Combination of Abstractions in the ASTRÉE Static Analyzer[⋆]

Patrick Cousot [2], Radhia Cousot [1], Jérôme Feret [2], Laurent Mauborgne [2], Antoine Miné [2], David Monniaux [1,2] & Xavier Rival [2]

[1] Centre National de la Recherche Scientifique (CNRS)
[2] École Normale Supérieure, Paris, France (*Firstname.Lastname*@ens.fr)

http://www.astree.ens.fr/

Abstract. We describe the structure of the abstract domains in the ASTRÉE static analyzer, their modular organization into a hierarchical network, their cooperation to over-approximate the conjunction/reduced product of different abstractions and to ensure termination using collaborative widenings and narrowings. This separation of the abstraction into a combination of cooperative abstract domains makes ASTRÉE extensible, an essential feature to cope with false alarms and ultimately provide sound formal verification of the absence of runtime errors in very large software.

1 Introduction

ASTRÉE is a static program analyzer based on abstract interpretation [1,2] which is aimed at proving automatically the absence of run time errors in programs written in a subset of the C programming language. It has been applied successfully to large embedded control/command safety-critical real-time software generated automatically from synchronous specifications, producing correctness proofs for complex software without any false alarm, within only a few hours of computation on personal computers [3,4,5,6]. More recently [7], it has been extended to handle other kinds of embedded software, some of which are hand-written.

ASTRÉE was designed using:

- a syntax-directed representation of the program control flow (functions, block structures);
- functional representation of abstract environments with sharing [3], for memory and time efficiency, and limited support for analysis parallelization [8];
- basic abstract domains, tracking variables independently (integer and floating-point intervals [9] using staged widenings);
- relational abstract domains tracking dependencies between variables
  - symbolic computation and linearization of expressions [10],
  - packed octagons [11],

⋆ This work was supported in part by the French exploratory project ASTRÉE of the Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL).

  - application-aware domains (such as the ellipsoid abstract domain for digital filters [12] or the arithmetic-geometric progression abstract domain [13], e.g. to bound potentially diverging computations);
- abstract domains tracking dependencies between boolean variables and other variables (boolean partitioning domain [4]), or the history of control flow branches and values along the execution trace (trace partitioning [14]);
- a memory abstract domain [4] recently extended to cope with unions and pointer arithmetics [7].

Contrary to many program analysis systems, ASTRÉE does not have separate phases for pointer/aliasing analysis and arithmetic analysis.

To adjust the cost/precision ratio of the analysis, some of the abstract domains are parametrized (e.g. maximal height of decision trees) and applied locally (e.g. to variables packs [4]) according to local directives automatically inserted by the analyzer[3].

The abstract domains communicate as an approximate reduced product [15] to organize the cooperation between abstract domains and allow for a modular design and refinement of the abstraction used by ASTRÉE. In this paper we describe how abstract domains are organized and do cooperate.

This modular design allows abstract domains to be turned on and off by runtime options, easy addition of new domains, and the suppression of older domains that have been superseded by newer ones (such as the clock domain [3], now superseded by the arithmetic-geometric progression abstract domain [5]). Finally, it allows the addition of new reductions / communications between existing domains. ASTRÉE is therefore an extensible abstract interpreter, an essential feature to cope with false alarms and ultimately reach zero false alarm.

ASTRÉE is programmed mostly in OCaml [16] (apart from the octagon domain library [17] and some platform-specific dependencies, e.g. to control the rounding behavior of the FPU). It is currently approximately 80 000 lines long.

2 Handling False Alarms

As all abstract interpretation-based static program analyzers, ASTRÉE may be subject to *false alarms*; that is, it may report potential bugs that happen in no possible concrete execution, because of the over-approximation of program behaviors entailed by abstractions. Thus, when ASTRÉE raises an alarm, it may be a true alarm, due to a runtime error appearing at least in one program execution, but it may also be a false alarm due to excessive over-approximation.

This is the case of all automatic sound formal methods which, because of undecidability and in absence of human interaction, must be incomplete, and hence, in many cases, either exhaust time or space resources or terminate with false alarms.

[3] These directives can also be inserted manually, but such intervention of end-users must be avoided, in particular for programs subject to long-term modifications.

2.1 Different Classes of Alarms

We distinguish between three classes of alarms:

1. Conditions that necessarily terminate the execution in the concrete world. Such is the case, for instance, of floating-point exceptions (invalid operations, overflows, etc.) if traps are activated, and also integer divisions by zero. We issue a warning and consider that the incorrect execution has stopped at the point of the error. The analyzer will continue by taking into account only the executions that did not trigger the run-time error.
2. Conditions that are defined to be incorrect with respect to the C specification or user requirements, but that do not terminate the execution and for which it is possible to supply a sound semantics for the outcome. For instance, overflows over signed integers will simply result in some signed integer. We issue a warning, but do not consider that the executions meeting the warning condition have stopped. The user may examine each such warning and determine if the condition signaled is really harmful (for instance, the user may decide to ignore some integer arithmetic overflows). If it is not, the user may safely ignore the warning.
3. Conditions that are defined to be incorrect with respect to the C specification, that may or may not terminate the execution when they are encountered, but for which it is next to impossible to provide a sound semantics for the remainder of the execution. They are handled by the analyzer as the first kind of alarms. The rest of the section is devoted to this third category as it deserves some explanation.

Some operations, such as pointer arithmetics across memory blocks or memory accesses out of bounds, are considered "undefined behaviors" or "implementation defined behaviors" by the specification of the C programming language [18]. They often result in no immediate runtime crash; but may result in e.g. memory corruptions, with consequences such as erratic behaviors or crashes much later.

For such conditions, Astrée considers that execution stops with an error when the first undefined behavior occurs (and signals an alarm at this point). Its operational semantics thus coincides exactly with actual program executions only if there is no (false or true) alarm of the third kind.

If such alarms are raised, particularly those related to memory safety, then the analysis will not flag all possible runtime errors, i.e. not those arising from traces that have done some "undefined" memory or pointer manipulation. In this event, it is insufficient to analyze these warnings and show that the "undefined" behavior is actually defined in a harmless way for platform-specific reasons (as one would do for the second class of alarms). Rather, one has to either reach *zero* alarm of the third class, or prove by other means that their preconditions are not met in the concrete.

Our experience shows that industrial programmers often use constructs that are nonstandard with respect to the C specification [18, 6.3.2.3], but have well-defined behaviors on the target platform, such as converting a 32-bit pointer into an integer, and then back into a pointer. Thus, we have tried to reduce the third category of warnings as much as possible and, in agreement with our end-users, defined a more precise yet platform- and even application domain-specific semantics that turns most of them into either correct statements, or warnings of the second class. In several instances, it required us to adapt our concrete semantics and develop specific abstractions (e.g. the memory abstraction of [7] to cope with some situations where pointers are manipulated using integer arithmetics).

2.2 Causes of False Alarms

There are several possible causes of false alarms:

– The abstract transformers are not the best possible, in which case the algorithm can be improved in the corresponding abstract domain, if this improvement is not algorithmically too expensive.

  Example 1. Consider the following program:

  ```
  y=x; z=sqrt(x*y+3);
  ```

  Starting from `x` $\in [-3,7]$, a simple interval analysis will derive `y` $\in [-3,7]$, then $x.y \in [-21,49]$, $x.y+3 \in [-18,52]$ and we flag a false alarm on `sqrt`: square root of a negative number. However, we can solve this issue with a minor alteration: when computing the interval for a product $x.y$, we issue a request to the reduced product (see Sect. 6.3) and ask whether x and y are provably equal; if they are, compute the interval for x^2, that is, $[0,49]$. We thus improve the precision at a very minor cost.
  A more general approach would be to compute polynomials or other expressions symbolically and extract minimal and maximal values depending on the range of their variables, but this would be more complex and more costly, and we have not found a need for this so far. □

– The automated parametrization (e.g. variable packing) fails to guess that some relation is important and, in order to save time, artificially limits a relational abstract domain to an inappropriate level of precision. In this case one must improve or adapt the pattern-matched program schemata.

  Example 2. Our first heuristics for reducing the cost of relational domains was to relate together only variables that appear simultaneously in an assignment or test. It prevents proving that `x` ≤ 21 at the end of the following program:

  ```
  x=10; for (i=0;i<=10;i++) x++;
  ```

  despite the ability of the octagon domain to infer the necessary inductive invariant `x` $-$ `i` $= 10$, simply because no octagon will hold both `x` and `i`. The problem was solved by considering octagon packs relating variables that act likely as counters (i.e. are incremented or decremented within the same loop). □

– The widening in the fixpoint approximation iteration strategy overshoots the most imprecise invariant allowing the proof of absence of runtime errors, in which case we must revise the widening. This can be very hard since at the limit only a precise infinite iteration might be able to compute the proper abstract invariant. In that case, it might be better to design a more refined abstract domain.
– The choice of a precise abstract transformer is not always the best. Indeed, our goal is to find a precise post-fixpoint: it can happen that a more relaxed abstract transformer helps the extrapolation process. In short, it is better to jump straight up to the limit, rather than try to be precise at each iteration, then fail to converge quickly and have to resort to interval widening techniques, which will in the end yield a poorer result.
  When considering arithmetic-geometric progressions [13], choosing the most precise abstract transformer is not appropriate at all: it would give no more information than the interval domain.
– The current combination of abstract domains is inexpressive i.e. indispensable local inductive invariants are not expressible in the abstract. In that case a new abstract domain must be added to the reduced product (e.g. filters, arithmetic-geometric progressions).

When a new abstract domain is introduced, a communication and reduction process is used so that the other abstract domains can benefit from the information computed by the new one, as described in Sect. 5.2. This may, but should not, have effects on the enforcement of convergence by widening as discussed in Sect. 7. The modular integration of new abstract domains allows coping with variations between the various families of software successfully analyzed by ASTRÉE.

