# Abstract Interpretation Based Formal Methods and Future Challenges (Electronic Version)

Patrick Cousot

École normale supérieure, Département d'informatique,
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr http://www.di.ens.fr/~cousot/

**Abstract.** In order to contribute to the solution of the software reliability problem, tools have been designed to analyze statically the run-time behavior of programs. Because the correctness problem is undecidable, some form of approximation is needed. The purpose of *abstract interpretation* is to formalize this idea of *approximation*. We illustrate informally the application of abstraction to the *semantics* of programming languages as well as to *static program analysis*. The main point is that in order to reason or compute about a complex system, some information must be lost, that is the observation of executions must be either partial or at a high level of abstraction.

In the second part of the paper, we compare static program analysis with deductive methods, model-checking and type inference. Their foundational ideas are briefly reviewed, and the shortcomings of these four methods are discussed, including when they should be combined. Alternatively, since program debugging is still the main program verification method used in the software industry, we suggest to combine formal with informal methods.

Finally, the grand challenge for all formal methods and tools is to solve the software reliability, trustworthiness or robustness problems. A few challenges more specific to static program analysis by abstract interpretation are briefly discussed.

## 1 Introductory Motivations

The evolution of hardware by a factor of $10^6$ over the past 25 years has lead to the explosion of the size of programs in similar proportions. The scope of application of very large programs (from 1 to 40 millions of lines) is likely to widen rapidly in the next decade. Such big programs will have to be designed at a reasonable cost and then modified and maintained during their lifetime (which is often over 20 years). The size and efficiency of the programming and maintenance teams in charge of their design and follow-up cannot grow in similar proportions. At a not so uncommon (and often optimistic) rate of one bug per thousand lines such huge programs might rapidly become hardly manageable in particular for safety critical systems. Therefore in the next 10 years, the *software* *reliability problem* is likely to become a major concern and challenge to modern highly computer-dependent societies.

In the past decade a lot of progress has been made both on *thinking/methodological tools* (to enhance the human intellectual ability) to cope with complex software systems and *mechanical tools* (using the computer) to help the programmer to reason about programs.

Mechanical tools for computer aided program verification started by executing or simulating the program in as much as possible environments. However debugging of compiled code or simulation of a model of the source program hardly scale up and often offer a low coverage of dynamic program behavior.

Formal program verification methods attempt to mechanically prove that program execution is correct in all specified environments. This includes *deductive methods*, *model checking*, *program typing* and *static program analysis*.

Since program verification is undecidable, computer aided program verification methods are all partial or incomplete. The undecidability or complexity is always solved by using some form of *approximation*. This means that the mechanical tool will sometimes suffer from practical time and space complexity limitations, rely on finiteness hypotheses or provide only semi-algorithms, require user interaction or be able to consider restricted forms of specifications or programs only. The mechanical program verification tools are all quite similar and essentially differ in their choices regarding the approximations which have to be done in order to cope with undecidability or complexity. The purpose of *abstract interpretation* is to formalize this notion of approximation in a unified framework [36, 43].

## 2 Abstract Interpretation

Since program verification deals with properties, that is sets (of objects with these properties), abstract interpretation can be formulated in an application independent setting, as a theory for approximating sets and set operations as considered in set (or category) theory, including inductive definitions [52]. A more restricted understanding of abstract interpretation is to view it as a theory of approximation of the behavior of dynamic discrete systems (e.g. the formal semantics of programs or a communication protocol specification). Since such behaviors can be characterized by fixpoints (e.g. corresponding to iteration), an essential part of the theory provides constructive and effective methods for fixpoint approximation and checking by abstraction [46, 50].

### 2.1 Fixpoint Semantics

The *semantics of a programming language* defines the semantics of any program written in this language. The *semantics of a program* provides a formal mathematical model of all possible behaviors of a computer system executing this program in interaction with any possible environment. In the following we will try to explain informally why the semantics of a program can be defined as the solution of a fixpoint equation. Then, in order to compare semantics, we will show that all the semantics of a program can be organized in a hierarchy by abstraction. By observing computations at different levels of abstraction, one can approximate fixpoints hence organize the semantics of a program in a lattice [41].

### 2.2 Trace Semantics

Our finer grain of observation of program execution, that is the most precise of the semantics that we will consider, is that of a *trace semantics* [41, 46], a model also frequently used in temporal logic. An execution of a program for a given specific interaction with its environment is a sequence of states, observed at discrete intervals of time, starting from an initial state, then moving from one state to the next state by executing an atomic program step or transition and either ending in a final regular or erroneous state or non terminating, in which case the trace is infinite (see Fig. 1).

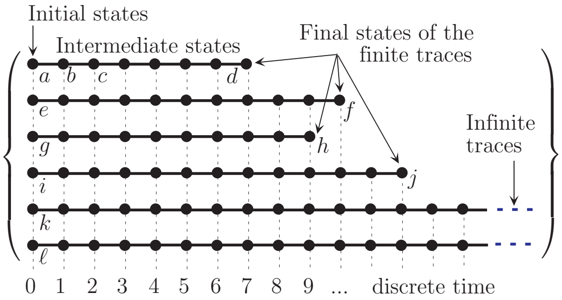



**Fig. 1.** Examples of Computation Traces

### 2.3 Least Fixpoint Trace Semantics

Introducing the *computational partial ordering* [41], we define the trace semantics in fixpoint form [41], as the least solution of an equation of the form $X = \mathcal{F}(X)$ where $X$ ranges over sets of finite and infinite traces.

More precisely, let **Behaviors** be the set of execution traces of a program, possibly starting in any state. We denote by $\mathbf{Behaviors}^+$ the subset of finite traces and by $\mathbf{Behaviors}^\infty$ the subset of infinite traces.

A finite trace $\overset{\text{a}}{\bullet}\!\!\longrightarrow\ldots\longrightarrow\!\!\overset{\text{z}}{\bullet}$ in $\mathbf{Behaviors}^+$ is either reduced to a final state (in which case there is no possible transition from state $\overset{\text{a}}{\bullet} = \overset{\text{z}}{\bullet}$) or the initial state $\overset{\text{a}}{\bullet}$ is not final and the trace consists of a first computation step $\overset{\text{a}}{\bullet}\!\!\longrightarrow\!\!\overset{\text{b}}{\bullet}$ after which, from the intermediate state $\overset{\text{b}}{\bullet}$, the execution goes on with the shorter finite trace $\overset{\text{b}}{\bullet}\!\!\longrightarrow\ldots\longrightarrow\!\!\overset{\text{z}}{\bullet}$ ending in the final state $\overset{\text{z}}{\bullet}$. The finite traces are therefore all well defined by induction on their length.

An infinite trace $\overset{\text{a}}{\bullet}\!\!\longrightarrow\ldots\longrightarrow\ldots$ in $\mathbf{Behaviors}^\infty$ starts with a first computation step $\overset{\text{a}}{\bullet}\!\!\longrightarrow\!\!\overset{\text{b}}{\bullet}$ after which, from the intermediate state $\overset{\text{b}}{\bullet}$, the execution goes on with an infinite trace $\overset{\text{b}}{\bullet}\!\!\longrightarrow\ldots\longrightarrow\ldots$ starting from the intermediate state $\overset{\text{b}}{\bullet}$. These remarks and $\mathbf{Behaviors} = \mathbf{Behaviors}^+ \cup \mathbf{Behaviors}^\infty$ lead to the following fixpoint equation:

$$\begin{aligned}\textbf{Behaviors} = {} & \{\overset{a}{\bullet} \mid \overset{a}{\bullet} \text{ is a final state}\}\\
& \cup \{\overset{a}{\bullet}\text{—}\overset{b}{\bullet}\text{—}\ldots\text{—}\overset{z}{\bullet} \mid \overset{a}{\bullet}\text{—}\overset{b}{\bullet} \text{ is an elementary step } \&\\
& \qquad\qquad\qquad \overset{b}{\bullet}\text{—}\ldots\text{—}\overset{z}{\bullet} \in \textbf{Behaviors}^{+}\}\\
& \cup \{\overset{a}{\bullet}\text{—}\overset{b}{\bullet}\text{—}\ldots\text{—}\ldots \mid \overset{a}{\bullet}\text{—}\overset{b}{\bullet} \text{ is an elementary step } \&\\
& \qquad\qquad\qquad \overset{b}{\bullet}\text{—}\ldots\text{—}\ldots \in \textbf{Behaviors}^{\infty}\}\end{aligned}$$

In general, the equation has multiple solutions. For example if there is only one non-final state $\overset{a}{\bullet}$ and only possible elementary step $\overset{a}{\bullet}\text{—}\overset{a}{\bullet}$ then the equation is $\textbf{Behaviors} = \{\overset{a}{\bullet}\text{—}\overset{a}{\bullet}\text{—}\ldots\text{—}\ldots \mid \overset{a}{\bullet}\text{—}\ldots\text{—}\ldots \in \textbf{Behaviors}\}$. One solution is $\{\overset{a}{\bullet}\text{—}\overset{a}{\bullet}\text{—}\overset{a}{\bullet}\text{—}\overset{a}{\bullet}\text{—}\ldots\text{—}\ldots\}$ but another one is the empty set $\emptyset$. Therefore, we choose the least solution for the *computational partial ordering* [41]:

*« More finite traces & less infinite traces »* .

### 2.4 Abstractions & Abstract Domains

A programming language semantics is more or less precise according to the considered observation level of program execution. This intuitive idea can be formalized by *Abstract interpretation* [41] and applied to different languages, including for proof methods.

The *theory of abstract interpretation* formalizes this notion of approximation and abstraction in a mathematical setting which is independent of particular applications. In particular, abstractions must be provided for all mathematical constructions used in semantic definitions of programming and specification languages [46, 50].

