Varieties of Static Analyzers: A Comparison with ASTRÉE

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival

Varieties of Static Analyzers: A Comparison with ASTRÉE

Patrick COUSOT[1], Radhia COUSOT[2], Jérôme FERET[1], Antoine MINÉ[1],
Laurent MAUBORGNE[1], David MONNIAUX[3], Xavier RIVAL[1]

Abstract

We discuss the characteristic properties of ASTRÉE*, an automatic static analyzer for proving the absence of runtime errors in safety-critical real-time synchronous control-command C programs, and compare it with a variety of other program analysis tools.*

1 Introduction

ASTRÉE [`www.astree.ens.fr`] is an automatic static analyzer for proving the absence of runtime errors in programs written in C [3, 4, 17, 18, 56]. ASTRÉE's design is formally founded on the theory of abstract interpretation [14, 15]. ASTRÉE is designed to be highly capable and extremely competitive on safety-critical real-time synchronous control-command programs. On this family of programs, ASTRÉE produces very few *false alarms* (i.e. signals of potential runtime errors that are due to the imprecision of the static analysis but can never happen at runtime in any actual execution environment). ASTRÉE can be tuned to get no false alarms thanks to parameters and directives, which inclusion can be automatized. In absence of any alarm, ASTRÉE's static analysis provides a *proof* of absence of runtime errors.

In this paper we discuss the main characteristic properties of ASTRÉE and compare it to a large variety of program analysis tools.

2 Dynamic Analyzers

Dynamic analyzers check program properties at run-time. Simple examples are runtime checks inserted by compilers to avoid out-of-bounds array accesses or user-provided `assert` statements. More sophisticated examples are provided by runtime analyzers (like Compuware BoundsChecker, Rational Purify™ [32] or PurifyPlus™, Valgrind [69], Parasoft Insure++) or even integrated debugging environments with automatic test generators (like a.o. Cosmic Software's IDEA) or dynamic program instrumentation [49].

[1]École Normale Supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France, `First.Last@ens.fr`

[2]CNRS, École Polytechnique, DGAR, 91128 Palaiseau Cedex, France, `Radhia.Cousot@polytechnique.fr`

[3]CNRS, École Normale Supérieure

The main defects of dynamic analyzers are that

- they can prove the presence of errors, usually not their absence;
- they cannot check for all interesting program properties like presence of dead-code or non-termination.

One enormous advantage of runtime tests should be that they are performed on the program which is executed in exploitation. Unfortunately this is not always true since sometimes the tested version of the program is not exactly the executed one since runtime test code may perturb the tested code (like in the source level instrumentation of Insure++).

Moreover when an error is discovered at runtime during program exploitation, some action must be taken (e.g. reboot of a faulty driver [80]). In safety critical systems this involves fault tolerance mechanisms for error recovery. Although widely accepted for hardware failures, software failure recovery is much harder and complex.

3 Static Analyzers

Static analyzers like ASTRÉE analyze the program at compile-time, inspecting directly the source code, and never executing the program. This is the code inspection idea [22], but automated.

Since the notion of "static analyzer" has a broad acceptance, it is sometimes understood as including *style checkers* looking for deviations from coding style rules (like Safer C checking for C fault and failure modes [33] or Cosmic Software's MISRA checker, checking for the guidelines prescribed by the Motor Industry Software Reliability Association [64]). Such style checkers are usually not "semantic-based", and thus cannot check for correct runtime behavior. However, style restrictions, as considered by ASTRÉE (no side effects in expressions, no recursion, no dynamic memory allocation, no calls to libraries), can considerably help the efficiency and precision of the static analysis.

To explore program executions without executing the program, code inspectors and static analyzers all use an explicit (or implicit) program semantics that is a formal (or informal) model of the program executions in all possible or restricted execution environments. If the execution environments are restricted, these restrictions are hypotheses for the static analyzer which must be validated by some other means. For example ASTRÉE uses a configuration file providing a maximal execution time, intervals of some input variables, etc. — the analysis results are correct only if these assumptions are correct.

4 Defining the Formal Semantics of the Source Code

Despite the impressive bibliography on programming languages semantics, it is extremely difficult to define rigorously the semantics of real-life programming languages such as C. Most languages are defined by informal (as in: non-mathematical) specifications, even those for which there exists an official standard, such as C [45] or C++. Not only can these informal specifications be ambiguous or misunderstood, they may also give significant leeway to implementors.

4.1 Undefined Runtime Behaviors

First there are conditions in which the source semantics is undefined e.g. after a runtime-error while the actual execution will do something unknown. For example an erroneous modification through an out-of-bounds array access or a dangling pointer might destroy the object code thus leading to erratic runtime behaviors. Then nothing is known statically in which case a static analysis is merely impossible.

To solve the semantics undefinedness problem, ASTRÉE considers two cases for the concrete execution of programs.

(a) For errors corresponding to *undefined behaviors*, ASTRÉE signals a potential error and goes on with the only concrete executions for which the runtime error does not occur.

Examples are integer division by zero, float overflow, `NaN` or invalid operations without mathematical meaning (so that e.g. the program might be stopped by an interrupt).

Other examples are invalid array or pointer access which might corrupt memory, including the executed code, or even bring about an invalid memory access (such as segmentation faults/address or bus errors). An example is

```
0  % cat unpredictable.c
1  #include <stdio.h>
2  int main () {
3   int n, T[0];
4   n = 2147483647;
5   printf("n = %i, T[n] = %i\n", n, T[n]);
6  }
```

which yields different results on different machines

```
n = 2147483647, T[n] = 2147483647      Macintosh PPC
n = 2147483647, T[n] = -1208492044     Macintosh Intel
n = 2147483647, T[n] = -135294988      PC Intel 32 bits
Bus error                              PC Intel 64 bits
```

For such undefined behavior errors, what will happen after the error is unforeseeable, so ASTRÉE does not attempt to make any sensible prediction. The program is merely assumed to stop after the error. Another example is

```
    0    5    10   15   20   25   30   35   40   45
    ----|----|----|----|----|----|----|----|----|
0  % cat dividebyzero.c
1  int main()
2  { int I;
3    I = 1 / 0;
4    I = 1 / 0;
5  }
```

ASTRÉE does not warn on the division by zero at line 4

```
6  % astree --exec-fn main dividebyzero.c |& egrep "WARN"
7  dividebyzero.c:3.6-11::[call#main@1:]: WARN: integer d
8  ivision by zero ({0} and {1} / {0})
9  %
```

since after the correctly warned division by zero at line 3, execution is supposed to have definitely stopped so that line 4 is unreachable.

(b) For errors corresponding to *unspecified behaviors*, ASTRÉE signals a potential error and goes on with all possible concrete executions, assuming an undefined result when the runtime error does occur. For such unspecified behavior errors, what will happen after the error is foreseeable, so ASTRÉE assumes that the execution will continue in the worst possible conditions.

Examples are integer overflow or invalid shifts `<<` or `>>` for which the actual computations are quite different from the expected mathematical meaning.

An example program for this second case (b) is provided for integer overflow in Sect. 4.2 (`minus-int.c`).

In both cases (a) and (b), the analysis will warn the user of the presence, position and precise nature of the threat and will go on with an over-approximation of the considered concrete executions.

Observe that the conclusions of ASTRÉE are always valid at runtime even for executions with unspecified behavior errors (since all behaviors after the error fall in a case considered by ASTRÉE). If the execution has undefined behavior errors, then the conclusions of ASTRÉE are valid on the execution before the first such undefined behavior error. Afterward, ASTRÉE considers that the program execution stopped on such a severe error and so its conclusions on the rest of the execution (if any, the program might have been stopped by the operating system) might be invalid.

In both circumstances, this means that in case of false alarm, all following actual errors will definitely be checked.

However for undefined behavior errors, some unpredictable program behaviors might have been disregarded by ASTRÉE than can actually happen after this error.

Nevertheless, in absence of alarms, the program behaviors at execution are perfectly defined by the semantics of C. So the objective of ASTRÉE is to prove statically that the program behaviors are always well-defined at execution, in which case there is not any need for runtime tests.

4.2 Machine Dependencies

Most languages let the choice of data structure representations, such as those of basic types, depend on the machine (architecture and operating system). For example in C, signed integers (`int` or `signed int`) have no specified size; the only requirement is that they should include all values between `INT_MIN` (always less than or equal to -32767) and `INT_MAX` (always greater than or equal to 32767). An integer can be represented either by a sign and an absolute value, by one's complement, or two's complement. [45, §6.2.6.2]. The last case, representation modulo 2^n where n is the number of bits in the data type, is the case on all current microprocessor architectures.