3 General Structure of Astrée

When ASTRÉE was designed, we knew that the first simple attempt with interval analysis could not be sufficient to achieve a precise analysis on the industrial software we were given. From the beginning, we had in mind the process of refinement which consists in finding the origin of false alarms and improving the information generated by the analysis by a modular extension. It is the reason why we developed ASTRÉE in a modular way, as permitted by the abstract interpretation theory.

ASTRÉE can be roughly decomposed into 4 parts:

1. a front-end, very similar to that of a compiler,
2. simple independent analyses,
3. an invariant computation mixing many interdependent analyses,
4. invariant checking and alarm reporting.

The first phase is pretty standard and did not change much during the evolution of ASTRÉE. An intermediate code is produced, typed and annotated, and then simple program transformations are applied, such as constant propagation.

The transformations we implemented aim at reducing the complexity of the subsequent analyses. One important aspect is the elimination of useless variables (e.g. after constant propagation) as the size of the invariants depends directly on the number of variables.

The second phase consists in simple independent analyses, producing information useful for the subsequent invariant computation, such as variable dependencies. It implements automatic parametrization strategies, such as the octagon packing strategy [11] or the trace partitioning strategy [14].

The third phase is the most important and also the most demanding. It consists in an iterator which follows the control flow of the program and gives orders (abstract transfer directives) to modules representing information about the program. Each of these modules is what we call an abstract domain, and each of them collects some specialized information about the iteration sequence leading to the invariants of the program ASTRÉE analyzes. Such abstract domains can deal with the trace approximation [14], the shape of the data structures and memory [7], or the numerical values occurring during the program execution. The way these abstract domains are designed independently and then interact to produce precise information about the program invariant is crucial to achieve a fast and precise abstract interpreter.

4 Abstract Domains

4.1 Interfaces, Properties, and Abstractions

An abstract domain collects properties about the potential computations of a program. In ASTRÉE, the abstract interpreter follows the control flow of the program; thus, our abstract domain collects some properties about the computations of the program reaching the current program point.

The design of our abstract domains fits with [2], that is:

– We abstract sets of execution traces, not mere sets of reachable states.
– An abstract domain is not necessarily a lattice, and may not even need a preorder.
– We do not define a Galois connection, but only a concretization function; that is, concrete properties may lack a most precise abstraction.
– Abstract transformers are not necessarily monotonic with respect to the information preorder induced by the concretization; that is, it may happen that a more precise abstract precondition is transformed into a less precise abstract postcondition.

The elements of the concrete domain D are sets of *trace fragments*.[4] A trace fragment is a sequence of one or more pairs (p, s) where p is a program point and s is a memory state. All our abstractions will take the following view of a set of execution traces $(p_1, s_1), \ldots, (p_n, s_n)$:

[4] In fact, it consists in sets of trace fragments collected for the *direct flow* (normal program executions) and the pending branching flows (`break`, `continue`, forward `goto`). In this paper, we shall ignore the latter for the sake of simplicity.

– only final states so that p_n is the current program point of the analysis are considered;
– the final memory state s_n is abstracted quite precisely;
– the strict prefix $(p_1, s_1), \ldots, (p_{n-1}, s_{n-1})$ is abstracted more coarsely; the trace partitioning domain keeps a sequence of program points p_k of interest (e.g. those related to if-then-else or loop branches) as well as the value of a few selected variables from s_k at given program points.

An abstract domain is a set $D^\sharp$ of abstract properties of trace fragments. Each abstract property $a \in D^\sharp$ is related to the set of concrete trace fragments that satisfy this property through a concretization function $\gamma_{D^\sharp} : D^\sharp \to D$. We consider different kinds of information:

– Some abstract properties define the mapping between structured C variables and the abstract scalar variables manipulated by most abstract domains. In particular, it handles the case where overlapping sequences of bytes are manipulated as scalar variables of possibly different types (e.g. through union types or pointer casts) and frees the other domains from the burden of coping with byte-level aliases and considering the binary memory representation of variables. This structural abstraction [7] is fully dynamic because, in our model of concrete executions, the pattern of data accesses is a run-time property that is not restricted by static typing.
– Some abstract properties constrain abstract variables: they may be non relational properties (such as a range for each variable) or relational properties (such as restricted linear relations, as in octagons; restricted polynomial relations, as in ellipsoids; restricted non-polynomial relations, as in arithmetic-geometric progressions).
– Some abstract properties may be guarded by constraints about some variables (as in boolean partitioning or by properties on the computation traces that have led to the current state (as in trace partitioning).

We do not assume that an abstract domain has a lattice structure. However, we suppose that it is provided with primitives to simulate the computation of the concrete semantics at the abstract level. This way, for any concrete n-ary primitive $\mathrm{F} : D^n \to D$, we have a sound abstraction $\mathrm{F}_{D^\sharp} : (D^\sharp)^n \to D^\sharp$ that satisfies: for any abstract properties $a_i \in D^\sharp$, $\mathrm{F}((\gamma_{D^\sharp}(a_i))_{1\leq i\leq n}) \subseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}((a_i)_{1\leq i\leq n}))$. However, we do not assume $\mathrm{F}_{D^\sharp}$ to be the most precise transformer that satisfy this property. These primitives not only update memory states, but also the information about the computation paths that lead to these memory states.

To ensure the termination of our analysis, the abstract domain is provided with extrapolation operators: the bottom element $\bot \in D^\sharp$ is the basis of abstract iterations, the widening operator $\nabla_{D^\sharp} \in D^\sharp \times D^\sharp \to D^\sharp$ is used to speed up the iterates (it may discard some information), and the narrowing operator $\triangle_{D^\sharp} \in D^\sharp \times D^\sharp \to D^\sharp$ is used to refine the iterates (after an imprecise extrapolation). We require no property whatsoever about the bottom element $\bot$. The widening operator (resp. the narrowing operator) is a sound abstraction of the union set operator (resp. the meet set operator): this way, for any pair $(a, b) \in D^\sharp \times D^\sharp$ of abstract properties, we require that $\gamma_{D^\sharp}(a) \cup \gamma_{D^\sharp}(b) \subseteq \gamma_{D^\sharp}(a \nabla_{D^\sharp} b)$ and $\gamma_{D^\sharp}(a) \cap$

$\gamma_{D^\sharp}(b) \subseteq \gamma_{D^\sharp}(a \triangle_{D^\sharp} b)$. Moreover, both widening and narrowing operators ensure the convergence of iterates, which means that for any sequence $(x_n) \in (D^\sharp)^{\mathbb{N}}$, the sequence $(x_n^\triangledown)$ (resp. $(x_n^\triangle)$) defined as $x_0^\triangledown = x_0$ (resp. $x_0^\triangle = x_0$) and $x_{n+1}^\triangledown = x_n^\triangledown \triangledown_{D^\sharp} x_{n+1}$ (resp. $x_{n+1}^\triangle = x_n^\triangle \triangle_{D^\sharp} x_{n+1}$) is ultimately stationary. More details are given about the usage of the widening in Sect. 7 and about the usage of the narrowing in Sect. 8.

Although we do not require abstract domains to be provided with an abstract order, there always exist a so-called information preorder $\sqsubseteq^\sharp$ induced by the concretization function: $a \sqsubseteq^\sharp b \iff \gamma_{D^\sharp}(a) \subseteq \gamma_{D^\sharp}(b)$. This preorder has little use in a practical analyzer as it is often computationally expensive and sometimes not even computable. Moreover, although all abstract transfer functions are abstractions of monotonic concrete transfer functions, they are often not monotonic with respect to the information preorder.

– The first cause of non monotonicity is nested loops. Internal loops may be analyzed using widening operators, and the abstraction of the least fixpoint obtained is in general not monotonic with respect to loop precondition.

  Example 3. Consider an interval analysis of the following program, using the standard widening [9]:

```
x=0;
while(random()) {
  x=x+1;
  if (random()) x=y
  if (x==10) x=0; }
```

  If we know at the beginning that $\mathtt{y} \in [0, 9]$, then we immediately obtain the invariant $\mathtt{x} \in [0, 9]$. Suppose however that we know the more precise property $\mathtt{y} = 0$, then our analysis gives $\mathtt{x} \in [0, +\infty[$. Thus, a more precise precondition yields a far worse outcome. □

– Another cause is transfer functions making use of additional information that, in fact, produce a less precise result.

  Example 4. Such is for instance the case of the interval domain [9] helped by the symbolic computation domain [10]. Consider the following example, depending on whether we use the rewrite rule $\mathtt{j} \mapsto \mathtt{i} + 1$ arising from the assignment `j=i+1`:

Code	Symbolic computation	Less precise symbolic
`int i, j=i+1;`	j ↦ i+1	NOTHING
`int k=j+1;`	j ↦ i+1, k ↦ j+1 ↦ i+2	k ↦ j+1
`if (j > 0) {`		
`  l=k;`	j ↦ i+1, k ↦ i+2, l ↦ i+2	k ↦ j+1, l ↦ j+1
`}`		

  By default, our symbolic computation domain performs all possible rewrites, thus we try to reduce the intervals using $\mathtt{k} \mapsto \mathtt{i} + 2$, which yields no additional precision. However, with the less precise third column, we have $\mathtt{k} \mapsto \mathtt{j} + 1$, and since we have the interval information $\mathtt{j} \in [1, +\infty[$ we conclude that $\mathtt{k} \in [2, +\infty[$. □

– Finally, some abstract domains are implemented using floating-point as over-approximations of an "ideal" abstract domain and this may introduce non-monotonicity.

