Form Methods Syst Des (2009) 35: 229–264
DOI 10.1007/s10703-009-0089-6

Why does ASTRÉE scale up?

Patrick Cousot · Radhia Cousot · Jérôme Feret · Laurent Mauborgne · Antoine Miné · Xavier Rival

Published online: 11 November 2009

Abstract ASTRÉE was the first static analyzer able to prove automatically the total absence of runtime errors of actual industrial programs of hundreds of thousand lines. What makes ASTRÉE such an innovative tool is its scalability, while retaining the required precision, when it is used to analyze a specific class of programs: that of reactive control-command software. In this paper, we discuss the important choice of algorithms and data-structures we made to achieve this goal. However, what really made this task possible was the ability to also take semantic decisions, without compromising soundness, thanks to the abstract interpretation framework. We discuss the way the precision of the semantics was tuned in ASTRÉE in order to scale up, the differences with some more academic approaches and some of the dead-ends we explored. In particular, we show a development process which was not specific to the particular usage ASTRÉE was built for, hoping that it might prove helpful in building other scalable static analyzers.

This work was supported by the INRIA project-team ABSTRACTION common to the CNRS and the École Normale Supérieure.

P. Cousot (✉) · R. Cousot · J. Feret · L. Mauborgne · A. Miné · X. Rival
École Normale Supérieure, 45 rue d'Ulm, 75230 Paris Cedex 05, France
e-mail: Patrick.Cousot@ens.fr

R. Cousot
e-mail: Radhia.Cousot@ens.fr

J. Feret
e-mail: Jerome.Feret@ens.fr

L. Mauborgne
e-mail: Laurent.Mauborgne@ens.fr

A. Miné
e-mail: Antoine.Mine@ens.fr

X. Rival
e-mail: Xavier.Rival@ens.fr

Keywords Abstract Interpretation · Embedded critical software · Formal methods · Safety · Scalability · Static analysis tool · Verification

1 Introduction

Program verification consists in formally proving that the strongest property of a semantic model of the program runtime executions implies a given specification of this model. This strongest property is called the *collecting semantics*. A semantic model of program executions classically used in program verification is an *operational semantics* mapping each program to a *transition system* [16] consisting of states (e.g. describing the control by a call stack and a program counter as well as the memory) and transitions (describing how a program execution evolves from one state to another). An example of collecting semantics is the reachable states by successive transitions from initial states. This collecting semantics can be expressed in fixpoint form [15].

Static analysis aims at determining statically dynamic properties of programs that is properties of their runtime executions. It is therefore an approximation of the fixpoint collecting semantics. This approximation is then used to prove the specification.

Abstract interpretation [10, 15, 16] can be used to guarantee soundness of this approach. Abstract interpretation is a theory of exact or approximate abstraction of mathematical structures allowing for the systematic derivation of correct approximations of undecidable or highly complex problems in computer science. When applied to static analysis, it can be used to formally derive and compute over- (or under-)approximations of the fixpoint collecting semantics thanks to direct or iterative fixpoint approximation methods.

1.1 Motivating example

In principle, the correctness proof could be done by exhaustive exploration of the state space which, but for limited cases, is subject to state explosion. For example, the verification of the filtering program of Fig. 1 (where `main` simulates a plausible calling environment and the loop runs for 10 hours at a clock rate of 10 ms) by CBMC [8] with `MiniSAT` leads to the following result on a Macintosh Intel 4 GB:

```
Script started on Tue Jul 29 23:44:00 2008
% time cbmc filter.c

file filter.c: Parsing

Converting
Checking filter
Generating GOTO Program
Pointer Analysis
Adding Pointer Checks
Starting Bounded Model Checking
Unwinding loop 1 iteration 1
Unwinding loop 1 iteration 2
Unwinding loop 1 iteration 3
```

```
 1 typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
 2 BOOLEAN INIT; float P, X;
 3 void filter () {
 4   static float E[2], S[2];
 5   if (INIT) { S[0] = X; P = X; E[0] = X; }
 6   else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
 7                + (S[0] * 1.5)) - (S[1] * 0.7)); }
 8   E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
 9 }
10 int main () {
11   int i = 1; X = 5.0; INIT = TRUE;
12   while (i < 3600000) {
13     X = 0.9 * X + 35; /* simulated filter input */
14     filter (); INIT = FALSE; i++; }
15 }
```

Fig. 1 Filter program

```
...
Unwinding loop 1 iteration 95479
cbmc(34799) malloc: *** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
terminate called after throwing an instance of 'std::bad_alloc'
  what():  St9bad_alloc
...
Abort
29668.051u 101.916s 8:20:41.88 99.0% 0+0k 1+10io 2680pf+0w
% ^Dexit
Script done on Wed Jul 30 09:08:58 2008
```

The analysis by ASTRÉE [2, 3, 22, 23, 25, 47] of the above program (modified with a directive to display the over-approximation of the set of reachable values of P):

```
% diff -U1 filter.c filter-a.c
--- filter.c 2009-06-08 16:34:18.000000000 +0200
+++ filter-a.c 2009-06-08 16:34:18.000000000 +0200
@@ -8,2 +8,3 @@
   E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
+  __ASTREE_log_vars((P));
 }
```

is as follows:

```
% (time astree --exec-fn main filter-a.c) |& egrep "WARN|pf+"
0.657u 0.060s 0:00.72 98.6% 0+0k 0+4io 0pf+0w
%
```

The absence of warning (lines containing the WARN keyword) indicates a proof of absence of runtime error.

It is frequently claimed that in abstract static analysis, "the properties that can be proved are often simple" [27]. Note that to check for the absence of runtime error, ASTRÉE must first *discover* an interval of variation of P:

```
% astree --exec-fn main filter-a.c |& grep "P in" | tail -n1
direct = <float-interval: P in [-1418.3854, 1418.3854] >
```

which is definitely a "weak property" (that is "simple to state") but is not so simple to prove, and even less to discover. Finding this property requires finding a non trivial non-linear invariant for second order filters with complex roots [28], which can hardly be found by exhaustive enumeration. Moreover, even given the interval, it is not possible to prove that it is invariant with intervals only. This phenomenon is also frequent in mathematics: there are theorems that are simple to state, harder to discover and difficult to prove. Lemma can be much more complex to state than theorems.

1.2 Objective of the paper

The objective of this paper is to explain why ASTRÉE does scale up. We show that one of the main reasons is that it is based on abstract interpretation [10, 15, 16]. From that base, it follows that it is entirely automatic and sound. The flexibility of the theory also allows the use of infinite abstract domains [18] with convergence acceleration by widening [10, 15], which can be exploited to derive precise results, while avoiding interminable exhaustive exploration. However, the theory of abstract interpretation offers a wide panel of choices that do not ensure precision and scalability *per se*. Judicious choices of abstractions and their algorithmic implementations have to be made to reach this goal. In this paper we discuss the main design choices ensuring the scalability of ASTRÉE.

The paper is organized as follows. We first recall in Sect. 2 important aspects of ASTRÉE, such as its target application and an overview of its organization. The following six sections describe various aspects of ASTRÉE, explaining choices and sometimes dead-ends on these aspects: Sect. 3 focuses on the way disjunctions are handled; Sect. 4 motivates the choice of non-uniform abstractions; Sect. 5 describes features of the iterator used in ASTRÉE; Sect. 6 describes choices in the abstract domains and in particular the way memory is modeled; Sect. 7 describes the particular way we implemented domain reductions, that is communications between abstract domains; finally, Sect. 8 presents (parts of) the development process which allowed the production of a precise and efficient static analyzer. Sect. 9 concludes.

2 ASTRÉE

ASTRÉE is a tool designed to show the absence of run-time errors in critical embedded C programs [2, 3, 22, 23, 25, 47]. The development of that tool started in 2001, answering needs from the AIRBUS company [26]. The tool takes a program (a set of source files), optional files describing the range of the inputs and the platform on which the program is supposed to run, inline options and outputs a textual log file describing the invariants found by ASTRÉE and warnings whenever the tool is not able to prove that the program is correct. The result is rather raw and typically one usually uses `grep` on the output to check for the `WARN` keyword but the tool respects all the requirements of the end-user. That is a *sound verification*, *precise* enough to show the absence of errors and capable of *scaling* to their large critical programs.

2.1 Requirements

Because of undecidability for infinite systems or high complexity for large finite systems, static verification is very difficult hence is often made unsound, or incomplete, or does not scale up.

2.1.1 Sound verification versus bug finding

We use the term *verification* as the mathematical process of establishing the validity of a specification with respect to a concrete collecting semantics (as opposed to the other meanings of the term corresponding to empirical means such as testing of some program executions, the partial exploration of formal models of programs, or inconclusive static analyses with false positives or negatives).

Unsound static analyzers do not cover the full collecting semantics (which may even be incorrect e.g. when skipping over parts of the program which are hard to analyze). The analysis has false negatives since it can miss bugs that exist but are not reported by the analysis procedure. This is sometimes accepted for bug-finding tools where total correctness is not the ultimate goal. This is not acceptable for safety or mission-critical software where no bug should be left undiscovered.

As opposed to bug-finding methods, such as tests where it is extremely difficult to decide when to end the debugging process, the absence of alarms signals the end of the automatic verification by ASTRÉE. Moreover, it is exactly known which category of bugs have been eliminated by ASTRÉE's validation.

2.1.2 Implicit versus explicit specifications

In ASTRÉE, the concrete operational semantics of C programs is given by the C ISO/IEC 9899:1999 standard, restricted by implementation-specific behaviors, programming guidelines and hypotheses on the environment (which can be tuned by analysis options). The specification is implicit: there should be no runtime error with respect to the concrete semantics. Whenever the standard is imprecise, ASTRÉE always takes a conservative approach, assuming that every behavior respecting the standard and the implementation-specific behaviors may happen. ASTRÉE has options to tune the specification, e.g., whether to warn on integer arithmetic overflows (including or excluding left shifts), on overflows on implicit and explicit integer conversions, on overflows in initializers, etc. The fact that ASTRÉE has an implicit specification (but for the static invariant assertions which may optionally be included by the end-user in the program text) is essential to its success since the end-users do not have to write formal specifications (which are often as large, complex, and difficult to maintain, if not more, than the program itself).

ASTRÉE checks the absence of runtime errors that is that no program execution, as defined by the concrete semantics, either:

- violates the C ISO/IEC 9899:1999 standard (e.g. division by zero, array index out of bounds),
- violates an implementation-specific undefined behavior (e.g. exceeding implementation-defined limits on floats, `NaN`, $+\infty$, $-\infty$),
- violates the programming guidelines (further restricting the correct standard and implementation-specific behaviors e.g. expressions have at most one side effect, signed integers should not overflow),
- violates a user-provided assertion `__ASTREE_assert((`*b*`))`, where *b* is a boolean C expression (which must all be statically verified, as opposed to the dynamic C `assert(`*b*`)`).

2.1.3 Precision versus incompleteness

By the very nature of approximation, abstract static analyzers do have *false alarms* (also called false positives or spurious warnings) that is error messages about potential bugs that

```
% cat -n falsealarm.c
     1 void main() { int x, y;
     2   if ((-4681 < y) && (y < 4681) && (x < 32767) &&
     3       (-32767 < x) && ((7*y*y - 1) == x*x))
     4    { y = 1 / x; };
     5 }
% astree --exec-fn main falsealarm.c |& grep "WARN" | fmt -s
-w 62
falsealarm.c:4.9-14::[call#main@1:]: WARN: integer division by
zero [-32766, 32766]
%
```

Fig. 2 A false alarm with Astrée

```
% cbmc falsealarm.c --overflow-check | tail -6 | fmt -s -w 62
Violated property:
  file falsealarm.c line 3 arithmetic overflow on * -4681 < y
  && y < 4681 && x < 32767 && -32767 < x => !overflow("*", 7,
  y)

VERIFICATION FAILED
%
```

Fig. 3 An example where CBMC is incomplete

do not exist in any execution of the program. When analyzing a given program for a given specification, a finite abstraction always exists to make the proof [13]. However, this is no longer true when considering an infinite family of programs [18]. In this last case, the challenge is to find the appropriate balance between coarse abstractions which are not costly but produce many false alarms and precise abstractions that produce no false alarms but are expensive, if not impossible, to compute. In practice, the challenge is met by Astrée on a significant family of programs. Nevertheless, and by incompleteness, there are infinitely many programs on which Astrée will produce false alarms, as in Fig. 2. Astrée is sound and precise enough on the family of programs for which it was designed (false alarms are very rare) but is likely to be imprecise on programs outside this family. Note that CBMC [8] is also incomplete (Fig. 3).

