1. Motivation

A Strong Need for Software Better Quality

– Poor software quality is not acceptable in safety and mission critical software applications.

– The present state of the art in software engineering does not offer sufficient quality guarantees.
The Complexity of Software Design

- The design of complex software is difficult and economically critical

  Boeing officials confirmed today that a fastener shortage and problems with flight control software have pushed “first flight” of the Boeing 787 Dreamliner to sometime between mid-November and mid-December (see News Releases).

  The software delays involve Honeywell Aerospace, which is responsible for flight control software. The work on this part of the 787 was simply underestimated, said Bair.”

The Security of Complex Software

- Complex software is subject to security vulnerabilities
- Example (www.wired.com/politics/security/news/2008/01/dreamliner_security)

  “FAA: Boeing’s New 787 May Be Vulnerable to Hacker Attack
  Kim Zetter, freelance journalist in Oakland, CA, Jan. 4, 2008

  Boeing’s new 787 Dreamliner passenger jet may have a serious security vulnerability in its onboard computer networks ...

  According to the FAA document published in the Federal Register (mirrored at Cryptome.org), the vulnerability exists because the plane’s computer systems connect the passenger network with the flight-safety, control and navigation network. It also connects to the airline’s business and administrative-support network, which communicates maintenance issues to ground crews.

Tool-Based Software Design Methods

- New tool-based software design methods will have to emerge to face the unprecedented growth and complexification of critical software
- E.g. FCPC (Flight Control Primary Computer)
  - A220: 20 000 LOCs,
  - A340 (V1): 130 000 LOCs
  - A340 (V2): 250 000 LOCs
  - A380: 1.000.000 LOCs
  - A350: static analysis to be integrated in the software production

Static Analysis

A static analyzer is a program that
- takes as input:
  - a program P (written in some given programming language P with a given semantics S_P)
  - a specification S (implicit S[P] or written in some specification language S with a given semantics S_S)
- always terminates and delivers automatically as output:
  - a diagnosis on the validity of the program semantics with respect the specification semantics
Difficulties of Static Analysis

- automatic + infinite state + termination $\implies$ \textbf{undecidable}!
- for a \textbf{programming (and a specification) language}, not for a given model of a given program:
  \[ \forall P \in \mathbb{P} : \forall S \in \mathbb{S} : \mathbb{G}_P[P] \subseteq \mathbb{G}_S[P, S]? \]
  or, more simply for an \textbf{implicit specification} $S[P]$:
  \[ \forall P \in \mathbb{P} : \mathbb{G}_P[P] \subseteq S[P]? \]

Soundness and Completeness

- \textbf{Soundness}: for all $P \in \mathbb{P}$, if the answer is \textit{yes} (no) then $\mathbb{G}_P[P] \subseteq S[P]$ (resp. $\mathbb{G}_P[P] \not\subseteq S[P]$)
- \textbf{Completeness}: for all $P \in \mathbb{P}$, if $\mathbb{G}_P[P] \subseteq S[P]$ ($\mathbb{G}_P[P] \not\subseteq S[P]$) then the answer is \textit{yes} (resp. no)

\textbf{We always require \textbf{SOUNDNESS}!}

Undecidability $\implies$ \textbf{NO completeness}

Problems with Formal Methods

- \textbf{Formal specifications} (abstract machines, temporal logic, \ldots) are costly, complex, error-prone, difficult to maintain, not mastered by casual programmers
- \textbf{Formal semantics} of the specification and programming language are inexistent, informal, unrealistic or complex
- \textbf{Formal proofs} are partial (static analysis), do not scale up (model checking) or need human assistance (theorem proving & proof assistants)
  \Rightarrow \textbf{High costs} (for specification, proof assistance, etc).

Avantages of Static Analysis

- \textbf{Formal specifications} are implicit (no need for explicit, user-provided specifications)
- \textbf{Formal semantics} are approximated by the static analyzer (no user-provided models of the program)
- \textbf{Formal proofs} are automatic (no required user-interaction)
- \textbf{Costs} are low (no modification of the software production methodology)
- \textbf{Scales up} to 100.000 to 1.000.000 LOC$S$
- \textbf{Rapid and large diffusion} in embedded software production industries
Disadvantages of Static Analysis

- **Imprecision** (acceptable in some applications like WCET or program optimization)
- **Incomplete** for program verification
- **False alarms** are due to unsuccessful automatic proofs in 5 to 15% of the cases

For example, 1% of 500,000 potential (true or false) alarms is 5,000, too much to be handled by hand!

