Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival

Foundations and Trends® in Programming Languages
Vol. 2, No. 2-3 (2015) 171–291
© 2015 J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival
DOI: 10.1561/2500000002

Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane
Département d'informatique, École normale supérieure
Julien.Bertrane@ens.fr

Patrick Cousot
Département d'informatique, École normale supérieure
Courant Institute of Mathematical Sciences, New York University
pcousot@cs.nyu.edu

Radhia Cousot
CNRS & Département d'informatique, École normale supérieure
Radhia.Cousot@ens.fr

Jérôme Feret
INRIA & Département d'informatique, École normale supérieure
Jerome.Feret@ens.fr

Laurent Mauborgne
AbsInt Angewandte Informatik
laurent.mauborgne@absint.com

Antoine Miné
Sorbonne University, University Pierre and Marie Curie, CNRS, LIP6
Antoine.Mine@lip6.fr

Xavier Rival
INRIA & Département d'informatique, École normale supérieure
Xavier.Rival@ens.fr

Contents

Abstract

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 programs, parallel programs, and target code validation as well. Future research directions on abstract interpretation are also discussed in the context of aerospace software.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. *Static Analysis and Verification of Aerospace Software by Abstract Interpretation*. Foundations and Trends$^{®}$ in Programming Languages, vol. 2, no. 2-3, pp. 171–291, 2015.

DOI: 10.1561/2500000002.

Nomenclature

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

1 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 [Cousot and Cousot, 1977] 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 floating-point operation. Another example is the range of inputs, like voltages, that can be overestimated by the full range of the hardware register where the value is sampled (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 registers).

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, floating-point overflows, data-races, deadlocks, live-locks, etc. are all frequent symptoms of software bugs, the absence of which 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 make fully automatic proofs on all programs ultimately impossible) and the desire not to rely on end-user interactive help (which can add a heavy, or even intractable cost), 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 raised which may be false. Intensive research work is done to discover appropriate abstractions eliminating this uncertainty about false alarms for domain-specific applications.

In this article, we report on the successful scalable and cost-effective application of abstract interpretation to the verification of the absence of runtime errors in aerospace control software by the Astrée static analyzer [Cousot et al., 2007a], illustrated first by the verification of the fly-by-wire primary software of commercial airplanes [Delmas and Souyris, 2007] and then by the validation of the Monitoring and Safing Unit (MSU) of the Jules Vernes ATV docking software [Bouissou et al., 2009]. We also 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.

An early version of this article appeared in [Bertrane et al., 2010]. We extend that version with more thorough explanations, additional examples, and updated experimental results.

2

Theoretical Background on Abstract Interpretation

The *static analysis* of a program consists in automatically determining properties of its executions at compile time, by examination of its source code without actually executing it. A *sound* static analysis discovers properties that are true of *all* its possible executions, in *all* possible execution environments (possibly constrained by user-provided knowledge, such as the range of inputs). Hence, sound static analysis differs from testing, symbolic execution, and hybrid methods (such as concolic execution) which cannot guarantee that all the behaviors of every program are covered. Sound static analysis provides formal guarantees and can be used in program verification activities requiring a high level of confidence, such as program certification of critical embedded software.

Static analysis is based on a formal *semantics*, which is a mathematical model of the executions of the program. Many such models exist, with varying levels of details and usefulness for a given problem. A key component of the abstract interpretation theory is the study of the relationships between different semantics, and the derivation of new semantics in a systematic way. One particular semantics, the *collecting semantics*, is a mathematical model of the strongest property of interest. For instance, if we are interested in proving that a program never reaches an error *state*, the collecting semantics consists of the set of states reachable during the execution while, if we are interested in temporal properties, the collecting semantics consists instead in the set of execution *traces* (a trace being a sequence of states). By setting the flavor of properties we can reason about, the collecting semantics plays a similar role as does the choice of a logic in other formal methods (such as first-order logic in deductive methods, or temporal logic in model-checking), but in semantics, set theoretical form rather than syntactic, logical form.

Program *verification* consists in proving that the program collecting semantics implies a given *specification*, which can be given explicitly by the user, or provided implicitly by a language specification (for instance, the absence of run-time error). We are interested in designing sound automatic static analyzers able to perform such verification. By "automatic," we mean without any human assistance during the analysis and the verification processes. This is in contrast to deductive methods which employ theorem provers that, despite some automation, ultimately rely on human assistance. Note that, as verification problems are undecidable, any static analyzer or verifier is either unsound (its conclusion may be wrong), incomplete (it is inconclusive), may not terminate, or all of the above, and this on infinitely many programs. The static analyzers we consider are sound on all programs and always terminate on all programs, but are incomplete.

In a nutshell, abstract interpretation is a theory of approximation of mathematical structures. It can thus formalize the fact that reachability semantics are less precise (more abstract) than trace-base semantics. It can also be applied to the systematic design of static analyzers and verifiers by abstraction of the collecting semantics into efficiently computable, approximate ones. For instance, one may only focus on the type, the sign or the range of the variables instead of their precise value. After the intellectual process (not automated, but guided by the theory) of choosing an abstraction and deriving the abstract semantics of the target programming language from the collecting semantics, an analyzer able to compute automatically the abstract semantics of arbitrary programs in that language is effectively implemented. The derived static analyzer is always sound and always terminates, and so is necessarily incomplete: it either returns in finite time “yes,” meaning that the program indeed satisfies its specification, or returns in finite time “don’t know,” meaning that the program may or may not satisfy its specification. As a result of incompleteness, abstract interpretation-based static analyzers will fail and return “don’t known” on infinitely many correct programs. Fortunately, they will also succeed in proving the correctness of infinitely many correct programs, while never considering any incorrect program as correct.

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.

The rest of this chapter presents some basic concepts of abstract interpretation as well as the formal principles underlying the design of sound static analysis, in a very general way, not tied to a specific program, language or application. The following chapters will then present, in a less formal way, applications, tools, and results on specific verification problems.

2.1 Semantics

Following Cousot [1978], we model programs using a *small-step operational semantics*. More precisely, a program is a *transition systems* (S, t, I) defined by:

- a set S of program states;
- a transition relation $t \subseteq S \times S$;
- a subset of initial states $I \subseteq S$.

A program state $s \in S$ represents a snapshot of the whole memory, including the value of variables, stacks, program counters, registers, etc. Program execution is discrete, and proceeds in steps. A transition $\langle s, s' \rangle \in t$ models an atomic step of execution, such as the effect of a single instruction, that allows transitioning from one state s to the next one s'. Additionally, we consider a subset $I \subseteq S$ of states denoting the initial program states, i.e., the possible program states in which the execution can begin. The transition relation is generally non-deterministic, i.e., some states $s \in S$ may have many possible successors. This is particularly useful to model inputs, that are out of the control of the program (such as reading a file or a sensor). Note that the transition systems corresponding to actual programs are very large or even, in certain cases, infinite (such as for parameterized programs, or for programs with unbounded memory). Transition systems are intended as a mathematical model, not as an actual data-structure to be manipulated directly by an analyzer.

Example 2.1. A very simple, finite example of transition system can be defined as $S \triangleq \{a, b, c\}$, $I \triangleq \{a\}$, and $t \triangleq \{\langle a, b\rangle, \langle b, b\rangle, \langle b, c\rangle\}$: starting in state a, we can transition from a to b, and then from b to either b or c. For the sake of simplicity, we will use this transition system to illustrate our definitions in the rest of the chapter. $\square$

Given a transition system, we can define an execution of the program as a sequence of states $\langle s_0\ s_1 \dots s_n\rangle$. An execution starts in an initial state $s_0 \in I$. It continues with a successor state s_1, 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 goes on until it reaches a state s_n which is final, i.e., that has no possible successor: $\forall s' \in S : \langle s_n, s'\rangle \notin t$. Alternatively, the execution can go on forever, which we denote as $\langle s_0\ s_1 \dots\rangle$. The latter case corresponds to non-termination, while the former case corresponds to either correct or erroneous program termination. Due to the non-determinism in the transition relation and the choice of an initial state, there may be many such executions. We call the set of execution traces spawned by a transition system with initial states I and transition relation t its *maximal trace semantics*, and denote it as $\mathcal{T}[\![t]\!]I$. It is defined formally as: $\mathcal{T}[\![t]\!]I \triangleq \{\langle s_0 \dots s_n\rangle \mid n \geqslant 0 \wedge s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t \wedge \forall s' \in S : \langle s_n, s'\rangle \notin t\} \cup \{\langle s_0 \dots\rangle \mid s_0 \in I \wedge \forall i \geq 0 : \langle s_i, s_{i+1}\rangle \in t\}$. 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.

Example 2.2. Our finite transition system example features a single final state c. The maximal trace semantics is $\mathcal{T}[\![t]\!]I = \{\langle ab \dots c\rangle, \langle ab \dots\rangle\}$. It is composed of the finite traces starting with a followed by one or more b and ending with a c, as well as the infinite trace staring with a followed by an infinite sequence of b. □

2.2 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, because it can contain infinite executions. However, we can observe program executions for a finite time. Looking at all the finite parts of all executions (including the finite parts of infinite executions) is actually sufficient to reason about *safety* properties, which informally state that "nothing bad happens." However, it prevents us from reasoning about program termination and, more generally, about *liveness* properties (informally stating that "something good happens").

Finite observations of program executions $\mathcal{P}[\![t]\!]I$ can be formally defined as $\mathcal{P}[\![t]\!]I \triangleq \{\langle s_0 \dots s_n\rangle \mid n \geqslant 0 \wedge s_0 \in I \wedge \forall i \in [0, n) : \langle s_i, s_{i+1}\rangle \in t\} \subseteq S^*$, where S^* is the set of finite sequences of states from S. $\mathcal{P}[\![t]\!]$I is called the finite *prefix trace semantics*. This semantics is simpler than the maximal trace semantics, because it is only composed of finite traces. Nevertheless, it is sufficient to answer any safety question about program behaviors, i.e., properties the failure of which is checkable by monitoring the program. 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.

Example 2.3. In our finite transition system example, the prefix trace semantics is $\mathcal{P}[\![t]\!]I = \{\langle a\rangle, \langle ab \dots b\rangle, \langle ab \dots c\rangle\}$. It contains the one-element trace a, as well as traces starting in a followed by a finite sequence of b and ending in b, and traces starting in a followed by a finite (non-empty) sequence of b and ending in c. An example of safety property would be that c cannot occur unless b has occurred before during the execution. The property can be checked by considering only $\mathcal{P}[\![t]\!]I$, and it is not necessary to consider $\mathcal{T}[\![t]\!]I$. However, observing $\mathcal{P}[\![t]\!]I$ only, it is impossible to deduce whether the program always terminates or not, as any finite trace in $\mathcal{P}[\![t]\!]I$ may actually be the beginning of a longer (possibly infinite) trace. $\square$

Ideally, computing this collecting semantics would answer all safety questions. Unfortunately, this is not possible: although each trace is now finite and can, individually, be checked automatically, the prefix trace semantics may contain an infinite (or impracticably large) number of such finite traces. Hence, we will use further 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. It depends on the level of observation of program behaviors required to solve the problem. We illustrate this point with another example of collecting semantics. When only interested in *invariance* properties, a possible choice for the collecting semantics would be the *reachability semantics* $\mathcal{R}[\![t]\!]I$, which collects the set of states that can be reached during any (finite or infinite) program execution. The reachability semantics is defined formally as: $\mathcal{R}[\![t]\!]I \triangleq \{s' \mid \exists s \in I : \langle s, s' \rangle \in t^\star\} \subseteq S$, where the *reflexive transitive closure* $t^\star \subseteq S \times S$ of a relation $t \subseteq S \times S$ is defined classically as $t^\star \triangleq \{\langle s, s' \rangle \mid \exists n \geqslant 0 : \exists s_0 \dots s_n : s_0 = s \land \forall i \in [0, n) : \langle s_i, s_{i+1} \rangle \in t \land s_n = s'\}$. Alternatively, $t^\star$ can be defined as $t^\star = \bigcup_{n \geq 0} t^n$ where, for all $n \geqslant 0$, the $n-$th iterate of t is defined as $t^n \triangleq \{\langle s, s' \rangle \mid \exists s_0 \dots s_n : s_0 = s \land \forall i \in [0, n) : \langle s_i, s_{i+1} \rangle \in t \land s_n = s'\} \subseteq S \times S$.

As an invariance property is defined as a set of states in which all program executions must stay, computing the reachability semantics is indeed sufficient to check the property. It reduces to checking that $\mathcal{R}[\![t]\!]I$ is included in the state set of the property. The reachability semantics is more abstract than the prefix trace semantics since, although 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 "no" 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.

Example 2.4. For our finite transition system example, the reachability semantics is $\mathcal{R}[\![t]\!]I = \{a, b, c\}$, i.e., all the states can be reached. An example of invariance property would be that the execution stays in the state set $\{a, b, c\}$, which can be checked obviously by only looking at $\mathcal{R}[\![t]\!]I$. $\square$

2.3 Fixpoint semantics

A major observation underlying the abstract interpretation framework is that program semantics can be expressed as *fixpoints* of functions over ordered mathematical structures [Cousot and Cousot, 1977].

As a simple example of fixpoint, consider the computation of the transitive closure $t^\star \subseteq S \times S$ of a relation $t \subseteq S \times S$, used in the definition of the reachability semantics $\mathcal{R}[\![t]\!]I$ in the previous section. By defining the identity relation 1_S on S as $1_S \triangleq \{\langle s, s\rangle \mid s \in S\}$, and the composition $\circ$ of relations as $t \circ r \triangleq \{\langle s, s''\rangle \mid \exists s' : \langle s, s'\rangle \in t \wedge \langle s', s''\rangle \in r\}$, we can state that t^* is the smallest relation X satisfying the equation $X = 1_S \cup (X \circ t)$. Indeed, t^* must contain the identity relation and be closed by an application of t. Moreover, as a relation is a subset of $S \times S$, the notion of "smallest" corresponds to the notion of set inclusion $\subseteq$: we reason in the partially ordered set $(\mathcal{P}(S \times S), \subseteq)$. Note that t^* is also the smallest relation satisfying the alternate equation $X = 1_S \cup (t \circ X)$.

Formally, a fixpoint of a function F is any element X satisfying $F(X) = X$. We denote by $\mathrm{lfp}^{\subseteq} F$ the least fixpoint of F with respect to $\subseteq$, i.e., not only $F(\mathrm{lfp}^{\subseteq} F) = \mathrm{lfp}^{\subseteq} F$, but also, for any Y such that $F(Y) = Y$, then $\mathrm{lfp}^{\subseteq} F \subseteq Y$. Being smaller than any other fixpoint, the least fixpoint is unique, if it exists. Based on the equations satisfied by the transitive closure $t^\star$, we have that $t^\star = \mathrm{lfp}^{\subseteq} F$ where $F(X) \triangleq 1_S \cup (X \circ t)$. Moreover, $t^\star = \mathrm{lfp}^{\subseteq} B$ where $B(X) \triangleq 1_S \cup (t \circ X)$. The mathematical theory of partial orders features a rich collection of fixpoint theorems. In our example, the existence of $\mathrm{lfp}^{\subseteq} F$ and $\mathrm{lfp}^{\subseteq} B$ follows from Tarski's theorem [Tarski, 1955]. The two hypotheses of the theorem are satisfied: firstly, the set of relations ordered by set inclusion forms a complete lattice $(\wp(S^*), \subseteq, \cup, \cap, \emptyset, S^*)$, i.e., a mathematical structure where every collection of elements has a least upper bound $\cup$ and a greatest lower bound $\cap$; secondly, the functions F and B are monotonic in this complete lattice, i.e., $X \subseteq Y$ implies $F(X) \subseteq F(Y)$ and $B(X) \subseteq B(Y)$.

Although it states the existence of fixpoints, Tarksi's theorem does not provide any guideline on how to compute them effectively. Thankfully, other results [Cousot and Cousot, 1979a] state that least fixpoints can be computed as limits of iterations. In our example, the iteration starts from $\emptyset$ and iterates $F^{i+1}(\emptyset) = F(F^i(\emptyset))$ by applying F repeatedly. We get, after one iteration: $X^0 \triangleq 1_S \cup (\emptyset \circ t) = 1_S = t^0$. The second iteration gives: $X^1 \triangleq 1_S \cup (X^0 \circ t) = 1_S \cup t$. We then get: $X^2 \triangleq 1_S \cup (X^1 \circ t) = 1_S \cup t \cup t^2$, etc. We see a pattern emerging, and we can prove by recurrence that the iterates are $\forall n \geqslant 0 : X^n = \bigcup_{i=0}^n t^i$. Indeed, if, by recurrence hypothesis, $X^n = \bigcup_{i=0}^n t^i$, we get $X^{n+1} \triangleq 1_S \cup (X^n \circ t) = 1_S \cup ((\bigcup_{i=0}^n t^i) \circ t) = 1_S \cup (\bigcup_{i=0}^n t^{i+1}) = \bigcup_{i=0}^{n+1} t^i$. Passing to the limit in the complete lattice of relations (i.e., joining the infinite sequence of iterates with $\cup$), we get: $X^\star \triangleq \bigcup_{n \geq 0} X^n = \bigcup_{n \geq 0} \bigcup_{i=0}^n t^i = \bigcup_{n \geq 0} t^n$, which is precisely $t^\star$.