More precisely, the programs for which Astrée was designed are control-command software automatically generated from a graphical language à la Scade. They are composed of a large reactive synchronous loop that calls at each clock tick a set of functions assembled with macro-expanded blocks. Each block is generally simple and consists in a few statements of C and at most two nested loops. It follows that Astrée was *not* designed to support:

- recursive function calls,
- dynamic memory allocations,
- backward gotos,
- long jumps,
- concurrency,
- very complicated data structures,
- or highly nested loops.

```
% cat -n explode.c
     1 void main() {
     2    /* uninitialized local variables */
     3    unsigned int a1, a2, a3, a4, a5, a6; int r;
     4    while(a1<20) { a1++; }
     5    while(a2<20) { a2++; }
     6    while(a3<20) { a3++; }
     7    while(a4<20) { a4++; }
     8    while(a5<20) { a5++; }
     9 }
% (time astree --exec-fn main --partition-all explode.c) |& egrep
  "error:|WARN|pf\+"
*** error: can't allocate region
Fatal error: out of memory.
5072.204u 44.410s 1:29:31.01 95.2% 0+0k 0+0io 45661pf+0w
%
```

Fig. 4 An example on which ASTRÉE runs out of memory

There were of course also difficult points in that family, such as the size of the programs (up to a million lines of C), with one big loop interrelating nearly all variables (so that we cannot slice programs and analyse each part in isolation), pointers on functions, or floating-point computations.

2.1.4 Scalability versus analysis in the small

Experience has shown that ASTRÉE does scale up for the family of programs for which it was designed which are typically of a few hundred thousands to a million lines of C code. This does not mean that ASTRÉE does scale up on all programs (with any parameterization and analysis directives). Figure 4 shows an example which essentially consists in exhaustively exploring each execution trace in the program separately (due to the option `--partition-all` specified by the user). However, with the appropriate abstraction (i.e. limiting the amount of case analysis performed on execution traces, which is the default when no `--partition-all` option is specified by the user), we immediately get:

```
% (time astree --exec-fn main explode.c) |& egrep
"error:|WARN|pf"
0.431u 0.058s 0:00.49 97.9% 0+0k 0+0io 0pf+0w
%
```

Note that CBMC [8] is also sensitive to an appropriate choice of options. We have

```
% time cbmc explode.c
...
Unwinding loop 1 iteration 1
Unwinding loop 1 iteration 2
...
Unwinding loop 1 iteration 8731
...
Abort
```

while the verification succeeds with an appropriate choice of the number of unrollings:

Fig. 5 Benchmark analyses with Astrée

version	code size	analysis time	iterations
1	70 Klines	1h 35mn	49
2	154 Klines	11h 36mn	218
3	173 Klines	14h 30mn	199
1	122 Klines	41mn	27
2	502 Klines	7h 34mn	69
3	544 Klines	22h 37mn	181
4	594 Klines	23h 00mn	123
5	780 Klines	32h 00mn	144
6	705 Klines	27h 54mn	141

```
% (time cbmc --unwind 21 explode.c) | tail -2
VERIFICATION SUCCESSFUL
0.334u 0.023s 0:00.36 97.2% 0+0k 0+0io 0pf+0w
%
```

The previous discussion on precision and scalability shows that the effectiveness of abstract static analyzers can only be evaluated with respect to precise objectives. Astrée aims at proving the absence of runtime errors in safety and mission critical synchronous control-command C code [2, 3, 22, 23, 47]. Typically, the analyzed C code is automatically generated from a specification language in terms of primitives written by hand (such as integrators, filters, limiters, etc.). The challenge met by Astrée is therefore to scale up without false alarms for that class of programs.

To illustrate the scalability of Astrée, Fig. 5 presents some analysis benchmarks on various versions of two families of industrial software for which Astrée was designed. Together with the approximate size of the C code (before any preprocessing and macro expansion), we give the analysis time and number of abstract iterations with widening (Sect. 3.3) required by Astrée to find an invariant for the reactive synchronous loop. All these benchmarks were run on a single core of a 2.6 GHz 64-bit Intel Linux workstation. Note that, in each family, the analysis time is roughly proportional to the code size but also to the number of iterations (see Sect. 5.2 for insights on how this number varies depending on the complexity of feedback loops in the program and the widening strategies adopted by Astrée). It is difficult to provide reliable memory consumption figures due to the garbage-collected nature of OCaml; at least we can say that the smaller analyses can be run on a 2 GB laptop while 8 GB is sufficient for the largest ones.

2.2 The structure of Astrée

Before discussing the choices made in Astrée and the reasons for these choices, we present briefly the results of these choices, i.e., the current state of its structure.

2.2.1 Engineering facts

Astrée is a self-contained tool. It consists in 80,000 lines of OCaml [46] and 12,000 lines of C. The C parser was developed for Astrée in OCamlYacc. It allows Astrée to handles full C with the exception of unstructured backward branching, conflicting side effects, recursion, dynamic heap allocation, library calls (which have to be stubbed), system calls (hence parallelism), exceptions (but the clock tick of synchronous programs).

The development of Astrée required about 7 years of effort for a small team of four to five researchers-programmers. The structure of the code is decomposed in OCaml modules allowing independent development.

2.2.2 *Iterator and abstract domains*

Following the abstract interpretation design, ASTRÉE is composed of an iterator and a number of abstract domains communicating to obtain precise results. The iterator follows the flow of the program, starting from a main function which is an argument of the analyzer.

Here is a succinct list of the abstract domains currently used in ASTRÉE:

- An interval domain [14] that infers for each integer or floating-point variable V_i a property of the form $V_i \in [a_i, b_i]$.
- A simple congruence domain [38] for integer properties of the form $V_i \equiv a_i[b_i]$.
- A non-relational bitfield domain that tells for each bit of each integer variable whether it must be 0, 1, or may be either.
- An octagon domain ([52], Sect. 6.3) for relations of the form $\pm V_i \pm V_j \leq c_{ij}$. The octagon domain does not relate all variables (which would be too costly) but only integer and floating-point variables selected automatically by a packing pre-analysis (Sect. 4.3).
- A boolean decision tree domain (Sect. 3.2) to infer disjunctions of properties based on the value of boolean variables. Decision trees also employ an automatic packing pre-analysis to pre-select the set of variables to relate together.
- A symbolic propagation domain ([53], Sect. 6.4) that remembers the last expression assigned to each variable and performs substitution and simplification in subsequent expressions.
- An abstract domain for digital filters [28, 30] that discovers that a variable V lies within the sequence defined by a first order $V_i = \alpha V_{i-1} + aI_i + bI_{i-1}$ or second order $V_i = \alpha V_{i-1} + \beta V_{i-2} + aI_i + bI_{i-1} + cI_{i-2}$ digital filter with input I, up to some error interval $[-\epsilon, \epsilon]$.
- An arithmetic-geometric progression domain [29] that infers properties of the form $|V| \leq \alpha C + \gamma$ or $|V| \leq \alpha(1+\beta)^C + \gamma$, where C is the implicit clock variable counting the number of iterations of the synchronous loop. The domain is helped by a dependency pre-analysis that selects a set of candidate variables V.
- A domain of generic predicates that enables an accurate analysis of the functions that implement Euclidean remainders.
- A domain to discover equivalence classes of equal variables. This information is used in the digital filter domain.
- A pointer domain ([51], Sect. 6.1.1) that associates with each pointer a set of possible pointed-to variables and abstracts pointer offsets using all the integer numerical abstract domains above.
- A memory structure domain ([51], Sect. 6.1.3) that dynamically splits composed data-structures (such as arrays, structures, or unions) into cells of integer, floating-point or pointer type.
- A trace partitioning domain ([58], Sect. 3.1) to infer disjunctions of properties based on the history of the computation using control or data criteria. An automatic pre-analysis is used to infer relevant places where to perform trace partitioning.

2.2.3 *Interface*

The input of ASTRÉE is a set of files containing the sources of the program to analyze. Optionally, it can take into account the range of some input variables in another file, and the specifications for the intended platform on which the program should run in yet another file. A vast number of options can be given to the analyzer, in order to specify the function to analyze or which abstract domains (not) to use or some parameters of the abstract domains or iterator. The growing number of such options (currently 150) justified the development of a graphical interface which organizes them. One last source of inputs for ASTRÉE is the directives (starting with `__ASTREE`) which can be written in the source of the code to analyze. Such directives can control the output (asking to display the value of a variable at a given point) or help abstract domains when testing new strategies or trying to find the origin of alarms. Ultimately, ASTRÉE is configured internally so that on the very specific kinds of code analysed by our end-users, no directive and only a minimal set of options and configuration is needed. Directives and options are used mainly by the developers during the development phase, and by the end-users when experimenting with the analysis of new codes [26, 60, 61] (e.g. new versions of their software, or after changes to the code generator).

The output can provide a human-readable description of the invariants found and proved by ASTRÉE. It can also provide a more structured flow that can be exploited by a graphical interface. Currently, two such interfaces have been developed.

3 Disjunctions

A major difficulty in abstract interpretation is to handle disjunctions. On the one hand, keeping precise disjunctive information is very costly (and can end up in handling the disjunction of the semantic state singletons). On the other hand, approximating disjunctions is the main source of imprecision in abstractions. The proper abstraction of disjunctions is therefore a key to achieve scalability.

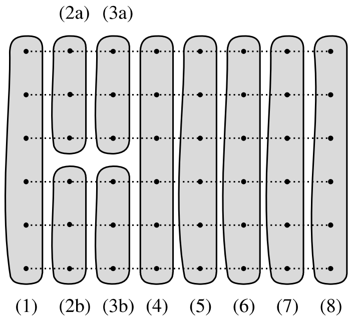
In case of finite abstractions, like in dataflow analysis [42] or predicate abstraction [37], one way to recover disjunctive information is to consider merge-over-all-paths abstractions [42] or, equivalently, to consider the disjunctive completion of the abstract domain [16, 31, 33], or to consider an automatic refinement to the disjunctive completion [40], or a refinement towards the concrete collecting semantics [7, 24]. These extreme solutions are too costly and so do not scale up in the large. For infinitary abstractions, the merge-over-all-paths is not computable but portions of paths can be abstracted together to make the analysis feasible [9, 41].

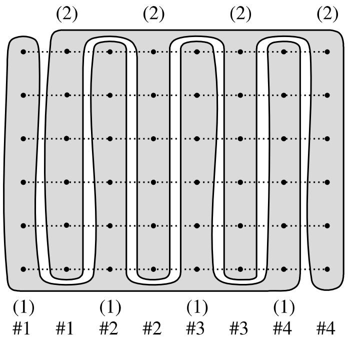
Weaker, but scalable, alternatives are discussed below. Note that disjunctive completion as well as the weaker alternatives discussed below cannot solve all false alarms and so the abstraction may ultimately have to be refined as discussed in later sections.

3.1 Trace partitioning versus local invariance analysis

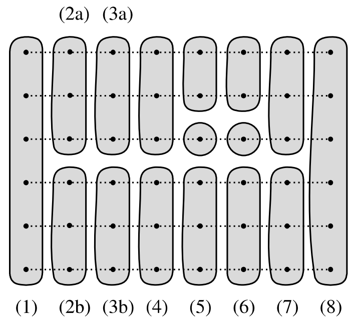
A non-distributive abstraction is an abstraction that may lose information whenever unions are performed. This is quite a frequent case. For example, intervals are a non-distributive abstraction (e.g. $[1,2]\cup[5,6]\subset[1,6]=[1,2]\sqcup[5,6]$). To gain precision in such cases, ASTRÉE proves that all program execution traces satisfy an invariance property by partitioning the set of all traces into sets of trace portions abstracted separately [48, 58].

