transitive closure of τ. Expressed in logical form, the reachability abstraction α provides a system invariant $\alpha(\mathcal{C}[\![\mathbb{p}]\!])$ that is the set of all states that can be reached along some execution of the system $\mathbb{p}$ [3, 6].

Floyd's method [16] to prove a reachability property $\alpha_r(T) \subseteq P$ consists in finding an *invariant* I stronger than P (i.e. $I \subseteq P$) which is *inductive* (i.e. $\iota \subseteq I$ and $\tau[I] \subseteq I$ where $\tau[I] \triangleq \{s' \mid \exists s \in I : \langle s, s'\rangle \in \tau\}$ is the right-image transformer for the transition system $\langle \iota, \tau\rangle = \alpha_{\iota\tau}(T)$). This induction principle has many equivalent variants [9], all underlying different static analysis methods (the equivalence may not be preserved by abstraction). In particular backward analyzes are based on $\langle \tau^{-1}, \alpha_\varphi(T)\rangle$ where τ^{-1} is the inverse of τ and $\alpha_\varphi(T) \triangleq \{\sigma_{n-1} \mid n < +\infty \wedge \sigma \in T \cap \Sigma^n\}$ collects final states.

Soundness and Completeness of Abstractions An abstraction is *sound* if the proof in the abstract $\mathcal{C}^\sharp[\![\mathbb{p}]\!] \subseteq P^\sharp$ implies the concrete property $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$. Abstract interpretation provides an effective theory to design sound abstractions. An abstraction is *complete* if the fact that the system is correct, that is $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$, can always be proved in the abstract as $\mathcal{C}^\sharp[\![\mathbb{p}]\!] \subseteq P^\sharp$. By refining the abstraction this is always possible [18], but this refinement is not effective (i.e. the algorithm does not terminate in general). For example in model-checking any abstraction of a trace logic may be incomplete [17].

Verification by Static Analysis *Static code analysis* is the analysis of computer system by direct inspection of the source or object code describing this system with respect to a semantics of this code (without executing programs as in *dynamic analysis*). The static code analysis is performed by an automated tool, as opposed to program understanding or program comprehension by humans. The proof $\mathcal{C}[\![\mathbb{p}]\!] \subseteq P$ is done in the abstract $\mathcal{C}^\sharp[\![\mathbb{p}]\!] \subseteq P^\sharp$, which involves the static analysis of $\mathbb{p}$ that is the effective computation of $\mathcal{C}^\sharp[\![\mathbb{p}]\!]$, as formalized by abstract interpretation [3, 8].

Adequation of Abstractions The reachability abstraction is sound and complete for invariance/safety proofs. That means that if $S \subseteq \Sigma$ is a set of safe states so that $\gamma_r(S)$ is a set of safe traces then the safety proof $\mathcal{C}[\![\mathbb{p}]\!] \subseteq \gamma_r(S)$ can always be done as $\alpha_r(\mathcal{C}[\![\mathbb{p}]\!]) \subseteq S$. This is the fundamental remark of Floyd [16] that it is not necessary to reason on traces to prove invariance properties. This does not mean that this abstraction is *adequate*, that is, informally, the most simple way to do the proof. For example Burstall's intermittent assertions may be simpler than Floyd's invariant assertions [10] or, in static analysis trace partitioning may be more adequate that state-based reachability analysis [19].

Property versus Model-based Abstraction Let $\langle \iota, \tau\rangle$ be a transition system *model* of a software or hardware system $\mathbb{p} \in \mathbb{P}$ (so that $\mathcal{S}[\![\mathbb{p}]\!] \triangleq \gamma_{\iota\tau}(\langle \iota, \tau\rangle)$). A *model-based abstraction* is an abstract transition system $\langle \iota^\sharp, \tau^\sharp\rangle$ which over-approximates $\langle \iota, \tau\rangle$ (so that, up to concretization, $\iota \subseteq \iota^\sharp$ and $\tau \subseteq \tau^\sharp$). The set of reachable abstract states for $\langle \iota^\sharp, \tau^\sharp\rangle$ over-approximate the reachable concrete states of $\langle \iota, \tau\rangle$ so the model-based abstractions yields sound abstractions of the concrete reachability states. Some abstractions defined by a Galois connection of sets of (reachable) states may not be model-based abstractions, in particular when the abstract domain is not a powerset of states (e.g. [20]).

Program-based versus Language-based Abstraction Static analysis has to define an abstraction $\alpha[\![\mathbb{p}]\!]$ for all programs $\mathbb{p} \in \mathbb{P}$ of a language $\mathbb{P}$. This is different from defining an abstraction specific to a given program. In particular an abstraction specific to a given program can always be refined to be complete using a finite abstract domain [4] whereas this is impossible in general for a language-based abstraction for which infinite abstract domains have been shown to always produce better results [11].