Recall that $t^\star = \mathrm{lfp}^\subseteq F = \mathrm{lfp}^\subseteq B$. In fact, $\mathrm{lfp}^\subseteq F$ and $\mathrm{lfp}^\subseteq B$ provide alternate ways to compute $t^\star$ by iteration. More precisely, $F(X)$ operates *forwards* as each application of F extends traces in X with new transitions at the end, while $B(X)$ operates *backwards* as each application of B prepends traces in X with transitions at the beginning. After abstraction, they lead to alternate (and often complementary), forwards and backwards static analysis techniques.

Based on these results, we can now see that the reachability semantics defined in the previous section can indeed be expressed in fixpoint form, as $\mathcal{R}[\![t]\!]I = \mathrm{lfp}^\subseteq F_R$ where $F_R(X) \triangleq I \cup \{s' \mid \exists s \in X : \langle s, s' \rangle \in t\}$. Likewise, the prefix trace semantics can be expressed as $\mathcal{P}[\![t]\!]I = \mathrm{lfp}^\subseteq F_P$ where $F_P(X) \triangleq \{\langle s_0 \rangle \mid s_0 \in I\} \cup \{\langle s_0 \dots s_n s_{n+1} \rangle \mid \langle s_0 \dots s_n \rangle \in X \wedge \langle s_n, s_{n+1} \rangle \in t\}$. The maximal trace semantics $\mathcal{T}[\![t]\!]I$ can also be expressed in fixpoint form, although the presence of infinite traces complicates the formulation [Cousot, 2002].

Example 2.5. To compute the reachability semantics of our finite transition system example, we iterate F_R from $\emptyset$ and get: $X^0 = F_R(\emptyset) = I = \{a\}$, then $X^1 = F_R(X^0) = \{a, b\}$, and finally $X^2 = F_R(X^1) = \{a, b, c\}$. We then observe that $X^3 = F_R(X^2) = X^2$ and, as a consequence, all the iterates X^n for $n \geq 2$ are equal. We have reached our fixpoint, which is indeed $\mathcal{R}[\![t]\!]I = \{a, b, c\}$. The iterates of F_P involved in the computation of the prefix trace semantics $\mathcal{P}[\![t]\!]I$ are more complex. At the $n-$th iterate, we obtain the set X^n of execution traces of length at most $n+1$: $X^n = \{\langle ab^k\rangle, \langle ab^{k+1}c\rangle \mid 0 \leq k \leq n\}$. The sequence X^n does not converge in finite time, but we can check that the limit $\bigcup_{i\geq 0} X^n$ indeed equals $\mathcal{P}[\![t]\!]I$. $\square$

2.4 Abstraction functions

We have seen several semantics and stated informally that some of them provide more information than others. This relation between semantics can be formalized by the notion of *abstraction*. An abstraction [Cousot and Cousot, 1977] is a mathematical relationship between two semantics, a so-called *concrete* (more precise) semantics and a so-called *abstract* (less precise) semantics. The key property of abstractions is *soundness*, which states that any program property proved to hold in the abstract semantics also holds in the concrete one. Generally, however, not all properties provable in the concrete semantics (such as the correctness of a specific program) can be proved in the abstract semantics: this is *incompleteness*, which causes false alarms. Another property of the abstract semantics, which is important when considering effective static analyzers, is termination; it is discussed in §2.14.

An example abstraction is the correspondence between the (concrete) prefix trace semantics and the (abstract) reachability semantics. It is defined as the *reachability abstraction function* $\alpha_R : \wp(S^*) \rightarrow \wp(S)$, such that $\alpha_R(X) \triangleq \{s \mid \exists \langle s_0 \ldots s_n\rangle \in X : s = s_n\}$: given a set of finite sequences in $\wp(S^*)$ as input, it outputs the set of states in $\wp(S)$ that can appear at the last position of the sequence. We can then check that $\mathcal{R}[\![t]\!]I = \alpha_R(\mathcal{P}[\![t]\!]I)$. Indeed, the prefix trace semantics collects all the finite observations of a program execution and, by only remembering the last state of every such finite observation, it collects the whole set of states reached along all (finite and infinite) executions. The abstraction records the reachable states but no longer the order in which these states appear during execution. Note that the abstraction function is monotonic, i.e., we have that $X \subseteq Y$ implies $\alpha_R(X) \subseteq \alpha_R(Y)$: programs with larger sets of prefix execution traces necessarily have larger sets of reachable states.

Another example of abstraction is the correspondence between the (concrete) maximal trace semantics and the (abstract) prefix trace semantics. We define the *prefix abstraction function* $\alpha_P(X) \triangleq \{\langle s_0 \ldots s_n \rangle \mid \exists m \geqslant 0 : \exists \langle s_{n+1} \ldots s_{n+m} \rangle : \langle s_0 \ldots s_{n+m} \rangle \in X \vee \exists \langle s_{n+1} \ldots \rangle : \langle s_0 \ldots s_n s_{n+1} \ldots \rangle \in X\}$ which, given a set of finite or infinite sequences of states, returns the set of all the finite prefixes of these sequences. We can then check that $\mathcal{P}[\![t]\!]I = \alpha_P(\mathcal{T}[\![t]\!]I)$.

Abstractions can be composed, which is useful in practice to design complex abstractions from simpler ones. In our example, the reachability semantics can be obtained directly from the maximal trace semantics by applying $\alpha_R \circ \alpha_P$, which is the composition of the reachability and prefix abstraction functions, and is an abstraction function in its own right. We have indeed $\mathcal{R}[\![t]\!]I = (\alpha_R \circ \alpha_P)(\mathcal{T}[\![t]\!]I)$.

2.5 Concretization functions

An abstraction function converts a semantic information from a more concrete semantic world to a more abstract semantic world. We can also consider a reverse function, called *concretization function*, to convert from an abstract world back to a concrete one.

For instance, the counterpart of the reachability abstraction function α_R is the concretization function $\gamma_R : \wp(S) \to \wp(S^*)$ defined as $\gamma_R(X) \triangleq \{\langle s_0 \ldots s_n \rangle \in X \mid \forall i \in [0, n] : s_i \in X\}$ and that, given a set of states in $\wp(S)$ outputs a set of finite sequences of states in $\wp(S^*)$. The concretization rebuilds the prefix 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, $\mathcal{R}[\![t]\!]I = \alpha_R(\mathcal{P}[\![t]\!]I)$, is an over-approximation of the concrete, $\mathcal{P}[\![t]\!]I$, 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. Concretization functions are also monotonic: they map more approximate abstract semantics (such as larger sets of reachable states) to more approximate concrete semantics (such as larger sets of prefix execution traces).

Example 2.6. Recall that the reachability semantics of our finite transition system is $\mathcal{R}[\![t]\!]I = \{a, b, c\}$, while its prefix trace semantics is $\mathcal{P}[\![t]\!]I = \{\langle a\rangle, \langle ab\ldots b\rangle, \langle ab\ldots c\rangle\}$. We can check that $\alpha_R(\mathcal{P}[\![t]\!]I) = \mathcal{R}[\![t]\!]I$. The concretization of the reachability semantics is however $\gamma_R(\mathcal{R}[\![t]\!]I) = S^*$, i.e., it contains the finite traces of arbitrary lengths composed of elements from $\{a, b, c\}$. Naturally, $\mathcal{P}[\![t]\!]I \subseteq \gamma_R(\mathcal{R}[\![t]\!]I)$, but the equality does not hold. Consider now the problem of proving that no c can occur in an execution without a b occurring before. The property is true in $\mathcal{P}[\![t]\!]I$, but not in $\gamma_R(\mathcal{R}[\![t]\!]I)$ as the latter contains the trace $\langle c\rangle$. Hence, the property can be proved using the prefix trace semantics, while the proof using the reachability semantics is inconclusive. $\square$

2.6 Galois connections

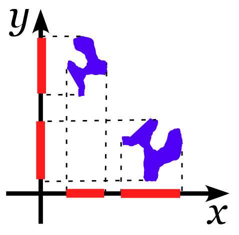
Intuitively, the abstraction and the concretization functions are "inverse" of one another. This intuition is formalized by the notion of *Galois connection* $\langle\alpha, \gamma\rangle$ [Cousot and Cousot, 1979b]. Formally, a Galois connection $\langle\alpha, \gamma\rangle$ is a pair of functions between two ordered sets: a concrete set $(C, \subseteq)$ equipped with some partial order $\subseteq$, and an abstract set $(A, \subseteq^\sharp)$, equipped with another partial order $\subseteq^\sharp$, such that:

1. $\alpha : C \to A$ converts a concrete element into an abstract one;
2. $\gamma : A \to C$ converts an abstract element into a concrete one;

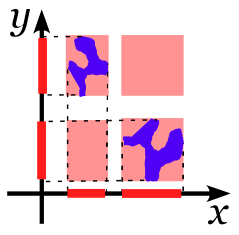
3. $\forall T \in C, R \in A : \alpha(T) \subseteq^\sharp R$ if and only if $T \subseteq \gamma(R)$, which is the core property of Galois connections.

The pair $\langle \alpha_R, \gamma_R \rangle$ is an example of Galois connection, linking the prefix trace (concrete) semantics to the reachability (abstract) semantics. The concrete (respectively, abstract) order $\subseteq$ (respectively, $\subseteq^\sharp$) denotes a notion of over-approximation on C (respectively, A). For instance, both the prefix trace semantics, in $(\wp(S^*), \subseteq)$, and the reachability semantics, in $(\wp(S), \subseteq)$, use set inclusion $\subseteq$ as partial order, since programs with more execution traces (respectively, more reachable states) satisfy less properties. Note that, in general, the abstract and the concrete partial order may differ. The core property $\alpha(T) \subseteq^\sharp R \iff T \subseteq \gamma(R)$ ensures that α and γ are monotonic functions; in particular, $x \subseteq^\sharp y$ implies $\gamma(x) \subseteq \gamma(y)$, so that $\subseteq^\sharp$ is the abstract version of the concrete order $\subseteq$: a less precise abstract information also represents, in the concrete, a less precise information. Moreover, proving in the abstract that a program semantics $P^\sharp$ satisfies a specification $S^\sharp$, i.e., that $P^\sharp \subseteq^\sharp S^\sharp$, implies that it also holds in the concrete $\gamma(P^\sharp) \subseteq \gamma(S^\sharp)$, which is the definition of soundness. Additionally, the existence of a Galois connection ensures that any concrete property $T \in C$ has a *best* abstraction in A. This best abstraction is exactly $\alpha(T)$. This means that $\alpha(T)$ is an over-approximation of T in that $T \subseteq \gamma(\alpha(T))$ — we have $\alpha(T) \subseteq^\sharp \alpha(T)$ so that, by the "if" part of the definition with $R = \alpha(T)$, we get $T \subseteq \gamma(\alpha(T))$. Moreover, if R is another over-approximation of T in that $T \subseteq \gamma(R)$, then $\alpha(T)$ is more precise, since $\alpha(T) \subseteq^\sharp R$ — if $T \subseteq \gamma(R)$ then, by the "only" if part of the definition, it follows that $\alpha(T) \subseteq^\sharp R$.

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



Cartesian abstraction α_c



Cartesian concretization γ_c

Given a set X of pairs $\langle x, y\rangle \in X$ (in blue in the figure above on the left), its abstraction is $\alpha_c(X) \triangleq \langle\{x \mid \exists y : \langle x, y\rangle \in X\}, \{y \mid \exists x : \langle x, y\rangle \in X\}\rangle$ (i.e., the red lines on the x and y axes in the figures above). The concretization is $\gamma_c(\langle X, Y\rangle) \triangleq \{\langle x, y\rangle \mid x \in X \wedge y \in Y\}$ (i.e., the red boxes above on the right). The abstract order $\subseteq_c$ is componentwise inclusion (i.e., $\langle X, Y\rangle \subseteq_c \langle X', Y'\rangle \iff X \subseteq X' \wedge Y \subseteq Y'$). 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: $1(x) = x$).

Example 2.7. Consider the set of pairs $X \triangleq \{\langle 0, 0\rangle, \langle 1, 1\rangle\}$. Then, its best abstraction as a pair of sets is, through the Cartesian abstraction, $X^\sharp = \alpha_c(X) = \langle\{0, 1\}, \{0, 1\}\rangle$. This abstract pair represents the set of pairs $\gamma_c(X^\sharp) = \{\langle 0, 0\rangle, \langle 0, 1\rangle, \langle 1, 0\rangle, \langle 1, 1\rangle\}$. We have $X \subseteq \gamma_c(X^\sharp)$. □

Not all abstractions are Galois connections, in which case one can always use a concretization function [Cousot and Cousot, 1992a]: the existence of a best abstraction is useful but not necessary to design a static analysis. A counter-example is provided by the polyhedra domain [Cousot and Halbwachs, 1978] which abstracts a set of points as a (possibly unbounded) convex polyhedron. Not every set of points has a best (i.e., smallest) over-approximation as a convex polyhedron; for instance, a disc does not, as shown by Euclid of Alexandria [fl. 300 BC]. Hence no total abstraction function α exists from sets of points to convex polyhedra.

2.7 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 trace semantics. Some abstractions are, however, not comparable. This is for instance the case when abstracting a natural number with its sign (positive, negative, or zero) and with its parity (even or odd): each one gives some information that cannot be recovered using the other abstraction only. It is natural to then ask whether it is possible, given two or more abstractions, to define an abstraction that contains as much information as all of them together, or that losses as much information as all of them. This is indeed the case: one can always define the most precise abstraction coarser than a set of abstractions, and the coarsest abstraction more precise than a set of abstractions. Abstract properties thus form a lattice with respect to the relation "is more abstract than" [Cousot and Cousot, 1977, 1979b]. 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. Additionally, the ability to build more precise abstractions by combining more basic abstractions with lattice operations helps the scalable design of modular static analyzers, as shown in §2.15.

Example 2.8. The coarsest abstraction more precise than both sign and parity represents, as properties, all combinations of being positive or negative with being even or odd, as well as the property "is zero." There is no property that can be exactly represented both as a sign and a parity property, so that the more precise abstraction less precise than both sign and parity abstractions is the trivial abstraction (i.e., no information). □

2.8 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, $\alpha(\mathcal{S}) \supseteq^\sharp \mathcal{S}^\sharp$, which in general is not equivalent, even for Galois connections). 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, $\alpha(\mathcal{S}) = \mathcal{S}^\sharp$).

As stated above, static analyzers are sound but incomplete. Nevertheless, abstract interpretation is not limited to reasoning about static analyzers, and we can find examples of complete abstractions in other fields of computer science. In particular, classic program proof methods can also be understood as reasoning on abstractions of the program semantics. For instance, Burstall's intermittent assertions proof method [Burstall, 1974] is based on the prefix abstraction [Cousot and Cousot, 1993] α_P, while Floyd's invariant assertions proof method [Floyd, 1967] is based on the reachability abstraction [Cousot and Cousot, 1987] α_R. The abstractions used in program proof methods are usually sound and complete. By undecidability, these abstractions yield proof methods that are not fully mechanizable. As a consequence, tools based on program proofs are not fully automatic, and may rely heavily on user help. By contrast, the abstractions used in static program analysis (such as the Cartesian abstraction in §2.6, the interval abstraction in §2.12, or type systems that do reject programs that can never go wrong [Cousot, 1997]) are usually sound and incomplete, in order to be fully mechanizable and lead to effective automatic analysis tools.

2.9 Abstract transformers

Recall that the concrete semantics $\mathcal{S}$ is usually expressed as the least fixpoint of a concrete transformer F: $\mathcal{S} = \mathrm{lfp}^{\subseteq} F$. Assuming now the existence of an abstract domain linked to the concrete one through a concretization γ, an abstraction α, or a Galois connection $\langle \alpha, \gamma \rangle$, it is natural to attempt to express the abstract semantics $\mathcal{S}^\sharp$ in a similar fixpoint form, $\mathcal{S}^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$, for some operator $F^\sharp$ to be determined.

The transformer $F^\sharp$ is said to be sound if $F \circ \gamma \subseteq \gamma \circ F^\sharp$. Intuitively, it means that the result of a computation step performed in the abstract ($F^\sharp$) represents (by γ) an overapproximation of the corresponding computation step performed in the concrete (F). When an abstraction function is provided instead of a concretization, the soundness criterion becomes $\alpha \circ F \subseteq^\sharp F^\sharp \circ \alpha$. Both are equivalent when $\langle \alpha, \gamma \rangle$ is a Galois connection and F and $F^\sharp$ are monotonic functions. In case of a Galois connection, there is a "best" abstract transformer, which is

$F^\sharp \triangleq \alpha \circ F \circ \gamma$. Intuitively, such an abstract transformer first evaluates its argument using the concrete transformer F and then constructs its best representation in the abstract domain using the abstraction function α. Note that this provides a mathematical definition of what the abstract transformer should compute, but not an algorithm to compute it (as F, α, and γ are generally not computable). Sometimes, even though such a best abstract transformer exists and is well defined mathematically, it may not be effectively or efficiently computable, in which case we can settle for a sound, non-optimal abstract transformer instead. In all cases, the main idea is that the abstract transformer $F^\sharp$ always over-approximates the result of the concrete transformer F (the "best" one, if any, providing the most precise over-approximation).