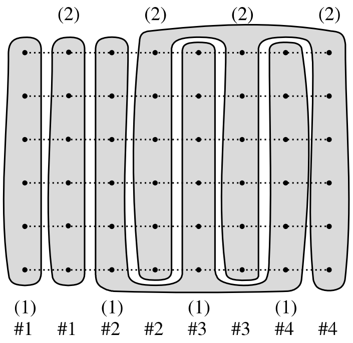
In the classic state partitioning by program points [10, 11], all reachable states corresponding to a given program point are over-approximated by a local invariant on memory states attached to this program point. The two figures below illustrate graphically this partitioning on two examples. Both cases present six traces (horizontal lines) composed of eight program states (dots) each. (1) to (8) denote program points, while #1 to #4 denote loop iterations. Each gray zone denotes a set of program states that are abstracted together. The left figure corresponds to an `if-then-else` statement, where program points (2a)–(3a) above are in the `then` branch, program points (2b)–(3b) below are in the `else` branch, and program points (1) and (4)–(8) are statements before and after the conditional. The right figure corresponds to a loop with four iterations #1 to #4 of a body consisting of two statements (1)–(2). In both cases, ASTRÉE collects together program states from all traces at each program point, even for program executions that reach several times the same point (as in the loop example). Thus, the left example results in ten abstract memory states computed and the right example results in two abstract memory states.





Conversely, when employing trace partitioning, program states at the same program point but belonging to different traces may be collected separately. Consider the two figures below, corresponding to the two same programs as before. On the left figure, tests cases are prolonged beyond the end of the `if`, so that we abstract separately the set of states at program points (4)–(7) depending on whether the `then` or `else` branch was taken. Moreover, it is possible to perform case analysis to distinguish traces based on the value some variable takes at some program point. Indeed, at program points (5)–(6), traces that come from the `then` branch are further partitioned into two sets depending on the value some variable takes at point (5). To ensure efficiency, the case analysis can be terminated by (partially) merging trace partitions at some program point, as shown in (7) and (8). The right figure corresponds to unrolling the loop body once, so that program states at both program points in the first iteration are abstracted separately from those in the following three iterations.





It follows that trace partitioning abstract interpretation [48, 58] combines the effects of case analysis and symbolic execution as in Burstall's intermittent assertion method [6, 19] as opposed to state partitioning as found in Floyd/Naur/Hoare invariant assertion proof method [17, 32, 43, 55]. It can be implemented easily by on-the-fly program transformation/code generation in the abstract interpreter. Trace partitioning is then much more efficient than the merge-over-all-paths or disjunctive completion since it is applied locally, not globally on the whole program: the case analysis always ends with an explicit merge of the cases (an implicit merge is performed at the end of functions).

Fig. 6 The clip program

```
% cat clip.c
int clip(int x, int max, int min) {
  if (max >= min) {
    if (x <= max) max = x;
    if (x <  min) max = min;
  }
  return max;
}
void main() {
  int m = 0; int M = 512; int x, y;
  y = clip(x, M, m);
  __ASTREE_assert(((m<=y) && (y<=M)));
}
```

```
% astree --exec-fn main clip.c --dump-partition |&
grep_from_until.pl "int max" "void"
signed int (clip)(signed int x, signed int max, signed int min);
signed int clip(signed int x, signed int max, signed int min)
{
  if ((max >= min))
  {
    __ASTREE_partition_control((0))
    if ((x <= max))
    {
      max = x;
    }
    if ((x < min))
    {
      max = min;
    }
    __ASTREE_partition_merge_last(());
  }
  return max;
}
```

Fig. 7 Clip program with partitioning directives

For example, the analysis of the `clip.c` program (Fig. 6) by ASTRÉE produces no false alarm:

```
% astree --exec-fn main clip.c |& grep WARN
%
```

This precision is obtained after automatic inclusion of partitioning directives by a preliminary analysis [48]. The result of such inclusion on the clip program is shown on Fig. 7. The effect of these directives is to distinguish, when analysing the second test, traces that come from the `then` and from the implicit `else` branch of the first test. Without the partitioning directives, the abstractions used by ASTRÉE would not be expressive enough to capture the weakest invariant necessary to make the partial correctness proof (including proving the user-supplied assertion that `m<=y<=M`):

```
% cat -n boolean.c
     1 typedef enum {F=0,T=1} BOOL;
     2 void main () {
     3    BOOL B; unsigned int X, Y;
     4    __ASTREE_known_fact((X < 256));
     5    while (1) {
     6       B = (X == 0);
     7       /* ... */
     8       if (!B) { Y = 1 / X; }
     9    }
    10 }
% astree --exec-fn main boolean.c --no-relational --unroll 0
|& grep WARN | fmt -s -w 62
boolean.c:8.20-25::[call#main@2:]: WARN: integer division by
zero [0, 255]
```

Fig. 8 Boolean control

```
% astree --exec-fn main clip.c --no-partition |& grep WARN
clip.c:11.19-35::[call#main@8:]: WARN: assert failure
%
```

3.2 Decision trees

Trace partitioning is used for control and data case analysis and applies equally to all abstract domains in the abstract interpreter. It is therefore not possible to perform a case analysis only on part of the abstract invariant handled by the abstract domains (e.g. only for one variable or two). Moreover, the case analysis is performed uniformly along the paths until they are merged. So, if the cases need to be distinguished only at the beginning of the path (where e.g. a variable is assigned) and at the end of the path (where it is used), it has to be propagated along paths in between (where e.g. the variable is never redefined, used or modified), which can be costly for very long paths. An example is the analysis of the program of Fig. 8 which succeeds using decision trees:

```
% astree --exec-fn main boolean.c --unroll 0 |& grep WARN |
fmt -s -w 62
%
```

and equally well using value partitioning (Fig. 9) but for the fact that, if the code `/*...*/` is very long, it will be partitioned unnecessarily, which is costly since the partitioning is on the whole abstract state.

Decision trees are an efficient implementation of the reduced cardinal product [16, Sect. 10.2] thus allowing expressing disjunctions on values v of variables $\mathtt{x}_1, \dots, \mathtt{x}_n$. Suppose that each variable $\mathtt{x}_i$ takes its values in a set D_i, then we can express $\bigvee_{v_1 \in D_1} \cdots \bigvee_{v_n \in D_n} (\bigwedge_{i=1}^{n} \mathtt{x}_i = v_i \wedge P^{\sharp}_{v_1,\dots,v_n})$. It is represented as a Shannon tree on $v_1, \dots, v_n$ with abstract domain information $P^{\sharp}_{v_1,\dots,v_n}$ at the leaves and opportunistic sharing like in BDDs [5]. It is an abstract domain functor in that the abstract domains of $P^{\sharp}_{v_1,\dots,v_n}$ at the leaves are parameters of the analyzer chosen at build time. It would be extremely expensive to partition on all variables so it must be restricted to few variables with few values. The automatic decision of where to use decision trees and on which variables is automated in ASTRÉE so

```
% cat -n boolean-partition.c
     typedef enum {F=0,T=1} BOOL;
     void main () {
        BOOL B; unsigned int X, Y;
        __ASTREE_known_fact((X < 256));
        while (1) {
           __ASTREE_partition_begin((X));
           B = (X == 0);
           /* ... */
           if (!B) {
              Y = 1 / X;
           }
           __ASTREE_partition_merge(());
        }
     }
% astree --exec-fn main boolean-partition.c --no-relational |&
grep WARN
```

Fig. 9 Analysis of boolean control with value partitioning

as to bridle its cost (thanks to automatic variable packing (Sect. 4.3) and parameterizable limitation on the tree height).

In conclusion, trace partitioning is a disjunction on control/values involving the whole abstract state so its use should remain local in the program so as to avoid very long program paths. In contrast, a decision tree is a disjunction on part of the abstract state. To scale up, it must be limited to a small part of the abstract state (e.g. a few values of a few variables). It is therefore less expressive than trace partitioning but does scale up on large parts of the program along very long program paths.

3.3 Widenings

For infinitary abstractions, a widening is necessary to pass to the limit of the iterates [16], which is at the origin of a loss of disjunctive information since the least upper bound of two elements is always more precise than their widening. This can be avoided when the fixpoint solution can be computed directly (for instance when we know that an arithmetic-geometric sequence such as $x = \alpha \cdot x + b$ is iterated). But such a direct computation requires either a global knowledge of the system (such as a set of semantic equations e.g. see Sect. 5.1.2), or an accurate analysis to extract dependencies between variables: on the one hand when iterating an arithmetic-geometric sequence $x = \alpha \cdot x + b$, we have to prove that the variable x is not otherwise updated between consecutive iterations; on the other hand, in order to compute the stable limit of a filter [28], we need to prove that the bounds on filter inputs will not increase during further iterations. Lastly, the direct computation of fixpoints would require the computation of complex relational invariants. As a consequence, it would be difficult to compute fixpoints directly without losing scalability.

In general, a fixpoint iteration with convergence acceleration by widening is necessary to ensure termination. The loss of disjunctive information can be limited both by adjusting the frequency of the widenings (see Sect. 5.2) and by designing widenings to be refinable e.g. by enriching the widening thresholds [14]. In ASTRÉE the default thresholds for integer interval widenings are 0, 1, and the biggest value of all signed and unsigned integer C types. In many cases, this allows the analysis to show the absence of modulo in integer arithmetic. We can increase the precision by adding more thresholds and the performance by suppressing useless thresholds. This is done through options in the analyzer.

As explained in the previous paragraph, we cannot compute directly fixpoints without losing scalability. However, in some situations, we are able to guess a potential bound of a fixpoint, but we cannot afford the computation of the global properties that would entail the fact that this is actually a sound bound (it might even happen that our guess is wrong and the bound is not sound). In such a case, we modify the widening thresholds dynamically, so that the widened iterates can jump directly to the bound that has been guessed. Then, it is up to the iterator to prove the stability of the bound: in case of failure, the analysis keeps on iterating with higher values until a fixpoint is reached. Special care must be taken in order to ensure the termination and efficiency of this process: in order to ensure that we never insert an infinite (or very large) number of thresholds, the computation of the dynamic thresholds to be inserted at each iteration is itself subject to widening. This strategy is widely used when analyzing filters [28, Sect. 5].

As a conclusion, the use of thresholds that are dynamically computed thanks to (potentially unsound) heuristics (controlled by a widening) is a light-weight alternative to the direct computation of fixpoints.

4 Locality versus uniformity

The design of ASTRÉE follows a locality principle. Only the information necessary at a given program point for a given program data should be computed and stored as opposed to using a uniform representation at each program point for all program data of the information statically collected.

4.1 Local versus global information

In proof methods (like Floyd-Naur-Hoare invariance proofs [17, 32, 43, 55]) and classic static analysis methods (like compiler dataflow analysis [42], or in simple abstract interpreters [12]), the properties (like invariants, bit-vectors, or abstract environments) attached to program points are chosen to be essentially of the same nature at each program point. So, the choice of which information should be attached to program points is made globally and is identical for each program point (e.g. an abstraction of a set of call stacks and memory states reachable at that program point). Formally, let $p \in \mathcal{P}$ be the set of program points (in a loose sense, this might include the full calling context). Let $\mathcal{D}$ be the global abstract domain (e.g. a predicate on visible variables, a bit-vector or a Cartesian abstraction). The uniform choice of the abstraction leads to the same abstraction attached to each program point: $\mathcal{P} \to \mathcal{D}$ (for example a local invariant on visible variables attached to each program point or a bit-vector for variables attached to program blocks). This choice is simple but hardly scales up because a little part of the information is needed locally but, more importantly, a different kind of information is in general needed at different program points. When the same abstract domain $\mathcal{D}$ is used everywhere, this means that either a universal representation of the abstract information is used (like predicates, bit-vectors or BDDs) or that the abstract domain $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ is the composition of various abstract domains $\mathcal{D}_1, \ldots, \mathcal{D}_n$ using different information encodings. In the case of a universal encoding of the information in $\mathcal{D}$, one cannot benefit from efficient algorithms for transformers since such algorithms are often based on tricky specific representations of the information (e.g. BDDs [5] may not be the best representation of numerical intervals). In the case of multi-representations, the information in $\mathcal{D}_1, \ldots, \mathcal{D}_n$ is represented everywhere whereas it is needed only locally, at some specific program points. The design of ASTRÉE mixes global information (like intervals for numerical variables attached to each program point but propagated locally, see Sect. 6.2) and local information attached to program points when needed (see e.g. Sect. 6.3).

