Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane*
École normale supérieure, Paris

Patrick Cousot*,**
Courant Institute of Mathematical Sciences, NYU, New York & École normale supérieure, Paris

Radhia Cousot*
École normale supérieure & CNRS, Paris

Jérôme Feret*
École normale supérieure & INRIA, Paris

Laurent Mauborgne*,‡
École normale supérieure, Paris & IMDEA Software, Madrid

Antoine Miné*
École normale supérieure & CNRS, Paris

Xavier Rival*
École normale supérieure & INRIA, Paris

We discuss the principles of static analysis by abstract interpretation and report on the automatic verification of the absence of runtime errors in large embedded aerospace software by static analysis based on abstract interpretation. The first industrial applications concerned synchronous control/command software in open loop. Recent advances consider imperfectly synchronous, parallel programs, and target code validation as well. Future research directions on abstract interpretation are also discussed in the context of aerospace software.

Nomenclature

S	program states	I	initial states	t	state transition
$\mathcal{C}[\![t]\!]I$	collecting semantics	$\mathcal{T}[\![t]\!]I$	trace semantics	$\mathcal{P}[\![t]\!]I$	prefix trace semantics
F_P	prefix trace transformer	$\mathcal{R}[\![t]\!]I$	reachability semantics	F_R	reachability transformer
F_ι	interval transformer	1_S	identity on S (also t^0)	$t \circ r$	composition of relations
$\mathtt{V}$	set of all program variables	$\mathtt{x}$	program variable		t and r
$t^\star$	reflexive transitive closure	t^n	powers of relation t	ρ	reduction
	of relation t	α	abstraction function	γ	concretization function
$\mathrm{lfp}^{\subseteq} F$	least fixpoint of F for $\subseteq$	∇	widening	Δ	narrowing
$\|x\|$	absolute value of x	$X^\sharp$	abstract counterpart of X	$\wp(S)$	parts of set S (also 2^S)
q	quaternion	$\overline{q}$	conjugate of quaternion q	$\|\|q\|\|$	norm of quaternion q
$\mathbb{N}$	naturals	$\mathbb{Z}$	integers	$\mathbb{R}$	reals

*École normale supérieure, Département d'informatique, 45 rue d'Ulm, 75230 Paris cedex 05, First.Last@ens.fr.
**Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street New York, N.Y. 10012-1185, pcousot@cs.nyu.edu.
‡Fundación IMDEA Software, Facultad de Informática (UPM), Campus Montegancedo, 28660-Boadilla del Monte, Madrid, Spain.

I. Introduction

The validation of software checks informally (e.g., by code reviews or tests) the conformance of the software executions to a specification. More rigorously, the verification of software proves formally the conformance of the software semantics (that is, the set of all possible executions in all possible environments) to a specification. It is of course difficult to design a sound semantics, to get a rigorous description of all execution environments, to derive an automatically exploitable specification from informal natural language requirements, and to completely automatize the formal conformance proof (which is undecidable). In model-based design, the software is often generated automatically from the model so that the certification of the software requires the validation or verification of the model plus that of the translation into an executable software (through compiler verification or translation validation). Moreover, the model is often considered to be the specification, so there is no specification of the specification, hence no other possible conformance check. These difficulties show that fully automatic rigorous verification of complex software is very challenging and perfection is impossible.

We present abstract interpretation[1] and show how its principles can be successfully applied to cope with the above-mentioned difficulties inherent to formal verification.

- First, semantics and execution environments can be precisely formalized at different levels of abstraction, so as to correspond to a pertinent level of description as required for the formal verification.

- Second, semantics and execution environments can be over-approximated, since it is always sound to consider, in the verification process, more executions and environments than actually occurring in real executions of the software. It is crucial for soundness, however, to never omit any of them, even rare events. For example, floating-point operations incur rounding (to nearest, towards 0, plus or minus infinity) and, in the absence of precise knowledge of the execution environment, one must consider the worst case for each float operation. Another example is inputs, like voltages, that can be overestimated by the maximum capacity of the hardware register containing the value (anyway, a well-designed software should be defensive, i.e., have appropriate protections to cope with erroneous or failing sensors and be prepared to accept any value from the register).

- In the absence of an explicit formal specification or to avoid the additional cost of translating the specification into a format understandable by the verification tool, one can consider implicit specifications. For example, memory leaks, buffer overruns, undesired modulo in integer arithmetics, float overflows, data-races, deadlocks, live-locks, etc. are all frequent symptoms of software bugs, which absence can be easily incorporated as a valid but incomplete specification in a verification tool, maybe using user-defined parameters to choose among several plausible alternatives.

- Because of undecidability issues (which makes fully automatic proofs ultimately impossible on all programs) and the desire not to rely on end-user interactive help (which can be lengthy, or even intractable), abstract interpretation makes an intensive use of the idea of abstraction, either to restrict the properties to be considered (which introduces the possibility to have efficient computer representations and algorithms to manipulate them) or to approximate the solutions of the equations involved in the definition of the abstract semantics. Thus, proofs can be automated in a way that is always sound but may be imprecise, so that some questions about the program behaviors and the conformance to the specification cannot be definitely answered neither affirmatively nor negatively. So, for soundness, an alarm will be raise which may be false. Intensive research work is done to discover appropriate abstractions eliminating this uncertainty about false alarms for domain-specific applications.

We report on the successful cost-effective application of abstract interpretation to the verification of the absence of runtime errors in aerospace control software by the Astrée static analyzer,[2] illustrated first by the verification of the fly-by-wire primary software of commercial airplanes[3] and then by the validation of the Monitoring and Safing Unit (MSU) of the Jules Vernes ATV docking software.[4]

We discuss on-going extensions to imperfectly synchronous software, parallel software, and target code validation, and conclude with more prospective goals for rigorously verifying and validating aerospace software.

Contents

II. Theoretical Background on Abstract Interpretation

The *static analysis* of a program consists in automatically determining properties of all its possible executions in any possible execution environment (possibly constrained by, e.g., hypotheses on inputs). The program *semantics* is a mathematical model of these executions. In particular, the *collecting semantics* is a mathematical model of the strongest program property of interest (e.g., reachable states during execution). So, the collecting semantics of abstract interpretation is nothing more than the logic considered in other formal methods (e.g., first-order logic for deductive methods or temporal logic in model-checking) but in semantic, set theoretical form rather than syntactic, logical form. The *verification* consists in proving that the program collecting semantics implies a *specification* (e.g., the absence of runtime errors). We are interested in applications of abstract interpretation to automatic static analysis and verification. By "automatic", we mean without any human assistance during the analysis and the verification processes (a counter-example is deductive methods where theorem provers need to be assisted by humans). Being undecidable, any static analyzer or verifier is either unsound (their conclusions may be wrong), incomplete (they are inconclusive), may not terminate, or all of the above, and this on infinitely many programs. Abstract interpretation is a theory of approximation of mathematical structures that can be applied to the design of static analyzers and verifiers which are always sound and always terminate, and so are necessarily incomplete (except, obviously, in the case of finite models). So, abstract interpretation-based static analyzers and verifiers will fail on infinitely many programs. Fortunately, they will also succeed on infinitely many programs. The art of the designer of such tools is to make them succeed most often on the programs of interest to end-users. Such tools are thus often specific to a particular application domain.

II.A. Semantics

Following Cousot,[5] we model program execution by a *small-step operational semantics*, that is, a set S of program states, a transition relation $t \subseteq S \times S$ between program states, and a subset $I \subseteq S$ of initial program states. Given a initial state $s_0 \in I$, a successor state is $s_1 \in S$ such that $\langle s_0, s_1 \rangle \in t$, and so on and so forth,

the $i+1$-th state $s_{i+1} \in S$ is such that $\langle s_i, s_{i+1}\rangle \in t$. The execution either goes on like this forever (in case of non-termination) or stops at some final state s_n without any possible successor by t (i.e., $\forall s' \in S : \langle s_n, s'\rangle \notin t$). This happens, e.g., in case of program correct or erroneous termination. Of course, the transition relation is often non-deterministic, meaning that a state $s \in S$ may have many possible successors $s' \in S : \langle s, s'\rangle \in t$ (e.g., on program inputs). The *maximal trace semantics* is therefore $\mathcal{T}[\![t]\!]I \triangleq \{\langle s_0 \dots s_n\rangle \mid n \geqslant 0 \wedge s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t \wedge \forall s' \in S : \langle s_n, s'\rangle \notin t\} \cup \{\langle s_0 \dots s_n \dots\rangle \mid s_0 \in I \wedge \forall i \geq 0 : \langle s_i, s_{i+1}\rangle \in t\}$: it is the set of maximal sequences of states starting from an initial state, either ending in some final state or infinite, and satisfying the transition relation.

II.B. Collecting semantics

In practice, the maximal trace semantics $\mathcal{T}[\![t]\!]I$ of a program modeled by a transition system $\langle S, t, I\rangle$ is not computable and even not observable by a machine or a human being. However, we can observe program executions for a finite (yet unbounded) time, which we do when interested in program *safety* properties. Finite observations of program executions can be formally defined as $\mathcal{P}[\![t]\!]I \triangleq \{\langle s_0 \dots s_n\rangle \mid s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t\}$, which is called the (finite) *prefix trace semantics*. This semantics is simpler, but it is sufficient to answer any safety question about program behaviors, i.e., properties which failure is checkable by monitoring the program execution. Thus, it will be our *collecting semantics* in that it is the strongest program property of interest and defines precisely the static analyses we are interested in. Ideally, computing this collecting semantics would answer all safety questions. But this is impossible: the collecting semantics is still impossible to compute (except, obviously, for finite systems), and so, we will use abstractions of this semantics to provide sound but incomplete answers.

The choice of the program properties of interest, hence of the collecting semantics, is problem dependent, and depends on the level of observation of program behaviors. For example, when only interested in *invariance* properties, another possible choice for the collecting semantics would be the *reachability semantics* $\mathcal{R}[\![t]\!]I \triangleq \{s' \mid \exists s \in I : \langle s, s'\rangle \in t^\star\}$, where the *reflexive transitive closure* $t^\star$ of a relation t is $t^\star \triangleq \{\langle s, s'\rangle \mid \exists n \geqslant 0 : \exists s_0 \dots s_n : s_0 = s \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t \wedge s_n = s'\}$, i.e., $t^\star = \bigcup_{n\geq 0} t^n$ where, for all $n \geqslant 0$, $t^n \triangleq \{\langle s, s'\rangle \mid \exists s_0 \dots s_n : s_0 = s \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t \wedge s_n = s'\}$. The reachability semantics is more abstract than the prefix trace semantics since, if we know exactly which states can be reached during execution, we no longer know in which order. This is a basic example of abstraction. Assume we are asked the question "does state s_1 always appear before state s_2 in all executions" and we only know the reachability semantics $\mathcal{R}[\![t]\!]I$. If $s_1 \notin \mathcal{R}[\![t]\!]I$ or $s_2 \notin \mathcal{R}[\![t]\!]I$, then we can answer "never" for sure. Otherwise $s_1, s_2 \in \mathcal{R}[\![t]\!]I$, so we can only answer "I don't know", which is an example of incompleteness which is inherent to reachability with respect to the prefix trace semantics.

II.C. Fixpoint semantics

Semantics can be expressed as fixpoints.[1] An example is $t^\star$, which is the $\subseteq$-least solution of both equations $X = 1_S \cup (X \circ t)$ and $X = 1_S \cup (t \circ X)$, where $1_S \triangleq \{\langle s, s\rangle \mid s \in S\}$ is the identity on S and $\circ$ is the composition of relations $t \circ r \triangleq \{\langle s, s''\rangle \mid \exists s' : \langle s, s'\rangle \in t \wedge \langle s', s''\rangle \in r\}$.

We write $t^\star = \mathrm{lfp}^{\subseteq} F$ where $F(X) \triangleq 1_S \cup (X \circ t)$ (or $t^\star = \mathrm{lfp}^{\subseteq} B$ where $B(X) \triangleq 1_S \cup (t \circ X)$) to mean that $t^\star$ is the $\subseteq$-least fixpoint of F (resp. B). This means that $t^\star$ is a fixpoint of F ($t^\star = F(t^\star)$) and in fact the least one (if $X = F(X)$ then $t^\star \subseteq X$). The existence of such least fixpoints follows from Tarski's theorem.[6]

Least fixpoints can be computed iteratively.[6] Starting iterating from $\emptyset$, we have $X^0 \triangleq 1_S \cup (\emptyset \circ t) = 1_S = t^0$, $X^1 \triangleq 1_S \cup (X^0 \circ t) = 1_S \cup t$, $X^2 \triangleq 1_S \cup (X^1 \circ t) = 1_S \cup t \cup t^2$, etc. If, by recurrence hypothesis, $X^n = \bigcup_{i=0}^n t^i$, we get $X^{n+1} \triangleq 1_S \cup (X^n \circ t) = 1_S \cup \left(\left(\bigcup_{i=0}^n t^i\right) \circ t\right) = 1_S \cup \left(\bigcup_{i=0}^n t^{i+1}\right) = \bigcup_{i=0}^{n+1} t^i$. By recurrence, the iterates are $\forall n \geqslant 0 : X^n = \bigcup_{i=0}^n t^i$. Passing to the limit, $X^\star \triangleq \bigcup_{n\geq 0} X^n = \bigcup_{n\geq 0} \bigcup_{i=0}^n t^i = \bigcup_{n\geq 0} t^n$, which is precisely $t^\star$.

Other examples of fixpoints are $\mathcal{R}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_R$ where $F_R(X) \triangleq I \cup \{s' \mid \exists s \in X : \langle s, s'\rangle \in t\}$ and $\mathcal{P}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_P$ where $F_P(X) \triangleq \{\langle s_0\rangle \mid s_0 \in I\} \cup \{\langle s_0 \dots s_n s_{n+1}\rangle \mid \langle s_0 \dots s_n\rangle \in X \wedge \langle s_n, s_{n+1}\rangle \in t\}$.

II.D. Abstraction

Abstraction[1] relates a concrete and an abstract semantics so that any property proved in the abstract is valid in the concrete (this is soundness) while a property in the concrete might not be provable in the abstract

(this is incompleteness). Termination will be considered later (Sect. II.O).

An example is the correspondence between the concrete prefix trace semantics and the abstract reachability semantics. Defining the *reachability abstraction function* $\alpha_R(X) \triangleq \{s \mid \exists \langle s_0 \dots s_n \rangle \in X : s = s_n\}$, we have $\mathcal{R}[\![t]\!]I = \alpha_R(\mathcal{P}[\![t]\!]I)$. The reachability abstraction consists in remembering only the last state in the finite observations of programs execution. This records the reachable states but no longer the order in which these states appear during execution.

Another example is the correspondence between the concrete maximal trace semantics and the abstract prefix trace semantics. Defining the *prefix abstraction function* $\alpha_P(X) \triangleq \{\langle s_0 \dots s_n \rangle \mid n \geqslant 0 \wedge \exists \langle s_{n+1} \dots \rangle : \langle s_0 \dots s_n s_{n+1} \dots \rangle \in X\}$, we have $\mathcal{P}[\![t]\!]I = \alpha_P(\mathcal{T}[\![t]\!]I)$.

By composition $\mathcal{R}[\![t]\!]I = (\alpha_R \circ \alpha_P)(\mathcal{T}[\![t]\!]I)$, that is, the composition of abstractions is an abstraction.

II.E. Concretization

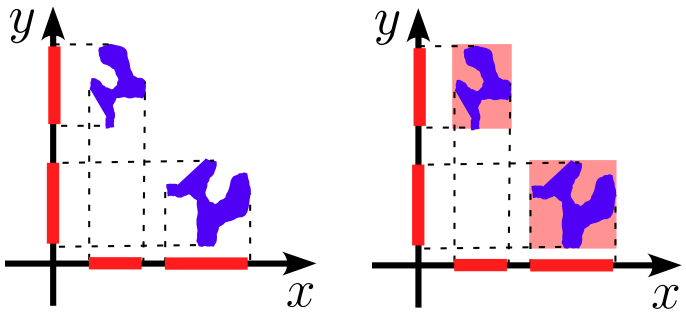
We can consider an "inverse" *concretization function* $\gamma_R(X) \triangleq \{\langle s_0 \dots s_n \rangle \in X \mid \forall i \in [0, n] : s_i \in X\}$. This rebuilds the partial execution traces from the reachable states, but considers that they can appear in any order since the order of appearance has been abstracted away. It follows that $\mathcal{P}[\![t]\!]I \subseteq \gamma_R(\mathcal{R}[\![t]\!]I)$, and we say that the abstract is an over-approximation of the concrete in that the concretization of the abstract $\gamma_R(\mathcal{R}[\![t]\!]I)$ has more possible program behaviors than the concrete $\mathcal{P}[\![t]\!]I$. This ensures soundness in that, if a property is true of the executions in the abstract, then it is true in the concrete (for example, if a behavior does not appear in the abstract, it certainly cannot appear in the concrete, which has fewer possible behaviors). However incompleteness appears in that, if we want to prove that a program behavior is possible and it does exist in the abstract, we cannot conclude that it exists in the concrete.

II.F. Galois connections

The pair $\langle \alpha_R, \gamma_R \rangle$ is an example of *Galois connection* $\langle \alpha, \gamma \rangle$[7] defined as $\forall T, R : \alpha(T) \subseteq^\sharp R$ if and only if $T \subseteq \gamma(R)$ (where the abstract inclusion $\subseteq^\sharp$ is a partial order such that $x \subseteq^\sharp y$ implies $\gamma(x) \subseteq \gamma(y)$, so that $\subseteq^\sharp$ is the abstract version of the concrete inclusion $\subseteq$, that is, logical implication).

Galois connections have many interesting mathematical properties. In particular, any concrete property T has a "best" abstraction $\alpha(T)$. This means that $\alpha(T)$ is an over-approximation of T in that $T \subseteq \gamma(\alpha(T))$ [a]. Moreover, if R is another over-approximation of T in that $T \subseteq \gamma(R)$, then $\alpha(T)$ is more precise, since $\alpha(T) \subseteq^\sharp R$ [b].

Another example of Galois connection is the Cartesian abstraction, where a set of pairs is abstracted to a pair of sets by projection:



Cartesian abstraction α_c Cartesian concretization γ_c

Given a set X of pairs $\langle x, y \rangle \in X$, its abstraction is $\alpha_c(X) \triangleq \langle \{x \mid \exists y : \langle x, y \rangle \in X\}, \{y \mid \exists x : \langle x, y \rangle \in X\} \rangle$. The concretization is $\gamma_c(\langle X, Y \rangle) \triangleq \{\langle x, y \rangle \mid x \in X \wedge y \in Y\}$. The abstract order $\subseteq_c$ is componentwise inclusion. Observe that α_c is surjective (onto) and γ_c is injective (one to one). This is characteristic of *Galois surjections* which are Galois connections $\langle \alpha, \gamma \rangle$ such that α is surjective or equivalently γ is injective or equivalently $\alpha \circ \gamma = 1$ (where 1 is the identity function: $1(x) = x$).

Not all abstractions are Galois connections, in which case one can always use a concretization function.[8] A counter-example is provided by a disk, which has no best over-approximation by a convex polyhedron,[9] as shown by Euclid.[10]

[a] We have $\alpha(T) \subseteq^\sharp \alpha(T)$ so that, by the if part of the definition (with $R = \alpha(T)$), we get $T \subseteq \gamma(\alpha(T))$.

[b] If $T \subseteq \gamma(R)$ then, by the only if part of the definition, it follows that $\alpha(T) \subseteq^\sharp R$.

II.G. The lattice of abstractions

We have seen that the reachability semantics is more abstract than the prefix trace semantics which is more abstract than the maximal traces semantics. Some abstractions (such as sign and parity of natural numbers) are not comparable. So, abstract properties form a lattice with respect to the relation "is more abstract than".[1,7] Exploring the world of (collecting) semantics and their lattice of abstractions is one of the main research subjects in abstract interpretation. In particular "finding the appropriate level of abstraction required to answer a question" is a recurrent fundamental and practical question.