Example 2.9. Recall that the reachability semantics can be expressed as $\mathcal{R}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_R$, while the prefix trace semantics can be expressed as $\mathcal{P}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_P$. Given the Galois connection $\langle \alpha_R, \gamma_R \rangle$, we can check that, indeed, $F_R = \alpha_R \circ F_P \circ \gamma_R$ holds. In fact, the stronger commutation property $\alpha_R \circ F_P = F_R \circ \alpha_R$ holds. We can retrieve $F_R = \alpha_R \circ F_P \circ \gamma_R$ from the commutation by applying γ_R on the right of each member to get $\alpha_R \circ F_P \circ \gamma_R = F_R \circ \alpha_R \circ \gamma_R$ and noting that $\alpha_R \circ \gamma_R$ is the identity. $\square$

Example 2.10. Anticipating on §2.12, we consider the interval abstraction that abstracts a set of integers $X \subseteq \mathbb{Z}$ into an interval, that is, a (possibly infinite) lower and an upper bound: $\alpha_\iota(X) \triangleq [\min X, \max X]$. The concretization of an interval $[\ell, h]$ is the set of integers it contains: $\gamma_\iota([\ell, h]) \triangleq \{\, x \in \mathbb{Z} \mid \ell \leq x \leq h \,\}$. The pair $\langle \alpha_\iota, \gamma_\iota \rangle$ forms a Galois connection. Consider finally the concrete operator $F(X) \triangleq \{\, x{+}1 \mid x \in X \,\}$ that increments all the values in its argument by one. Then the best abstraction $F^\sharp \triangleq \alpha_\iota \circ F \circ \gamma_\iota$ of F simply increases interval bounds $F^\sharp([\ell, h]) = [\ell + 1, h + 1]$. Note that not all sets $X \subseteq \mathbb{Z}$ are machine-representable (as some contain infinitely may elements), but $\alpha_\iota(X)$ is always machine-representable (as a pair of integers). Likewise, neither α_ι, γ_ι not F is effectively computable, while $F^\sharp$ is easy to implement.

The alternate abstract operator defined as $F^\sharp([\ell, h]) = [-\infty, +\infty]$ is sound, in that $\forall [\ell, h] : (F \circ \gamma_\iota)([\ell, h]) \subseteq (\alpha_\iota \circ F^\sharp)([\ell, h]) = \alpha_\iota([-\infty, \infty]) = \mathbb{Z}$. Naturally, this second $F^\sharp$ version is not optimal. $\square$

2.10 Sound abstract fixpoint semantics

Given a concrete fixpoint semantics $\mathcal{S} = \text{lfp}^{\subseteq} F$ and an 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 or when $\alpha \circ F \circ \gamma$ cannot be effectively or efficiently implemented), we can now consider the abstract fixpoint $\mathcal{S}^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$. Under appropriate hypotheses [Cousot and Cousot, 1979b], the abstract semantics is then guaranteed to be sound: $\mathcal{S} = \text{lfp}^{\subseteq} F \subseteq \gamma(\mathcal{S}^\sharp) = \gamma(\text{lfp}^{\subseteq^\sharp} F^\sharp)$. 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.

When considering static analysis, once an algorithm implementing $F^\sharp$ is constructed, the effective computation or approximation of $\mathcal{S}^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$ can be achieved through iteration of $F^\sharp$ (described earlier in §2.3), when such iteration converge in finite time, or using extrapolation operators when they do not, as described later in §2.14. This effectively decouples the problem of abstracting the effect of elementary execution steps (in $F^\sharp$), and the problem of considering program executions comprised of an arbitrarily long sequence of such steps.

The fixpoint transfer theorems [Cousot and Cousot, 1979b] transporting a soundness proof on operators into a soundness proof on fixpoints of operators also yield the basis to formally verify the soundness of static analyzers using theorem provers or proof checkers [Besson et al., 2009, Jourdan et al.].

2.11 Sound and complete abstract fixpoints semantics

In some circumstances, the soundness property discussed in the last section (semantic inclusion) can be strengthened into a soundness and completeness property (semantic equality). More precisely, under the hypotheses of Tarski's fixpoint theorem [Tarski, 1955] and the additional commutation hypothesis [Cousot and Cousot, 1979b] $\alpha \circ F = F^\sharp \circ \alpha$ for Galois connections, we have $\alpha(\mathcal{S}) = \alpha(\text{lfp}^{\subseteq} F) = \mathcal{S}^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$. Otherwise stated, the fact that the concrete semantics is a least fixpoint is preserved in the abstract.

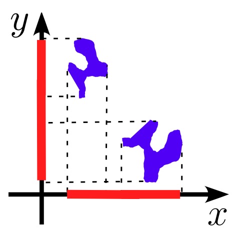
Example 2.11. $\mathcal{R}[\![t]\!]I = \mathrm{lfp}^{\subseteq} F_R$ and $\alpha_R \circ F_P = F_R \circ \alpha_R$ imply that $\mathcal{R}[\![t]\!]I \triangleq \alpha_R(\mathcal{P}[\![t]\!]I) = \alpha_R(\mathrm{lfp}^{\subseteq} F_P) = \mathrm{lfp}^{\subseteq} F_R$. The intuition is that, in order to reason on reachable states, it is unnecessary to consider execution traces. $\square$

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.

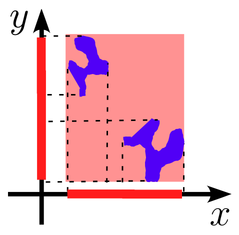
2.12 Infinite abstraction example: interval abstraction

It may seem that, to be effectively computable, a semantics must be abstracted into a *finite* semantic domain. This is not the case. It is possible, and desirable, to choose, as abstract set of properties, an infinite set, only requiring that every abstract property is finitely representable. One example is the set of intervals $[\ell, h]$, where $\ell \in \{-\infty\} \cup \mathbb{Z}$ and $h \in \{+\infty\} \cup \mathbb{Z}$, which is infinite but composed of finitely representable elements (pairs of bounds).

Assume that a program has numerical (integer, floating-point, 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 (which maps each variable to its value set), and then by the interval abstraction independently on each variable:



Interval abstraction α_c



Interval concretization γ_c

The abstract semantics is then $\alpha_\iota(\mathcal{R}[\![t]\!]I)$, 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 discrete and 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 $\wp(S)$, so that it is still useful, for efficiency reasons, to consider intervals instead of state sets.

One may wonder why infinite abstractions are of any practical interest since computers can only do finite computations anyway. The answer comes from a very simple example: $P \equiv$ `x=0; while (x<=`n`) { x=x+1 }`, where the symbol $n \geqslant 0$ stands for a numeric constant (so that we have an infinite family of programs for all integer constants $n = 0, 1, 2, \ldots$) [Cousot and Cousot, 1992b]. For any given constant value of n, an abstract interpretation-based analyzer using the interval abstraction [Cousot and Cousot, 1976] will determine that, on loop body entry, $\mathtt{x} \in [0, n]$ always holds. 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 (§2.14), without the benefit of avoiding the combinatorial state explosion.

2.13 Abstract domains and functions

Encoding program properties uniformly (e.g., as terms in theorem provers or BDDs [Bryant, 1986] 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., for intervals: either $\emptyset$ or a pair $[\ell, h]$ of numbers or infinities 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 [Jeannet and Miné, 2007, 2009] for numerical abstract domains — abstracting (possibly infinite) sets of points in vector spaces.

2.14 Convergence acceleration by extrapolation and interpolation

Even when an effective abstraction $F^\sharp$ of the concrete transformer F is known, it may not be possible or practical to compute its fixpoint $\mathcal{S}^\sharp = \mathrm{lfp}^{\subseteq^\sharp} F^\sharp$, necessary to abstract the concrete semantics $\mathcal{S} = \mathrm{lfp}^{\subseteq} F$. Convergence acceleration methods [Cousot and Cousot, 1976] have been proposed in order to approximate effectively and efficiently such abstract fixpoints.

2.14.1 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* [Cousot and Cousot, 1976, 1977, Cousot, 1978], 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 $x \mathbin{\triangledown} \emptyset = x$, $\emptyset \mathbin{\triangledown} x = x$, $[a,b] \mathbin{\triangledown} [c,d] \triangleq [\ell,h]$, 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).

Example 2.12. 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. □

2.14.2 Narrowing

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

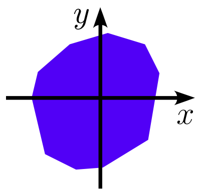
$X^0 \nabla ([1,1] \cup ((X^0 + [2,2]) \cap (-\infty, 99])) = \emptyset \nabla [1,1] = [1,1]$, $X^2 = X^1 \nabla ([1,1] \cup ((X^1 + [2,2]) \cap (-\infty, 99])) = [1,1] \nabla [1,3] = [1,+\infty)$. This result is clearly imprecise as, in the concrete, `x` is always less than 99 within the loop.

After that upwards iteration (where intervals are wider and wider), we can go on with a downwards iteration (where intervals are narrower and narrower, hence, more precise). To avoid infinite decreasing chains (such as $[0,+\infty)$, $[1,+\infty)$, $[2,+\infty)$, $\dots$, which limit is $\emptyset$), we use an interpolation operator called a *narrowing* [Cousot and Cousot, 1977, Cousot, 1978] Δ for the equation $X = X \Delta F_\iota(X)$. The narrowing should ensure both soundness and termination. In finite abstract domains, narrowings are useless and can be replaced with the intersection $\cap$.

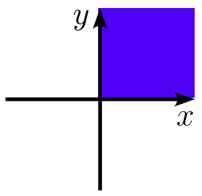
An example of narrowing for intervals is $\emptyset \Delta x = \emptyset$, $x \Delta \emptyset = \emptyset$, $[a,b] \Delta [c,d] = [\ell, h]$ where $\ell = c$ when $a = -\infty$ and $\ell = a$ when $a \neq -\infty$ and similarly $h = d$ when $b = +\infty$ and otherwise $h = b$. So, infinite bounds are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

Example 2.13. For the program `x=1; while (x<100) { x=x+2 }`, the downwards iteration is now $Y = Y \Delta F_\iota(Y)$ starting from the fixpoint obtained after widening: $Y^0 = [1,+\infty)$, $Y^1 = Y^0 \Delta([1,1] \cup ((Y^0 + [2,2]) \cap (-\infty, 99])) = [1,+\infty) \Delta ([1,1] \cup ([3,+\infty) \cap (-\infty,99])) = [1,+\infty) \Delta [1,99] = [1,99]$. The next iterate is $Y^2 = Y^1 \Delta ([1,1] \cup ((Y^1 + [2,2]) \cap (-\infty, 99])) = [1,99] \Delta ([1,1] \cup ([3,101] \cap (-\infty, 99])) = [1,99] \Delta [1,99] = [1,99] = Y^1$ so that a fixpoint is reached (although it may not be the least one, in general). $\square$

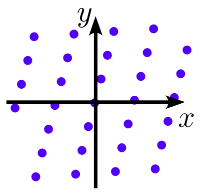
Of course, for finite abstractions (where strictly increasing chains are finite) no widening nor narrowing is needed since the brute-force iteration in the finite abstract domain always terminates. However, to avoid a time and space explosion, convergence acceleration with widening and narrowing may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



Polyhedra
[Cousot and Halbwachs, 1978]:
too costly



Signs
[Cousot and Cousot, 1979b]:
too imprecise



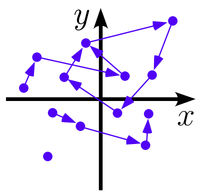
Linear congruences
[Granger, 1991]:
out of scope

Figure 2.1: Classic abstract domain examples *not* used in ASTRÉE.

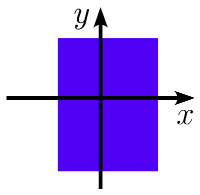
2.15 Combination of abstract domains

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. Figure 2.2 presents a few abstract domain examples used in the ASTRÉE static analyzer [Cousot et al., 2007a]. Such abstract domains (and more) are described in more details in §3.9–3.10.

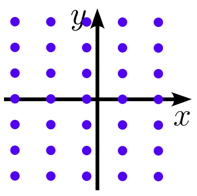
The classic abstract domains presented in Figure 2.1, however, are not used in ASTRÉE because they are either too imprecise, not scalable, difficult to implement correctly (for instance, soundness may be an issue in the event of floating-point rounding), or out of scope (determining program properties which are usually of no interest to prove the specification).



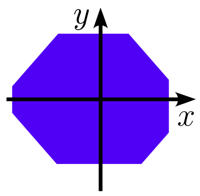
Collecting semantics
[Cousot and Cousot, 1977]:
prefix traces



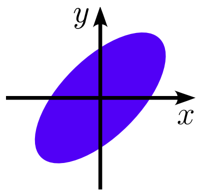
Intervals
[Cousot and Cousot, 1976]:
$\mathtt{x} \in [a, b]$



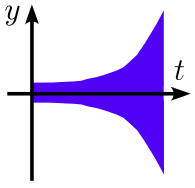
Simple congruences
[Granger, 1989]:
$\mathtt{x} \equiv a \pmod{b}$



Octagons
[Miné, 2006a]:
$\pm\mathtt{x} \pm \mathtt{y} \leqslant a$



Ellipses
[Feret, 2004]:
$\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{xy} \leqslant d$



Exponentials
[Feret, 2005a]:
$-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

Figure 2.2: Example abstract domains used in the Astrée static analyzer. See also Chapter 3.

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* [Cousot and Cousot, 1979b] 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 D_1, …, D_n and computed $\mathrm{lfp}^{\subseteq} F_1 \in D_1$, …, $\mathrm{lfp}^{\subseteq} F_n \in D_n$ in these abstract domains, 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 allow 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 $\mathcal{C}[\![t]\!]I \subseteq \gamma_1(r_1) \cap \cdots \cap \gamma_n(r_n)$", i.e., we perform a fixpoint computation simultaneously on all the abstract domains and apply a *reduction* function ρ to propagate information between the abstract domains at each iteration of the fixpoint computation. Considering again the combination of intervals and parity, the reduction function would give: $\rho(\langle [0, 100], \mathit{odd}\rangle) = \langle [1, 99], \mathit{odd}\rangle$.

To define ρ, first consider the case of two abstract domains D_1 and D_2 with Galois connections $\langle \alpha_1, \gamma_1\rangle$ and $\langle \alpha_2, \gamma_2\rangle$. 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. This two-by-two reduction may be less precise than a global reduction [Cousot et al., 2011], but is definitely easier to design and implement. For the sake of efficiency, an over-approximation of the iterated pairwise reduction can be used, where only some of the reductions ρ_{ij} are applied, in a fixed order [Cousot et al., 2006]. These reduction ideas also apply in the absence of Galois connections. Detailed examples are given in §3.11.

2.16 Partitioning abstractions

Another useful tool to design complex abstractions from simpler ones is *partitioning* [Cousot, 1981]. In its simplest form, it consists in considering a collecting semantics on a powerset concrete domain $C = \wp(S)$, and a finite partition (or covering) $S_1, \ldots, S_n$ 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 $\alpha(X) \triangleq \langle \alpha_1(X \cap S_1), \ldots, \alpha_n(X \cap S_n) \rangle$.

Example 2.14. Consider refining the interval domain by maintaining a distinct interval for positive values and for negative values of x: $\gamma(\langle [\ell^+, h^+], [\ell^-, h^-] \rangle) \triangleq ([\ell^+, h^+] \cap [0, +\infty)) \cup ([\ell^-, h^-] \cap (-\infty, 0))$. Furthermore, $\alpha(X) = \langle [\min X \cap [0, +\infty), \max X \cap [0, +\infty)], [\min X \cap (-\infty, 0), \max X \cap (-\infty, 0)] \rangle$. 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 (redundant representations are a common and useful occurrence in abstract domains, for instance representing the same polyhedron [Cousot and Halbwachs, 1978] with various intersections of half-spaces). 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 [Bourdoncle, 1992] 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 (but possibly unbounded) sets of pairs in $D_s \times D_d$, with concretization $\gamma(\{\langle s_1, d_1 \rangle, \ldots, \langle s_n, d_n \rangle\}) \triangleq \bigcup_{i=1}^{n} \gamma_s(s_i) \cap \gamma_d(d_i)$.

In contrast to (possibly reduced) products, which are useful to express conjunctions of heterogeneous properties coming from incomparable abstractions (such as intervals and parity), partitioning abstractions are useful to express disjunctive properties, where several elements of the same domain express distinct disjuncts (such as a disjunction of disjoint intervals) instead of a single domain element. In general, a precise analysis requires both reduced products and partitioning abstractions.

Later sections present examples of partitioning abstractions for program control states (§3.4), program traces (§3.6), and program data states (§3.9.5).

2.17 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 [Cousot and Cousot, 1976] 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 and execute their programs (which may not terminate, while static analysis always does).

An abstract interpretation-based static analyzer is built by combining abstract domains, e.g., using a reduced product 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: complex statements and expressions are broken into a simpler language with few constructions, so that F_D can be constructed by combining a small set of primitives directly handled by abstract domains (such as simple assignments and tests). Then, an over-approximation of $\mathrm{lfp}^{\subseteq} F_D$ is computed by iteration with convergence acceleration 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.

2.18 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` during execution is always between two bounds $[\ell_{\mathtt{x}}, h_{\mathtt{x}}]$.

2.19 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.

