A gentle introduction to formal verification of computer systems by abstract interpretation

Patrick COUSOT [a], Radhia COUSOT [b]
[a] *École normale supérieure and New York University*
[b] *École normale supérieure and CNRS*

Abstract. We introduce and illustrate basic notions of abstract interpretation theory and its applications by relying on the readers general scientific culture and basic knowledge of computer programming.

Keywords. Abstract interpretation, Formal methods, Verification, Static analysis.

1. Introduction

Software is in all mission-critical and safety-critical industrial infrastructures since it is, in principle, the cheapest and most effective way to control complex systems in real time. However, all computer scientists have experienced costly bugs in embedded software. The failure of the Ariane 5.01 maiden flight [40] (due to an overflow), the failure of the Patriot missile [57] during the Gulf war (due to an accumulated rounding error), the loss of Mars orbiter [1] (due to a unit error), the crash of the twin-engined Chinook ZD 576 helicopter [8][1] are a few examples showing that mission-critical and safety-critical software can be far from being safe.

Computer scientists agree on the fact that it is preferable to verify that mission-critical or safety critical software (and nowadays hardware) programs do not go wrong before running them. As an alternative to testing, which hardly scales up at reasonable costs and with a satisfactory coverage, automated formal verification has emerged, this last decade, as a promising useful complement, with interesting potential industrial applications.

Formal methods date back to the early days of computer science (the Floyd/Naur/Hoare verification method [33,51,37] appeared in the sixties with antecedents going back to Von Neumann and Turing [38]) and the power of present-day computers make them applicable to large scale industrial projects. The idea is to make automatic proofs at compile-time to verify program runtime properties.

[1] "In the summer of 1993 an independent defence IT contractor, EDS-SCICON, was instructed to review the FADEC [Full Authority Digital Engine Control] software; after examining only 18 per cent of the code they found 486 anomalies and stopped the review." [8]

Beyond the difficulty of specifying which runtime properties are of interest, all formal methods are faced with undecidability (the mathematical impossibility for a computer, which is a finite device, to prove for sure non-trivial properties of the (infinite or extremely large) runtime behaviors of computer programs) and complexity (the impossibility for computers to solve decidable questions within a reasonable amount of time and memory for large input data, such as program executions observed over very long periods of time).

Besides testing which is not a formal method, three main approaches have been considered for formal verification, all of them being approximations of the program semantics (formally defining the possible executions in all possible environments) formalized by abstract interpretation theory:

- Deductive methods produce formal mathematical correctness proofs using theorem provers or proof assistants and need human interaction to provide inductive arguments (which hardly scales up for large programs which are modified over long periods of times) and help in proofs (such as proof hints or strategies);
- Model-checking exhaustively explores finitary models of program executions, which can be subject to combinatorial explosion, requires the human production of models (or may not terminate in case of automatic refinement of the model). An alternative is to explore partially the model but this is then debugging, not verification;
- Static analysis, which automates the abstraction of the program execution, always terminates but can be subject to false alarms (that is warnings that the specification may not be satisfied although no actual execution of the program can violate this specification).

In this paper, we explain informally and intuitively the underlying ideas of abstract interpretation-based static analysis, which consists in abstracting the program semantics (formalizing program executions) to provide a sound over-approximation of the potential errors with respect to a specification.

Static analysis involves abstractions of the program semantics that must be coarse enough to be effectively computable and precise enough to imply the properties required by the specification. To show that such abstractions do exist for given families of applications and specifications, we report on the ASTRÉE analyzer (`www.astree.ens.fr/`), which is a static analyzer for proving the absence of runtime-errors in synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language. It has been successfully applied to prove the absence of runtime errors in the control-command part of the primary flight control software of the fly-by-wire system of airplanes.

2. Software bugs

2.1. Numerical software bugs

Let us start by considering classical bugs in numerical computations.

2.1.1. Integer bugs

The following C program reads an integer n (typed at the keyboard) and prints the factorial $n! = \texttt{fact}(n) = 2 \times \ldots \times (n-1) \times n$.

```
#include <stdio.h>

int fact (int n ) {
int r, i;
r = 1;
for (i = 2; i<=n; i++) {
r = r * i;
}
return r;
}

int main() {
int n;
scanf("%d",&n);
printf("fact(%d)=%d\n",n, fact(n));
}
```

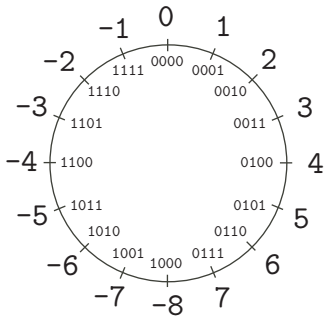
Compiling and testing the program yields

```
ä gcc fact.c -o fact.exec
% ./fact.exec
3
fact(3)=6
% ./fact.exec
4
fact(4)=24
%
```

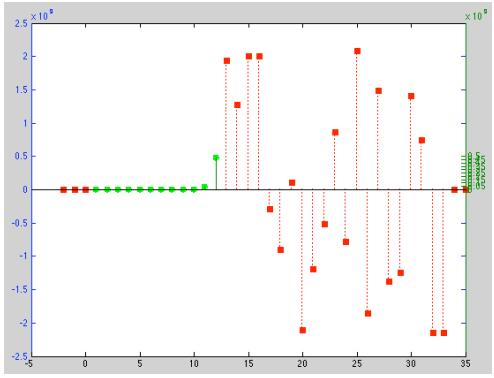
Bad surprises come when trying further inputs.

```
% ./fact.exec
100
fact(100)=0
% ./fact.exec
20
fact(20)=-2102132736
%
```

Computers having a finite memory cannot use mathematical integers but do use signed modular arithmetic on N bits, typically $N = 32$ or $N = 64$. The N-bit two's complement case is illustrated below for the case $N = 4$. The integers are represented outside the circle and their respective binary representations inside.



A positive number is in binary form. A negative number is the binary representation of the positive number obtained by adding this negative number to 2^N (for example, -7 has the binary representation of $2^4 - 7 = 9$ which is 1001). It follows that the first bit on the left is the sign (0 for positive and 1 for negative). The two's-complement system has the advantage of not requiring that the addition and subtraction circuitry examine the signs of the operands to determine whether to add or subtract. For example $3+(-4)$ is 0011+1100 = 1111 that is -1. This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic. Also, zero has only one representation (i.e. $+0$ and -0 are 0000). The inconvenience is that machine integers can no longer be understood as the mathematical integers. For example $7+2 = -7$ and $7+9 = 0$. This modulo arithmetics is often misunderstood by programmers which leads to numerous program bugs and explains the zero and negative results for the factorial. So `fact`(n) is not $n! = 2 \times 3 \times \ldots \times n$ and indeed they do coincide only for $1 \leqslant n \leqslant 12$, as shown by the figure below.



Notice that the recursive definition of the factorial in OCAML [39]

```
let rec fact n = if (n = 1) then 1 else n * fact(n-1);;
```

yields different results

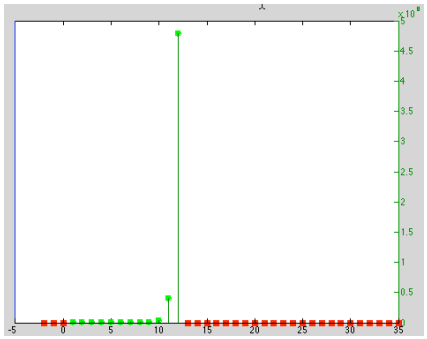
fact(n)	C	OCaml
fact(1)	1	1
...	...	...
fact(12)	479001600	479001600
fact(13)	*1932053504*	*-215430144*
fact(14)	*1278945280*	*-868538368*
fact(15)	*2004310016*	*-143173632*
fact(16)	*2004189184*	*-143294464*
fact(17)	-288522240	-288522240
fact(18)	-898433024	-898433024
fact(19)	109641728	109641728
fact(20)	*-2102132736*	*45350912*
fact(21)	*-1195114496*	*952369152*
fact(22)	-522715136	-522715136
fact(23)	862453760	862453760
fact(24)	-775946240	-775946240
fact(25)	*2076180480*	*-71303168*
fact(26)	*-1853882368*	*293601280*
fact(27)	*1484783616*	*-662700032*
fact(28)	*-1375731712*	*771751936*
fact(29)	*-1241513984*	*905969664*
fact(30)	*1409286144*	*-738197504*
fact(31)	738197504	738197504
fact(32)	*-2147483648*	*0*
fact(33)	*-2147483648*	*0*
fact(34)	0	0

This is because integers are represented on 31 bits instead of 32 (one bit is used for garbage collection). Moreover, `fact(-1)` yields a stack overflow due to a looping recursion (instead of `fact(-1)=1` for the C program).