An *abstract domain* is an abstraction of the concrete semantics in the form of *abstract properties* (approximating the concrete properties **Behaviors**) and *abstract operations* (including abstractions of the concrete approximation and computational partial orderings, an approximation of the concrete fixpoint transformer $\mathcal{F}$, etc.). Abstract domains for complex approximations of designed by composing abstract domains for simpler components [46], see Sec. 2.10.

If the approximation is coarse enough, the abstraction of a concrete semantics can lead to an abstract semantics which is less precise, but is *effectively computable* by a computer. By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them [42]. *Abstract interpretation algorithms* provide approximate methods for computing this abstract semantics. The most important algorithms in abstract interpretation are those providing effective methods for the exact or approximate iterative resolution of fixpoint equations [43].

We will first illustrate formal and effective abstractions for sets. Then we will show that such abstractions can be lifted to functions and finally to fixpoints.

The abstraction idea and its formalization are equally applicable in other areas of computer science such as artificial intelligence e.g. for intelligent planning, proof checking, automated deduction, theorem proving, etc.

### 2.5 Hierarchy of Abstractions

As shown in Fig. 2 (from [41], where **Behaviors**, denoted $\tau^{\vec{\infty}}$ for short, is the lattice infimum), all abstractions of a semantics can be organized in a lattice (which is part of the lattice of abstract interpretations introduced in [46]). The *approximation partial ordering* of this lattice formally corresponds to logical implication, intuitively to the idea that one semantics is more precise than another one.

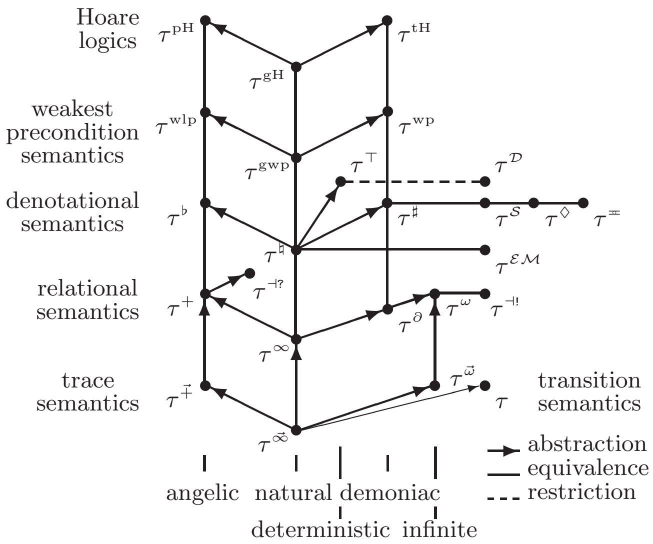



**Fig. 2.** The Hierarchy of Semantics

Fig. 3 illustrates the derivation of a *relational semantics* (denoted $\tau^{\infty}$ in Fig. 2) from a trace semantics (denoted $\tau^{\vec{\infty}}$ in Fig. 2). The abstraction $\alpha_r$ from trace to relational semantics consists in replacing the finite traces $\overset{a}{\bullet}\!\!-\!\!-\!\!\ldots\!\!-\!\!-\!\!\overset{z}{\bullet}$ by the pair $\langle a, z\rangle$ of the initial and final states. The infinite traces $\overset{a}{\bullet}\!\!-\!\!-\!\!\overset{b}{\bullet}\!\!-\!\!-\ldots\!\!-\!\!-\ldots$ are replaced by the pair $\langle a, \bot\rangle$ where the symbol $\bot$ denotes non-termination. Therefore the abstraction is:

$$\alpha_r(X) = \{\langle a, z\rangle \mid \overset{a}{\bullet}\!\!-\!\!-\!\!\ldots\!\!-\!\!-\!\!\overset{z}{\bullet} \in X\} \cup \{\langle a, \bot\rangle \mid \overset{a}{\bullet}\!\!-\!\!-\!\!\overset{b}{\bullet}\!\!-\!\!-\ldots\!\!-\!\!-\ldots \in X\} .$$

The *denotational semantics* (denoted $\tau^{\natural}$ in Fig. 2) is the isomorphic representation of a relation by its right-image:

$$\alpha_d(R) = \boldsymbol{\lambda}\, a \cdot \{x \mid \langle a, x\rangle \in R\}.$$

The abstraction from relational to *big-step operational or natural semantics* (denoted $\tau^{+}$ in Fig. 2) simply consists in forgetting everything about non-termination, so $\alpha_n(R) = \{\langle a, x\rangle \in R \mid x \neq \bot\}$, as illustrated in Fig. 3.

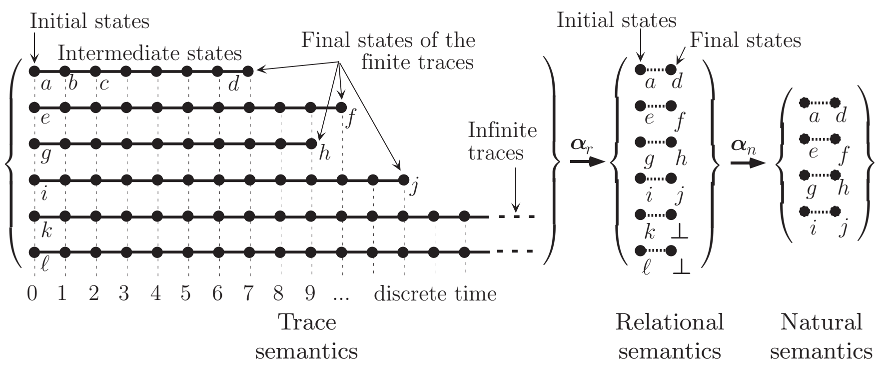



**Fig. 3.** Abstraction from Trace to Relational and Natural Semantics

A non comparable abstraction consists in collecting the set of initial and final states as well as all transitions $\langle x, y\rangle$ appearing along some finite or infinite trace $\overset{a}{\bullet}$—. . . $\overset{x}{\bullet}$—$\overset{y}{\bullet}$. . . of the trace semantics. One gets the *small-step operational or transition semantics* (denoted $\tau$ in Fig. 2 and also called Kripke structure in modal logic) as illustrated in Fig. 4.

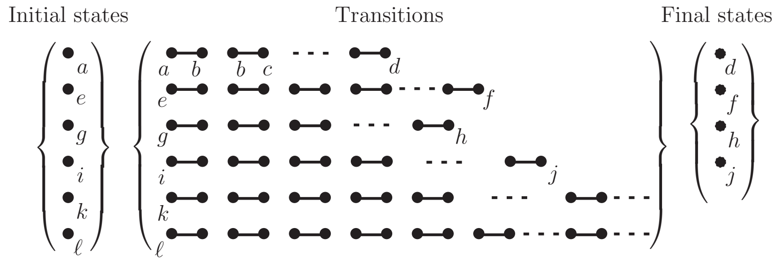



**Fig. 4.** Transition Semantics

A further abstraction consists in collecting all states appearing along some finite or infinite trace as illustrated in Fig. 5. This is the *partial correctness semantics* or the *static/collecting semantics* for proving invariance properties of programs.

All abstractions considered in this paper are "from above" so that the abstract semantics describes a superset or logical consequence of the concrete semantics. Abstractions "from below" are dual and consider a subset of the concrete semantics. An example of approximation "from below" is provided by debugging techniques which consider a subset of the possible program executions or by existential checking where one wants to prove the existence of an execution trace prefix fulfilling some given specification. In order to avoid repeating two times dual concepts and as we do usually, we only consider approximations

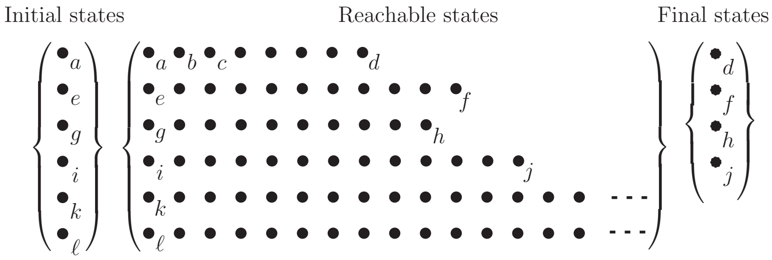
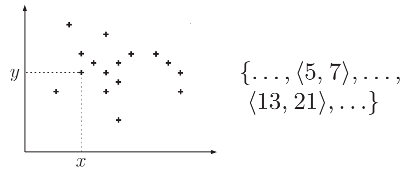
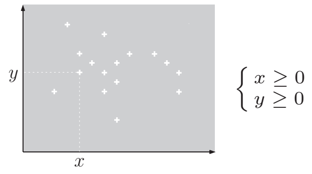


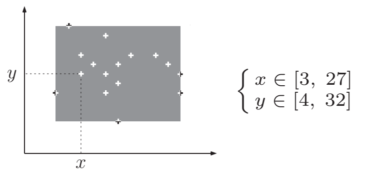

**Fig. 5.** Static / Collecting / Partial Correctness Semantics



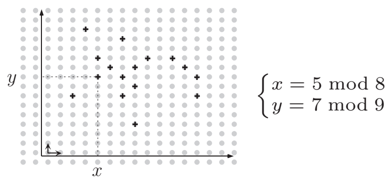

(a) [In]finite Set of Points




(b) Sign Abstraction




(c) Interval Abstraction




(d) Simple Congruence Abstraction

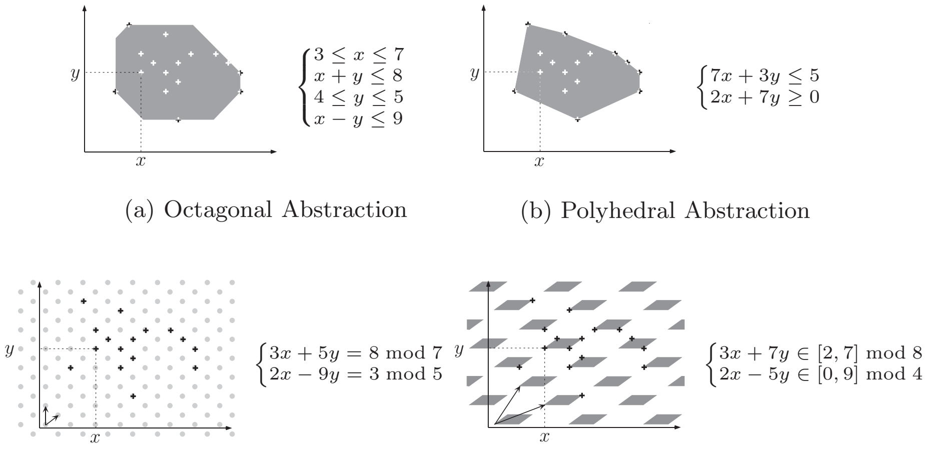
**Fig. 6.** Non-relational Abstractions

"from above", knowing that approximations "from below" can be easily derived by applying the duality principle (as found e.g. in lattice theory).