  Example 5. The octagon abstract domain uses a propagation scheme based on an incremental Floyd–Warshall shortest-path-closure algorithm to infer and refine constraints. On reals or rationals, this propagation is both sound and complete; in particular, the outcome does not depend on the order of the variables. On floating-point numbers, soundness can be achieved easily by rounding all computations towards $+\infty$ (as only upper bounds are manipulated). However, the propagation is no longer complete and different variable orderings give incomparable sound approximations of the most precise result. Starting from the same precondition, but two different internal encodings, and applying the same transfer function, we can obtain slightly different postconditions, hence the non-monotonicity. □

4.2 Comparison with Predicate Abstraction

Constraint messages (Sect. 5) and abstract properties are, essentially, *predicates* over the set of traces that we abstract. However, our analysis is not what is usually referred to as *predicate abstraction* [19].

Predicate abstraction generally refers to the following approach:

– one considers a (small) finite, given, set S of predicates; each predicate $p \in S$ has a semantics $[\![p]\!]$ in terms of possible program or variable states (thus, the predicate $\mathtt{x} < 5$ will include all program states where variable $\mathtt{x}$ is less than 5); in the simplest case, predicates simply reflect the value of the boolean variables in the program;
– one computes abstract states as subsets S' of S, such that $[\![S']\!] = \bigcap_{p \in S'} [\![p]\!]$;
– transformers over these abstract states may be defined using an automatic theorem prover;
– once transformers are defined, the program is reduced to a boolean program and a model checker is used;
– if one cannot prove the desired property, and a fake "counterexample" is obtained the analysis is insufficiently precise; additional predicates have to be added, often generated through a process of automatic refinement based on the examination of the fake counterexample.

Differences with predicate abstraction are as follows:

– Our analyses do not consider a priori a small set of predicates, but rather operate on *parametric* predicates [20]. That is, where predicate abstraction considers different predicates $\mathtt{x} < 4$, $\mathtt{x} < 5$, etc., our analysis considers a generic predicate $\mathtt{x} < C$ and tries to adjust C.

- We do not use an automatic theorem prover to generate the program transformers. We could perhaps do so, provided that the theorem prover is capable of handling efficiently our parametric predicates; obviously, it is more difficult to generate transformers over parametric predicates, perhaps with some measure of optimality of the result of the transformer, than to decide or even semi-decide whether a particular ground predicate ensues from a program construct in the context of some particular ground predicates. However, such automated generation of parametric transfer functions is itself a research issue.
- We do not use automatic refinement techniques in the sense of adding new predicates and starting the analysis again. However, if our analysis fails to find an invariant, then we extrapolate the invariant "candidates" through *widening* techniques.

4.3 Domain Constructors

Some of the abstract domains used in ASTRÉE are based on similar algebraic constructs or are parametrized by the choice of an underlying abstract domain. In order to factor code and allow easy parametrization, we defined domain *constructors* [21] that are naturally implemented as OCaml functors [16], while abstract domains are OCaml modules.

Non-Relational Lifting Functor. Some abstract domains used in ASTRÉE are *non-relational*; that is, they abstract the values of each scalar abstract variable separately. For instance, we have the following abstractions for scalar values:

- integer intervals with thresholds;
- floating-point intervals with thresholds;
- integer congruences.

We *lift* these abstractions to non-relational domains on multiple variables by considering abstract environments mapping each variable to an abstract value. As described in [3], environments are implemented using balanced binary trees, which allows a fast abstract union operator in $O(m \log n)$, where n is the total number of variables and m the number of variables that differ in the two environment arguments, instead of $O(n)$ for a plain array. This pays off in the kind of code we analyze, where the number of if-then-else as well as the number of variables are linear in the size $|P|$ of the program, but then and else branches have a small size. We obtain a combined cost for the meet operation at the end of all if-then-else in $O(|P| \log |P|)$ instead of $O(|P|^2)$. The same optimization is used for other binary operators, e.g. widening.

Packed Relational Lifting Functor. A similar system is used for relational domains (such as the octagon abstract domain: the set of abstract variables is partitioned[5] into *packs* of bounded size. All the variables in a pack are related

[5] Actually we consider a covering where one variable may appear in several packs. This is useful when a single variable is used in different contexts. However, to maintain an almost linear cost, no information flows between packs sharing variables.

together by an instance of the relational domain, but not with variables in other packs. Each transfer function only modifies a small set of packs, while abstract unions operate point-wisely on packs. The lifting of a standard relational domain to a packed domain is similar to the non-relational lifting. The resulting domain enjoys the same almost-linear asymptotic cost, assuming a bound on the size of the packs.

Trace Partitioning. Most abstract domains deal with scalar values and data structures, thus, memory states of the program. However, it is sometimes necessary to distinguish between values according to the history of the computation.

Example 6. Consider the following implementation of a piecewise linear function:

```
if (x < tx[0] || x > tx[N]) fail();
for (i=0; i<N-1; i++)
  if (x <= tx[i+1]) break;
return ty[i]+(ty[i+1]-ty[i])*(x-tx[i])/(tx[i+1]-tx[i]);
```

where `tx` and `ty` are constant arrays of size `N+1`, and `tx` is increasing. The plain interval domain would show a warning for division by zero, since it will compute the least upper bound of all `tx[i+1]` for all values of `i`, the same for `tx[i]`, and the two would overlap. Precise analysis thus seems to require inferring complex relationships between `i`, `tx[i]`, and `ty[i]` and handling affine functions.

However, the interval domain can find the most precise result provided that we partition the last assignment with respect to the number of iterations before exiting the loop. This is semantically equivalent to analyzing the following code:

```
if (x < tx[0] || x > tx[N])
   fail();
if (x < tx[1])
   return ty[0]+(ty[1]-ty[0])*(x-tx[0])/(tx[1]-tx[0]);
else if (x < tx[2])
   return ty[1]+(ty[2]-ty[1])*(x-tx[1])/(tx[2]-tx[1]);
else ...
else
   return ty[N-1]+(ty[N]-ty[N-1])*(x-tx[N-1])/(tx[N]-tx[N-1]);
```

□

Trace partitioning [14] is a functor parametrized by two abstractions: an abstraction of the history of former memory and control states (e.g. a sub-sequence of branches taken), and an abstraction of the current memory state. Abstract elements are maps and are implemented as trees: each path corresponds to a different control history and each leaf contains the corresponding memory state. This makes it easy to dynamically adjust the precision of history abstractions by simply splitting leaves and folding sub-trees, which is exactly what Astrée does, driven by heuristics that achieve a trade-off between cost and precision.

Boolean Partitioning. An alternate way of partitioning is to distinguish between the possible values of a subset of the variables with respect to the value of one or more boolean variables.

Example 7. Consider the following code that stores an arithmetic condition in a boolean for future use[6]:

```
b = x < 5;
/* unrelated computations */
if (b) x = 5;
```

In order to prove that `x` ≥ 5 at the end, one should distinguish between the case where `b` is true and where it is false, at least for the information concerning `x`. □

Partitioned abstract elements are implemented as decision diagrams [4, 2.6.4], i.e. trees with boolean variables at internal nodes and abstract memory states at the leaves, with opportunistic sharing of equivalent sub-trees. Thus, the boolean partitioning domain it is a functor parametrized by the choice of an abstraction of memory states. As for relational domains, almost-linear cost is achieved by only partitioning small, bounded sets of arithmetic variables with respect to small, bounded sets of booleans. Thus, it also reuses the packing functor.

Abstract Product. In general, the abstract domain used by the analyzer is formed of the partially reduced product of several abstract domains. Reduction is implemented through a network of communication channels, as explained in the following section. We use a binary product functor that takes two domains and implements communications between them. It returns a new domain that can be used as argument of any functor, including the product functor itself. Thus, a full network of domains can be constructed using several product applications. As the binary product multiplexes communication channels, every domain in the resulting network can communicate with every other one.

5 Network of Domains

5.1 Hierarchies

ASTRÉE handles very heterogeneous kinds of abstract properties. Each class of abstract properties is gathered inside a small, independent abstract domain. Domains are fitted with all the primitives needed to handle their particular class of abstract properties. Nevertheless, ASTRÉE is not a neutral product of separate abstract domains (which would be equivalent to running separate analyses); it organizes an active collaboration between them.

[6] This kind of code, where the definition and the use of `b` are far apart, appears frequently in automatically generated programs (e.g. compiled from graphical languages *à la* Simulink).

We use the binary product functor to gather several abstract domains together into a hierarchy and form a reduced product. The product is not commutative because it sets which abstract domain will be processed before the others. As a consequence, the domains that are computed first may communicate partial results to others that have not yet started their own computations. When a domain $D_1^\sharp$ is computed before domain $D_2^\sharp$, we say that $D_1^\sharp$ is an *underlying* domain of the domain $D_2^\sharp$. By construction, being an underlying domain is an acyclic relation (see Fig. 1).

The argument of a unary functor, such as the trace or boolean partitioning domain, may also be an abstract domain constructed by applying the binary product functor. As a consequence, there are generally several hierarchies at work in an analysis. Each unary functor spawning a hierarchy will be called a *root.* Roots have a special role in the communication between domains, as we will see in Sects. 5.3–5.4.

Example 8. Fig. 1 is a small but realistic hierarchy used in Astrée. Its main root is the trace partitioning domain. The boolean partitioning domain is used to spawn a simpler sub-hierarchy. In the example, variable ranges can be partitioned with respect to the value of some boolean variables, while octagon invariants cannot. Also note that the interval domain is the most underlying domain in both hierarchy, hence the first evaluated. On the one hand, it is the least precise, and so, the one most likely to benefit from refinement by other domains. On the other hand, it is the only one to handle all C constructs, and so, provides a base information we can always resort to. □

5.2 Communication Channels

Domains communicate abstract properties to each other. For that purpose, we introduce a particular abstract domain of messages. This domain is defined as a set $IO^\sharp$ of abstract properties and by a concretization $\gamma_{IO^\sharp} : IO^\sharp \to D$ that maps any such abstract property to the set of concrete trace fragments that satisfy this property.