2.20 Verification in the abstract

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

An example is the rule of signs $\texttt{pos} \times \texttt{pos} = \texttt{pos}$, $\texttt{neg} \times \texttt{neg} = \texttt{pos}$, $\texttt{pos} \times \texttt{neg} = \texttt{neg}$, etc., where $\gamma(\texttt{pos}) \triangleq \{z \in \mathbb{Z} \mid z \geqslant 0\}$ and $\gamma(\texttt{neg}) \triangleq \{z \in \mathbb{Z} \mid z \leqslant 0\}$. 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 ($\texttt{pos} + \texttt{neg}$ 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. Indeed, intervals is the most abstract abstraction which is complete for determining the sign of additions. As another example, we can consider the ASTRÉE static analyzer, described in details in Chapter 3. Indeed, ASTRÉE checks for run-time errors, such as overflows, which require inferring intervals. Nevertheless, the interval domain is not sufficient for this task: although it can express the optimal intervals, inferring precise intervals requires more complex domains, such as the filter domain illustrated in Figure 2.2 and explained in details in §3.10.1.

In general, a most abstract complete abstraction to prove a given abstract specification does exist [Giacobazzi et al., 2000] 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 automatized — e.g., because designing efficient data structures and algorithms for abstract domains is not automatizable — and out of the scope of automatic refinement of abstractions [Cousot et al., 2007b]. The refinement process to build an analyzer such as ASTRÉE from an interval analyzer is thus a complex, manual one.

3

Verification of Synchronous Control/Command Programs

We now present Astrée, a static analyzer for automatically verifying the absence of runtime errors in synchronous control/command embedded C programs [Blanchet et al., 2002, 2003, Cousot et al., 2005, 2006, 2007a, Mauborgne, 2004], successfully used in aeronautics [Delmas and Souyris, 2007] and aerospace [Bouissou et al., 2009] and now industrialized by AbsInt [AbsInt, Angewandte Informatik].

3.1 Analyzed C subset

Astrée can analyze a fairly large subset of C99. The most notable unsupported features are: non-local jumps (`longjmp`) and recursive procedures. Such features are most often unused (or even forbidden) in embedded programming to keep a strict control over resource usage and control-flow. Parallel programs are now supported as a recent extension, and thus discussed separately, in Chapter 6.

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

```
double acos(double x)
{
    double r;
    __ASTREE_assert(( x >= -1. && x <= 1. ));
    __ASTREE_known_fact(( r >= 0. && r <= 3.2 ));
    return r;
}
```

Figure 3.1: Stub for the arccos function.

relevance to the application domain (§3.9), while others were developed specially for it (§3.10).

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. A stub example for the arccos function is provided in Figure 3.1: using ASTRÉE-specific directives, it checks that the argument is a valid floating-point number in the range $[-1, 1]$ (any violation of this specification triggers an alarm) and states that the returned value is in $[0, 3.2]$ (to be on the safe side and allow implementations that overflow the mathematical interval $[0, \pi]$). Local variables without a specific initializer (such as `r` in `acos`) are considered to be created uninitialized (and so can take any initial value in their type). Moreover, input variables set by the environment (such as variables mapped to hardware registers updated asynchronously with sensor values) need to be declared as `volatile`. Their range should also be specified (otherwise, they are considered to have the full range of their type, including infinities and *NaN* for floating-point numbers).

3.2 Operational semantics of C

ASTRÉE is based on the C ISO/IEC 9899:1999/Cor 3:2007 standard [ISO/IEC JTC1/SC22/WG14 Working Group, 2007], 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 potentially leads to a runtime error. Runtime errors that are looked for include: overflows in unsigned or signed integer or floating-point arithmetics, integer or floating-point 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 emits an alarm and 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 but, at least, the instruction at the root of this undefined

```
void main ()
{
    int i = 0;
    while (i <= 10)
        i = i + 1;
}
```

Figure 3.2: Correct program that can be proved free of run-time error by Astrée.

```
% astree loop.c --exec-fn main
/*
[...]
Executing <main>
Time spent in analysis of function main: 0.003305 s
The analysis ended without errors or warnings
/* Max heap size: 248.00 Mb, 4 major collections */
/* 1 procedure(s) executed */
%
```

Figure 3.3: Analysis with Astrée of the correct program in Figure 3.2.

behavior is faithfully 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 other means, to be impossible.

3.3 Analysis examples

Figure 3.2 shows a very simple C program featuring a loop that increments `i` from 0 to 11. The command-line invocation of Astrée on this example is presented in Figure 3.3: only the file name and entry point need to be specified. The analysis result (with irrelevant parts omitted)

```
void main()
{
    int i;
    int a[10];
    for (i=0;i<10;i++)
         if (a[i]) break;
    a[i] = 0;
}
```

Figure 3.4: Program with a possible runtime error.

```
% astree alarm.c --exec-fn main
[...]
Executing <main>
alarm.c:5.2-6.19:up iteration #0
alarm.c:5.2-6.19:down iteration #0
alarm.c:7.4-5:[call#main@1:]: WARN: out-of-bound array index
[0, 10] not included in [0, 9]
alarm.c:7.2-6:[call#main@1:]: WARN: invalid dereference:
dereferencing 4 byte(s) at offset(s) 4*[0;10] may overflow
the variable a of byte-size 40
Time spent in analysis of function main: 0.009742 s
[...]
%
```

Figure 3.5: Analysis with ASTRÉE of the erroneous program in Figure 3.4.

is also shown: in this simple case, ASTRÉE is able to prove the absence of runtime errors (in particular, the absence of integer overflow at line 5), as witnessed by the absence of any alarm message during the execution of `main` (i.e., between lines 4 and 5 of Figure 3.3).

Figure 3.4 presents a slightly more complex example: the first non-zero element in the local array `a` is searched and set to zero. As the array is local, its initial contents is random, and ASTRÉE considers that the loop can be exited by the `break` statement after 0, 1,. . . or 9 loop iterations, or when the condition `i<10` is false. In that last case, the instruction `a[i] = 0` is erroneous as $\texttt{i} = 10$. The analysis by ASTRÉE in Figure 3.5 shows two alarms related to writing into `a` at line 7 (ASTRÉE can also report reading of uninitialized variables, but these alarms have been omitted in the report to focus on the alarms at line 7). Each alarm, recognizable by the `WARN` keyword, indicates the source code position of the offending statement as well as the call context (here, in the function `main`, see §3.4 on the flow- and context-sensitivity of ASTRÉE), the nature of the alarm, and the exact specification that was violated (e.g., a range computed by the analyzer not included in the range required to ensure the absence of alarm). The first alarm states that the index `i` is invalid in the array `a`, and the second one that the access to `a[i]` is a memory error, which is a direct consequence of the first alarm (note that invalid array indices do not necessarily result in a memory error, for instance when the array is embedded into a larger structure, as in `t.a[10]` where `t` has type `struct { int a[10]; int b; } t`, hence the need for ASTRÉE to distinguish between these two kinds of alarms). Figure 3.5 also illustrates some information about the analysis progress: each increasing and decreasing iteration in a fix-point computation (here, caused by the `for` loop) is logged, which is useful to judge the amount of work spent analyzing loops (which often dominates analysis time) and the effectiveness of the extrapolation operators in reducing the number of abstract iterations (here, two abstract iterations are sufficient, although the loop has ten iterates in the concrete). If one corrects the program, for instance by iterating from 0 to 9 instead of 10, then both alarms disappear and ASTRÉE is able to prove the absence of runtime errors.

Figure 3.6 gives an example program performing an integer division by `x` in a context where $7\mathtt{y}^2 - 1 = \mathtt{x}^2$ and $\mathtt{x} \in [-32766, 32766]$, $\mathtt{y} \in [-4680, 4680]$. Its analysis is shown by Figure 3.7. In that case, ASTRÉE warns of a possible division by 0. However, there does not exist any solution to the Diophantine equation $7\mathtt{y}^2 - 1 = \mathtt{x}^2$ and, *a fortiori* no solution such that $\mathtt{x} = 0$ (to produce a division by zero) and $\mathtt{x} \in [-32766, 32766]$, $\mathtt{y} \in [-4680, 4680]$ (a condition which was added in order to prevent any arithmetic overflow in the test `7*y*y - 1 == x*x`). Hence, this is an example of false alarm and incompleteness of ASTRÉE. There are good theoretical reasons for this incompleteness: the theory of Diophantine equations is extremely complex and no general algorithm to solve them can exist (Hilbert's tenth problem, answered negatively by Matiyasevich). Moreover, designing specific abstract domains to solve restricted classes of equations would be time-consuming for analysis designers and result in a slower analysis, for no practical benefit as these equations do not actually appear in the synchronous software targeted by ASTRÉE (as opposed to other features for which specific solutions have been designed, see §3.10).

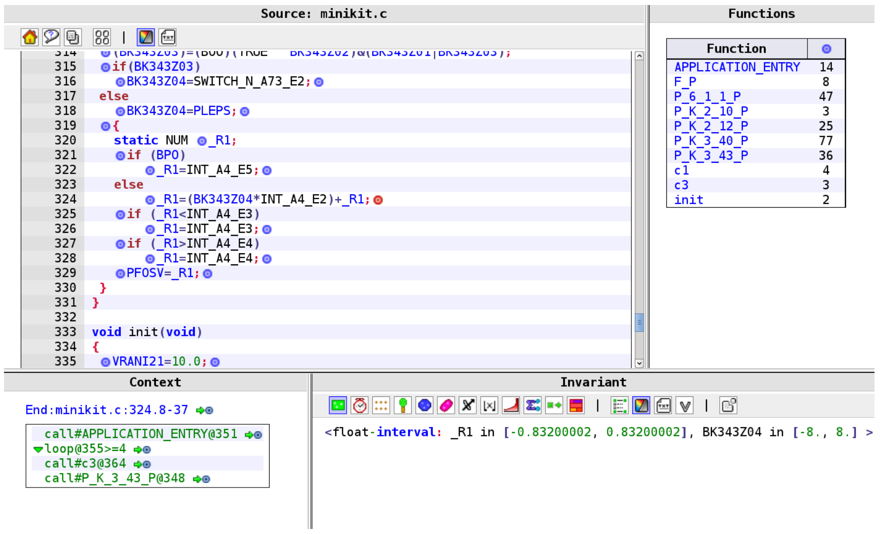
Finally, ASTRÉE is also able to export all the properties (with various degrees of verbosity) inferred during the analysis to a binary file, which can then be explored interactively at leisure using a graphical interface. Figure 3.8 presents a screenshot of an academic graphical interface prototype we designed at the end of ASTRÉE's development. It

```
void main()
{
    int x, y;
    if (-4681 < y && y < 4681 && x < 32767 && -32767 < x &&
        7*y*y - 1 == x*x)
    {
        y = 1 / x;
    }
}
```

Figure 3.6: Another correct program.

```
% astree false-alarm.c --exec-fn main
[...]
Executing <main>
false-alarm.c:5.9-14:[call#main@1:]: WARN: integer division by
zero [-32766, 32766]
Time spent in analysis of function main: 0.002006 s
[...]
%
```

Figure 3.7: Analysis with ASTRÉE of the correct program in Figure 3.6 showing a false alarm.



```
 if(BK343Z03)
   BK343Z04=SWITCH_N_A73_E2;
 else
   BK343Z04=PLEPS;
 {
   static NUM _R1;
   if (BPO)
       _R1=INT_A4_E5;
   else
       _R1=(BK343Z04*INT_A4_E2)+_R1;
   if (_R1<INT_A4_E3)
       _R1=INT_A4_E3;
   if (_R1>INT_A4_E4)
       _R1=INT_A4_E4;
   PFOSV=_R1;
 }
}

void init(void)
{
 VRANI21=10.0;
```

Function	
APPLICATION_ENTRY	14
F_P	8
P_6_1_1_P	47
P_K_2_10_P	3
P_K_2_12_P	25
P_K_3_40_P	77
P_K_3_43_P	36
c1	4
c3	3
init	2

```
End:minikit.c:324.8-37

  call#APPLICATION_ENTRY@351
 loop@355>=4
  call#c3@364
  call#P_K_3_43_P@348
```

```
<float-interval: _R1 in [-0.83200002, 0.83200002], BK343Z04 in [-8., 8.] >
```

Figure 3.8: Academic graphical interface for ASTRÉE.

displays the range (§3.9.1) of two variables at some hi-lighted point in some call context (§3.4). This prototype GUI has served as the basis for a mature, industrial-strength interface developed by AbsInt as part of ASTRÉE's industrialisation process (§3.16).

3.4 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, while the later whether it can distinguish 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* component: $S \triangleq C \times D$. 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 (§2.16), i.e., keeping an abstraction of the data state D (§3.7) 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 Chapter 6).

3.5 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

$$\begin{tikzcd}
\text{trace abstraction of } \wp((C \times (\mathtt{V} \to \mathbb{V}))^*) \arrow[d] & (\S 3.6) \\
\text{memory abstraction of } \wp(\mathtt{V} \to \mathbb{V}) \arrow[d] & (\S 3.7) \\
\text{scalar value abstraction of } \wp(\mathtt{V}_c \to \mathbb{V}_b) \arrow[d] & (\S 3.8) \\
\text{product of numerical abstractions of } \wp(\mathtt{V}_n \to \mathbb{R}) & (\S 3.9\text{–}3.10)
\end{tikzcd}$$

Figure 3.9: Hierarchy of semantics and domains in ASTRÉE.

rich concrete universe). Indeed, although ASTRÉE ultimately abstracts a collecting semantics of prefix 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 $(C \times (\mathtt{V} \to \mathbb{V}))^*$). The functor 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 an abstraction of scalar values (i.e., pointers and numerical values), itself parameterized by a purely numerical abstraction, where each abstraction handles simpler and simpler data-types. This hierarchy is shown in Figure 3.9

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 (§3.11), which makes it easy to add or remove such domains.

3.6 Trace abstraction

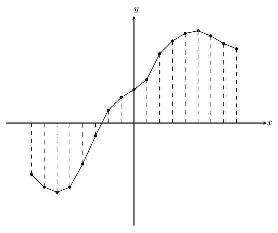
Floyd's program proof method [Floyd, 1967] is complete in the sense that all invariance properties can be proved using only state invariants, that is, sets of states. However, this does not mean that it is always the best solution, in the sense that such invariants are not always the easiest way to represent concisely or compute efficiently these invariance properties. It is sometimes much easier to use intermittent invariants as in Burstall's proof method [Burstall, 1974] (i.e., an abstraction of the prefix trace semantics [Cousot and Cousot, 1993]).

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 the 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 cause a 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 to be joined together. In other words, context sensitivity (§3.4) is not enough to guarantee a high level of precision, and some other mechanisms to distinguish sets of reachable states should be implemented (such as path-sensitivity).

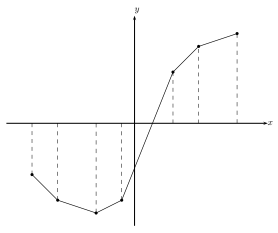
The main difficulty which needs to be solved in order to achieve this is to compute automatically good partitions. In many cases, such information can be derived from the control-flow of the program to analyze (e.g., which branch of some if-statement was executed or how many times the body of some loop 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 prefix executions up to that point. This is the reason why ASTRÉE abstracts prefix traces rather than just reachable states. An element of the trace partitioning abstract domain [Mauborgne and Rival, 2005, Rival and Mauborgne, 2007] should map each element of a *finite partition* of the traces of the program to an abstract invariant. Thus, it is a partitioning abstraction (§2.16), 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



(a)



(b)

Figure 3.10: Regular and irregular interpolations.

```
float f(float x)
{
    const float ys[5] = { 1., 10., 10., 20.,  1. };
    int i = floor(x);
    if (i <  0) return ys[0];
    if (i >= 4) return ys[4];
    return ys[i] + (ys[i+1] - ys[i]) * (x - i);
}
```

Figure 3.11: One-dimensional regular linear interpolation function.

```
float f(float x)
{
    const float xs[5] = { 0.,  1.,  8., 10., 15. };
    const float ys[5] = { 1., 10., 10., 20.,  1. };
    int i = 0;
    if (x <  xs[0]) return ys[0];
    if (x >= xs[4]) return ys[4];
    while (i < 4 && x > xs[i+1]) i++;
    return ys[i] + (ys[i+1] - ys[i]) *
           (x - xs[i]) / (xs[i+1] - xs[i]);
}
```

Figure 3.12: One-dimensional irregular linear interpolation function.

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 Figure 3.10.(a), the localization can be done using simple arithmetic. A simple, one-dimensional regular linear interpolation implementation function is presented in Figure 3.11, using a truncation to locate the cell `i`. In the more general case of non-regular grids, as in Figure 3.10.(b), localizing the input needs to be done using a search algorithm. An example implementation is shown in Figure 3.12 using a linear search loop. Using plain interval arithmetic, the evaluation of the return expression at line 7 of Figure 3.11 gives: `ys[i]` $+$ (`ys[i+1]` $-$ `ys[i]`) $\times$ (`x` $-$ `i`) $=$ $[1, 20] + ([1, 10] - [1, 20]) \times ([0, 5) - [0, 4]) = [-94, 96]$, which is far from the optimal interval $[1, 20]$ (i.e., the result of the interpolation should lie between the lowest and highest tabulated function value). This low precision comes from the way each sub-expression is abstracted independently into an interval, while implicit relationships between `ys[i+1]`, `ys[i]`, `x` and `i` are completely forgotten. In Figure 3.12, a naive interval analysis considers that the sub-expression `xs[i+1]` $-$ `xs[i]` evaluates to $[1, 15] - [0, 10] = [-9, 15]$, which triggers a spurious division by zero alarm. In both cases, the interpolation can be precisely analyzed only if a close relationship between the input values `x` and the grid cells `i` can be established, so that the 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" (where p_i relates i, `i`, `x`, `xs[i]`, `xs[i+1]`, `ys[i]`, and `ys[i+1]`) and using such relations when evaluating the return expression at line 7 (resp. line 9) of Figure 3.11 (respectively, Figure 3.12). This is possible since the cell the input is contained in is an abstraction of the history of the execution: in the regular case of Figure 3.11, an adequate partition criterion is the value stored into `i` at line 4 while, in the non-regular case of Figure 3.12, it is the number of iterations spent in the loop at line 8. ASTRÉE is able to partition the analysis (i.e., perform case analysis) based on these two kinds of criteria, as well as other criteria (such as control-flow splitting through "if" statements). Trace partitioning generalizes the concept of *path sensitivity* sometimes used in static analysis. However, to avoid a combinatorial explosion of the number of cases, partitions are periodically merged (e.g., at the end of functions): the partitioning is only local.