---

Remedies to False Alarms in ASTRÉE

- ASTRÉE is specialized to **specific program properties**
- ASTRÉE is specialized to **real-time synchronous control/command programs written in C**
- ASTRÉE offers possibilities of **refinement**

The cost of adapting ASTRÉE to a specific program, should be a small fraction of the cost to test the specific program properties verified by ASTRÉE.

---

Abstract Interpretation

There are two **fundamental concepts** in computer science (and in sciences in general):

- **Abstraction** : to reason on complex systems
- **Approximation** : to make effective undecidable computations

These concepts are formalized by **abstract interpretation**

[CC77, Cou78, CC79, Cou81, CC92a]

---

References


Applications of Abstract Interpretation

- Static Program Analysis [CC77], [CH78], [CC79] including Dataflow Analysis; [CC79], [CC00], Set-based Analysis [CC95], Predicate Abstraction [Cou03], ...
- Grammar Analysis and Parsing [CC03];
- Hierarchies of Semantics and Proof Methods [CC92b], [Cou02];
- Typing & Type Inference [Cou97];
- (Abstract) Model Checking [CC00];
- Program Transformation (including program optimization, partial evaluation, etc) [CC02];

Applications of Abstract Interpretation (Cont’d)

- Software Watermarking [CC04];
- Bisimulations [RT04, RT06];
- Language-based security [GM04];
- Semantics-based obfuscated malware detection [PCJD07].
- Databases [AGM93, BPC01, BS97]
- Computational biology [Dan07]
- Quantum computing [JP06, Per06]

All these techniques involve sound approximations that can be formalized by abstract interpretation

**Principle of Abstraction**

Operational semantics
Safety property

Test/Debugging is Unsafe

Bounded Model Checking is Unsafe

Abstraction
Over-Approximation

Abstract Interpretation is Sound

Soundness and Incompleteness

4 This situation is always excluded in static analysis by abstract interpretation.
Soundness Requirement: Erroneous Abstraction

\[ x(t) \]

Forbidden zone

\[ \text{Error} !!! \]

Possible trajectories

Erroneous trajectory abstraction

---

This situation is always excluded in static analysis by abstract interpretation.

Seminar, Colloquia Patavina, Padova, 19/2/2008

P. Cimrman

Soundness Requirement: Erroneous Abstraction

\[ x(t) \]

Forbidden zone

\[ \text{Error} !!! \]

Possible trajectories

Erroneous trajectory abstraction

---

This situation is always excluded in static analysis by abstract interpretation.

Seminar, Colloquia Patavina, Padova, 19/2/2008

P. Cimrman

Soundness Requirement: Erroneous Abstraction

\[ x(t) \]

Forbidden zone

\[ \text{False alarm} \]

Possible trajectories

Imprecise trajectory abstraction

---

Imprecision \(\Rightarrow\) False Alarms

Seminar, Colloquia Patavina, Padova, 19/2/2008

P. Cimrman
Refinement is necessary to distinguish from true alarms

Design by Refinement

Global Interval Abstraction $\rightarrow$ False Alarms

Local Interval Abstraction $\rightarrow$ False Alarms
Refinement by Partitionning

State-based versus Trace-based Partitioning

State-based partitionning at control points:

Trace-based partitionning at control points:

Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).

Intervals with Partitionning

Trace Partitioning

Principle:
- Semantic equivalence:

\[
\text{if } (B) \{ \text{C1} \} \text{ else } \{ \text{C2} \}; \text{ C3}
\]

\[
\downarrow
\]

\[
\text{if } (B) \{ \text{C1}; \text{ C3} \} \text{ else } \{ \text{C2}; \text{ C3} \};
\]

- More precise in the abstract: concrete execution paths are merged later.