A correct version of the factorial program should avoid overflows and provide a standard answer in case the function is undefined (such as an exception or returning 0 as chosen below).

```
% cat -n fact_lim.c
     1 int MAXINT = 2147483647;
     2 int fact (int n) {
     3 int r, i;
     4 if (n < 1) || (n = MAXINT) {
     5 r = 0;
     6 } else {
     7 r = 1;
     8 for (i = 2; i<=n; i++) {
     9 if (r <= (MAXINT / i)) {
    10 r = r * i;
    11 } else {
    12 r = 0;
    13 }
    14 }
    15 }
    16 return r;
    17 }
    18
    19 int main() {
    20 int n, f;
    21 f = fact(n);
    22 }
```

The function computed by the program is now more intelligible.

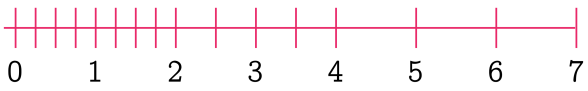


Notice that `i++` at line 8 does not overflow since `i` $\leqslant$ `n` $<$ `MAXINT`. At line 9, `MAXINT / i` does not divide by zero since `i` $\geqslant 2$ in the loop. At line 10, `r * i` does not overflow since `r <= (MAXINT / i)`. Such a proof can be done automatically by ASTRÉE (the absence of warning meaning that all arithmetic operations in the program are well-defined in the mathematical integers).

```
% astree --exec-fn main --unroll 12 fact_lim.c |& grep WARN
%
```

2.1.2. Float bugs

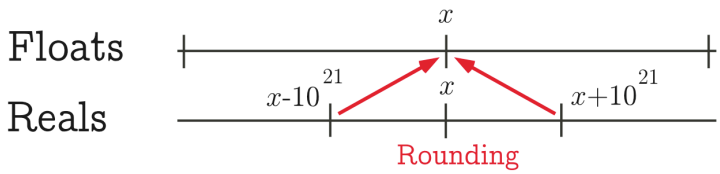
The mathematical models designed by engineers (e.g. to control physical systems) are built upon the mathematical reals $\mathbb{R}$. The specification languages (such as SIMULINK™ and SCADE™) use mathematical reals in $\mathbb{R}$ to describe program models. Because mathematical reals are hard to represent in machines (π has infinitely many decimals), they are replaced in programming languages by floating point numbers as defined by the IEEE 754 Standard [3] (or its variants on numerous machines). Floats do not behave at all like reals $\mathbb{R}$ or even rationals $\mathbb{Q}$ [35,50] because they represent only a finite subset of $\mathbb{Q}$. For example, on six bits the 16 positive floats are distributed on a real segment as follows:



Observe that the density is small for large floats so the distance between two consecutive floats is much larger than the distance between two consecutive small floats. The float arithmetic operations are equivalent to taking the float arguments, performing the operation on the reals, and then converting the real result to a float by rounding (either to the nearest, towards 0, $+\infty$ or $-\infty$ which can be chosen by the programmer). There can be exceptions due to range violations, divisions by zero, and indeterminate or undefined cases such as $0/0$ or $\sqrt{x}$ with $x < 0$ [36]. Otherwise the result may be a float but most often there is a rounding error which is too often neglected by programmers, and is a source of surprises. For example the mathematical identity $(x+a)-(x-a) = 2a$ is no longer valid in the floats, as shown by the following program for which a naïve interpretation in the reals $\mathbb{R}$ should definitely yield $2a$ = `2.0e21`.

```
int main () {
float x, y, z, r;
x = 1.000000019e+38;
y = x + 1.0e21;
z = x - 1.0e21;
r = y - z;
printf("%f\n", r);
}
% gcc float-error.c
% ./a.out
0.00000
```

The explanation of this phenomenon is that, by definition of the floating point computation, `x + 1.0e21` and `x - 1.0e21` should be evaluated in the reals and then rounded to the closest float which happens to be `x = 1.000000019e+38` in both cases so that their difference is zero.

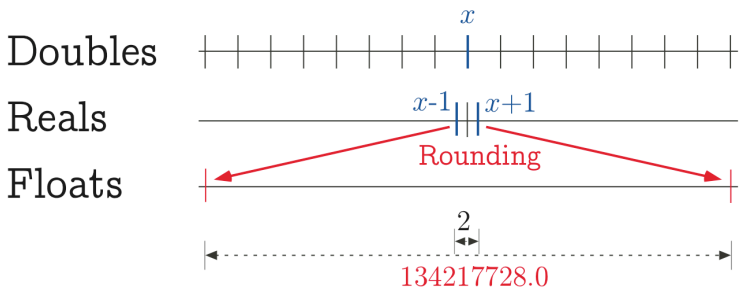


If we now consider mixing floats (e.g. for sensors) with doubles (e.g. for precise program computations), we can have a program like the following one where the expected result is now surprisingly very large (although $2a = 2$ in the reals $\mathbb{R}$):

```
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951488.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
134217728.000000
```

(1)

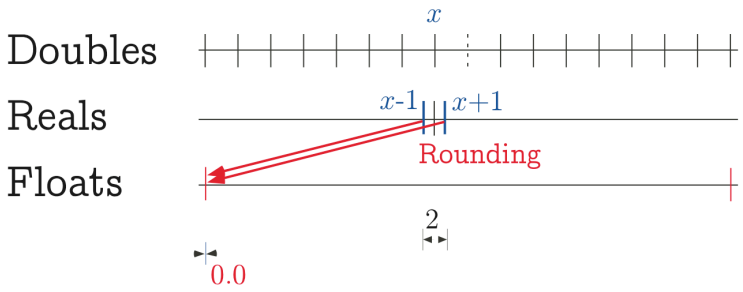
This is because of the rounding of the expressions $x+1$ and $x-1$ to the closest floats which are very far from one another:



Although one can produce such an example for each double exactly located between any two consecutive floats, this remains a rare event likely to be omitted by tests, since if the double value is not exactly in the middle, the rounding will be towards the same float and the result will be 0. For example, just changing the last digit in the above example, one gets:

```
% cat double-error.c
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951487.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
0.000000
```

x is not exactly in the middle of two consecutive floats, so that the values of both $x+1$ and $x-1$ are rounded to the same float.



Examples of famous bugs on floats include the Patriot missile bug due to a drift of a software clock incremented by $\frac{1}{10}$th of the second, which is not exact in binary (since $(0.1)_{10} = (0.\ 0001100110011001100\ldots)_2$) [57] and the Excel 2007 bug [41] where 77,1 $\times$ 850 which is 65.535 displays as 100.000 due to a rounding error during the translation of IEEE 754 floats in 64 bits into a Unicode character string which yields to an erroneous

alignment in a conversion table for exactly six numbers between 65534.99999999995 and 65535 and six between 65535.99999999995 and 65536.