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 the following 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, such as assignments, 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 at the selected statements is itself *dynamic*, which means that the sets of partitions are chosen during the analysis, and not fixed statically. This ensures both precision and efficiency.

3.7 Memory abstraction

Given a concrete program state $(c, d) \in S \triangleq C \times D$, 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, $d \in \mathtt{V}(c) \to \mathbb{V}$. Note also that, for each c, $\mathtt{V}(c)$ is finite. 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 finite, fixed, and statically known. This simplification is only possible because ASTRÉE takes care not to abstract together program states from different control locations.

The C language offers a rich type system that includes a few fixed *base types* (machine integers of various size and signedness, floating-point types of various size, pointers to data or functions) 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 recursively and statically into finite collections of *cells* of base type occupying disjoint memory locations. 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 (i.e., values of variables of base type). 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 $\mathtt{V}_c \triangleq \{ v0a, v0b, v1a, v1b \}$, where $v0a$ stands for `v[0].a`, $v0b$ for `v[0].b`, $v1a$ for `v[1].a`, and $v1b$ 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 $\mathtt{V}'_c \triangleq \{ va, vb \}$, where va abstracts the union of values of `v[0].a` and `v[1].a`, while vb 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 $\mathtt{V}_c \triangleq \{ v0a, v0b, v1a, v1b \}$ (the case of pointer access is considered in §3.8). Depending on the possible abstract value of `i` (exemplified here in the interval domain), it can be translated into either $v0a$ `=` $v0b$ `+ 1` (if `i` is $[0,0]$), $v1a$ `=` $v0b$ `+ 1` (if `i` is $[1,1]$), or `if (?)` $v0a$ `=` $v0b$ `+ 1 else` $v1a$ `=` $v0b$ `+ 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 $\mathtt{V}'_c \triangleq \{ va, vb \}$ is considered instead of $\mathtt{V}_c$, then the statement is interpreted as `if (?)` va `=` vb `+ 1` whatever the value of `i`, which always involves a weak update (hence, it is less precise in the case where `i` is fully determined).

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 C99 standard (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, we also need to support it in ASTRÉE. In a way similar to 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 [Miné, 2006b]. 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. This knowledge currently includes: how signed and unsigned integer types can be decomposed into or recomposed from individual bytes, the equality of

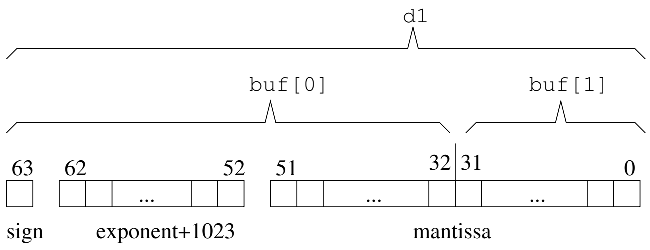


Figure 3.13: IEEE 754-1985 representation of 64-bit double precision floating-point numbers. `buf[0]` and `buf[1]` denote respectively the high-order and low-order 32-bit part of a double `d1` on a big endian architecture (used in Figure 3.14).

```
double int_to_double(int x)
{
    unsigned buf[2];
    double d1, d2;
    buf[0] = 0x43300000;
    buf[1] = 0x80000000;
    d1 = *((double*)buf);
    buf[1] ^= (unsigned)x;
    d2 = *((double*)buf);
    return d2 - d1;
}
```

Figure 3.14: Manual cast from integer to double.

unsigned and (two's complement) signed versions of the same integer type modulo $2^{\texttt{sizeof}}$, and the decomposition of floating-point numbers into their binary representation (sign, mantissa, and exponents fields) according to the IEEE 754–1985 [IEEE Computer Society, 1985] norm (see Figure 3.13).

The memory domain can also handle accesses through *pointer casts*, which can simulate the effect of `union` types. For instance, given the variable `unsigned i`, the expression `*((unsigned char*)&i)` references the first byte of `i` (i.e., `i & 255` on a little endian architecture). The difference is that, for `union` types, the memory block of a variable can be populated statically, at creation time, with a set of (possibly overlapping) cells based on the type of the variable, while pointer casts can dynamically reinterpret a variable according to some new type never associated to the variable before. The decomposition of variables into cells is thus dynamic in Astrée and evolves according to a variable use during the analysis. In particular, new cells can be *materialized*, and their initial values are synthesized through a reduction process by exploiting information from preexisting cells overlapping them. We end this section with another example, in Figure 3.14. This complex function converts a 32-bit signed integer `x` into a 64-bit floating-point number. A representation of the floating-point numbers $2^{52} + 2^{31}$ and $2^{52} + 2^{31} + \texttt{x}$ is first constructed using only integer and bit-level operations. These are stored respectively in `d1` and `d2`. Then, a floating-point subtraction is used to recover `x` as a floating-point value. Such code is very common (in C programs, C libraries, or directly inlined by compilers) when targeting PowerPC processors, as these lack a proper conversion opcode implementing directly `(double)x`. A precise handling of pointer casts and intimate knowledge of floating-point binary representations allows Astrée to analyze precisely such code and deduce that the returned value has the same range as the argument.

3.8 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 at a very low level as: either a pair $\langle \texttt{v}, o \rangle$ composed of a variable identifier (or name) `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 $[0, \texttt{sizeof(v)})$.

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 variable identifiers (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*`. Recall that such a statement 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 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 `* sizeof(int) + sizeof(int)`, where $c_{o(q)}$ and $c_{o(p)}$ are the integer-valued cells corresponding to the offset 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.

Following the C norm, ASTRÉE assumes the base address `&v` of a variable `v` to be a symbolic, unspecified value, and that the concrete numeric value may change from execution to execution and cannot be relied on. The only guarantee is that all addresses in a live variable `v` are contiguous and distinct from any address of any other live variable. This influences the semantics of pointer comparison operators: one assumes that `(char*)&v+i < (char*)&v+j` if and only if `i<j`, while the truth value of `(char*)&v+i < (char*)&w+j` is unspecified (i.e., it evaluates to $[0, 1]$). Moreover, a local variable may not be given the same address each time it is reallocated when entering its scope, hence, pointers to a local variable are reset to dangling when the variable is destroyed. Additionally, ASTRÉE supports a notion of *absolute pointer* values, where the address of a global variable is a user-specified integer constant, which permits lossless conversions between pointers and integers and is useful in some embedded software contexts (e.g., when variables correspond to memory-mapped hardware registers and are addressed using their known numeric address).

3.9 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.

3.9.1 Intervals

The interval abstract domain [Cousot and Cousot, 1976, 1977] maintains a lower and an upper bound for every integer and floating-point cell. This is one of the simplest domains, 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 [Moore, 1966]. 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 (presented in §2.14) 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 `+` $[0,\beta]$`; }`, 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 cells. ASTRÉE uses a functional map data-structure with sharing to exploit this property. This makes the interval domain scalable to tens of thousands cells.

3.9.2 Congruences

The simple congruence domain [Granger, 1989] over-approximates the value of each integer cell with a set of the form $a\mathbb{Z}+b \triangleq \{\, ak+b \mid k \in \mathbb{Z} \,\}$ where $a \in \mathbb{N}$, $b \in \mathbb{Z}$ (also called "coset"). Congruence properties are extremely useful to express the alignment of pointer offsets expressed in bytes. In C, each type `t` indeed has an *alignment* (often equal to `sizeof(t)` for base types, and the largest alignment of fields for aggregates) that constrains where objects of type `t` lie in memory. On many processors (such as PowerPC), dereferencing a pointer of type `t*` on an address that is not a multiple of the alignment of `t` causes a runtime error. Assuming that the base address of variables respect the alignment constraints of the target platform, it is necessary and sufficient to check that the byte offset of each dereferenced pointer of type `t*` is a multiple of the alignment of `t` to prove the absence of such errors. Additionally, congruence information can refine interval bounds through reduction (§2.15). For instance, $2\mathbb{Z} \cap [1,5] = 2\mathbb{Z} \cap [2,4]$. The congruence domain is a very simple and cheap domain: it is based on straightforward arithmetic notions, such as greatest common divisors and Bézout's identity.

Congruences also appear naturally due to the limited precision of machine integers. Given a $n-$bit word size, unsigned integers range in $[0, 2^n - 1]$ while two's complement signed integers range in $[-2^{n-1}, 2^{n-1}-1]$, and all arithmetic computations are performed modulo 2^n. While ASTRÉE can detect all overflows, i.e., all operations that trigger the wrap-around effect of modular arithmetics, it can also compute the precise modular result and continue the analysis using this value. This is extremely useful to analyze precisely programs using modular arithmetics on purpose for special effect. Consider, for instance, the statement `x = (short) ((unsigned short)` `y + (unsigned short) z)`, where `x`, `y`, `z` are signed short variables. Signed arguments are converted to unsigned so that the addition is performed in the unsigned world, and the result is converted back to a signed number. Although such code triggers systematic overflows when `y` or `z` is negative, the final result (assuming modular arithmetics) is identical to that of a signed, non-overflowing addition. This technique, called "compute-through-overflow," is used by C code generators such as TargetLink [dSpace] for improved efficiency on embedded platforms. Unfortunately, a simple interval analysis will be very imprecise due to the overflows. Assume, for instance, that `y` and `z` lie in $[-1, 0]$. Then, the cast to `unsigned short` gives the set $\{0, 65535\}$ (assuming that `short` is 16-bit long), which is abstracted as the whole range $[0, 65535]$. Thus, an interval analysis finds that `x` is full range $[-32768, 32767]$, while it is in fact in $[-2, 0]$. To solve this issue, we have added to ASTRÉE a domain of *modular intervals* that can express properties of the form $x \in [a, b] + c\mathbb{Z}$. Such properties are obviously invariant by any operation that is invariant modulo $d\mathbb{Z}$ whenever c divides d (for instance, $d = 2^n$ when converting between signed and unsigned $n-$bit integers). Given $\texttt{y}, \texttt{z} \in [-1, 0]$, we can derive the information $(\texttt{unsigned short})\texttt{y}, (\texttt{unsigned short})\texttt{z} \in [-1, 0] + 2^{16}\mathbb{Z}$, then $(\texttt{unsigned short})\texttt{y} + (\texttt{unsigned short})\texttt{z} \in [-2, 0] + 2^{16}\mathbb{Z}$, and finally $\texttt{x} \in [-2, 0] + 2^{16}\mathbb{Z}$. A reduction with the interval information $\texttt{x} \in [-32768, 32767]$ gives $\texttt{x} \in [-2, 0]$, which is the most precise result.

3.9.3 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 floating-point 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 floating-point computations following the IEEE 754–1985 [IEEE Computer Society, 1985] semantics.

Reasoning on floating-point arithmetics is generally difficult because, due to rounding, most mathematical properties of operations

```
float x,y,z;
x = 1.000000019e+38;
y = x + 1.0e21;
z = x - 1.0e21;
r = y - z;
__ASTREE_assert(( r == 2.0e21 ));
```

Figure 3.15: Program triggering an assertion failure due to unintuitive floating-point rounding.

are no longer true, for instance, the associativity and distributivity of $+$ and $\times$. Consider the simple program in Figure 3.15. According to standard algebraic rules, one would expect that $\mathtt{r} = \mathtt{y} - \mathtt{z} = (\mathtt{x} + 10^{21}) - (\mathtt{x} - 10^{21}) = 2 \times 10^{21}$. However, due to rounding, we actually get $\mathtt{y} = \mathtt{z}$, and so, $\mathtt{r} = 0$, which triggers an assertion failure runtime error.

Conceptually, a floating-point operation is defined in two steps: first the exact real result is computed, and then it is rounded either to the floating-point number immediately greater or lower than the exact real result, depending on the current rounding mode (which can be: to nearest, towards $+\infty$, $-\infty$, or 0). Note that rounding is monotonic. A consequence is that interval arithmetics, and so, the interval domain (§3.9.1), are very easy to adapt soundly to floating-point arithmetics: it is sufficient to evaluate interval bounds with floating-point arithmetics, rounding the upper bound towards $+\infty$ and the lower bound towards $-\infty$ (to account for all possible rounding modes). However, other domains (such as octagons in §3.9.4 or filters in §3.10.1) rely on symbolic manipulations of expressions that would not be sound when replacing real operators with floating-point ones. Our solution is to apply a preprocessing step to floating-point expressions and turn them soundly into expressions on reals, which can then be safely manipulated. More precisely, Astrée implements an abstract domain [Miné, 2004a,b] 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. Then, g can be soundly used in place of f, for all arguments in R. In practice, a floating-point expression $f(\vec{\mathtt{x}})$ appearing in the program source is abstracted as a linear expression with interval coefficients $g(\vec{\mathtt{x}}) = [\alpha, \beta] + \sum_i [\alpha_i, \beta_i] \times \mathtt{x}_i$, and R is some condition on the bounds of variables. Note that $+$ and $\times$ in g now denote real additions and multiplications and no longer floating-point ones, and so, g can be fed to abstract domains assuming a real semantics. Interval linear expressions have been selected because they 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 $[1.9999995, 2.0000005]\mathtt{Y} + [0.99999988, 1.0000001]\mathtt{X} + [-1.1754944 \times 10^{-38}, 1.1754944 \times 10^{-38}]$. 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 now scalars and not intervals) but exhibits a larger constant term (i.e., absolute rounding error). In more complex cases, such as multiplying two variables `X*Y`, one variable must be turned into an interval so that one obtains an interval linear form $[a, b] \times \mathtt{X}$ or $[a', b'] \times \mathtt{Y}$. In this case Astrée use specific heuristics to select which sub-expression to abstract as intervals. Note that the linearization is not static (fixed before the analysis starts) but dynamic as it often needs information on variable bounds at the point where the expression is evaluated, and this information is computed on-the-fly by the analysis itself. Finally, because all operations in the expression abstract domain can be broken down to (real or floating-point) interval arithmetics, it can be soundly implemented using floating-point arithmetics with outwards rounding, which guarantees its efficiency. Other, more precise abstractions of floating-point computations exist [Goubault, 2001], but are not used in Astrée; the reason is that they are more costly, while the extra precision is not useful when checking solely for the absence of runtime errors.

An additional complexity in floating-point 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 floating-point type is almost but not exactly a finite subset of $\mathbb{R}$. The presence

```
int x,y;
if (x > y) x = y;
if (y <= 10) {
  // here, x <= 10
}
```

Figure 3.16: Program fragment showing the need for relational properties.

```
void main()
{
    int i;
    int x = 0;
    for (i=0; i<1000; i++)
      if (random()) x++; else x = 0;
}
```

Figure 3.17: Simple loop requiring an octagonal loop invariant.

of a special value is abstracted as a separate boolean flag in the abstraction of each floating-point cell, while 0 actually abstracts both concrete values $+0$ and -0. Special values are propagated according to the IEEE 754 rules [IEEE Computer Society, 1985], which are sometimes subtle (for instance, *NaN* compares unequal to all values, including *NaN* itself) and, additionally, either using or generating special values in or as a result of some arithmetic operation is reported as a runtime error by ASTRÉE (this includes in particular floating-point arithmetic overflows and divisions by zero). Moreover, ASTRÉE checks that an expression does not compute any special value before performing symbolic manipulations defined only over reals (otherwise, such manipulations are disabled locally and only interval arithmetics is used).

3.9.4 Octagons

Given a finite set $\mathtt{V}_n$ of numerical cells, we denote by $Oct(\mathtt{V}_n)$ the subset of linear expressions with unit coefficients and at most two variables: $Oct(\mathtt{V}_n) \triangleq \{\ \pm\mathtt{X} \pm \mathtt{Y} \mid \mathtt{X}, \mathtt{Y} \in \mathtt{V}_n\ \} \cup \{\ \pm\mathtt{X} \mid \mathtt{X} \in \mathtt{V}_n\ \}$. The octagon domain [Miné, 2001, 2004b, 2006a] abstracts a concrete set of points $X \in \wp(\mathtt{V}_n \to \mathbb{R})$ as a finite map $\alpha_{Oct}(X) : Oct(X) \to (\mathbb{R} \cup \{+\infty\})$ defined as $\alpha_{Oct}(X)(e) \triangleq \max_{x \in X} e(x)$, that is, it expresses the tightest set of constraints of the form $\pm\mathtt{X} \pm \mathtt{Y} \leq c$ or $\pm\mathtt{X} \leq c$ (i.e., with c

```
void main()
{
    int x, y, s, d, r;
    y = 0;
    while (1) {
      x = inputX(); // in [-128,128]
      d = inputD(); // in [0,16]
      s = y;
      r = x - s;
      y = x;
      if (r <= -d) y = s - d; else
      if (r >=  d) y = s + d;
    }
}
```

Figure 3.18: Rate limiter.

minimal) that encloses X. Then, $\gamma_{Oct}(S)$ is the set of points that satisfy the conjunction of the constraints of the form $e \leq S(e)$, for every $e \in Oct(\mathtt{V}_n)$. The name *octagon* comes from the shape of $\gamma_{Oct}(S)$ in two dimensions.