### 2.6 Effective Abstractions

**Numerical Abstractions** Assume that a program has two integer variables `X` and `Y`. The trace semantics of the program (Fig. 1) can be abstracted in the static/collecting semantics (Fig. 5). A further abstraction consists in forgetting in a state all but the values $x$ and $y$ of variables `X` and `Y`. In this way the trace semantics is abstracted to a set of points (pairs of values), as illustrated in the plane by Fig. 6(a).

We now illustrate informally a number of effective abstractions of an [in]finite set of points.




(a) Octagonal Abstraction

(b) Polyhedral Abstraction

(c) Relational Congruence Abstraction

(d) Trapezoidal Congruence Abstraction

**Fig. 7.** Relational Abstractions

**Non-relational Abstractions** The non-relational, attribute independent or cartesian abstractions [46, example 6.2.0.2] consists in ignoring the possible relationships between the values of the `X` and `Y` variables. So a set of pairs is approximated through projection by a pair of sets. Each such set may still be infinite and in general not exactly computer representable. Further abstractions are therefore needed.

The *sign abstraction* [46] illustrated in Fig. 6(b) consists in replacing integers by their sign thus ignoring their absolute value. The *interval abstraction* [42] illustrated in Fig. 6(c) is more precise since it approximates a set of integers by it minimal and maximal values (including $-\infty$ and $+\infty$ as well as the empty set if necessary).

The *congruence abstraction* [82] (generalizing the parity abstraction [46]) is not comparable, as illustrated in Fig. 6(d).

**Relational Abstractions** Relational abstractions are more precise than non relational ones in that some of the relationships between values of the program states are preserved by the abstraction.

For example the *polyhedral abstraction* [60] illustrated in Fig. 7(b) approximates a set of integers by its convex hull. Only non-linear relationships between the values of the program variables are forgotten.

The use of an *octagonal abstraction* illustrated in Fig. 7(a) is less precise since only some shapes of polyhedra are retained or equivalently only linear relations between any two variables are considered with coefficients +1 or -1 (of the form $\pm x \pm y \le c$ where $c$ is an integer constant).

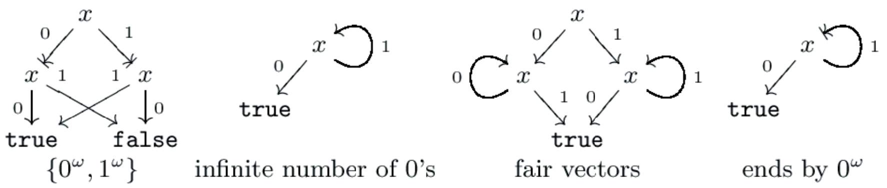



**Fig. 8.** Binary Decision Graphs

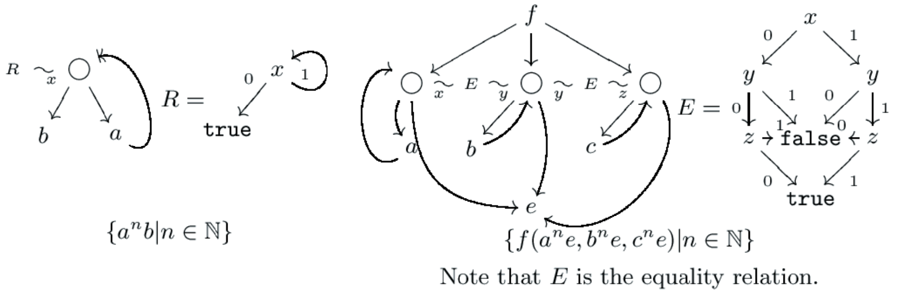



**Fig. 9.** Tree Schemata

A non comparable relational abstraction is the *linear congruence abstraction* [83] illustrated in Fig. 7(c).

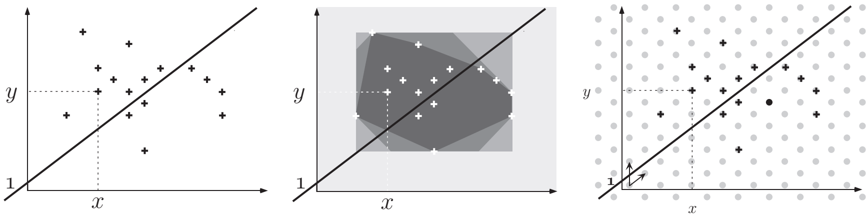
A combination of non-relational dense approximations (like intervals) and relational sparse approximations (like congruences) is the *trapezoidal linear congruence abstraction* [110] as illustrated in Fig. 7(d).

**Symbolic Abstractions** Most structures manipulated by programs are *symbolic structures* such as control structures (call graphs), data structures (search trees, pointers [66, 67, 136, 142]), communication structures (distributed & mobile programs [71, 87, 141]), etc. It is very difficult to find compact and expressive abstractions of such sets of objects (sets of languages, sets of automata, sets of trees or graphs, etc.). For example B chi automata or automata on trees are very expressive but algorithmically expensive.

A compromise between semantic expressivity and algorithmic efficiency was recently introduced by [111] using *Binary Decision Graphs* and *Tree Schemata* to abstract infinite sets of infinite trees as illustrated in Fig. 8 & 9.

### 2.7 Information Loss

Any abstraction introduces some loss of information. For example the abstraction of the trace semantics into relational or denotational semantics loses all information on the computation cost since all intermediate steps in the execution are removed.




(a) yes (b) unkown (c) yes

**Fig. 10.** Is `1/(X+1-Y)` well-defined?

All answers given by the abstract semantics are always correct with respect to the concrete semantics. For example, if termination is proved using the relational semantics then there is no execution abstracted to $\langle a, \perp\rangle$, so there is no infinite trace $\overset{a}{\bullet}$——$\overset{b}{\bullet}$——. . .——. . . in the trace semantics, whence non termination is impossible when starting execution in initial state $a$.

However, because of the information loss, not all questions can be definitely answered with the abstract semantics. For example, the natural semantics cannot answer questions about termination as can be done with the relational or denotational semantics. These semantics cannot answer questions about concrete computation costs.

The more concrete is the semantics, the more questions it can answer. The more abstract semantics are simpler. Non comparable abstract semantics (such as intervals and congruences) answer non comparable sets of questions.

To illustrate the loss of information, let us consider the problem of deciding whether the operation `1/(X+1-Y)` appearing in a program is always well defined at run-time. The answer can certainly be given by the concrete semantics since it has no point on the line $x + 1 - y = 0$, as shown in Fig. 10(a).

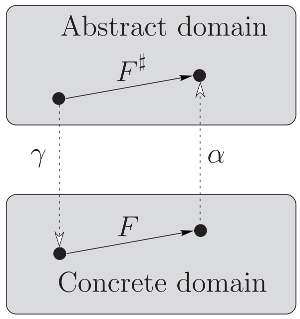
In practice the concrete abstraction is not computable so it is hardly usable in a useful effective tool. The dense abstractions that we have considered are too approximate as is illustrated in Fig. 10(b).

However the answer is positive when using the relational congruence abstraction, as shown in Fig. 10(c).

### 2.8 Function Abstraction

We now show how the abstraction of complex mathematical objects used in the semantics of programming or specification languages can be defined by composing abstractions of simpler mathematical structures.

For example knowing abstractions of the parameter and result of a monotonic function on sets, a function $F$ can be abstracted into an abstract function $F^\sharp$ as illustrated in Fig. 11 [46]. Mathematically, $F^\sharp$ takes its parameter $x$ in the abstract domain. Let $\gamma(x)$ be the corresponding concrete set ($\gamma$ is the adjoined, intuitively the inverse of the abstraction function $\alpha$). The function $F$ can be applied to get the concrete result $\circ\ F \circ \gamma(x)$. The abstraction function $\alpha$ can then be applied to approximate the result $F^\sharp(x) = \alpha \circ F \circ \gamma(x)$.
In general, neither $F$, $\alpha$ nor $\gamma$ are computable even though the abstraction $\alpha$ may be effective. So we have got a formal specification of the abstract function $F^\sharp$ and an algorithm has to be found for an effective implementation.




$$F^\sharp = \alpha \circ F \circ \gamma$$

**Fig. 11.** Function Abstraction

### 2.9 Fixpoint Abstraction

A fixpoint of a function $F$ can often be obtained as the limit of the iterations of $F$ from a given initial value $\perp$. In this case the abstraction of the fixpoint can often be obtained as the abstract limit of the iteration of the abstraction $F^\sharp$ of $F$ starting from the abstraction $\alpha(\perp)$ of the initial value $\perp$. The basic result is that the concretization of the abstract fixpoint is related to the concrete fixpoint by the approximation relation expressing the soundness of the abstraction [46]. This is illustrated in Fig. 12.