This choice should be parameterized in a static analyzer since the semantics are different. For the sake of simplicity ASTRÉE

makes the choice of complement to 2 so INT_MIN = -INT_MAX - 1 (and thus cannot be used for analyzing programs running on systems using other representations, at least not without some reprogramming). According to the C standard, the result of an overflowing arithmetic operation is undefined. [45, § 6.5] However most modern machines do not signal any error on an integer overflow, and perform arithmetic modulo 2^n where n is the number of bits in the data type. Nevertheless, ASTRÉE will signal such overflows. The analysis considers that program execution continues in the case when the result of the operation is machine representable (through modular arithmetic) and that it stops otherwise (for instance, on a division by zero).

Consider for example the program

```
    0    5    10   15   20   25   30   35   40   45
    |----|----|----|----|----|----|----|----|----|
0   % cat minus-int.c
1   void main () {
2     int i;
3     if (i < 0) { i = -i; };
4     __ASTREE_assert((i != -1));
5   }
    |----|----|----|----|----|----|----|----|----|
    0    5    10   15   20   25   30   35   40   45
```

The variable i initially takes any value in the interval $[-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}]$ (where $\texttt{INT\_MAX} = 2147483647$) and when i < 0, the operation -i leads to a potential overflow for the value $-\texttt{INT\_MAX} - 1$ outside the $[-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}]$ interval of machine representable integers. This potential error is signaled by ASTRÉE. Following this error, the analysis goes on as if the result could be any value in the interval $[-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}]$. This choice includes

- modular arithmetics in two's complement where the result of the -i belongs to $\{-\texttt{INT\_MAX} - 1\} \cup [1, \texttt{INT\_MAX}]$ when the value of i belongs to $[-\texttt{INT\_MAX} - 1, -1]$;
- two's complement arithmetics limited to the interval $[-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}]$ where the result of the operation -i belongs to the set $[1, \texttt{INT\_MAX} + 1] \cap [-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}] = [1, \texttt{INT\_MAX}]$ when the value of i belongs to $[-\texttt{INT\_MAX} - 1, -1]$.

So ASTRÉE considers that because of the potential overflow in -i, the value of i after the test is in the representation interval $[-\texttt{INT\_MAX} - 1, \texttt{INT\_MAX}]$, thus the assertion might be wrong.

```
6   % astree --exec-fn main minus-int.c |& egrep "WARN"
7   minus-int.c:3.19-21::[call#main@1:]: WARN: signed int
8   arithmetic range [1, 2147483648] not included in [-214
9   7483648, 2147483647]
10  minus-int.c:4.19-26::[call#main@1:]: WARN: assert fail
11  ure
12  %
```

By the way, notice the user check __ASTREE_assert which, together with the user hypothesis __ASTREE_known_fact, allows ASTRÉE to verify simple functional properties.

4.3 Compiler Dependencies

The C standard leaves open numerous possible choices for compilers such as the simplification and order of evaluation of arithmetic expressions [45, § 6.5], which, in the presence of side effects, may lead to quite different results. Since the order of evaluation chosen by compilers is usually not documented (except in the compiler source code, if available), a sound generic static analyzer would have to consider all possible execution orders allowed by the C standard. Unfortunately this would be costly and imprecise. Thus, ASTRÉE chooses the left to right evaluation order, but also performs a simple analysis that verifies that there is no ambiguity due to side effects in expressions; it flags all possible ambiguities. So ASTRÉE is sound if the compiler evaluates expressions from left to right or there is no alarm on the evaluation order (in which case the program is portable).

4.4 Linker Dependencies

Linkers may modify the program semantics e.g. by absolute memory allocation (when symbolically referring to absolute locations of memory), mixing big/little-endian object linking, etc. In principle a sound static analyzer would have to take the linker options into account to define the program semantics. ASTRÉE makes common sense hypotheses to be checked by translation validation (Sect. 5).

4.5 Operating System Dependencies

Although embedded programs often do not rely on operating systems components (like libraries), some program behaviors (e.g. after a reboot) may depend on the operating system. For example, static are implicitly initialized to 0 according to the C standard [45, § 6.7.8], usually by the operating system or program loader zeroing the uninitialized data segment. In that case, global variables may also be initialized to zero, which is not standard. However, some embedded systems may do otherwise and not zero these data. ASTRÉE thus offers the option to assume that static variables are uninitialized (which is sound regardless of what the embedded system does, since it is an over-approximation of the standard behavior).

4.6 Clean Semantics

Obviously programming languages with clean semantics can considerably help the ease of design and the effectiveness of static analyzers. Machine-checkable style restrictions [64] or better, type-safe C subsets (like CCured [68] or Cyclone [31]) should definitely be considered for the development of high-quality software.

5 Proving the Equivalence of the Source and Object Code

The formal semantics used for the static analysis must be proved to be compatible with actual execution on a computer. This can be done by *compiler verification* [53, 52] (proving equivalence of the source and object code for all programs of a language and a given machine), *translation validation* (for a given source program and the corresponding object code) [70] or *proof translation* (showing that the proof done on the source code remains valid on the compiled code) [73]. ASTRÉE uses translation validation [74].

These approaches all suffer from the fact that the semantics of the object code must be defined formally and itself validated. Usually it is considered to be simple enough not to need validation, but the question remains opened.

6 Unsoundness

A program analyzer is *unsound* when it can omit to signal an error that can appear at runtime in some execution environment. Unsound analyzers are *bug hunters* or *bug finders* aiming at finding some of the bugs in a well-defined class. Their main defect is unreliability, being subject to *false negatives* thus claiming that they can no longer find any bug while many may be left in the considered class.

One reason for unsoundness is the analysis cost. Obviously all dynamic analyzers are *unsound* since it is impossible to explore all possible executions of a program on a machine.

Another reason for unsoundness is the analysis complexity. For example ESC Java does not handle correctly modulo arithmetic (in absence of efficient provers to do so) [51]. CMC [21] from Coverity.com ignores code with complex semantics which analysis would be too hard to design.

The main reason for unsoundness is often to avoid reporting too many unwanted false positives. A low false alarm rate of 1% in a 1.000.000 LOCs programs would mean examining manually 10.000 potential errors which would be extremely costly to sort manually. To cope with this false alarm plague, many static analyzers choose to avoid overwhelming messages by deliberately not reporting all errors, which is unsound. We consider several approaches, all unsound.

- skipping program parts which are hard to analyze, Sect. 6.1;
- ignoring some types of errors, Sect. 6.2 & 6.3;
- disregarding some runtime executions, Sect. 6.4;
- changing the program semantics, Sect. 6.5.

Such unsound approaches are all excluded in ASTRÉE.

6.1 Unsoundness by deliberate skipping of instructions

In some cases, the analysis may encounter instructions for which it knows it is probably very imprecise, or instructions that it fails to understand. The latter case may occur with dynamic dispatch of methods in object-oriented languages: if the analysis for the dynamic type of objects is too imprecise, it may be difficult to know which functions may actually be called and on which operands. The former case can occur even in very simple settings, such as a write through a pointer:

```
*p = 42;
```

If the pointer analysis is imprecise, then very little information may be known as to where `p` may point. If the analysis concludes that `p` may point to any among 1000 variables, then the only sound way to proceed is to assume that any of these may receive the value 42. This is likely to discard a lot of information previously known (for instance, if the analysis had established for 900 of these variables that they were in $[0, 1]$, then this information will be lost), thus leading to great imprecision and possibly many false alarms down the road. Also, imprecise analysis may result in slow analysis, since many more program configurations may have to be explored than in reality.

For these reasons, some unsound analyzers may decide to ignore instructions on which they know they "probably" are very imprecise. There is a trade-off here between soundness, on the one hand, and precision and speed of analysis on the other hand.

6.2 Unsoundness by under-exploration of the potential error space

Alarms, either true or false, can be reduced by considering only a subset of the potential errors (while being exhaustive on the explored runtime execution space, or not). This consists in selecting typical occurrences of bugs in a well-defined class, as opposed to all bugs in this class.

One such example is static type verification and inference systems proving that "well-typed programs do not go wrong" [58]. Typically, an expression of type `int` will be typable despite the fact that it might divide by zero for some executions.

Another example is UNO concentrating exclusively on uninitialized variables, nil-pointer references, and out-of-bounds array indexing [35].

A last example is *bug pattern mining* as in Klocwork K7™ [41], looking for a few key types of programmer errors [79], PREfast (and PREfast for Drivers), a lightweight version of PREfix [7], looking for common basic coding errors in C (and driver-specific rules) [67] or PMD (checking for unused fields, empty try/catch/finally/if/while blocks, unused method parameters, etc in Java™) [12].

6.3 Unsoundness by underreporting the found runtime execution errors

Although all potential errors of all potential runtime executions could have been found, it is always possible not to report all found alarms.

This can be done manually (although this is extremely costly). For example, the publicly available analysis by Reasoning, Inc. on version 4.1.24 of Apache Tomcat [43] is obtained by a static analysis [6] followed by manual auditing and removal of false positives before a report is given to their customers.