II.H. Sound (and complete) abstract semantics

Given a concrete semantics $\mathcal{S}$ and an abstraction specified by a concretization function γ (respectively an abstraction function α), we are interested in an abstract semantics $\mathcal{S}^\sharp$ which is *sound* in that $\mathcal{S} \subseteq \gamma(\mathcal{S}^\sharp)$ (respectively $\alpha(\mathcal{S}) \subseteq^\sharp \mathcal{S}^\sharp$, which is equivalent for Galois connections). This corresponds to the intuition that no concrete case is ever forgotten in the abstract (so there can be no false negative). It may also happen that the abstract semantics is *complete*, meaning that $\mathcal{S} \supseteq \gamma(\mathcal{S}^\sharp)$ (respectively $\alpha(\mathcal{S}) \supseteq^\sharp \mathcal{S}^\sharp$ [c]). This corresponds to the intuition that no abstract case is ever absent in the concrete (so there can be no false positive). The ideal case is that of a sound and complete semantics such that $\mathcal{S} = \gamma(\mathcal{S}^\sharp)$ (respectively $\alpha(\mathcal{S}) = \mathcal{S}^\sharp$).

The abstractions used in program proof methods (such as α_P for Burstall's intermittent assertions proof method[11,12] or α_R for Floyd's invariant assertions proof method[13,14]) are usually sound and complete (but, by undecidability, these abstractions yield proof methods that are not fully mechanizable). The abstractions used in static program analysis (such as the Cartesian abstraction in Sect. II.F, the interval abstraction in Sect. II.M, or type systems that do reject programs that can never go wrong[15]) are usually sound and incomplete (but fully mechanizable).

II.I. Abstract transformers

Because the concrete semantics $\mathcal{S} = \mathrm{lfp}^{\subseteq} F$ is usually the least fixpoint of a concrete transformer F, one can attempt to express the abstract semantics in the same fixpoint form $\mathcal{S}^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$ for an abstract transformer $F^\sharp$ where $X \subseteq^\sharp Y$ if and only if $\gamma(X) \subseteq \gamma(Y)$. The abstract transformer $F^\sharp$ is said to be sound when $F \circ \gamma \subseteq \gamma \circ F^\sharp$ (respectively $\alpha \circ F \subseteq^\sharp F^\sharp \circ \alpha$ [d]). In case of a Galois connection, there is a "best" abstract transformer, which is $F^\sharp \triangleq \alpha \circ F \circ \gamma$. In all cases, the main idea is that the abstract transformer $F^\sharp$ always over-approximates the result of the concrete transformer F.

II.J. Sound abstract fixpoint semantics

Given a concrete fixpoint semantics $\mathcal{S} = \mathrm{lfp}^{\subseteq} F$, the task of a static analysis designer is to elaborate an abstraction $\langle \alpha, \gamma \rangle$ and then the abstract transformer $F^\sharp \triangleq \alpha \circ F \circ \gamma$ (or $F^\sharp$ such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$ in the absence of a Galois connection). Under appropriate hypotheses,[7] the abstract semantics is then guaranteed to be sound: $\mathcal{S} = \mathrm{lfp}^{\subseteq} F \subseteq \gamma(\mathcal{S}^\sharp) = \gamma(\mathrm{lfp}^{\subseteq^\sharp} F^\sharp)$. Otherwise stated, the abstract fixpoint over-approximates the concrete fixpoint, hence preserves the soundness: if $\mathcal{S}^\sharp \subseteq^\sharp P^\sharp$ then $\mathcal{S} \subseteq \gamma(P^\sharp)$ (any abstract property $P^\sharp$ which holds in the abstract also holds for the concrete semantics). This yields the basis to formally verify the soundness of static analyzers using theorem provers or proof checkers.[16]

II.K. Sound and complete abstract fixpoints semantics

Under the hypotheses of Tarski's fixpoint theorem[6] and the additional commutation hypothesis[7] $\alpha \circ F = F^\sharp \circ \alpha$ for Galois connections, we have $\alpha(\mathcal{S}) = \alpha(\mathrm{lfp}^{\subseteq} F) = \mathcal{S}^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$. Otherwise stated, the fact that the concrete semantics is a least fixpoint is preserved in the abstract.

For example, $\mathcal{R}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_R$ and $\alpha_R \circ F_P = F_R \circ \alpha_R$ imply that $\mathcal{R}[\![t]\!]I \triangleq \alpha_R(\mathcal{P}[\![t]\!]I) = \alpha_R(\mathrm{lfp}^{\subseteq} F_P) = \mathrm{lfp}^{\subseteq} F_R$. The intuition is that, to reason on reachable states, it is useless to consider execution traces. Complete abstractions exactly answer the class of questions defined by the abstraction of the collecting semantics. However, for most interesting questions on programs, the answer is not algorithmic or very severe restrictions have to be considered, such as finiteness.

[c] Which in general is not equivalent, even for Galois connections.

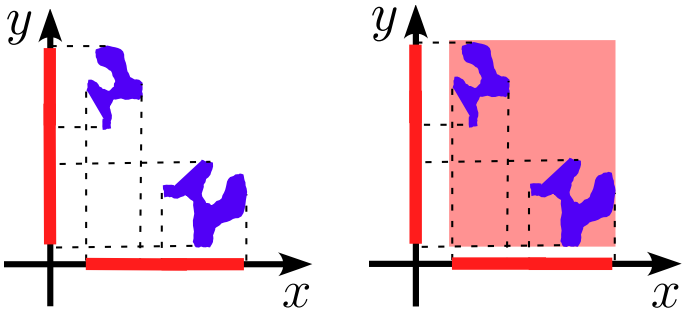
[d] Which is equivalent when $\langle \alpha, \gamma \rangle$ is a Galois connection and F and $F^\sharp$ are increasing.

II.L. Example of finite abstraction: model-checking

Model-checking[17] abstracts systems to finite state systems so that partial traces or reachable states can be computed by finite iterative fixpoint concretization. In fact, Cousot[18] showed that any abstraction α to a *finite* domain D is amenable to model-checking by bit-wise encoding of $\{\gamma(P) \mid P \in D\}$ (where D is the finite set of abstract properties, e.g., $D = \wp(S)$ for reachability over a finite set S of states).

II.M. Example of infinite abstraction: interval abstraction

Assume that a program has numerical (integer, float, etc.) variables $\mathtt{x} \in \mathtt{V}$, so that program states $s \in S$ map each variable $\mathtt{x} \in \mathtt{V}$ to its numerical value $s(\mathtt{x})$ in state s. An interesting problem on programs is to determine the interval of variation of each variable $\mathtt{x} \in \mathtt{V}$ during any possible execution. This consists in abstracting the reachability semantics first by the Cartesian abstraction and then by the interval abstraction:



Interval abstraction α_c Interval concretization γ_c

This is $\alpha_\iota(\mathcal{R}[\![t]\!]I)$ where $\alpha_\iota(X)(\mathtt{x}) = [\min_{s\in X} s(\mathtt{x}), \max_{s\in X} s(\mathtt{x})]$ which can be $\emptyset$ when X is empty and may have infinite bounds $-\infty$ or $+\infty$ in the absence of a minimum or a maximum. This abstraction is infinite in that, e.g., $[0,0] \subseteq [0,1] \subseteq [0,2] \subseteq \cdots \subseteq [0,+\infty]$. Assuming that numbers are bounded (e.g., $-\infty =$ `minint` and $+\infty =$ `maxint` for integers) yields a finite set of states but this is of no help due to combinatorial explosion when considering all subsets of states in $\wp(S)$.

One may wonder why infinite abstractions are of any practical interest since computers can only do finite computations anyway. The answer comes form a very simple example: $P \equiv$ `x=0; while (x<=`n`) { x=x+1 }`, where the symbol $n \geqslant 0$ stands for a numerical constant (so that we have an infinite family of programs for all integer constants $n = 0, 1, 2, \ldots$).[19] For any given constant value of n, an abstract interpretation-based analyzer using the interval abstraction[20] will determine that, on loop body entry, $\mathtt{x} \in [0, n]$ always hold. If we were restricted to a finite domain, we would have only finitely many such intervals, and so, we would find the exact answer for only finitely many programs while missing the answer for infinitely many programs in the family. An alternative would be to use an infinite sequence of finite abstractions of increasing precision (e.g., by restricting abstract states to first $\{1\}$, then $\{1,2\}$, then $\{1,2,3\}$, etc.), and run finite analyses until a precise answer is found, but this would be costly and, moreover, the analysis might not terminate or would have to enforce the termination with exactly the same techniques as those used for infinite abstractions (Sect. II.O), without the benefit of avoiding the combinatorial state explosion.

II.N. Abstract domains and functions

Encoding program properties uniformly (e.g., as terms in theorem provers or BDDs[21] in model-checking) greatly simplifies the programming and reusability of verifiers. However, it severely restricts the ability for programmers of these verifiers to choose very efficient computer representations of abstract properties and dedicated high-performance algorithms to manipulate such abstract properties in transformers. For example, an interval is better represented by the pair of its bounds rather than the set of its points, whatever computer encoding of sets is used.

So, abstract interpretation-based static analyzers and verifiers do not use a uniform encoding of abstract properties. Instead, they use many different *abstract domains* which are algebras (for mathematicians) or modules (for computer scientists) with data structures to encode abstract properties (e.g., either $\emptyset$ or a pair $[\ell, h]$ of numbers or infinity for intervals, with $\ell \leq h$). Abstract functions are elementary functions on abstract properties which are used to express the abstraction of the fixpoint transformers F_P, F_R, etc.

For example, the interval transformer F_ι will use interval inclusion ($\emptyset \subseteq \emptyset \subseteq [a,b]$, $[a,b] \subseteq [c,d]$ if and only if $c \leqslant a$ and $b \leqslant d$), addition ($\emptyset + \emptyset = \emptyset + [a,b] = [a,b] + \emptyset = \emptyset$ and $[a,b] + [c,d] = [a+c, b+d]$), union ($\emptyset \cup \emptyset = \emptyset$, $\emptyset \cup [a,b] = [a,b] \cup \emptyset = [a,b]$ and $[a,b] \cup [c,d] = [\min(a,c), \max(b,d)]$), etc., which will be basic operations available in the abstract domain. Public implementations of abstract domains are available such as APRON[22,23] for numerical domains.

II.O. Convergence acceleration by extrapolation

II.O.1. Widening

Let us consider the program `x=1; while (true) { x=x+2 }`. The interval of variation of variable `x` is the least interval solution to the equation $X = F_\iota(X)$ where $F_\iota(X) \triangleq [1,1] \cup (X + [2,2])$. Solving iteratively from $X^0 = \emptyset$, we have $X^1 = [1,1] \cup (X^0 + [2,2]) = [1,1]$, $X^2 = [1,1] \cup (X^1 + [2,2]) = [1,1] \cup [3,3] = [1,3]$, $X^3 = [1,1] \cup (X^2 + [2,2]) = [1,1] \cup [3,5] = [1,5]$, which, after infinitely many iterates and passing to the limit, yields $[1, +\infty)$. Obviously, no computer can compute infinitely many iterates, nor perform the reasoning by recurrence and automatically pass to the limit as humans would do.

An idea is to accelerate the convergence by an extrapolation operator called a *widening*,[1,5,20] solving $X = X \mathbin{\triangledown} F_\iota(X)$ instead of $X = F_\iota(X)$. The widening $\triangledown$ uses two consecutive iterates X^n and $F_\iota(X^n)$ in order to extrapolate the next one X^{n+1}. This extrapolation should be an over-approximation ($X^n \subseteq X^{n+1}$ and $F_\iota(X^n) \subseteq X^{n+1}$) for soundness and enforce convergence for termination. In finite abstract domains, widenings are useless and can be replaced with the union $\cup$.

An example widening for intervals is $x \mathbin{\triangledown} \emptyset = \emptyset$, $\emptyset \mathbin{\triangledown} x = x$, $[a,b] \mathbin{\triangledown} [c,d] \triangleq [\ell, h]$, where $\ell = -\infty$ when $c < a$ and $\ell = a$ when $a \leqslant c$. Similarly, $h = +\infty$ when $b < d$ and $h = b$ when $d \leqslant b$. The widened interval is always larger (soundness) and avoids infinitely increasing iterations (e.g., [0,0], [0,1], [0,2], etc.) by pushing to infinity limits that are unstable (termination).

For the equation $X = X \mathbin{\triangledown} F_\iota(X)$ where $F_\iota(X) \triangleq [1,1] \cup (X + [2,2])$, the iteration is now $X^0 = \emptyset$, $X^1 = X^0 \mathbin{\triangledown} ([1,1] \cup (X^0 + [2,2])) = \emptyset \mathbin{\triangledown} [1,1] = [1,1]$, $X^2 = X^1 \mathbin{\triangledown} ([1,1] \cup (X^1 + [2,2])) = [1,1] \mathbin{\triangledown} [1,3] = [1, +\infty)$, $X^3 = X^2 \mathbin{\triangledown} ([1,1] \cup (X^2 + [2,2])) = [1, +\infty) \mathbin{\triangledown} [1, +\infty) = [1, +\infty) = X^2$, which converges to a fixpoint in only three steps.

II.O.2. Narrowing

For the program, `x=1; while (x<100) { x=x+2 }`, we would have $X = X \mathbin{\triangledown} F_\iota(X)$ where $F_\iota(X) \triangleq [1,1] \cup ((X + [2,2]) \cap (-\infty, 99])$ with iterates $X^0 = \emptyset$, $X^1 = X^0 \mathbin{\triangledown} ([1,1] \cup ((X^0 + [2,2]) \cap (-\infty, 99])) = \emptyset \mathbin{\triangledown} [1,1] = [1,1]$, $X^2 = X^1 \mathbin{\triangledown} ([1,1] \cup ((X^1 + [2,2]) \cap (-\infty, 99])) = [1,1] \mathbin{\triangledown} [1,3] = [1, +\infty)$.

After that upwards iteration (where intervals are wider and wider), we can go on with a downwards iteration (where intervals are narrower and narrower). To avoid infinite decreasing chains (such as $[0, +\infty)$, $[1, +\infty)$, $[2, +\infty)$, ..., which limit is $\emptyset$), we use an extrapolation operator called a *narrowing*[1,5] $\vartriangle$ for the equation $X = X \mathbin{\vartriangle} F_\iota(X)$. The narrowing should ensure both soundness and termination. In finite abstract domains, narrowings are useless and can be replaced with the intersection $\cap$.

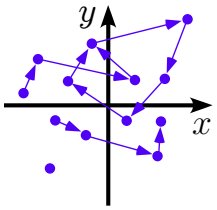
An example of narrowing for intervals is $\emptyset \mathbin{\vartriangle} x = \emptyset$, $x \mathbin{\vartriangle} \emptyset = \emptyset$, $[a,b] \mathbin{\vartriangle} [c,d] = [\ell, h]$ where $\ell = c$ when $a = -\infty$ and $\ell = a$ when $a \neq -\infty$ and similarly $h = d$ when $b = +\infty$ and otherwise $h = b$. So, infinite bounds are refined but not finite ones, so that the limit of $[0, +\infty)$, $[1, +\infty)$, $[2, +\infty)$, ..., $\emptyset$ will be roughly over-approximated as $[0, +\infty)$. This ensures the termination of the iteration process. The narrowed interval $[a,b] \mathbin{\vartriangle} [c,d]$ is wider than $[c,d]$, which ensures soundness.

For the program `x=1; while (x<100) { x=x+2 }`, the downwards iteration is now $Y = Y \mathbin{\vartriangle} F_\iota(Y)$ starting from the fixpoint obtained after widening: $Y^0 = [1, +\infty)$, $Y^1 = Y^0 \mathbin{\vartriangle} ([1,1] \cup ((Y^0 + [2,2]) \cap (-\infty, 99])) = [1, +\infty) \mathbin{\vartriangle} ([1,1] \cup ([3, +\infty) \cap (-\infty, 99])) = [1, +\infty) \mathbin{\vartriangle} [1,99] = [1,99]$. The next iterate is $Y^2 = Y^1 \mathbin{\vartriangle} ([1,1] \cup ((Y^1 + [2,2]) \cap (-\infty, 99])) = [1,99] \mathbin{\vartriangle} ([1,1] \cup ([3,101] \cap (-\infty, 99])) = [1,99] \mathbin{\vartriangle} [1,99] = [1,99] = Y^1$ so that a fixpoint is reached (although it may not be the least one, in general).

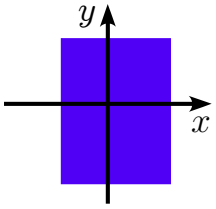
Of course, for finite abstractions (where strictly increasing chains are finite) no widening nor narrowing is needed since the brute-force iteration in the finite abstract domain always terminates. However, to avoid a time and space explosion, convergence acceleration with widening and narrowing may be helpful (at the price of incompleteness in the abstract, which is present anyway in the concrete except for finite transition systems).

II.P. Combination of abstract domains

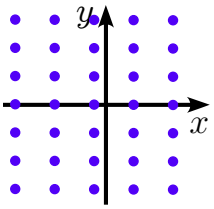
Abstract interpretation-based tools usually use several different abstract domains, since the design of a complex one is best decomposed into a combination of simpler abstract domains. Here are a few abstract domain examples used in the ASTRÉE static analyzer:[2]



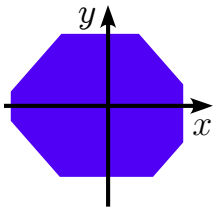
Collecting semantics:[1,5]
partial traces



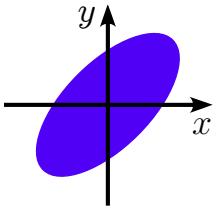
Intervals:[20]
$\mathtt{x} \in [a, b]$



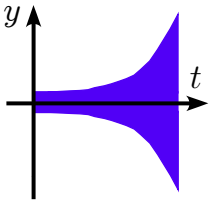
Simple congruences:[24]
$\mathtt{x} \equiv a[b]$



Octagons:[25]
$\pm\mathtt{x} \pm \mathtt{y} \leqslant a$



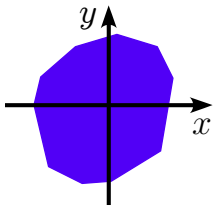
Ellipses:[26]
$\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{xy} \leqslant d$



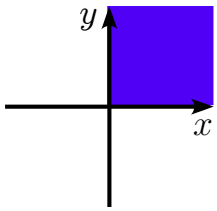
Exponentials:[27]
$-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

Such abstract domains (and more) are described in more details in Sects. III.H–III.I.

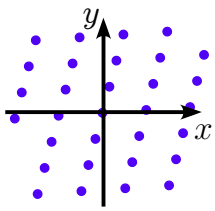
The following classic abstract domains, however, are not used in ASTRÉE because they are either too imprecise, not scalable, difficult to implement correctly (for instance, soundness may be an issue in the event of floating-point rounding), or out of scope (determining program properties which are usually of no interest to prove the specification):



Polyhedra:[9]
too costly



Signs:[7]
too imprecise



Linear congruences:[28]
out of scope

Because abstract domains do not use a uniform machine representation of the information they manipulate, combining them is not completely trivial. The conjunction of abstract program properties has to be performed, ideally, by a *reduced product*[7] for Galois connection abstractions. In absence of a Galois connection or for performance reasons, the conjunction is performed using an easily computable but not optimal over-approximation of this combination of abstract domains.

Assume that we have designed several abstract domains and compute $\mathrm{lfp}^{\subseteq} F_1 \in D_1$, ..., $\mathrm{lfp}^{\subseteq} F_n \in D_n$ in these abstract domains D_1, ..., D_n, relative to a collecting semantics $\mathcal{C}[\![t]\!]I$. The combination of these analyses is sound as $\mathcal{C}[\![t]\!]I \subseteq \gamma_1(\mathrm{lfp}^{\subseteq} F_1) \cap \cdots \cap \gamma_n(\mathrm{lfp}^{\subseteq} F_n)$. However, only combining the analysis results is not very precise, as it does not permit analyses to improve each other during the computation. Consider, for instance, that interval and parity analyses find respectively that $\mathtt{x} \in [0, 100]$ and $\mathtt{x}$ is *odd* at some iteration. Combining the results would enable the interval analysis to continue with the interval $\mathtt{x} \in [1, 99]$ and, e.g., avoid a useless widening. This is not possible with analyses carried out independently.