2	65535-2^(-37)	100000		65536-2^(-37)	100001
3	65535-2^(-36)	100000		65536-2^(-36)	100001
4	65535-2^(-35)	100000		65536-2^(-35)	100001
5	65535-2^(-34)	65535		65536-2^(-34)	65536
6	65535-2^(-36)-2^(-37)	100000		65536-2^(-36)-2^(-37)	100001
7	65535-2^(-35)-2^(-37)	100000		65536-2^(-35)-2^(-37)	100001
8	65535-2^(-35)-2^(-36)	100000		65536-2^(-35)-2^(-36)	100001
9	65535-2^(-35)-2^(-36)-2^(-37)	65535		65536-2^(-35)-2^(-36)-2^(-37)	65536

2.1.3. Memory usage bugs

Besides numerical bugs, bugs due to bad memory usage (such as buffer overruns and memory leaks) are also frequent. Here is a trivial example of infinite memory allocation in C which looks so frequent that the runtime warning advices on how to debug the program.

```
% cat leak.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
     while (malloc(1000));
     return 0;
}% gcc leak.c
% ./a.out
a.out(87070) malloc: *** mmap(size=16777216) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
%
```

Such bugs can be avoided, e.g. in safety critical software by disallowing dynamic memory allocation. But this programming practice rule does not prevent allocating memory within arrays, thus paving the way for buffer overruns. The results of buffer overruns can sometime be surprising such as shown by the following example.

```
#include <stdio.h>
int main () { int n, T[1];
n = 2147483647;
printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

which yields completely different results on different machines:

`n = 2147483647, T[n] = 2147483647`	Macintosh PPC
`n = 2147483647, T[n] = -1208492044`	Macintosh Intel Core Duo
`n = 2147483647, T[n] = 0`	Macintosh Intel Core 2 Duo

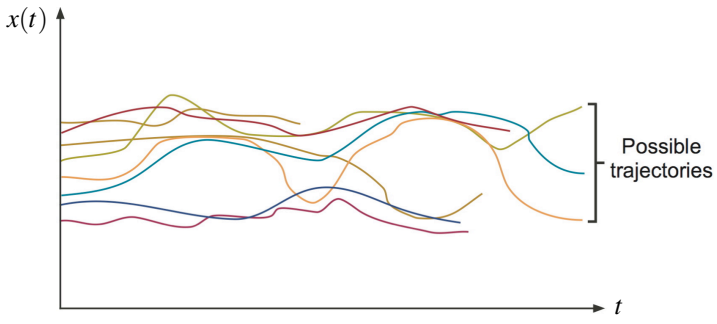
```
n = 2147483647, T[n] = -135294988      PC Intel 32 bits
Bus error                              PC Intel 64 bits
```

The software bugs that we have illustrated are common to all software and so can be tracked without requiring a program-specific specification that often exists in an informal form only, if any. Such bugs are therefore a choice morsel for formal methods since the specification follows from the semantics of the programming language and so requires no additional effort from end-users. We now introduce abstract interpretation, and its application to static analysis, in particular for tracking such pervasive runtime errors.

3. An informal introduction to abstract interpretation

Abstract interpretation is a theory of sound approximation of the semantics of programming languages [9,14,15,21] which main application is static analysis that is the automatic, compile-time determination of run-time properties of programs which can be used, among others, for formal verification of computer systems. Abstraction, as formalized by abstract interpretation, is tantamount to all formal methods (deductive methods to derive predicate transformers or Hoare logic [12] from the program operational semantics, software model-checking to get a model of the program to be checked [18], and static analysis to get an abstract semantics of the program [14]).

We can imagine the behavior of a computer system (e.g. program executions/semantics) as trajectories showing the evolution of the state $x(t)$ of the system as a function of the time t (which is discrete for computer-based controllers). The state can be e.g. values of the program variables. The evolution is governed by the program depending upon its execution environment (e.g. the evolution of volatile variables or periodic inputs). The program execution can be finite (e.g. due to a runtime error) or infinite (in case of non-termination).



An example of trace for the factorial function call `fact(4)` is