The end-user may also be offered the possibility of suppressing false positives so they do not show up in subsequent analysis runs (like with CodeSonar [40] or using assert statements as recommended for PC-lint™ and FlexeLint™ [39], which is unfortunately not enough to counter the very high false positive rates).

There are also several ways to automate this alarm selection activity by appropriate filtering. Splint [50] sorts the most probable messages according to common programming practices while Airac5 relies on classification methods [78]. ORION like UNO [35] attempts to reasons symbolically about execution traces while FindBugs™ uses empirical heuristics (like reporting a null pointer warning if there is a statement or branch that, if executed/taken, at least potentially (ignoring e.g. aliasing or exceptions), guarantees a null pointer exception) [37].

6.4 Unsoundness by under-exploration of the runtime execution space

One way of avoiding overwhelming (false) alarms is to explore only a subset of the runtime execution space. This is the case for dynamic analysis (Sect. 2).

In the case of static analysis (Sect. 3), static type verification and inference systems proving that "well-typed programs do not go wrong" [58] may be unable to type well-going programs. This is the case for example of errors that are impossible because of guards that are not taken into account with enough precision by the type system. The problem is that of the coarse approximation of sets of values by types which is too imprecise to encode the possible values of the guard.

Another example in the case of static analysis (Sect. 3), is that of model-checkers which usually explore only part of the state space. In particular bounded model-checkers explore only finite prefixes of some of the execution traces. Some bugs like overflow by accumulation of small rounding errors over long periods of computation time (see Sect. 12.3) obviously stay beyond the scope of such techniques exploring a relatively small subset of the state space. An example is the Bounded Model Checker CBMC [11] for ANSI-C programs (with restrictions like `float` and `double` using fixed-point arithmetic). It allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions. This is done by unwinding the loops in the program up to a finite depth and passing the resulting equation to a SAT solver. This unwinding may not stop when the built-in simplifier is not able to determine a runtime bound for the loop, in which case a loop bound must be specified by the user. The CBMC user manual [10] provides such an example for a binary search. This was also the main example of [13] (handled by hand at the time!)

```
0  % cat dichotomy.c
1  int main () {
2     int R[100], X; short lwb, upb, m;
3     lwb = 0; upb = 99;
4     while (lwb <= upb) {
5        m = upb + lwb;
6        m = m >> 1;
7        if (X == R[m]) { upb = m; lwb = m+1; }
8        else if (X < R[m]) { upb = m - 1; }
9        else { lwb = m + 1; }
10    }
11    __ASTREE_log_vars((m));
12 }
```

for which ASTRÉE proves the absence of runtime errors by providing a correct interval for the middle position

```
13 % astree --exec-fn main dichotomy.c |& egrep "(WARN)|(
14 m in)"
15 direct = <integers (intv+cong+bitfield+set): m in [0,
16 99] /\ Top >
17 %
```

6.5 Unsoundness by considering a mathematical semantics

Besides being not automatic since end-users are required to provide invariants, if not proof templates or even complete proofs, theorem-prover/proof assistant based static analysis approaches such as CAVEAT [72] are based on a Hoare logic with mathematical meaning so that e.g. integers are assumed to be in $\mathbb{Z}$, not machine integers, reals are assumed to be in $\mathbb{R}$, not floats or double, etc. It follows that sound static analyzers like ASTRÉE, sticking closely to the machine semantics are able to find runtime errors in programs which correctness has been machine-checked by program provers (but with a mathematical semantics, not the programming language one). An example is the binary search considered at Sect. 6.4, modified as follows

```
0  % diff dichotomy.c dichotomy-bug.c
1  2,3c2,3
2  <     int R[100], X; short lwb, upb, m;
3  <     lwb = 0; upb = 99;
4  ---
5  >     int R[30000], X; short lwb, upb, m;
6  >     lwb = 0; upb = 29999;
```

for which, although the algorithm is the same and "correct", ASTRÉE signals errors, starting with

```
7  % astree --exec-fn main dichotomy-bug.c |& egrep "WARN
8  " | head -n2
9  dichotomy-bug.c:5.6-19::[call#main@1:loop@4=2:]: WARN:
10  implicit signed int->signed short conversion range [1
11 4998, 44999] not included in [-32768, 32767]
12 dichotomy-bug.c:7.15-19::[call#main@1:loop@4=2:]: WARN
13 : invalid dereference: dereferencing 4 byte(s) at offs
14 et(s) [0;4294967295] may overflow the variable R of by
15 te-size 120000 or mis-aligned pointer (1Z+0) may not a
16  multiple of 4
17 %
```

Some program provers like CADUCEUS [25] can handle bounded integers or floats at the price of heavy manual interactions and a complex model [5]. In contrast, ASTRÉE can be used in completely automatic mode.

7 Soundness

In contrast, ASTRÉE is a *sound* program analyzer since it never omits to signal an error that can appear in some execution environment. Another example is PolySpace Verifier [20]. These static analyzers are therefore *bug eradicators* since they find all bugs in a well-defined class (that of runtime errors).

7.1 Exhaustivity

ASTRÉE covers the whole runtime execution space, with a correct semantics, never omitting potential errors. As discussed in previous Sect. 6, the main problem is then for the static analysis to be sufficiently precise to avoid false alarms.

7.2 Floats

One problem with the semantic soundness for which ASTRÉE has been very carefully designed is the rounding errors in floating-point computations [60, 63]. Most static analyzers either do not handle floats or handle them incorrectly because they are based on mathematical properties of reals not valid for floats. For example $(x + a) - (x - a) = 2a$ is not valid for floats as shown be the following example

```
% cat double-float1.c
int main () {
  double x; float a, y, z, r1, r2;
  a = 1.0; x = 1125899973951488.0; y = (x + a); z = (x
 - a);
  r1 = y - z; r2 = 2 * a;
  printf("(x + a) - (x - a) = %f\n", r1);
  printf("2a                = %f\n", r2);
}
% gcc double-float1.c >& /dev/null; ./a.out
(x + a) - (x - a) = 134217728.000000
2a                = 2.000000
```

The situation is explained in Fig. 1. The double `x` = 1125899973951488.0 is just in the middle of two consecutive floats so `x - 1` and `x + 1` are rounded to these floats, the distance between them is 134217728.0.

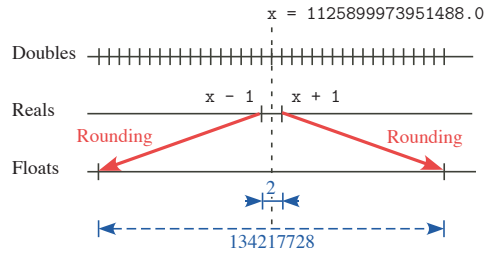

Figure 1. Rounding to different nearest floats

Changing the least significant digit, yields a completely different result explained in Fig. 2. The double `x` = 1125899973951487.0 is no longer in the middle of two consecutive floats so `x - 1` and `x + 1` are both rounded to the same float, whence the difference of 0.0.

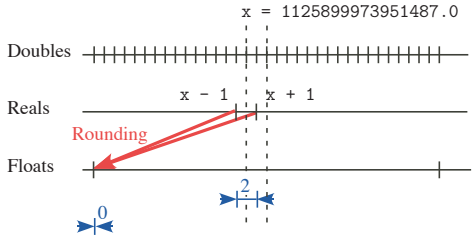

Figure 2. Rounding to same float

```
% cat double-float2.c
int main () {
  double x; float a, y, z, r1, r2;
  a = 1.0; x = 1125899973951487.0; y = (x + a); z = (x
 - a);
  r1 = y - z; r2 = 2 * a;
  printf("(x + a) - (x - a) = %f\n", r1);
  printf("2a                = %f\n", r2);
}
% gcc double-float2.c >& /dev/null; ./a.out
(x + a) - (x - a) = 0.000000
2a                = 2.000000
```

ASTRÉE is able to make a correct analysis

```
% cat double-float1-analyze.c
int main () {
  double x; float a, y, z, r1, r2;
  a = 1.0; x = 1125899973951488.0; y = (x + a); z = (x
 - a);
  r1 = y - z; r2 = 2 * a;
  __ASTREE_log_vars((r1, r2));
}
% astree --exec-fn main double-float1-analyze.c |& egr
ep "r1 in"
direct = <float-interval: r2 in {2}, r1 in [-1.34218e+
08, 1.34218e+08] >
%
```

Note that the value $1.34218e+08$ is an artifact of the printing system that rounds the actual value 134217728.0 internally computed by the analyzer. Actually, the interval $[-1.34218e+08, 1.34218e+08]$ is not reduced to the singleton $\{134217728.0\}$ due to possible rounding errors when compiling the constant into its binary form and the overestimation of the roundings in the computations of `y = (x + a)`, `z = (x - a)`, and `r1 = y - z;`[4].