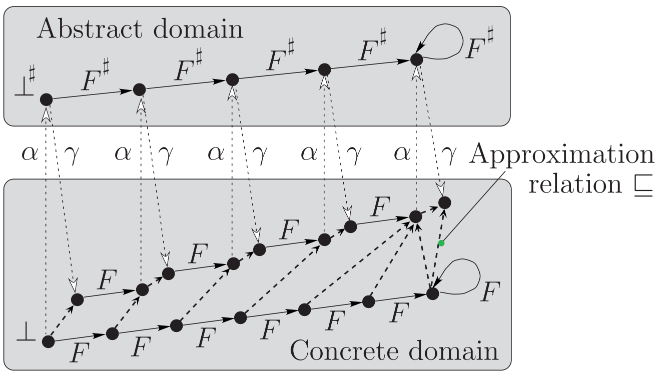



**Fig. 12.** Fixpoint Abstraction $\text{lfp}\, F \sqsubseteq \gamma(\text{lfp}\, F^\sharp)$

Often states have some finite component (e.g. a program counter) which can be used to partition into fixpoint system of equations by projection along that component. Then *chaotic* [44] and *asynchronous iteration strategies* [36] can be used to solve the equations iteratively. Various efficient iteration strategies have been studied, including ones taking particular properties of abstractions into account and others to speed up the convergence of the iterates [51].

### 2.10 Composing Abstractions

Abstractions hence abstract interpreters for static program analysis can be designed compositionally by stepwise abstraction, combination or refinement [75, 39].

An example of stepwise abstraction is the functional abstraction of Sec. 2.8. The abstraction of a function is parameterized by abstractions for the function parameters and the function result which can be chosen later in the modular design of the abstract interpreter.

An example of abstraction combination is the *reduced product* of two abstractions [46] which is the most abstract abstraction more precise than these two abstractions or the *reduce cardinal power* [46] generalizing case analysis. Such combination of abstract domains can be implemented as parameterized modules in static analyzer generators (e.g. [104]) so as to partially automate the design of expressive analyses from simpler ones.

An example of refinement is the *disjunctive completion* [46] which completes an abstract domain by adding concrete disjunctions missing in the abstract domain. Another example of abstract domain refinement is the *complementation* [34] adding concrete negations missing in the abstract domain.

### 2.11 Sound and Complete Abstractions

Abstract interpretation theory has mainly been concerned with the *soundness* of the abstract semantics/interpreter, relative to which questions can be answered correctly despite the loss of information [43]. Soundness is essential in practice and leads to a formal design method [46].

However *completeness*, relative to the formalization of the loss of information in a controlled way so as to answer a given set of questions, has also been intensively studied [46, 75], including in the context of model checking [40].

In practice complete abstractions, including a most abstract one, always exist to check that a given program semantics satisfies a given specification. Moreover any given abstraction can be refined to a complete one. Nevertheless this approach has severe practical limitations since, in general, the design of such complete abstractions or the refinement of a given one is logically equivalent to the design of an inductive argument for the formal proof that the given program satisfies the given specification, while the soundness proof of this abstraction logically amounts to checking the inductive verification conditions or proof obligations of this formal proof [40]. Such proofs can hardly be fully automated hence human interaction is unavoidable. Moreover the whole process has to be repeated each time the program or specification is modified.

Instead of considering such strong specifications for a given specific program, the objective of static program analysis is to consider (often predefined) specifications and all possible programs. The practical problem in static program analysis is therefore to design useful abstractions which are computable for all programs and expressive enough to yield interesting information for most programs.

## 3 Static Program Analysis

Static program analysis is the automatic static determination of dynamic runtime properties of programs.

### 3.1 Foundational Ideas of Static Program Analysis

Given a program and a specification, a program analyzer will check if the program semantics satisfies the specification (Fig. 13). In case of failure, the analyzer will provide hints to understand the origin of errors (e.g. by a backward analysis providing necessary conditions to be satisfied by counter-examples).

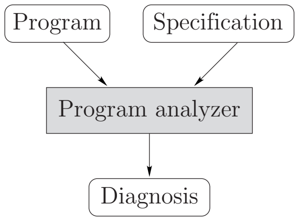



**Fig. 13.** Program Analysis

The principle of the analysis is to compute an approximate semantics of the program in order to check a given specification. Abstract interpretation is used to derive, from a standard semantics, the approximate and computable abstract semantics. The derivation can often be done by composing standard abstractions to fit a particular kind of information which has to be discovered about program execution. This derivation is itself not (fully) mechanizable but *static analyzer generators* such as PAG [108], GENA [69]and others can provide generic abstractions to be composed with problem specific ones.

In practice, the program analyzer contains a *generator* reading the program text and producing equations or constraints whose solution is a computer representation of the program abstract semantics. A *solver* is then used to solve these abstract equations/constraints. A popular resolution method is to use iteration. Of the numerical abstractions considered in Sec. 2.6 , only the sign and simple congruence abstractions ensure the finite convergence of the iterates. If the limit of the iterates is inexistent (which may be the case e.g. for the polyhedral abstraction) or it is reached after infinitely many iteration steps (e.g. interval and octagonal abstractions), the convergence may have to be ensured and/or accelerated using a *widening* to over estimate the solution in finitely many steps followed by a *narrowing* to improve it [36 , 43 , 51].

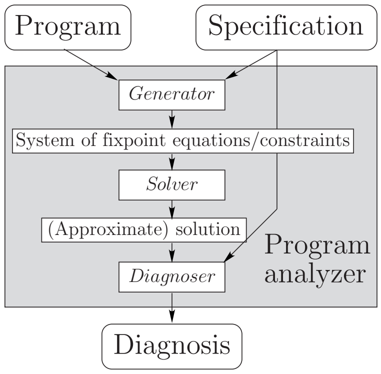



**Fig. 14.** Principle of Program Analysis

In *abstract compilation*, the generator and solver are directly compiled into a program which directly yields the approximate solution.

This solution is an approximation of the abstract semantics which is then used by a *diagnoser* to check the specification. Because of the loss of information, the diagnosis is always of the form "yes", "no", "unknown" or "irrelevant" (e.g. a safety specification for unreachable code). The general structure of program analyzers is illustrated in Fig. 14. Besides diagnosis, static program analysis is also used for other applications in which case the diagnoser is replaced by an *optimiser* (for compile-time optimization), a *program transformer* (for partial evaluation [98]), etc.

### 3.2 Shortcomings of Static Program Analysis

Static program analysis can be used for large programs (e.g. 220,000 lines of C) without user interaction. The abstractions are chosen to be of wide scope without specialization to a particular program. Abstract algebras can be designed and implemented into libraries which are reusable for different programming languages. The objective is to discover invariants that are likely to appear in many programs so that the abstraction must be widely reusable for the program analyzer to be of economic interest.

The drawback of this general scope is that the considered abstract specifications and properties are often simple, mainly concerning elementary safety properties such as absence of run-time errors. For example non-linear abstractions of sets of points are very difficult and very few mathematical results are of practical interest and directly applicable to program analysis. Checking termination and similar liveness properties is trivial with finite state systems, at least from a theoretical if not algorithmic point of view (e.g. finding loops in finite graphs). The same problem is much more difficult for infinite state systems because of fairness [111] or of potentially infinite data structures (as considered e.g. in partial evaluation) which do not amount to finite cycles so that termination or inevitability proofs require the discovery of *variant functions on well-founded sets* which is very difficult in full generality.

Even when considering restricted simple abstract properties, the semantics of real-life programming languages is very complex (recursion, concurrency, modularity, etc.) whence so is the corresponding abstract interpreter. The abstraction of this semantics, hence the design of the analyzer is mostly manual (and beyond the ability of casual programmers or theorem provers) whence costly. The considered abstractions must have a large scope of application and must be easily reusable to be of economic interest.

From a user point of view, the results of the analysis have to be presented in a simple way (for example by pointing at errors only or by providing abstract counter-examples, or less frequently concrete ones). Experience shows that the cases of uncertainty represent 5 to 10 % of the possible cases. They must be handled with other empirical or formal methods (including more refined abstract interpretations).

### 3.3 Applications of Static Program Analysis

Among the numerous applications of static program analysis, let us cite *data flow analysis* [130 , 57]; *program optimization and transformation* (including partial evaluation and program specialization [98] and data dependence analysis for the parallelisation of sequential languages); *set-based analysis* [55]; *type inference* [38] (including undecidable systems and soft typing); verification of *reactive* [86 , 93], *real-time* and *(linear) hybrid systems* including state space reduction; *cryptographic protocol* analysis; *abstract model-checking* of infinite systems [57]; *abstract debugging*, testing and verification ; *cache and pipeline behavior prediction* [70]; *probabilistic analysis* [115]; *communication topology analysis* for mobile/distributed code [71 , 87 , 141]; *automatic differentiation* of numerical programs; *abstract simulation* of temporal specifications; Semantic tattooing/*watermarking* of software [59]; etc.