```
int fact (int n ) {
  int r = 1, i;
  for (i=2; i<=n; i++) {
    r = r*i;
  }
  return r;
}
```

n ← 4; r ← 1;

i ← 2; r ← 1 × 2 = 1;

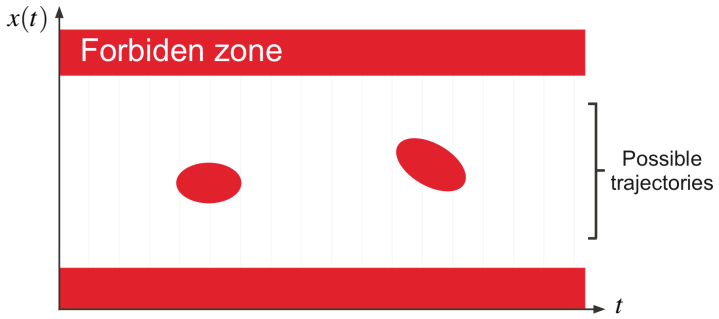
i ← 3; r ← 2 × 3 = 6;

i ← 4; r ← 6 × 4 = 24;

i ← 5;

return 24;

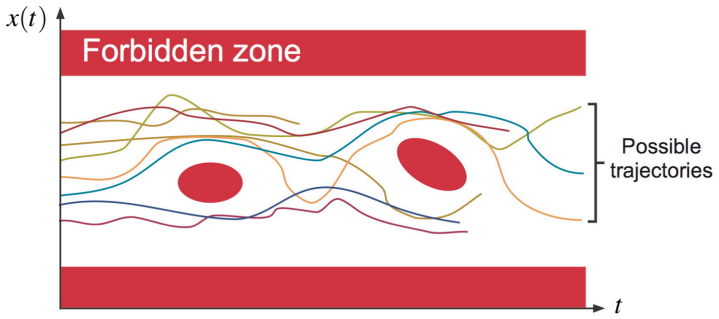
The specification of the system safety can be given in the form of forbidden zones which should not be crossed by any safe trajectory (e.g. the state might be bounded and the island might represent the avoidance of isolated obstacles, such as a division by 0):



A specification of absence of runtime errors for the factorial program could be

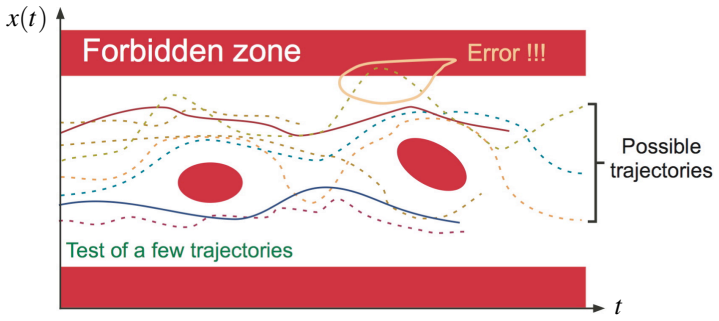
```
int fact (int n ) {
  int r, i;
  r = 1;
  for (i=2; i<=n; i++) {    ←  no overflow of i++
    r = r*i;                ←  no overflow of r*i
  }
  return r;
}
```

A formal proof of correctness consists in proving that the semantics implies the specification that is none of the possible execution traces does reach the forbidden zone.

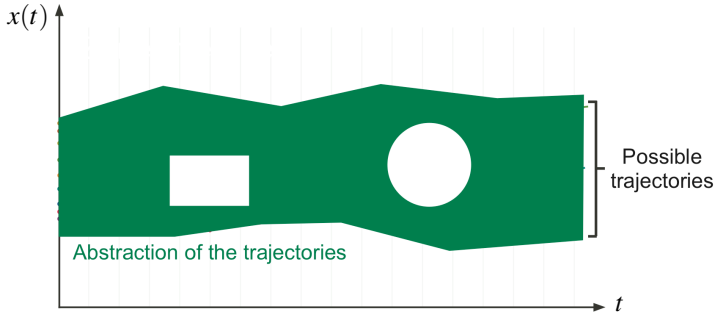


Testing or debugging consists in exploring a few trajectories (or a prefix of the trajectories for infinite ones). Therefore testing computes an under-approximation of the program

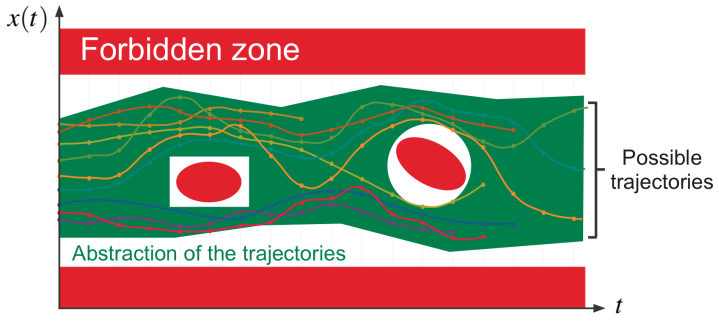
semantics, which is fundamentally unsafe since some erroneous trajectories might be forgotten:



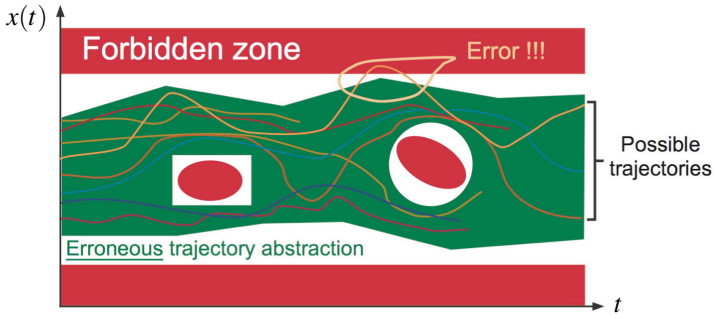
The idea of abstraction for static analysis is to compute an over-approximation of the possible trajectories that is both computer representable and effectively computable from the program text (so without any program execution):



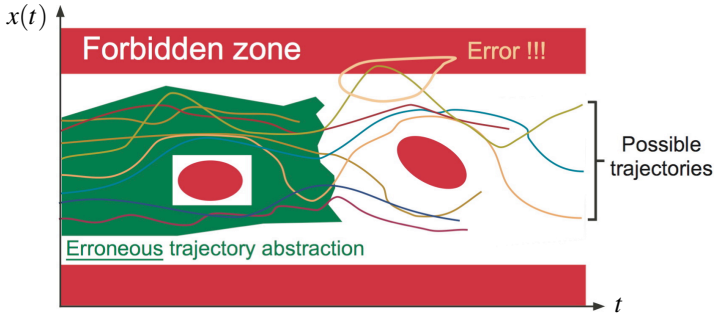
The fundamental idea of abstract interpretation is to ensure soundness (no error can be forgotten by the analysis) by a full coverage of all possible trajectories. The proof that the program satisfies the specification follows from the fact that all trajectories are in the abstraction and this abstraction does not intersect the bad states.



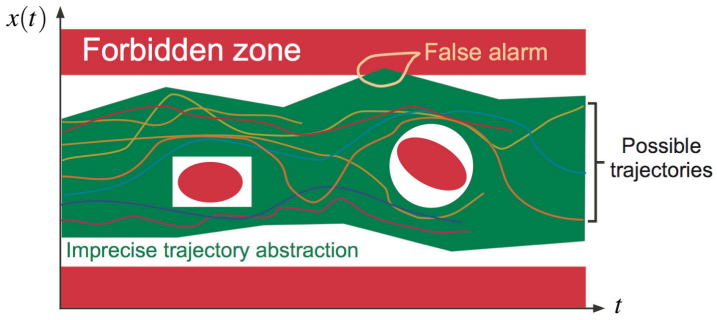
The theory of abstract interpretation is sound and so systematically excludes the following unsound abstraction where one erroneous execution is not covered:



Another example of an unsound abstraction is provided by bounded model-checking [4] where the model of the program is explored partially (maybe starting in the middle of the program execution) but this does not exclude missing an error because the over-approximation is partial:

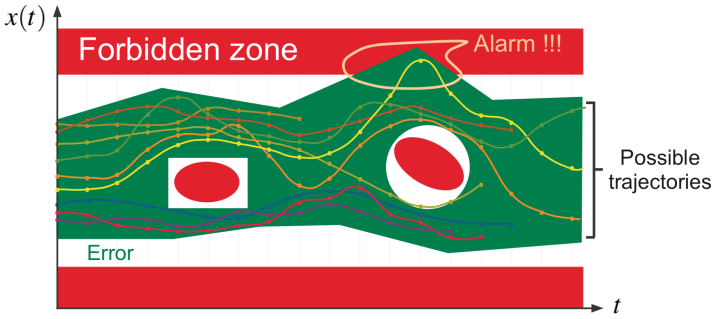


In static analysis, which is always sound, the abstraction may yield an over-approximation which is too large and covers the forbidden zone:



(2)

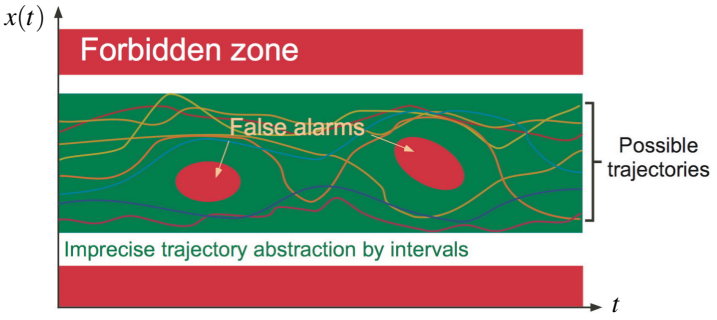
Since the static analysis provides the safe envelop but no other information on the trajectories other than the coverage, it is impossible in that case to decide whether there is a trajectory in the envelop going into the unsafe zone (actual error), as follows



or none (false alarm) as in shown in figure (2).

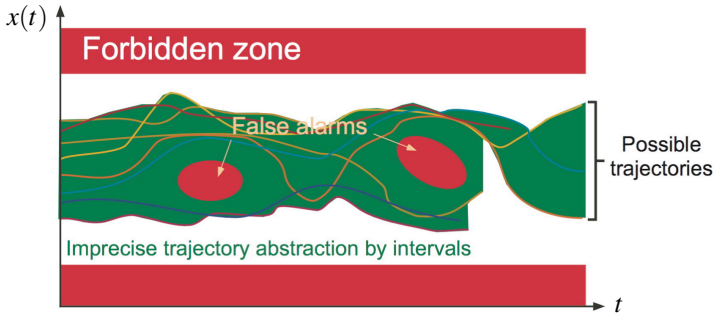