4.2 Functional versus imperative data-structures

So far, we have discussed only semantic choices and not algorithmic ones. The main reason is that, in its design, each abstract domain comes fully equipped with its specialized data-structures and algorithms optimized for best efficiency, and so, we refer the reader to articles describing each one of them in details (e.g., [52] for the octagon domain). There is however one particular data-structure of pervasive use in ASTRÉE that we wish to discuss here: binary balanced trees with sharing.

Consider, for instance, the cheapest abstract domain used in ASTRÉE: intervals. It is generally described as having a "linear cost" in the number of variables, a guarantee of its scalability. Actually, this denotes the memory cost per abstract state as well as the worst-case time cost per abstract operation. The cost of an actual analysis is more subtle and a careful choice of data-structures and algorithms is required to actually scale up to tens of thousands of variables. To represent an environment, one needs a data-structure akin an associative array supporting the following operations:

1. retrieve and change the interval associated to a variable;
2. add or remove variables (preferably in arbitrary order);
3. copy an environment;
4. apply a binary function or predicate point-wise to two environments.

Operation 3 is used when the analysis performs case analysis (due to control-flow split, or trace or state partitioning). Operation 4 is used for control-flow join, widening, and inclusion testing. Using a naive array-based or hash-table-based approach leads to a unit cost for operations 1 and 2, but linear cost for operations 3 and 4. In programs where the numbers of both variables and control-flow joins are linear in the program size, one iteration of the interval domain on the whole code actually incurs a *quadratic* cost.

To improve the efficiency, a crucial observation is that all operators used point-wisely are idempotent ($f(a,a)=a$) and often applied to environments that are very similar (e.g., a control-flow join occurring a few statements only after the corresponding split). Our efficiency problem can thus be solved using *binary balanced trees*. These serve as functional associative arrays, and thus, the copy operation is free. Moreover, given two copies of the same initial array, both changed in small ways, most of their structure is shared in memory (intuitively, only the nodes along paths leading to updated variables are fresh, and they point mostly to subtrees of the original tree). We have designed binary operators on balanced binary trees that apply a binary function to all non-equal elements, ignoring subtrees with the same memory address. Its cost is in $r \times \log n$, where n is the total number of variables and r is the number of changed variables since the common ancestors of both arguments. Although the cost of element retrieval, modification, addition, and deletion is increased from constant to logarithmic, this is far out-weighed by the improved cost of the copy and binary operations.

Another benefit of subtree-sharing is the improved memory footprint. Many environments are transient: they are created at control-flow splits only to be merged a few statements later. These do not occupy much memory as most of the data-structure remains shared. Note that we solely rely on subtree-sharing that *naturally* arise from the functional nature of tree algorithms, but we do not enforce *maximal sharing* (as often used in BDDs [5]).

```
% cat -n octagon-packs.c
     1 void main() { int A, B, X, Y;
     2    if (A > B) { while (A <= 0) { A++; B++; }; };
     3    if (X < Y) { while (X >= 0) { X--; Y--; }; };
     4    __ASTREE_log_vars((A,X));
     5 }
% astree --exec-fn main octagon-packs.c --print-packs |&
grep_from_until.pl " octagon-packs.c@" "]" | sed -e"/\//d" -e/^\$/d
  octagon-packs.c@2@2 { A B }
  octagon-packs.c@1@17 { X Y }
   octagon-packs.c@2@2 =
   { -2.1474836e+09 <= B <= 2.1474836e+09,
     -2.1474836e+09 <= A <= 2.1474836e+09,
     -4.2949673e+09 <= A-B <= 4.2949673e+09,
     -4.2949673e+09 <= A+B <= 4.2949673e+09
   },
   octagon-packs.c@1@17 =
   { -2.1474836e+09 <= Y <= 2.1474836e+09,
     -2.1474836e+09 <= X <= 2.1474836e+09,
     -4.2949673e+09 <= X-Y <= 4.2949673e+09,
     -4.2949673e+09 <= X+Y <= 4.2949673e+09
   } >
```

Fig. 10 Octagon packs

Binary balanced trees with sharing are used in many other abstract domains in ASTRÉE besides intervals: other non-relational domains (such as congruences), but also domains of relational properties attached to variables (see, e.g., Sect. 6.4) or domains with packing (see Sect. 4.3).

4.3 Packed versus global relationality

Relational domains are critical to prove the absence of run-time errors (see, e.g., Sect. 6.3) but are, unfortunately, very costly. For instance, even limited linear inequalities such as octagons [52] have a quadratic memory and cubic time worst-case cost and these costs are often encountered in practice as the existence of bounds on the sum and difference of any two variables is very likely, whether the variables are actually logically related to each other or not. Although most relational information is useless or even redundant (consider, e.g., bounds on variable sums which are sums of variable bounds), redundancy removal techniques do not help scaling up as they are also costly (e.g. cubic for octagons [1]). No domain with a supra-linear cost is likely to scale up to tens of thousands of variables.

In ASTRÉE, costly relational domains do not operate on the full set of variables, but only on small *packs* of variables. Relational domains such as octagons relate all variables in one pack but not variables in different packs, as shown in Fig. 10. Non-relational information (such as variable bounds) can still be exchanged between packs through variables appearing in several packs or through reductions with other domains (Sect. 7).

Our packing heuristics are rather simple syntax-directed analyses. For octagons, they gather variables that are used together in linear expressions, as well as variables that are incremented or decremented within loops. We perform a transitive-closure of linear dependencies but, to avoid snowball effects resulting in all variables put in the same octagon, the closure is stopped at syntactic block boundaries (in particular, it is intra-procedural and we do not relate together formal and actual function arguments). Statistics output at the end of this pre-processing hint at whether the subsequent analysis may fail to scale up due to the octagon domain, and why. Experimentally, using a strategy adapted to the structure of our analyzed codes (adjustment being necessary only when the code generator is substantially changed), the number of packs is linear in the code size, while their size is constant. Moreover, experimentally, a variable appears in at most three packs, which limits the number of packs updated at each statement. The resulting analysis exhibits a linear cost.

Note that our packing strategy is local, as it is statement-based, but the subsequent octagon analysis is global: relational information on variables in a pack are also tracked outside the block statement that hinted at the pack creation. We experimented with local octagon analyses, using our packing strategy to create and destroy octagons according to scope, but it was not retained as it was more complex but did not actually impact the precision nor the efficiency. A more promising research direction towards more dynamic packing strategy would be to monitor the information requested by other domains through reduction and respond to their needs with the creation of new packs (Sect. 7).

5 Abstract interpreter

5.1 Fixpoint iterator

Static analyses reduce to computing fixpoint approximations of dynamic systems [15], and they differ in the way the system is posed, the way it is solved and the algebraic domain in consideration.

5.1.1 Global versus separate analysis

A first choice is whether our domain represents states (or traces) or state (or trace) transformers. The later allows modular analyses [20] which feature, in theory, improved efficiency by avoiding recomputing the effect of reused procedures, especially when the analyzed program is written in a modular fashion (that is, features natural boundaries at which simple invariants can be computed to account for the effect of large amounts of encapsulated code). Unfortunately, this is not the case for the synchronous control-command programs targeted by ASTRÉE. Indeed, they are composed of a (linear in the code size) number of instances of a small (a few dozens) set of small (generally less than 10 lines) C macros that are scattered in long linear sequences by the scheduler of the code-generator. No higher-level modularity is visible and exploitable, and so, a modular analysis is not likely to offer improved efficiency. On the contrary, modular analyses require the inference of much more complex properties (such as relationality) which puts a huge strain on the abstract domains. Thus, ASTRÉE abstracts traces and not transformers. Moreover, function calls are completely *inlined*, resulting in a full context-sensitive analysis. An abstract control point is thus a program control point together with the full stack of function calls, from the main procedure up to the current function. We believe that the cost of reanalyzing some code is largely compensated by the comparatively simpler (and cheaper) abstractions sufficient to abstract traces precisely.

5.1.2 Control flow driven interpreter versus abstract equation solver

Another choice is the way semantic equations are solved to reach a (approximation of a) fixpoint. Classic frameworks assume that all equations and the current abstract value associated to all abstract control points reside in memory so that a wide range of chaotic iteration strategies can be deployed (see [4]), with the idea that a global view permits improved performance and/or precision (up to finding an exact solution in some very restricted cases [62]). This approach is not scalable to a large number of abstract control points and large abstract state spaces. In ASTRÉE, we chose instead to closely follow the control-flow of the analyzed program, so that the analyzer is very similar to an actual interpreter. This solution is very memory efficient as, apart from the abstract value associated to the concrete control point currently executed, the analyzer must only remember one abstract value for each nested loop (to perform extrapolation) and nested `if-then-else` instruction (to perform control-flow join at end of both branches) that syntactically encompasses the control point. It is equivalent to one particular flavor of chaotic iterations where:

- widening points are loop heads,
- the iteration strategy is *recursive*; in case of imbricated loops, inner loops are stabilized first, for each iteration of the outer ones.

This corresponds to the strategy advocated in [4].

5.2 Widening frequencies

Widening after each iterate may be a cause of unrecoverable loss of information. Let us consider the following program, for instance:

```
% cat widen.c
typedef enum {F=0,T=1} BOOL;
float X, Y;
int main () {
  X = 0 ; Y = 0 ;
  while (1) {
    BOOL B;
    if (B) {X = Y + 2;};
    Y = 0.99 * X + 3;}
}
%
```

In this example, the boolean `B` switches between two iteration modes: in the first mode, the variable `X` is first updated with the value of the expression `Y + 2`, then the variable `Y` is updated with the value of the expression `0.99 * X + 3`; in the second mode, the variable `X` remains unchanged whereas the variable `Y` is updated with the value of the expression `0.99 * X + 3`. At run-time, the range of the variable `X` is $[0, 500]$ and the range of the variable `Y` is $[0, 498]$. Nevertheless, an iteration with widening thresholds when widening is applied at each iteration fails to bound variables `X` and `Y`. Indeed, if at each iteration we widen both `X` and `Y`, when computing odd iterates, the variable `Y` is unstable because the bound of the variable `X` has changed, whereas when computing even iterates, the variable `X` is unstable because the bound of the variable `Y` has changed. We give in Fig. 11(a) the values for the widened iterates having powers of 10 as thresholds. Iterations stop with the $[0, +\infty[$ interval.