Combining the analyses by a reduced product, the proof becomes "let $F(\langle x_1, \ldots, x_n \rangle) \triangleq \rho(\langle F_1(x_1), \ldots, F_n(x_n) \rangle)$ and $\langle r_1, \ldots, r_n \rangle = \mathrm{lfp}^{\subseteq} F$ in $\mathcal{C}[\![t]\!]I \subseteq \gamma_1(r_1) \cap \cdots \cap \gamma_n(r_n)$" where ρ performs the *reduction* between abstract domains. For example $\rho(\langle [0, 100], \mathit{odd} \rangle) = \langle [1, 99], \mathit{odd} \rangle$.

To define ρ, first consider the case of two abstract domains. The conjunction of $p_1 \in D_1$ and $p_2 \in D_2$ is $\gamma_1(p_1) \cap \gamma_2(p_2)$ in the concrete, which is over-approximated as $\alpha_1(\gamma_1(p_1) \cap \gamma_2(p_2))$ in D_1 and $\alpha_2(\gamma_1(p_1) \cap \gamma_2(p_2))$ in D_2. So, the reduced product of D_1 and D_2 is $\{\rho_{12}(\langle p_1, p_2 \rangle) \mid p_1 \in D_1 \wedge p_2 \in D_2\}$, where the reduction is $\rho_{12}(\langle p_1, p_2 \rangle) \triangleq \langle \alpha_1(\gamma_1(p_1) \cap \gamma_2(p_2)), \alpha_2(\gamma_1(p_1) \cap \gamma_2(p_2)) \rangle$.

If more than two abstract domains are considered, a global reduction ρ can be defined by iterating the two-by-two reductions ρ_{ij}, $i \neq j$ until a fixpoint is reached. For the sake of efficiency, an over-approximation of the reduced product can be used, where only some of the reductions ρ_{ij} are applied, in a fixed order.[29] These reduction ideas also apply in absence of Galois connection; detailed examples are given in Sect. III.J.

II.Q. Partitioning abstractions

Another useful tool to design complex abstractions from simpler ones is *partitioning*.[30] In its simplest form, it consists in considering a collecting semantics on a powerset concrete domain $C = \wp(S)$, and a finite partition $S_1, \ldots, S_n$ of the set S. Each part $\wp(S_i)$ is abstracted by an abstract domain D_i (possibly the same for all partitions). An abstract element is thus a tuple $\langle d_1, \ldots, d_n \rangle \in D_1 \times \cdots \times D_n$, with concretization $\gamma(\langle d_1, \ldots, d_n \rangle) \triangleq (\gamma_1(d_1) \cap S_1) \cup \cdots \cup (\gamma_n(d_n) \cap S_n)$ and abstraction $\alpha(X) \triangleq \langle \alpha_1(X \cap S_1), \ldots, \alpha_n(X \cap S_n) \rangle$. For instance, one may consider refining the interval domain by maintaining a distinct interval for positive values and for negative values of `x`: $\gamma(\langle [\ell^+, h^+], [\ell^-, h^-] \rangle) \triangleq ([\ell^+, h^+] \cap [0, +\infty)) \cup ([\ell^-, h^-] \cap (-\infty, 0))$. This domain can represent some disjoint sets (such as the set of non-zero integers $(-\infty, -1] \cup [1, +\infty)$), while the non-partitioned interval domain cannot.

Instead of a partition $S_1, \ldots, S_n$ of S, one can choose a covering of S. In this case, several syntactically distinct abstract elements may represent the same concrete element, but this does not pose any difficulty. It is also possible to choose an infinite family of sets $(S_i)_{i \in N}$ covering S such that each element of $\wp(S)$ can be covered by finitely many parts in $(S_i)_{i \in N}$. A common technique[31] is to choose the family $(S_i)_{i \in N}$ as an abstract domain, so that, given two abstract domains D_s and D_d, the partitioning of D_d over D_s is an abstract domain containing finite sets of pairs in $D_s \times D_d$, where $\gamma(\langle \langle s_1, d_1 \rangle, \ldots, \langle s_n, d_n \rangle \rangle \triangleq \bigcup_{i=1}^{n} \gamma_s(s_i) \cap \gamma_d(d_i)$.

Later sections present examples of partitioning abstractions for program control states (Sect. III.C), program traces (Sect. III.E), and program data states (Sect. III.H.4).

II.R. Static analysis

Static analysis consists in answering an implicit question of the form "what can you tell me about the collecting semantics of this program?", e.g., "what are the reachable states?". Because the problem is undecidable, we provide an over-approximation by automatically computing an abstraction of the collecting semantics. For example, interval analysis[20] over-approximates $\mathrm{lfp}^{\subseteq} F_\iota$ using widening and narrowing. The benefit of static analysis is to provide complex information about program behaviors without requiring end-users to provide specifications.

An abstract interpretation-based static analyzer is built by combining abstract domains, e.g., in order to automatically compute abstract program properties in $D = D_1 \times \cdots \times D_n$. This consists in reading the program text from which an abstract transformer F_D is computed using compilation techniques. Then, an over-approximation of $\mathrm{lfp}^{\subseteq} F_D$ is computed by iteration with extrapolation by widening and narrowing. This abstract property can be interactively reported to the end-user through an interface or used to check an abstract specification.

II.S. Abstract specifications

A *specification* is a property of the program semantics. Because the specification must be specified relative to a program semantics, we understand it with respect to a collecting semantics. For example, a reachability specification often takes the form of a set $B \in \wp(S)$ of bad states (so that the good states are the complement $S \setminus B$). The specification is usually given at some level of abstraction. For example, the interval of variation of the values of a variable `x` at execution is always between two bounds $[\ell_{\mathtt{x}}, h_{\mathtt{x}}]$.

II.T. Verification

The *verification* consists in proving that the collecting semantics implies the specification. For example the reachability specification with bad states $B \in \wp(S)$ is $\mathcal{R}[\![t]\!]I \subseteq (S \setminus B)$, that is, "no execution can reach a

bad state". Because the specification is given at some level of abstraction, the verification needs not be done in the concrete.

For the interval example, we would have to check $\forall \mathtt{x} \in \mathtt{V} : \alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x}) \subseteq [\ell_\mathtt{x}, h_\mathtt{x}]$. To do that, we might think of checking with the abstract interval semantics $\mathcal{I}[\![t]\!]I(\mathtt{x}) \subseteq [\ell_\mathtt{x}, h_\mathtt{x}]$, where the abstract interval semantics $\mathcal{I}[\![t]\!]$ is an over-approximation in the intervals of the reachability semantics $\mathcal{R}[\![t]\!]$. This means that $\mathcal{I}[\![t]\!]I(\mathtt{x}) \supseteq \alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x})$. Observe that $\forall \mathtt{x} \in \mathtt{V} : \mathcal{I}[\![t]\!]I(\mathtt{x}) \subseteq [\ell_\mathtt{x}, h_\mathtt{x}]$ implies $\alpha_\iota(\mathcal{R}[\![t]\!]I)(\mathtt{x}) \subseteq [\ell_\mathtt{x}, h_\mathtt{x}]$, proving $\forall \mathtt{x} \in \mathtt{V}, \forall s \in \mathcal{R}[\![t]\!]I : s(\mathtt{x}) \in [\ell_\mathtt{x}, h_\mathtt{x}]$ as required.

II.U. Verification in the abstract

Of course, as in mathematics, to prove a result, a stronger one is often needed. So, proving specifications at some level of abstraction often requires much more precise abstractions of the collecting semantics.

An example is the rule of signs $\mathtt{pos} \times \mathtt{pos} = \mathtt{pos}$, $\mathtt{neg} \times \mathtt{neg} = \mathtt{pos}$, $\mathtt{pos} \times \mathtt{neg} = \mathtt{neg}$, etc., where $\gamma(\mathtt{pos}) \triangleq \{z \in \mathbb{Z} \mid z \geqslant 0\}$ and $\gamma(\mathtt{neg}) \triangleq \{z \in \mathbb{Z} \mid z \leqslant 0\}$). The sign abstraction is complete for multiplication (knowing the sign of the arguments is enough to determine the sign of the result) but incomplete for addition ($\mathtt{pos} + \mathtt{neg}$ is unknown). However, if an interval is known for the arguments of an addition, the interval of the result, hence its sign, can be determined for sure. So, intervals is the most abstract abstraction which is complete for determining the sign of additions.

In general, a most abstract complete abstraction to prove a given abstract specification does exist[32] but is unfortunately uncomputable, even for a given program. In practice, one uses a reduced product of different abstractions which are enriched by new ones to solve incompleteness problems, a process which, by undecidability, cannot be fully automatizable — e.g., because designing efficient data structures and algorithms for abstract domains is not automatizable (see for example Sect. III.I.1), which is a severe limitation of automatic refinement of abstractions.[33]

III. Verification of Synchronous Control/Command Programs

We now briefly discuss Astrée, a static analyzer for automatically verifying the absence of runtime errors in synchronous control/command embedded C programs[2,29,34–37] successfully used in aeronautics[3] and aerospace[4] and now industrialized by AbsInt.[38,39]

III.A. Subset of C analyzed

Astrée can analyze a fairly large subset of C 99. The most important unsupported features are: dynamic memory allocation (`malloc`), non-local jumps (`longjmp`), and recursive procedures. Such features are most often unused (or even forbidden) in embedded programming to keep a strict control on resource usage and control-flow. Parallel programs are also currently unsupported (although parallel constructs are not strictly speaking in the language, but rather supported by special libraries), but some progress is made to support them (Sect. VI).

Although Astrée can analyze many programs, it cannot analyze most of them precisely and efficiently. Astrée is specialized for control/command synchronous programs, as per the choice of included abstractions. Some generic existing abstractions were chosen for their relevance to the application domain (Sect. III.H), while others were developed specially for it (Sect. III.I).

Astrée can only analyze stand-alone programs, without undefined symbols. That is, if a program calls external libraries, the source-code of the libraries must be provided. Alternatively, *stubs* may be provided for library functions, to provide sufficient semantic information (range of return values, side-effects, etc.) for the analysis to be carried out soundly. The same holds for input variables set by the environment, the range of which may have to be specified.

III.B. Operational semantics of C

Astrée is based on the C ISO/IEC 9899:1999/Cor 3:2007 standard,[40] which describes precisely (if informally) a small-step operational semantics. However, the standard semantics is high-level, leaving many behaviors not fully specified so that implementations are free to choose their semantics (ranging from producing a consistent, documented outcome, to considering the operation as undefined with catastrophic consequences

when executed). Sticking to the norm would not be of much practical use for our purpose, that is, to analyze embedded programs that are generally not strictly conforming but rely instead on specific features of a platform, processor, and compiler. Likewise, Astrée makes semantics assumptions, e.g., on the binary representation of data-types (bit-size, endianess), the layout of aggregate data-types in memory (structures, arrays, unions), the effect of integer overflows, etc. These assumptions are configurable by the end-user to match the actual target platform of the analyzed program (within reasonable limits corresponding to modern mainstream C implementations), being understood that the result of an analysis is only sound with respect to the chosen assumptions.

Astrée computes an abstraction of the semantics of the program and emits an alarm whenever it leads to a runtime error. Runtime errors that are looked for include: overflows in unsigned or signed integer or float arithmetics, integer or float divisions or modulos by zero, integer shifts by an invalid amount, values outside the definition of an enumeration, out-of-bound array accesses, dereferences of a NULL or dangling pointer, of a mis-aligned pointer, or outside the space allocated for a variable. In case of an erroneous execution, Astrée continues with the worst-case assumption, such as considering that, after an arithmetic overflow, the result may be any value allowed by the expression type (although, in this case, the user can instruct Astrée to assume that a modular semantics should be used instead). This allows Astrée to find all further errors following any error, whatever the actual semantics of errors chosen by the implementation. Sometimes, however, the worst possible scenario after a runtime error is completely meaningless (e.g., accessing a dangling pointer which destroys the program code), in which case Astrée continues the analysis for the non-erroneous cases only. An execution of the program performing the erroneous operation may not actually fail at the point reported by Astrée and, instead, exhibit an erratic behavior and fail at some later program point not reported by Astrée. In all cases, the program has no runtime error if Astrée does not issue any alarm, or if all executions leading to alarms reported by Astrée can be proved, by external means, to be impossible.

III.C. Flow- and context-sensitive abstractions

Static analyses are often categorized as being either flow-sensitive or flow-insensitive, and either context-sensitive or context-insensitive. The former indicates whether the analysis can distinguish properties holding at some control point and not other ones, and the later whether the analyzer distinguishes properties at distinct call contexts of the same procedure.

Astrée is both flow- and context-sensitive, where a call context means here the full sequence of nested callers. Indeed, in the collecting semantics, the set of program states S is decomposed into a *control* and a *data* components: $S \triangleq C \times D$. The control component C contains the syntactic program location of the next instruction to be executed, as well as the stack of the program locations in the caller functions indicating where to jump back after a return instruction. The data component D contains a snapshot of the memory (value of global variables and local variables for each activation record). Full flow- and context-sensitivity is achieved by partitioning (Sect. II.Q), i.e., keeping an abstraction of the data state D (Sect. III.F) for each reachable control state in C. This simple partitioning simplifies the design of the analysis while permitting a high precision. However, it limits the analyzer to programs with a finite set of control states C, i.e., programs without unbounded recursion. This is not a problem for embedded software, where the control space is finite, and indeed rather small. More abstract control partitioning abstractions must be considered in cases where C is infinite or very large (e.g., for parallel programs, as discussed in Sect. VI).

III.D. Hierarchy of parameterized abstractions

Astrée is constructed in a modular way. In particular, it employs abstractions parameterized by abstractions, which allows constructing a complex abstraction from several simpler ones (often defined on a less rich concrete universe). Indeed, although Astrée ultimately abstracts a collecting semantics of partial traces, the trace abstraction is actually a functor able to lift an abstraction of memory states (viewed as maps from variables in $\mathtt{V}$ to values in $\mathbb{V}$) to an abstraction of state traces (viewed as sequences in $(C \times (\mathtt{V} \to \mathbb{V}))^*$). It handles all the trace-specific aspects of the semantics, and delegates the abstraction of states to the memory domain. The memory abstraction is in turn parameterized by a pointer abstraction, itself parameterized by a numerical abstraction, where each abstraction handles simpler and simpler data-types. This gives:

$$\text{trace abstraction of } \wp((C \times (\mathtt{V} \to \mathbb{V}))^*) \text{ (Sect. III.E)}$$
$$\downarrow$$
$$\text{memory abstraction of } \wp(\mathtt{V} \to \mathbb{V}) \text{ (Sect. III.F)}$$
$$\downarrow$$
$$\text{pointer abstraction of } \wp(\mathtt{V}_c \to \mathbb{V}_b) \text{ (Sect. III.G)}$$
$$\downarrow$$
$$\text{product of numerical abstractions of } \wp(\mathtt{V}_n \to \mathbb{R}) \text{ (Sects. III.H–III.I)}$$

An improvement to any domain will also benefit other ones, and it is easy to replace one module parameter with another. The numerical abstraction is itself a large collection of abstract domain modules with the same interface (i.e., abstracting the same concrete semantics) linked through a reduced product functor (Sect. III.J), which makes it easy to add or remove such domains.

III.E. Trace abstraction

Floyd's method[13] is complete in the sense that all invariance properties can be proved using only state invariants, that is, sets of states. However, this does not mean it is always the best solution, in the sense that such invariants are not always the easiest to represent concisely or compute efficiently. It is sometimes much easier to use intermittent invariants as in Burstall's proof method[11] (i.e., an abstraction of the partial trace semantics[12]).

In particular, in many numerical abstract domains, all abstract values represent *convex* sets of states (for instance, this is the case of intervals, octagons, polyhedra): this means that the abstract join operation will induce a serious loss of precision in certain cases. For instance, if variable `x` may take any value except 0, and if the analysis should establish this fact in order to prove the property of interest (e.g., that a division by `x` will not crash), then we need to prevent states where $\mathtt{x} > 0$ from being confused with states where $\mathtt{x} < 0$. As a consequence, the set of reachable states of the program should be partitioned carefully so as to avoid certain subsets be joined together. In other words, context sensitivity (Section III.C) is not enough to guarantee a high level of precision, and some other mechanisms to distinguish sets of reachable states should be implemented.

The main difficulty which needs to be solved in order to achieve this is to compute automatically good partitions. In many cases, such information can be derived from the control-flow of the program to analyze (e.g., which branch of some if-statement was executed or how many times the body of some loop was executed). In other cases, a good choice of partitions can be provided by a condition at some point in the execution of the program, such as the value of a variable at the call site of a function.

To formalize this intuition, we can note that it amounts to partitioning the set of reachable states at each control state, depending on properties of the whole executions before that point is reached. This is the reason why Astrée abstracts partial traces rather than just reachable states. An element of the trace partitioning abstract domain[41,42] should map each element of a *a finite partition* of the traces of the program to an abstract invariant. Thus, it is a partitioning abstraction (Sect. II.Q), at the level of traces.

Let us consider a couple of examples illustrating the impact of trace partitioning. Embedded software often need to compute interpolation functions (in one, two, or more dimensions). In such functions, a fixed input grid is supplied together with output values for each point in the grid. Typical interpolation algorithms first localize in which cell in the grid the input is, and then apply linear or non-linear local interpolation formulas. In the case of regular grids as in the left of Fig. 1, the localization can be done using simple arithmetic, whereas in the more general case of non-regular grids, as in the right, localizing the input may be done using a search loop. In all cases, the interpolation can be precisely analyzed only if a close relationship between the input values and the grid cells can be established, so that the right interpolation formula can be applied to the right (abstract) set of inputs. Trace partitioning allows expressing such relations, with invariants which consist in conjunctions of properties of the form "if the input is in cell i, then the current state satisfies condition p_i". This is possible since the cell the input is contained in is an abstraction of the history of the execution: for instance, when the cell is determined using a loop, it is determined by the number of iterations spent in that loop. Such invariants allow for a fine analysis of such codes, since it amounts to analyzing the interpolation function "cell by cell". Last, we can note that this partitioning should only be local, for efficiency reasons.

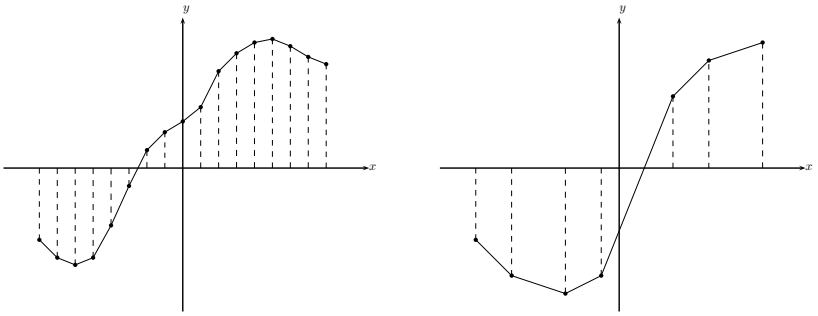


Figure 1. Regular and irregular interpolations.

In practice, this abstraction consists in a functor, which lifts a memory abstract domain (i.e., abstracting elements of $\wp(\mathtt{V} \to \mathbb{V})$) into a domain which abstracts elements of $\wp((C \times (\mathtt{V} \to \mathbb{V}))^*)$. Abstract operations implemented by this functor can be classified into three categories:

- *partition creation*, by splitting existing partitions, e.g., at the entry of an if statement, or at each iteration in a loop;
- *partition collapse*, by merging (some or all) existing partitions;
- *underlying operations*, i.e., operations supported by the underlying domain, which can be realized by applying the underlying operation independently on each partition.

Partition creation and collapse are usually guided by *heuristics*, which point out cases where trace partitioning could be helpful: for example, when several if statements test correlated conditions, partitioning the first one may result in increased precision for the next ones. However, the partitioning is *dynamic*, which means the sets of partitions are chosen during the analysis, and not fixed statically. This ensures both precision and efficiency.