So testing is an under-approximation which can miss errors (false negatives) while static analysis is an over-approximation which cannot miss any potential error but may yield false alarms (false positives), that is cases where the abstraction looks unsafe while the actual executions are all safe. The objective is therefore to get precise enough abstractions (since unsound ones are excluded, by principle).

The design of a static analyzer can start with classical abstractions. An example is the interval abstraction that overestimates the lower and upper bounds of states. This can exclude some types of errors (e.g. overflows) but certainly not all, as shown in the following example:

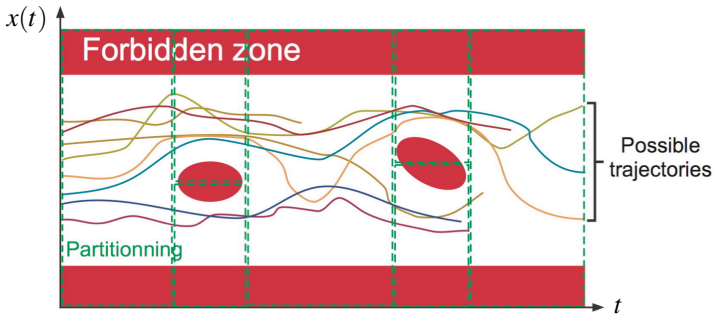


By analyzing the origin of the false alarms, the designers of the static analyzer can design refinements of the abstraction[2]. In our example, one can consider an interval abstraction as a function of the time, which is certainly more precise than a time-independent overestimation:

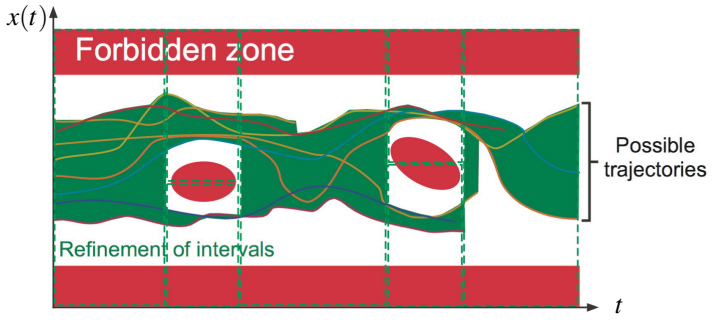
[2]The most abstract abstraction which is precise enough to make the proof can be defined formally but is neither automatically computable [11,34] nor computer representable for complex infinite systems.



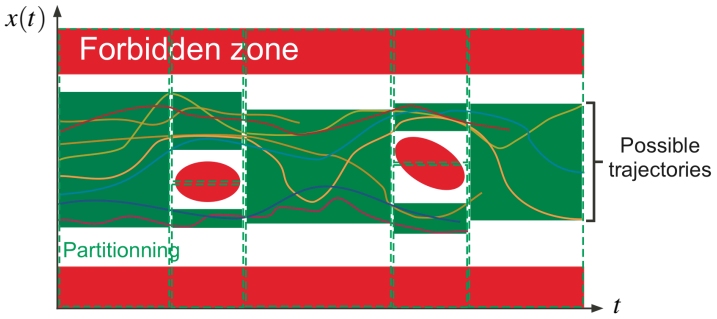
However this is not precise enough. Observe that if the trajectories do avoid the islands, there must be an appropriate detection mechanism in the code, such as a conditional. It follows that the state space of the program can be partitioned according to such conditionals so as to distinguish between true and false alternatives:



Notice that at the end of the conditional the blocks of the partition are merged in order to avoid combinatorial explosion e.g. if such a conditional appears in the loop. One can then apply the interval idea within each block of the partition to get:



Now the safety proof is finished. However, using an interval as a function of time is a bit costly, so the performances of the static analysis can certainly be improved by using a time independent abstraction within each block of the partition:



Similarly loops must be folded to ensure termination. The design of a static analyzer requires a subtle equilibrium between the precision and the cost of the abstraction, more precise abstraction avoiding more false alarms while, generally, coarser abstractions reduce computation costs.

Many reasonings on programs and more generally systems can be formalized by abstract interpretation such as

- typing and type inference [10],
- abstract model-checking [18],
- parsing [20,22],
- program transformation [19],
- the definition of semantics and logics at various levels of abstraction [12,17,23],
- static analysis to check for the absence of bugs [7],
- the verification of security protocols [5]
- abstract modelling of live systems [28],
- etc.

4. Application of abstract interpretation to static analysis

4.1. Equations

In order to illustrate the application of abstract interpretation to static analysis, let us consider the following program assumed, for simplicity, to compute with the mathematical integers $\mathbb{Z}$.

$\{y \geqslant 0\}$ $\leftarrow$ pre-condition hypothesis