A first attempt to solve this problem (e.g. see [3, Sect. 7.1.3]) consisted in not applying the widenings when at least one variable has become stable during the iteration. This is a global method where the widening of a given variable always depends on the iterates of all variables. We also added a fairness condition to avoid livelocks in cases where, at each iteration, there exists a variable that becomes stable while not all variables are stable. We give in Fig. 11(b) the values for the widened iterates using the same widening thresholds (i.e. the

Fig. 11 Abstract iterates for the three widening methods

iterate	X	Y
0	$[0,0]$	$[0,0]$
1	$[0,10]$	$[0,10]$
2	$[0,10^2]$	$[0,10^2]$
3	$[0,10^3]$	$[0,10^3]$
4	$[0,10^4]$	$[0,10^3]$
5	$[0,10^4]$	$[0,10^3]$
6	$[0,10^5]$	$[0,10^4]$
$\vdots$	$\vdots$	$\vdots$
$(2\cdot n)+4$	$[0,10^{4+n}]$	$[0,10^{3+n}]$
$(2\cdot n+1)+4$	$[0,10^{4+n}]$	$[0,10^{4+n}]$
$\vdots$	$\vdots$	$\vdots$
	$[0,+\infty[$	$[0,+\infty[$

(a) First method: using widening after each step.

iterate	X	Y
0	$[0,0]$	$[0,0]$
1	$[0,10]$	$[0,10]$
2	$[0,100]$	$[0,100]$
3	$[0,1000]$	$[0,1000]$
4	$[0,1002]$	$[0,1000]$
5	$[0,1002]$	$[0,1000]$
6	$[0,1002]$	$[0,1000]$

(b) Second method: delaying widening when one variable is stable.

iterate	X		Y	
	range	freshness	range	freshness
0	$[0,0]$	2	$[0,0]$	2
1	$[0,2]$	1	$[0,4.98]$	1
2	$[0,10]$	2	$[0,10]$	2
3	$[0,12]$	1	$[0,14.88]$	1
4	$[0,100]$	2	$[0,100]$	2
5	$[0,102]$	1	$[0,103.98]$	1
6	$[0,1000]$	2	$[0,1000]$	2
7	$[0,1002]$	1	$[0,1000]$	2
8	$[0,1002]$	1	$[0,1000]$	2

(c) Third method: using freshness counters.

powers of 10). At iteration 4, the range of the variable Y is stable, so, we do not widen X. Then the iteration converges at iterate 4 when the range of the variable X is [0, 1002] and the range of the variable Y is [0, 1000]. While solving the accuracy problem for small applications, this solution has turned out to be not scalable for large programs. In practice, at each iteration, at least one variable becomes stable. As a result, the widening was only scheduled by the fairness condition.

We now use a local solution. Each piece of abstract information (such as an interval, an octagon, a filter predicate, etc.) is fitted with a freshness counter that allows regulating when this piece of information is widened. More precisely, when we compute the join between two iterations, the counter of each unstable piece of information is decremented. When it reaches zero, then the piece of information is widened, and the counter is reset. We give in Fig. 11(c) the values for the widened iterates using freshness counters initialized to 2 and the same widening thresholds (i.e. powers of 10). The iteration converges at iterate 7 when the range of the variable `X` is [0, 1002] and the range of the variable `Y` is [0, 1000].

During an analysis by ASTRÉE, the more a given piece of information has been widened, the smaller the value used to reset the freshness counter is. A fairness condition ensures that, after a certain time, counters are always reset to zero, that is, widening occurs in all subsequent iterations, which ensures the termination of the analysis.

The main advantage of freshness counters is that each piece of abstract information is dealt with separately and they are not necessarily widened at the same time. This is very important to analyze some variables that depend on each others.

The analysis by ASTRÉE of the above program is as follows:

```
% astree --exec-fn main widen.c --unroll 0 --dump-invariants
|& grep "X in" | fmt -s -w 62
direct = <float-interval: Y in [0., 994.98005], X in [0.,
1002.] >
%
```

5.3 Stabilization

We have also encountered another issue in the stabilization of increasing iterates. When considering a slow convergence (such as when iterating the assignment `X = 0.9 * X + 0.2`), the analyzer can linger just below the limit, not being able to jump above the limit without a widening step. However, perturbing the iterates by relaxing them by a small amount allows the analyzer to jump above the limit without any widening step.

The situation is exacerbated when the analyzer uses floating-point arithmetic to compute abstract transformers. Indeed, the iterates can be above a post-fixpoint, but not sufficiently above and, because of sound rounding errors in the analyzer, it is unable to prove the induction. To solve this problem, we use perturbation in increasing sequences in the computation of post-fixpoints. Since the perturbation is meant to counteract rounding error noise in the abstract computation, we use a perturbation that is tailored to be slightly bigger than the estimated abstract rounding errors. For most domains we use a perturbation that is proportional to the current value of the iteration. For octagons [52], the situation is more involved due to the shortest-path-closure algorithm in the reduction of octagons that propagates all constraints. Thus, we use a perturbation proportional to the biggest bounded entry in the octagon.

This heuristic has allowed us to significantly reduce the number of iterations during analyses.

5.4 Concurrent versus sequential analysis

Recently [54] a parallel implementation of ASTRÉE has been designed so that it can take advantage of multi-processor computers that are now mainstream.

Currently, we have only experimented with a coarse granularity parallelization of the iterator. As explained in Sect. 5.1, the iterator follows roughly the control-flow of the program. When it splits (due to a conditional or a call to a function pointer with several targets), ASTRÉE follows each branch in turn and then merges the results. Computations along the branches are completely independent, and so, can be easily parallelized. The kind of code currently analyzed by ASTRÉE presents an ideal situation as it is composed of a top-level loop that performs different large tasks at regular periods, and the actual order of task execution is not important when checking for run-time errors. We thus abstract the control of this loop as an execution of a random task at each iteration. Each task can be analyzed by a dedicated independent analysis process. As there are up to 16 tasks, parallelizing the top-level loop exhausts the number of cores in current computers and achieves the maximum amount of parallelization possible. Actually, we have observed that the use of more than four cores is not advisable as the cost of synchronization (communication and merge of invariants at the end of each top-level loop iteration) then out-weights the speedup of parallel execution.

Our solution can be extended to more general kinds of analyzed codes (i.e., without a top-level loop executing non-deterministic tasks) as every control-flow split can be parallelized. These splits generally have a very short duration, so, light-weight parallelization is advised (e.g., POSIX threads as opposed to processes). Unfortunately, ASTRÉE is programmed in OCaml [46], which is not well equipped in this department (mainly due to the garbage collector locking all threads), and thus, we currently use heavy-weight processes not suitable for fine-grained parallelization. Moreover, even this solution may not be sufficient to exploit the tens or hundreds of cores in future computers.

6 Abstract domains

The abstraction in ASTRÉE is implemented in the form of several abstract domains combined by reduction [16, Sect. 10.1]. The abstract domains are OCaml modules [46] that can be chosen by parameters and assembled statically. It means that many different combinations of abstract domains must be compiled to offer some flexibility in choosing the abstract domain for the end-user. But this design method has several advantages:

- Each abstract domain uses its own efficient algorithms for abstract operations operating on efficient specific computer representations of abstract properties. Such elaborated data representations and algorithms can hardly be automatically inferred by abstraction refinement [7, 24], a severe limitation of this approach.
- Being designed to scale up precisely for the family of synchronous control-command programs, ASTRÉE produces few or no false alarms for programs in this specific family (typically less than one alarm per 5000 lines of code, before tuning of the parameters). Occasionally, an adaptation of ASTRÉE parameters or analysis directives may be required. When extending the considered family of programs, it may also happen that the weakest inductive property necessary to prove the specification may not be expressible with the currently available abstraction. Examples we encountered include filters [28] and arithmetic-geometric sequences [29] which cannot be expressed using linear numerical abstractions. On such rare occasions, the abstraction must be refined. This consists in adding a new abstract domain and its interaction with the existing abstract domains [25].
- The specification of the abstraction as the reduction of several abstract domains [16, Sect. 10.1] divides the design and programming of the abstract interpreter into independent tasks which are specified by the interface of abstract domains, including reductions, and by the concretization operator for that abstraction. The complexity of the programming task is thus significantly reduced.

We first discuss the memory abstraction mapping the concrete memory model to the abstract memory model used by abstract domains. Then we discuss some important choices regarding numerical abstractions. The combination of abstract domains by reduction is discussed in Sect. 7 while the refinements of abstractions is considered in Sect. 8.

6.1 Memory abstraction

6.1.1 *Points-to analysis versus shape analysis*

Mission-critical synchronous control-command programs targeted by ASTRÉE feature very simple data-structures: mostly scalar numerical or boolean variables and statically allocated arrays or structures of constant size. No recursive nor dynamically allocated data-structures are used. This greatly simplifies the design of ASTRÉE and avoids the need for memory shape abstractions (such as [59]) that do not currently scale up well. In particular, due to our full context-sensitive control abstraction (Sect. 5.1), the set of existing memory locations at each abstract control point can be determined statically: it is the set of global variables and local variables of all functions in the call stack.

Our programs feature pointers, but these are used only to model passing-by-reference and encode array accesses through pointer arithmetic. We thus use a straightforward scalable points-to analysis: the (often singleton) set of pointed-to variables is represented explicitly, while the byte-offset from the starting byte of the variable is abstracted as an integer variable would.

6.1.2 *Combined versus separate pointer and value analyses*

Many authors only consider value analyses on pointer-free programs and rely on a prior alias analysis to remove the use of pointers. However, there is experimental proof [56] that combined pointer-value analyses are more precise. This is especially the case for languages such as C that feature pointer arithmetic, with expressions mixing pointers and numbers. ASTRÉE performs such a combined pointer-value analysis. In particular, the offset abstraction benefits from all the (possibly relational) numerical domains available for integers, trace partitioning, etc. This is key in proving the absence of out-of-bound array accesses encoded through pointer arithmetic, as in the example of Fig. 12. Note the abstraction of the pointer `p` as a symbolic base `base(p)` and a numerical offset `off(p)`. The information that `off(p)` is less than `i`, provided by the octagon domain, allows proving that there is not access outside `a` in the loop. Note that this example would not work with a pointer to `int` (or, indeed, any type whose byte-size is greater than one) as it would require an integer information of the form $\texttt{off(p)} \leqslant 4 \times \texttt{i}$, which is currently outside the scope of ASTRÉE.

6.1.3 *Field-sensitive versus field-insensitive analysis*

To perform a value analysis in the complex C memory model, we need a map from pointer values (variable/offset pairs) to dimensions in value (e.g. numerical) abstract domains, so called *cells*. ASTRÉE uses a mostly field-sensitive model, where each scalar component in a structure or array is associated its own cell. However, to scale up, it is necessary to abstract large arrays in a field-insensitive way, where a single cell accounts for the whole contents of the array. Indeed, this allows a unit cost for abstract operations, even when an imprecision in an index causes a large portion of the array to be potentially read or updated. Although less precise than a fully field-sensitive analysis (we lose the relationship between indexes and values), this is sufficient to analyze all large arrays in our programs as they mainly model homogeneous sequences of data.

Recently [51] the concrete memory semantics and its abstraction have been extended to support union types, pointer casts, as well as platform-specific ill-typed memory accesses (such as reading individual bytes in the byte representation of an integer or a float). This has

```
% cat -n memory.c
     1 void main() { char a[100], *p; int i;
     2   for (i=0,p=&a[0];i<100;i++) {
     3     if (*p) { *p = 1; p++; }
     4   }
     5   __ASTREE_log_vars((p));
     6 }
% astree --exec-fn main memory.c |& egrep -A2
"(call#main@1|memory.c@1@18|WARN)" | fmt -s -w 62
call#main@1: direct = <pointer: base(p) = { a } >
  <integers (intv+cong+bitfield+set): off(p) in [0, 100] >
--
   memory.c@1@18 = { 100 <= i <= 101, 0 <= off(p) <= 101, -101
   <= off(p)-i <= 0,
     100 <= off(p)+i <= 202
```

Fig. 12 Example with pointer arithmetic

```
% cat -n union.c
     1 typedef union { int x; char b[4]; } u;
     2 void main() { u a;
     3   a.b[0] = 0; a.b[1] = 0; a.b[2] = 2; a.b[3] = 1;
     4   __ASTREE_log_vars((a));
     5 }
% astree --exec-fn main union.c |& grep -A1 "<integers" | fmt
-s -w 62
  <integers (intv+cong+bitfield+set): a.b[3] in {1}, a.b[2] in
  {2},
   a.b[1] in {0}, a.x in {513}, a.b[0] in {0} >
```

Fig. 13 Example with union type

been done without changing the main assumption that the memory can be represented as a finite set of abstract cells, each accounting for a fixed number of concrete memory locations. However, in this new model, the mapping is dynamic: as we cannot rely on misleading static C types (they do not account for dynamic casts), the mapping is updated according to the actual access pattern during the analysis. A second difference is that cells can now represent overlapping byte segments, which requires careful handling of updates. As each memory byte can be abstracted by only finitely many cells (as many as atomic C types but, in practice, a single one often suffices), the scalability of the analysis is not affected. Consider the example of Fig. 13. Indeed, because ASTRÉE is aware of the big endian, 2's complement bit representation of integers used in PowerPC processors, it discovers that the byte sequence 0 0 2 1 (field `a.b`) corresponds to the integer 513 (field `a.x`). For performance reasons, this information is only inferred in some cases (such as when converting a byte sequence to an integer) but not necessarily in others (e.g., converting back an integer to a byte sequence) if it was not deemed useful to prove the absence of run-time errors.