Indeed ASTRÉE approximates arbitrary numerical expressions in the form $[a_0, b_0] + \sum_k ([a_k, b_k] \times V_k)$ for program variables V_k [60, 63]. For example, the floating-point assignment `Z = X - (0.25 * X)` is linearized as the following assignment interpreted in real arithmetics $\mathtt{Z} = ([0.749\cdots, 0.750\cdots] \times \mathtt{X}) + (2.35\cdots 10^{-38} \times [-1, 1])$ making rounding errors explicit as small intervals. This abstraction is an example of good compromise between cost and precision. It allows simplification, even in the interval domain, since if $\mathtt{X} \in$ [-1,1], we get $|\mathtt{Z}| \leq 0.750\cdots$ instead of $|\mathtt{Z}| \leq 1.25\cdots$. It also allows the use of powerful relational abstract domains (like octagons [61], see Sect. 11.5).

Observe that ASTRÉE always overestimates rounding errors, cumulating all possible origins. So ASTRÉE will detect catastrophic losses of precision leading to overflows, and their significance and source. Static analyzers like FLUCTUAT [55] are more specifically designed to analyze the relative contributions of the rounding errors at all stages of a floating point computation.

7.3 Memory Model

Another problematic feature for the soundness of a static analyzer is the C memory model, which can be used at very low level, in particular in automatically generated C code, almost at the assembly level. The memory model of ASTRÉE has been recently revisited to include pointer arithmetic and `union` [62], remarkably leaving most of the existing design of ASTRÉE untouched.

For example, on the pointer cast

```
0    5    10   15   20   25   30   35   40   45
|----|----|----|----|----|----|----|----|----|
% cat memcpy.c
/* memcpy.c (polymorphic memcpy) */

/* byte per byte copy of src into dst */
void memcpy(char* dst, const char* src, unsigned size)
{
  int i;
  for (i=0;i<size;i++) dst[i] = src[i];
}
```

[4] ASTRÉE always choose the worst case among the four possible rounding modes *round to nearest*, *round toward* $+\infty$, *round toward* $-\infty$, and *round toward 0*.

```
10  void main()
11  {
12    float x = 10.0, y;
13    int zero = 0;
14    /* copy of x into y (well-typed) */
15    memcpy(&y,&x,sizeof(y));
16    __ASTREE_assert((y==10.0));
17    /* copy of zero into y (not well-typed but allowed i
18  n C) */
19    memcpy(&y,&zero,sizeof(y));
20    __ASTREE_assert((y==0.0));
21  }
      ----|----|----|----|----|----|----|----|----|
      0    5    10   15   20   25   30   35   40   45
```

ASTRÉE is able to correctly handle the indirect assignments

```
22  % astree --exec-fn main --unroll 5 memcpy.c |& egrep "
23  WARN"
24  %
```

ASTRÉE also correctly handles unions, as in

```
      0    5    10   15   20   25   30   35   40   45
      ----|----|----|----|----|----|----|----|----|
0   % cat union.c
1   /* union.c (union type) */
2
3   union {
4     int type;
5     struct { int type; int data; } A;
6     struct { int type; char data[3]; } B;
7   } u;
8
9   void main()
10  {
11    /* no assert failure */
12    u.type = 12;
13    __ASTREE_assert((u.A.type==12));
14    __ASTREE_assert((u.B.type==12));
15
16    /* assert failure because the modification of u.B.da
17  ta also modifies u.A.data */
18    u.A.data = 0;
19    u.B.data[0] = 12;
20    __ASTREE_assert((u.A.data==0));
21  }
      ----|----|----|----|----|----|----|----|----|
      0    5    10   15   20   25   30   35   40   45
```

where overlaps and side-effects are correctly taken into account.

```
22  % astree --exec-fn main --full-memory-model union.c |&
23   egrep "WARN"
24  union.c:19.19-30::[call#main@9:]: WARN: assert failure
25
26  %
```

8 [In]completeness

A static analyzer is *complete* when it never produces false alarms on any program. This terminology is justified by the fact that such an analyzer, when queried whether or not some true property holds on the program (for instance: "the program never ends with an overflow"), will never issue an alarm. Never reporting any alarm, or only a few definite ones, would be complete (and useless). So this completeness requirement should be understood in the context of soundness. Apart for trivial programs, sound static analyzers are all *incomplete*, by undecidability. ASTRÉE for example is subject to *false positives*, that is may report alarms that can never appear at runtime, in any execution environment. An example is the following

```
      0    5    10   15   20   25   30   35   40   45
      ----|----|----|----|----|----|----|----|----|
0   % cat false-alarm.c
1   void main()
2   {
3     int x, y;
4     if ((-4681 < y) && (y < 4681) && (x < 32767) && (-32
5   767 < x) && ((7*y*y - 1) == x*x)) {
6       y = 1 / x;
7     };
8   }
      ----|----|----|----|----|----|----|----|----|
      0    5    10   15   20   25   30   35   40   45
```

for which ASTRÉE produces a false positive

```
9   % astree --exec-fn main false-alarm.c |& egrep "WARN"
10  false-alarm.c:5.9-14::[call#main@1:]: WARN: integer di
11  vision by zero ([-32766, 32766] and {1} / Z)
12  %
```

since it is unable to prove the mathematical fact $\forall x, y \in \mathbb{Z} : 7y^2 - 1 \neq x^2$.

It is therefore very easy to produce (even automatically) infinitely many programs on which ASTRÉE will produce at least one false alarm. The real question is therefore how can ASTRÉE be made precise enough on programs of practical interest, at a reasonable computation cost?

9 Termination

Some static analyzers are subject to non termination, running forever until exhaustion of resources. This is the case of static analysis by model-checking a finite model of the program obtained by predicate abstraction [29] with counterexample guided abstract refinement (CEGAR) [9, 34]. In predicate abstraction, program states are finitely partitioned into blocks corresponding to atomic predicates. A finite program model is derived assuming that concrete execution transitions between states are abstracted by partition transitions covering all execution steps of all states in any block of the partition. Because the abstract state space is finite, it can be exhaustively explored, if small enough, by a model-checker. In case of alarm along an abstract partition path, a concrete path has to be found to find if it is effectively feasible. Otherwise CEGAR will split a block of the partition to make the analysis more precise. The limit of this process is to explore all possible concrete executions. In particular the same false alarm may be considered by counterexamples of increasing length, leading CEGAR not to terminate (or, rather, terminate by memory or time exhaustion). Examples of static analyzers subject to that problem are BLAST [2], MAGIC [8], and SLAM [1].

For example, ASTRÉE produces no false alarm on the following program

```
0   % cat slam.c
1   int main()
2   { int x, y;
3     x = 0; y = 0;
4     while (x < 2147483647)
5        { x = x + 1; y = y + 1; }
6     __ASTREE_assert((x == y));
7   }
8   % astree --exec-fn main slam.c |& egrep "WARN"
9   %
```

while SLAM [1] being unable to generate the essential intermediate predicate `y = x - 1` via CEGAR will not properly terminate (since all counterexamples $x = i + 1 \wedge y = i$, $i = 0, 1, 2, 3, \ldots$ will be successively generated).

10 Scaling-Up

As long as one does not worry about termination, it is very easy to make precise static analyzers (e.g. by enumeration of all possible executions or by using sophisticated abstractions like Presburger's arithmetics in Omega [71]), but the real difficulty is to scale up. This requires a careful choice of the abstractions used in the static analyzer to get a good trade-off between cost and precision. ASTRÉE was explicitly designed with objective in mind. For example ASTRÉE does not use the polyhedral abstraction [19] (whose cost is, in practice, exponential in the number of variables). ASTRÉE has shown to scale up on industrial codes of up to 1,000,000 LOCs.

11 Abstraction

Abstraction consists in reasoning on the set of possible runtime executions of a program, as defined by the concrete semantics, using a simpler abstract model of this semantics [14]. An abstraction is *sound* if it does not omit program behaviors; an abstraction is *incomplete* if it adds spurious program behaviors and thus precludes proving some properties that are true of the original system but no longer true on the abstraction.

11.1 Manual Abstraction

Some static analyzers require a manual abstraction since they do not accept executable programs but micro-models of programs written in specification languages (like PROMELA in SPIN [36] or Alloy in the Alloy Analyzer [46]). The abstraction should be sound for the considered properties (for example ignoring a test in a program is always sound for proving invariance properties like runtime errors but may be invalid for liveness properties like termination).

Such specification models (e.g. in SIMULINK® [42] or SCADE™ [38]) can be useful if the code can be automatically generated from the specification. But the translation often does not preserve all properties of this specification (typically real numbers in the specification become floats in the generated program so stable computations in the model may be unstable in the program, or inversely). If some properties were proved correct on the idealized specification, they may not necessarily hold on the automatically generated code. It follows that static analyses of both the specification and the program are required for soundness.