```
x = y
```

$\{I(x,y)\}$ $\leftarrow$ loop invariant

```
while (x > 0) {
  x = x - 1;
}
```

The problem is to compute the invariant which characterizes the reachable states upon loop entrance, just before the iteration test. The Floyd-Hoare-Naur verification conditions are the following:

$$(y \geqslant 0 \wedge x = y) \Longrightarrow I(x,y) \qquad \textit{initialization condition}$$

$$(I(x,y) \wedge x > 0 \wedge x' = x-1) \Longrightarrow I(x',y) \qquad \textit{iteration condition}$$

The *initialization condition* states that the invariant should hold upon loop entry (hence is a consequence of the pre-condition hypothesis propagated through the assignment of `y` to `x`). The *iteration condition* states that if the loop invariant holds after any number of iterates and one more iteration of the loop is performed (so that the iteration test does hold) then the invariant must remain true after this iteration (that is after the decrement of `x`).

This can be rewritten in an equivalent fixpoint equation $I = F(I)$ of the form

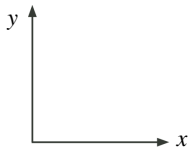
$$I(x,y) = x \geqslant 0 \wedge (x = y \vee I(x+1,y))$$

which expresses that the invariant holds at the current iterate if it is the first iterate or did hold at the previous iterate (and so because `x` is decremented in the loop body did hold for the previous value of `x` which is the current one incremented by one). We look for the strongest invariant, the one that implies all others. Consequently, we are interested in the least solution to the equation for logical implication (where A is "less than or equal to" B whenever $A \Longrightarrow B$) that is $I = F(I)$ and $I' = F(I')$ implies that $I \Longrightarrow I'$.

4.2. Iteration with convergence acceleration

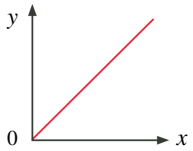
This least solution can be computed as the limit $I = \lim_{n\to\infty} F^n(\text{false})$ of the following iterates (presented together with their geometric interpretation in the plane). The iterates start with false, that is the empty set, meaning that no state is reachable.

$$I^0(x,y) = \text{false}$$



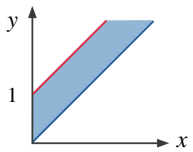
The first iterate describes the states that are reachable the first time the loop entry is reached, that is `x` $= x =$ `y` $\geqslant 0$ that is the bisecting half-line of the first quarter of plane.

$$\begin{aligned} I^1(x,y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1,y)) \\ &= 0 \leqslant x = y \end{aligned}$$



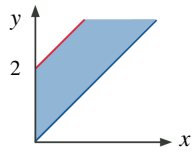
The second iterate describes the states that are reachable the first time the loop entry is reached, or after one loop iterate, which is a strip of height 1 over the bisecting.

$$\begin{aligned} I^2(x,y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1,y)) \\ &= 0 \leqslant x \leqslant y \leqslant x+1 \end{aligned}$$



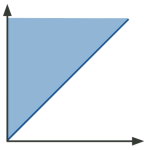
The third iterate describes the states that are reachable the first time the loop entry is reached, or after one or two loop iterates, which is a strip of height 2 over the bisecting.

$$\begin{aligned} I^3(x,y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1,y)) \\ &= 0 \leqslant x \leqslant y \leqslant x+2 \end{aligned}$$



We can go forever like this. A mathematician would make an inductive reasoning by guessing that after n iterates, we have a description of the states that are reachable the first time the loop entry is reached, or after at most $n-1$ loop iterates, which is a strip of height $n-1$ over the bisecting. Replacing in the fixpoint equation, one would prove that this remains true for $n+1$ hence is true for all $n \geqslant 0$ and passing to the limit, would conclude that the reachable states are exactly described by $0 \leqslant x \leqslant y$. However, we must automate this formal reasoning knowing that computers are not good at guessing automatically inductive hypotheses and not much better at proofs by recurrence. The idea in abstract interpretation is to use a simple operation known as *widening* [13,14] which in our case, will consist in observing two consecutive iterates $I^2(x,y)$ and $I^3(x,y)$ and forgetting about the constraints which are not stable between these iterates (that is in the above example $y \leqslant x+1$ and $y \leqslant x+2$).

$$\begin{aligned} I^4(x,y) &= I^2(x,y) \triangledown I^3(x,y) \leftarrow \text{widening} \\ &= 0 \leqslant x \leqslant y \end{aligned}$$



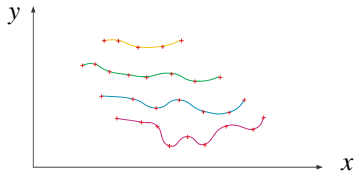
One more iteration shows that we have reached a fixpoint (which may not be the desired least one, but is implied by this least fixpoint, and so is a valid loop invariant, albeit not necessarily the strongest one).

$$\begin{aligned} I^5(x,y) &= x \geqslant 0 \wedge (x = y \vee I^4(x+1,y)) \\ &= I^4(x,y) \quad \text{fixpoint!} \end{aligned}$$

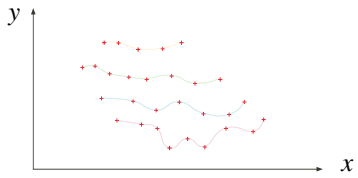
In the above iterates we used logical formulae (or their geometric interpretations). Such formulae can grow exponentially large (although their growth can be limited by widenings). In abstract interpretation-based static analysis, one restricts the form of such formulae in a way that allows for efficient machine representations.

4.3. Abstractions

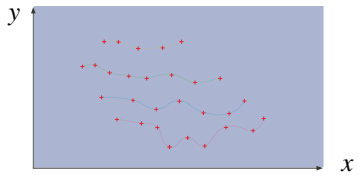
Assume for example that we have to abstract the following set of traces (more generally an infinite set of finite or infinite sequences of states).



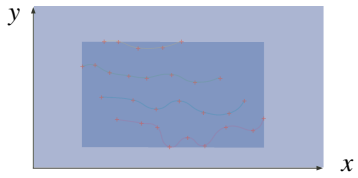
By collecting the set of all states appearing along any of these states, we get a global invariant that is a set of reachable states during execution (which can be equivalently decomposed into a set of local invariants on memory states attached to each program point, meaning that the local invariant does hold whenever the program point is reached during execution, if ever). So we get a generally infinite set of points $\{(x_i, y_i) : i \in \Delta\}$, which formalizes Floyd/Naur proof method and Hoare logic [12].



A simple abstraction is by quarters of plane covering all the states. This provides a sign analysis [15] that is local invariants of the form $x \geqslant 0, y \geqslant 0$.

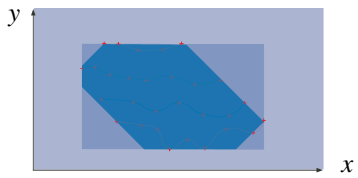


A more refined abstraction is that of intervals [13,14] recording only the minimum and maximum value of variables and thus ignoring their mutual relationships. This provides invariants of the form $a \leqslant x \leqslant b$, $c \leqslant y \leqslant d$ where a, b, c, d are numerical constants automatically discovered by the analysis.



This abstraction is too coarse to prove the invariant $0 \leqslant x \leqslant y$ in the above Section 4.2.

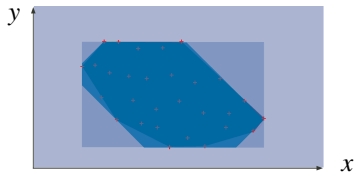
A more refined abstraction is that of octagons [47]



The octagonal abstraction discovers invariants of the form $x \leqslant a$, $x - y \leqslant b$ or $x + y \leqslant c$ and their inverses where a, b, c are numerical constants automatically discovered by the analysis.

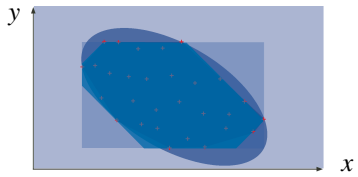
This abstraction is now precise enough to prove the invariant $0 \leqslant x \leqslant y$ in the above Section 4.2. Moreover each term considered in the iterates is exactly representable geometrically as an octagon (although the internal computer representation is a matrix, not symbolic as above). Observe that the equation $I = F(I)$ is an octagon transformer mapping an octagon to another octagon. It follows that the iterates with widening exactly represent the computation which would be performed by such an octagonal analysis.

A more precise abstraction is given by considering convex polyhedra [27].



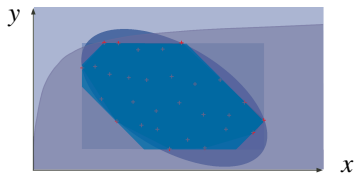
This yields linear inequalities such as $a.x + b.y \leqslant c$ where a, b, c are numerical constants automatically discovered by the analysis.

A typical example of non-linear abstraction is provided by ellipsoïds [32].



The ellipsoïdal abstraction is able to discover invariants of the form $(x-a)^2 + (y-b)^2 \leqslant c$ where a, b, c are numerical constants automatically discovered by the analysis.

Another non-linear abstraction is the exponential [31]



where invariants have the form $a^x \leq y$.

The various abstractions can be combined efficiently in static analyzers such as ASTRÉE to get very precise abstractions of the set of all possible execution traces.

5. The ASTRÉE analyzer

5.1. The considered programming language an its operational semantics

Designed according to the principles of abstract interpretation very roughly sketched above, the ASTRÉE analyzer (`www.astree.ens.fr/`) [6,7,24,25,26,30,31,42,43,44,45,

46,48,49,53,54,55] can formally verify the absence of runtime errors in C programs with Boolean, integer and floating point computations; structures and arrays; pointers (including on functions); conditionals, loops and function calls; limited branching (forward goto, break, continue), pointer arithmetics and unions without dynamic memory allocation, recursive function calls, unstructured backward branching, conflicting side effects (the ASTRÉE analyzer checks the absence of ambiguous side effects since otherwise the semantics of the C program would not be well-defined), C libraries and system calls (but for the synchronization on a clock tick or the use of stubs).

This subset of C corresponds to the specific application domain of large safety-critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

The operational semantics of the language, defining all possible computations at execution is defined by the international norm of C (ISO/IEC 9899:1999), restricted by implementation-specific behaviors depending upon the machine and compiler (such as the representation and size of integers, the IEEE 754-1985 norm for floats and doubles, etc), restricted by user-defined programming guidelines (such as no modular arithmetic for signed integers, even though this might be the hardware choice), restricted by program specific user requirements (such as assert commands, static variables cannot be assumed to be initialized to 0 or execution stops after the first runtime error since the semantics of C is not clearly defined in case of runtime error and this is equivalent if the absence of such errors can be formally proved) finally restricted by a volatile environment as specified by a trusted configuration file (e.g. providing bounds on the possible values of some sensors).

5.2. Implicit Specification: Absence of Runtime Errors

The specification formally verified by ASTRÉE is that of absence of runtime error in a broad sense (including potential memory corruptions and memory leaks). It is therefore implicit and requires no user intervention except maybe to design the configuration file. ASTRÉE checks that no program execution violates the norm of C (e.g. array index out of bounds, division by zero), the absence of implementation-specific undefined behaviors (e.g. maximum short integer is 32767, floating point exceptions, NaN and Inf results), the absence of violation of the programming guidelines (e.g. Booleans are all 0 or 1) and that there is no violation of the programmer assertions (which must all be statically verified).

5.3. Characteristics of ASTRÉE

ASTRÉE computes an abstraction of the feasible traces hence the reachable/attainable states as defined by the program operational semantics. The analysis is static (i.e. at compile time analysis) as opposed to run time analysis (during program execution).

ASTRÉE is a program analyzer whence analyses the semantics of programs as compiled and executed (not user-provided models of the program behavior).

ASTRÉE is automatic (and requires no end-user intervention or external tool such as theorem provers).

The analysis is sound whence covers the whole state space. It never omits potential errors or sort out the most probable ones.

ASTRÉE is multi-abstractions that is uses many different numerical/symbolic abstract domains (as opposed to e.g. symbolic constraints of a given form or a canonical abstraction).

ASTRÉE is infinitary (all abstractions use infinite abstract domains with widening/narrowing [9,14,15,21] as opposed to analyzers which use finitary abstractions which are provably less powerful [16]).

ASTRÉE is efficient in that it always terminates (as opposed e.g. to counterexample-driven analyzers with automatic abstraction refinement which termination is not guaranteed).

ASTRÉE is specializable. It can easily incorporate new abstractions (and reduction with already existing abstract domains [15,26]), as opposed to general-purpose analyzers.

ASTRÉE is domain-aware since it knows about control/command (e.g. digital filters, feedback loops) (as opposed to specialization to a mere programming style).

ASTRÉE is parametric in that the precision/cost of the analysis can be tailored to the user needs by options and directives in the code. ASTRÉE incorporates an automatic parametrization (so that by default end-users can use ASTRÉE without requiring a detailled understanding of the tool). This means that the generation of parametric directives in the code can be programmed (to be specialized for a specific application domain).

The design of ASTRÉE is modular in that it can have different instances built by selection of modules from a collection each implementing an abstract domain suitable for analyzing various aspects of programs.

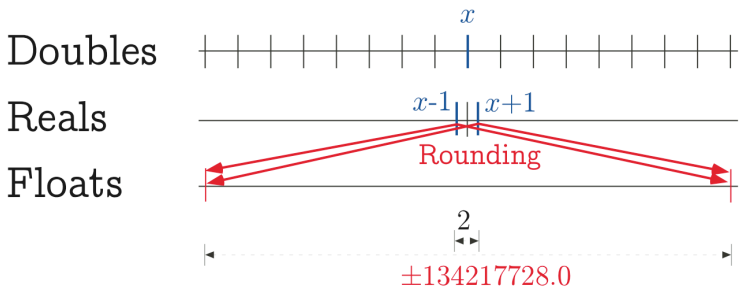
Thanks to these unique characteristics, ASTRÉE can be made precise, producing very few or no false alarm when conveniently adapted to an application domain. It can then performs static proofs of absence of runtime error without any false alarm, whence becomes a formal verifier.

5.4. General-purpose abstract domains

ASTRÉE is built upon general-purpose abstract domains such as intervals [9,13,14] ($a \leqslant x \leqslant b$ where x is the value of a variable at a program location and a, b are constants determined by the analysis) which is the minimum information required to be able to check the implicit specification) and octagons [44,45,48] ($x \pm y \leq c$ where x, y are values of program variables and c is a constant automatically determined by the analysis) which is more precise but too costly but when used locally. The difficulty is to be sound with floating-point computations [44,45]. ASTRÉE is able to analyze such programs with numerous floating point computations in a sound way, which is beyond the state of the art of many other formal methods. For example, the analysis of program (1) yields