III.F. Memory abstraction

Given a concrete program state $(c, d) \in S = C \times D$, its data part $d \in D$ represents a snapshot of the memory, i.e., it associates a value to each variable live in the control state $c \in C$ (including global variables and local variables in all active stack frames). Let us denote by $\mathtt{V}(c)$ this variable set, and by $\mathbb{V}$ the universe of variable values of any type. Then, $d \in \mathtt{V}(c) \to \mathbb{V}$. As Astrée partitions the memory state with respect to the control state, in order to design an abstraction of $\wp(S)$ it is sufficient to design an abstraction of $\wp(\mathtt{V}(c) \to \mathbb{V})$ for each $\mathtt{V}(c)$, i.e., the set of variables in each abstract memory state is fixed and statically known. This simplification is only possible because Astrée takes care not to abstract together program states from different control locations and does not support dynamic memory allocation.

The C language offers a rich type system that includes a few fixed *base types* (machine integers of various size and signedness, floats of various size, pointers) as well as user-defined *aggregate types* (possibly nested structures and arrays) and *union types*. Disregarding union types for now, variables of non-base type can be decomposed statically into finite collections of disjoint *cells* of base type. The role of the memory abstraction is to manage this mapping and "dumb down" expressions and pass them to a parameter domain abstracting sets in $\wp(\mathtt{V}_c \to \mathbb{V}_b)$ for any given finite cell set $\mathtt{V}_c$, $\mathbb{V}_b$ being the set of integers, floating-point, and pointer values. One way to perform this decomposition is to recursively flatten all aggregates, such as considering a variable `struct { int a; char b; } v[2]` as four distinct cells $\mathtt{V}_c \triangleq \{ c_1, c_2, c_3, c_4 \}$, where c_1 stands for `v[0].a`, c_2 for `v[0].b`, c_3 for `v[1].a`, and c_4 for `v[1].b`. This natural and most concrete choice achieves full field-sensitivity. However, it may become memory intensive for large data-structures, and it is uselessly detailed to represent uniform arrays where all elements have similar properties. Thus, Astrée allows representing several concrete cells by a single abstract cell by *folding* arrays. Folding the variable `v`, for instance, would give two cells $\mathtt{V}'_c \triangleq \{ c'_1, c'_2 \}$, where c'_1 abstracts the union of values of `v[0].a` and `v[1].a`, while c'_2 abstracts the union of values of `v[0].b` and `v[1].b`.

As the memory domain abstracts the mapping between variables and cells, it is its responsibility to translate complex C *lvalues* appearing in expressions into the set of cells they target. This translation

is dynamic and may depend on the computed (abstract) set of possible variable values at the point of the expression, hence the necessary interaction between the memory abstraction and its parameter abstract domain. Consider, for instance, the assignment `v[i].a = v[0].b + 1` to be translated into the cell universe $\mathtt{V}_c \triangleq \{ c_1, c_2, c_3, c_4 \}$ (the case of pointer access is considered in Sect. III.G). Depending on the possible abstract value of `i` (exemplified here in the interval domain), it can be translated into either c_1 `=` c_2 `+ 1` (if `i` is $[0, 0]$), c_3 `=` c_2 `+ 1` (if `i` is $[1, 1]$), or `if (?)` c_1 `=` c_2 `+ 1 else` c_3 `=` c_2 `+ 1` (if `i` is $[0, 1]$). This last statement involves a non-deterministic choice (so-called "weak update"), hence a loss of precision. Values of `i` outside the range $[0, 1]$ lead to runtime errors that stop the program. If the folded memory abstraction $\mathtt{V}'_c \triangleq \{ c'_1, c'_2 \}$ is considered instead, then the statement is interpreted as `if (?)` c'_1 `=` c'_2 `+ 1` whatever the value of `i`, which always involves a non-deterministic choice (hence, is less precise).

We now discuss the case of union types. Union types in C allow reusing the same memory block to represent values of different types. Although not supported by the C 99 standard[40] (except in very restricted cases), it is possible to write a value to a union field, and then read back from another field of the same union; the effect is to reinterpret (part of) the byte-representation of a value of the first field type as the byte-representation of a value of the second type (so-called *type punning*). This is used to some extent in embedded software, and so, it is supported in Astrée. As for aggregate types, union types are decomposed into cells of base type. Unlike aggregate types, such cells are not actually disjoint, as modifying one union field also has an effect on other union fields. However, the memory domain hides this complexity from its parameter domain, which can consider them as fully distinct entities. The memory domain will issue extra cell-based statements to take aliasing into account.[43] Consider, for instance, the variable `union { unsigned char c; unsigned int i; } v` with two cells: c_c for `v.c`, and c_i for `v.i`. Any write to `v.i` will update c_i and also generate the assignment c_c `=` c_i `& 255` (assuming the user configured the analyzer for a little endian architecture). The memory domain of Astrée thus embeds a partial knowledge of the bit-representation of integer and floating-point types.

III.G. Pointer abstraction

Pointers in C can be used to implement references as found in many languages, but also generalized array access (through pointer arithmetics) and type punning (through pointer conversion). To handle all these aspects in our concrete operational semantics, a pointer value is considered as either a pair $\langle \mathtt{v}, o \rangle$ composed of a variable `v` and an integer offset o, or a special `NULL` or *dangling* value. The offset o counts a number of *bytes* from the beginning of the variable `v`, and ranges in $[0, \mathtt{sizeof(v)})$.

A set of pointer values is then abstracted in Astrée as a pair of flags indicating whether the pointer can be `NULL` or dangling, a set of variables (represented in extension), and a set of offset values. The pointer abstract domain maintains this information for each cell of pointer type, except for the offset abstraction which is delegated to a numerical domain through the creation of a cell of integer type. Moreover, pointer arithmetics is converted into integer arithmetics on offsets. Consider, for instance, the pointer assignment `q = p + i + 1`, where `p` and `q` have type `int*`, which is translated into c_q `=` c_p `+` c_i `+ 1` by the memory domain, where c_p, c_q, and c_i are the cells associated with respectively `p`, `q`, and `i`. The pointer domain then replaces the pointed-to variable set component of `q` with that of `p` and passes down to the numerical abstraction the statement $c_{o(q)}$ `=` $c_{o(p)}$ `+ (`c_i `+ 1) * sizeof(int)`, where $c_{o(q)}$ and $c_{o(p)}$ are the integer-valued cells corresponding to the offset components of `p` and `q`.

Additionally, the pointer abstraction is used by the memory domain to resolve dereferences in expressions. For instance, given the lvalue `*(p + i)`, the memory domain relies on the pointer domain to provide the target of c_p `+` c_i, which is returned as a set of variable/offset pairs (involving offset computation in the parameter numerical domain) and possibly `NULL` or dangling. To handle type punning, the memory abstraction is able to generate a cell for any combination of a variable, an offset (smaller than the variable byte-size), and a base type, while `NULL` and dangling accesses are considered runtime errors. Cells that overlap in memory are handled as in the case of union types.

III.H. General-purpose numerical abstractions

Through a sequence of trace, memory, and pointer abstractions, we are left to the problem of abstracting concrete sets of the form $\wp(\mathtt{V}_n \to \mathbb{R})$, for any finite set $\mathtt{V}_n$ of numerical cells. This is achieved by using a combination of several numerical abstract domains. Note that the concrete semantics is expressed using reals $\mathbb{R}$ as they include all integers and (non-special) floating-point values for all C implementations.

III.H.1. Intervals

The interval abstract domain[1,20] maintains a lower and an upper bound for every integer and floating-point cell. This is one of the simplest domain, yet its information is crucial to prove the absence of many kinds of runtime errors (overflows, out-of-bound array accesses, invalid shifts). Most abstract operations on intervals rely on well-known interval arithmetics.[44] Soundness for abstract floating-point operations (considering all possible rounding directions) is achieved by rounding lower bounds downwards and upper bound upwards with the same bit-precision as that of the corresponding concrete operation.

An important operation specific to abstract interpretation is the widening ∇ used to accelerate loops. ASTRÉE refines the basic interval widening (recalled in Sect. II.O) by using thresholds: unstable bounds are first enlarged to a finite sequence of coarser and coarser bounds before bailing out to infinity. For many floating-point computations that are naturally stable (for instance `while (1) { X = X * α + [0,β]; }`, which is stable at $\mathtt{X} \in [0, \beta/(1-\alpha)]$ when $\alpha \in [0,1)$), a simple exponential ramp is sufficient (`X` will be bounded by the next threshold greater than $\beta/(1-\alpha)$). Some thresholds can also be inferred from the source code (such as array bounds, to use for integer cells used as array indices). Finally, other abstract domains can dynamically hint at (finitely many) new guessed thresholds.

One benefit of the interval domain is its very low cost: $\mathcal{O}(|\mathtt{V}_c|)$ in memory and time per abstract operation. Moreover, the worst-case time $\mathcal{O}(|\mathtt{V}_c|)$ can be significantly reduced to a practical $\mathcal{O}(\log |\mathtt{V}_c|)$ cost by a judicious choice of data-structures. It is sufficient to note that binary operations (such as $\cup$) are often applied to arguments with only a few differing variables. ASTRÉE uses a functional map data-structure with sharing to exploit this property. This makes the interval domain scalable to tens of thousands cells.

III.H.2. Abstraction of floating-point computations

Many control/command software rely on computations that are designed with the perfect semantics of reals $\mathbb{R}$ in mind, but are actually implemented using hardware float arithmetics, which incurs inaccuracies due to pervasive rounding. Rounding can easily accumulate to cause unexpected overflows or divisions by zero in otherwise well-defined computations. An important feature of ASTRÉE is its sound support for float computations following the IEEE 754–1985[45] semantics.

Reasoning on float arithmetics is generally difficult because, due to rounding, most mathematical properties of operations are no longer true, for instance, the associativity and distributivity of $+$ and $\times$. Most numerical abstract domains in ASTRÉE rely on symbolic manipulations of expressions (such as linear algebra) that would not be sound when replacing real operators with float ones (one exception being interval arithmetics, which is easy to implement using float arithmetics on bounds). Thus, ASTRÉE implements an abstract domain[46,47] able to soundly abstract expressions, i.e., a function $f : X \to Y$ is abstracted as a (non-deterministic) function $g : X \to \wp(Y)$ such that $f(x) \in g(x)$, at least for all x in some given reachable subset R of X. g can then be soundly used in place of f, for all arguments in R. In practice, a float expression $f(\vec{\mathtt{V}})$ appearing in the program source is abstracted as a linear expression with interval coefficients $g(\vec{\mathtt{V}}) = [\alpha,\beta] + \sum_i [\alpha_i,\beta_i] \times \mathtt{V}_i$, and R is some condition on the bounds of variables. g can be fed to abstract domains on reals as $+$ and $\times$ in g denote real additions and multiplications. Interval linear expressions can easily be manipulated symbolically (they form an affine space), while the intervals provide sufficient expressiveness to abstract away complex non-linear effects (such as rounding or multiplication) as non-determinism. For instance, the C statement `Z = X + 2.f * Y`, where `X`, `Y`, and `Z` are single-precision floats, will be *linearized* as $[1.9999995, 2.0000005]\mathtt{Y} + [0.99999988, 1.0000001]\mathtt{X} + [-1.1754944 \times 10^{-38}, 1.1754944 \times 10^{e-38}]$. Alternatively, under the hypothesis $\mathtt{X},\mathtt{Y} \in [-100, 100]$, it can be linearized as $2\mathtt{Y} + \mathtt{X} + [-5.9604648 \times 10^{-5}, 5.9604648 \times 10^{5}]$, which is simpler (variable coefficients are scalars) but exhibits a larger constant term (i.e., absolute rounding error).

An additional complexity in float arithmetics is the presence (e.g., in inputs) of special values $+\infty$, $-\infty$, and *NaN* (*Not a Number*), as well as the distinction between $+0$ and -0. Thus, a float type is almost but not exactly a finite subset of $\mathbb{R}$. The presence of a special value is abstracted as a boolean flag in the abstraction of each float cell, while 0 actually abstracts both concrete values $+0$ and -0. ASTRÉE checks that an expression does not compute any special value before performing symbolic manipulations defined only over $\mathbb{R}$.

Note that, because all operations in the expression abstract domain can be broken down to real interval arithmetics, it can be soundly implemented using float arithmetics with outwards rounding, which guarantees its efficiency. Other, more precise abstractions of float computations[48] exist, but are not used in ASTRÉE as

they are more costly while the extra precision is not useful when checking solely for the absence of runtime errors.

III.H.3. Octagons

Given a finite set $\mathtt{V}_n$ of numerical cells, we denote by $Oct(\mathtt{V}_n)$ the subset of linear expressions $Oct(\mathtt{V}_n) \triangleq \{ \pm\mathtt{X} \pm \mathtt{Y} \,|\, \mathtt{X},\mathtt{Y} \in \mathtt{V}_n \}$. The octagon domain[25,47,49] abstracts a concrete set of points $X \in \wp(\mathtt{V}_n \to \mathbb{R})$ as $\alpha_{Oct}(X) \triangleq \{ \max_{x \in X} e(x) \,|\, e \in Oct(\mathtt{V}_n) \}$, that is, the tightest set of constraints of the form $\pm\mathtt{X} \pm \mathtt{Y} \leq c$ (i.e., with c minimal) that enclose X. The name *octagon* comes from the shape of $\alpha_{Oct}(X)$ in two dimensions.

The octagon domain is able to express relationships between variables (unlike non-relational domains, such as the interval domain). For instance, it can discover that, after the test `if (X>Y) X=Y`, the relation $\mathtt{X} \leq \mathtt{Y}$ holds. If, later, the range of `Y` is refined, as per a test `if (Y<=10)`, it can use this information to deduce that $\mathtt{X} \leq 10$. Although $\mathtt{X} \leq 10$ is an interval property, the interval domain does not have the necessary symbolic power to infer it. Another example is the case of loops, such as `for (i=0,x=0; i<1000; i++) { if (?) x++; if (?) x=0; }`, where the octagon domain finds the symbolic relation $\mathtt{x} \leq \mathtt{i}$ for the loop head, although `x` and `i` are never explicitly assigned nor tested together. As $\mathtt{i} = 1000$ when the loop exits, we also have $\mathtt{x} \in [0, 1000]$. The interval domain can find the former (using widenings and narrowings) but not the later (no test on `x` refines the result $[0, +\infty)$ after widening), while the octagon domain finds both.

The octagon domain is based on a matrix data-structure[50] with memory cost $\mathcal{O}(|\mathtt{V}_n|^2)$, and shortest-path closure algorithms with time cost $\mathcal{O}(|\mathtt{V}_n|^3)$. This is far less costly than general polyhedra[9] (which have exponential cost), yet octagons correspond to a class of linear relations common in programs. As the interval domain, the octagon domain can be implemented efficiently in float arithmetics. Moreover, it can abstract integer (including pointer offset) and float arithmetics (provided these are linearized, as in Sect. III.H.2), and even infer relationships between cells of different type.

III.H.4. Decision trees

The octagons abstract domain expresses linear numerical relations. However, other families of relations have to be expressed and inferred, in order to ensure the success of the verification of absence of runtime errors. In particular, when some integer variables are used as booleans, relations of the form "if `x` is (not) equal to 0 then `y` satisfies property P" may be needed, especially if part of the control-flow is stored into boolean variables.

For instance, let us consider the program `b = (x >= 6); ... if (b) y = 10 / (x - 4);`. This program is safe in the sense that no division by 0 will occur, since the division is performed only when `b` denotes the boolean value true (i.e., is not equal to 0), that is only when `x` is greater than 6 (so that `x - 4` is not 0). However, this property can be proved only if a relation between `b` and `x` is established: otherwise, if no information is known about `x` at the beginning of the program, no information will be gained at the entry of the true branch of the if statement, so that a division by 0 alarm should be raised. In such cases, linear constraints are of no help, since only the boolean denotation of `b` matters here. Thus, we use a relational abstraction where some variables are treated as boolean whereas other variables are treated as pure numeric variables. Abstract values consist in decision trees containing numeric invariants at the leaves. Internal nodes of the trees are labeled by boolean variables and the two sub-trees coming out of an internal node correspond to the decision on whether that variable is true or false. The kind of invariants that can be stored at the leaves is a parameter of the domain. By default in Astrée, the leaf abstract domain is the product of basic inexpensive domains, such as the interval or equality abstract domains. The concretization of such a tree comprises all the stores which satisfy the numeric condition which can be read at the leaf of the unique branch which it satisfies (i.e., such that it maps each variable into the boolean value assigned to it on the branch). In the above example, the decision tree we need simply states that "when `b` is true, `x` is greater than 6".

This abstraction retains some of the properties of binary decision diagrams:[51] the efficiency is greatly improved by ordering the boolean variables and using sub-tree sharing techniques.

This abstract domain defines a form of partitioning in the sense of Sect. II.Q: given a boolean tree, we can express its concretization as the join of the concretization of the boolean condition at each leaf intersected with a set of states depending on the branch. Yet, it operates in a very different manner compared to trace partitioning (Sect. III.E): indeed, the boolean decision trees abstract domain is based on partitions of the set of *states*, and not traces. This remark has a great impact on the definition of transfer functions:

partitions are not affected by control-flow branches; however, when an assignment is made, partitions may be modified. For instance, if a program contains the assignment `x = x || y`, then some partitions need to be merged (after the assignment, `x` is true if and only if either `x` or `y` was true before) or intersected (after the assignment, `x` is false if and only if both `x` and `y` were false before).

III.H.5. Packing

Although the octagon domain has a very light, cubic cost compared to many relational domains (such as polyhedra[9]), it would be too high for embedded programs with tens of thousands variables as found in aeronautics. The same complexity issue occurs for boolean decision trees. A practical solution implemented in ASTRÉE is not to try and relate all cells together, but make many small packs of cells. This solution is adequate in practice since it is usually not meaningful to track relations between all pairs of variables in the program. For octagons, a simple syntactic pre-analysis groups cells in a neighbourhood that are interdependent (used together in expressions, loop indices, etc.). For boolean decision trees, more semantic criteria are used, but the pre-analysis is still very light. An octagon is associated to each octagon pack, but no relation is kept between cells in distinct packs. The total cost thus becomes linear, as it is linear in the number of packs (which is linear in the code size, and so, in the number of cells) and cubic in the size of packs (which depends on the size of the considered neighbourhoods, and is a constant).

Packing can also be viewed as a particular case of a product abstraction. For instance, let us consider the case of octagons, given a finite set $\mathtt{V}_n$ of numerical cells, the standard octagon abstraction $Oct(\mathtt{V}_n)$ would abstract functions in $\wp(\mathtt{V}_n \to \mathbb{R})$. By contrast, the packing abstraction is based on the choice of a family $\mathcal{V}_1, \ldots, \mathcal{V}_p$ of subsets of $\mathtt{V}_n$, and using a product of abstractions $D_1 \times \ldots \times D_p$, where D_i forgets about cells not in $\mathcal{V}_i$ and uses an octagon to abstract the cells in $\mathcal{V}_i$ (thus, D_i is obtained by composing a "forget abstraction" with the abstraction into $Oct(\mathcal{V}_i)$). Note that a cell may belong to no pack at all, or to several packs. In the latter case, (interval) information may flow from one pack to another using a generic reduction mechanism (Sect. III.J).

III.I. Domain-specific numerical abstractions

III.I.1. Filters

Embedded software usually interact with a physical external environment that is driven by continuous differential equations. Streams of values can be read from this external environment by using sensors. Then, digital filters are small numerical algorithms, which are used to smooth such streams of input values and implement differential equations. In the software, these equations are discretized. Moreover, they are usually linearized as well. It is hardly possible to bound the range of the variables that are involved in digital filtering algorithms without using specific domains. In ASTRÉE, we have implemented a specific domain[26] that deals with linear filters (which encode discrete linear equations). This domain uses both quadratic inequalities (representing ellipses) and formal expansions.