6.2 Domain of observable

A core domain in ASTRÉE is the interval domain [14] that infers bounds on variables and expressions. It allows expressing the minimal information requested for checking the implicit specification. Indeed, bound properties are crucial as they can express the absence of run-time errors for most atomic language operations (absence of arithmetic overflows, conversion overflows, out-of-bound array accesses, square root of negative numbers, etc.), i.e., they are the main observable properties for our implicit specification of programs. Additionally, bound properties are useful to parameterize more elaborate abstractions (Sects. 6.3, 6.5).

Although several complex abstract domains used in ASTRÉE are able to express bounds (e.g., octagons in Sect. 6.3), we still keep bound information for all variables in a dedicated interval abstract domain for two reasons. Firstly, this ensures that some bound information is always available even when other domains are turned off globally or locally (in some program parts or for some variables) for performance reasons. Secondly, the interval domain is quite simple and well understood so that it is easy to design precise abstractions for all language operators, including complex non-linear ones (such as integer bit-operations or floating-point arithmetic). This is not the case for more complex domains (such as octagons) that, although more expressive in theory, can only handle precisely a few arithmetic operators, and so, compute bounds that may or may not be tighter than the plain interval domain in other cases. To get the best of all domains, variable bounds are computed in parallel in them, and then exchanged through reduction (see Sect. 7).

To supplement the interval domain, ASTRÉE also features a congruence domain [38] that associates a congruence information $V \equiv a\,[b]$ to each integer variable and pointer offset V. This is necessary to prove the absence of mis-aligned pointer dereferences, another observable property.

6.3 Specialized versus general inequality domains

If intervals and congruences are sufficient to *express* the absence of run-time errors, they are often insufficient to *prove* it, i.e., provide local and/or inductive invariants that imply the sufficient bounds. Linear inequalities have been recognized early [21] as a class of useful invariants for proving correctness. They can express, for instance, the relational loop invariant linking `i` and `j` in code of the form shown in Fig. 14, which is key in proving that `j++` never performs an overflow.

Another feature is their ability to gather relational information from tests and exploit them in subsequent assignments, for instance in the code of Fig. 15 implementing a rate limiter, where linear invariants are necessary to bound `OUT` and prove that no overflow can occur.

General polyhedra [21] unfortunately suffer from a very high complexity (exponential at worse), exacerbated by the need to resort to arbitrary precision arithmetic to guarantee the soundness of the algorithms. Thus, ASTRÉE employs octagons [52] instead, which allow discovering restricted forms of linear constraints on variables: $\pm X \pm Y \leqslant c$. They feature a lower and *predictable* quadratic-memory and cubic-time complexity, as well as a fast yet sound floating-point implementation [44]. Their limited expressiveness is still sufficient in most cases: they find the same loop invariant as polyhedra in the first program, while they find a less precise bound for `OUT` (due to their inability to model exactly assignments involving three variables) which is still sufficient to prove the absence of overflow.

6.4 Symbolic constants versus domain completion

Most abstract domains in ASTRÉE are specialized for a specific kind of properties and, as a result, can only handle precisely specific statements and are sensitive to even the simplest

```
void main() {
  int i,j;
  j = 5;
  for (i=0;i<100;i++) {
    __ASTREE_log_vars((i,j));
    int random;
    if (random) j++;
  }
}
% astree --exec-fn main --unroll 0 oct.c | & grep -A1
oct.c@2@5 | tail -n2 | fmt -s -w 62
   oct.c@2@5 = { 5 <= j <= 104, 0 <= i <= 99, -5 <= i-j <= 94,
   5 <= i+j <= 203 } >
```

Fig. 14 Octagon usage

```
void main() {
  float OUT, LAST_OUT;
  OUT = 0;
  while (1) {
    float IN, MAX_SLOPE, SLOPE;
    __ASTREE_known_fact((IN>=-128 && IN<=128));
    __ASTREE_known_fact((MAX_SLOPE>=0 && MAX_SLOPE<=16));
    LAST_OUT = OUT;
    OUT = IN;
    SLOPE = OUT - LAST_OUT;
    if (SLOPE < - MAX_SLOPE) OUT = LAST_OUT - MAX_SLOPE;
    if (SLOPE >   MAX_SLOPE) OUT = LAST_OUT + MAX_SLOPE;
  }
}
```

Fig. 15 Rate limiter

program transformations. Consider, for instance, an abstract domain for linear interpolation. Given an assignment `Y=B+(A-B)*T` and the knowledge that `T` $\in [0, 1]$, it infers that `Y` is a linear combination of `A` and `B`, and thus, that the range of `Y` is the join of that of `A` and `B` (which is outside the scope of the interval domain due to the relationality of the expression as `B` appears twice). Now, consider the following code: `X=(1-T)*B; ...; Y=A*T+X` (where `...` denotes an arbitrary long piece of code that does not modify `A`, `B`, `T`, nor `X`). Our domain will fail as neither assignment is a linear interpolation between two variables. A common solution to this problem is domain completion [16, 34], that is, enriching domains with properties holding at intermediate statements. Unfortunately, managing a richer set of properties may cause scalability issues.

As an alternate, lightweight solution, we have designed in ASTRÉE a special-purpose symbolic constant propagation domain [53]. This domain remembers which expression is assigned to which variable (e.g., `X` maps to `(1-T)*B`) and is able to substitute on demand variables with the corresponding expression in subsequent assignments and tests (e.g., `Y=A*T+X` becomes `Y=A*T+(1-T)*B`). It was indeed observed that most intermediate properties are only required due to the fine granularity of abstract transfer functions, but their need disappears once we take past assignments into account and derive more synthetic transfer functions. Consider the analysis of the program in Fig. 16, with only the interval do-

```
% cat -n symb.c
     1 void main () { short int x; int y, z, k;
     2   y = x + 1; z = x - 1; k = y - z;
     3   __ASTREE_log_vars((y,z,k));
     4   __ASTREE_assert((k == 2));
     5 }
% astree symb.c --no-symb --exec-fn main --no-octagon
--no-clock --no-dev-geo |& grep WARN
symb.c:4.19-25::[call#main@1:]: WARN: assert failure
```

Fig. 16 No symbolic propagation

main and no symbolic evaluation. There is a false alarm on the assertion `(k == 2)` which is removed with the symbolic evaluation of the expression `y - z`:

`y - z`:

```
% astree symb.c --symb --symb-max-depth 2 --exec-fn main
--no-octagon --no-clock --no-dev-geo |& egrep "(symbolic|WARN| in
\[)"
  <integers (intv+cong+bitfield+set): y in [-32767, 32768],
   z in [-32769, 32766], k in {2} >
  <symbolic: y = (x +i 1), z = (x -i 1), k = (y -i z) >
%
```

Another difference between our symbolic domain and more traditional relational domains is that the symbolic domain does not represent general equations that can be combined, solved, etc., but only substitutions (that is, directed equations) that can be applied to expressions. Its implementation is akin that of non-relational domains and enjoys its good scalability if one uses proper data-structures (Sect. 4.2) and takes care to bound the substitution depth to a reasonable value (which can be changed through a parameter `--symb-max-depth` which defaults to 20).

6.5 Floating-point error abstraction versus accumulation

Relational domains in ASTRÉE (such as octagons, see Fig. 15) are able to handle floating-point computations, although their algorithms are based on mathematical properties (such as linear arithmetic) that only hold on algebraic fields such as rationals or reals. We use the method of [49] to abstract floating-point expressions into ones on reals by abstracting rounding as a non-deterministic choice within a small interval. These expression manipulations are sound only in the absence of overflows and are parameterized by bounds on variables and expressions. Thus, we rely on the self-sufficient interval domain (Sect. 6.2) to provide coarse but sound initial bounds and bootstrap a refinement process that can then involve relational domains (Sect. 7).

Furthermore, when abstracting properties on floating-point numbers, all computations in our domains are also handled using floating-point arithmetic, rounded in a sound way, to ensure the scalability of the analysis.

In Fig. 17, the expression `z = x - (0.25 * x)` is linearized as

$$\mathtt{z} = ([0.749\cdots, 0.750\cdots] \times \mathtt{x}) + (2.35\cdots 10^{38} \times [-1, 1])$$

which takes rounding errors into account and allows some simplification even in the interval domain so that we get $|\mathtt{z}| \leqslant 0.750\cdots$ instead of $|\mathtt{z}| \leqslant 1.25\cdots$.

```
% cat -n linearization.c
     1 int main () { float x, z;
     2   __ASTREE_known_fact(((-1.0 <= x) && (x <= 1)));
     3   z = x - (0.25 * x);
     4   __ASTREE_log_vars((z));
     5 }
% astree --exec-fn main linearization.c |& grep float-interval
direct = <float-interval: z in [-0.75000009, 0.75000009] >
```

Fig. 17 Linearization

```
% cat -n congruence.c
     1 int main() { int X;
     2   X = 0;
     3   while (X <= 128) { X = X + 4; };
     4   __ASTREE_log_vars((X));
     5 }
% astree --exec-fn main congruence.c |& grep "X in"
direct = <integers (intv+cong+bitfield+set): X in {132} >
```

Fig. 18 Example using domain reduction

A key point in our method is the eagerness to abstract away rounding errors as soon as possible. This is unlike methods such as [35] which accumulate symbolically the effects of local rounding errors over the whole program. That method provides more information (e.g., the drift between a real and a floating-point computation across complex loops) but has difficulties scaling up due to the complex symbolic expressions involved [36]. The abstractions used in ASTRÉE scale up to large programs by keeping only the information relevant to the proof of absence of run-time errors.

7 Approximate reduction

Finding a single appropriate abstraction for a family of complex programs is an insurmountable intellectual task. The approach chosen in ASTRÉE is to use dozens of abstract domains which are relatively simple when taken separately but perfectly complement each other and which can be combined to achieve precision.

7.1 Reduced product

The reduced product of abstract domains [16, Sect. 10.1] is such a combination of abstractions. It is the most precise abstraction of the conjunction of the concrete properties expressed by each abstract domain which can be expressed using only these abstract domains. An example is displayed in Fig. 18. The interval analysis [14] determines that $\mathtt{X} \in [129, 132]$ and the congruence analysis [38] that $\mathtt{X} = 0 \bmod 4$ on loop exit. The reduction between these two abstract domains yields $\mathtt{X} = 132$.

The reduced cardinal product is a semantic notion, the definition of which refers to the concrete semantics. It can be approximated by taking the iterated fixpoint of the composition of lower closure operators performing the reduction between pairs of abstract domains [12, 39]. However, the iteration cost is high and the convergence is not guaranteed. It follows that it is not effectively computable for non-trivial programs and therefore must be approximated by enforcing the convergence by a narrowing [12].

We discuss the weaker but efficient approximate reduction adopted in ASTRÉE [25]. The static analyzer is extensible in that it is easy to add new abstract domains because all abstract domains share a common interface and a common reduction method.

7.2 Interactions between abstract domains

Abstract domains are implemented as independent modules that share a common interface. Any such module implements the usual predicate transformers (such as abstract assignments, abstract guards, control-flow joins) and extrapolation primitives (widening operators). Moreover, in order to enable the collaboration between domains, each abstract domain contains some optional primitives that enable the expression of properties about abstract pre-conditions and abstract post-conditions in a common format that can be understood by all other domains. For instance, the symbolic abstract domain provides the capability to linearize expressions (e.g. see Sect. 6.5), that is to replace an arbitrary expression with a linear expression with interval coefficients; the linearization requires bounds on the variables that occur in the expression (in order to take into account rounding errors and to replace some variables with their range in case of non-linearity). Another example is the symbolic constant domain (e.g. see Sect. 6.4) that provides the capability to substitute variables with the expression they have been assigned to last, in order to enable further simplification.