```
% astree --exec-fn main --print-float-digits 10 arrondi3.c \
  |& grep "r in "
direct = <float-interval: r in [-134217728, 134217728]} >
```

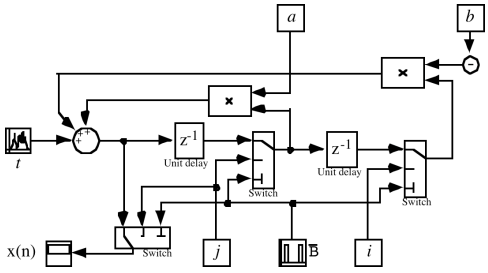
This is because, in absence of knowledge of the rounding mode, ASTRÉE considers the worst of all possible roundings (either to $+\infty$, $-\infty$, 0, or to the closest) on each float operation, as follows



hence the possibility to obtain -134217728.

5.5. Domain-aware abstract domains

ASTRÉE is precise because it incorporates and applies abstractions which are tailored to its application domain, that of synchronous feedback control/command software. Such programs are often produced according to a strictly disciplined programming methodology where over 75% of the code is automatically generated from a high-level specification language describing block diagrams as illustrated below in SIMULINK™:



After discretization, the corresponding program is the following:

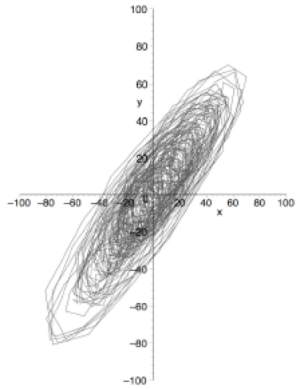
```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT;
float P, X;

void filter () {
  static float E[2], S[2];
  if (INIT) {
      S[0] = X;
      P = X;
      E[0] = X;
  } else {
      P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4)) +
          (S[0] * 1.5)) - (S[1] * 0.7));
  }
  E[1] = E[0];
  E[0] = X;
  S[1] = S[0];
  S[0] = P;
}
```

This program computes a second order filter of the form $X_n = I_n$ in case of reinitialization and $X_n = \alpha X_{n-1} + \beta X_{n-2} + Y_n$ otherwise where Y_n can be bounded by other abstractions.

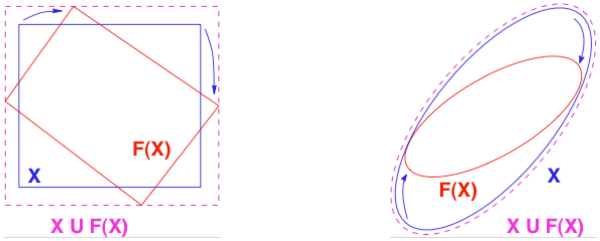
For the purpose of illustration, we can add some context and call the filter function as follows