11.2 Automatic [In]finitary Abstraction

Since manual abstraction is hard and costly, ASTRÉE performs an automatic abstraction of the program concrete semantics. The abstract properties of the abstract semantics should be machine-representable and can be taken either in finite (*finitary abstraction*) or in infinite sets (*infinitary abstraction*). Model-checking based static analyzers like such as Bandera [44], Bogor [75], Java PathFinder [77], SPIN [36], VeriSoft [28] can only use finitary abstractions. Infinitary abstractions are provably more powerful than finitary ones [16], even a potentially infinite sequence of finitary ones as in CEGAR. However infinitary abstract semantics may not be computable. In this case, some static analyzers, like ESC Java 2 [26], may require some help in the effective computation of the abstract semantics (e.g. in the form of loop invariants).

In contrast, ASTRÉE computes the abstract semantics iteratively using convergence acceleration techniques with widening/narrowing to enforce termination [13, 14]. This can be parameterized by the end-user (e.g. by providing likely thresholds for interval ranges). For example, on the following toy example

```
0  % cat TwoCountersAutomaton.c
1  typedef enum {F=0,T=1} BOOL;
2  int main () {
3    volatile BOOL B;
4    int x = 0,  y= 0;
5    while ((x < 2147483646) && (y > -2147483647)) {
6      __ASTREE_assert((y <= x) && (0 <= x) && (y != x +
7  1));
8      if (B) {
9        if (y == 0) { x = x + 1; y = x; }
10     } else { y = y - 1; }
11   }
12 }
13 % cat TwoCountersAutomaton.config
14 __ASTREE_volatile_input((B [0,1]));
15
```

(where the configuration file indicates that `B` can be externally modified as opposed to a mere use of `volatile` to prevent compiler optimizations), ASTRÉE is able to prove the user assertion correct

```
16 % astree --exec-fn main --config-sem TwoCountersAutoma
17 ton.config TwoCountersAutomaton.c |& egrep "WARN"
18 %
```

whereas the CEGAR approach will consider counterexamples of increasing length and thus will fail on this toy example.

11.3 Multi-Abstraction

Some static analyzers use a single encoding of abstract program properties like BDDs in model-checkers, symbolic constraints in BANE [27] and BANSHEE [48], or the canonical abstraction of TVLA [54]. The advantage of this uniform encoding choice is the ease of design. The inconvenience is the potentially high algorithmic cost when encoding information with inappropriate encodings.

In contrast, ASTRÉE uses many numerical/symbolic abstract domains, some of which are illustrated below. Each abstract domain has an efficient encoding for the specific information it aims at representing together with primitives to propagate this information along the program flow.

11.4 Non-relational Abstractions

Non-relational abstractions such as interval analysis [13, 14] cannot express invariant relations between values of the program variables whence are inexpensive (but most often too imprecise).

ASTRÉE also includes congruence information [30, 59] (to cope with pointer alignment), bit-pattern analysis (to cope with memory masks), finite integer sets (to cope with enumeration types), etc.

11.5 Relational Abstractions

Relational abstractions, expressing relations between objects manipulated by the program, are indispensable for precision.

In the following program, the decrementation of `Y` does not overflow because it is bounded by the test on `X`.

```
    0    5    10   15   20   25   30   35   40   45
    |----|----|----|----|----|----|----|----|----|
0   % cat octagon.c
1   void main() {
2     int X = 10, Y= 100;
3     while (X >= 0)
4       { X--; Y--; };
5     __ASTREE_assert((X <= Y));
6   }
    |----|----|----|----|----|----|----|----|----|
    0    5    10   15   20   25   30   35   40   45
```

This cannot be proved by ASTRÉE using non-relational abstract domains only.

```
7   % astree --no-octagon --exec-fn main octagon.c |& egre
8   p "WARN"
9   octagon.c:4.11-14::[call#main@1:loop@3>=4:]: WARN: sig
10  ned int arithmetic range [-2147483649, 2147483646] not
11   included in [-2147483648, 2147483647]
12  octagon.c:5.19-25::[call#main@1:]: WARN: assert failur
13  e
14  %
```

However, the octagon abstract domain [61] is used by default by ASTRÉE to discover relations of the form $\mathtt{X} \pm \mathtt{Y} \leqslant c$ where `X` and `Y` are program variables (or components of structured variables) and c is a numerical constant automatically discovered by the static analysis.

```
15  % astree --exec-fn main octagon.c |& egrep "WARN"
16  %
```

The imprecision of non-relational analyses is sometimes well-localized and can be enhanced by simple relational analyses. For example, the subtraction of intervals is $x \in [a, b]$ and $y \in [c, d]$ implies $x - y \in [a - d, b - c]$ so if $x \in [0, 100]$ then $x - x \in [-100, 100]$ when applying the general formula without knowing that $y = x$.

```
0   % cat x-x.c
1   void main () { int X, Y;
2     __ASTREE_known_fact(((0 <= X) && (X <= 100)));
3     Y = (X - X);
4     __ASTREE_log_vars((Y));
5   }
6   % astree --exec-fn main --no-relational x-x.c |& egrep
7    "Y in"
8   direct = <integers (intv+cong+bitfield+set): Y in [-10
9   0, 100] /\ Top >
10  %
```

However, the symbolic abstract domain [63] locally propagates the symbolic value of variables and performs simplifications (maintaining the maximal possible rounding error for float computations, as overestimated by intervals, for soundness).

```
11  % astree --exec-fn main x-x.c |& egrep "Y in"
12  direct = <integers (intv+cong+bitfield+set): Y in {0}
13  /\ {0} >
14  %
```

11.6 Abstraction Composition

ASTRÉE has abstract domain constructors allowing for the definition of complex abstract domains as a function of simpler ones, like simple forms of *reduced cardinal power* [15, Sect. 10.2]. For example, decision trees can be used by ASTRÉE for case analysis on abstract values of variables taken in finite domains (e.g. Booleans), with abstract properties chosen in basic domains at the leaves (see Fig. 3).

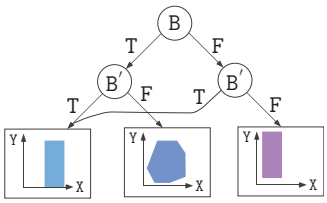

Figure 3. Decision tree (of height 2)

For the following program where control is encoded in Booleans,

```
    0    5    10   15   20   25   30   35   40   45
    |----|----|----|----|----|----|----|----|----|
0   % cat decision.c
1   typedef enum {F=0,T=1} BOOL; BOOL B;
2   int main () {
3       unsigned int X, Y;
4       while (1) {
5           B = (X == 0);
6           if (!B) { Y = 1 / X; }
7            }
8   }
9
    |----|----|----|----|----|----|----|----|----|
    0    5    10   15   20   25   30   35   40   45
```

a non-relational analysis fails

```
10  % astree --no-relational --exec-fn main decision.c |&
11  egrep "WARN"
12  decision.c:6.22-27::[call#main@2:loop@4=1:]: WARN: int
13  eger division by zero ([0, 4294967295] and {1} / Z)
14  decision.c:6.22-27::[call#main@2:loop@4=2:]: WARN: int
15  eger division by zero ([0, 4294967295] and {1} / Z)
16  decision.c:6.22-27::[call#main@2:loop@4=3:]: WARN: int
17  eger division by zero ([0, 4294967295] and {1} / Z)
18  decision.c:6.22-27::[call#main@2:loop@4>=4:]: WARN: in
19  teger division by zero ([0, 4294967295] and {1} / Z)
20  %
```

(the error is reported 4 times because ASTRÉE distinguishes, by default, the first three iterations of the loop and the behavior starting at the 4-th iteration) while the use of decision trees succeeds

```
21  % astree --exec-fn main decision.c |& egrep "WARN"
22  %
```

12 Application Domain Awareness

Beyond the basic abstractions described above which may be found, at least in part, in *general-purpose static analyzers* (such as Airac5 [78], CodeSonar [40], or PolySpace Verifier [20]), some

static analyzers are *specialized* to specific programming style (like C Global Surveyor [76]).

Beyond its specialization for synchronous programs (as e.g. generated by SCADE™ [38]), ASTRÉE is *application domain aware*. Proving the absence of overflows on programs of the kind that ASTRÉE targets, which implement complex numerical filtering, entails proving many numerical properties, including the stability of certain numeric schemes. Without considering these properties, it is impossible to prove the desired properties, and analysis produces false alarms that would be very difficult to eliminate, even by hand.