Each regular abstract domain $(D^\sharp, \gamma_{D^\sharp})$ is fitted with a primitive $\textsc{extract}_{D^\sharp} : D^\sharp \times IO^\sharp \to IO^\sharp$ that it can use to emit some constraints on communicating channels. In the expression $io' = \textsc{extract}_{D^\sharp}(c, io)$, the message io denotes the contents of the channel before the constraint is emitted, the abstract element c denotes an abstract element in the domain $D^\sharp$, and the message io' denotes the contents of the channel enriched with constraints extracted from c (hence the name $\textsc{extract}$). This way, we require that $\gamma_{D^\sharp}(c) \cap \gamma_{IO^\sharp}(io) \subseteq \gamma_{IO^\sharp}(\textsc{extract}_{D^\sharp}(c, io))$.

Conversely, each abstract domain $(D^\sharp, \gamma_{D^\sharp})$ has a primitive $\textsc{refine}_{D^\sharp} : D^\sharp \times IO^\sharp \to D^\sharp$ allowing the reception of constraints from a communication channel. In the expression $c' = \textsc{refine}_{D^\sharp}(c, io)$, the abstract element c' is a refinement (hence the name $\textsc{refine}$) of the abstract element c having taken into account the constraints denoted by the contents io of the channel. We require that $\gamma_{D^\sharp}(c) \cap \gamma_{IO^\sharp}(io) \subseteq \gamma_{D^\sharp}(\textsc{refine}_{D^\sharp}(c, io))$.

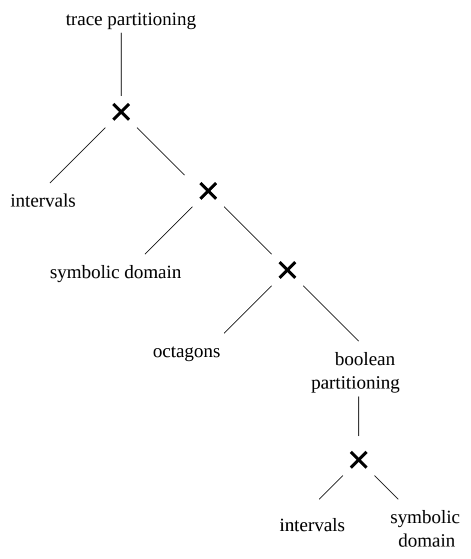

Fig. 1. Example hierarchy.

In the following subsections, we introduce some communicating channels between domains. We distinguish between two kinds of communications:

1. a domain may ask whether a more precise constraint is available; the channel used is then called an *input* channel (Sect. 5.3);
2. a domain may decide to communicate some of its constraints to other domains in order to refine them, in which case the channel used is called an *output* channel (Sect. 5.4).

Note that the abstract domain $IO^\sharp$ of messages does not need to be the same for input and output channels. In ASTRÉE, they are indeed different and implemented using different data-structures (a product type for the input, and a sum type for output). This means that we actually have two versions of $\textsc{refine}_{D^\sharp}$ and $\textsc{extract}_{D^\sharp}$.

5.3 Input Channels

Input channels provide information on both the postcondition being computed and the precondition computed in the last computation step. A domain $D^\sharp$ may update the contents of the postcondition channel at the end of its own computation (using $\textsc{extract}_{D^\sharp}$). It may read the pre-condition information from all domains, but may only access post-condition constraints that have already been computed by another domain (triggered before itself in the hierarchy). At the end of a computation step, the network root collects the contents of the channel and makes it available to all domains as the precondition of the next step.

The contents of the channel is implemented as a *functional record type*. Each field denotes a particular class of properties in $IO^\sharp$. For each field, there is a default value $\top$ which corresponds to the absence of information (when no domain has filled the field yet). To avoid useless computations, we rely on lazy evaluation: each field is a function that is evaluated only if/when required. To update a closure f, a domain replaces it with a new closure f'. When applied, the new closure f' may or may not evaluate f. Moreover, we use memoization to avoid computing the same information several times.

The advantage of this design is that adding a new kind of input communication between two domains is straightforward. First, we add a field in the signature of the channel and we update the default value of the channel contents. Then, we modify the primitive $\textsc{extract}_{D^\sharp}$ of the domain $D^\sharp$ that provides this information. Last, we update the primitives that use this information. The code for the other domains that do not generate nor use this information does not require any modification.

5.4 Output Channels

Output channels are used when a domain wants to send a message to others. There are two output channels:

- The first one is used to refine the computation just performed by the underlying domains; we call it the *oriented output channel*.
- The second one broadcasts a message to be used by all domains (including those that have not performed their computation yet); we call it the *broadcast output channel*.

A domain $D^\sharp$ may send messages (using $\textsc{extract}_{D^\sharp}$) and fetch messages to use them (using $\textsc{refine}_{D^\sharp}$). In the case of oriented outputs, the contents of the channel is simply handed from one domain to the next by the product functor so that it can be directly used, refined, or both; then, the (possibly updated) contents is forwarded to all the underlying domains. In the case of broadcast outputs, the channel is only updated during the network evaluation; no domain may use its contents. Then, once the root of the network is reached, the contents of the channel is sent to all domains using $\textsc{refine}_{D^\sharp}$ primitives, so that domains have the opportunity to use (and even refine further) the information.

The contents of an output channel is implemented as a list of constraints. Constraints are implemented with a *sum type*, where each summand is a different kind of constraints. After each computation, each domain collects a list of constraints from each output channel. The primitive $\textsc{refine}_{D^\sharp}$ scans the list and refines the abstract properties accordingly. It may also generate new constraints to be communicated to other domains but, to avoid infinite loops, we only allow $\textsc{refine}_{D^\sharp}$ to use the oriented output channel, not the broadcast one.

From the point of view of analyzer maintenance, this design is very convenient since it makes the addition of a new kind of output communication between two domains easy. First, we add a summand in the signature of the output constraints. Then, we modify the domain that outputs the constraints. Last, we update the primitive $\text{EXTRACT}_{D^\sharp}$ in the domain $D^\sharp$ that wishes to receive the constraint. The other domains do not require any modification: they will simply ignore the new information, which is safe.

6 Domain Cooperation

Abstract computations are made under assumptions about pre/postconditions. Indeed, any n-ary concrete transfer function $\mathrm{F} \in D^n \to D$ is simulated by an abstraction $\mathrm{F}_{D^\sharp} \in (D^\sharp \times IO^\sharp)^n \times IO^\sharp \to D^\sharp$. The abstract element $a_0 = \mathrm{F}_{D^\sharp}((a_i, io_i)_{1\leq i\leq n}, io_0)$ should be understood as: compute in the abstract the image of F, knowing that each argument a_i satisfies the constraints in io_i, and that the result a_0 satisfies the constraints in io_0. This gives the following soundness criterion: if there exists some $(c_i)_{0\leq i\leq n} \in D^n$ such that

- $c_i \in \gamma_{D^\sharp}(a_i)$, for any i such that $1 \leq i \leq n$,
- $c_i \in \gamma_{IO^\sharp}(io_i)$, for any i such that $0 \leq i \leq n$,
- $c_0 \in \mathrm{F}((c_i)_{1\leq i\leq n})$.

then we must have $c_0 \in \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}((a_i, io_i)_{1\leq i\leq n}, io_0))$.

We now distinguish between several cases of collaboration. Some domains may be used to refine the abstract properties of other domains. This kind of reduction boils down to replacing an abstract operation $F_{D^\sharp}((a_i)_{1\leq i\leq n}))$ with a refined counterpart $\rho_0(F_{D^\sharp}((\rho_i(a_i, io_i))_{1\leq i\leq n}), io_0)$ such that, for any integer i such that $0 \leq i \leq n$, ρ_i is a sound abstraction of the meet: $(\gamma_{D^\sharp} \circ \rho_i)(a_i, io_i) \supseteq \gamma_{D^\sharp}(a_i) \cap \gamma_{IO^\sharp}(io_i)$. Whenever $i > 0$, the reduction $\rho_i : D^\sharp \times IO^\sharp \to D^\sharp$ is used to refine the precondition: this kind of refinement is discussed in Sect. 6.1. The reduction $\rho_0 : D^\sharp \times IO^\sharp \to D^\sharp$ is used to refine the postcondition: this kind of refinement is discussed in Sect. 6.2.

In ASTRÉE, this is not the only way the domains may collaborate. We also perform some refinement of abstract transformers that cannot be expressed as merely refining abstract states. This is discussed in Sect. 6.3.

6.1 Precondition Refinement

Some domains refine abstract states before they are fed to their abstract transformers. This is made possible thanks to the input channel as it makes the information that has been computed by all domains accessible to any domain. Since the abstract interpretation of the program follows the control flow graph, the abstract computation of the properties that are valid at a given iteration and just before interpreting an instruction are fully computed before the abstract interpretation of the instruction begins.

This kind of reductions is used whenever the domain is a partial mapping from some tuples of variables to parametric constraints (as in the filters domain or the arithmetic-geometric progressions domain). In such domains, the support (i.e. the set of tuples that are mapped to a constraint) changes during the iteration. Whenever both arguments of a binary operator do not have the same support or whenever a unary abstract transformer needs a given constraint to be precise, the domain uses the input channel to synthesize missing constraints.

Example 9. The ellipsoid domain can simulate an assignment of the form $X = a.Y + b.Z + t$ by mapping a constraint of the form $Y^2 - a.Y.Z - b.Z^2 \leq k^2$ to a constraint of the form $X^2 - a.X.Y - b.Y^2 \leq (f(k))^2$. When this constraint is missing, the ellipsoid domain synthesizes an ellipse using interval constraints about the variables Y and Z, and a possible equality relation between Y and Z. □

6.2 Postcondition Refinement

Domains may collaborate to refine the result of an abstract transformer. There are two cases: the refinement is initiated either by the domain that has computed the information, or by the domain that misses the information.