More precisely, a simple linear filter is implemented by a linear recursion, where the value of a variable `o` at each loop iteration is defined as a linear combination of an input value `i` and a fixed number k of the values of the variable `o` at the last k iterations. Important cases are first order filters, when $k = 1$, and second order filters, when $k = 2$. Denoting as o_n (resp. i_n) the value of the variable `o` (resp. `i`) at the n-th loop iteration, it follows that $o_{n+k} = i_{n+k} + \sum_{1 \leq j \leq k} \alpha_j \cdot o_{n+k-j}$. An appropriate change of variables can be found[52] by factoring the polynomial $X^k - \sum_{1 \leq j \leq k} \alpha_j \cdot X^{k-j}$, so as to express the sequence $(o_n)_{n \in \mathbb{N}}$ as a linear combination of the output values of some first and second order (simple) linear filters. Then, first order filters can be analyzed accurately by using intervals and widening with thresholds (Sect. III.H.1), whereas second order filters can be analyzed by using filled ellipses relating the values of two consecutive outputs. In the later case, the templates for the constraints (that is, the coefficients of the quadratic forms which describe the ellipses) are extracted directly from the analyzed code: for instance, the appropriate quadratic form for bounding the output value of a recursion of the form $o_{n+2} = \alpha_1 \cdot o_{n+1} + \alpha_2 \cdot o_n + i_{n+2}$ is $o_{n+2} - \alpha_1 \cdot o_{n+2} \cdot o_{n+1} - \alpha_2 \cdot o_{n+1}$.

In practice, filter algorithms do not use only the last input value at each loop iteration, but a fixed number l of consecutive input values. That is to say, recursions are of the form $o_{n+k} = \sum_{0 \leq j \leq l} \beta_j \cdot i_{n+k-j} + \sum_{1 \leq j \leq k} \alpha_j \cdot o_{n+k-j}$ (instead of $o_{n+k} = i_{n+k} + \sum_{1 \leq j \leq k} \alpha_j \cdot o_{n+k-j}$). One could abstract the overall contribution of the input values at each iteration as one global value, but this would lead to a very inaccurate abstraction,

since the contributions of the same output value at successive loop iterations are likely to partially cancel each other (e.g., when the β_j do not have the same sign). A better accuracy can be obtained by isolating the constraints on the last N input values, where N is a fixed parameter. Doing so, we can express, for $n+k > N$, the value of o_{n+k} as the sum $o'_{n+k} + \sum_{0 \leq j < N} \delta_j^N \cdot i_{n+k-j}$ where the linear combination $\sum_{0 \leq j < N} \delta_j^N \cdot i_{n+k-j}$ encodes the exact contribution of the last N input values, and the new variable o'_{n+k} encodes the output value of a fictitious filter which satisfies the recursion $o'_{n+k} = \sum_{0 \leq j \leq l} \beta_j \cdot \varepsilon_j^N \cdot i_{n+k-j} + \sum_{1 \leq j \leq k} \alpha_j \cdot o'_{n+k-j}$. The coefficients (δ_j^N) and (ε_j^N) can be computed automatically. If they were computed in the real field, the coefficients (ε_j^N) would converge towards 0, so that, the bigger N is chosen, the better the accuracy of the abstraction would be (we would even converge towards the exact analysis of the filter). Yet, in the implementation of the domain, the coefficients (δ_j^N) and (ε_j^N) are safely over-approximated by lower and upper floating-point bounds. This ensures the soundness of the abstract domain, but the drawback is that, if N is chosen too big, then the domain starts loosing some accuracy because of rounding errors. The best choice for N is computed automatically for each filter in the program. Then, a safe bound for the value of the sequence (o'_n) is computed by abstracting the overall contribution of the input values at each iteration as one global value. The loss of information is amortized by the coefficients (ε_j^N) which are very small.

Filters provide an example of impossible automatic refinement. Even in the case of simple filters (of the form $o_{n+k} = i_{n+k} + \sum_{1 \leq j \leq k} \alpha_j \cdot o_{n+k-j}$), it is very difficult to bound the values of the output stream (o_n) without using quadratic inequalities. Although one can enclose an ellipse within linear inequalities (using many tangents to the ellipse), the choice of the half-planes is almost as difficult to make as discovering the ellipse itself. In practice, this can only be done by using an iteration over polyhedra while delaying the use of the widening operator as late as possible, which is very costly. The situation is even worse in the case of general filters, when one has to deal with the canceling effects between consecutive contributions of the same input values.

III.I.2. Exponential evolution in time

Critical embedded software often contain computations that would be stable if they were computed in real arithmetics, but which diverge slowly due to the accumulation of floating-point rounding errors. An example is a variable `x` that is divided by a constant `c` at the beginning of a loop iteration, and then multiplied by the same constant `c` at the end of the loop iteration (this kind of pattern usually occurs in codes that are automatically generated from a higher-level specification). Another example is when the value of a variable `x` at a given loop iteration is computed as a barycentric mean of the values of the variable `x` at some (not necessary consecutive) previous loop iterations.

We use geometric-arithmetic series[27] to safely bound such computations. The main idea is to over-approximate, for each variable `x`, the action of the loop iteration over the absolute value of `x` as a linear transformation. Doing so, we get a bound for the absolute value of `x` that depends exponentially on the loop counter `t` (counting clock ticks). More precisely, we get a constraint of the following form:

$$|\mathtt{x}| \leq (1+a)^{\mathtt{t}} \cdot \left(m - \frac{b}{1-a} \right) + \frac{b}{1-a},$$

where m is a bound on the initial absolute value of the variable `x`, `t` is the value of the loop counter, and a and b are very small coefficients inferred by the analysis. More precisely, a is of the order of magnitude of floating-point relative errors and b is of the order of magnitude of floating-point absolute errors. For instance, if `t` ranges between 0 and $3,600,000$ (which corresponds to 10 hours of computation with 100 loop iterations per second) with $a \simeq 10^{-7}$, which may correspond to computations with 32-bit floating-point arithmetics, we get $(1+a)^{\mathtt{t}} \simeq 1.43$, whereas with $a \simeq 10^{-13}$, which may correspond to computations in 64-bit floating-point arithmetics, we get $(1+a)^{\mathtt{t}} \simeq 1 + 10^{-7}$. In practice, $(1+a)^{\mathtt{t}}$ is small enough so as to prove that these slow divergences do not cause overflows.

III.I.3. Quaternions

Quaternions provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions by a tuple of four numbers. Quaternions are widely used in aerospace control/command programs. In order to handle rotations, quaternions are fitted with some algebraic operators. Basically, quaternions can be added $+$, subtracted $-$, multiplied by a scalar $\cdot$, multiplied together $\times$, and

conjugated $\overline{\cdot}$. Quaternions are usually converted to rotation matrices and conversely. Quaternion computations can cause arithmetic overflows. An interesting value for a quaternion $q = (u, i, j, k)$ is its norm $||q||$ which is defined as $||q|| \triangleq \sqrt{u^2 + i^2 + j^2 + k^2}$. Quaternions are usually meant to be normalized, that is to have always a unit norm: $||q|| = 1$. Yet, because of rounding errors and approximated algorithms, their norm may diverge along the execution of the program. Thus, quaternions are often re-normalized (or divided by their norm) so as to avoid overflows.

Hopefully, the norm behaves well with respect to algebraic operations, as stated by the following properties:

$$\begin{array}{rl} |\,||q_1|| - ||q_2||\,| \leq ||q_1 + q_2|| \leq ||q_1|| + ||q_2|| & \text{(triangle inequality)} \\ |\,||q_1|| - ||q_2||\,| \leq ||q_1 - q_2|| \leq ||q_1|| + ||q_2|| & \text{(triangle inequality)} \\ ||\lambda \cdot q|| = |\lambda| \cdot ||q|| & \text{(positive homogeneity)} \\ ||q_1 \times q_2|| = ||q_1|| \cdot ||q_2|| & \\ ||\overline{q}|| = ||q||. & \end{array} \tag{1}$$

Since it is quite difficult to prove the absence of overflows without tracking the computations over quaternions, we have designed a quaternion domain. This domain handles predicates of the form $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$, where $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4$ are four variables and I is an interval. The meaning of such a predicate is that the value of the expression $\sqrt{\mathtt{x}_1^2 + \mathtt{x}_2^2 + \mathtt{x}_3^2 + \mathtt{x}_4^2}$ ranges within the interval I. So these predicates encode the properties of interest in our domain. In order to infer such properties, we need intermediate properties, so as to encode the fact that a given variable is the given coordinate of a quaternion that is being computed. As a matter of fact, the domain also handles predicates of the form $P(\mathtt{x}, i, \phi, \varepsilon)$, where $\mathtt{x}$ is a variable, i is an integer in the set $\{1, 2, 3, 4\}$, ϕ is an arithmetic formula over quaternions as defined by the following grammar: $\phi \triangleq [\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4] \mid \lambda \cdot \phi \mid \phi_1 \times \phi_2 \mid \phi_1 + \phi_2 \mid \overline{\phi}$, and ε is a non-negative real number. The meaning of a predicate $P(\mathtt{x}, i, \phi, \varepsilon)$ is that the value of the i-th coordinate of the quaternion denoted by ϕ ranges in the interval $[x - \varepsilon, x + \varepsilon]$, this way, the number ε can be used to model the rounding errors accumulated during operations over quaternions. The interpretation of arithmetic formulas is the following: the formula $[\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4]$ denotes the quaternion the four coordinates of which are the values of the variables $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, and $\mathtt{x}_4$, whereas other constructs denote the algebraic operations over quaternions.

Whenever the four coordinates of a given quaternion have been discovered (that is to say that Astrée has inferred four predicates $P(\mathtt{x}_1, 1, \phi, \varepsilon_1)$, $P(\mathtt{x}_2, 2, \phi, \varepsilon_2)$, $P(\mathtt{x}_3, 3, \phi, \varepsilon_3)$, and $P(\mathtt{x}_4, 4, \phi, \varepsilon_4)$ where ϕ is a formula, $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, and $\mathtt{x}_4$ are four program variables, and ε_1, ε_2, ε_3, ε_4 are four non-negative real numbers), the corresponding quaternion is promoted (that is to say that Astrée infers a new predicate $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$ where the interval I is obtained by applying the formulas about norm (1) and the first triangle inequality in order to handle the contribution of rounding errors, which is encoded by the numbers ε_1, ε_2, ε_3, and ε_4). Moreover, the depth of the formulas ϕ which can occur in predicates can be bounded for efficiency purposes.

Some tuples $(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4)$ of variables can be declared as a quaternion with a norm in a given interval I by using a directive, so that the end-user can assert some hypotheses about volatile inputs. In such a case, Astrée assumes that the predicate $Q(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4, I)$ holds, without any check. Moreover, whenever the values x_1, x_2, x_3, x_4 of four variables $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$, and $\mathtt{x}_4$ are divided by the value of the expression $\sqrt{x_1^2 + x_2^2 + x_3^2 + x_4^2}$, Astrée promotes them to a new quaternion and computes an interval for its norm (taking into account rounding errors).

III.J. Combination of abstractions

Astrée uses dozens of abstract domains which can interact with each others.[29] These interactions enable Astrée to refine abstract elements (as with a partially reduced product of abstract domains (Sect. II.P)), but also to refine their predicate transformers. Special care has to be taken when reduction is used after extrapolation (widening or narrowing) steps, in order not to break the construction of inductive invariants.

In Astrée, abstract domains are implemented as independent modules that share a common interface. Each module implements some primitives such as predicate transformers (abstract assignments, abstract guards, control-flow joins) and extrapolation primitives (widening and narrowing operators). Moreover, in order to enable the collaboration between domains, each abstract domain is fitted with some optional primitives so as to express properties about abstract states in a common format which can be understood by all abstract domains. Basically, a reduction has to be requested by a computation in an abstract domain. We distinguish between two kinds of reductions: either the reduction is requested by the domain which misses an information, or by the domain which discovers an information. This asymmetry enables a fine tuning

of the reduction policy: for efficiency reasons, it is indeed important not to consider all common properties between each computation step.

We now give more details about these two kinds of reduction. As already mentioned, abstract domains can at crucial moments ask for a constraint that they miss and that they need in order to perform accurate computations. For that purpose, we use a so-called *input channel*. The input channel carries a collection of constraints that have been inferred so far. Moreover, the pool of available constraints can be updated after each computation in an abstract domain. For efficiency reasons, the input channel is dealt with in a lazy way, so that only computations that are used at least once are made; moreover, the use of a cache avoids making the same computation twice. Thanks to the input channel, a domain may ask for a constraint about a precondition (that is the properties about the state of the program before the currently computed step). For instance, at the first iteration of a second order filter, no quadratic constraint has been inferred yet; thus the filter domain asks for the range of the variables that contain the first output values of the filter, so as to build an initial ellipse. Besides, a domain may also ask for a constraint about a post-condition (that is the properties about the state after the currently computed step). For instance, in some cases, the octagon domain is not able to compute the effect of a transfer function at all. Consider for instance, the assignment `x = y` on an octagon containing `x`, but not `y`. In such a case, the octagon domain relies on the underlying interval domain to give the actual range of `x` in the post-condition. It is worth noting that the order of computations matters: only the properties which are inferred by the domains that have already made their computation are available.

As a consequence, it is necessary to provide another channel that enables the reduction of a domain by a constraint that will be computed later. For that purpose, we use a so-called *output channel*. The output channel is used whenever a new constraint is discovered and the domain which has inferred this constraint wants to propagate this constraint to the domains which have already made their computation. For instance, whenever a filter is iterated, the interval for the value of the output stream that is found by the filter domain is always more precise than the one that has been discovered by the interval domain: the bound that is found by the filter domain is sent via the output channel to the interval domain.

Most reduction steps can be seen as a reduction of the abstract state, i.e., a partially reduced product (as defined in Sect. II.P). Let us denote by D the compound domain, and by γ the concretization function, which maps each abstract element $d \in D$ to a set of concrete states. A partial reduction operator ρ is a conservative map, that is to say $\gamma(d) \subseteq \gamma(\rho(d))$ for any abstract element $d \in D$. Replacing d with $\rho(d)$ in a computation is called a reduction of an abstract state. Yet, in some cases, collaborations between domains also enable the refinement of predicate transformers. For instance, most abstract domains use linear expressions (with interval coefficients) and, whenever an expression is not linear, it is *linearized* by replacing some sub-expressions with their interval. There is no canonical way of linearizing an expression and Astrée has to use some heuristics. For instance, when linearizing the product of the values of the variables `x` and `y`, Astrée has to decide to replace either the variable `x` with its range, or the variable `y` with its range. This choice is made by asking properties about the values of `x` and `y` to the other domains (via the input channel).

Lastly, reductions can also be used to refine the result of an extrapolation (widening or narrowing) step. Such refinements must be done carefully, since they may break the extrapolation process, leading to non-termination of the analysis. Examples of wrong interactions between extrapolation and reduction can be found in [47, p. 85]. Even worse, alternating the application of a widening operator with a classic join operator during upward iterations, or intersecting at each iteration the abstract state with a constant element may lead to non-termination. More formally, we gave in a publication [29, Sect. 7] an example of a sequence $(d_n) \in D^{\mathbb{N}}$ of abstract elements in an abstract domain D that is related to a set of states by a concretization map γ and that is fitted with a join operator $\sqcup$ (such that $\gamma(d_1) \cup \gamma(d_2) \subseteq \gamma(d_1 \sqcup d_2)$), a meet operator (such that $\gamma(d_1) \cap \gamma(d_2) \subseteq \gamma(d_1 \sqcap d_2)$), a reduction operator ρ (such that $\gamma(d) \subseteq \gamma(\rho(d))$), so that none of the three sequences (a_n), (b_n), (c_n) defined as follows:

$$\begin{cases} a_1 = d_1 \\ a_{n+1} = (a_n \nabla d_{n+1}) \sqcap d_0 \end{cases} \qquad \begin{cases} b_1 = d_1 \\ b_{n+1} = \rho(b_n \nabla d_{n+1}) \end{cases} \qquad \begin{cases} c_1 = d_1 \\ c_{2\cdot n+2} = c_{2\cdot n+1} \cup d_{2\cdot n+2} \\ c_{2\cdot n+1} = c_{2\cdot n} \nabla d_{2\cdot n+1}, \end{cases}$$

is ultimately stationary.

Yet, reducing the result of a widening is very important, so as to avoid unrecoverable loss of information.

In order to solve this issue, we ask specific requirements on abstract domains, their primitives, and the reduction steps ρ_∇ that may be used after widening steps. Namely, we require that (1) each abstract domain D is a finite Cartesian product $\prod_{j\in J} D_j$ of components $(D_j, \leq_j)$ that are totally ordered sets (for instance, an interval for a given program variable is seen as a pair of two bounds, an octagon is seen as a family of bounds, etc.), that (2) the join, meet, and widening operators are defined component-wise from operators over the components (e.g., for intervals, independent widening on each bound), that (3) for each component $j \in J$, the join operator $\sqcup_j$, the meet operator $\sqcap_j$, and the widening operator ∇_j are compatible with the total order $\leq_j$ (that is to say, for any $j \in J$, $a_j, b_j \in D_j$, we have: $a_j \leq_j a_j \sqcup_j b_j$, $b_j \leq_j a_j \sqcup_j b_j$, $a_j \sqcap_j b_j \leq_j a_j$, $a_j \sqcap_j b_j \leq_j b_j$, $a_j \leq_j a_j \nabla_j b_j$, and $b_j \leq_j a_j \nabla_j b_j$) and that (4) there are no cycle of reductions between components — that is to say, there is an acyclic relation $\rightarrow$ over J, so that for any two abstract elements $d, d' \in D$ (noting $d = (x_j)_{j\in J}$, $d' = (x'_j)_{j\in J}$, $\rho_\nabla(d) = (y_j)_{j\in J}$ and $\rho_\nabla(d') = (y'_j)_{j\in J}$), we have: for any $j \in J$, $[x_j = x'_j \text{ and } \forall k \in J : k \rightarrow j \implies x_k = x'_k] \implies y_j = y'_j$. These assumptions ensure the termination of the analysis even if the reduction operator ρ_∇ is applied to each upward iterate, and if, for each component $j \in J$, some widening computations are replaced with join computations, provided that for each component $j \in J$, an unbounded number of widening steps would be applied during each infinite sequence. Last, it is worth noting that other reduction steps can be performed during the computation of predicate transformers (such as the classic closure algorithm in the octagon domain).

III.K. Abstract iterator

Formally, the concrete collecting semantics of any program can be expressed as a single fixpoint equation $X = \mathrm{lfp}^{\subseteq} F$, where the function F is derived from the text of the program. The abstract semantics has a similar form $X^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$, where $X^\sharp$ is representable in memory, $F^\sharp$ is built from calls to abstract domain functions, and the equation can be solved algorithmically by iteration with widening. However, maintaining $X^\sharp$ in memory is not feasible: due to partitioning by control state (Sect. III.C), $X^\sharp$ has as many components as program locations in the fully unrolled source code (that numbers in millions).

Instead, Astrée operates similarly to an interpreter which would execute a program following its control-flow, except that it maintains an abstract element representing many partial traces ending at the current control point instead of an environment representing a single concrete memory snapshot. For instance, if the current iterator state is the control state and abstract component pair $(c :: \ell, X^\sharp_c)$ (where :: denotes the concatenation of a stack of program locations c with a location ℓ) and the program has some assignment $X = Y$ between locations ℓ and ℓ', then the effect of the assignment on $X^\sharp_c$ is computed as $Y^\sharp_c$, and the new iterator state is $(c :: \ell', Y^\sharp_c)$. The abstract state $X^\sharp_c$ can be erased from memory. The situation is a little more complex for conditionals `if then else` and other forms of branching (such as calls to function pointers with many targets) as branches must be explored independently and then merged with an abstract union $\cup^\sharp$, which requires storing an extra abstract component. Note that such independent computations can be performed in parallel[53] on multiprocessor systems and networks to achieve better analysis times. Loops also require an extra abstract component to store and accumulate iterations at the loop head. We only need to apply the widening ∇ at the loop heads, stabilizing inner loops first and outer loops last. Moreover, more precision can be obtained by bounded *unrolling*, i.e., analysing the first few iterations of the loops separately from the rest.

The number of abstract components that need to be stored simultaneously depends only on the number of nested conditionals and loops, which is very small (i.e., a dozen) for most programs. This iteration technique is less flexible than general iterations, which can be optimized[54] while requiring that $X^\sharp$ is fully available in memory; however, it is much more memory-efficient and can scale to very large programs.

III.L. Analysis parameters

Astrée has many configuration options to easily adapt the analysis to a target program. This includes 148 command-line parameters, 32 directives that can be inserted at some point in the source-code, and two configuration files.