12.1 Digital Filters

Digital filters typically appear in control/command programs to smooth signals, or extract some spectral components. ASTRÉE can analyze the runtime behavior of digital filters with great precision as shown on the following second order filter example (where the current output `P` is a function of the two previous outputs `S[0,1]`, the current input `X` $\in [-10, 10]$ and the two previous inputs `E[0,1]`)

```
0  % cat filter.c
1  typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
2  BOOLEAN INIT;
3  float P, X;
4  volatile float RANDOM_INPUT;
5  __ASTREE_volatile_input((RANDOM_INPUT [-10.0,10.0]));
6
7
8  void filter2 () {
9    static float E[2], S[2];
10   if (INIT) {
11       S[0] = X; P = X; E[0] = X;
12   } else {
13       P = (((((0.4677826 * X) - (E[0] * 0.7700725)) +
14 (E[1] * 0.4344376)) + (S[0] * 1.5419)) - (S[1] * 0.674
15 0477));
16   }
17   E[1] = E[0]; E[0] = X;
18   S[1] = S[0]; S[0] = P;
19 }
20
21 void main () {
22   X = RANDOM_INPUT;
23   INIT = TRUE;
24   while (TRUE) {
25          X = RANDOM_INPUT;
26     filter2 ();
27     INIT = FALSE;
28   }
29 }
    |----|----|----|----|----|----|----|----|----|
    0    5    10   15   20   25   30   35   40   45

30 % astree --exec-fn main --dump-invariants filter.c |&
31 tail -25 | egrep "Bound on the"
32 Bound on the input                : 10.
33 Bound on the output               : 13.3881164652
34 %
```

On Fig. 4, one can observe that a typical behavior of a filter with random input is not linear whence cannot be captured using octagons [61] or convex polyhedra [19], so ASTRÉE must resort to specialized ellipsoidal abstract domains [23] to provide realistic and non-trivial bounds on the filter output. More complex computations may be necessary in order to provide tight bounds on complex filtering schemes. [23, 66]

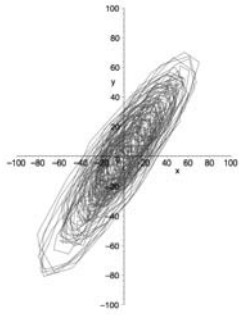

Figure 4. Typical behavior of a 2nd order filter

12.2 Time

The synchronous programs analyzed by ASTRÉE typically have the form of a loop synchronized by a clock. By taking the clock rate and the maximal execution time into account, as given by a configuration file, ASTRÉE can bound the variable `R` below, counting how long a volatile input `I` has been enabled, so as to set up a condition `T` if the condition was continuously enabled during `n` clock ticks (this can be used e.g. to check that a button has been pushed down long enough before reacting)

```
0  % cat clock.c
1  int R, T, n = 10;
2  void main()
3  { volatile int I;
4    R = 0;
5    while (1) {
6      if (I) { R = R+1; }
7      else { R = 0; }
8      T = (R>=n);
9      __ASTREE_wait_for_clock(());
10 }}
11 % cat clock.config
12 /* clock.config */
13 __ASTREE_volatile_input((I [0,1]));
14 __ASTREE_max_clock((3600000));
15 % astree --exec-fn main --config-sem clock.config --du
16 mp-invariants clock.c |& egrep "(WARN)|(R in)"
17    T in [0, 1] /\ Top, R in [0, 3600001] /\ Top, n in
18 {10} /\ {10} >
19 %
```

In absence of time bounds, `I` could be permanently enabled whence leading to an overflow of `R`. Thus, any sound analysis ignoring time bounds should conclude that `R` can overflow. Because of programs such as the above one, ASTRÉE can relate variables incremented by a bounded value to the clock, and can thus bound these variables, given the maximal execution time.

12.3 Time-dependent Deviations

We have just seen that some very simple software modules, such as one counting the number of clock ticks during which some button is pressed, will overflow unless some bound on the execution time is assumed. Recognizing the recurrence $\mathtt{X}(t+1) = [A, A']\mathtt{X}(t) + [B, B'] \cup [C, C']$, ASTRÉE can discover arithmetic-geometric predicates [24] of the form $|\mathtt{X}(t)| \leqslant a(a'i + b')^t + b$ where $\mathtt{X}(t)$ is the value of the program variable `X` at clock time t

and the numerical constants a, b (the bounds for the current computation), a', b' (bounds for computations at the t previous clock ticks), and i (initial bound) are automatically discovered by the static analysis. Indeed this abstract domain was used in the example of Sect. 12.2 (replacing an older less powerful "clock domain" [3]). For example, the accumulation of rounding errors in the following program

```
0  % cat bary.c
1  typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
2  float INIT,C1,I;
3  float RANDOM_INPUT;
4  __ASTREE_volatile_input((RANDOM_INPUT [-1.,1.]));
5
6  void bary () {
7    static float X,Y,Z;
8    if (C1>0.) { Z = Y; Y = X;}
9    if (INIT>0.) { X=I; Y=I; Z=I; }
10   else
11     {X = 0.50000001 * X + 0.30000001*Y + 0.20000001*Z;
12 };
13   __ASTREE_log_vars((X,Y,Z));
14
15 }
16
17 void main () {
18  INIT=1.; C1=RANDOM_INPUT; I=RANDOM_INPUT;
19
20  while (1) {
21    bary();
22    INIT=RANDOM_INPUT; C1=RANDOM_INPUT; I=RANDOM_INPUT;
23
24    __ASTREE_wait_for_clock(());
25  }
26 }
```

computing a barycentric mean, can be bounded for various execution times

```
27 % cat bary10.config
28 __ASTREE_max_clock((3600000));%
29 % astree --exec-fn main --config-sem bary10.config bar
30 y.c |& tail -n100 | egrep "(Z in)|(Time)"
31   <float-interval: Z in [-1.71113, 1.71113], Y in [-1.
32 71113, 1.71113],
33 Time spent in Astrée: 1.156628 s (0 h 0 mn 1 s)
34 % cat bary100.config
35 __ASTREE_max_clock((36000000));%
36 % astree --exec-fn main --config-sem bary100.config ba
37 ry.c |& tail -n100 | egrep "(Z in)|(Time)"
38   <float-interval: Z in [-215.193, 215.193], Y in [-21
39 5.193, 215.193],
40 Time spent in Astrée: 1.150548 s (0 h 0 mn 1 s)
41 % cat bary1000.config
42 __ASTREE_max_clock((360000000));%
43 % astree --exec-fn main --config-sem bary1000.config b
44 ary.c |& tail -n100 | egrep "(Z in)|(Time)"
45   <float-interval: Z in [-2.1295e+23, 2.1295e+23],
46 Time spent in Astrée: 2.619297 s (0 h 0 mn 2 s)
47 %
```

Observe that the static analysis time is significatively longer when the program is assumed to run for 10 or 1000 hours (which would not be the case with testing, simulation or model-checking). This kind of divergence analysis can be used to exhibit potential instability problems.

13 Precision/Cost Trade-off Adaptability

Hiding alarms from users in order not to frighten them is tantamount to burying one's head in the sand. ASTRÉE eschews such unsound policies while producing very few false alarms, thanks to the enhanced precision of its analysis. Nevertheless, ASTRÉE may leave a few false alarms, which can be eliminated by parametrization and analysis directives.

13.1 Abstraction Parametrization

ASTRÉE has a number of parameters to chose between alternative semantics (e.g. whether or not static variables should be considered initialized to 0, how to apply integer promotion following [45, 6.3.1.1/2], etc).

Many options concern the choice of the abstract domains to be used (such as no relational domains or which types of filters should be considered).

Some options concern the iteration strategy (when to unroll loops and to what extent, when to perform widenings, with which thresholds, etc) and the implementation of the analysis (sequential or parallel [65]).

A number of options control ASTRÉE warnings (e.g. whether to warn on overflow in initializers, on implicit conversions, etc), output (such as which invariants should be exported to the viewer), and verbosity (like tracing the analyzer computations).

Most options of ASTRÉE allows for the adaptation of the precision/cost trade-off to the user needs, in particular for relational abstractions which can be very memory and computation-time demanding. For example

- Octagons algorithms are in $\mathcal{O}(n^3)$ where n is the number of variables [61]. This cost is controlled by ASTRÉE by limiting the set of variables to which the abstract domain is applied (*variable packing* [4]).
- Decision trees can grow exponentially in the number of variables. This cost is controlled by variable packing and by providing the maximal height of decision trees (default 3).
- Similarly, the maximum depth of symbolic constant trees propagated by ASTRÉE is an adjustable parameter (default 20).
- Arrays are handled either as a collection of individual elements or by smashing all elements. The end-user can choose which arrays should be smashed (by giving a global threshold parameter or using abstraction directives for individual arrays).
- The end-user can choose which type of commands or which functions should be partitioned.
- etc.

Sometimes ASTRÉE is too precise and reducing its precision by parameter adjustment (e.g. eliminating some widening thresholds) can speed-up the analysis while introducing no false alarm.

13.2 Automatic Abstraction Directives

Some expensive analyses (like partitioning for case analysis) need not be done everywhere in the program. In that case, ASTRÉE will guide the analysis by automatically inserting directives in the program thanks to pattern-matched predefined program schemata.