Application:

```c
if (B)
{ X=0; Y=1; }
else
{ X=1; Y=0; }
R = 1 / (X-Y);
```

cannot result in a division by zero.
Case analysis with loop unrolling

```c
#include <stdio.h>

int main() {
    float x[] = {-10.0, -10.0, 0.0, 10.0, 10.0};
    float y[] = {0.0, 2.0, 2.0, 0.0};
    float z[] = {-20.0, -20.0, 0.0, 20.0};
    float w, x, y, z;
    int i = 0;
    __ASTREE_known_fact({((-30.0 <= x) && (x <= 30.0))});
    while (i < 3) {  // i >= 0
        if (x >= y[i] - 100) {
            w = x - t[i] * c[i] + d[i];
            __ASTREE_log_vars(1);
        }
        i = i + 1;
    }
    return 0;
}
```

Examples of abstractions:

- Set of points \( \{(x_i, y_i) : i \in \Delta \} \)
- Signs \( x \geq 0, y \geq 0 \) [CC79]
Examples of abstractions

Intervals $a \leq x \leq b$, $c \leq y \leq d$  [CC77]

Examples of abstractions

Octagons $x - y \leq a$, $x + y \leq b$ [Min06b]

Examples of abstractions

Ellipsoids $(x - a)^2 + (y - b)^2 \leq c$  [?]

Examples of abstractions

Exponentials $a^x \leq y$  [Fer05]
3. The ASTRÉE static analyzer

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

Programs Analyzed by ASTRÉE and their Semantics

Project Members

- Bruno Blanchet
- Patrick Cousot
- Radhia Cousot
- Jérôme Feret
- Laurent Mauborgne
- Antoine Miné
- David Monniaux
- Xavier Rival

Programs analysed by ASTRÉE

- Application Domain: large safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.
- C programs:
  - basic numeric datatypes, structures and arrays
  - pointers (including on functions),
  - floating point computations
  - tests, loops and function calls
  - limited branching (forward goto, break, continue)
The Class of Considered Periodic Synchronous Programs

\begin{verbatim}
declare volatile input, state and output variables;
initialize state and output variables;
\textbf{loop forever}
  - read volatile input variables,
  - compute output and state variables,
  - write to output variables;
  __ASTREE_wait_for_clock ();
\end{verbatim}

Task scheduling is static:
- Requirements: the only interrupts are clock ticks;
- Execution time of loop body less than a clock tick, as verified by the aiT WCET Analyzers [FHL+01].

Implicit Specification: Absence of Runtime Errors

- No violation of the norm of C (e.g. array index out of bounds, division by zero)
- No implementation-specific undefined behaviors (e.g. maximum short integer is 32767, NaN)
- No violation of the programming guidelines (e.g. static variables cannot be assumed to be initialized to 0)
- No violation of the programmer assertions (must all be statically verified).

Such limitations are quite common for embedded safety-critical software.

Specification Proved by ASTRÉE
Different Classes of Run-time Errors

1. **Errors terminating the execution**. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.

2. **Errors not terminating the execution with predictable outcome**. ASTRÉE warns and continues with worst-case assumptions.

3. **Errors not terminating the execution with unpredictable outcome**. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.

⇒ ASTRÉE is sound with respect to C standard, unsound with respect to C implementation, unless no false alarm.

---

Modular Arithmetic

Modular arithmetics is not very intuitive

In C:

```c
#include <stdio.h>
int main () {
    int x,y;
    x = -2147483647 / -1;
    y = ((-x) -1) / -1;
    printf("x = %i, y = %i\n",x,y);
    return 0;
}
```

```bash
% gcc modulo.c.c
% ./a.out
x = 2147483647, y =
```

---

Floating-point exceptions e.g. (invalid operations, overflows, etc.) when traps are activated

- e.g. overflows over signed integers resulting in some signed integer
- e.g. memory corruptions

- modular arithmetic is not very intuitive

In C:

```c
#include <stdio.h>
int main () {
    int x,y;
    x = -2147483647 / -1;
    y = ((-x) -1) / -1;
    printf("x = %i, y = %i\n",x,y);
    return 0;
}
```

```bash
% gcc modulo.c.c
% ./a.out
x = 2147483647, y = -2147483648
```
Static Analysis with ASTRÉE

```c
% cat -n modulo.c
1 int main () {
2 int x,y;
3 x = -2147483647 / -1;
4 y = ((-x) -1) / -1;
5 __ASTREE_log_vars((x,y));
6 }
7
% astree -exec-fn main -unroll 0 modulo.c
|& egrep -A 1 "(<integers>)(WARN)"
modulo.c:4.4-18:[:call#main0:] WARN: signed int arithmetic range
{2147483648} not included in [-2147483648, 2147483647]
<integers (intv+cong+bitfield+set): y in [-2147483648, 2147483647] \ Top
x in {2147483647} \ {2147483647} >

**ASTRÉE signals the overflow and goes on with an unknown value.**
```