A first set of parameters allows altering the concrete semantics. This includes an Application Binary Interface file that sets the `sizeof` and alignment of all C types, the endianess, and the default signedness of `char` and bitfields. Then, some behaviors that are undefined in the C standard[40] and can give unexpected results can be set to be reported as errors or not (e.g., integer overflow on explicit casts, implicit casts, or arithmetic operations). The semantics of erroneous operations (such as integer overflows) can be selected

(e.g., whether to use a modular or non-deterministic semantics). Finally, extra semantic hypotheses can be provided in the form of ranges for `volatile` variables (e.g., modeling inputs from the environment), C boolean expressions that are assumed to hold at given program points (e.g., to model the result of a call to an external library which is not provided), or a maximum clock tick for which a synchronous program runs (so that ASTRÉE can bound the accumulation of rounding errors). The provided hypotheses should be reviewed with the uttermost care as these are taken for granted by ASTRÉE, and the results are sound only for executions where they actually hold.

A second set of parameters allows altering the abstraction performed by ASTRÉE, which has an influence on the efficiency and the number of false alarms, without altering the concrete semantics. A first set of flags allows enabling or disabling individual abstract domains so that, e.g., ASTRÉE does not spend its energy looking for quaternions in code that is known not to contain any. Then, several abstract domains have parameters to tweak the cost/precision trade-off. One such parameter, used in several domains, is the density of widening thresholds (more thresholds means more iterations, that is, an increased cost, but also the possibility of finding a tighter invariant and avoid a false alarm). Other aspects of loop iterations are also configurable, such as the amount of unrolling (globally or on a per-loop basis), or decreasing iterations. For costly domains that are not used uniformly on all variables (such as packed domains) or all program parts (such as trace partitioning), the user can insert local directives to force the use of these domains and improve the precision. Finally, the amount of array folding can be configured, either globally or per-variable.

Various printing facilities are available, in particular the ability to trace the abstract value of one or several variables to help discovering the origin of alarms.

III.M. Application to aeronautic industry

The first application of ASTRÉE (from 2001 up to now) is the proof of absence of runtime errors in two families of industrial embedded avionic control/command software.[3] These are synchronous C programs which have the global form:

```
declare input, output, and state variables
initialize state variables
loop for 10h
    read input variables from sensors
    update state and compute output
    output variables to actuators
    wait for next clock tick (10ms)
end loop
```

The analysis is opened loop in that the relationship between output and input variables is abstracted away (e.g., by range hypotheses on inputs and requirements on outputs). A closed loop analysis would have to take a more precise abstraction of the properties of a model of the plant into account.

A typical program in the families has around 10 K global variables (half of which are floats, the rest being integers and booleans) and 100 K to 1 M code lines (mainly in the "update state and compute output" part). The program is automatically generated from a higher-level synchronous data-flow specification, which makes the program very regular (a large sequence of instances of a few hand-written macros) but very difficult to interpret (the synchronous scheduler flattens higher-level structures and disperses computations across the whole source code). Additional challenges include domain-specific computations, such as digital filters, that cannot be analyzed by classic abstractions.

To be successful, ASTRÉE had to be adapted by abstraction parametrization to each code family, which includes designing new abstract domains (Sects. III.I.1, III.I.2), reductions (Sect.III.J), and strategies to apply costly generic domains only where needed (Sects. III.E,III.H.5). This kind of hard-coded parametrization can only be performed by the analysis designers, but it is generally valid for a full family of similar programs. Initial development of ASTRÉE and parametrisation to the first family took three years (2001–2003). The task of adapting ASTRÉE to the second program family in the same application domain (2003–2004) was easier, as many abstract domains could be reused, despite an increased software complexity and major changes in macro implementations and coding practices. In the end, user-visible configuration options (Sect. III.L) are enough (if needed at all) for the industrial end-users[3] to adapt the analysis to a specific program instance

in any supported family, without intervention from the analysis designers. On the two considered families, the analysis time ranges from 2h to 53h (depending on code size) on a 64-bit 2.66GHz Intel server using a single core, and achieves the zero false alarm goal.

III.N. Application to space industry

The second application (2006–2008) of ASTRÉE was the analysis of space software.[4] ASTRÉE could prove the absence of runtime errors in a C version of the Monitoring and Safing Unit (MSU) software of the European Space Agency Automated Transfer Vehicle (ATV). The analyzed version differs from the operational one written in Ada in that it is a C program, generated from a Scade[55] V6 model, but it is otherwise representative of the complexity of space software, in particular the kinds of numerical algorithms used.

This case study illustrates the effort required to adapt a specialized analyzer such as ASTRÉE to a new application domain. Although the MSU software has some similarity with the avionic software of Sect. III.M (both are synchronous embedded reactive control/command software featuring floating-point computations), it implements different mathematical theories and algorithms (such as quaternion computations) that need to be supported in ASTRÉE by the addition of new abstract domains (Sect. III.I.3). After specialization by the analysis designers, ASTRÉE could analyze the 14 K code lines MSU software in under 1h, with zero false alarm.

It is our hope that a tool such as ASTRÉE could be used efficiently by industrial end-users to prove the absence of runtime errors in large families of software for the embedded control/command industry, once a library of abstractions dedicated to the various application domains (automotive, power plant, etc.) is available.

III.O. Industrialization

First developed as a research prototype (starting from 2001), ASTRÉE has slowly matured into a tool usable by trained industrial engineers.[3] Its success resulted in the demand to industrialize it, so as to ensure its compliance to software quality expectations in industry (robustness, integration, graphical user interfaces, availability on all platforms, etc.), its continuous development, its distribution and support. Since 2009, ASTRÉE is developed, made commercially available, and distributed[39] by AbsInt Angewandte Informatik GmbH,[38] a company that develops various other abstract interpretation-based tools (e.g., to compute guaranteed worst case execution time[56]).

IV. Verification of Imperfectly-Clocked Synchronous Programs

For safety and design purpose, it is frequent that a synchronous control/command software is actually built as several computers connected by asynchronous communication channels. The clocks of these systems may then desynchronize and the communication channels have some latency. We now introduce a theory aiming at extending the static analysis of embedded systems to such sets of communicating imperfectly-clocked synchronous systems. This theory enables the development of a static analyzer independent of ASTRÉE.

In the previous sections of this article, programs in the C language were analyzed. In order to study the temporal properties of those systems with imperfect clocks, we assume that systems were first developed in a higher-level language as it is often the case for embedded systems: we analyze sets of *synchronous language* programs.

IV.A. Motivation

The problem of desynchronization is often neglected. Some techniques (e.g. *alternating bit protocol*) make sure to detect desynchronization and are used commonly in industry. But this may be easily implemented in erroneous way.[57] Another risk is to degrade performance. For example, consider the system in Fig. 2. It depicts two identical systems whose only computation is to perform the boolean negation of their own previous output. They should therefore implement two alternating boolean values. In the `System 1` on the left, an additional system compares the previous outputs of both systems in order to check if both units agree.

But these computations are performed according to two clocks C and C′. It may be that these clocks are synchronous. This case is depicted in the lower left part of Fig. 2. The two alternating boolean outputs of the two systems being always equal, the comparison always results in no alarm (OK statement).

But maybe the clocks C and C′ are *slightly* desynchronized by a small delay ε. This case is depicted in the lower right part of Fig.2. The two alternating boolean outputs of the two systems are then almost always equal, but they differ near every clock tick. Then, the comparison being made precisely on those tick, it always results into an alarm ("!=" statement). However, this alarm is probably unnecessary in that case, since the desynchronization delay is very small. This desynchronization delay is in practice unavoidable, since clocks are physical objects and cannot be perfect. This implementation of an alarm is therefore flawed. Such errors cannot be always discovered by hand. Their detection has to be done automatically and statically.

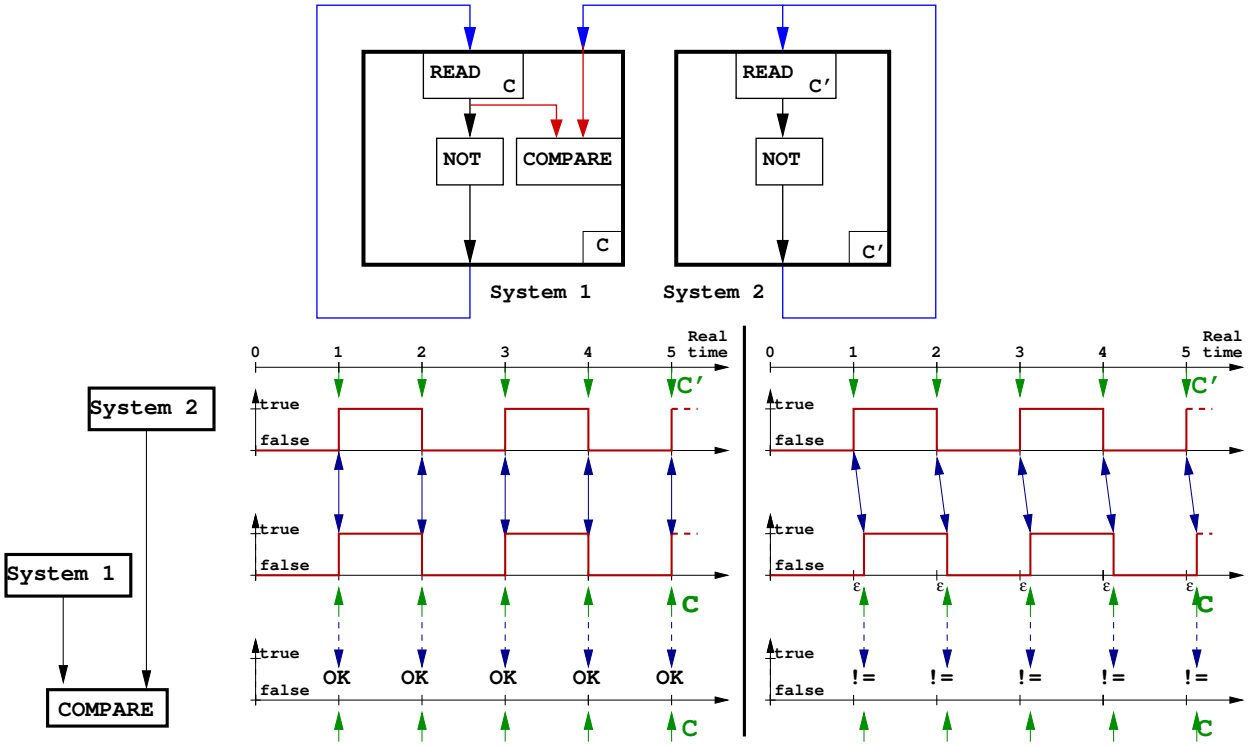


Figure 2. Example of two similar imperfectly-synchronous systems with an alarm watching differences in their outputs.

IV.B. Syntax and semantics

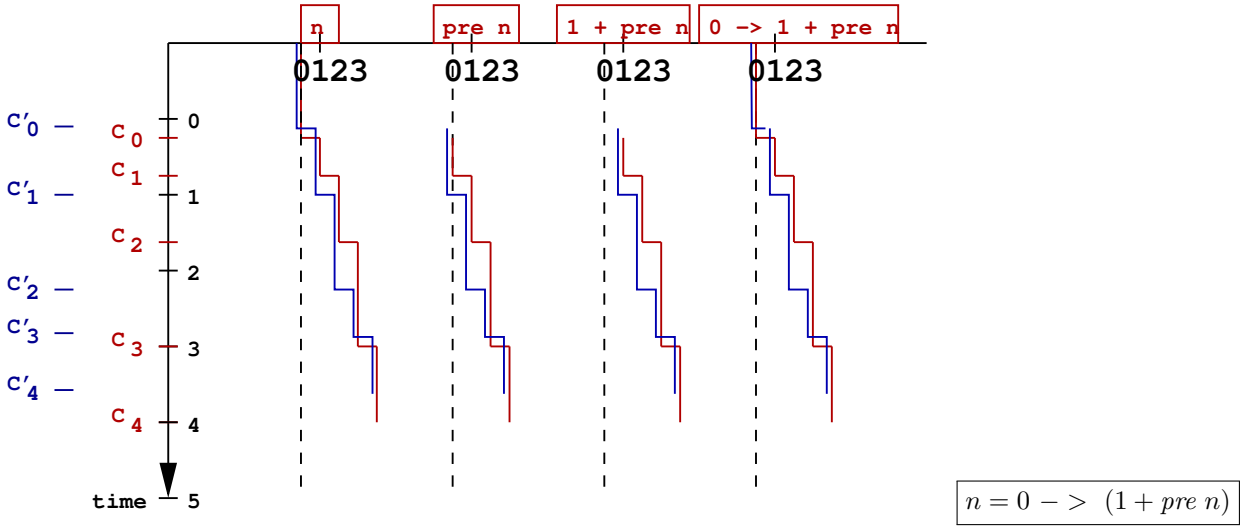
We assume that each part of the synchronous software compiled for one precise computer will execute according to the clock C of that computer with a period (the time between two consecutive clock ticks) remaining inside a known interval $[\mu_{\mathtt{C}}; \nu_{\mathtt{C}}]$, with $0 < \mu_{\mathtt{C}} \leqslant \nu_{\mathtt{C}}$. In the quasi-synchronous framework introduced formally by Caspi et al.,[58] two clocks supposed to tick synchronously are allowed to desynchronize in the following way: at most two consecutive ticks of one of the clock may happen between two consecutive ticks of the other clock. This hypothesis is quite weak, and we usually work with a clock whose parameter is such that $2 \times \mu_{\mathtt{C}} \geqslant \nu_{\mathtt{C}}$, which implies quasi-synchrony compared to a perfect clock whose period is between $\mu_{\mathtt{C}}$ and $\nu_{\mathtt{C}}$. When $\mu_{\mathtt{C}}$ is close to $\nu_{\mathtt{C}}$, our hypothesis is stronger and we expect to be able to prove more properties.

Furthermore, each communication channel *ch* has an interval $[\alpha_{ch}; \beta_{ch}]$ as parameter such that the delays between the emission of a value and its reception must always belong to this interval. The communications over a given channel are still considered serial, which means that if a value a is sent over channel *ch* before a value b, then a is received before b.

In this realistic framework, idealistic cases usually considered can still be modelled. It is then assumed that all clocks C, C', ... are perfect: $\mu_{\mathtt{C}} = \nu_{\mathtt{C}} = \mu_{\mathtt{C}'} = \nu_{\mathtt{C}'} = \ldots$ and that communications are instantaneous, i.e., $0 = \alpha_{ch} = \beta_{ch} = \alpha_{ch'} = \beta_{ch'}$ for all the channels *ch*, *ch'*, ... in the system.

Apart from this tags for clocks and communication channels, the syntax only differs from that of classic synchronous languages by the way we handle inputs and outputs. Since we allow several synchronous systems to desynchronize, some kind of buffers have to be used to store data that has been received and not yet read. In embedded systems, it is often the case that blackboards are used at entrance of each imperfectly synchronous subsystem instead of buffers. Proving that no buffer overflow may happen is indeed very complex. A blackboard is a memory cell at the end of any communication channel that is overwritten by any new value targeted at this cell, even if the previous value has not been used.

For example, a simple counter with an imperfect clock of average period 1 with 10% clock skew allowed is defined in synchronous languages by the equation $n = 0 -> (1 + pre\ n), \mathtt{C}_{[0.9,1.1]}$, where $a -> b$ means a at first cycle then b, $pre\ c$ means c but delayed by one cycle, and C is a clock of parameters $[0.9, 1.1]$. We depict below (respectively in red and blue) two very different behaviors of this counter n for two imprecise clocks C and C'.



$n = 0 -> (1 + pre\ n)$

The semantics is continuous-time in the sense that it gives a value at each point of the program at any time. This means that the semantics of a system is an element of $\wp(\mathtt{V} \to (\mathbb{R} \to \mathbb{V}))$. This semantics can be mathematically defined as the solution of continuous-time equations. For example, the trace y is in the semantics at the output point of an operator *pre* with a clock C if and only if their exists a trace x at the input point of *pre* such that $y = x \circ \pi_{\mathtt{C}}$ with:

$$\pi_{\mathtt{C}}(t) = \begin{cases} -1 & \text{if } t < \mathtt{C}_1 \\ \mathtt{C}_p + \frac{(t-\mathtt{C}_{p+1}) \times (\mathtt{C}_{p+1} - \mathtt{C}_p)}{\mathtt{C}_{p+2} - \mathtt{C}_{p+1}} & \text{when } t \in [\mathtt{C}_{p+1}, \mathtt{C}_{p+2}) \,. \end{cases}$$

This operator connects $\mathtt{C}_{p+1}$ to $\mathtt{C}_p$ and thus clearly allows the definition of a continuous-time semantics for the *pre* operator.

However, the properties of this semantics are difficult to discover automatically, since the solution of the equation is very sensitive to the parameters of this equation. We therefore abstract in a canonical way this semantics to an element of $\mathtt{V} \to (\wp(\mathbb{R} \to \mathbb{V}))$.

As presented in Sect. II.C, this semantics is abstracted as a fixpoint of a concrete operator.

IV.C. Abstraction

Even if this over-approximation of the initial semantics is now mathematically computable, it is still far from being computable statically and automatically by a computer, so that we introduce an abstract domain and operators inside this domain, that are proved sound with respect to concrete ones. Following the theory introduced in Sect. II.D, the abstract fixpoint is an over-approximation of the concrete fixpoint and thus of the semantics of the system.

This abstraction is actually a reduced product of several abstract domains. We now present two of them. The common point between these domains is that they involve an abstract continuous time (seen as $\mathbb{R}$), since we abstract sets in $\wp(\mathbb{R} \to \mathbb{V})$. This abstraction is precise and inexpensive. This is because these systems

were in fact designed in a continuous world (through differential equations) in an environment (made of space and time) that is a continuous object. In addition, using a continuous-time semantics enables the use of very well-known mathematical theories about continuous numbers which are not so frequently used in static analysis.

IV.D. Temporal abstract domains

IV.D.1. Abstract constraints

A first domain of abstract constraints[59] abstracts $\wp(\mathbb{R} \to \mathbb{V})$ as conjunctive sets of *universal* and *existential* constraints. A universal constraint over a signal $s \in \mathbb{R} \to \mathbb{V}$ is defined by a time interval $[a; b]$ and a value x, and denoted $\forall t \in [a; b] : s(t) = x$. Its concretization is the set of signals in $\mathbb{R} \to \mathbb{V}$ that take the value x during the whole time interval $[a; b]$. An existential constraint over a signal s is defined by a time interval $[a; b]$ and a value x, and denoted $\exists t \in [a; b] : s(t) = x$. Its concretization is the set of signals in $\mathbb{R} \to \mathbb{V}$ that take the value x at least once during the time interval $[a; b]$. For example, $\exists t \in [0; 1] : s(t) = \mathit{true} \wedge \exists t \in [0; 1] : s(t) = \mathit{false}$ is the abstraction of functions in $\mathbb{R} \to \mathbb{B}$ that change their boolean value at least once between $t = 0$ and $t = 1$.

The operators defined for usual operations in abstract domains $(\cup, \cap)$ as well as the backward abstract operators corresponding to synchronous language primitives $(- >, \mathit{pre}$, blackboard reading, etc.) are quite precise in this domain.

IV.D.2. Changes counting domain

A second domain of change counting[60] was designed in order to deal automatically with reasoning on the stability and the variability of systems. The abstract properties $(\leqslant k, a \blacktriangleright \blacktriangleleft b)$ and $(\geqslant k, a \blacktriangleright \blacktriangleleft b)$, for $a, b \in \mathbb{R}^+$ and $k \in \mathbb{N}$, respectively mean that behaviors do not change their value more (respectively less) than k times during the time interval $[a; b]$.

This domain is more precise for forward operators and defines a very precise reduced product with the abstract constraint domain.

An example of reduction is (with times $a < b < c < d < e < f$) when abstract an property $u = (\leqslant 1, a \blacktriangleright \blacktriangleleft e)$ interacts with the abstract properties $v = \exists t \in [b; c] : s(t) = x$ and $w = \forall t \in [d; f] : s(t) = x$. Then, if there is at least one value change between c and d, then there are actually at least two changes. Indeed, at some time $t \in [c; d)$, the value has to be some $y \neq x$, since at time d it has to be x (by w) and it changes at least once in $[c, d]$. Then, at some point $t' \in [b; c]$, the value has to be x (by v) which makes two value changes: one between t' and t, and one between t and d. This is excluded by the stability property u. As a consequence, there is no value change between c and d and, since the value at time d is x and does not change, the value has to remain equal to x during whole time interval, which can be translated into $\forall t \in [c; d] : s(t) = x$. This constraint merges with the constraint $\forall t \in [d; f] : s(t) = x$ and yields $\forall t \in [c; f] : s(t) = x$.