Static program analysis has been intensively studied for grammars and polynomial systems [37], term graph rewriting [79], sequent calculi [2], typesetting languages [96], procedural languages [16 , 45] (for alias analysis [142], pointer analysis [66, 67], parameter boxing/unboxing [80], copy elimination [131], dependence analysis [109], exception analysis [129], constant propagation [102], (linear) equality or inequality relationships analysis [60] etc.), parallel procedural languages [61 , 78], functional languages (for binding time analysis [140], strictness analysis [21 , 53 , 117], inverse image analysis [68], projection analysis [20], comportment analysis [54], dependency analysis [12], path/trace analysis [32], closure analysis [120], control flow analysis [134], value flow analysis [13], compile-time garbage collection [97], stackability and escape analysis [10], data structures and abstract data type analysis [106], heap shape analysis [99 , 136], exception analysis [143], polymorphic function analysis [4], kind/sort analysis [84], typing [38], effect systems [100], termination analysis [121], time complexity analysis [127], parallelization [138], etc.), parallel functional languages [62], data parallel languages [25], logic languages including Prolog [49 , 64] (for mode [112] and type analysis [95] and their combination [17], finiteness analysis [8], relational argument size analysis [132], dependency analysis [116], detecting determinate/functional computations [76], mutually exclusive rules detection [124], occur check reduction [135], WAM code optimization [3], copy avoidance [74], groundness analysis [35], sharing analysis [33], freeness analysis [30] and their combinations [31], termination analysis [101], time complexity and cost analysis [65], parallelisation [18], etc.) including its search rule and the cut [73] and database programming languages [1], concurrent logic languages [22], functional logic languages [88], constraint logic languages [9], concurrent constraint logic languages [144], specification languages [77], synchronous languages [86] (such as LUSTRE [24]), concurrent/parallel languages [48], communicating and distributed languages [47, 113] and more recently object-oriented languages [11].

Abstract interpretation based static program analyses have been used for the static analysis of the embedded ADA software of the Ariane 5 launcher[1] and the ARD[2] [103]. The static program analyser aims at the automatic detection of the *definiteness*, *potentiality*, *impossibility* or *inaccessibility* of run-time errors such as scalar and floating-point overflows, array index errors, divisions by zero and related arithmetic exceptions, uninitialized variables, data races on shared data structures, etc. The analyzer was able to automatically discover the Ariane 501 flight error. The static analysis of embedded safety critical software (such as avionic software [126]) is very promising [58].

### 3.4 Industrialization of Static Analysis by Abstract Interpretation

The impressive results obtained by the static analysis of real-life embedded critical software [103, 126] is quite promising for the industrialization of abstract interpretation.

This is the explicit objective of AbsInt Angewandte Informatik GmbH (`www.absint.com`) created in Germany by R. Wilhelm and C. Ferdinand in 1998 commercializing the program analyzer generator PAG and an application to determine the worst-case execution time for modern computer architectures with memory caches, pipelines, etc [70].

Polyspace Technologies (`www.polyspace.com`) was created in France by A. Deutsch and D. Pilaud in 1999 to develop and commercialize ADA and C program analyzers.

Other companies like Connected Components Corporation (`www.concmp.com`) created in the U.S.A. by W.L. Harrison in 1993 use abstract interpretation internally e.g. for compiler design [92].

## 4 Abstract Formal Methods

No automatic formal method can ultimately find all errors in a software system nor can their combinations. We will briefly review the automatic formal methods for computer-aided program verification, discussing their principles, advantages and shortcomings. Since static program analysis has already been discussed, we now consider typing, model-checking, deductive methods and their combination.

### 4.1 Typing

Polymorphic typing and type inference [114] was a definite step in the design of programming languages and compilers [91]. The question for the next decade seems to be to scale to more expressive properties.

[1] Flight software (60,000 lines of Ada code) and Inertial Measurement Unit (30,000 lines of Ada code).

[2] Atmospheric Reentry Demonstrator.

**Foundational Ideas of Typing** Typing is based on decidable program analyses. This approach is always possible by restricting both on specifications (allowed types) and on programs, as shown when considering types as abstract interpretations [38]. In theory, type systems have a clean presentation of the type analysis (inference algorithm [114]) through an equivalent logical formal system (type verification [63]). Monomorphic typing [94] was extended to polymorphism [114], complex data structures, references [89], exceptions and separate modules [90] in a way that scales up for very large programs. It is nicely integrated in the compiler and the certification can go down to the generated code (proof-carrying code [118], certified compiler [137]).

**Shortcomings of Typing** Type systems (e.g. with subtle subtyping) can be very complex to understand for the casual user. One difficulty is that typing is compositional but not fully abstract (e.g. the same polymorphic code can type differently in different utilization contexts). The interaction with the user is often crude (no hint is given to understand why wrong programs do not type well). It is hardly possible for the user to provide hints to help the typing process. The logical specification of the type system is often inexistent in the reference manual, not equivalent to the type inference algorithm or so inextricable that it is useless both to the programmer and the compiler designer. The programs considered in type theory are both complex (higher-order modules) and too restricted (mainly functional languages). The most severe restrictions are on the considered properties (arithmetic, out of range array indexing, null pointer dereferencing, … errors are checked at run-time, all liveness properties are ignored). These restrictions and the difficulty to generalize to more expressive properties mainly follow from the encoding of types as terms/formulæ and from the one iterate fixpoint approximation.

### 4.2 Deductive Methods

**Foundational Ideas of Deductive Methods** Deductive methods use a (manually designed abstraction of) the program semantics to obtain minimal verification conditions to prove program correctness. These verification conditions can be derived from the program trace semantics by abstract interpretation [41]. Then a theorem prover [119] or a proof assistant [122] is used to check the verification conditions.

**Shortcomings of Deductive Methods** Deductive methods use the schema of Fig. 14 but for the fact that the solver is replaced by a verifier or checker thus avoiding fixpoint computations. So the constraints or equations corresponding to the verification conditions are not solved. This means that an inductive argument (e.g. invariant, variant function) has to be provided, generally by the user. Since the implication involved in the verification condition is itself undecidable, the proof verification can only be partially automatized, even though the solution to the equations/constraints is provided. Therefore interaction of the programmer with the prover is ultimately needed. This (wo)man/prover interaction is hard if not despairing, in particular because the size of the proof is often exponential in the program size. Therefore debugging an unsuccessful proof (because of a program error or a prover weakness) can be as complex as (if not much more complex than) debugging the program itself.

An alternative [105] consists in restricting the form of predicates considered by the prover, (which is an abstract interpretation [46, Sec. 5]). This can go up to unsound verification condition simplifications, essentially to make the verifier simpler (e.g. modular arithmetic).

Because theorem provers are driven by unformalized heuristics, and these heuristics and their interactions are changed over time for improving proof strategies, theorem provers are often unstable over time (e.g. proof strategies get changed so that old proofs no longer work). Another weakness which makes interaction with other formal methods somewhat difficult is the uniform encoding of properties as syntactical terms/formulæ (so that e.g. BBDs are hardly efficiently encodable). It follows that the theorem prover has ultimately to be extended with program analyzers, model checkers, typing, among others [133], often without supporting theory, in particular for mechanizing and combining abstractions.

### 4.3 Model Checking

Model checking [26, 125] has been very successful for the verification of hardware [7], communication protocols [29], cryptographic protocols [5], and real-time [23] or probabilistic [139] processes. As far as software systems are concerned, the question for the next decade is whether model checking can be extended to the verification of very large real-life programs.

**Foundational Ideas of Model Checking** First a model of the program (i.e. manually designed abstraction of the program semantics) must be designed (in the form of a transition system similar to a small step operational semantics). Then a specification of the program must be provided by the user in a very expressive temporal logic [123]. A model checker can then check the specification by exhaustive search/symbolic exploration of the state space.

The spectacular success of model checking followed from the clever design of data structures (e.g. BDDs or QDDs [14]) and algorithms (e.g. minimal state graph generation [15], fixpoint computation [107] or SAT [6]) for representing very large sets of booleans and their transformations.

The approximation is that the model must be finite-state or some form of abstract interpretation must be used [28, 81] to reduce the verification problem to finite state, including symmetries [27], etc. Also clever semantics of concurrent systems have been considered, e.g. to avoid the combinatorial explosion of interleaving [29].

Another trend in infinite-state model checking is to consider safety properties only and polyhedral abstractions, with variants (e.g. Presburger arithmetic [19]).

This is a direct application of polyhedral static program analysis [60], including the use of widenings. This allows e.g. for the analysis of reactive [72], real-time [85] and hybrid systems [86].

**Shortcomings of Model Checking** Although model checking gained a factor of 100 in 10 years, it is very difficult to scale up because of the state explosion problem. So, the necessary restriction to available computer resources often reduces the model checker from formal verification to debugging on part of the state space. Since the model must ultimately be finite (to allow for exhaustive search/symbolic exploration), abstraction is mandatory, which is a very difficult task to do manually and/or is left informal. Moreover, some forms of abstractions (such as interval [43] or polyhedral [60] abstractions) do not abstract concrete transition systems into abstract transition systems so that the model checker may not be reusable in the abstract. One can use abstraction for model checking which are complete in that there always exists a program specific abstraction into a finite model to prove a given specification correct (see [40] for safety properties) but none will be complete for all programs, even for simple properties as considered in static program analysis [51]. It follows that complete abstractions are difficult and not reusable, hence not cost effective.

## 5 Combining Program Verification Methods

Since no single formal method can ultimately solve the verification problem, a current trend is to combine formal methods.

For example, one can rely on a user designed abstraction and derive a finite abstract model of the program semantics by abstract interpretation, prove the correctness of the abstraction by deductive methods and later verify the abstract model by model-checking [128].

A fundamental limitation [40] is that the abstraction discovery and the derivation of the abstract semantics are respectively logically equivalent hence practically as difficult as invariant discovery and invariant verification in a formal proof. So we have the feeling that the combination of tools might simplify formal proofs but still will ultimately not solve the program verification problem.

## 6 Combining Empirical and Formal Methods

Formal methods have made a lot of progress in the last decade. Nevertheless there are few automatic light weight tools to apply them in practice. Integration of such tools is difficult and cannot ultimately solve all verification problems.