False Alarms Static analysis being undecidable, it relies on incomplete language-based abstractions. This means that the analyzer will produce false alarms on infinitely many programs (which can even be generated automatically). A *false alarm* is a case when a concrete property holds but this cannot be proved in the abstract for the given abstraction. An example in reachability analysis is when no inductive invariant can be expressed in the abstract. The experience of ASTRÉE (`www.astree.ens.fr`, [2]) shows that it is possible to design precise language-based abstractions which produce no false alarm on a well defined family of programs[2].

Design of Abstractions The design of a sound and precise language-based abstraction is difficult. First from a mathematical point of view, one must discover the appropriate set of abstract properties that are needed to represent the necessary inductive invariants. Of course mathematical completion techniques could be used [18] but because of undecidability, they do not terminate in general. Second, from a computer-science point of view, one must find an appropriate *computer representation* of abstract properties and abstract transformers. Universal representations (e.g. using symbolic terms, automata or BDDs) are in general inefficient and the discovery of appropriate computer representations is far from being automatized.

Local versus Global Abstractions A simple approach to static analysis is to use the same *global abstraction* everywhere in the program, which hardly scales up. More sophisticated abstractions, as used in ASTRÉE are not uniform, different *local abstractions* being in different program regions [2].

Multiple versus Single Abstractions Because of the complexity of abstractions, it is simpler to design a precise abstraction by composing many elementary abstractions which are simple to understand and implement. ASTRÉE uses many weakly relational domains (such as octagons [20], digital filters [14], arithmetico-geometric progressions [15], etc) that could hardly be encoded efficiently using a universal representation of program properties as found in theorem provers, proof assistants or model-checkers.

Abstraction Reduction If several abstractions are used, the static analyzer must implement their conjunction in the concrete, that is their *reduced product* in the abstract [8]. The implementation must be *extensible*, allowing for the easy incorporation of new abstractions [13].

Refinement For a single program-based abstraction, refinement consists in strengthening the abstract invariant until it is inductive, through a fixpoint computation which in general does not terminate or explodes combinatorially. The problem is even harder for language-based abstractions. The pragmatic approach used in ASTRÉE is to manually design new abstractions which are incorporated in the reduced product of all abstractions used by the analyzer thus allowing for the strengthening of the abstract invariants for all programs until no false alarm is left [13].

[2]Synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language for ASTRÉE.

References

[1] B. Alpern and F. Schneider. Defining liveness. *Inf. Process. Lett.*, 21:181–185, 1985.

[2] 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. *ACM PLDI*, 196–207, 2003.

[3] 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 scientifique et médicale de Grenoble, Grenoble, 1978.

[4] P. Cousot. Partial completeness of abstract fixpoint checking. *SARA*, LNAI 1864, 1–25. Springer, 2000.

[5] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, 277(1—2):47–103, 2002.

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

[7] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. *IFIP Conf. on Formal Description of Programming Concepts*, 237–277, North-Holland, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *6^{th} ACM POPL*, 269–282, 1979.

[9] P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. *Tools & Notions for Program Construction*, 43–119. Cambridge U. Press, 1982.

[10] P. Cousot and R. Cousot. Sometime = always + recursion $\equiv$ always: on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs. *Acta Informat.*, 24:1–31, 1987.

[11] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. *PLILP '92*, LNCS 631, 269–295. Springer, 1992.

[12] P. Cousot and R. Cousot. Modular static program analysis. *11^{th} CC*, LNCS 2304, 159–178, Springer, 2002.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer, invited paper. *11^{th} ASIAN*, LNCS, Springer (to appear).

[14] J. Feret. Static analysis of digital filters. *30^{th} ESOP*, LNCS 2986, 33–48. Springer, 2004.

[15] J. Feret. The arithmetic-geometric progression abstract domain. *6^{th} VMCAI*, LNCS 3385, 42–58, Springer, 2005.

[16] R. Floyd. Assigning meaning to programs. *Proc. Symp. in Applied Math.*, vol. 19, 19–32. AMS, 1967.

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

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

[19] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. *14^{th} ESOP*, LNCS 3444, 5–20. Springer, 2005.

[20] A. Miné. The octagon abstract domain. *Higher-Order and Symb. Comp.*, 19:31–100, 2006.

[21] A. Pnueli. The temporal logic of programs. *18^{th} ACM FOCS*, 46–57, 1977.