Float Arithmetics does Overflow

**In C:**

```c
% cat -n overflow.c
1 void main () {
2 double x,y;
3 x = 1.0e+256 * 1.0e+256;
4 y = 1.0e+256 * -1.0e+256;
5 __ASTREE_log_vars((x,y));
6 }
7
% gcc overflow.c
% ./a.out
x = inf, y = -inf
```

The Ariane 5.01 maiden flight

– June 4th, 1996 was the maiden flight of Ariane 5
A 16 bit piece of code of Ariane 4 had been reused within the new 32 bit code for Ariane 5. This caused an uncaught overflow, making the launcher uncontrolable.
Example of rounding error

```c
/* float-error.c */
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  0.000000
```

Example of rounding error

```c
/* double-error.c */
int main () {  double x; float y, z, r;
   x = ldexp(1.,50)+ldexp(1.,26); */
   y = x + 1.0e21;
z = x - 1.0e21;
r = y - z;
   printf("%f\n", r);
}  gcc double-error.c
  %./a.out  134217728.00000
```

Explanation of the huge rounding error

1. Floats
2. Reals
3. Doubles
   - Rounding
   - x - 10^21
   - x + 10^21

Static analysis with ASTREE

```c
% cat -n double-error.c
2  int main () {
3   double x; float y, z, r;
4   /* x = ldexp(1.,50)+ldexp(1.,26); */
5   x = 1125899973951488.0;
6   y = x + 1.0e21;
7   z = x - 1.0e21;
8   r = y - z;
9   printf("%f\n", r);
10  }
}  gcc double-error.c
  %./a.out  134217728.00000
% astree -exec-fn main -print-float-digits 10 double-error.c |& grep "r in direct = <float-interval: r in [-134217728, 134217728] >"
```

**ASTREE** makes a worst-case assumption on the rounding ($+\infty$, $-\infty$, 0, nearest) hence the possibility to get $-134217728$. 

Example of accumulation of small rounding errors

```
# cat -n rounding.c.c
1 #include <stdio.h>
2 int main () {
3   int i; double x; x = 0.0;
4   for (i=1; i<1000000000; i++) {
5     x = x + 1.0/10.0;
6  }
7  printf("x = %f\n", x);
8 }
```

```
% gcc rounding.c.c
% ./a.out
x = 99999998.745418
```

since $(0.1)_10 = (0.0001100110011001100\ldots)_2$

Static analysis with ASTRÉE

```
% cat -n rounding.c
1 int main () {
2   double x; x = 0.0;
3   while (1) {
4     x = x + 1.0/10.0;
5     __ASTREE_log_vars(x);
6     __ASTREE_wait_for_clock();
7   }
8 }
```

```
% cat rounding.config
% __ASTREE_max_clock((1000000000));
% __astree -exec-fn main -config-sem rounding.config -unroll 0 rounding.c\n% | & egrep "(x in|\(\{x\}\))[W]" | tail -2
% direct = <float-interval: x in [0.1, 0.000000040.938] >
% |x| <= 1. *(0. + 0.1/(1.1-1)*)*(1.)"clock - 0.1/(1.1-1) + 0.1
% <= 2000000040.938
```

The Patriot missile failure

- “On February 25th, 1991, a Patriot missile ... failed to track and intercept an incoming Scud (°).”
- The software failure was due to accumulated rounding error (°)

(°) This Scud subsequently hit an Army barracks, killing 28 Americans.

(°) “Time is kept continuously by the system’s internal clock in tenths of seconds”

- “The system had been in operation for over 100 consecutive hours”
- “Because the time calculation caused the range gate to shift so much that the system could not track the incoming Scud”
Static Analysis of Scaling with ASTRÉE

```c
% cat -n scale.c
1 int main () {
2 float x; x = 0.70000001; x = 0.69999988079071
3 while (1) {
4 x = x / 3.0;
5 x = x * 3.0;
6 __ASTREE_log_vars(x);
7 __ASTREE_wait_for_clock();
8 }
9 }
%
% cat scale.config
__ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem scale.config -unroll 0 scale.c \
|& grep "x in" | tail -1
direct = <float-interval: x in [0.6999998888, 0.700000047684] > 
```

Ellipsoid Abstract Domain for Filters

- Computes $X_n = \{ \alpha X_{n-1} + \beta X_{n-2} + Y_n \}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.