A first use is when a domain synthesizes a very useful information and propagates it to its underlying domains using the oriented output channel[7]. For instance, many domains can infer interval information and use the oriented output channel to inform the interval domain of their discoveries.

Example 10. Consider the following code fragment computing an absolute value:

```
X=Y;
if (X<0) X=-Y;
if (X<100) { ...Y... }
```

The interval domain does not track the relationship between `X` and `Y` and so cannot prove that, in the last then block, `Y` $\in [-100, 100]$. However, the octagon domain can (see [11, §5.2]). Suppose now that the interval domain is underlying with respect to the octagon one, thus allowing the later to refine the result of the former through an oriented output channel (this is always the case in practice). After each assignment or test, the octagon domain gathers the variables that had their value updated (for tests, this includes all the variables appearing in the expression; for assignments, only the left-value is considered) and extracts their range in all octagons by projection. Each range is then compared to the one computed by the underlying domains (using the input channel on the postcondition). When the one computed from the octagon domain is more precise, it is output to the oriented output channel. When it is not, no interval constraint is output. This may happen because some modified variables do not appear in any octagon, or the expression has been so aggressively abstracted (due to floating-point arithmetics [22] or non-linearity [10]) that the interval domain performs better. □

[7] Although a domain may use the broadcast output channel to propagate information to underlying domains, this has not been used in ASTRÉE until now. The broadcast channel has a more specific use, illustrated in Ex. 12.

A second use is when a domain cannot synthesize a precise constraint, and so, asks for other domains to generate it. A first way to receive constraints from underlying domains is to use the input channel.

Example 11. 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 that case, the octagon domain relies on the underlying interval domain to give it the actual range of `X` in the postcondition. To compute the effect of the assignment, the domain first forgets all constraints about `X` (are they may no longer hold in the postcondition), and then adds range constraints gathered from the input channel for the postcondition. □

Alternatively, a domain may use the broadcast output channel to request some information about some variables. The root of the network will propagate this message to all domains. Each domain will then try and compute some constraints over the given variables and communicate them using the oriented output channel. This is more powerful because all domains participate in the refinement, but also more costly.

It is important to note that we cannot rely solely on precise domains to initiate communications towards less precise ones. Suppose, for instance, that we wish an unstable constraint to be refined by a precise and stable constraint after some widening application. Because of our systematic optimization of binary operations (which is key to the scalability of ASTRÉE, see Sect. 4.3), stable constraints are not even looked at during widening application. As a consequence, the precise and stable constraint has no opportunity to initiate the communication and it is up to the unstable one to ask for help.

Example 12. The widening of the interval domain checks for the stability of variable ranges. When it detects that some variable range is not stable, it enlarges it. It also sends a broadcast message to the root of the network and informs it of the precision loss. As a consequence, each domain will collects all the information it has on the variable and try to infer a range information. When successful, this process issues a “refine range” message that is acknowledged by the interval domain. □

6.3 Abstract Transformer Refinement

Some refinements cannot be expressed as state refinements: domains actually collaborate to set up their abstract transformers.

First, some domains require expressions to be presented with a given level of abstraction. In such cases, a domain may ask its underlying domains to abstract an expression. This kind of communication is ensured by the input channel.

Example 13. Most abstract domains are purely numerical abstract domains that abstract sets of points in a vector space $\mathbb{R}^n$. They require expressions to be arithmetic expressions over some finite set of scalar variables. However, C expressions also allow structured variables (such as arrays or structures) as well as pointers.

A specific domain [7] is devoted to abstracting the memory into a set of independent scalar cells. It is its responsibility to evaluate array and structure accesses, pointer dereferences, and translate C expressions into arithmetic ones.

In order to do this, it abstracts information on the layout of the memory, as well as information on the memory blocks pointed to by pointers. However, it relies on the underlying numerical domains to abstract the contents of integer and floating-point variables, as well as pointer offsets (viewed as integers). For instance, it may ask the underlying domain for the value of some array index in the current precondition. The case of complex expressions with several levels of indirections is solved by structural induction and spawns many communications.

Another example is pointer arithmetics, which is simply translated into integer arithmetics on offsets, to be evaluated by underlying numerical domains. It is important to note that pointer and value analyses are performed at the same time and that expressions are transformed on-the-fly given the abstraction of the precondition currently available in the network of domains. □

Example 14. Most relational domains are based on real arithmetic, because it enjoys convenient properties (e.g. associativity, distributivity). However, concrete programs use floating-point operations that violate those. Thus, we use a linearization domain [22] to soundly translate floating-point arithmetics into real arithmetics. This may decrease the precision because perfectly deterministic but highly non-linear rounding errors are abstracted into non deterministic intervals. But, it outputs simple linear forms with interval coefficients and real semantics that can be fed directly to numerical domains, even relational ones.[8] □

Note that some domains require precise information that is lost by linearization or array access resolution. Thus, each domain should be allowed to interpret expressions at the level of abstraction it chooses (possibly in a dynamic way).

Example 15. The floating-point interval domain bounds tightly each floating-point operation, enabling us to analyze $x < y$ differently from $x \leq y$. The small non deterministic rounding errors introduced by the linearization would, however, make this impossible. □

Example 16. When interpreting an expression of the form `A[i]+B[i]`, where both `A` and `B` are arrays and `i` is an integer, there may be an invariant about the expression `A[i]+B[i]` for any `i` within the bounds of the arrays `A` and `B`. This invariant is lost when resolving array accesses. If we want abstract domains to use this invariant, we have to give them the opportunity to access expressions before array access resolution. Another solution is partitioning, but that may be too impractical or too costly. □

[8] Interestingly, in relational numerical abstract domains, algorithms are usually proved on real numbers, but the implementation is done using floating-point numbers, and soundness is achieved using rounding towards $\pm\infty$ as appropriate. Thus, there exists a real abstract semantics $[\![e]\!]^\sharp_{\mathbb{R}}$ and an over-approximation thereof using floating-point numbers $[\![e]\!]^\sharp_F$. Let us call $[\![e]\!]_F$ the concrete semantics over floating-point values and $[\![e]\!]_{\mathbb{R}}$ the over-approximation using real numbers and intervals for rounding errors, we have a tower of semantics: $[\![e]\!]_F \sqsubseteq [\![e]\!]_{\mathbb{R}} \sqsubseteq [\![e]\!]^\sharp_{\mathbb{R}} \sqsubseteq [\![e]\!]^\sharp_F$.

Abstract domains may use more precise abstract transformers whenever some properties are satisfied. There are two cases. In the first case, a special-purpose domain is used to check whether the properties hold and inform other domains using the broadcast output channel.

Example 17. Consider the following code:

```
volatile float vI;
void main () {
   float I, O = 0, R, OLD;
   while (1) {
      I = vI;
      OLD = O; R = I - OLD; O = I;
      if (R <= -0.2) { O = OLD - 0.2; }
      else if (0.2 <= R) { O = OLD + 0.2; }
   }
}
```

An input stream (denoted by the variable `I`) is modified by a rate limiter that bounds the difference between two successive outputs by the value 0.2. The variable `OLD` denotes the last output; the variable `R` is the difference between the new input and the last output; the variable `O` denotes the output stream.

When a rate limiter is involved in a complex dependence cycle, it is crucial that the arithmetic-progression domain be able to compute precise abstract properties all along the cycle. A specific domain collects the guards in "if-then-else" statements. When both guards `not(R <= -0.2)` and `not(0.2 <= R)` are satisfied, it checks (using the input channel) in the symbolic domain that both `R` matches `I-OLD` and `O` matches `I` (in floating-point arithmetics). Then, it warns the arithmetic-geometric progression domains that the absolute value of the variable `O` is less than the expression $|(1+\varepsilon_1)\texttt{OLD}+0.2+\varepsilon_2|$, where the floating-point numbers ε_1 and ε_2 model rounding errors and are computed automatically. □

In the second case, a domain may check that a special property holds using the input channel and performing abstract pattern matching over the concrete instruction it is currently abstracting. We have already seen in Ex. 1 that some transformers can be made more precise if they apply a special case when two variables are provably equal. In Ex. 18, we describe a more complex example.

Example 18. Digital filtering domains perform computations only when they discover that a variable X is equal to a linear combination of several other variables (the number of variables depends on the class of the filter). There are several trade-offs for detecting this property. The less generic way is to perform pattern matching of expressions in assignments (using a variety of abstraction levels for expressions: floating-point expressions, linearized expressions, etc.). This solution is very fragile: if a filter iteration is not computed using a single assignment (but, e.g. a loop scanning a parameter array), it is not discovered. It would be possible, but maybe costly, to collect the desired properties in a specially-designed abstract domain. Our solution is in-between those: we use the symbolic domain. Abstract pattern matching takes a pattern and an expression, and tries to unify them by replacing expression variables with the floating-point expression they are equal to in the symbolic domain. This collaboration uses the input channel. □

Finally, some abstract domains provide several implementations for an abstract transformer and rely on strategies to select which one should be used, depending on constraints computed by other domains.

Example 19. In arithmetic-geometric progression domains, a pair of elements may lack a least upper bound. We have implemented three ways to compute a bound: the first one favors the right argument, the second one favors the left argument, whereas the third one is a trade-off. The selection between the three strategies depends not only on the arithmetic-geometric constraints, but also on the dependency graph among variables and on the range of involved variables, which can be fetched from the input channel of the precondition. □

6.4 Reduction after Widenings

Special care should be taken when refining an extrapolation operator. Although it is always safe to refine its right argument; refining its left argument or its output may break the extrapolation process. Indeed, the standard assumption required to ensure the termination of the widening iterates (Sect. 4.1 and [1]) may not be applicable anymore. As a consequence, the analyzer may loop forever.