Abstract domains have two basic ways to interact with each others. Either a given abstract domain decides to propagate some properties to other domains (this is the case for instance when the filter domain [28] interprets the iteration of a filter, since it knows that the interval it has found is more accurate than the one that can be found by the other domains); or a given domain needs a particular kind of properties in order to make a precise computation, in which case the domain requests the other domains for such a property (this is the case when the filter domain detects a filter initialization and it requests information about the ranges of initial values of the filter). This two-way mechanism is detailed in [25].

This scheme of interactions between abstract domains has been chosen in order to avoid *a priori* limitations (such as deciding that relational domains always access already linearized expressions, which might no longer be true for future domains). Indeed, each domain has access to a hierarchy of interpretations for expressions. Each interpretation is sound and the algorithms of the transformers use the more adequate interpretation according to the abstract domain.

The resulting architecture [25] is highly extensible. It is very easy to add a new domain. We can also easily add a new capability for abstract domains: we only have to modify the implementation of abstract domains that may use this capability. Usually, we only modify few domains when we add a new capability. Moreover, in order to avoid recomputing the same information requested by various domains (such as the linearization of a given expression), we use information caches. This ensures the efficiency of the approach.

7.3 Interactions of reduction with widening

As explained in [25], in ASTRÉE we use reductions after widening steps in order to limit the loss of information. Namely, let us denote by $\triangledown$ the widening operator, by ρ the reduction operator, and by $F^\sharp$ the abstract transformer for a loop iteration. In ASTRÉE, we compute the sequence $X_0 = \bot$ and $X_{n+1} = \rho(X_n \triangledown [\rho \circ F^\sharp](X_n))$. Alternative methods exist, for instance, we can delay the application of the reduction operator until the first time the result of the widening is used as a precondition. Namely, we would obtain the following sequence: $Y_0 = \bot$ and $Y_{n+1} = Y_n \triangledown [\rho \circ F^\sharp \circ \rho](Y_n)$. Due to the non-monotonicity of the widening operator (there usually exists a, b, c such that $a \sqsubset b$ and $a \triangledown c \not\sqsubseteq b \triangledown c$), the accuracy of the two strategies cannot be universally compared.

We have chosen to use reductions after widening because, in general, some precision lost by widening in one abstract domain can be recovered by reduction with another abstract domain. However, it raises other issues. It is well-known that mixing widenings and reductions can prevent the convergence of the iteration sequence in some cases (e.g. see [50, Example 3.7.3, p. 99]). Intuitively, this is because the reduction and widening operators have contradicting purposes: the widening operator builds an induction that can be destroyed by the reduction operator. For that reason, we require extra properties [25] about our abstract domains, the reduction operator that is used after widening, and the widening operator. Namely, we require that (1) each abstract domain is a finite Cartesian product of components 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.), (2) the widening operator is defined component-wise from widening operators over the components (e.g. for intervals, independent widening on each bound), and that (3) the graph of reduction between components is acyclic (where $a \to b$ is an edge in the reduction graph whenever the outcome of the reduction on component b depends on the value of component a). This way, we avoid cyclic reductions and ensure that abstract constraints stabilize themselves progressively.

8 Refinement

Because the cost/precision trade-off for a family of very large programs can only be adjusted experimentally, easy refinement capability is a key to ultimately guarantee precision and scalability.

8.1 Shortcomings of abstraction refinement

One technique for tuning the precision/cost ratio of an abstraction is to start from a rough one and then refine it automatically until a specification can be proved. The refinement can be driven by the concrete collecting semantics using e.g. counter-example based abstractions [7] or fixpoint refinement [24] following from the notion of completion in abstract interpretation [16, 34]. The refinement can also be guided by the disjunctive completion [16, 31, 33] of the current abstraction.

Software verification by abstraction completion/refinement is not used in ASTRÉE because it faces serious problems:

- completion involves computations in the *infinite domain* of the concrete semantics (with undecidable implication) so refinement algorithms assuming a finite concrete domain [7, 24] are inapplicable;
- completion is an infinite iterative process (in general not convergent) which does not provide an easy way to *pass to the limit*;
- completion relies on sets of states or traces representations of abstract properties hence does not provide an effective *computer representation* of refined abstract properties;
- completion does not provide effective *algorithmic implementations* of the abstract domain transformers (but ineffective sets of states or traces transformers).

The alternative solution used in ASTRÉE to control the cost/precision ratio is:

- to use very precise and potentially costly abstract domains which expressiveness can be bridled or relaxed as necessary (by parameterization, widening tuning, and/or analysis directives);
- and, when necessary, to introduce new abstract domains (with reduction with the current abstraction).

8.2 Refinement by parameterization of abstract domains

ASTRÉE has very expressive relational abstract domains that may be necessary to express the complex properties that are required to make the correctness proof. Using the full expressive power of these complex abstract domains may not scale up. That is why the expressive power of these domains is deliberately constrained.

An example is the packing described in Sect. 4.3. The global packing strategy may be parameterized using command-line options, and influenced locally by analysis directives inserted by the end-user or automatically thanks to an automatic heuristic.

Examples of command-line options that can be used to restraint/refine the expressive power of abstract domains are:

- `--smash-threshold` n ($n = 400$ by default) fixes the array size limit above which arrays are abstracted in a field-insensitive way.
- The automatic packing of octagons can be controlled in several ways. For example `--fewer-oct` prevents the creation of packs at the top-level block of functions (packs are only created for inner blocks) while `--max-array-size-in-octagons` n indicates that elements of arrays abstracted in a field-sensitive way and of size greater than n should not be included in octagon packs.

8.3 Refinement by widening tuning

Another way of controlling the precision/coarseness of the analysis is to control widening. Beyond the choice of thresholds in widenings (see Sect. 3.3), it is possible to control their frequency (see Sect. 5.2). For example, `--forced-union-iterations-beginning` n delays the use of widening for the first n iterations where it is replaced by unions (which is less precise than unrolling, more precise than widening, but does not enforce termination). Lastly, the option `--fewer-widening-steps` n will use n times fewer widenings between the first loop unrolling and the iterations where widenings are always enforced. It is also possible to select the widening frequency on a per abstract domain basis.

8.4 Refinement by analysis directives

Analysis directives inserted manually or automatically in the program can locally refine an abstraction as shown e.g. in Sects. 3.1 and 4.3.

Packing can be specified by insertion of analysis directives in the source. For example, the analysis of Fig. 19 fails because no relation is established between `b` and `x`. Adding a packing directive for the boolean decision tree abstract domain as follows:

```
% diff -U1 repeat.c repeat-a.c
--- repeat.c 2008-08-03 15:04:53.000000000 +0200
+++ repeat-a.c 2008-08-03 15:04:31.000000000 +0200
@@ -2,2 +2,3 @@
 int main () { int x = 100; BOOL b = TRUE;
```

```
% cat -n repeat.c
     1 typedef enum {FALSE=0,TRUE=1} BOOL;
     2 int main () { int x = 100; BOOL b = TRUE;
     3   while (b) {
     4     x = x - 1;
     5     b = (x > 0);
     6   }
     7 }
% astree --exec-fn main --unroll 0 repeat.c |& grep WARN | fmt
-s -w 62
repeat.c:4.8-13::[call#main@2:]: WARN: signed int arithmetic
range [-2147483649, 2147483646] not included in [-2147483648,
2147483647]
```

Fig. 19 Missing boolean pack

```
+  __ASTREE_boolean_pack((b,x));
   while (b) {
```

solves the precision problem:

```
% astree --exec-fn main --unroll 0 repeat-a.c |& grep WARN |
fmt -s -w 62
%
```

ASTRÉE includes fast pre-analyses in order to automatically insert packing directives in the source (the automatic packing strategy could be easily extended to include the above case, if needed) as well as trace partitioning directives (Sect. 3.1).

8.5 Refinement by local improvements to the analyzer

Many transfer functions in ASTRÉE do not correspond to the best (i.e. sound and most precise) transformer either because such transformers do not exist (the abstraction is not a Galois connection), for efficiency of the functions, or to minimize the coding effort. This may be at the origin of false alarms where the precondition is precise enough but not the post-condition. In that case the abstract transformer may have to be manually refined in the abstract domain.

A similar situation has been observed for reduction.

Firstly, because the verification is done with the interval abstract domain, some information present in another abstract domain may not be reflected in the intervals, making the verification impossible in the abstract while it is feasible in the concrete. In this case, the interval must be reduced by this information present in the other abstract domain by refinement of the approximate reduction.

Secondly, an abstract transformer for an abstract domain may be imprecise although the use of information in another abstract domain could drastically improve precision. In this case, the abstract transformer may ask the other abstract domains for the required information using the interaction mechanism between abstract domains described in Sect. 7.2.

8.6 Refinement by addition of new abstract domains

In case no invariant sufficient to prove the specification can be expressed in the currently available combination of abstractions in their most refined form, false alarms cannot be avoided by any of the methods discussed previously. There is then no other solution than refining the abstraction by adding a new abstract domain. This cannot be done by the end-user but by the designers of the analysis or specialists with a deep understanding of its structure. For such specialists, this involves discovering the appropriate abstraction, devising global parameterization and local directives able to adjust its cost/precision trade-off, finding a representation of the abstract properties, designing abstract property transformers for all language primitives, widening operators, and reductions with other abstractions. Experimentation usually indicates how the parameters should be adjusted and how the insertion of local directives can be automated. Throughout the life of ASTRÉE, the situation has been encountered several times, which lead to the design of several new abstract domains, such as ellipsoids to handle digital filtering [30], exponentials for accumulation of small rounding errors [29], etc.

8.7 Choice of abstractions for cost/precision tuning

For a given family of programs, some domains must be enabled while some other domains that are not useful for this family should be disabled to save time and space by avoiding useless computation.

However, since the domains are interconnected, it might be difficult for the end-user to guess which domains can be safely disabled. For instance, a given domain may require a capability (such as the linearization) that is only provided by another domain. To solve this issue, we use a dependency graph between abstract domains, so that when an abstract domain is used, all the domains that are necessary to make the computation in this abstract domain are automatically included.

9 Conclusion

A frequent criticism of abstract interpretation-based static analysis is that "Due to the undecidability of static analysis problems, devising a procedure that does not produce spurious warnings and does not miss bugs is not possible" [27]. This was indeed the case for the first generation of industrial analyzers [45, 57] which could produce thousands of false alarms on programs of few hundred thousands lines. This difficulty has been surmounted by domain-specific static analyzers such as ASTRÉE [26, 60, 61] which produce few alarms on embedded synchronous control-command programs and can be easily refined to reach the no false alarm objective. This approach is sound, precise, and scales up, unlike shallow bug detection methods [8] ultimately not suitable for safety and mission critical applications where the quest for total correctness has definitely intensified [63]. In particular, and contrary to tests where it is not obvious to decide when to stop testing, sound static analyses guarantee that all bugs in a given well-specified category have been extirpated. As a consequence, software engineering methodology should evolve in the near future from the present-day process-based methodology controlling the design, coding, and testing processes to a product-based methodology incorporating a systematic control of the final software product by static analyzers.

The theory of abstract interpretation offers several keys for the design of abstract sound, precise, and scalable static analyzers. The key to scalability is, on the one hand, to use very effective abstract domains with efficient representations of abstract properties and quasi-linear cost abstract transformers and, on the other hand, an iterator acting locally and accelerating the convergence. The key to extreme precision is to design the abstractions and widenings for the considered domains of application together with a flexible extension mechanism to easily add new abstractions and their reductions with previous ones. The key to the successful design of a complex static analyzer is to exploit the modularity. Finally, the success comes from capable and determined end-users [26, 60, 61] able to understand the potential of software verification by abstract interpretation.