It follows that the only mechanical tool for verifying programs is still *debugging*, despite its well-known defaults, incompleteness and cost. There again progress was slow, in particular because theory never took debugging seriously. The main advantage of debugging is that a debugger is a light weight tool which is very easily understood by all programmers. Because of its well-known incredible cost for weak results, debugging may not scale up in the next decade for very large or safety critical software.

An alternative which still remains to be investigated is the combination of informal methods like debugging with verification tools. Let us consider for example *abstract testing* [56].

The classical debugging methodology consists in running the program on test data, checking if the execution satisfies informal specifications. This process is repeated by providing more tests until reaching a satisfactory coverage.

By an easily understandable analogy, the abstract testing methodology [56] consists in computing the abstract semantics for a finitary or infinitary abstraction chosen by the programmer among a predefined palette (not user defined, which would be too difficult). The abstract semantics is then checked against user-provided abstract assertions or the abstraction of a formal specification. This process is repeated with more refined abstractions until enough assertions are proved or no predefined abstraction can do.

Observe that one can prove the absence of (some categories of) bugs, not only their presence. Moreover, abstract evaluation can range from an analogy to program execution to the application of proof methods (using e.g. forward as well as backward reasonings providing abstract counter-examples) without attempting to make a one-shot complete formal proof of the specification.

## 7 Conclusions on the Past Decade

Full program verification by formal methods (e.g. model checking/deductive methods), which requires user interaction (for discovering an abstraction or inductive argument) is very costly in human resources, hence is not likely to scale up for very large software. Abstraction is mandatory for program verification, but difficult, hardly automatizable and beyond the common capabilities of most programmers.

Partial program verification by static analysis (with typing being considered as a particular and successful case) is *cost-effective*[3] because no user intervention is mandatory for performing the analysis and universal abstractions are reusable, hence commercializable.

For large and complex programs, complete verification by formal methods is not likely to be viable at low cost. Program debugging is still and will probably remain for some time the prominent industrial program “verification” method.

In this context, abstract interpretation based static program analysis can be extended to *abstract program testing*. Abstract interpretation based methods offer powerful techniques which, in the presence of approximation, can be viable alternatives or complements both to the exhaustive search of model-checking and to the partial exploration methods of classical debugging.

[3] e.g. less than 0.25$ per program line costing 50 to 80$.

## 8 Grand Challenge for the Next Decade

We believe that in the next decade the software industry will certainly have to face its responsibility imposed by a computer-dependent society, in particular for safety critical systems. Consequently, *Software reliability*[4] will be a grand challenge for computer science and practice.

The grand challenge for formal methods, in particular abstract interpretation based formal tools, is both the large scale industrialization and the intensification of the fundamental research effort.

General-purpose, expressive and cost-effective abstractions have to be developed e.g. to handle floating point numbers, data dependences (e.g. for parallelization), liveness properties with fairness (to extend finite-state model-checking to software), timing properties for embedded software, probabilistic properties, etc. Present-day tools will have to be enhanced to handle higher-order compositional modular analyses and to cope with new programming paradigms involving complex data and control concepts (such as objects, concurrent threads, distributed/mobile programming, etc.), to automatically combine and locally refine abstractions in particular to cope with "unknow" answers, to interact nicely with users and other formal or informal methods.

The most challenging objective might be to integrate formal analysis by abstract interpretation in the full software development process, from the initial specifications to the ultimate program development.

**Acknowledgements** I thank Radhia Cousot and Reinhard Wilhelm for their comments on a preliminary version of this paper. This work was supported by the DAEDALUS [58] and TUAMOTU [59] projects.

## References

1. G. Amato, F. Giannotti, and G. Mainetto. Data sharing analysis for a database programming language via abstract interpretation. In R. Agrawal, S. Baker, and D.A. Bell, editors, *Proc. 19th Int. Conf. VLDB '93*, Dublin, IE, pages 405–415. Morgan Kaufmann Pub., 24–27 Aug. 1993.
2. G. Amato and G. Levi. Abstract interpretation based semantics of sequent calculi. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 38–57. Springer-Verlag, 29 June – 1 Jul. 2000.
3. D. Baldan and G. Filé. Abstract interpretation for improving WAM code. In P. Van Hentenryck, editor, *Proc. 4th Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, page 364. Springer-Verlag, 1997.
4. G. Baraki and R.J.M. Hughes. Abstract interpretation of polymorphic functions. In K. Davis and J. Hughes, editors, *Functional Programming, Glasgow 1989*, Proc. 1989 Glasgow Workshop, Fraserburgh, UK. Springer-Verlag and BCS, 31–40 Aug. 1989.

[4] other suggestions were "trustworthiness" (C. Jones) and "robustness" (R. Leino).

5. A. Biere. $\mu$cke - efficient $\mu$-calculus model checking. In O. Grumberg, editor, *Proc. 9$^{th}$ Int. Conf. CAV '97*, Haifa, IL, LNCS 1254, pages 468–471. Springer-Verlag, 22–25 Jul. 1997.
6. A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36$^{th}$ Conf. DAC '99*, New Orleans, LA, US, pages 317–320. ACM Press, 21–25 June 1999.
7. A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu. Properties of a power PC microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *Proc. 11$^{th}$ Int. Conf. CAV '99*, Trento, IT, LNCS 1633, pages 60–71. Springer-Verlag, 6–10 Jul. 1999.
8. P.A. Bigot, S.K. Debray, and K. Marriott. Understanding finiteness analysis using abstract interpretation. In K.R. Apt, editor, *Proc. JICSLP '92*, Washington, DC, US, pages 735–749. MIT Press, Nov. 1992.
9. S. Bistarelli, P. Codognet, and F. Rossi. An abstraction framework for soft constraints and its relationship with constraint propagation. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4$^{th}$ Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 71–86. Springer-Verlag, 26–29 Jul. 2000.
10. B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *25$^{th}$ POPL*, pages 25–37, San Diego, CA, US, 19–21 Jan. 1998. ACM Press.
11. B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proc. ACM SIGPLAN Conf. OOPSLA '99. ACM SIGPLAN Not. 34(10)*, pages 20–34, Denver, CO, US, 1–5 Nov. 1999.
12. M. Blume. Dependency analysis for Standard ML. *TOPLAS*, 21(4):790–812, Jul. 1999.
13. R. Bod k and S. Anik. Path-sensitive value-flow analysis. In *25$^{th}$ POPL*, pages 237–251, San Diego, CA, US, 19–21 Jan. 1998. ACM Press.
14. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs (extended abstract). In P. Van Hentenryck, editor, *Proc. 4$^{th}$ Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 172–186. Springer-Verlag, 1997.
15. A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Sci. Comput. Programming*, 18:247–269, 1992.
16. F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In P. Deransart and J. Małuszyński, editors, *Proc. Int. Work. PLILP '90*, Link ping, SE, LNCS 456, pages 307–323. Springer-Verlag, 20–22 Aug. 1990.
17. M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing (extended abstract). In R. Kowalski and K. Bowen, editors, *Proc. 5$^{th}$ Int. Conf. & Symp. on Logic Programming, Volume 1*, Seattle, WA, US, pages 669–683. MIT Press, 15–19 Aug. 1988.
18. F. Bueno, M.J. Garc a de la Banda, and M.V. Hermenegildo. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *TOPLAS*, 21(2):189–239, Mar. 1999.
19. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In O. Grumberg, editor, *Proc. 9$^{th}$ Int. Conf. CAV '97*, Haifa, IL,LNCS 1254, pages 400–411. Springer-Verlag, 22–25 Jul. 1997.
20. G.L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *17$^{th}$ POPL*, pages 151–156, San Francisco, CA, 1990. ACM Press.
21. G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis of higher-order functions. *Sci. Comput. Programming*, 7:249–278, Nov. 1986.

22. M.-M. Corsini C. Codognet, P. Codognet. Abstract interpretation for concurrent logic languages. In S.K. Debray and M.V. Hermenegildo, editors, *NACLP 1997*, Austin, TX, US, pages 215–232. MIT Press, 29 Oct. – 1 Nov. 1990.
23. S.V.A. Campos, E.M. Clarke, and M. Minea. The Verus tool: A quantitative approach to the formal verification of real-time systems. In O. Grumberg, editor, *Proc. 9$^{th}$ Int. Conf. CAV '97*, Haifa, IL,LNCS 1254, pages 452–455. Springer-Verlag, 22–25 Jul. 1997.
24. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14$^{th}$ POPL*, Munchen, DE, 1987. ACM Press.
25. S. Chatterjee, B.E. Blelloch, and A.L. Fisher. Size and access inference for data-parallel programs. In *Proc. ACM SIGPLAN '91 Conf. PLDI. ACM SIGPLAN Not. 26(6)*, pages 130–144, Toronto, Ontario, CA, 26–28 June 1991.
26. E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *IBM Workshop on Logics of Programs*, Yorktown Heights, NY, US, LNCS 131. Springer-Verlag, May 1981.
27. E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In A.J. Hu and M.Y. Vardi, editors, *Proc. 10$^{th}$ Int. Conf. CAV '98*, Vancouver, BC, CA,LNCS 1427, pages 147–158. Springer-Verlag, 28 June – 2 Jul. 1998.
28. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *19$^{th}$ POPL*, pages 343–354, Albuquerque, NM, 1992. ACM Press.
29. E.M. Clarke, S. Jha, and W.R. Marrero. Partial order reductions for security protocol verification. In S. Graf and M.I. Schwartzbach, editors, *Proc. 6$^{th}$ Int. Conf. TACAS '2000*, Berlin, DE, 25 Mar. – 2 Apr. 2000, LNCS 1785, pages 503–518. Springer-Verlag, 2000.
30. M. Codish, D. Dams, G. Fil , and M. Bruynooghe. Freeness analysis for logic programs – and correctness? In D.S. Warren, editor, *Proc. 10$^{th}$ ICLP '93*, Budapest, HU, pages 116–131. MIT Press, 21–25 June 1993.
31. M. Codish, H. Søndergaard, and P.J. Stuckey. Sharing and groundness dependencies in logic programs. *TOPLAS*, 21(5):948–976, Sep. 1999.
32. C. Colby and P. Lee. Trace-based program analysis. In *23$^{rd}$ POPL*, pages 195–207, St. Petersburg Beach, FL, 1996. ACM Press.
33. A. Cortesi and G. Filé. Sharing is optimal. *J. Logic Programming*, 38(3):371–386, 1999.
34. A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *TOPLAS*, 19(1):7–47, Jan. 1997.
35. A. Cortesi, G. Filé, and W.H. Winsborough. Optimal groundness analysis using propositional logic. *J. Logic Programming*, 27(2):137–167, 1996.
36. P. Cousot. *M thodes it ratives de construction et d'approximation de points fixes d'op rateurs monotones sur un treillis, analyse s mantique de programmes.* Th se d'État s sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, FR, 21 Mar. 1978.
37. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997. `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages.
38. P. Cousot. Types as abstract interpretations, invited paper. In *24$^{th}$ POPL*, pages 316–331, Paris, FR, Jan. 1997. ACM Press.
39. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbr ggen, editors, *Calculational System Design*, volume 173, pages

421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, 1999.