Filter Example [Fer04]

```c
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 = (((X * S[0] + (E[0] * 0.7)) + (E[1] * 0.4)) 
                 + (S[1] * 1.5)) - (S[1] * 0.7)); } 
    E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P; 
    /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE; 
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE; }
}
```
Time Dependence

Arithmetic-Geometric Progressions (Example 1)

% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I, int R, BOOLEAN T;
void main()
{
    R = 0;
    while (TRUE) {
        __ASTREE_log_vars((R)); ← potential overflow!
        if (I) { R = R + 1; }
        else { R = 0; }
        T = (R >= 100);
        __ASTREE_wait_for_clock();
    }
% cat count.config
__ASTREE_volatile_input((I [0,1]));
__ASTREE_max_clock((3600000));
% astree -exec-fn main -config-sem count.config count.c|grep 'R'
[R] <= 0. + clock *1. <= 3600001.

Arithmetic-Geometric Progressions: Example 2

% cat retro.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
void main()
{
    FIRST = TRUE;
    while (TRUE) {
        dev();
        FIRST = FALSE;
        __ASTREE_wait_for_clock();
    }
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));
    |P| <= (15. + 5.87747175411e-39
        / 1.19209290217e-07) * (1
            + 1.19209290217e-07) "clock
            - 5.87747175411e-39 /
        1.19209290217e-07 <= 23.0393526881

% cat retro.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
void main()
{
    FIRST = TRUE;
    while (TRUE) {
        dev();
        FIRST = FALSE;
        __ASTREE_wait_for_clock();
    }
}
Overapproximation with an Arithmetic-Geometric Progression

Arithmetic-geometric progressions\(^{13}\) [Per05]

- Abstract domain: \((\mathbb{R}^+)^5\)
- Concretization:
  \[
  \gamma \in (\mathbb{R^+})^5 \leftrightarrow \varphi(\mathbb{N} \mapsto \mathbb{R})
  \]
  \[
  \gamma(M, a, b, a', b') = \\
  \{ f \mid \forall k \in \mathbb{N} : |f(k)| \leq (\lambda x \cdot ax + b \circ (\lambda x \cdot a'x + b')^k)(M) \}
  \]

i.e. any function bounded by the arithmetic-geometric progression.

\(^{13}\) here in \(\mathbb{R}\)

---

Reference:


---

4. The industrial use of ASTRÉE

---

References:

Digital Fly-by-Wire Avionics

Example application

- Primary flight control software of the Airbus A340 family/A380 fly-by-wire system
  - C program, automatically generated from a proprietary high-level specification (à la Simulink/SCADE)
  - A340 family: 132,000 lines, 75,000 LOCs after preprocessing, 10,000 global variables, over 21,000 after expansion of small arrays, now $\times 2$

A380: $\times 3/7$

Benchmarks (Airbus A340 Primary Flight Control Software)

- V1, 132,000 lines, 75,000 LOCs after preprocessing
- Comparative results (commercial software):
  4,200 (false?) alarms, 3.5 days;
- Our results:
  0 alarms,
  40mn on 2.8 GHz PC, 300 Megabytes
  $\rightarrow$ A world première in Nov. 2003!

14 The electrical flight control system is placed between the pilot’s controls (sidesticks, rudder pedals) and the
control surfaces of the aircraft, whose movement they control and monitor.