The octagon domain is able to express relationships between variables (unlike non-relational domains, such as the interval or congruence domains). It is often necessary to compute, at least locally, on such properties even if we are interested only in variable bounds eventually. Consider, for instance, the program fragment in Figure 3.16 that bounds `x` by `y` before testing whether `y` ≤ 10. Obviously, at line 4, we also have `x` ≤ 10. Although this is an interval property, the interval domain will not be able to infer it. Indeed, it does not track the relationship `x` $\leq$ `y` which is required to deduce `x` ≤ 10 from `y` ≤ 10, while the octagon domain does. Another, more subtle, example is the case of loops, such as that of Figure 3.17 that increments `x` in a loop with index `i`. In order to prove that `x` ≤ 1000 when the loop exits (which is a purely interval property), it is required to find the relational invariant loop `x` $\leq$ `i` first, and combine it with the loop exit condition `i` $= 1000$, which is possible with the octagon domain but not the interval one. Unlike the case of Figure 3.16, the required relation does not appear syntactically in the program. Our last example is the program of Figure 3.18, extracted from an actual use case. It is a synchronous loop that takes as input in `x` a flow of numbers in the interval $[-128, 128]$ and outputs a flow of numbers in `y` which tries to follow the input `x` but is constrained to change (in absolute value) between two iterations no faster than `d` (which is also set dynamically in $[0, 16]$ at each iteration). The variable `s` stores the output value at the last iteration, and `r` stores the actual rate of change. When evaluating either branch bounding `y`, the interval domain cannot maintain the relation between `r`, `x`, `s`, and `d`; hence, the assignment `y = s - d;` (respectively, `y = s + d;`) decrements (respectively, increments) the lower (respectively, upper) bound of `y` by 16 at each iteration. Hence, the interval domain cannot find any finite bound and will warn of spurious overflows. In fact, some simple linear arithmetics can show that `y` stays in the interval $[-128, 128]$. Indeed, the polyhedra domain [Cousot and Halbwachs, 1978], able to represent and manipulate affine expressions, could find this exact result. The octagon domain is less precise: it cannot handle statements involving three variables (such as `r = x - s;`) exactly. Nevertheless, approximate semantic functions can be designed (such as inferring that $\mathtt{r}-\mathtt{x} \leq -\min \mathtt{s}$, which has an octagonal form), so that $|\mathtt{y}| \leq M$ becomes an inductive loop invariant for any $M \geq 144$ (an inductive invariant is an invariant that is true when entering the loop the first time and that, if assumed true at some loop iteration, is sufficient to prove in the abstract domain that it is also true of the next iteration). When combined with a widening with thresholds (§3.9.1), ASTRÉE deduces automatically that $|\mathtt{y}|$ is smaller than the smallest threshold greater than 144. While not as good as the result obtained with a polyhedra analysis, it is sufficient to prove the absence of arithmetic overflow.

The octagon domain is based on a matrix data-structure [Larsen et al., 1997] 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 [Cousot and Halbwachs, 1978] (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 floating-point arithmetics. Moreover, it can abstract integers (including pointer offsets) and floating-point arithmetics (provided these are linearized, as in §3.9.3), and even infer relationships between cells of different type. For instance, the original code that inspired the rate limiter in Figure 3.18 was actually a floating-point computation, for which the octagon domain can prove the absence of runtime error and bound the output |`y`| by approximately 144.00005 (taking rounding errors into account).

3.9.5 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 fragment in Figure 3.19. 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 value 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 and congruence abstract domains. An example decision tree with two boolean variables (B_1 and B_2) and two numeric variables (X and Y) is depicted in Figure 3.20.

```
int b,x,y;
b = (x >= 6);
[...]
if (b) y = 10 / (x - 4);
```

Figure 3.19: Program fragment requiring partitioning.

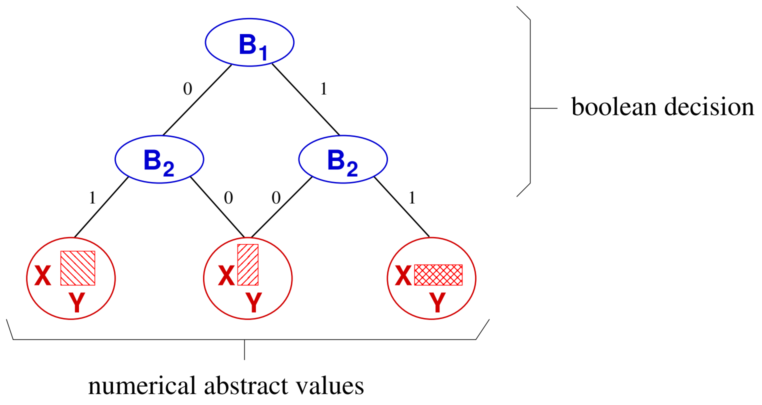


Figure 3.20: Decision tree.

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 example of Figure 3.19, the required decision tree needs simply state that "when `b` is true, `x` is greater than 6."

This abstraction retains some of the properties of binary decision diagrams [Bryant, 1986]: the efficiency is greatly improved by ordering the boolean variables and using sub-tree sharing techniques (for instance, in Figure 3.20, when B_2 is false, then the values of X and Y do not depend upon the value of B_1, hence the numerical abstract leaf value can be shared).

This abstract domain defines a form of partitioning in the sense of §2.16: 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 (§3.6): indeed, the boolean decision trees abstract domain is based on partitions of the set of *current states*, and not traces. This remark has a great impact on the definition of transfer functions: partitions are not affected by control-flow branches (as in trace partitioning); however, when an assignment is made, partitions may need to be modified (which is not the case with trace partitioning). 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).

3.9.6 Packing

Although the octagon domain has a very light, cubic cost compared to many relational domains (such as polyhedra [Cousot and Halbwachs, 1978]), it would still be too high for embedded programs with tens of thousands variables as found in aerospace. 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 where they appear to be interdependent. More precisely, a pack is associated to each syntactic program block, and when two or more variables are used in the same expression, they are put into the pack corresponding to the closest enclosing block. Additionally, variables in expressions at the boundary between two blocks (appearing, e.g., in the test part of "if" statements or as loop indices) are stored in both packs. Then, an octagon is associated to each pack, to relate cells of the variables in this pack, but no relation is kept between cells in distinct packs. Syntactic blocks are only used in the pre-analysis, as boundaries to limit the size of dependency chains and avoid getting large packs but, during the analysis, all the packs are tracked even when outside the scope of the associated block. Note that a cell may appear in several packs, in which case some non-relational interval information can still flow between these packs by reduction (§3.11). The

# lines	# vars.	# packs	size	$\sqrt{\text{size}^2}$	$\sqrt[3]{\text{size}^3}$
370	103	20	4	5.2	6.6
70 000	13 400	2 435	3.6	5.5	7.6
166 000	35 600	3 546	3.9	6.2	8.8
82 000	21 400	139	3.8	3.9	4
290 000	73 300	7 307	3.6	4.6	6
492 000	128 300	12 950	3.4	4.3	5.4
647 000	172 900	17 752	3.3	4.1	5.2
808 800	203 464	23 255	3	3.5	3.9

Figure 3.21: Octagon packing statistics on two families of avionic applications (§3.14). size^n denotes the expectation of the size raised to the $n-$th power.

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). Figure 3.21 gives the number and average size of octagon packs for two families of programs of increasing size (described in more details in §3.14). The quadratic and cubic expectation of pack size give a good idea of the expected (quadratic) memory consumption and (cubic) time cost per pack, which appears to be small and mostly independent from program size. Moreover, in our experiments, cells tend to appear in less than two packs on average.

Boolean decision trees are also packed, but according to slightly more semantic criteria. However, the pre-analysis is still very light, and also results in a time and memory consumption that is experimentally linear in the program size.

From a theoretical point of view, 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 $\mathcal{V}_i$. Each D_i is thus itself obtained by composing a “forget abstraction” with the abstraction into $Oct(\mathcal{V}_i)$.

3.10 Domain-specific numerical abstractions

3.10.1 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 [Feret, 2004] 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 (respectively, i_n) the value of the variable `o` (respectively, `i`) at the n-th loop iteration, it follows that $o_{n+k} = i_{n+k} + \sum_{1 \le j \le k} \alpha_j \cdot o_{n+k-j}$. An appropriate change of variables can be found [Feret, 2005b] by factoring the polynomial $X^k - \sum_{1 \le j \le k} \alpha_j \cdot X^{k-j}$, 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 (§3.9.1), 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 $o_{n+k} = \sum_{0\le j\le l} \beta_j \cdot i_{n+k-j} + \sum_{1\le j\le k} \alpha_j \cdot o_{n+k-j}$ (instead of $o_{n+k} = i_{n+k} + \sum_{1\le j\le k} \alpha_j \cdot o_{n+k-j}$). 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 input 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 contribution of 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 $o'_{n+k} + \sum_{0\le j<N} \delta_j^N \cdot i_{n+k-j}$ where the linear combination $\sum_{0\le j<N} \delta_j^N \cdot i_{n+k-j}$ 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 $o'_{n+k} = \sum_{0\le j\le l} \beta_j \cdot \varepsilon_j^N \cdot i_{n+k-N-j} + \sum_{1\le j\le k} \alpha_j \cdot o'_{n+k-j}$. 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.

Figure 3.22 presents an example implementation of a second order digital filter that uses the last two inputs (`i[0]`, `i[1]`) from a stream (`input_x`) and the last two outputs (`o[0]` and `o[1]`) to compute the next output (`p`), but can also be re-initialized non-deterministically (`input_init`) to match the current input. An example execution trace

```
void main()
{
    static float i[2], o[2];
    while (1)
    {
        float p;
        float x = input_x();
        if (input_init())
        {
            i[0] = x;
            o[0] = x;
            p = x;
        }
        else
            p = (0.5 * x) - (0.7 * i[0]) + (0.4 * i[1]) +
                (1.5 * o[0]) - (0.7 * o[1]);
        i[1] = i[0];
        i[0] = x;
        o[1] = o[0];
        o[0] = p;
        output(p);
    }
}
```

Figure 3.22: Second order digital filter.

(i.e., the successive values `o[1]` as a function of `o[0]`) is presented on the left of Figure 3.23, which shows clearly that such traces are contained in an ellipse (on the right). More importantly, Figure 3.24 shows that the interval domain cannot express any accurate inductive loop invariant, while ellipses can. Indeed, denoting by F the effect (as a geometric transformation) of one iteration of the filter loop and by X a set of environments (as points with coordinate `o[0]`,`o[1]`), then $F(X)$ is a slightly rotated smaller version of X. No box X satisfies $F(X) \subseteq X$, while this is possible for an ellipse Y of adequate parameters, which are automatically discovered by widening in the digital filter domain. An ellipse inductive invariant then implies a box (non-inductive) invariant, i.e., variable bounds. For instance, assuming that the values returned by `input_x` are in the range $[-10, 10]$, Astrée is able to deduce that the output is bounded by $[-14.169716, 14.169716]$ and that there is no arithmetic overflow.

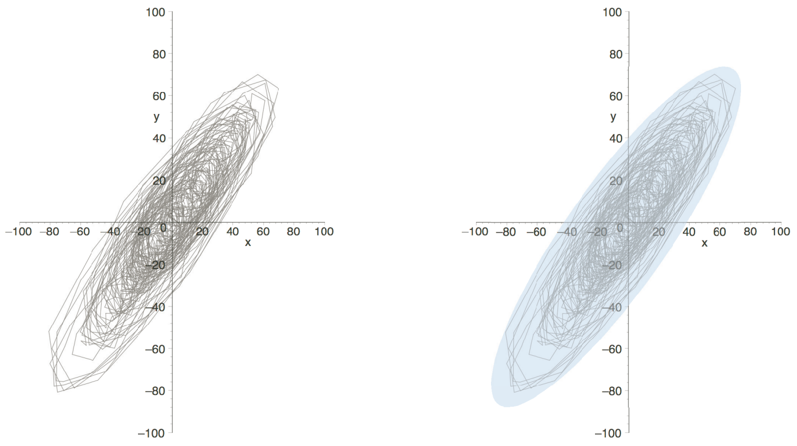


Figure 3.23: Left: example execution trace of a second order digital filter (`o[1]` as a function of `o[0]`), and right: its over-approximation as an ellipsoid.

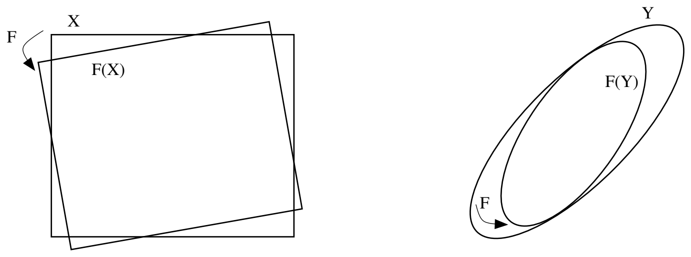


Figure 3.24: Effect of one iteration of the loop in Figure 3.22 on a box and on an ellipse.

3.10.2 Exponential evolution in time

Critical embedded software often contain computations that would be stable if they were computed in real arithmetics, but which may 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, e.g., to implement unit conversion). 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 [Feret, 2005a] 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 corresponds to computations with 32-bit floating-point arithmetics, we get $(1+a)^{\mathtt{t}} \simeq 1.43$, whereas with $a \simeq 10^{-13}$, which corresponds 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.

```
void mult(double dst[4], const double a[4], const double b[4])
{
    dst[0] = a[0]*b[0] - a[1]*b[1] - a[2]*b[2] - a[3]*b[3];
    dst[1] = a[0]*b[1] + a[1]*b[0] + a[2]*b[3] - a[3]*b[2];
    dst[2] = a[0]*b[2] + a[2]*b[0] + a[3]*b[1] - a[1]*b[3];
    dst[3] = a[0]*b[3] + a[3]*b[0] + a[1]*b[2] - a[2]*b[1];
}
```

Figure 3.25: Quaternion multiplication.

3.10.3 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 spatial 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 $\bar{\cdot}$. Quaternions are usually converted to rotation matrices and conversely. An example implementation of quaternion multiplication is presented in Figure 3.25. Obviously, applying many such operations in sequence without care can may result in 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: $||q|| = 1$. This ensures in particular that all quaternion coefficients are smaller than 1 in absolute value, and so, that quaternion operations do not cause overflow. Yet, because of rounding errors and approximated algorithms, their norm may diverge along the execution of the program. Thus, quaternions are often re-normalized (i.e., divided by their norm) so as to avoid overflows. Figure 3.26 presents an example renormalization function, including a protection against division by a too small norm (to avoid division by zero and overflow errors).

```
void normalize(double dst[4], const double a[4])
{
    int i;
    double norm = sqrt(a[0]*a[0] + a[1]*a[1] +
                       a[2]*a[2] + a[3]*a[3]);
    if (norm < 0.001)
    {
        dst[0] = 1.;
        for (i=1; i<4; i++) dst[i] = 0.;
    }
    else
    {
        for (i=0; i<4; i++) dst[i] = a[i] / dst;
    }
}
```

Figure 3.26: Quaternion normalization.

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

$$\begin{array}{rcl} |\,||q_1|| - ||q_2||\,| \leq ||q_1 + q_2|| \leq ||q_1|| + ||q_2|| & & \text{(triangle inequality)} \\ |\,||q_1|| - ||q_2||\,| \leq ||q_1 - q_2|| \leq ||q_1|| + ||q_2|| & & \text{(triangle inequality)} \\ ||\lambda \cdot q|| = |\lambda| \cdot ||q|| & & \text{(positive homogeneity)} \\ ||q_1 \times q_2|| = ||q_1|| \cdot ||q_2|| & & \\ ||\overline{q}|| = ||q|| \ . & & \end{array} \tag{3.1}$$

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 (3.1) 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 of its norm.

3.11 Combination of abstractions

Astrée uses dozens of abstract domains which can interact with one another [Cousot et al., 2006]. These interactions enable Astrée to refine abstract properties, as with a partially reduced product of abstract domains (§2.15), but also to refine their predicate transformers. Special care has to be taken when reduction is used after convergence acceleration (widening or narrowing) steps, in order not to break the construction of inductive invariants: reductions should not break termination of the analysis.

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 convergence acceleration 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 performed; moreover, the use of a cache avoids performing 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 performed 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 performed 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 §2.15). 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 (§3.9.3).

There is no canonical way to linearize 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 interpolation (narrowing) step. Such refinements must be done carefully, since they may break the convergence acceleration process, leading to non-termination of the analysis. Examples of problematic interactions between extrapolation and reduction can be found in [Miné, 2004b, p. 85]. Even worse, alternating the application of a widening operator with a classic join operator during upward iteration, or intersecting at each iteration step the abstract state with a constant element may lead to non-termination. More formally, [Cousot et al., 2006, §7] presents 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:

$$\begin{cases} a_1 = d_1 \\ a_{n+1} = (a_n \mathbin{\nabla} d_{n+1}) \sqcap d_0 \end{cases}$$

$$\begin{cases} b_1 = d_1 \\ b_{n+1} = \rho(b_n \mathbin{\nabla} d_{n+1}) \end{cases}$$

$$\begin{cases} c_1 = d_1 \\ c_{2\cdot n+2} = c_{2\cdot n+1} \cup d_{2\cdot n+2} \\ c_{2\cdot n+1} = c_{2\cdot n} \mathbin{\nabla} d_{2\cdot n+1}, \end{cases}$$

is ultimately stationary.