40. P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. $4^{th}$ Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 1–25. Springer-Verlag, 26–29 Jul. 2000.
41. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, To appear (Preliminary version in [37]).
42. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. $2^{nd}$ Int. Symp. on Programming*, pages 106–130. Dunod, 1976.
43. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *$4^{th}$ POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.
44. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, 1977.
45. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.
46. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *$6^{th}$ POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.
47. P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. In J.W. de Bakker and J. van Leeuwen, editors, *$7^{th}$ ICALP*, LNCS 85, pages 119–133. Springer-Verlag, Jul. 1980.
48. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, 1984.
49. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Programming*, 13(2–3):103–179, 1992. (The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).
50. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, Aug. 1992.
51. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. $4^{th}$ Int. Symp. PLILP '92*, Leuven, BE, 26–28 Aug. 1992, LNCS 631, pages 269–295. Springer-Verlag, 1992.
52. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *$19^{th}$ POPL*, pages 83–94, Albuquerque, NM, 1992. ACM Press.
53. P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis, invited paper. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pages 98–127. Springer-Verlag, 28 June – 2 Jul. 1993.
54. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL*, pages 95–112, Toulouse, FR, 16–19 May 1994. IEEE Comp. Soc. Press.

55. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. $7^{th}$ FPCA*, pages 170–181, La Jolla, CA, 25–28 June 1995. ACM Press.
56. P. Cousot and R. Cousot. Abstract interpretation based program testing, invited paper. In *Proc. SSGRR 2000 Computer & eBusiness International Conference*, Compact disk paper 248, L'Aquila, IT, 31 Jul. – 6 Aug. 2000. Scuola Superiore G. Reiss Romoli.
57. P. Cousot and R. Cousot. Temporal abstract interpretation. In *$27^{th}$ POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.
58. P. Cousot, R. Cousot, A. Deutsch, C. Ferdinand, É. Goubault, N. Jones, D. Pilaud, F. Randimbivololona, M. Sagiv, H. Seidel, and R. Wilhelm. DAEDALUS: Validation of critical software by static analysis and abstract testing. Project IST-1999-20527 of the european $5^{th}$ Framework Programme (FP5), Oct. 2000 – Oct. 2002.
59. P. Cousot, R. Cousot, and M. Riguidel. TUAMOTU: Tatouage électronique sémantique de code mobile Java. Project RNRT 1999 n° 95, Oct. 1999 – Oct. 2001.
60. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *$5^{th}$ POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.
61. R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. PEPM '95*, La Jolla, CA, 21–23 June 1995. ACM Press.
62. R. Cridlig and É. Goubault. Semantics and analysis of Linda-based languages. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Proc. $3^{rd}$ Int. Work. WSA '93*, Padova, IT, LNCS 724, pages 72–86. Springer-Verlag, 22–24 Sep. 1993.
63. L. Damas and R. Milner. Principal type-schemes for functional programs. In *$9^{th}$ POPL*, pages 207–212, Albuquerque, NM, Jan. 1982. ACM Press.
64. S.K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, Int. Schools for Computer Scientists, section 3, pages 115–182. Clarendon Press, 1994.
65. S.K. Debray, P. López-Garc a, M.V. Hermenegildo, and N.-W. Lin. Lower bound cost estimation for logic programs. In J. Małuszyński, editor, *Proc. Int. Symp. ILPS '1997*, Port Jefferson, Long Island, NY, US, pages 291–305. MIT Press, 13–16 Oct. 1997.
66. A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers, invited paper. In *Proc. PEPM '95*, pages 226–229, La Jolla, CA, 21–23 June 1995. ACM Press.
67. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In J. Palsberg, editor, *Proc. $7^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 115–134. Springer-Verlag, 29 June – 1 Jul. 2000.
68. P. Dybjer. Inverse image analysis generalises strictness analysis. *Inform. and Comput.*, 90:194–216, 1991.
69. C. Fecht. GENA – a tool for generating Prolog analyzers from specifications. In A. Mycroft, editor, *Proc. $2^{nd}$ Int. Symp. SAS '95*, Glasgow, UK, 25–27 Sep. 1995, LNCS 983, pages 418–419. Springer-Verlag, 1995.
70. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Programming, Special Issue on SAS'96*, 35(1):163–189, September 1999.

71. J. Feret. Confidentiality analysis of mobile systems. In J. Palsberg, editor, *Proc. 7$^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 135–154. Springer-Verlag, 29 June – 1 Jul. 2000.
72. J.-C. Fernandez. Abstract interpretation and verification of reactive systems. In P. Cousot, P. Falaschi, G. Filé, and A. Rauzy, editors, *Proc. 3$^{rd}$ Int. Work. WSA '93*, Padova, IT, LNCS 724, pages 60–71. Springer-Verlag, 22–24 Sep. 1993.
73. G. Filé and S. Rossi. Static analysis of Prolog with cut. In A. Voronkov, editor, *Proc. 4$^{th}$ Int. Conf. LPAR '93*, pages 134–145, St. Petersburg, RU, LNCS 698, 13–20 Jul. 1993. Springer-Verlag.
74. I.T. Foster and W.H. Winsborough. Copy avoidance through compile-time analysis and local reuse. In K. Ueda V.A. Saraswat, editor, *Proc. 1991 Int. Symp. ISLP '91*, San Diego, CA, US, pages 455–469. MIT Press, 28 Oct. – 1 Nov. 1997.
75. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
76. R. Giacobazzi and L. Ricci. Detecting determinate computations by bottom-up abstract interpretation. In B. Krieg-Br ckner, editor, *Proc. 4$^{th}$ ESOP '92*, Rennes, FR, LNCS 582, pages 167–181. Springer-Verlag, 26–28 Feb. 1992.
77. F. Giannotti and D. Latella. Gate splitting in LOTOS specifications using abstract interpretation. *Sci. Comput. Programming*, 23((2-3)):127–149, 1994.
78. É. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proc. PEPM '95*, La Jolla, CA, pages 134–145. ACM Press, 21–23 June 1995.
79. É. Goubault and C. Hankin. A lattice for the abstract interpretation of term graph rewriting systems. In R. Sleep, R. Plasmeijer, and van M. Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 10, pages 131–140. Wiley & S., 1993.
80. J. Goubault. Generalized boxings, congruences and partial inlining. In B. Le Charlier, editor, *Proc. 1$^{st}$ Int. Symp. SAS '94*, Namur, BE, 20–22 Sep. 1994, LNCS 864, pages 147–161. Springer-Verlag, 1994.
81. S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In C. Courcoubetis, editor, *Proc. 5$^{th}$ Int. Conf. CAV '93*, Elounda, GR, LNCS 697, pages 71–84. Springer-Verlag, 28 June –1 Jul. 1993.
82. P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.
83. P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. Int. J. Conf. TAPSOFT '91, Volume 1 (CAAP '91)*, Brighton, GB, LNCS 493, pages 169–192. Springer-Verlag, 1991.
84. C.A. Gunter, E.L. Gunter, and D.B. MacQueen. Computing ML equality kinds using abstract interpretation. *Inform. and Comput.*, 107(2):303–323, Dec. 1993.
85. N. Halbwachs. Delays analysis in synchronous programs. In C. Courcoubatis, editor, *Proc. 5$^{th}$ Int. Conf. CAV '93*, Elounda, GR, LNCS 697, pages 333–346. Springer-Verlag, 28 June –1 Jul. 1993.
86. N. Halbwachs. About synchronous programming and abstract interpretation. *Sci. Comput. Programming*, 31(1):75–89, May 1998.
87. R.R. Hansen, J.G. Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. In A. Cortesi and G. Filé, editors, *Proc. 6$^{th}$ Int. Symp. SAS '99*, Venice, IT, 22–24 Sep. 1999, LNCS 1694, pages 134–138. Springer-Verlag, 1999.