For example *trace partitioning* [57] consists in partitioning executions in blocks where executions along each control-flow path are analyzed separately, all cases being joined together after some time. A typical example is the computation of the value of a tabulated function

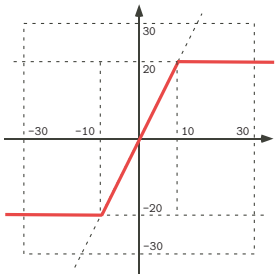

Figure 5. Tabulated function

```
% cat trace-partitioning.c
void main() {
float t[5] = {-10.0, -10.0, 0.0, 10.0, 10.0};
float c[4] = {0.0, 2.0, 2.0, 0.0};
float d[4] = {-20.0, -20.0, 0.0, 20.0};
float x, r;
  __ASTREE_known_fact(((-30.0 <= x) && (x <= 30.0)));
  int i = 0;
  while ((i < 3) && (x >= t[i+1])) {
    i = i + 1;
  }
  r = (x - t[i]) * c[i] + d[i];
  __ASTREE_log_vars((r));
}
```

as shown in Fig. 5. A simple analysis joins all cases for `i` and considers all possible combinations of any slope `c[0,3]` with any abscisse `d[0,3]`, whence is rather imprecise

```
% astree --exec-fn main --no-partition trace-partition
ing.c |& egrep "r in "
direct = <float-interval: r in [-100, 100] >
%
```

Delaying abstract unions in tests and loops by trace partitioning is more precise[5].

```
% astree --exec-fn main trace-partitioning.c |& egrep
"r in "
direct = <float-interval: r in [-20, 20] >
%
```

13.3 Manual Abstraction Directives

It may happen that the generation of parametric directives in the code which is programmed in ASTRÉE for a specific application domain may not be effective on other styles of programming. This is the case on the following example (from [15, Sect. 10.2])

```
    0    5    10   15   20   25   30   35   40   45
    |----|----|----|----|----|----|----|----|----|
% cat repeat1.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;
  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
```

[5] for non-distributive abstract domains and is much less expensive than disjunctive completion [15].

for which ASTRÉE signals a false alarm.

```
% astree --exec-fn main repeat1.c |& egrep "WARN"
repeat1.c:5.8-13::[call#main@2:loop@4>=4:]: WARN: sign
ed int arithmetic range [-2147483649, 2147483646] not
included in [-2147483648, 2147483647]
%
```

Obviously, the false alarm is raised because of a missing relation between `b` and `x`. The manual directive for the decision trees abstract domain will allow ASTRÉE to discover this relation.

```
% cat repeat2.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;
  __ASTREE_boolean_pack((b,x));
  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
```

Now, the program is shown to be runtime error free.

```
% astree --exec-fn main repeat2.c |& egrep "WARN"
%
```

If necessary, ASTRÉE could be easily modified to include a partitioning strategy for such cases.

14 Extensibility and Adaptability

When ASTRÉE has no abstract domain to express a program invariant which is indispensable for proving a program correct, there is no other way than designing a new abstract domain to cope with the inexpressiveness problem (we have seen examples, like octagons in Sect. 11.5 or filters in Sect. 12.1).

ASTRÉE has a *modular design* since an analyzer instance is built by selection of OCAML modules from a rich collection, each implementing an abstract domain, some of which being publicly available [47]. It follows that ASTRÉE is *extensible* in that it is easy to include new abstractions in the analyzer to cope with imprecision.

To be useful, the new abstract domains must interact with the existing ones and reciprocally, to enhance the global precision of the analysis. In particular, the conjunction of the pieces of information encoded in each abstract domain is realized by an approximate *reduced cardinal product* [15, Sect. 10.1] discussed in [18].

For example when the memory model of ASTRÉE was extended to cope with `union` and pointer arithmetics [62], an arithmetic congruence analysis [30, 59] was added to cope with pointer alignments. Being able to interact with the existing basic interval domain, it can yield a precise analysis of the following program

```
% cat congruence.c
void main () { int I = 0;
        while (I < 13) { I = I + 2; }
        __ASTREE_log_vars((I));
}
% astree --exec-fn main congruence.c |& grep "I in"
direct = <integers (intv+cong+bitfield+set): I in {14}
 /\ Top >
%
```

while a mere interval analysis would yield $\mathtt{I} \in [13, 14]$ (or even $\mathtt{I} \geqslant 13$ for imprecise ones).

15 Conclusion

Sophisticated static program analysis techniques, looking for deep program runtime properties, have not found their way into widespread use in low-quality software production environments in which easily designed, unsound, and imprecise static analyzers, looking for superficial error patterns, can be very successful (provided they do not discourage their users by performing a careful selection of the most likely alarms thus minimizing apparent false positives).

In such low-quality software production environments false negatives are no problem since there always remain enough bugs in the program, or enough new ones are introduced when trying to correct old ones, so that the analyzer will always find some bug to report on, with a reasonable probability of finding an actual bug in a handful of false positives ([79] cites common rates of 50 false alarms for an actual error). Finding bugs in large programs in this empirical way may keep programmers busy for a very long time so any help, even of low quality is welcomed. However, software not developed rigorously, have very little chance to become high-quality software, whichever dynamic/static analyzer is used to find bugs.

In high-quality software production environments, which are extremely rare, unsound and imprecise static analyzers will not be satisfactory because they provide even less guarantees than testing, e.g. with respect to control and data coverage. In mission and safety critical software, the problem is not to find some of the remaining bugs but to definitely prove the absence of any bug, at least in a given category of unacceptable ones (like software errors abruptly stopping the computer).

For checking the absence of runtime errors, ASTRÉE has the unprecedented qualities of being

- sound (no false negatives),
- precise (no false or very few positives) thanks to knowledge of the domain of synchronous control/command for which it was designed, and its capacity of adaptation either by the designers or, to a lesser extend, by knowledgeable end-users
- terminating (and efficient: 1/2 hours per 100,000 LOCs),
- scaling-up (to 1.000.000 LOCs).

Whence ASTRÉE is not only a reliable debugging aid, it is a *rigorous formal verifier* which can be included in a stringent certification process. Hopefully ASTRÉE can contribute to the production of high-quality software in a cost-effective, timely, and reproducible manner.

References

[1] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *29th ACM Symp. on Principles of Prog. Lang., POPL '02*, pp. 1–3, 2002.

[2] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with BLAST. In *8th Int. Conf. on Fundamental Approaches to Soft. Eng., FASE '05*, LNCS 3442, pp. 2–18. Springer, 2005.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pp. 85–108. Springer, 2002.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM Conf. on Prog. Lang. Design and Impl., PLDI '03*, pp. 196–207, 2003.

[5] S. Boldo and J.-C. Filliâtre. Formal Verification of Floating-Point Programs. In *18th IEEE Int. Symp. on Computer Arithmetic, ARITH '18*, 2007.

[6] B. Brew and M. Johnson. Value Lattice Static Analysis, A New Approach to Static Analysis. *Dr. Dobbs J.*, 2001.

[7] W. Bush, J. Pincus, and D. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Soft. Pract. and Exp.*, 30(7):775–802, 2000.

[8] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. *IEEE Trans. on Soft. Eng.*, 30(6):388–402, 2004.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. Acm*, 50(5):752–794, 2003.

[10] E. Clarke and D. Kroening. ANSI-C Bounded Model Checker User Manual. Technical report, School of Computer Science, Carnegie Mellon University, 2006.

[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS '04*, LNCS 2988, pp. 168–176. Springer, 2004.

[12] T. Copeland. *PMD Applied*. Centennial Books, 2005.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*, pp. 106–130, Paris, France, 1976. Dunod.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. 4th ACM Symp. on Principles of Prog. Lang., POPL '77*, pp. 238–252, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conf. Rec. 6th ACM Symp. on Principles of Prog. Lang., POPL '79*, pp. 269–282, 1979.

[16] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *4th Int. Symp. Prog. Lang. Implementation and Logic Programming, PLILP '92*, LNCS 631, pp. 269–295. Springer, 1992.

[17] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *14th European Symp. on Prog. Lang. and Systems, ESOP '05*, LNCS 3444, pp. 21–30. Springer, 2005.

[18] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In *11th Asian Comp. Sci.*

Conf., ASIAN 06, LNCS. Springer, 2006.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. 5th ACM Symp. on Principles of Prog. Lang., POPL '78*, pp. 84–97, 1978.

[20] A. Deutsch. Static Verification Of Dynamic Properties. PolySpace Technologies, `www.polyspace.com`.

[21] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *18th ACM Symp. on Operating Systems Principles*, 2001.

[22] M. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems J.*, 15(3):258–287, 1976.

[23] J. Feret. Static Analysis of Digital Filters. In *13th European Symp. on Prog. Lang. and Systems, ESOP '2004, Barcelona, Spain*, LNCS 2986, pp. 33–48. Springer, 2004.