Yet, reducing the result of a widening is very important, so as to avoid unrecoverable losses 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 componentwise 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).

3.12 Abstract iterator

Formally, the concrete collecting semantics of any program can be expressed as a single fixpoint equation $X = \text{lfp}^{\subseteq} F$, where the function F is derived from the text of the program. The abstract semantics has a similar form $X^\sharp = \text{lfp}^{\subseteq^\sharp} F^\sharp$, 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 (§3.4), $X^\sharp$ 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 prefix 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 $(c :: \ell, X_c^\sharp)$ (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 $X_c^\sharp$ is computed as $Y_c^\sharp$, and the new iterator state is $(c :: \ell', Y_c^\sharp)$. The abstract state $X_c^\sharp$ can then 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 [Monniaux, 2005] 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. In the case of nested loops, a fixpoint for the inner loop is computed for each iteration of the outer loop, i.e., loops nesting is handled in a recursive way. While other techniques exist (such as computing a single global fixpoint), prior work [Bourdoncle, 1993] suggests that the recursive strategy is faster and more precise (although they are not comparable in theory [Cousot and Cousot, 1992b]). Finally, the C language features non-structured control flow transfer statements (i.e., jumps) such as `goto`, `break`, `continue`, and `return`. ASTRÉE is essentially a functional program; its interpreter is designed by structural induction on the syntax tree of the program, which does not support jumps easily. We handle them using *continuations* [Reynolds, 1993], a standard solution to express the semantics of imperative (in particular jump-based) programs in a functional way: additionally to a so-called direct flow of abstract information, the interpreter maintains abstract elements corresponding to pending jump statements that have been encountered but the target of which have not been reached by the syntax tree iterator yet.

The number of abstract components that need to be stored simultaneously at a given point depends only on the number of nested conditionals, loops, and jumps, which is very small (i.e., a dozen) for most programs. This iteration technique is less flexible than general iterations, which can be optimized [Bourdoncle, 1993] while requiring that $X^\sharp$ is fully available in memory; however, it is much more memory-efficient and can scale to very large programs.

3.13 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 points 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 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). In addition, the semantics of erroneous operations (such as integer overflows) can be selected (e.g., whether to use a modular or non-deterministic semantics) to specify how to continue the analysis after reporting an error. 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 the 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 the hypotheses 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 false alarms). 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.

3.14 Application to aeronautic industry

The first application of ASTRÉE (from 2001 up to now) was the proof of absence of runtime errors in two families of industrial embedded avionic control/command software [Delmas and Souyris, 2007]. These are synchronous C programs which have the global form shown in Figure 3.27.

The analysis is an "open loop" in that the relationship between the output variables at one iteration and input variables at the next (i.e., the fact that outputs to actuators have an effect on the environment that in turn effects the sensors) is abstracted away (e.g., by range hypotheses on inputs and requirements on outputs). A closed loop anal-

```
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
```

Figure 3.27: General form of a synchronous program.

ysis 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 to 200 K global variables (half of which have floating-point type, the rest being integers and booleans) and 100 K to 1 M lines of code (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 (§3.10.1, §3.10.2), reductions (§3.11), and strategies to apply costly generic domains only where needed (§3.6, §3.9.6). 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. The initial development of ASTRÉE and parametrization 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 (§3.13) are enough (if needed at all) for the industrial end-users [Delmas and Souyris, 2007] to adapt the analysis to a specific program instance in any supported family, without intervention from the analysis designers.

Figure 3.28 presents benchmarks analyses on various programs of increasing size from two families of embedded avionic control/command software (smaller programs are actually representative fragments, while larger ones are several development revisions of full programs). The analysis was performed on our 64-bit 2.66GHz Intel server using a single core (significant speed-ups can be achieved by using several cores, as reported in [Monniaux, 2005], which is not discussed here). The memory consumption is approximate as Astrée uses caches and a garbage collector, but shows nevertheless that the analyses fit easily on memory sizes that are currently mainstream. The number of alarms is sufficiently low (a dozen or less) so that they can be examined by hand. More importantly, industrial users [Delmas and Souyris, 2007] report zero false alarm (i.e., proof of absence of runtime error) on programs in these families after analysis parametrization.

In addition to these two families of software, Astrée was also used to analyze two other kinds of software: two 45 000 lines self-test programs stressing the hardware before actual programs are run, and a 60 000 lines communication program that formats data from input media to output media [Delmas and Souyris, 2007]. Although neither are control/command, they are nevertheless embedded avionic software written using similar programming guidelines and libraries, and running on the same platforms. Thus, Astrée proved sufficiently precise (7 to 100 alarms) and fast (less than 2h) on these programs.

3.15 Application to space industry

The second application (2006–2008) of Astrée was the analysis of space software [Bouissou et al., 2009]. 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

# lines	time	memory	alarms
370	5s	205 MB	0
70 000	2h 10mn	740 MB	2
166 000	6h 14mn	1.2 GB	10
82 000	41mn	588 MB	2
290 000	7h 2mn	1.2 GB	3
492 000	13h 21mn	2.2 GB	2
647 000	22h 40mn	2.2 GB	13
808 800	50h 13mn	2.7 GB	1

Figure 3.28: Analysis with ASTRÉE of two families of avionic applications.

Vehicle (ATV). The analyzed version differs from the operational one written in Ada in that it is a C program generated from a Scade [Esterel Technologies] 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 §3.14 (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 (§3.10.3). After specialization by the analysis designers, ASTRÉE could analyze the 14 K 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.

3.16 Industrialization

First developed as a research prototype (starting from 2001), ASTRÉE has slowly matured into a tool usable by trained industrial engineers [Delmas and Souyris, 2007]. 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 [Kästner et al., 2010] by [AbsInt, Angewandte Informatik], a company that develops various other (complementary) abstract interpretation-based tools (e.g., to compute guaranteed worst case execution time [Heckmann and Ferdinand, 2004]).

4

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 chapter, programs in the C language were analyzed. In order to study the temporal properties of 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.

4.1 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 the industry. But this may be easily implemented in erroneous way [Ginosar, 2003]. Another risk is to degrade performance. For example, consider the system in Figure 4.1. 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 Figure 4.1. 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 Figure 4.1. 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 in 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.

4.2 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. [2001], 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

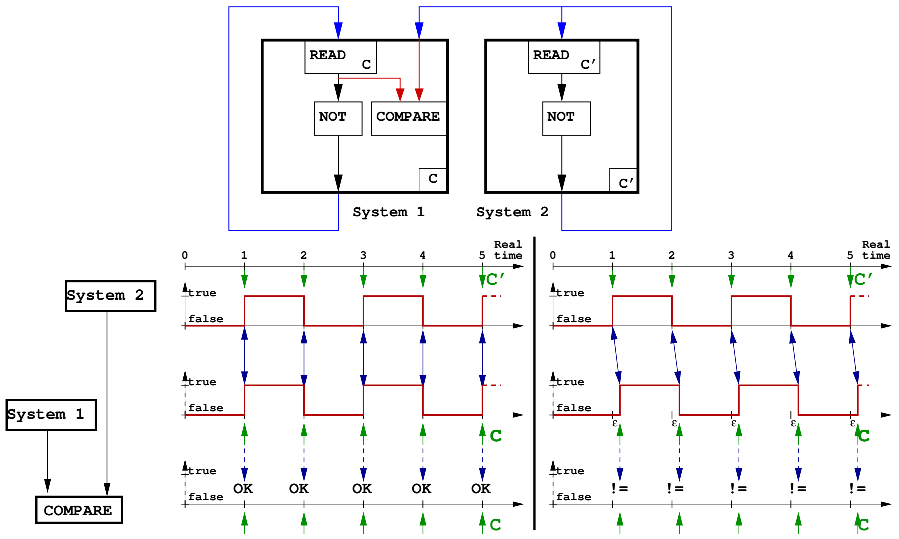


Figure 4.1: Example of two similar imperfectly-synchronous systems with an alarm watching differences in their outputs.

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 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 modeled. It is then assumed that all clocks `C`, `C'`, ... are perfect: $\mu_{\mathtt{C}} = \nu_{\mathtt{C}} = \mu_{\mathtt{C'}} = \nu_{\mathtt{C'}} = \dots$ 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 these 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

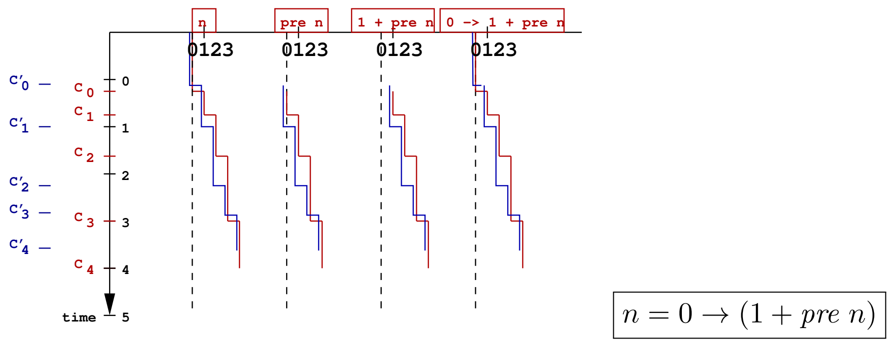


Figure 4.2: Two examples of imperfect clock behaviors.

data that has been received and not yet read. In embedded systems, it is often the case that blackboards are used at the 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 $n = 0 \to (1 + \mathit{pre}\ n), \mathtt{C}_{[0.9,1.1]}$, where $a \to b$ means a at first cycle then b, *pre* c means c but delayed by one cycle, and `C` is a clock of parameter $[0.9, 1.1]$. We depict in Figure 4.2 (respectively, in red and blue) two very different behaviors of this counter n for two imprecise clocks `C` and `C'`.

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 §2.3, this semantics is abstracted as a fixpoint of a concrete operator.

4.3 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 §2.4, 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.

4.4 Temporal abstract domains

4.4.1 Abstract constraints

A first domain of abstract constraints [Bertrane, 2005] abstracts $\wp(\mathbb{R} \to \mathbb{V})$ as conjunctions 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 as $\forall t \in [a, b] : s(t) = x$. 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 as $\exists t \in [a, b] : s(t) = x$. 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 ($\to$, *pre*, blackboard reading, etc.) are quite precise in this domain.

4.4.2 Changes counting domain

A second domain of change counting [Bertrane, 2006] 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 an abstract 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 the 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 $\forall t \in [c, f] : s(t) = x$.

4.5 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 chapter 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 to analyze 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 (Chapter 3) and AstréeA (Chapter 6). 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.

5

Verification of Target Programs

5.1 Verification requirements and compilation

In Chapter 3, 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 [Technical Commission on Aviation, 1999] 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 to the verification of properties which can be defined reliably only at the compiled code level, such as worst case execution time properties [Heckmann and Ferdinand, 2004]. 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.

5.2 Semantics of compilation

When a direct target code level verification is too hard, other approaches can be applied, which allow 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 framework allows defining program transformations at the semantic level [Cousot and Cousot, 2002]. 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 correctness of the transformation.

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 [Rival, 2004]. 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.

5.3 Invariant translation 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* [Necula, 1997]. This techniques generalizes to invariants computed by abstract interpretation-based static analyses of the source code [Rival, 2003].

A disadvantage of such techniques is that not all kinds of invariants can be efficiently checked at assembly level; actually, case studies [Rival, 2003] 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 to be analyzed precisely and part of the structure lost at compile time has to be reconstructed by the invariant checker.

5.4 Compilation verification

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* [Pnueli et al., 1998, Necula, 2000, Rival, 2004]) or only once, for the compiler itself (this technique is usually called *compiler verification* [Leroy, 2006]). In both cases, the compilation is verified correct at some level of abstraction in the sense of §5.2. 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 §5.2, 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 [Pnueli et al., 1998] or special purpose provers [Necula, 2000, Rival, 2004]. These tools consider the compiler as a black box; thus, they apply to a range of compilers. On the other hand, compiler verification [Leroy, 2006] 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.

6

Verification of Parallel Programs

We now briefly discuss on-going work [Miné, 2011] on the verification of the absence of runtime errors in embedded parallel C software running under realtime operating systems, with application to aeronautics.

6.1 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 threads [IEEE and The Open Group]) 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 Chapter 3, in order to design an effective analyzer that eschews decidability and efficiency issues, we will focus on one such model and one application domain.

We will focus on applications for realtime operating systems and, more precisely, ARINC 653 [Aeronautical Radio, Inc. (ARINC)] which is an avionic specification for safety-critical realtime operating systems (OS). An application is composed of a bounded number of processes that communicate implicitly through a shared memory and explicitly through a bounded number of synchronization objects (events, semaphores, message queues, blackboards, etc.) provided by the OS.

The realtime hypothesis imposes that processes and resources are only created during a special initialization phase of the program. We assume that they do not vary from one run of the program to another. The realtime hypothesis also indicates that the scheduling adheres strictly to process priorities (which, we assume, is fixed at process creation time): a lower priority process can only run when all higher priority processes are blocked in system calls. This is unlike the default, non-realtime, POSIX [IEEE and The Open Group] semantics used in most desktop computers and most servers, where any unblocked process, whatever its priority, eventually gets to run, possibly preempting higher priority processes.

We also assume that all the processes are scheduled on a *single* processor, i.e., only one process can run at a given time. The real-time and single-processor hypotheses are neither gratuitous nor added to simplify the analysis, but are actually relied on for the correctness of our target applications; thus, the analyzer must take them into account to construct a proof of absence of runtime errors (see the next section for an example where these hypotheses are necessary).

Finally, we assume the same restrictions as in Chapter 3: programs are in a subset of C without `longjmp` nor recursive procedures. In addition to runtime errors, we wish to report data-races (i.e., concurrent accesses to the same variable by two different processes, which are not protected by a synchronization primitive, and one of which is a write), but do not consider other parallel-related threats (such as deadlocks, livelocks, starvation).

6.2 Program example

Figure 6.1 presents a simple program with two processes, `low` and `high`, with respective priority 0 and 1 (higher values denoting higher priorities), and a mutual exclusion lock (or *mutex*).

The `main` entry point creates these resources and stores the mutex identifier into the variable `l`. When `main` returns, both processes are started concurrently. They share a global variable `x` that denotes abstractly some kind of resource that can be consumed by `low` (`x--`) and created by `high` (`x++`). In order for the lower priority process `low` to be able to run, it is necessary for the higher priority process `high` to voluntarily relinquish the control, which is done with the `yield` instruction. Note that `yield` does not specify when the control returns back to the `high` process: its concrete semantics consists in waiting for a non-deterministic amount of time and then preempting the `low` process at any point of its execution, prompting the analysis to consider a large number of preemption points and process interleavings to cover all cases.

Accesses to `x` are protected by the mutex `l` in *critical sections*, so that the other process cannot change the value of `x` between a test (`x <= 10` or `x > 0`) and the subsequent update (`x++` and `x--`). The `low` process uses a standard `lock`/`unlock` bracketing pair to protect its access to `x`. We could have done the same for `high`, but chose instead to present an alternate, lock-less protection mechanism. Indeed, because `high` has higher priority, and only a single process can run at a given time, it is sufficient to test whether the mutex is locked (without locking it) and, if it is not, `high` can modify `x`, confident that `low` cannot preempt `high`, lock `l`, and start modifying `x` before `high` has finished modifying `x` and called `yield`.

Note that this reasoning is only possible if the program runs under a real-time, mono-processor OS.

This programming pattern is used in our main target applications (§6.5), and so, must be precisely handled by our analysis to avoid spurious alarms.

```
int l; // mutex identifier
int x; // shared variable
void low() {
    while (1) {
        mutex_lock(l);
        if (x > 0) x--;
        mutex_unlock(l);
    }
}
void high() {
    while (1) {
        if (!mutex_is_locked(l)) {
            if (x <= 10) x++;
        }
        yield();
    }
}
void main() {
    process_create(low,  0);
    process_create(high, 1);
    l = mutex_create();
    x = 5;
}
```

Figure 6.1: Parallel program creating two processes, `low` and `high`, that modify a shared variable `x` and synchronize using a mutex `l`.

6.3 Concrete collecting semantics

An execution of the program is an interleaving of instructions from each process obeying some scheduling rules (§6.3.2). 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 synchronization 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.

```
                   init: flag1 = flag2 = 0
          process 1:              |          process 2:
  flag1 = 1;                      |  flag2 = 1;
  if (!flag2)                     |  if (!flag1)
  {                               |  {
      /* critical section */      |      /* critical section */
  }                               |  }
```

Figure 6.2: Mutual exclusion algorithm inspired by Dekker.

6.3.1 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 at 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* [Saraswat et al., 2007]. The most natural one, *sequential consistency* [Lamport, 1979], 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 heavyweight measures from the compiler (e.g., preventing most optimizations, introducing fences) with a huge negative impact on performance. Following Java [Gosling et al., 2005], the latest C and C++ standards [ISO/IEC JTC1/SC22/WG21 Working Group, 2010] only require that perfectly synchronized programs (i.e., where all accesses to shared variables are protected by synchronization primitives, such as mutexes) behave in a sequentially consistent way (e.g., because calls to synchronization primitives are recognized and treated specially by the compiler, disabling optimization and generating fences). In earlier (and still commonly used) C standards, unprotected accesses are implicitly understood as causing undefined behaviors while, in Java, a weak memory consistency model [Manson et al., 2005] specifies a set of possible valid behaviors. Consider, for instance, the incorrectly synchronized program (inspired from Dekker's mutual exclusion algorithm [Dijkstra, 1968]) from Figure 6.2.