In ASTRÉE, we reduce the output of the extrapolation operator using the broadcast output channel, as in Ex. 12. We also refine the left argument of widenings when a constraint is missing, as in Ex. 9. Nevertheless, these kinds of refinements follow the hierarchic structure of domain networks, which prevents cyclic reductions. We ensure the termination of ASTRÉE by strengthening the definition of the widening.

7 Widenings

7.1 Framework

We use widenings to abstract the computation of post-fixpoints in a finite amount of time [1]. Formally, let D be a concrete domain and $D^\sharp$ be an abstract domain related via a concretization map $\gamma_{D^\sharp} \in D^\sharp \to D$. A widening operator $\triangledown_{D^\sharp}$ over an abstract domain $D^\sharp$ is a mapping in $D^\sharp \times D^\sharp \to D^\sharp$ such that [23, Lect. 18]:

- $\forall a, b \in D^\sharp$, $\gamma_{D^\sharp}(b) \subseteq \gamma_{D^\sharp}(a \triangledown_{D^\sharp} b)$; (W$_1$)
- for all (a_i), the sequence $(a_i^\triangledown)$ defined as $a_0^\triangledown = a_0$ and $a_{n+1}^\triangledown = a_n^\triangledown \triangledown_{D^\sharp} a_{n+1}$ is ultimately stationary. (W$_2$)

The second property implies that the widening relation $\to$ that is defined as $a \to b$ if and only if there exists c such that $a \triangledown_{D^\sharp} c = b$ is well founded. Nevertheless, there may be no relation between the information preorder $\sqsubseteq^\sharp$ (defined as $a \sqsubseteq^\sharp b$ if and only if $\gamma_{D^\sharp}(a) \subseteq \gamma_{D^\sharp}(b)$) and the relation $\to$.

Least fixpoint approximation is performed in the following way. Let F be a monotonic map in $D \rightarrow D$ and $\mathrm{F}_{D^\sharp} : D^\sharp \rightarrow D^\sharp$ be an abstraction of F satisfying $\forall a \in D^\sharp,\ (\mathrm{F} \circ \gamma_{D^\sharp})(a) \subseteq (\gamma_{D^\sharp} \circ \mathrm{F}_{D^\sharp})(a)$. The abstract operation $\mathrm{F}_{D^\sharp}$ needs not be monotonic with respect to the information preorder $\sqsubseteq^\sharp$. The abstract upward iteration $(x_n^\triangledown)$ of $\mathrm{F}_{D^\sharp}$ is defined as $x_0^\triangledown = \bot$ and $x_{n+1}^\triangledown = x_n^\triangledown \triangledown_{D^\sharp} \mathrm{F}_{D^\sharp}(x_n^\triangledown)$. The sequence $(x_n^\triangledown)$ is ultimately stationary and we denote its limit by l. The following lemma ensures that the limit of the abstract upward iteration is a post-fixpoint abstraction of the map F:

Lemma 1. *We have* $\mathrm{F}(\gamma_{D^\sharp}(l)) \subseteq \gamma_{D^\sharp}(l)$.

Proof. l is the limit of the upward-iteration, so $l = l \triangledown_{D^\sharp} \mathrm{F}_{D^\sharp}(l)$. By (W_1), we obtain: $\gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(l)) \subseteq \gamma_{D^\sharp}(l)$. By soundness of $\mathrm{F}_{D^\sharp}$, we also have $\mathrm{F}(\gamma_{D^\sharp}(l)) \subseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(l))$. So $\mathrm{F}(\gamma_{D^\sharp}(l)) \subseteq \gamma_{D^\sharp}(l)$. □

The fact that the information preorder $\sqsubseteq^\sharp$ and the widening relation $\rightarrow$ are not assumed to be related limits the usage of the widening far too much for our need. Indeed, as explained in Sect. 7.2, we would like to refine the abstract properties after a widening step. Moreover, as explained in Sect. 7.3, we would like to refrain from widening abstract properties at every iterate. Doing these carelessly could break the termination of the widening process. Thus, in Sect. 7.4, we strengthen the definition of the widening to safely allow these manipulations.

7.2 Reduction of Widenings

The interval abstract domain is in many ways the "base" abstract domain. Most of the properties that we check are properties of bounds, directly expressed in the interval domain; also, while for each variable we keep an interval, we do not necessarily keep the other kinds of abstract properties. However, the interval domain, by default, applies simple preset widening thresholds (enriched with constants encountered in comparisons). In order to prevent the intervals from being widened too much, which would result in false alarms, it is necessary to reduce them using more refined abstract properties. Thus, most numerical domains reduce the intervals after widening.

However, care must be taken not to reduce too much after a widening in order not to break the termination property of the widening. A classical example is the closure operation in the octagon abstract domain, which can be considered a reduction between separate domains, each considering only a couple of variables: if one applies the classical widening operation on octagons followed by closure (reduction), then termination is no longer ensured (e.g. see [11, Fig. 25–26]).

An alternate approach would be to modify the abstract transformers, refining both its inputs and output, instead of modifying the widening. We denote by $\rho : D^\sharp \rightarrow D^\sharp$ our reduction function (i.e. it satisfies $\forall a \in D^\sharp,\ \gamma_{D^\sharp}(a) \subseteq \gamma_{D^\sharp}(\rho(a))$). We can define the two following sequences:

$$\begin{cases} u_0 = \bot, \\ u_{n+1} = \rho(u_n \triangledown_{D^\sharp} (\rho(\mathrm{F}_{D^\sharp}(u_n)))); \end{cases} \qquad \begin{cases} v_0 = \bot, \\ v_{n+1} = v_n \triangledown_{D^\sharp} (\rho(\mathrm{F}_{D^\sharp}(\rho(v_n)))). \end{cases}$$

In ASTRÉE, we compute the sequence (u_n), whereas the alternate strategy implements the sequence (v_n). The sequence (v_n) is ultimately stationary even without strengthening the definition of the widening. But the sequence (u_n) can be computed more easily, while taking benefit of functional data-structures (i.e. balanced trees). Moreover, the sequence (u_n) provides a modular definition for the widening operator of reduced product domains.

Lemma 2. *The limit $\bar{u}$ of (u_i) (resp. $\bar{v}$ of (v_i)) satisfies* $\mathrm{F}(\gamma_{D^\sharp}(\bar{u})) \subseteq \gamma_{D^\sharp}(\bar{u})$ *(resp.* $\mathrm{F}(\gamma_{D^\sharp}(\bar{v})) \subseteq \gamma_{D^\sharp}(\bar{v})$*).*

Proof. For the limit $\bar{u}$ of (u_i), we have $\bar{u} = \rho(\bar{u}\nabla_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\bar{u}))))$ so $\gamma_{D^\sharp}(\bar{u}) = \gamma_{D^\sharp}(\rho(\bar{u}\nabla_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\bar{u})))))$ whence $\gamma_{D^\sharp}(\bar{u}\nabla_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\bar{u})))) \subseteq \gamma_{D^\sharp}(\bar{u})$ by soundness of the reduction ρ. By (W_1), $\gamma_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\bar{u}))) \subseteq \gamma_{D^\sharp}(\bar{u})$, whence $\gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(\bar{u})) \subseteq \gamma_{D^\sharp}(\bar{u})$ by soundness of ρ and so $\mathrm{F}(\gamma_{D^\sharp}(\bar{u})) \subseteq \gamma_{D^\sharp}(\bar{u})$ by soundness of $\mathrm{F}_{D^\sharp}$. Similarly $\bar{v} = \bar{v}\nabla_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\rho(\bar{v}))))$ implies $\gamma_{D^\sharp}(\bar{v}) = \gamma_{D^\sharp}(\bar{v}\nabla_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\rho(\bar{v})))))$ so $\gamma_{D^\sharp}(\bar{v}) \supseteq \gamma_{D^\sharp}(\rho(\mathrm{F}_{D^\sharp}(\rho(\bar{v})))) \supseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(\rho(\bar{v}))) \supseteq \mathrm{F}(\gamma_{D^\sharp}(\bar{v}))$ by (W_1) and soundness of ρ and $\mathrm{F}_{D^\sharp}$. □

Due to the non-monotonic behavior of the widening (especially with respect to the second argument), it seems difficult to compare the theoretical accuracy of the two approaches.

7.3 Delaying Strategies

Premature widenings may result in excessive over-approximation. This is particularly true when the first few iterations of the system perform some kind of initialization and do not give a good insight on the regular behavior of the loop. We therefore delay the application of the widening, replacing it with a mere abstract union, until a specified iteration, and start extrapolating afterwards.

Unfortunately, this is not sufficient and we may want to interleave unions and widenings even after the first iteration with widening. Consider the following example:

Example 20.
```
while (1) {
      X := Y + b;
      Y := a*X + c;
}
```
The sequence of assignments is equivalent to `Y := a*X + d` (with $d = c + a.b$), and so a widening with thresholds should find a stable interval. But if we perform a widening with thresholds at each step, each time we widen `Y`, `X` is increased to a value surpassing the threshold for `Y`, and so `X` is widened to the next stage, which in turn increases `Y` further and the next widening stage increases the value of `Y`. This eventually results in $\top$ abstract values for `X` and `Y`. We can see, however, that if we replace the widening with a union at every other step, `X` and `Y` will stabilize to the smallest threshold larger than their respective concrete bound. □

Our previous approach was the following: we first do N_0 iterations with unions on all abstract domains, then we do widenings unless a variable which was not stable becomes stable (this is the case of `Y` here when the threshold is big enough). We add a fairness condition to avoid livelocks in case for each iteration there exists a variable that becomes stable. Unfortunately, with large programs this strategy gives the following behavior: we never do widenings until the fairness condition is taken into account, then we do widenings at each iteration. So we fail in certifying our example.