[24] J. Feret. The Arithmetic-Geometric Progression Abstract Domain. In *6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI '2005, Paris, France*, LNCS 3385, pp. 42–58. Springer, 2005.

[25] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *6th Int. Conf. on Formal Engineering Methods, ICFEM '04*, LNCS 3308, pp. 15–29. Springer, 2004.

[26] C. Flanagan, K. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conf. on Prog. Lang. Design and Impl., PLDI '02*, pp. 234–245, 2002.

[27] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *7th Int. Sym. on Static Analysis, SAS '00*, LNCS 1824, pp. 175–198. Springer, 2000.

[28] P. Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[29] S. Graf and H. Saïdi. Verifying Invariants Using Theorem Proving. In *8th Int. Conf. on Computer Aided Verification, CAV '97*, LNCS 1102, pp. 196–207. Springer, 1996.

[30] P. Granger. Static Analysis of Arithmetical Congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.

[31] D. Grossman, M. Hicks, T. Jim, and G. Morrisett. Cyclone: a Type-safe Dialect of C. *C/C++ Users J.*, 23(1), 2005.

[32] S. Gupta and G. Sreenivasamurthy. Navigating "C" in a "leaky" boat? Try Purify. *www-128.ibm.com/developerworks/rational/library/06/0822_satish-Giridhar/*, 2006.

[33] L. Hatton. *Safer C: Developing for High-Integrity and Safety-Critical Systems*. McGraw-Hill, 1995.

[34] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29th ACM Symp. on Principles of Prog. Lang., POPL '02*, pp. 58–70. ACM Press, 2002.

[35] G. Holzmann. UNO: Static Source Code Checking for User-Defined Properties. In *6th World Conf. on Integrated Design and Process Technology, IDPT '02*, 2002.

[36] G. Holzmann. *The* SPIN MODEL CHECKER*, Primer and Reference Manual*. Addison-Wesley, Sept. 2003.

[37] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM Workshop on Program Analysis For Software Tools and Engineering, PASTE '05*, pp. 13–19, 2005.

[38] Esterel Technologies. SCADE Suite™, The Standard for the Development of Safety-Critical Embedded Software in the Avionics Industry. `www.esterel-technologies.com/products/scade-suite/`.

[39] Gimpel Software®. PC-lint™/ FlexeLint™ Value Tracking. `www.gimpel.com`.

[40] GrammaTech ®. CodeSonar: A code-analysis tool that identifies complex bugs at compile time. `www.grammatech.com/products/codesurfer/`.

[41] Klocwork®. Klocwork K7™. `www.klocwork.com`.

[42] The MathWorks. Simulink® — Simulation and Model-Based Design. `www.mathworks.com/products/simulink/`.

[43] Reasoning, Inc. Reasoning inspection service defect data, Tomcat, version 4.1.24. *www.reasoning.com/pdf/Tomcat_Defect_Report.pdf*, 2003.

[44] R. Iosif, M. Dwyer, and J. Hatcliff. Translating Java for Multiple Model Checkers: The Bandera Back-End. *Formal Methods in System Design*, 26(2):137–180, 2005.

[45] ISO/IEC. *International standard – Programming languages – C*, 1999. Standard 9899:1999.

[46] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press. Cambridge, MA., 2006.

[47] B. Jeannet and A. Miné. The `Apron` Numerical Abstract Domain Library. `apron.cri.ensmp.fr/library/`.

[48] J. Kodumal and A. Aiken. Banshee: A Scalable Constraint-Based Analysis Toolkit. In *7th Int. Sym. on Static Analysis, SAS '07*, LNCS 3672, pp. 218–234. Springer, 2005.

[49] N. Kumar and R. Peri. Transparent Debugging of Dynamically Instrumented Programs. *ACM SIGARCH Computer Architecture News*, 33(5):57–62, 2005.

[50] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *2001 USENIX Security Symposium, Washington, D.C.*, 2001.

[51] K. Leino and G. Nelson. An Extended Static Checker for Modula-3. In *7th Int. Conf. on Compiler Construction, CC '98*, LNCS 1383, pp. 302–305. Springer, 1998.

[52] X. Leroy. Coinductive Big-Step Operational Semantics. In *15th European Symp. on Prog. Lang. and Systems, ESOP '2006*, LNCS 3924, pp. 54–68. Springer, 2006.

[53] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conf. Rec. 33rd ACM Symp. on Principles of Prog. Lang., POPL'06*, pp. 42–54, 2006.

[54] T. Lev-Ami, R. Manevich, and M. Sagiv. TVLA: A System for Generating Abstract Interpreters. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pp. 367–376. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2004.

[55] M. Martel. An Overview of Semantics for the Validation of Numerical Programs. In *6th Int. Conf. on Verification, Model Checking, and Abstract Interpretation, VMCAI '05*, LNCS 3385, pp. 59–77, 2005.

[56] L. Mauborgne. ASTRÉE: verification of absence of run-time error. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pp. 385–392. Kluwer Acad. Pub. Dordrecht, The Netherlands, 2004.

[57] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzer. In *14th European Symp. on Prog. Lang. and Systems, ESOP '05*, LNCS 3444, pp. 5–20. Springer, 2005.

[58] R. Milner. A theory of type polymorphism in programming. *J. of Comp. and Sys. Sciences*, 17:348–375, 1978.

[59] A. Miné. A Few Graph-Based Relational Numerical Abstract Domains. In *9th Int. Symp. on Static Analysis, SAS '02*, LNCS 2477, pp. 117–132. Springer, 2002.

[60] A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *13th European Symp. on Prog. Lang. and Systems, ESOP '04*, LNCS 2986, pp. 3–17. Springer, 2004.

[61] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[62] A. Miné. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics. In *ACM Conf. on Languages, Compilers, and Tools for Embedded Systems, LCTES '2006*, pp. 54–63, 2006.

[63] A. Miné. Symbolic Methods to Enhance the Precision of Numerical Abstract Domains. In *7th Int. Conf. on Verification, Model Checking and Abstract Interpretation VMCAI '06*, LNCS 3855, pp. 348–363. Springer, 2006.

[64] MISRA (The Motor Industry Software Reliability Association). Guidelines for the use of the C language in vehicle based systems Software. `www.misra.org.uk`, 1998.

[65] D. Monniaux. The Parallel Implementation of the ASTRÉE Static Analyzer. In *3rd Asian Symp. on Prog. Lang. and Systems, APLAS '05*, LNCS 3780, pp. 86–96. Springer, 2005.

[66] D. Monniaux. Compositional Analysis of Floating-Point Linear Numerical Filters. In *17th Int. Conf. on Computer Aided Verification, CAV '05*, LNCS 3576, pp. 199–212. Springer, 2005.

[67] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proc. 27th ACM SIGSOFT Int. Conf. on Software engineering*, pp. 580–586. ACM Press, 2005.

[68] G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[69] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conf. on Prog. Lang. Design and Impl., PLDI '07*, 2007.

[70] A. Pnueli, O. Shtrichman, and M. Siegel. The Code Validation Tool CVT: Automatic Verification of a Compilation Process. *Int. J. on Soft. Tools for Tech. Trans.*, 2(2):192–201, 1998.

[71] W. Pugh and D. Wonnacott. Static Analysis of Upper and Lower Bounds on Dependences and Parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, 1994.

[72] F. Randimbivololona, J. Souyris, P. Baudin, A. Pacalet, J. Raguideau, and D. Schoen. Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In *World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1709, pp. 1798–1815. Springer, 1999.

[73] X. Rival. Abstract Interpretation Based Certification of Assembly Code. In *4th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAIS '03*, LNCS 2575, pp. 41–55. Springer, 2003.

[74] X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Conf. Rec. 31st ACM Symp. on Principles of Prog. Lang., POPL '01*, pp. 1–13, 2004.

[75] Robby, M. Dwyer, and J. Hatcliff. Domain-specific Model Checking Using The Bogor Framework. In *21st IEEE/ACM Int. Conf. on Automated Software Engineering, ASE '06, Tokyo, Japan*, 2006.

[76] A. Venet and G. Brat. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In *Int. Conf. on Prog. Lang. Design and Impl., PLDI '04*, pp. 231–242. ACM Press, 2004.

[77] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Soft. Eng. J.*, 10(2), 2003.

[78] K. Yi, H. Choi, J. Kim, and Y. Kim. An Empirical Study on Classification Methods for Alarms from a Bug-Finding Static C Analyzer. *Inf. Proc. Let.*, 102(2-3):118–123, 2007.

[79] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk. On the value of static analysis for fault detection in software. *IEEE Trans. on Soft. Eng.*, 32(4):240–253, 2006.

[80] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Operating System Design and Implementation, OSDI '06*. USENIX Assoc., 2006.