88. M. Hanus. Towards the global optimization of functional logic programs. In P.A. Fritzson, editor, *Proc. 5th Int. Conf. CC '94*, Edinburg, UK, LNCS 786, pages 68–82. Springer-Verlag, Apr. 1994.
89. R. Harper. A simplified account of polymorphic references. *Inf. Process. Lett.*, 54(4):201–206, 1994.
90. R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. Int. J. Conf. TAPSOFT '87, Volume 2 (AFISD/CFLP)*, Pisa, IT, LNCS 250, pages 308–319. Springer-Verlag, 23–27 Mar. 1987.
91. R. Harper and J.C. Mitchell. On the type structure of Standard ML. *TOPLAS*, 15(2):211–252, 1993.
92. W.L. Harrison. Can abstract interpretation become a main stream compiler technology? (abstract). In P. Van Hentenryck, editor, *Proc. 4th Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, page 395. Springer-Verlag, 1997.
93. T.A. Henzinger, R. Majumbar, F. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In J. Palsberg, editor, *Proc. 7th Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 220–239. Springer-Verlag, 29 June – 1 Jul. 2000.
94. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.
95. K. Horiuchi and T. Kanamori. Polymorphic type inference in Prolog by abstract interpretation. In K. Furukawa, H. Tanaka, and T. Fujisaki, editors, *Proc. 6th Conf. on Logic Programming '87*, Tokyo, JP, LNCS 315, pages 195–214. Springer-Verlag, June 1987.
96. N.R. Horspool and J. Vitek. Static analysis of PostScript code. *Comput. Lang.*, 19(2):65–78, 1993.
97. S. Hughes. Compile-time garbage collection for higher-order functional languages. *J. Logic and Comp.*, 2(4):483–464, Aug. 1992.
98. N.D. Jones. Combining abstract interpretation and partial evaluation (brief overview). In P. Van Hentenryck, editor, *Proc. 4th Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 396–405. Springer-Verlag, 1997.
99. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of LISP-like structures. In *6th POPL*, pages 244–256, San Antonio, TX, 1979. ACM Press.
100. P. Jouvelot and D.K. Gifford. Algebraic reconstruction of types and effects. In *18th POPL*, pages 303–310, Orlando, FL, 1991. ACM Press.
101. T. Kanamori, K. Horiuchi, and T. Kawamura. Detecting termination of logic programs based on abstract hybrid interpretation. Tech. rep. 398, ICOT, Tokyo, JP, 1987.
102. G. Kildall. A unified approach to global program optimization. In *1st POPL*, pages 194–206, Boston, MA, Oct. 1973. ACMpress.
103. P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The Ariane 5 example. In *Proceedings DASIA 98 – DAta Systems In Aerospace*, Athens, GR. ESA Publications, SP-422, 25–28 May 1998.
104. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. In *Proc. 1992 ICCL*, Oakland, CA, pages 137–146. IEEE Comp. Soc. Press, 20–23 Apr. 1992.
105. K.R.M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Proc. 7th Int. Conf. CC '98*, Lisbon, PT, LNCS 1383, pages 302–305. Springer-Verlag, 28 Mar. – 4 Apr. 1998.

106. Y.A. Liu and S.D. Stroller. Eliminating dead code on recursive data. In A. Cortesi and G. Filé, editors, *Proc. 6<sup>th</sup> Int. Symp. SAS '99*, Venice, IT, 22–24 Sep. 1999, LNCS 1694, pages 179–193. Springer-Verlag, 1999.
107. D.E. Long, A. Browne, E.M. Clarke, S. Jha, and W.R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. *Theoret. Comput. Sci.*, 178(1-2):237–255, 1997.
108. F. Martin. *Generating Program Analyzers*. Pirrot Verlag, Saarbr cken, DE, 1999.
109. F. Masdupuy. Using abstract interpretation to detect array data dependencies. In *Proc. Int. Symp. on Supercomputing*, pages 19–27, Fukuoka, JP, Nov. 1991. Kyushu U. Press.
110. F. Masdupuy. Semantic analysis of interval congruences. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Akademgorodok, Novosibirsk, RU, LNCS 735, pages 142–155. Springer-Verlag, 28 June – 2 Jul. 1993.
111. L. Mauborgne. Tree schemata and fair termination. In J. Palsberg, editor, *Proc. 7<sup>th</sup> Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 302–321. Springer-Verlag, 29 June – 1 Jul. 2000.
112. C.S. Mellish. Abstract interpretation of Prolog programs. In E. Shapiro, editor, *3<sup>rd</sup> ICLP '86*, London, GB, LNCS 225, pages 463–474. Springer-Verlag, 14–18 Jul. 1986.
113. N. Mercouroff. An algorithm for analyzing communicating processes. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proc. 7<sup>th</sup> Int. Conf. on Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, pages 312–325. Springer-Verlag, 25–28 Mar. 1991.
114. R. Milner. A theory of polymorphism in programming. *J. Comput. System Sci.*, 17(3):348–375, Dec. 1978.
115. D. Monniaux. Abstract interpretation of probabilistic semantics. In J. Palsberg, editor, *Proc. 7<sup>th</sup> Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 322–339. Springer-Verlag, 29 June – 1 Jul. 2000.
116. K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Programming*, 13(2–3):315–347, Jul. 1992.
117. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, Edinburg, UK, Dec. 1981.
118. G.C. Necula. Proof-carrying code. In *24<sup>th</sup> POPL*, pages 106–119, Paris, FR, Jan. 1997. ACM Press.
119. S. Owre, N. Shankar, and D.W.J. Stringer-Calvert. PVS: An experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *PROC Applied Formal Methods - FM-Trends'98, International Workshop on Current Trends in Applied Formal Method*, Boppard, DE, LNCS 1641, pages 338–345. Springer-Verlag, 7–9 Oct. 1999.
120. J. Palsberg. Closure analysis in constraint form. *TOPLAS*, 17(1):47–62, Jan. 1995.
121. S.E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In P. Van Hentenryck, editor, *Proc. 4<sup>th</sup> Int. Symp. SAS '97*, Paris, FR, 8–10 Sep. 1997, LNCS 1302, pages 345–360. Springer-Verlag, 1997.
122. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *J. Symbolic Logic*, 15(5/6):607–640, 1993.
123. A. Pnueli. The temporal logic of programs. In *Proc. 18<sup>th</sup> FOCS*, pages 46–57, Providence, RI, Nov. 1977.

124. K. Post. Mutually exclusive rules in logic programming. In M. Bruynooghe, editor, *Proc. Int. Symp. ILPS '1994*, Ithaca, NY, US, pages 472–486. MIT Press, 13–17 Nov. 1994.
125. J.-P. Queille and J. Sifakis. Verification of concurrent systems in CESAR. In *Proc. Int. Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1982.
126. F. Randimbivololona, J. Souyris, and A. Deutsch. Improving avionics software verification cost-effectiveness: Abstract interpretation based technology contribution. In *Proceedings DASIA 2000 – DAta Systems In Aerospace*, Montreal, CA. ESA Publications, 22–26 May 2000.
127. B. Reistad and D.K. Gifford. Static dependent costs for estimating execution time. In *Proc. ACM Conf. Lisp & Func. Prog.*, Orlando, FL, US, pages 65–78. ACM Press, 27–29 June 1994.
128. S. Sa di. Model checking guided abstraction and analysis. In J. Palsberg, editor, *Proc. $7^{th}$ Int. Symp. SAS '2000*, Santa Barbara, CA, US, LNCS 1824, pages 377–396. Springer-Verlag, 29 June – 1 Jul. 2000.
129. C.F. Schaefer and G.N. Bundy. Static analysis of exception handling in Ada. *Soft.–Pract. & Exp.*, 23(10):1157–1174, Oct. 1993.
130. D.A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In G. Levi, editor, *Proc. $5^{th}$ Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 351–380. Springer-Verlag, 1998.
131. P. Schnorf, M. Ganapathi, and J.L. Hennessy. Compile-time copy elimination. *Soft.–Pract. & Exp.*, 23(11):1175–1200, Nov. 1993.
132. D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Gen. Comp.*, 13(2):117–154, 1995.
133. N. Shankar. Unifying verification paradigms. In *FTRTFT'96*, 1996.
134. O. Shivers. The semantics of scheme control-flow analysis. In P. Hudak and N.D. Jones, editors, *Proc. PEPM '91*, Yale U., New Haven, CT, US, 17–19 June 1991, ACM SIGPLAN Not. 26(9), pages 190–198. ACM Press, Sep. 1991.
135. H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP '86*, Saarbr cken, DE, 17-19 Mar. 1986, LNCS 213, pages 327–338. Springer-Verlag, 1986.
136. J. Stransky. A lattice for abstract interpretation of dynamic (LISP-like) structures. *Inform. and Comput.*, 101(1):70–102, Nov. 1992.
137. D. Tarditi, J.G. Morrisett, P. Cheng, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. PLDI. ACM SIGPLAN Not. 31(5)*, pages 181–192, Philadephia, PA, US, 21–24 May 1996.
138. K.R. Traub, D.E. Culler, and K.E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. *LISP Pointers*, 5(1):324–334, Jan. – Mar. 1992.
139. M.Y. Vardi. Probabilistic linear-time model checking: An overview of the automata-theoretic approach. In J.-P. Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, $5^{th}$ Int. Symp. AMAST Workshop, ARTS '99*, Bamberg, DE, 26–28 May 1999, LNCS 1601, pages 265–276. Springer-Verlag, 1993.
140. F. Védrine. Binding-time analysis and strictness analysis by abstract interpretation. In A. Mycroft, editor, *Proc. $2^{nd}$ Int. Symp. SAS '95*, Glasgow, UK, 25–27 Sep. 1995, LNCS 983, pages 400–417. Springer-Verlag, 1995.
141. A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proc. $5^{th}$ Int. Symp. SAS '98*, Pisa, IT, 14–16 Sep. 1998, LNCS 1503, pages 152–167. Springer-Verlag, 1998.

142. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Programming, Special Issue on SAS'96*, 35(1):223–248, September 1999.
143. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Sci. Comput. Programming*, 31(1):147–173, May 1998.
144. E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting synchronization in concurrent constraint programming. In M.V. Hermenegildo and J. Penjam, editors, *Proc. 6$^{th}$ Int. Symp. PLILP '94*, Madrid, ES, 14–16 Sep. 1994, LNCS 844, pages 57–72. Springer-Verlag, 1994.

A broader bibliography is available in the extended electronic version of this paper.