IV.E. Application to redundant systems

It is often the case that similar (if not identical) systems run in parallel so that, in case one system has a hardware failure, it is detected, either by the other similar systems or by a dedicated unit, and only redundant units keep performing the computation. The continuous-time semantics presented in this section has been precisely designed to prove the properties of such systems.

Another classic embedded unit aims at treating sensor values. Sensor values are indeed very unstable and usually get stabilized by a synchronous system. The temporal abstract domains we introduced are precise as well on the code for those systems.

A prototype static analyzer has been developed implementing the two temporal abstract domains presented as well as other, less central domains. This prototype is independent from Astrée (Sect. III) and Thésée (Sect. VI).

The prototype analyzer was able to prove some temporal specification of redundant systems with a voting system deciding between them. Furthermore, when some property did not hold, looking at the remaining abstract set sometimes led to actual erroneous traces in altered implementations.

An example analysis involved the code used in industry as a test for such systems where clocks may desynchronize and communication might be delayed. No hypothesis was given on the inputs of the studied

system but a specification was given for the output. We started several automatic analyses with several hypothesis of value k as input stability, which led to discovering a constant value k_0 such that:

- with an input stability of k_0 milliseconds at least, the analyzer could prove the specification;
- with an input stability of $\frac{2}{3} \times k_0$ milliseconds or less, the analyzer could not prove the specification, but in the computed abstract result, it was very easy to find (and this process could have been made automatic) a counter-example to the specification;
- with an input stability between $\frac{2}{3} \times k_0$ and k_0, the analyzer could not prove the specification, while the abstract result did not provide any obvious counter-example to the specification. It is therefore unknown whether the specification holds in this case.

This result is very interesting since it demonstrates the necessity to stabilize input signals. But the analyzer also provides a safe lower bound for this stabilization. In order for the analyzer to get closer to the optimal stabilization, i.e., suggest a smaller minimal stability requirement, a new abstract domain may be added in a modular way. Then, the properties proved by the previous domains would still be discovered and, thanks to the added precision of the new domain, new properties may be discovered as well.

V. Verification of Target Programs

V.A. Verification requirements and compilation

In Sect. III, we addressed the verification of properties at the source level. However, source level code is compiled into assembly code prior execution; thus, one may object that the verification performed at the source level may not be convincing enough or may not be sound at all. Indeed, if the compiler itself contains a bug, then a C code which is correct according to the C semantics may be compiled into a target program which may crash abruptly. More generally, certification authorities usually require a precise definition of the properties which are validated (by verification or by testing) to be provided *at the target level*, which requires, at least, a good correspondence between the source and the target level to be established. As a consequence, certification norms such as DO-178B[61] often include target code verification requirements.

An alternative solution is to perform all verification tasks at the target level; yet, this approach is not adequate for all properties. It is well adapted for the verification of properties which can be defined reliably only at the compiled code level, such as worst case execution time properties.[56] On the other hand, inferring precise invariants in order to prove the absence of runtime errors is harder at assembly level, due to the loss of control and data structures induced by compilation. For instance, expressions are translated into sequences of atomic, lower-level operations, which makes it harder for an analyzer to compute a precise invariant for the whole operation: as an example, a conversion from integer data-type to floating-point data-type on a PowerPC target typically involves dozens of instructions, including bit masks and bit-field concatenations.

V.B. Semantics of compilation

When a direct target code level verification is too hard, other approaches can be applied, which allows exploiting the results of a source level verification. Such approaches are based on a common pattern: either compilation should be *correct* (for some definition of correctness which we need to make explicit) or the verification of properties at the target level should be expected to fail. Thus, we need to formalize compilation correctness first.

The abstract interpretation frameworks allows defining program transformations at the semantic level.[62] Indeed, a program transformation usually aims at preserving some property of the semantics of the program being transformed, which we can define as an abstraction of the standard semantics. Then, the correctness of the transformation boils down to a condition (such as equivalence) at this abstract level. Furthermore, the notion of fixpoint transfer extends to this case, and allows a *local* definition of the transformation correctness.

In the case of compilation, and when the compiler performs no optimization, the target level abstraction should forget about intermediate compilation steps; then, the correctness of the compilation can be expressed as a tight relation (such as the existence of a bi-simulation between the non-standard semantics, or as an inclusion between them). This relation is usually characterized by a relation between subsets of control points and variables of both the source and the target programs. A subtle yet important property is that this relation

can only be defined for *valid* source programs: for instance, if a C program violates the C semantics (e.g., exhibits undefined behaviors), it is not possible to give a meaning for the compilation correctness in that case. When the compiler performs optimization, this simple definition can be extended, using more complex abstractions; for instance, dead variable elimination will cause some variables to be abstracted away at some program points.

We have formalized this framework.[63] Given a compiler and compilation option, this framework allows expressing what property we should expect compilation to preserve, so that we can derive how to exploit the results of source code verification so as to verify the target code.

V.C. Translation of invariants applied to target level verification

When the relation between program control points and variables in the source and compiled codes is known, and after invariants are computed for the source code, it is possible to translate automatically those invariants into invariants for the target code. When the compilation is indeed correct, the translated invariants are sound by construction. However, it is also possible to check the soundness of the translated invariants independently, which is much easier than inferring precise invariants for the compiled program directly. When this independent check succeeds, the invariants are provably correct, independently from any assumption on the correctness of the compiler or source code analyzer. Nevertheless, the fact that we expect the compilation to be correct is very important here, as this guides the translation of invariants.

This is the principle of *proof carrying codes*.[64] This techniques generalizes to invariants computed by abstract interpretation-based static analyses of the source code.[65]

A disadvantage of such techniques is that not all kinds of invariants can be efficiently checked at assembly level; actually, case studies[65] show that significant refinements are required in order to make the checking of invariants of the same kind as those produced by Astrée succeed: indeed, checking invariants requires the sequence of instructions corresponding to each statement in the source code be analyzed precisely and part of the structure lost at compile time has to be reconstructed by the invariant checker.

V.D. Verification of compilation

An alternate solution consists in verifying the compilation itself. The principle of this approach is simple: we need to verify that the target code and the source code are correct in the sense stated above, i.e., that the tight relation between their abstract semantics holds. This can either be proved automatically after each compilation run (this technique is known as *translation validation*[63,66,67]) or only once, for the compiler itself (this technique is usually called *compiler verification*[68]). In both cases, the compilation is verified correct at some level of abstraction in the sense of Sect. V.B. Moreover, the proof of equivalence is based on a set of *local* proofs: to prove the compilation correct, we only need to prove that each execution step in the source level is mapped into an equivalent step in the assembly and the converse (this is actually a consequence of the fixpoint transfer-based definition of compilation correctness).

When the compilation verification succeeds, it is possible to translate interesting properties to the target code, provided they can be established at the source level. This is the case for the absence of runtime errors, for instance. On the other hand, the correctness of compilation verification *cannot* usually be guaranteed if the source code cannot be proved correct: indeed, as we observed in Sect. V.B, the compilation correctness cannot be defined in the case where the source program itself is not valid. Thus, the verification of the compilation should follow the verification of the absence of runtime errors at the source level, e.g., using a tool such as Astrée.

Translation validation algorithms rely on automatic equivalence proof algorithms, which are either based on model-checking[66] or special purpose provers.[63,67] These tools consider the compiler as a black box; thus, they apply to a range of compilers. On the other hand, compiler verification[68] requires a manual formalization and proof of the compiler; thus, it needs to be done by the compiler designers; however, once a compiler is verified, there is no need to run a (incomplete) verification tool to verify each compilation run.

VI. Verification of Parallel Programs

We now discuss on-going work on the verification of the absence of runtime errors in embedded parallel C software for realtime operating systems, with application to aeronautics.

VI.A. Considered programs

Parallel programming allows a better exploitation of processors, multi-processors, and more recently, multi-core processors. For instance, in the context of *Integrated Modular Avionics* (IMA), it becomes possible to replace a physical network of processors with a single one that executes several tasks in parallel (through multi-tasking with the help of a scheduler). The use of light-weight processes in a shared memory (e.g., as in POSIX[69] threads) ensures an efficient communication between the tasks. There exists several models of parallel programming, giving rise to as many different concrete semantics. Moreover, parallel programming is used in various widely different applications, requiring different abstractions. As in Sect. III, in order to design an effective analyzer that eschews decidability and efficiency issues, we will focus on one such model and one application domain only.

We will focus on applications for realtime operating systems and, more precisely, ARINC 653[70] (*Aeronautical Radio Inc.*'s Avionics Application Standard Software Interface) which is an avionic specification for safety-critical realtime operating systems (OS). An application is composed of a bounded number processes that communicate implicitly through a shared memory and explicitly through synchronisation objects (events, semaphores, message queues, blackboards, etc.) provided by the OS. The realtime hypothesis imposes that OS resources are only created during a special initialization phase of the program. We assume they do not vary from one run of the program to the other. It also indicates that the scheduling adheres strictly to process priority (which we assume is fixed): a lower priority process can run only if higher priority processes are blocked in system calls (this is unlike the default, non-realtime POSIX[69] semantics used in most desktop computers and servers). We also assume that all the processes are scheduled on a *single* processor. Finally, we assume the same restrictions as in Sect. III: programs are in a subset of C without dynamic memory allocation, `longjmp`, nor recursive procedures. In addition to run-time errors, we wish to report data-races, but do not consider other parallel-related threats (such as deadlocks, livelocks, starvation).

VI.B. Concrete collecting semantics

An execution of the program is an interleaving of instructions from each process obeying some scheduling rules (Sect. VI.B.2). Our concrete collecting semantics is then a prefix trace semantics (in $\wp(S^*)$) corresponding to prefixes of sequences of states (in S) encountered during any such execution. A concrete state can be decomposed into several components $S = D \times U \times (C_1 \times D_1) \times \ldots \times (C_n \times D_n)$, where n is the number of processes, D is a snapshot of the global, shared variables, U is the state of the scheduler (including the state of all synchronisation objects), C_i is the control state of process i (a stack of control points), and D_i is a snapshot of the local, private variables available for process i at its control state (i.e., the local variables in all activation records). The D_i and C_i components are identical to those used in synchronous programs, while the D and U components are described below.

VI.B.1. Shared memory

The main property of a memory is that a program will read back at a given address the last value it has written to that address. This is no longer true when several processes use a shared memory. Which values a process can read back is defined by a *memory consistency model*.[71] The most natural one, *sequential consistency*,[72] assumes that memory operations appear to be performed in some global sequential order respecting the order of operations in each process. This means that the D state component can be modeled as a map from global variables to values $\mathtt{V} \to \mathbb{V}$.

Unfortunately, this model is not valid in practice as enforcing sequential consistency requires some heavy-weight measures from the compiler (e.g., preventing most optimizations, introducing fences) with a huge negative impact on performance. Following Java,[73] the upcoming C++ standard[74] only requires that perfectly synchronised programs (i.e., where all accesses to shared variables are protected by synchronisation primitives, such as mutexes) behave in a sequentially consistent way (e.g., because calls to synchronisation primitives are recognized and treated specially by the compiler, disabling optimisation and generating fences). In C++, unprotected accesses will result in an undefined behavior while, in Java, a weak memory consistency model[75] specifies the set of possible behaviors. Consider, for instance, the following incorrectly synchronized program (inspired from Dekker's mutual exclusion algorithm):

```
init:  flag1 = flag2 = 0
```

process 1:	process 2:
`flag1 = 1;`	`flag2 = 1;`
`if (!flag2)`	`if (!flag1)`
`{`	`{`
`  /* critical section */`	`  /* critical section */`

In the Java memory model, both processes can enter their critical section simultaneously. The rationale is that, due to process-wide program optimisation without knowledge of other processes, a compiler might assume that, e.g., in process 1, the write to `flag1` and the read from `flag2` are independent and can be reordered, and the same for process 2, `flag2` and `flag1` respectively. As a consequence, each process can read the other process' flag *before* setting its own flag. Multi-processors with out-of-order execution or not fully synchronized caches can also cause similar behaviors, even in the absence of compiler optimizations.

There is currently no memory consistency model for C; however, we need to choose one in order to define our concrete semantics. It is safe to assume that, as C++, C will guarantee sequential consistency for programs without data-race. We also draw from the Java model[75] to give a semantics to unprotected accesses, so that we can analyze the behavior of a program after a data-race. More precisely, assume that a run of a process p performs a sequence of synchronisation operations at times $t_1 < \ldots < t_n$, and a run of another process p' performs two synchronisation operations at time $t'_1 < t'_2$; denote i and j such that $t_i \leq t'_1 < t_{i+1}$ and $t_j \leq t'_2 < t_{j+1}$; then, a read from a variable `v` in p' between t'_1 and t'_2 can return either: 1) any value written to `v` by p between t_i and t_{j+1} (unsynchronized access), or 2) the last value written to `v` by p before t_i if any or its initial value if none (synchronized access), or 3) the last value written to `v` by p' if either the value was written after t'_1 or there is no write from p to `v` before t_i. This can be formalized in fixpoint form[76,77] and requires the D state components to store sets of values written to global variables by processes (instead of a simple map). This semantics is sound to analyze data-race-free programs, and it is also sound for programs with data-races under reasonable hypotheses[75] on the optimizations used by the compiler and the hardware consistency model enforced by the processor(s).

VI.B.2. Scheduling and synchronisation

The U state component in our concrete semantics models the scheduler state, which in turns defines which process can run and which must wait. Firstly, it maintains the state of synchronisation objects, e.g., for each mutex (there are finitely many), whether it is unlocked or locked by a process (and which one). Secondly, it remembers, for each process, whether it is waiting for a resource internal to the system (e.g., trying to lock an already locked mutex), for an external event (e.g., a message from the environment or a timeout), or is runnable (i.e., either actually running or preempted by another process). As we assume that the scheduler obeys a strict real-time semantics and there is a single processor, only one process can be scheduled in a given state: the runnable process with highest priority. All higher priority processes are waiting at a system call, while lower priority processes can be either waiting at a system call, or be runnable and preempted at any program point.

The execution of a synchronisation primitive by the running process updates the scheduling state U. For instance, trying to lock an already locked mutex causes the process to enter a wait state, while unlocking a locked mutex causes either the mutex to be unlocked (if no process is waiting for it), or the mutex ownership to pass to the highest priority process waiting for it (which then becomes runnable, and possibly preempts the current process). Moreover, U might change due to external events, which we assume can take place at any time. For instance, a process performing a timed wait enters a non-deterministic wait phase but can become runnable at any time (as we do not model physical time), and possible preempt a lower priority running process.

VI.C. Abstraction

Our prototype analyzer of parallel embedded realtime software, named Thésée, is based on Astrée (Sect. III). It has a similar structure, reuses most of its abstractions (e.g., general-purpose numerical abstractions for D_i, trace partitioning with respect to C_i, etc.) and adds some more.

VI.C.1. Control and scheduler state abstraction

ASTRÉE was fully flow- and context-sensitive thanks to a partitioning with respect to the control state C (Sect. III.C). Full flow- and context-sensitivity for a parallel program, i.e., partitioning with respect to $G = U \times C_1 \times \cdots C_n$, is not practical as there are generally too many reachable control points in G.

To solve this problem, we first make use of an abstraction,[78] derived from proof techniques for parallel programs,[79] that consists in decomposing a global property partitioned on G into several local properties, one for each process. More precisely, given a process i, we can represent a set of states as a map from $U \times C_i$ to $\wp(D \times D_i \times \prod_{j \neq i}(C_j \times D_j))$. Then, we abstract away all the information purely local to other processes, which gives a map from $U \times C_i$ to $\wp(D \times D_i)$. We also abstract U into a domain U_i that only distinguishes, for each mutex, whether it is unlocked, locked by process i, by a process of higher priority than i, or of lower priority. The set of reachable configurations in $U_i \times C_i$ is now small enough that we can perform a partitioning of an abstraction of $\wp(D \times D_i)$ (described below) with respect to $U_i \times C_i$.

VI.C.2. Interference abstraction

Our map from $U_i \times C_i$ to $\wp(D \times D_i)$ can be decomposed into a map from $U_i \times C_i$ to $\wp(D_i)$ and a map from $U_i \times C_i$ to $\wp(D)$. As D_i is a simple map from local, non-shared variables to values, $\wp(D_i)$ is abstracted exactly as in the synchronous case (Sect. III.F). Following the concrete semantics of weakly consistent shared memory (Sect. VI.B.1), $(U_i \times C_i) \to \wp(D)$ is abstracted as a disjunction. Firstly, we track for each $(u, c) \in (U_i, C_i)$ an abstraction of the reachable environments ignoring the effect of other processes (i.e., considering the variables as non-shared). Secondly, we track for each $u \in U_i$ and each variable `v` an abstraction of *write actions*. These consist in two components: the set of all the values written to `v` while in state u (to handle incorrectly synchronized accesses), and the set of values written last before exiting state u (to handle correctly synchronized accesses). This abstraction of write-actions is both flow-insensitive and non-relational (e.g., using the interval domain).

When process i writes to a global variable, both the non-shared and the write-action abstractions are updated. When process i reads from a global variable, the value read can come from both the non-shared abstraction of process i and the write actions of other processes $j \neq i$. Such write actions will be empty for global variables that are not actually shared, and these are abstracted as precisely as local ones (i.e., in a flow-sensitive and relational way). Moreover, we only read from write actions associated to scheduler states not mutually exclusive with that of process i, which allows handling synchronization (e.g., exploiting mutexes or process priorities) quite precisely.

A limitation of this abstraction, though, is that we cannot discover relational invariants holding at critical section boundaries, as the value of correctly synchronized variables is abstracted in a non-relational way. Another possible improvement would be to perform a trace abstraction (Sect. III.E) of sequences of scheduler states to analyze precisely behaviors that depend on the history of interleavings of process execution (e.g., initialization sequences).

VI.C.3. Abstract iterator

Due to the abstraction of states as a collection of maps $(U_i \times C_i) \to \wp(D \times D_i)$, one for each process i, the abstract semantics of each process can almost be computed in isolation. The only coupling between processes is the flow-insensitive abstraction of write actions $\prod_i (U_i \to \wp(D))$. Thus, the analysis is constructed as a least fixpoint computation over an abstract domain of write actions. It starts from an empty set and iterates the analysis of each process in turn (reusing the iterator of Sect. III.K for synchronous programs, which includes inner fixpoint computations) using a widening ∇ to enforce convergence of write actions. This is very memory efficient as, when analysing a process, no information about other processes need to be kept in memory, except for the abstraction of write actions, which has a linear memory cost (as it is non-relational). The cost of the analysis depends directly on the number of fixpoint iterations required to stabilize the write action sets, which is fortunately quite low in practice (Sect. VI.D).

VI.C.4. Operating system modeling

THÉSÉE provides only support for a minimum set of operations on low-level resources: finitely many processes, events, and non-recursive mutexes indexed by `int` identifiers. Applications, however, perform system calls to an ARINC 653 OS that provides higher-level resources (e.g., semaphores, logbooks, sampling and

queuing ports, buffers, blackboards, all identified by string names). Thus, in order to analyze an application, we must complete its source code with a model of the OS, i.e., a set of *stub* C functions that implement all OS entry-points, mapping high-level operations to low-level ones that THÉSÉE understands (e.g., an ARINC 653 buffer is implemented as a C array to store the message contents and a mutex to protect its access). The model also performs error checking and bookkeeping (e.g., maintaining a map between resource names and THÉSÉE identifiers).

The model is not executable as it uses many THÉSÉE-specific intrinsic (e.g, mutex lock). As details of the implementation of parallelism are forgotten, the model embeds some abstraction. The analysis is sound in that it computes (an over-approximation of) all the behaviors of the program when run under an OS respecting the ARINC 653 specification.