The following approach supersedes the previous one: Each abstract property (or set of abstract properties) is fitted with a freshness indicator. At each iteration, the freshness indicator of unstable abstract properties is incremented. For each abstract property, the choice between taking the union or the widening among two consecutive iterates is determined according to the freshness indicator: each domain is fitted with a piece-wise affine function, which determines how often it is widened according to the freshness indicator of the abstract property.

7.4 Enforcing Termination

Both reductions (Sect. 7.2) and delayed widenings (Sect. 7.3) may break the termination of the extrapolation process. Even intersecting the abstract iterates with a constant abstract property (i.e. a weak form of reduction) may break the termination of the extrapolation process.

Example 21. We consider the set $D^\sharp$ of all parts of the interval $[0;1]$ containing both 0 and 1. We want to extrapolate the iterates of the function f that inserts in a set S each rational $\frac{1}{2^{n+1}}$ whenever $\frac{1}{2^n}$ is in S (e.g. $f(S) = \{S \cup \{\frac{1}{2^{n+1}} \mid \frac{1}{2^n} \in S\})$. We define both $\cap$ and $\cup$ as the classical set operators. We define $\triangledown_{D^\sharp}$ as: $a \triangledown_{D^\sharp} b = \rho(a, a \cup b)$ where $\rho(a,b)$ is obtained by making the convex union of several connected components of b until there are fewer connected components than in a, and fewer than five connected components. It is obvious that $\triangledown_{D^\sharp}$ is a widening, since along the abstract iterates the number of connected components decreases until it reaches 1, and the interval $[0;1]$ is the only element with one connected component. Then:

1. The sequence:
$$\begin{cases} u_0 = \{0;1\}, \\ u_{2n+1} = u_{2n} \cup f(u_{2n}), \\ u_{2n} = u_{2n-1} \triangledown_{D^\sharp} f(u_{2n-1}), \end{cases}$$
is not ultimately stationary.
Indeed, we have $u_{2n} = \{0\} \cup [\frac{1}{2^{2n}};1]$ and $u_{2n+1} = \{0; \frac{1}{2^{2n+1}}\} \cup [\frac{1}{2^{2n}};1]$.
2. The sequence:
$$\begin{cases} u_0 = \{0;1\}, \\ u_{n+1} = (u_n \triangledown_{D^\sharp} f(u_n)) \cap \{0;1;\frac{1}{2^k} \mid k > 0\}, \end{cases}$$
is not ultimately stationary.
Indeed, we have $u_n = f^n(u_0)$. □

We solve this problem in two steps. First we strengthen the definition of the widenings, so that we can both delay the widening steps, and intersect the iterates with a constant value, without loosing the convergence of abstract iterates. Then, we restrict the kind of reductions that can be made after a widening step.

Strengthened Definition. To solve our problem, we require that the widening relation $\rightarrow$ and the information preorder $\sqsubseteq^\sharp$ are strongly related. We suppose that the abstract domain can be written as a finite product of totally ordered sets $(D_i^\sharp, \sqsubseteq_i^\sharp)_{i\in I}$. Moreover, we suppose that each sub-domain $D_i^\sharp$ is fitted with a widening operator $\triangledown_i$ such that:

- for any $a, b \in D_i^\sharp$, we have both $a \sqsubseteq_i^\sharp a \triangledown_i b$ and $b \sqsubseteq_i^\sharp a \triangledown_i b$, (W_1')
- and the relation $\rightarrow_i'$ defined as: "for any $a, d \in D_i^\sharp$, $a \rightarrow_i d$ if and only if there exist $b, c \in D_i^\sharp$ such that $a \sqsubseteq_i^\sharp b$ and $d = b \triangledown_i c$" is well-founded. (W_2')

In the following theorem, we want to extrapolate the iterates of a mapping $\mathrm{F}_{D^\sharp}$. Each iterate is intersected with the abstract property $(\rho_i)_{i\in I} \in D^\sharp$ (we recall that $D^\sharp = \prod_{i\in I} D_i^\sharp$). The sequence $((b_i)_n)$ of boolean families denotes the delaying strategy of the widening.

Theorem 1. *Let $\mathrm{F}_{D^\sharp} \in D^\sharp \rightarrow D^\sharp$ be a map. Let $(\rho_i)_{i\in I} \in D^\sharp$ be a finite family of abstract elements. Let $((b_i)_n) \in (\{false; true\}^I)^{\mathbb{N}}$ be a family of booleans such that for any $i \in I$, the sequence $((b_i)_n)_{n\in\mathbb{N}}$ takes the value true an unbounded number of times. We write $\bot = (\bot_i)_{i\in I}$.*

Then, the sequence:

$$\begin{cases} (w_i)_0 = \bot_i, \\ u_{n+1} = \mathrm{F}_{D^\sharp}(w_n), \\ (v_i)_{n+1} = \begin{cases} \max_{\sqsubseteq_i^\sharp}((w_i)_n, (u_i)_{n+1}) & \textit{whenever } (b_i)_n = \textit{false}, \\ (w_i)_n \triangledown_i (u_i)_{n+1} & \textit{otherwise}\ , \end{cases} \\ (w_i)_{n+1} = \min_{\sqsubseteq_i^\sharp}((v_i)_{n+1}, \rho_i), \end{cases}$$

is ultimately stationary and sound.

Proof. Let i be an element of I. It is easy to see that sequence (w_i) is increasing. Moreover,

1. In the case where for any $n \in \mathbb{N}$, $(w_i)_n = (v_i)_n$, we introduce a sequence $(j_n) \in \mathbb{N}^{\mathbb{N}}$ such that $(b_i)_{j_n}$ is *true* for any $n \in \mathbb{N}$ (a such sequence exists by assumption). Then, we have $(w_i)_{j_n} \rightarrow_i (w_i)_{j_{n+1}}$. Since $\rightarrow_i$ is well founded, $(w_i)_{j_n}$ is ultimately stationary.
2. Otherwise, the sequence (w_i) is ultimately equal to the value ρ_i.

For the limits $\bar{u}$, $\bar{v}$, and $\bar{w}$ of $((u_i)_n)_{n\in\mathbb{N}}$, $((v_i)_n)_{n\in\mathbb{N}}$, and $((w_i)_n)_{n\in\mathbb{N}}$, we have $\bar{u} = \mathrm{F}_{D^\sharp}(\bar{w})$, $\gamma_{D^\sharp}(\bar{v}) \supseteq \gamma_{D^\sharp}(\bar{u})$, $\gamma_{D^\sharp}(\bar{v}) \supseteq \gamma_{D^\sharp}(\bar{w})$ (by $\gamma_{D_i^\sharp}(\max_{\sqsubseteq_i^\sharp}(a,b)) \supseteq \gamma_{D_i^\sharp}(a) \cup \gamma_{D_i^\sharp}(b)$ or (W_1')) and $\bar{w} \sqsubseteq^\sharp \bar{v}$ so $\gamma_{D^\sharp}(\bar{v}) = \gamma_{D^\sharp}(\bar{w}) \supseteq \gamma_{D^\sharp}(\bar{u}) = \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(\bar{w})) \supseteq \mathrm{F}(\gamma_{D^\sharp}(\bar{w}))$ proving soundness. □

This definition is satisfied by the widening with thresholds, which is applied to each single abstract constraint in all our domains.

Example 22. The widening $W = A \triangledown_{D^\sharp} B$ of two octagons [11] is performed point-wisely: given a constraint $\pm X \pm Y \leq c$ in A and $\pm X \pm Y \leq d$ in B with the same left member, we set the bound of the corresponding constraint in W to c if $c \geq d$, and to the next threshold greater than d if $d \geq c$. In order to gain precision, we may replace some widening application with a so-called *pre-widening* which sets this bound to $\max(c, d)$ instead. Note that contrary to the abstract union, our pre-widening does not apply constraint propagation to its argument, and thus, is less precise. Indeed, [11, Fig. 25–26] shows that mixing the widening (that tends to loosen bounds) with constraint propagation (that tends to refine bounds) breaks the convergence of the iterates. Mixing widenings with unions would have the same ill-effect. However, we can prove that mixing widenings with pre-widenings enforces termination by viewing the octagon domain as a finite product of totally ordered sets: there is one set for each left member $\pm X \pm Y$ of octagonal constraint; all sets are simply the set of reals with the standard order $\leq$; each set corresponds to the possible upper bounds of a single left member. □

Restricting Reduction Then, we must avoid cyclic reductions between components of the product domain. For that purpose, we use the hierarchical structure of the domain network: after a widening step, a domain can only refine its underlying domains. This ensures the termination of the analysis: the abstract iterates in the abstract domains that are at the top of the hierarchy are ultimately stationary. Once the abstract properties in the domains that are above one domain are stable, the reduction of abstract properties in this domain can be seen as an intersection with a constant abstract property. Thus, its abstract iterates are ultimately stationary.

8 Narrowings

The widening jumps above the abstraction of the concrete least fixpoint. Then, the result may be refined using downward iterations thanks to a narrowing operator. The ASTRÉE analyzer takes as parameter the number of downward iterations to compute.

Decreasing iterations raise several issues. The main problem is that the use of downward iterations may make the checking of the fact that we have computed an abstraction of the concrete least fixpoint much harder for an external procedure. Although, by construction, the limit of these decreasing iterations is indeed an abstraction of a concrete post-fixpoint, it may be hard to check *in the abstract* (i.e. without resorting to a more concrete, and thus costly, decision procedure). Performing this check instead of relying on the correctness of a complex iteration scheme is desirable to add confidence in the result of the analysis.