In the Java memory model, both processes can enter their critical section simultaneously. The rationale is that, due to process-level program optimization 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, local buffers, or not fully synchronized caches can also cause similar behaviors, even in the absence of compiler optimization.

In order to define our concrete semantics, we had to choose a memory model that is consistent with newer standards but also currently used programming practices that often predate these standards. Firstly, we assume that the semantics guarantees sequential consistency for programs without data-races. Secondly, our analysis reports data-races as errors. Finally, we also give a proper semantics to unprotected accesses, so that we can analyze the behavior of a program after a data-race (this is consistent with our choice to continue the analysis with the modular behavior after reporting an integer overflow). More precisely, assume that a run of a process p performs a sequence of synchronization operations at times $t_1 < \ldots < t_n$, and a run of another process p' performs two synchronization 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 [Miné, 2011, Ferrara, 2008, 2009] 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 on the optimizations used by the compiler and the hardware consistency model enforced by the processor(s) [Miné, 2011].

6.3.2 Scheduling and synchronization

The U state component in our concrete semantics models the scheduler state, which in turn defines which process can run and which must wait. Firstly, it maintains the state of synchronization 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 a higher priority 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 point.

The execution of a synchronization 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 be transferred 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. This is also formalized in [Miné, 2011].

6.4 Abstractions

Our prototype analyzer for parallel embedded realtime software, originally named Thésée [Miné, 2011] and now rebranded AstréeA, is based on Astrée (Chapter 3). 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.

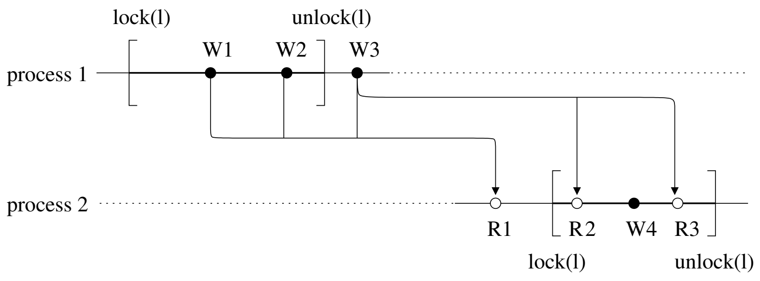
6.4.1 Control and scheduler state abstraction

Astrée was fully flow- and context-sensitive thanks to a partitioning with respect to the control state C (§3.4). Full flow- and context-sensitivity for a parallel program, i.e., partitioning with respect to $G = U \times C_1 \times \cdots C_n$ (i.e., the scheduler state and control states for all processes) is not practical as there are generally too many reachable control points in G.

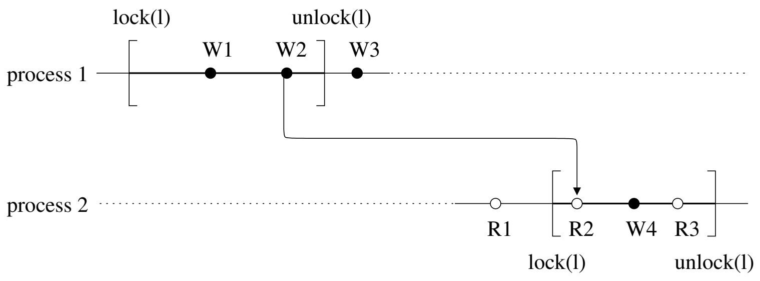
To solve this problem, we first make use of an abstraction [Cousot and Cousot, 1984], derived from proof techniques for parallel programs [Lamport, 1977, Owicki and Gries, 1976, Jones, 1981], that consists in decomposing a global property partitioned on G into several local properties, one for each process. 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 $U_i \times C_i$.

6.4.2 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 (§3.7). Following the concrete semantics of weakly consistent shared memory (§6.3.1), $(U_i \times C_i) \to \wp(D)$ is abstracted as a disjunction. Firstly, we track for each $(u, c) \in U_i \times C_i$ 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 cor-



(a) Incorrectly synchronized interferences.



(b) Correctly synchronized interferences.

Figure 6.3: Actions of the writes from one process (1) on the reads of another process (2).

rectly synchronized accesses). This abstraction of write actions is both flow-insensitive and non-relational (e.g., using the interval and the congruence domains).

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 $j \neq i$. As such write actions are empty for global variables that are not actually shared, these are abstracted as precisely as local ones (i.e., in a flow-sensitive and relational way). The soundness with respect to the weakly consistent memory model of §6.3.1 is ensured because the order of write actions has been abstracted away.

Not all write actions from j influence a read in i. First, we consider write actions associated to scheduler states in j not mutually exclusive with that of process i (e.g., not locking the same mutex). This is illustrated in Figure 6.3.(a): writes W1 and W2 by process 1 while the lock l is held do not influence reads R2 and R3 by process 2 because these reads are also protected by the lock l. However, this is not sufficient. There is indeed a flow of information between two critical sections protected by the same mutex: the last write before unlocking l in process 1 (i.e., W2) can be seen by the reads in process 2 until it overwrites the variable while holding mutex l. Hence, W2 influences R2, but not R3, as seen in Figure 6.3.(b) (the fact that W2 influences R1 is already taken into account in Figure 6.3.(a)). Distinguishing these cases is made possible because write actions are partitioned with respect to (an abstraction of) the scheduler state. It makes the analysis partially aware of synchronization processes (including synchronization through priority, as in Figure 6.1), which is key to avoid spurious interferences and achieve a sufficient level of precision to eliminate false alarms. Other interesting properties, such as the presence of data-races (i.e., incorrectly synchronized interferences), or simply the set of shared variables and which processes use them, can be extracted easily from the computed write actions.

A limitation of this abstraction, though, is that it cannot discover relational invariants holding at critical section boundaries, as the value of correctly synchronized variables is abstracted in a non-relational way. This has been addressed in later work [Miné, 2014]. Another possible improvement would be to perform a trace abstraction (§3.6) of sequences of scheduler states to analyze precisely behaviors that depend on the history of interleavings of process execution (e.g., initialization sequences run only once in a specific order).

6.4.3 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 §3.12 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 (§6.5).

6.4.4 Operating system modeling

AstréeA 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 AstréeA 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 AstréeA identifiers).

This model is approximately 2 500 lines long. Although it is written in C, the model is not executable as it uses many AstréeA-specific primitives (e.g., mutex locking). As details of the implementation of parallelism are forgotten, the model embeds some abstraction. The analysis is then sound in that it computes (an over-approximation of) all the behaviors of the program when run under any OS respecting the ARINC 653 specification.

6.5 Preliminary application to aeronautic industry

Our main application targets a family of industrial avionic software. We have analyzed one instance consisting of 15 processes and 1.6 M code lines. It runs on a single *partition* (ARINC 653 can run several such partitions on the same processor but on separate time frames and memory spaces). Intra-partition inter-process communication is precisely modeled by computing an abstraction of the shared memory as well as all the messages sent. 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 runtime error, independently from (and without requiring hypotheses on) the behavior of other (unanalyzed) applications and the initial contents of the non-volatile memory. Our target software is rather complex and heterogeneous. Some processes have a synchronous control/command flavor similar to software targeted by Astrée (§3.14), while others focus on string formatting, list manipulation and sorting, or network protocols (such as TFTP), among other tasks.

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.

We then turned to the analysis of the full 1.6 M line software. The analysis currently takes 30h of computation time [Miné, 2014]. An important experimental result is that the number of iterations required to compute write action sets is quite low (up to 6), which validates our choice of control abstraction for parallel programs (§6.4.2). Moreover, the number of scheduler states for write actions is low (around 50) so that the analysis stays efficient in memory: the analysis fits in 32 GB of memory (due to the use of caches and a garbage collector, it is difficult to assess more precisely the memory consumption). The analysis outputs around 1 100 alarms. This high number of alarms is expected as AstréeA inherits most of its abstract domains from Astrée, which was specialized for synchronous control/command software, while our application is not purely control/command. New abstract domains need to be developed to account for algorithms and data-structures (such as lists, strings, and network message) used in the analyzed software, and these are not tied to the use of parallelism but rather to an application domain. This is ongoing work, and the figures we now provide are already an improvement over earlier published results of 12 000 alarms in 50h in [Bertrane et al., 2010], and 7 600 alarms later in [Miné, 2011].

7 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 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, at least for specific classes of specifications. 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 [Cousot et al.] took 8 years before being available as an industrial product [AbsInt, Angewandte Informatik]. This neither includes decades of research on abstract interpretation nor 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 [Technical Commission on Aviation, 1999]. Work is on-going on the 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. Although the state of the art in formally certified sound C static analyzers is not yet on par with the precision and efficiency of analyzers such as ASTRÉE, promising recent work [Jourdan et al.] shows that it is possible to reason about abstract interpretation using proof assistants and derive mechanically checked sound tools.

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, the semantics of which 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 [Martel, 2009]).

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

AbsInt, Angewandte Informatik. ASTRÉE run-time error analyzer. `http://www.absint.com/astree/`.

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

J. Bertrane. Static analysis by abstract interpretation of the quasi-synchronous composition of synchronous programs. In R. Cousot, editor, *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 97–112. Springer (Berlin), 2005.

J. Bertrane. Proving the properties of communicating imperfectly-clocked synchronous systems. In Kwangkeun Yi, editor, *Proc. of the 13th Int. Static Analysis Symposium (SAS'06)*, volume 4134 of *LNCS*, pages 370–386. Springer (Berlin), 2006.

J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace (I@A 2010)*, number AIAA-2010-3385, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), 2010.

F. Besson, D. Cachera, T.P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, pages 223–257. Springer (Berlin), 2009.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer (Berlin), 2002.

B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press (New York), 2003.

O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, E. Goubault, K. Ghorbal, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conference, Data Systems In Aerospace (DASIA'09)*, pages 1–7. ESA publications, 2009.

F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.

F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–14. Springer (Berlin), 1993.

R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

R. M. Burstall. Program proving as hand simulation with a little induction. In *Proc. of IFIP Congress*, pages 308–312, 1974.

P. Caspi, C. Mazuet, and N. Reynaud Paligot. About the design of distributed control systems: The quasi-synchronous approach. In Udo Voges, editor, *20th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2001)*, volume 2187 of *LNCS*, pages 215–226. Springer (Berlin), 2001.

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

P. Cousot. Semantic foundations of program analysis, invited chapter. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, 1981.

P. Cousot. Types as abstract interpretations. In *Conf. Rec. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'97)*, pages 316–331. ACM Press (New York), 1997.

P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the Second Int. Symp. on Programming (ISOP'76)*, pages 106–130. Dunod, Paris, 1976.

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

P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979a.

P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. of the 6th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79)*, pages 269–282. ACM Press (New York), 1979b.

P. Cousot and R. Cousot. *Invariance proof methods and analysis techniques for parallel programs*, chapter 12, pages 243–271. Macmillan, 1984.

P. Cousot and R. Cousot. Sometime = always + recursion ≡ always: on the equivalence of the intermittent and invariant assertions methods for proving inevitability properties of programs. *Acta Informatica*, 24:1–31, 1987.

P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992a.

P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'92)*, volume 631 of *LNCS*, pages 269–295. Springer (Berlin), 1992b.

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

P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conf. Rec. of the 29th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'02)*, pages 178–190. ACM Press (New York), 2002.

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

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

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyser. In M. Sagiv, editor, *Proc. of the 14th European Symposium on Programming Languages and Systems (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30. Springer (Berlin), 2005.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In M. Okada and I. Satoh, editors, *Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer (Berlin), 2006.

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée. In M. Hinchey, He Jifeng, and J. Sanders, editors, *Proc. of the First Symp. on Theoretical Aspects of Software Engineering (TASE'07)*, pages 3–17. IEEE Computer Society Press, 2007a.

P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In G. Filé and H. Riis-Nielson, editors, *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, volume 4634 of *LNCS*, pages 333–348. Springer (Berlin), 2007b.

P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In M. Hofmann, editor, *14th Int. Conf. on Fondations of Software Science and Computation Structures (FoSSaCS 2011)*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer-Verlag, 2011.

D. Delmas and J. Souyris. Astrée: from research to industry. In G. Filé and H. Riis-Nielson, editors, *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, volume 4634 of *LNCS*, pages 437–451. Springer (Berlin), 2007.

E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

dSpace. TargetLink code generator. `http://www.dspaceinc.com`.

Esterel Technologies. Scade suite™, the standard for the development of safety-critical embedded software in the avionics industry. `http://www.esterel-technologies.com/`.

Euclid of Alexandria. Elementa geometriæ, book xii, proposition 17. fl. 300 BC.

J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer (Berlin), 2004.

J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 42–58. Springer (Berlin), 2005a.

J. Feret. Numerical abstract domains for digital filters. In *Proc. of the First Int. Workshop on Numerical & Symbolic Abstract Domains (NSAD'05)*, 2005b.

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

P. Ferrara. *Static analysis via abstract interpretation of multithreaded programs.* PhD thesis, École Polytechnique, Palaiseau, France, May 2009.

R. W. Floyd. Assigning meanings to programs. In *Proc. of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–32. Springer Netherlands, 1967.

R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the Association for Computing Machinery*, 47(2):361–416, 2000.

R. Ginosar. Fourteen ways to fool your synchronizer. In *Proc. of the 9th Int. Symposium on Asynchronous Circuits and Systems (ASYNC'03)*, pages 89–97. IEEE Computer Society, 2003.

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

É. Goubault. Static analyses of floating-point operations. In *Proc. of the 8th Int. Static Analysis Symposium (SAS'01)*, volume 2126 of *LNCS*, pages 234–259. Springer (Berlin), 2001.

P. Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3 & 4):165–190, 1989.

P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of the Int. Joint Conf. on Theory and Practice of Software Development (TAP-SOFT'91), Volume 1 (CAAP'91)*, volume 493 of *LNCS*, pages 169–192. Springer (Berlin), 1991.

R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. In *Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 26–30. IEEE Computer Society, 2004.

IEEE and The Open Group. Portable operating system interface (POSIX). `http://www.opengroup.org`, `http://standards.ieee.org`.

IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std. 745-1985, 1985.

ISO/IEC JTC1/SC22/WG14 Working Group. C standard. Technical Report 1124, ISO & IEC, 2007.

ISO/IEC JTC1/SC22/WG21 Working Group. Working draft, standard for programming language C++. Technical Report 3035, ISO & IEC, 2010.

B. Jeannet and A. Miné. The `Apron` numerical abstract domain library. `http://apron.cri.ensmp.fr/library/`, 2007.

B. Jeannet and A. Miné. `Apron`: A library of numerical abstract domains for static analysis. In *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV'09)*, volume 5643 of *LNCS*, pages 661–667. Springer (Berlin), 2009.

C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, Jun. 1981.

J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *Conf. Rec. of the 42nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2015).*

D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. ASTRÉE: Proving the absence of rutime errors. In *Proc. of Embedded Real-Time Software and Systems (ERTS'10)*, pages 1–5, 2010.

L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28:690–691, Sept. 1979.

K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proc. of the 18th IEEE Real-Time Systems Symp. (RTSS'97)*, pages 14–24. IEEE CS Press, 1997.

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

J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Conf. Rec. of the 32nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'05)*, pages 378–391. ACM Press (New York), 2005.

M. Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Formal Methods in System Design*, 35 (3):265–278, Dec. 2009.

L. Mauborgne. ASTRÉE: Verification of absence of run-time error. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 385–392. Kluwer Academic Publishers, Dordrecht, 2004.

L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proc. of the 14th European Symp. on Programming Languages and Systems (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20. Springer (Berlin), 2005.

A. Miné. The octagon abstract domain. In *Proc. of the Analysis, Slicing and Transformation Workshop (AST'01)*, pages 310–319. IEEE Computer Society Press (Los Alamitos), 2001.

A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer (Berlin), 2004a.

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

A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006a.

A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN-SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM Press (New York), 2006b.

A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. of the 20th European Symposium on Programming (ESOP'11)*, volume 6602 of *LNCS*, pages 398–418. Springer, 2011.

A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *Proc. of the 15th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*, volume 8318 of *Lecture Notes in Computer Science (LNCS)*, pages 39–58. Springer, 2014.

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

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

G. C. Necula. Proof-Carrying Code. In *Conf. Rec. of the 24th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Langauges (POPL'97)*, pages 106–119. ACM Press (New York), 1997.

G. C. Necula. Translation Validation for an Optimizing Compiler. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI'00)*, pages 83–94. ACM Press (New York), 2000.

S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, Dec. 1976.

A. Pnueli, O. Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In *Proc. of the 25th Int. Coll. on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 235–246. Springer (Berlin), 1998.

J. C. Reynolds. The discoveries of continuations. *Lisp and Sy,bolic Computation*, 6(3–4):233–248, 1993.

X. Rival. Abstract Interpretation-based Certification of Assembly Code. In *Proc. of the 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, volume 2575 of *LNCS*, pages 41–55. Springer (Berlin), 2003.

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

X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages Systems*, 29(5), 2007.

V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'07)*, pages 161–172. ACM Press (New York), 2007.

A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

Radio Technical Commission on Aviation. DO-178B. Technical report, Software Considerations in Airborne Systems and Equipment Certification, 1999.