15 “Flight Control and Guidance Unit” (FCGU) running on the “Flight Control Primary Computers” (FCPC).
The three primary computers (PCPC) and two secondary computers (PCSC) which form the A340 and
A380 electrical flight control system are placed between the pilot’s controls (sidesticks, rudder pedals) and
the control surfaces of the aircraft, whose movement they control and monitor.
The main loop invariant for the A340 V1
A textual file over 4.5 Mb with
- 6,900 boolean interval assertions \(x \in [0; 1]\)
- 9,600 interval assertions \(x \in [a; b]\)
- 25,400 clock assertions \((x + \text{clk} \in [a; b] \land x - \text{clk} \in [a; b])\)
- 19,100 additive octagonal assertions \((a \leq x + y \leq b)\)
- 19,200 subtractive octagonal assertions \((a \leq x - y \leq b)\)
- 100 decision trees
- 60 ellipse invariants, etc ...
involved over 16,000 floating point constants (only 550 appearing in the program text) \(\times 75,000\) LOCs.

Possible origins of imprecision and how to fix it
In case of false alarm, the imprecision can come from:
- Abstract transformers (not best possible) \(\rightarrow\) improve algorithm;
- Automatized parametrization (e.g. variable packing) \(\rightarrow\) improve pattern-matched program schemata;
- Iteration strategy for fixpoints \(\rightarrow\) fix widening \footnote{This can be very hard since at the limit only a precise infinite iteration might be able to compute the proper abstract invariant. In that case, it might be better to design a more refined abstract domain.};
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract \(\rightarrow\) add a new abstract domain to the reduced product (e.g. filters).

(Airbus A380 Primary Flight Control Software)

- 0 alarms (Nov. 2004), after some additional parametrization and simple abstract domains developments
- Now at 1,000,000 lines!
  34h,
  8 Gigabyte
  \(\rightarrow\) A world grand première!

5. Conclusion
Characteristics of the **Astrée** Analyzer

**Sound:** - **Astrée** is a *bug eradicator*: finds all bugs in a well-defined class (runtime errors)
- **Astrée** is *not* a *bug hunter*: finding some bugs in a well-defined class (e.g. by *bug pattern detection* like FindBugs™, PREfast or PMD)
- **Astrée** is *exhaustive*: covers the whole state space (≠ MAGIC, CBMC)
- **Astrée** is *comprehensive*: never omits potential errors (≠ UNO, CMC from coverity.com) or sort most probable ones to avoid overwhelming messages (≠ Splint)

**Characteristics of the **Astrée** Analyzer (Cont’d)**

**Multiabstraction:** uses many numerical/symbolic abstract domains (≠ symbolic constraints in Bane or the canonical abstraction of TVLA)

**Infinitary:** all abstractions use infinite abstract domains with widening/narrowing (≠ model checking based analyzers such as Bandera, Bogor, Java PathFinder, Spin, VeriSoft)

**Efficient:** always terminate (≠ counterexample-driven automatic abstraction refinement BLAST, SLAM)

**Characteristics of the **Astrée** Analyzer (Cont’d)**

**Static:** compile time analysis (≠ run time analysis Rational Purify, Parasoft Insure++)

**Program Analyzer:** analyzes programs not micromodels of programs (≠ PROMELA in SPIN or Alloy in the Alloy Analyzer)

**Automatic:** no end-user intervention needed (≠ ESC Java, ESC Java 2), or PREfast (annotate functions with intended use)

**Characteristics of the **Astrée** Analyzer (Cont’d)**

**Extensible/Specializable:** can easily incorporate new abstractions (and reduction with already existing abstract domains) (≠ general-purpose analyzers PolySpace Verifier)

**Domain-Aware:** knows about control/command (e.g. digital filters) (as opposed to specialization to a mere programming style in C Global Surveyor)

**Parametric:** the precision/cost can be tailored to user needs by options and directives in the code
Characteristics of the ASTRÉE Analyzer (Cont’d)

**Automatic Parametrization**: the generation of parametric directives in the code can be programmed (to be specialized for a specific application domain)

**Modular**: an analyzer instance is built by selection of OCAML modules from a collection each implementing an abstract domain

**Precise**: very few or no false alarm when adapted to an application domain — it is a VERIFIER!

---

The Future of the ASTRÉE Analyzer

- ASTRÉE has shown **usable and useful** in one industrial context (*electric flight control*):
  - as a R & D tool for A340 V2 and A380,
  - as a production tool for the A350;
- **More applications** are forthcoming (ES_PASS project);
- **Industrialization** is simultaneously under consideration.

---

THE END

THE END, THANK YOU
6. Bibliography


Seminar, Colloquia Patavina, Padova, 19/2/2008  


Seminar, Colloquia Patavina, Padova, 19/2/2008  