8.1 Frameworks

A narrowing operator $\triangle_{D^\sharp}$ over an abstract domain $D^\sharp$ [1] is a mapping in $D^\sharp \times D^\sharp \to D^\sharp$ such that: $\forall a, b \in D^\sharp, \gamma_{D^\sharp}(a) \cap \gamma_{D^\sharp}(b) \subseteq \gamma_{D^\sharp}(a \triangle_{D^\sharp} b)$. We require no termination criterion since, in ASTRÉE, the number of downward iterations is bounded by some user-chosen constant.

Given any $x \in D^\sharp$ (in practice $x = l$ is the limit of the upward iterations), the downward iteration $(x_n^\triangle)$ of $\mathrm{F}_{D^\sharp}$ from x is defined as $x_0^\triangle = x$ and $x_{n+1}^\triangle = x_n^\triangle \triangle_{D^\sharp} \mathrm{F}_{D^\sharp}(x_n^\triangle)$. In the following, we consider $\mathrm{F} \in D \to D$ a monotonic function and $\mathrm{F}_{D^\sharp}$ an abstraction of F (i.e. we have $\forall a \in D^\sharp$, $\mathrm{F}(\gamma_{D^\sharp}(a)) \subseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(a))$. We want to prove that downward iterates preserve abstractions of concrete post-fixpoints.

Theorem 2. *If there exists a concrete element $a \in D$ such that $\mathrm{F}(a) \subseteq a$ and $a \subseteq \gamma_{D^\sharp}(x)$ then, for any integer $n \in \mathbb{N}$, there exists a concrete element $a' \in D$ such that $\mathrm{F}(a') \subseteq a'$ and $a' \subseteq \gamma_{D^\sharp}(x_n^\triangle)$.*

First, we prove the following lemmas:

Lemma 3. *For any $a \in D$ and $x \in D^\sharp$, $a \subseteq \gamma_{D^\sharp}(x) \implies a \cap \mathrm{F}(a) \subseteq \gamma_{D^\sharp}(x \triangle_{D^\sharp} \mathrm{F}_{D^\sharp}(x))$.*

Proof. Let $a \in D$ and $x \in D^\sharp$ such that $a \subseteq \gamma_{D^\sharp}(x)$. Since F is monotonic, we have $\mathrm{F}(a) \subseteq \mathrm{F}(\gamma_{D^\sharp}(x))$. Then by soundness of $\mathrm{F}_{D^\sharp}$, we have $\mathrm{F}(\gamma_{D^\sharp}(x)) \subseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(x))$. Thus $\mathrm{F}(a) \subseteq \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(x))$. So $a \cap \mathrm{F}(a) \subseteq \gamma_{D^\sharp}(x) \cap \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(x))$. By definition of the narrowing operator, we have $\gamma_{D^\sharp}(x) \cap \gamma_{D^\sharp}(\mathrm{F}_{D^\sharp}(x)) \subseteq \gamma_{D^\sharp}(x \triangle_{D^\sharp} \mathrm{F}_{D^\sharp}(x))$. We conclude that $a \cap \mathrm{F}(a) \subseteq \gamma_{D^\sharp}(x \triangle_{D^\sharp} \mathrm{F}_{D^\sharp}(x))$. □

Lemma 4. *For any $a \in D$, $\mathrm{F}(a) \subseteq a \implies \mathrm{F}(\mathrm{F}(a) \cap a) \subseteq \mathrm{F}(a) \cap a$.*

Proof. Let $a \in D$ such that $\mathrm{F}(a) \subseteq a$. Since F is monotonic, we have $\mathrm{F}(\mathrm{F}(a)) \subseteq \mathrm{F}(a)$. Moreover, we have $\mathrm{F}(a) \cap a = \mathrm{F}(a)$. We conclude that $\mathrm{F}(\mathrm{F}(a) \cap a) = \mathrm{F}(\mathrm{F}(a)) \subseteq \mathrm{F}(a) = \mathrm{F}(a) \cap a$. □

Then, Thm. 2 can easily be proved by induction on $n \in \mathbb{N}$ with $a' = F(a) \cap a$.

8.2 Practical Aspects

One may be satisfied by the fact that downward iteration provides, by construction, an abstraction of the concrete post-fixpoint (**Th. 2**). Nevertheless, we could expect more such as improvement $\gamma_{D^\sharp}(a \triangle_{D^\sharp} b) \subseteq \gamma_{D^\sharp}(a)$ [1]. Moreover, termination tests (for upward iterations) are performed using a decidable relation $\sqsubseteq^\sharp$ such that $a \sqsubseteq^\sharp b$ implies $\gamma_{D^\sharp}(a) \subseteq \gamma_{D^\sharp}(b)$ (but which is not necessarily a preorder). Then, by definition, the limit l of upward iterates of the abstract operation $\mathrm{F}_{D^\sharp}$ satisfies $\mathrm{F}_{D^\sharp}(l) \sqsubseteq^\sharp l$. But there is no reason for the downward iterates to satisfy this property (even if $a \sqsubseteq^\sharp b \iff \gamma_{D^\sharp}(a) \subseteq \gamma_{D^\sharp}(b)$), because the abstract operation $\mathrm{F}_{D^\sharp}$ is in practice not monotonic with respect to $\sqsubseteq^\sharp$.

To solve this problem, whenever downward iterations provide an abstract property that is not a post-fixpoint of $\mathrm{F}_{D^\sharp}$ (with respect to our $\sqsubseteq^\sharp$), we start upward iterations again. And then, downward iterations again. To get a finite analysis, we only "rebound" a fixed number of times. For our last try, we do not perform downward iterations, so that we get a checkable post-fixpoint of $\mathrm{F}_{D^\sharp}$.

9 Conclusion and Further Challenges

Astrée has shown to be easily extensible. Our early successes have incited our industrial partners to try and apply the analyzer to classes of programs for which it was not designed. It was thus necessary to improve domains or create new ones without a complete overhaul of the system. As with any major software endeavor, our experience is that dependencies should be limited and orthogonality encouraged; one abstract domain should be able to perform correctly (if sometimes suboptimally) if another one is not present, or has been modified.

We have targeted applications where the objective is a sound result of zero false alarms. This is very different from some other (commercial or academic) static analyzers whose objective is to find bugs. While the two approaches share common tools, they differ in that bug finding does not need to be sound (some real errors may be ignored) while a high degree of completeness (few false alarms) is expected on a variety of programs. A bug finder should thus probably ignore constructs that it fails to "understand" properly (for instance, writes through a pointer possibly aliased to many variables because of over-approximation). A program verification tool such as Astrée does not have that luxury, and this is why we claim that such programs should often contain domain-specific abstractions, capable of addressing constructs, structures and algorithms that generic abstraction do not "understand". The consequence for the designer of the analysis is that it should be easy to plug new abstractions at any level.

So far, Astrée has been targeted towards single-threaded programs. However, we have already implemented analyses for a restricted class of parallel programs, and we expect to consider wider classes (e.g. multi-threaded code in shared-memory systems). Our memory abstraction is currently a simple non-relational one, but we expect that more precise analyses (e.g. shape analysis and separation properties) will be necessary to tackle programs featuring dynamic manipulations of memory. One difficulty will be our stringent efficiency constraints, since we consider large programs. The other will be to achieve zero false alarms.

Bibliography

1. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM POPL. (1977) 238–252
2. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation **2** (1992) 511–547
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In Mogensen, T., Schmidt, D., Sudborough, I., eds.: The Essence of Computation: Complexity,

Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566. Springer (2002) 85–108

4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN '2003 Conf. PLDI, San Diego, ACM Press (2003) 196–207
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In Sagiv, M., ed.: Proc. 14th ESOP '2005, Edinburgh. LNCS 3444, Springer (2005) 21–30
6. Mauborgne, L.: ASTRÉE: Verification of absence of run-time error. In Jacquart, P., ed.: Building the Information Society. Kluwer Academic Publishers (2004) 385–392
7. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. LCTES 2006. Ottawa, Ontario, Canada, 14–16 June 2006, ACM Press (2006) 54–63
8. Monniaux, D.: The parallel implementation of the ASTRÉE static analyzer. In Yi, K., ed.: APLAS. Volume 3780 of LNCS., Springer (2005)
9. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Paris, France, Dunod, Paris, France (1976) 106–130
10. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: VMCAI'06. Volume 3855 of LNCS., Springer (2002) 348–363
11. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19** (2006) 31–100
12. Feret, J.: Static analysis of digital filters. In Schmidt, D., ed.: Proc. 30th ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 33–48
13. Feret, J.: The arithmetic-geometric progression abstract domain. In Cousot, R., ed.: Proc. 6th VMCAI '2005, Paris. LNCS 3385, Springer (2005) 2–58
14. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In Sagiv, M., ed.: Proc. 14th ESOP '2005, Edinburgh. LNCS 3444, Springer (2005) 21–30
15. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6th ACM POPL. (1979) 269–282
16. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, documentation and user's manual (release 3.06). Technical report, INRIA, Rocquencourt, France (2002)
17. Miné, A.: The octagon abstract domain library (2006) `www.di.ens.fr/~mine/oct/`.
18. ANSI/ISO: Programming languages – C. (1999) Standard ISO/IEC 9899:1999(E).
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: SPIN. Volume 2648 of LNCS., Springer (2003) 235–239
20. Cousot, P.: Verification by abstract interpretation, invited chapter. In Dershowitz, N., ed.: Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday. LNCS 2772, Springe, Taormina, Italy (2003) 243–268
21. Cousot, P.: The calculational design of a generic abstract interpreter, invited chapter. In Broy, M., Steinbrüggen, R., eds.: Calculational System Design. Volume 173. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, The Netherlands (1999) 421–505
22. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In Schmidt, D., ed.: Proc. 30th ESOP '2004, Barcelona. LNCS 2986, Springer (2004) 3–17
23. Cousot, P.: MIT course 16.399: Abstract Interpretation. `web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/` (2005)