References

1. Bagnara R, Hill PM, Mazzi E, Zaffanella E (2005) Widening operators for weakly-relational numeric abstractions. In: Hankin C, Siveroni I (eds) Proc 12th int symp SAS '05, London, 7–9 Sep 2005. LNCS, vol 3672. Springer, Berlin, pp 3–18
2. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2002) Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen T, Schmidt DA, Sudborough IH (eds) The essence of computation: complexity, analysis, transformation. Essays dedicated to Neil D Jones. LNCS, vol 2566. Springer, Berlin, pp 85–108
3. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2003) A static analyzer for large safety-critical software. In: Proc ACM SIGPLAN '2003 conf PLDI, San Diego, 7–14 June 2003. ACM, New York, pp 196–207
4. Bourdoncle F (1993) Efficient chaotic iteration strategies with widenings. In: Bjørner D, Broy M, Pottosin IV (eds) Proc FMPA, Akademgorodok, Novosibirsk, 28 June–2 July 1993. LNCS, vol 735. Springer, Berlin, pp 128–141
5. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput C 35(8)
6. Burstall RM (1974) Program proving as hand simulation with a little induction. In: Rosenfeld JL (ed) Information Processing 74, Stockholm, Aug 5–10 1974. Proc IFIP congress, vol 74. North-Holland, Amsterdam, pp 308–312
7. Clarke EM, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Emerson EA, Sistla AP (eds) Proc 12th int conf CAV '00, Chicago, 15–19 Jul 2000. LNCS, vol 1855. Springer, Berlin, pp 154–169
8. Clarke EM, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Jensen K, Podelski A (eds) Proc 10th int conf TACAS '2004, Barcelona, 29 Mar–2 Apr 2004. LNCS, vol 2988. Springer, Berlin, pp 168–176
9. Colby C, Lee P (1996) Trace-based program analysis. In: 23rd POPL, St Petersburg Beach, 1996. ACM, New York, pp 195–207
10. Cousot P (1978) Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, 21 Mar 1978
11. Cousot P (1981) Semantic foundations of program analysis. In: Muchnick SS, Jones ND (eds) Program flow analysis: theory and applications. Prentice Hall, New York, pp 303–342. Chap 10
12. Cousot P (1999) The calculational design of a generic abstract interpreter. In: Broy M, Steinbrüggen R (eds) Calculational system design. NATO science series, series F: computer and systems sciences, vol 173. IOS Press, Amsterdam, pp 421–505
13. Cousot P (2000) Partial completeness of abstract fixpoint checking. In: Choueiry BY, Walsh T (eds) Proc 4th int symp SARA '2000, Horseshoe Bay, 26–29 Jul 2000. LNAI, vol 1864. Springer, Berlin, pp 1–25
14. Cousot P, Cousot R (1976) Static determination of dynamic properties of programs. In: Proc 2nd int symp on programming. Dunod, Paris, pp 106–130
15. Cousot P, Cousot R (1977) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th POPL, Los Angeles, 1977. ACM, New York, pp 238–252
16. Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: 6th POPL, San Antonio, 1979. ACM, New York, pp 269–282
17. Cousot P, Cousot R (1982) Induction principles for proving invariance properties of programs. In: Néel D (ed) Tools & notions for program construction. Cambridge University Press, Cambridge, pp 43–119
18. Cousot P, Cousot R (1992) Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe M, Wirsing M (eds) Proc 4th int symp on PLILP '92, Leuven, 26–28 Aug 1992. LNCS, vol 631. Springer, Berlin, pp 269–295
19. Cousot P, Cousot R (1993) "À la Burstall" intermittent assertions induction principles for proving inevitability properties of programs. Theor Comput Sci 120:123–155

20. Cousot P, Cousot R (2002) Modular static program analysis. In: Horspool RN (ed) Proc 11th int conf CC ’2002, Grenoble, 6–14 Apr 2002. LNCS, vol 2304. Springer, Berlin, pp 159–178
21. Cousot P, Halbwachs N (1978) Automatic discovery of linear restraints among variables of a program. In: 5th POPL, Tucson, 1978. ACM, New York, pp 84–97
22. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2005) The ASTRÉE analyser. In: Sagiv M (ed) Proc 14th ESOP ’2005, Edinburg, 2–10 Apr 2005. LNCS, vol 3444. Springer, Berlin, pp 21–30
23. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2007) Varieties of static analyzers: A comparison with ASTRÉE. In: Hinchey M, Jifeng H, Sanders J (eds) Proc 1st TASE ’07, Shanghai, 6–8 June 2007. IEEE Comput Soc, Los Alamitos, pp 3–17
24. Cousot P, Ganty P, Raskin J-F (2007) Fixpoint-guided abstraction refinements. In: Filé G, Riis-Nielson H (eds) Proc 14th int symp SAS ’07, Kongens Lyngby, 22–24 Aug 2007. LNCS, vol 4634. Springer, Berlin, pp 333–348
25. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2008) Combination of abstractions in the ASTRÉE static analyzer. In: Okada M, Satoh I (eds) 11th ASIAN 06, Tokyo, 6–8 Dec 2006. LNCS, vol 4435. Springer, Berlin, pp 272–300
26. Delmas D, Souyris J (2007) ASTRÉE: from research to industry. In: Filé G, Riis-Nielson H (eds) Proc 14th int symp SAS ’07, Kongens Lyngby, 22–24 Aug 2007. LNCS, vol 4634. Springer, Berlin, pp 437–451
27. D’Silva V, Kroening D, Weissenbacher G (2008) A survey of automated techniques for formal software verification. IEEE Trans Comput-Aided Des Integr Circuits 27(7):1165–1178
28. Feret J (2004) Static analysis of digital filters. In: Schmidt D (ed) Proc 30th ESOP ’2004, Barcelona, Mar 27–Apr 4, 2004. LNCS, vol 2986. Springer, Berlin, pp 33–48
29. Feret J (2005) The arithmetic-geometric progression abstract domain. In: Cousot R (ed) Proc 6th int conf VMCAI 2005, Paris, 17–19 Jan 2005. LNCS, vol 3385. Springer, Berlin, pp 42–58
30. Feret J (2005) Numerical abstract domains for digital filters. In: 1st int work on numerical & symbolic abstract domains, NSAD ’05, Maison Des Polytechniciens, Paris, 21 Jan 2005
31. Filé G, Ranzato F (1994) Improving abstract interpretations by systematic lifting to the powerset. In: Bruynooghe M (ed) Proc int symp ILPS ’1994, Ithaca, 13–17 Nov 1994. MIT Press, Cambridge, pp 655–669
32. Floyd RW (1967) Assigning meaning to programs. In: Schwartz JT (ed) Proc symposium in applied mathematics, vol 19. AMS, Providence, pp 19–32
33. Giacobazzi R, Ranzato F (1998) Optimal domains for disjunctive abstract interpretation. Sci Comput Program 32(1–3):177–210
34. Giacobazzi R, Ranzato F, Scozzari F (2000) Making abstract interpretations complete. J ACM 47(2):361–416
35. Goubault É (2001) Static analyses of floating-point operations. In: Cousot P (ed) Proc 8th int symp SAS ’01, Paris, Jul 2001. LNCS, vol 2126. Springer, Berlin, pp 234–259
36. Goubault É, Martel M, Putot S (2002) Asserting the precision of floating-point computations: a simple abstract interpreter. In: Le Métayer D (ed) Proc 11th ESOP ’2002, Grenoble, 8–12 Apr 2002. LNCS, vol 2305. Springer, Berlin, pp 209–212
37. Graf S, Saïdi H (1996) Verifying invariants using theorem proving. In: Alur R, Henzinger TA (eds) Proc 8th int conf CAV ’97, New Brunswick, Jul 31–Aug 3, 1996. LNCS, vol 1102. Springer, Berlin, pp 196–207
38. Granger P (1989) Static analysis of arithmetical congruences. Int J Comput Math 30(3 & 4):165–190
39. Granger P (1991) Improving the results of static analyses of programs by local decreasing iterations. Res rep. LIX/RR/91/08, Laboratoire d’Informatique, École polytechnique, Palaiseau, Dec 1991
40. Gulavani BS, Chakraborty S, Nori AV, Rajamani SK (2008) Automatically refining abstract interpretations. In: Ramakrishnan CR, Rehof J (eds) Proc 14th int conf TACAS ’2000, Budapest, 29 Mar–6 Apr 2008. LNCS, vol 4963. Springer, Berlin, pp 443–458
41. Handjieva M, Tzolovski S (1998) Refining static analyses by trace-based partitioning using control flow. In: Levi G (ed) Proc 5th int symp SAS ’98, Pisa, 14–16 Sep 1998. LNCS, vol 1503. Springer, Berlin, pp 200–214
42. Hecht MS (1977) Flow analysis of computer programs. North-Holland/Elsevier, Amsterdam
43. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580
44. Jeannet B, Miné A (2007) The `Apron` numerical abstract domain library. http://apron.cri.ensmp.fr/library/
45. Lacan P, Monfort JN, Ribal LVQ, Deutsch A, Gonthier G (1998) The software reliability verification process: The ARIANE 5 example. In Proceedings DASIA 98—DAta Systems In Aerospace, Athens. ESA Publications, SP-422, 25–28 May 1998

46. Leroy X, Doligez D, Garrigue J, Rémy D, Vouillon J (2007) The Objective Caml system, documentation and user's manual (release 3.10). Technical report, INRIA, Rocquencourt, France, 19 May 2007. http://caml.inria.fr/pub/docs/manual-ocaml/
47. Mauborgne L (2004) ASTRÉE: Verification of absence of run-time error. In: Jacquart P (ed) Building the information society. Kluwer Academic, Norwell, pp 385–392. Chap 4
48. Mauborgne L, Rival X (2005) Trace partitioning in abstract interpretation based static analyzer. In: Sagiv M (ed) Proc 14th ESOP '2005, Edinburg, Apr 2–10, 2005. LNCS, vol 3444. Springer, Berlin, pp 5–20
49. Miné A (2004) Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt D (ed) Proc 30th ESOP '2004, Barcelona, Mar 27–Apr 4, 2004. LNCS, vol 2986. Springer, Berlin, pp 3–17
50. Miné A (2004) Weakly relational numerical abstract domains. Thèse de doctorat en informatique, École polytechnique, Palaiseau, 6 Dec 2004
51. Miné A (2006) Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc LCTES '2006. ACM, New York, pp 54–63
52. Miné A (2006) The octagon abstract domain. High-Order Symb Comput 19:31–100
53. Miné A (2006) Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson EA, Namjoshi KS (eds) Proc 7th int conf VMCAI 2006, Charleston, 8–10 Jan 2006. LNCS, vol 3855. Springer, Berlin, pp 348–363
54. Monniaux D (2005) The parallel implementation of the ASTRÉE static analyzer. In: Proc 3rd APLAS '2005, Tsukuba, 3–5 Nov 2005. LNCS, vol 3780. Springer, Berlin, pp 86–96
55. Naur P (1966) Proofs of algorithms by general snapshots. BIT 6:310–316
56. Pioli A, Hind M (1999) Combining interprocedural pointer analysis and conditional constant propagation. Technical Report 99-103, IBM
57. Randimbivololona F, Souyris J, Deutsch A (2000) Improving avionics software verification cost-effectiveness: Abstract interpretation based technology contribution. In: Proceedings DASIA 2000—DAta Systems In Aerospace, Montreal. ESA Publications, 22–26 May 2000
58. Rival X, Mauborgne L (2007) The trace partitioning abstract domain. TOPLAS 29(5)
59. Sagiv M, Reps T, Wilhelm R (1999) Parametric shape analysis via 3-valued logic. In: 26th POPL, San Antonio, 1999. ACM, New York, pp 105–118
60. Souyris J (2004) Industrial experience of abstract interpretation-based static analyzers. In: Jacquart P (ed) Building the information society. Kluwer Academic, Norwell, pp 393–400. Chap 4
61. Souyris J, Delmas D (2007) Experimental assessment of ASTRÉE on safety-critical avionics software. In: Saglietti F, Oster N (eds) Proc int conf on computer safety, reliability, and security (SAFECOMP 2007), Nuremberg, 18–21 Sep 2007. LNCS, vol 4680. Springer, Berlin, pp 479–490
62. Su Z, Wagner D (2005) A class of polynomially solvable range constraints for interval analysis without widenings. Theor Comput Sci 345(1):122–138
63. Traverse P, Lacaze I, Souyris J (2004) Airbus fly-by-wire—a total approach to dependability. In: Jacquart P (ed) Building the information society. Kluwer Academic, Norwell, pp 191–212, Chap 3