VI.D. Preliminary application to aeronautic industry

Our application is a software on a single partition (ARINC 653 can run several such partitions on the same processor but separate time frames and memory space). It consists of 15 processes running in a shared memory and 1.6 M code lines. Intra-partition inter-process communication is precisely modeled by computing an abstraction of the messages sent. However, inter-partition communication as well as non-volatile memory are treated as non-deterministic inputs. This way, we aim at proving that the application has no run-time error, whatever the behavior of other (unanalyzed) applications and contents of the non-volatile memory.

Because the software is very large, we started by analyzing a lighter version reduced to 5 processes and 100 K code lines but similar in complexity (i.e., it corresponds to a coherent functional subset of the total application). Preliminary results indicate an analysis time of 1h 40mn on our 64-bit 2.66GHz Intel server, and approximately 100 alarms. On the full 1.6 M lines code, the analysis takes 50h and outputs a dozen thousands alarms. The high number of alarms is expected as the abstractions have not yet been tuned as much as for the case of synchronous control/command software of Sect. III. In both analyses, the number of iterations required to compute write action sets is quite low (up to 4), which validates our choice of control abstraction for parallel programs[78,79] (Sect. VI.C.2).

VII. Conclusion

There are numerous examples of static analyzers and verifiers which are unsound, imprecise, unscalable, or trivial (i.e., confined to programs with too many restrictions to be of any practical use). It is much more difficult to design sound, precise, and scalable static analyzers with a broad enough scope. We have shown that the theory of abstract interpretation is a good basis for the formal design of such static analyzers and verifiers. We have also provided examples, such as ASTRÉE, now industrialized, built using abstractions designed for specific application domains to eliminate all false alarms. This shows that that program quality control can effectively evolve from control of the design process to the verification of the final product. This opens a new avenue towards a more rigorous programming practice producing verified programs. Of course, much work remains to be done. In particular, considerable research effort is needed on the static analysis prototypes for imperfect synchrony, for parallelism, and for target code certification.

The development of a sound, precise, and scalable static analyzer is a long-term effort. For example, the development of ASTRÉE[80] took 8 years before being available as an industrial product.[38] This does not include the development of flexible user interfaces, the refinements of the analyzer which were necessary to serve a broader community, etc. Moreover the qualification of the tool must go beyond the classic methods based on tests.[61] Work is on-going on formal verification of analyzers, that is, the computer-checked formal proof that the static analyzer is designed according to the abstract interpretation theory so as to be provably sound.

ASTRÉE analyzes control programs in open loop, meaning that the plant is known only by hypotheses on the inputs to the control program (such as e.g., bounds on values returned by sensors). Closing the loop is necessary since a model of the plant must be analyzed together with the control program to prove, e.g., reactivity properties. Obtaining a sound abstract model of the plant is itself a problem to which abstract interpretation can contribute. Considering more expressive properties, such as reactivity, variability, stability, etc., would considerably extend the scope of application of static analysis.

The design of a plant control program follows different stages during which successive models of the plant and its controller are refined until reaching a computer program that can be implemented on hardware. Such

refinements include physical limitations (such as coping with sensor or actuator failures), implementation decisions (such as synchronous or asynchronous implementation on a mono- or multi-processor), etc. Waiting for the final program to discover bugs by static analysis that have not been discovered by simulation of the various models is certainly useful but not completely efficient (in particular when functional bugs are discovered by analysis of the origin of non-functional bugs such as overflows). Development methods can be significantly improved to enhance safety, security, and reduce costs. In particular, static analyses of the different models would certainly speed up the development process and reduce the cost of bugs by earlier detection. Such static analysis tools are missing but can be developed. This raises the question of high-level languages for describing models, which semantics is often vague or unrealistic (e.g., confusing reals and floats) and that of the translation of a specification from one model to another with different semantics. Automatically generating correct, efficient, and structured code would considerably help static analyzers (since dirty code is always more difficult to analyze) and static analyzers can definitely help in code generation (e.g., to choose the translation of real expressions to float so as to minimize rounding errors[81]).

Beyond safety concerns in embedded software, aeronautics is now confronted to security concerns with the generalization of on-board Internet. Here again, security properties can be formalized and verified by abstract interpretation, which is a brand-new and rapidly developing domain of application.

In conclusion, static analysis can go well-beyond classic sequential programming languages and implicit specifications towards the verification of complex computational models and systems. This is certainly nowadays a natural, challenging, and promising orientation for research in abstract interpretation.

References

[1]Cousot, P. and Cousot, R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Conf. Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'77)*, ACM Press (New York), Los Angeles, USA, Jan. 1977, pp. 238–252.

[2]Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "Varieties of Static Analyzers: A Comparison with Astrée," *Proc. of the First Symp. on Theoretical Aspects of Software Engineering (TASE'07)*, edited by M. Hinchey, H. Jifeng, and J. Sanders, IEEE Comp. Soc. Press, Shanghai, China, 6–8 June 2007, pp. 3–17.

[3]Delmas, D. and Souyris, J., "Astrée: from Research to Industry," *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, edited by G. Filé and H. Riis-Nielson, Vol. 4634 of *LNCS*, Springer (Berlin), Kongens Lyngby, Denmark, 22–24 Aug. 2007, pp. 437–451.

[4]Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Goubault, E., Ghorbal, K., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., and Turin, M., "Space Software Validation using Abstract Interpretation," *Proc. of the Int. Space System Engineering Conference, Data Systems In Aerospace (DASIA'09)*, ESA publications, Istanbul, Turkey, 26–29 May 2009, pp. 1–7.

[5]Cousot, P., *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, France, 21 Mar. 1978.

[6]Tarski, A., "A lattice theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, Vol. 5, 1955, pp. 285–310.

[7]Cousot, P. and Cousot, R., "Systematic design of program analysis frameworks," *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, ACM Press (New York), San Antonio, USA, 1979, pp. 269–282.

[8]Cousot, P. and Cousot, R., "Abstract Interpretation Frameworks," *Journal of Logic and Computation*, Vol. 2, No. 4, Aug. 1992, pp. 511–547.

[9]Cousot, P. and Halbwachs, N., "Automatic discovery of linear restraints among variables of a program," *Conf. Rec. of the 5th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'78)*, ACM Press (New York), Tucson, USA, 1978, pp. 84–97.

[10]of Alexandria, E., "Elementa Geometriæ, Book XII, proposition 17," fl. 300 BC.

[11]Burstall, R. M., "Program proving as hand simulation with a little induction," *Information Processing*, 1974, pp. 308–312.

[12]Cousot, P. and Cousot, R., ""À la Burstall" intermittent assertions induction principles for proving inevitability properties of programs," *Theoretical Computer Science*, Vol. 120, 1993, pp. 123–155.

[13]Floyd, R. W., "Assigning meanings to programs," *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, Vol. 19, Providence, USA, 1967, pp. 19–32.

[14]Cousot, P. and Cousot, R., "Sometime = Always + Recursion ≡ Always: on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs," *Acta Informatica*, Vol. 24, 1987, pp. 1–31.

[15]Cousot, P., "Types as Abstract Interpretations," *Conf. Rec. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97)*, ACM Press (New York), Paris, France, Jan. 1997, pp. 316–331.

[16]Besson, F., Cachera, D., Jensen, T., and Pichardie, D., "Certified Static Analysis by Abstract Interpretation," *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, Vol. 5705 of *LNCS*, Springer (Berlin), 2009, pp. 223–257.

[17]Clarke, E., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, Cambridge, 1999.

[18]Cousot, P., "Verification by Abstract Interpretation," *Proc. of the Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, edited by N. Dershowitz, Vol. 2772 of *LNCS*, Springer (Berlin), Taormina, Italy, 29 June–4 July 2003, pp. 243–268.

[19]Cousot, P. and Cousot, R., "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation," *Proc. of the 4th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'92)*, edited by M. Bruynooghe and M. Wirsing, Vol. 631 of *LNCS*, Springer (Berlin), Leuven, Belgium, 26–28 Aug. 1992, pp. 269–295.

[20]Cousot, P. and Cousot, R., "Static determination of dynamic properties of programs," *Proc. of the Second Int. Symp. on Programming (ISOP'76)*, Dunod, Paris, Paris, France, 1976, pp. 106–130.

[21]Bryant, R., "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, Vol. C-35, No. 8, 1986.

[22]Jeannet, B. and Miné, A., "The `Apron` Numerical Abstract Domain Library," `http://apron.cri.ensmp.fr/library/`.

[23]Jeannet, B. and Miné, A., "`Apron`: A library of numerical abstract domains for static analysis," *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, Vol. 5643 of *LNCS*, Springer (Berlin), Grenoble, France, 26 June–2 July 2009, pp. 661–667.

[24]Granger, P., "Static Analysis of Arithmetical Congruences," *Int. J. Comput. Math.*, Vol. 30, No. 3 & 4, 1989, pp. 165–190.

[25]Miné, A., "The Octagon Abstract Domain," *Higher-Order and Symbolic Computation*, Vol. 19, 2006, pp. 31–100.

[26]Feret, J., "Static Analysis of Digital Filters," *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, edited by D. Schmidt, Vol. 2986 of *LNCS*, Springer (Berlin), 27 Mar.–4 Apr. 2004, pp. 33–48.

[27]Feret, J., "The Arithmetic-Geometric Progression Abstract Domain," *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, edited by R. Cousot, Vol. 3385 of *LNCS*, Springer (Berlin), Paris, France, 17–19 Jan. 2005, pp. 42–58.

[28]Granger, P., "Static Analysis of Linear Congruence Equalities among Variables of a Program," *Proc. of the Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91), Volume 1 (CAAP'91)*, edited by S. Abramsky and T. Maibaum, Vol. 493 of *LNCS*, Springer (Berlin), Brighton, UK, 1991, pp. 169–192.

[29]Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "Combination of Abstractions in the Astrée Static Analyzer," *Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, edited by M. Okada and I. Satoh, Vol. 4435 of *LNCS*, Springer (Berlin), Tokyo, Japan, 6–8 Dec. 2006, pp. 272–300.

[30]Cousot, P., "Semantic Foundations of Program Analysis, invited chapter," *Program Flow Analysis: Theory and Applications*, edited by S. Muchnick and N. Jones, chap. 10, Prentice-Hall, Inc., Englewood Cliffs, 1981, pp. 303–342.

[31]Bourdoncle, F., "Abstract Interpretation by Dynamic Partitioning," *Journal of Functional Programming*, Vol. 2, No. 4, 1992, pp. 407–423.

[32]Giacobazzi, R., Ranzato, F., and Scozzari, F., "Making Abstract Interpretations Complete," *Journal of the Association for Computing Machinery*, Vol. 47, No. 2, 2000, pp. 361–416.

[33]Cousot, P., Ganty, P., and Raskin, J.-F., "Fixpoint-Guided Abstraction Refinements," *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, edited by G. Filé and H. Riis-Nielson, Vol. 4634 of *LNCS*, Springer (Berlin), Kongens Lyngby, Denmark, 22–24 Aug. 2007, pp. 333–348.

[34]Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter," *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, edited by T. Mogensen, D. Schmidt, and I. Sudborough, Vol. 2566 of *LNCS*, Springer (Berlin), 2002, pp. 85–108.

[35]Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "A Static Analyzer for Large Safety-Critical Software," *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI'03)*, ACM Press (New York), San Diego, USA, 7–14 June 2003, pp. 196–207.

[36]Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "The Astrée analyser," *Proc. of the 14th European Symposium on Programming Languages and Systems (ESOP'05)*, edited by M. Sagiv, Vol. 3444 of *LNCS*, Springer (Berlin), 2–10 Apr. 2005, pp. 21–30.

[37]Mauborgne, L., "Astrée: Verification of Absence of Run-time error," *Building the Information Society*, edited by P. Jacquart, chap. 4, Kluwer Academic Publishers, Dordrecht, 2004, pp. 385–392.

[38]AbsInt, Angewandte Informatik, "Astrée Run-Time Error Analyzer," `http://www.absint.com/astree/`.

[39]Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X., "Astrée: Proving the Absence of Rutime Errors," *Proc. of Embedded Real-Time Software and Systems (ERTS'10)*, Toulouse, France, May 2010, pp. 1–5, (to appear).

[40]ISO/IEC JTC1/SC22/WG14 Working Group, "C standard," Tech. Rep. 1124, ISO & IEC, 2007.

[41]Mauborgne, L. and Rival, X., "Trace Partitioning in Abstract Interpretation Based Static Analyzer," *Proc. of the 14th European Symp. on Programming Languages and Systems (ESOP'05)*, edited by M. Sagiv, Vol. 3444 of *LNCS*, Springer (Berlin), Edinburg, UK, 4–8 Apr. 2005, pp. 5–20.

[42]Rival, X. and Mauborgne, L., "The trace partitioning abstract domain," *ACM Trans. Program. Lang. Syst.*, Vol. 29, No. 5, 2007.

[43]Miné, A., "Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics," *Proc. of the ACM SIGPLAN-SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, ACM Press (New York), June 2006, pp. 54–63.

[44]Moore, R. E., *Interval Analysis*, Prentice Hall, Englewood Cliffs N. J., USA, 1966.

[45]IEEE Computer Society, "IEEE Standard for Binary Floating-Point Arithmetic," Tech. rep., ANSI/IEEE Std. 745-1985, 1985.

[46]Miné, A., "Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors," *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, edited by D. Schmidt, Vol. 2986 of *LNCS*, Springer (Berlin), 27 Mar.27–4 Apr. 2004, pp. 3–17.

[47]Miné, A., *Weakly Relational Numerical Abstract Domains*, Thèse de doctorat en informatique, École polytechnique, Palaiseau, France, 6 Dec. 2004.

[48]Goubault, E., "Static analyses of floating-point operations," *Proc. of the 8th Int. Static Analysis Symposium (SAS'01)*, Vol. 2126 of *LNCS*, Springer (Berlin), 2001, pp. 234–259.

[49]Miné, A., "The Octagon Abstract Domain," *Proc. of the Analysis, Slicing and Transformation Workshop (AST'01)*, IEEE Computer Society Press (Los Alamitos), Oct. 2001, pp. 310–319.

[50]Larsen, K., Larsson, F., Pettersson, P., and Yi, W., "Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction," *Proc. of the 18th IEEE Real-Time Systems Symp. (RTSS'97)*, IEEE CS Press, 1997, pp. 14–24.

[51]Bryant, R. E., "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Vol. 35, 1986, pp. 677–691.

[52]Feret, J., "Numerical Abstract Domains for Digital Filters," *Proc. of the First Int. Workshop on Numerical & Symbolic Abstract Domains (NSAD'05)*, Maison Des Polytechniciens, Paris, France, 21 Jan. 2005.

[53]Monniaux, D., "The parallel implementation of the Astrée static analyzer," *Proc. of the 3rd Asian Symp. on Programming Languages and Systems (APLAS'05)*, Vol. 3780 of *LNCS*, Springer (Berlin), 2005, pp. 86–96.

[54]Bourdoncle, F., "Efficient Chaotic Iteration Strategies with Widenings," *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, Vol. 735 of *LNCS*, Springer (Berlin), 1993, pp. 128–14.

[55]Esterel Technologies, "SCADE Suite™, The Standard for the Development of Safety-Critical Embedded Software in the Avionics Industry," `http://www.esterel-technologies.com/`.

[56]Heckmann, R. and Ferdinand, C., "Worst-Case Execution Time Prediction by Static Program Analysis," *Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, IEEE Computer Society, 2004, pp. 26–30.

[57]Ginosar, R., "Fourteen Ways to Fool Your Synchronizer," *Proc. of the 9th International Symposium on Asynchronous Circuits and Systems (ASYNC'03)*, 2003.

[58]Caspi, P., Mazuet, C., and Paligot, N. R., "About the Design of Distributed Control Systems: The Quasi-Synchronous Approach," *20th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2001)*, edited by U. Voges, Vol. 2187 of *LNCS*, Springer (Berlin), Budapest, Hungary, Sept. 2001, pp. 215–226.

[59]Bertrane, J., "Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs," *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, edited by R. Cousot, Vol. 3385 of *LNCS*, Springer (Berlin), Paris, France, 17–19 Jan. 2005, pp. 97–112.

[60]Bertrane, J., "Proving the Properties of Communicating Imperfectly-Clocked Synchronous Systems," *Proc. of the 13th Int. Static Analysis Symposium (SAS'06)*, edited by K. Yi, Vol. 4134 of *LNCS*, Springer (Berlin), Seoul, Korea, 29–31 Aug. 2006, pp. 370–386.

[61]Technical Commission on Aviation, R., "DO-178B," Tech. rep., Software Considerations in Airborne Systems and Equipment Certification, 1999.

[62]Cousot, P. and Cousot, R., "Systematic Design of Program Transformation Frameworks by Abstract Interpretation," *Conf. Rec. of the 29th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'02)*, ACM Press (New York), Portland, USA, Jan. 2002, pp. 178–190.

[63]Rival, X., "Symbolic transfer functions-based approaches to certified compilation," *Conf. Rec. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'04)*, ACM Press (New York), Venice, Italy, Jan. 2004, pp. 1–13.

[64]Necula, G. C., "Proof-Carrying Code," *Conf. Rec. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Langauges (POPL'97)*, ACM Press (New York), Paris, France, Jan. 1997, pp. 106–119.

[65]Rival, X., "Abstract Interpretation-based Certification of Assembly Code," *Proc. of the 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, Vol. 2575 of *LNCS*, Springer (Berlin), New York, USA, Jan. 2003, pp. 41–55.

[66]Pnueli, A., Shtrichman, O., and Siegel, M., "Translation Validation for Synchronous Languages," *Proc. of the 25th Int. Coll. on Automata, Languages and Programming (ICALP'98)*, Vol. 1443 of *LNCS*, Springer (Berlin), Aalborg, Denmark, Jul. 1998, pp. 235–246.

[67]Necula, G. C., "Translation Validation for an Optimizing Compiler," *Proc. of the Conf. on Programming Language Design and Implementation (PLDI'00)*, ACM Press (New York), Vancouver, Canada, June 2000, pp. 83–94.

[68]Leroy, X., "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," *Conf. Rec. of the 33rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'06)*, ACM Press (New York), Charleston, USA, 2006, pp. 42–54.

[69]IEEE and The Open Group, "Portable Operating System Interface (POSIX)," `http://www.opengroup.org`, `http://standards.ieee.org`.

[70]Aeronautical Radio, Inc. (ARINC), "ARINC 653," `http://www.arinc.com/`.

[71]Saraswat, V., Jagadeesan, R., Michael, M., and von Praun, C., "A theory of memory models," *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'07)*, ACM Press (New York), San Jose, USA, Mar. 2007, pp. 161–172.

[72]Lamport, L., "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, Vol. 28, Sept. 1979, pp. 690–691.

[73]Gosling, J., Joy, B., Steele, G., and Bracha, G., *The Java language specification, third edition*, Addison Wesley, 2005.

[74]ISO/IEC JTC1/SC22/WG21 Working Group, "Working draft, standard for programming language C++," Tech. Rep. 3035, ISO & IEC, 2010.

[75]Manson, J., Pugh, W., and Adve, S. V., "The Java memory model," *Conf. Rec. of the 32nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'05)*, ACM Press (New York), Long Beach, USA, 2005, pp. 378–391.

[76]Ferrara, P., "Static analysis via abstract interpretation of the happens-before memory model," *Proc. of the Second Int. Conf. on Tests and Proofs (TAP'08)*, Vol. 4966 of *LNCS*, Springer (Berlin), 2008, pp. 116–133.

[77]Ferrara, P., *Static analysis via abstract interpretation of multithreaded programs*, Ph.D. thesis, École Polytechnique, Palaiseau, France, May 2009.

[78]Cousot, P. and Cousot, R., *Invariance proof methods and analysis techniques for parallel programs*, chap. 12, Macmillan, New York, USA, 1984, pp. 243–271.

[79]Owicki, S. and Gries, D., "An axiomatic proof technique for parallel programs I," *Acta Informatica*, Vol. 6, No. 4, Dec. 1976, pp. 319–340.

[80]Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X., "The Astrée static analyzer," `www.astree.ens.fr` and `www.absint.com/astree/`.

[81]Martel, M., "Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics," *Formal Methods in System Design*, Vol. 35, No. 3, Dec. 2009, pp. 265–278.