```
void main () {
  X = 0.2 * X + 5;
  INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35;
    filter ();
    INIT = FALSE;
  }
}
```

An example of execution is shown below (X_{n+1} as a function of X_n, with a segment from a point at time n to the next one at time $n+1$)



An attempt to compute a convex hull U of the possible execution at time n (in the form of intervals, octagons, polyhedra, etc) will result at time $n+1$ in a convex hull $F(U)$ which is a little turned and shrunk. Because the corners of $F(U)$ are outside the original hull U, the union of U and $F(U)$ has to be expanded into a larger hull, which is unstable and ultimately covers all the plane. On the contrary an ellipsoid provides an ultimately stable over-approximation of digital filters (see [30] for details going beyond this simplified explanation):

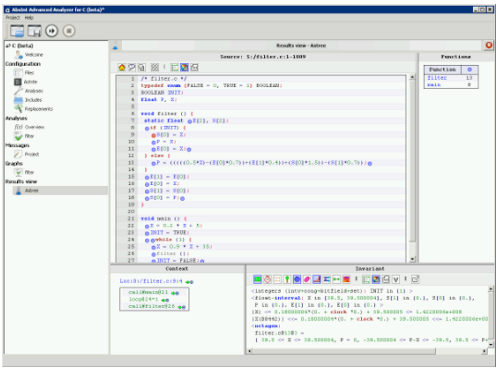


Thanks to this abstraction ASTRÉE will determine that S[0], S[1] are all in [-1327.02698354, 1327.02698354] which is beyond the scope of static analyzers using polyhedral abstractions or automatic refinements (an infinite series of counter examples will be produced as illustrated by the above execution trace).

5.6. Formal verification of embedded software

Thanks to a careful adaptation of the abstraction to control-command programs ASTRÉE was able to prove the absence of runtime errors in two large and critical synchronous avionic software developed by an European civil plane manufacturer. The design of the analyzer started with general-purpose abstract domains and went on first by refinement with application dependent abstractions (such as partitioning [43,55]) and then by application dependent abstractions [30,31] until all false alarms were solved. Because the software is generated automatically, the analyzer will perform well (in precision and performance i.e. a few hours and hundred of megabytes for hundreds of thousands of lines of code) on all programs in the family. This is essential since such programs are maintained over very long periods of time and the formal verification must be able to follow this evolution at extremely low costs. More recently, the scope of application of AS-

TRÉE was extended by considering more general memory models (with union) as found e.g. in communication of untyped data [46]. It is now commercially available from AbsInt Angewandte Informatik (`www.absint.de/astree/`) which has produced so far a professional user interface. Further improvements are pending.



5.7. Conclusion

To understand the numerical world, that is properties of computerized systems, several levels of abstraction are necessary. Abstract interpretation theory ensures the coherence between these abstractions and offers effective approximation techniques. By choosing abstractions that are coarse enough to be effectively computable and precise enough to avoid false alarms, one can fight undecidability and complexity in the analysis and verification of models and programs. Moreover abstract interpretation applies not only to computerized systems but also to evolving systems which discrete or continuous complex behavior can be formally described as a function of time (image processing [56], quantum computing [52], live systems [28], etc).

Many classical applications of abstract interpretation in computer science tolerate a small rate (typically 5 to 15%) of false alarms. For example an optimizing compiler will not perform a program transformation in case of uncertainty of its applicability; typing will reject some correct programs; worst-case execution time analysis (WCET) [29] will slightly overestimate the maximal execution time because e.g. of an imprecise cache analysis [2]. On the contrary recent applications of abstract interpretation such as automatic formal verification are more demanding since they require no false alarm. This was shown to be theoretically possible [11] and ASTRÉE shows that this is practically feasible for well-defined families of similar programs.

To go beyond the actual state of the art, one must be able to consider asynchronous concurrency (with additional potential runtime errors such as data races, deadlocks, livelocks, etc.), functional properties (such as reactivity), the verification of systems (quasi-synchrony, distribution), the grand challenge being to be able to perform formal verification from system specifications or models to the machine code.

The ultimate objective is to change the way programs are designed and validated. Present day control of the design process and testing should evolve to include a formal verification of the final product.

Acknowledgements

The development of the ASTRÉE Static Analyzer (Nov. 2001 – Nov. 2009) by Bruno Blanchet (Nov. 2001 – Nov. 2003), Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux (Nov. 2001 – Aug. 2007), and Xavier Rival was supported in part by the École normale supérieure, the CNRS, the French exploratory project ASTRÉE of the Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL), and, since 2007, by INRIA .

References

[1] A.G. Stephenson Et Al. Mars Climate Orbiter mishap investigation boord phase I report. Technical report, George C. Marshall Space Flight Center, 10 Nov. 1999. `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf`.

[2] M. Alt, Ferdinand C., F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In R. Cousot and D.A. Schmidt, editors, *Proc. 3rd Int. Symp. SAS'96*, Aachen, 24–26 Sep. 1996, LNCS 1145, pages 52–66. Springer, 1996.

[3] American National Standards Institute, Inc. IEEE 754: Standard for binary floating-point arithmetic. Technical Report 754-1985, 754-2008, ANSI/IEEE, 1985, revised 2008. `http://grouper.ieee.org/groups/754/`.

[4] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[5] B. Blanchet. Security protocols: From linear to classical logic by abstract interpretation. *Inf. Process. Lett.*, 95(5):473–479, Sep. 2005.

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

[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. ACM SIGPLAN'2003 Conf. PLDI*, pages 196–207, San Diego, 7–14 June 2003. ACM Press.

[8] Committee to review Chinook ZD 576 crash. Report from the Select Committee on Chinook ZD 576. Technical report, House of Lords, London, Feb. 2002. `http://www.publications.parliament.uk/pa/ld200102/ldselect/ldchin/25/2504.htm`.

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

[10] P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, Jan. 1997. ACM Press.

[11] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. SARA'2000*, Horseshoe Bay, LNAI 1864, pages 1–25. Springer, 26–29 Jul. 2000.

[12] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, 277(1—2):47–103, 2002.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130, Paris, 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 *4th POPL*, pages 238–252, Los Angeles, 1977. ACM Press.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, 1979. ACM Press.

[16] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. 4th Int. Symp. on PLILP'92*, Leuven, 26–28 Aug. 1992, LNCS 631, pages 269–295. Springer, 1992.

[17] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th POPL*, pages 83–94, Albuquerque, 1992. ACM Press.

[18] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, Jan. 2000. ACM Press.

[19] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th POPL*, pages 178–190, Portland, Jan. 2002. ACM Press.

[20] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theoret. Comput. Sci.*, 290(1):531–544, Jan. 2003.

[21] P. Cousot and R. Cousot. Basic concepts of abstract interpretation, invited chapter. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 359–366. Kluwer Acad. Pub., 2004.

[22] P. Cousot and R. Cousot. Grammar analysis and parsing by abstract interpretation, invited chapter. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, LNCS 4444, pages 178–203. Springer, 2006.

[23] P. Cousot and R. Cousot. Bi-inductive structural semantics. *Inform. and Comput.*, 207(2):258–283, Feb. 2009.

[24] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In M. Sagiv, editor, *Proc. 14th ESOP'2005, Edinburg*, volume 3444 of *LNCS*, pages 21–30. Springer, 2–10 Apr. 2005.

[25] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with ASTRÉE, invited paper. In M. Hinchey, He Jifeng, and J. Sanders, editors, *Proc. 1st TASE'07*, pages 3–17, Shanghai, 6–8 June 2007. IEEE Comp. Soc. Press.

[26] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *11th ASIAN06*, pages 272–300, Tokyo, 6–8 Dec. 2006, 2008. LNCS 4435, Springer.

[27] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, 1978. ACM Press.

[28] V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract interpretation of cellular signalling networks. In F. Loggozzo, D. Peled, and L.D. Zuck, editors, *Proc. 9th Int. Conf. VMCAI 2008*, pages 83–97, San Francisco, 7–9 Jan. 2008. LNCS 4905, Springer.

[29] C. Ferdinand, R. Heckmann, and R. Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. In M. Broy, I.H. Krüger, and M. Meisinger, editors, *Automotive Software – Connected Services in Mobile Networks, First Automotive Software Workshop, ASWSD 2004*, volume 4147 of *LNCS*, pages 1–14. Springer, 10–12 Jan. 2004.

[30] J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proc. 30th ESOP'2004, Barcelona*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 27 – Apr. 4, 2004.

[31] J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Proc. 6th Int. Conf. VMCAI 2005*, pages 42–58, Paris, 17–19 Jan. 2005. LNCS 3385, Springer.

[32] J. Feret. Numerical abstract domains for digital filters. In *1st Int. Work. on Numerical & Symbolic Abstract Domains, NSAD"05*, Maison Des Polytechniciens, Paris, 21 Jan. 2005.

[33] R.W. Floyd. Assigning meaning to programs. In J.T. Schwartz, editor, *Proc. Symposium in Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[34] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proc. 6th Int. Conf. AMAST'97, Sydney*, volume 1349 of *LNCS*, pages 231–245. Springer, 13–18 Dec. 1997.

[35] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, Mar. 1991.

[36] J.R. Hauser. Handling floating-point exceptions in numeric programs. *TOPLAS*, 18(2):139–174, Mar. 1996.

[37] C.A.R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, Oct. 1969.

[38] C.B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, Apr. 2003.

[39] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, documentation

and user's manual (release 3.06). Technical report, INRIA, Rocquencourt, 19 Aug. 2002. http://caml.inria.fr/ocaml/.

[40] J.L. Lions. ARIANE 5 flight 501 failure, report by the inquiry board. http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html, 19 Jul. 1999.

[41] C. Lomont. An analysis of the excel 2007 "65535" bug, version 1.21. Technical report, www.lomont.org, Nov. 2007. .

[42] L. Mauborgne. ASTRÉE: Verification of absence of run-time error. In P. Jacquart, editor, *Building the Information Society*, chapter 4, pages 385–392. Kluwer Acad. Pub., 2004.

[43] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proc. 14th ESOP'2005, Edinburg*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 4–8, 2005.

[44] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proc. 30th ESOP'2004, Barcelona*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 27 – Apr. 4, 2004.

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

[46] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. LCTES'2006*, pages 54–63. ACM Press, June 2006.

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

[48] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E.A. Emerson and K.S. Namjoshi, editors, *Proc. 7th Int. Conf. VMCAI 2006*, pages 348–363, Charleston, 8–10, Jan. 2006. LNCS 3855, Springer.

[49] D. Monniaux. The parallel implementation of the ASTRÉE static analyzer. In *Proc. 3rd APLAS'2005*, pages 86–96, Tsukuba, 3–5 Nov. 2005. LNCS 3780, Springer.

[50] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3):Article No. 12, may 2008.

[51] P. Naur. Proofs of algorithms by general snapshots. *BIT*, 6:310–316, 1966.

[52] S. Perdrix. Quantum entanglement analysis based on abstract interpretation. In M. Alpuente and G. Vidal, editors, *Proc. 15th Int. Symp. SAS'08*, Valencia, 16–18 Jul. 2008, LNCS 5079, pages 270–282. Springer, 2008.

[53] X. Rival. Abstract dependences for alarm diagnosis. In *Proc. 3rd APLAS'2005*, pages 347–363, Tsukuba, 3–5 Nov. 2005. LNCS 3780, Springer.

[54] X. Rival. Understanding the origin of alarms in ASTRÉE. In C. Hankin and I. Siveroni, editors, *Proc. 12th Int. Symp. SAS'05*, pages 303–319, London, LNCS 3672, 7–9 Sep. 2005.

[55] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *TOPLAS*, 29(5), Aug. 2007.

[56] J. Serra. Morphological filtering: An overview. *Signal Processing*, 38:3–11, 1994.

[57] H. Wolpe, M. Blair, S. Obenski, and P. Bridickas. Gao/imtec-92-26 patriot missile software problem. Technical report, Information Management and Technology Division, Washington, D.C., 14 Feb. 1992. http://www.fas.org/spp/starwars/gao/im92026.htm.