Motivation

All Computer Scientists Have Experienced Bugs

- Ariane 5.01 failure (overflow)
- Patriot failure (float rounding)
- Mars orbiter loss (unit error)

It is preferable to verify that mission/safety-critical programs do not go wrong before running them.
Static Analysis by Abstract Interpretation

**Static analysis:** analyze the program at compile-time to verify a program runtime property

Undecidability →

**Abstract interpretation:** effectively compute an abstraction/sound approximation of the program semantics,
- which is precise enough to imply the desired property, and
- coarse enough to be efficiently computable.

Reference


States

Values of given type:
\[ \mathcal{V}[T] : \text{values of type } T \in T \]
\[ \mathcal{V}[\text{int}] \equiv \{ z \in \mathbb{Z} \mid \text{min\_int} \leq z \leq \text{max\_int} \} \]

Program states \( \Sigma[P] \):\(^1\)
\[ \Sigma[D C] \equiv \Sigma[D] \]
\[ \Sigma[T X ;)] \equiv \{ X \} \mapsto \mathcal{V}[T] \]
\[ \Sigma[T X ; D] \equiv (\{ X \mapsto \mathcal{V}[T] \}) \cup \Sigma[D] \]

Concrete Semantic Domain of Programs

Concrete semantic domain for reachability properties:
\[ \mathcal{D}[P] \equiv \varphi(\Sigma[P]) \]
sets of states

i.e. program properties where \( \sqsubseteq \) is implication, \( \emptyset \) is false, \( \sqcup \) is disjunction.

---

Concrete Reachability Semantics of Programs

\[ S[X = E;] R \equiv \{ \rho[X \leftarrow \mathcal{E}[E]\rho] \mid \rho \in R \cap \text{dom}(E) \} \]
\[ \rho[X \leftarrow v](X) \equiv v, \quad \rho[X \leftarrow v](Y) \equiv \rho(Y) \]
\[ S[\text{if } B C' R] \equiv S[C'](B[B]R) \cup B[\neg B]R \]
\[ B[B]R \equiv \{ \rho \in R \cap \text{dom}(B) \mid B \text{ holds in } \rho \} \]
\[ S[\text{if } B C' \text{ else } C'' R] \equiv S[C'](B[B]R) \cup S[C'']((B[B]R) \neg B) \]
\[ S[\text{while } B C' R] \equiv \text{let } \mathcal{W} = \sqcap_{\rho \in \text{dom}(B)} \lambda X. R \cup S[C'](B[B]X) \text{ in } (B[\neg B]W) \]
\[ S[\{\} R] \equiv R \]
\[ S[C_1 \ldots C_n] R \equiv S[C_1] \circ \cdots \circ S[C_n] R \quad n > 0 \]
\[ S[D C] R \equiv S[C](\Sigma[D]) \quad \text{(uninitialized variables)} \]

Not computable (undecidability).

Abstract Semantic Domain of Programs

\[ \langle \mathcal{D}[P], \sqsubseteq, \bot, \sqcup \rangle \]

such that:
\[ \langle \mathcal{D}[P], \sqsubseteq \rangle \xrightarrow{\gamma} \langle \mathcal{D}[P], \sqsubseteq \rangle \]
i.e.
\[ \forall X \in \mathcal{D}[P], Y \in \mathcal{D}[P] : \alpha(X) \sqsubseteq Y \iff X \sqsubseteq \gamma(Y) \]
hence \( \langle \mathcal{D}[P], \sqsubseteq, \bot, \sqcup \rangle \) is a complete lattice such that \( \bot = \alpha(\emptyset) \) and \( \sqcup X = \alpha(\sqcup \alpha(X)) \)

---

\(^1\) States \( \rho \in \Sigma[P] \) of a program \( P \) map program variables \( X \) to their values \( \rho(X) \)
Example 1 of Abstraction

Set of traces: set of finite or infinite maximal sequences of states for the operational transition semantics

Strongest liberal postcondition: final states $s$ reachable from a given precondition $P$

$$\alpha(X) = \lambda P \cdot \{ s | \exists \sigma_0 \sigma_1 \ldots \sigma_n \in X : \sigma_0 \in P \land s = \sigma_n \}$$

We have ($\Sigma$: set of states, $\subseteq$ pointwise):

$$\langle \rho(\Sigma^\infty), \subseteq \rangle \xleftarrow{\gamma} \langle \rho(\Sigma) \cup \rho(\Sigma), \subseteq \rangle$$

Example 2 of Abstraction

Set of traces: set of finite or infinite maximal sequences of states for the operational transition semantics

Trace of sets of states: sequence of set of states appearing at a given time along at least one of these traces

$$\alpha_0(X) = \lambda i \cdot \{ \sigma_i | \sigma \in X \land 0 \leq i < |\sigma| \}$$

Set of reachable states: set of states appearing at least once along one of these traces (global invariant)

$$\alpha_1(\Sigma) = \bigcup \{ \Sigma_i | 0 \leq i < |\Sigma| \}$$

Partitionned set of reachable states: project along each control point (local invariant)

$$\alpha_2(\{ \langle c_i, \rho_i \rangle | i \in \Delta \}) = \lambda c \cdot \{ \rho_i | i \in \Delta \land c = c_i \}$$

Partitionned cartesian set of reachable states: project along each program variable (relationships between variables are now lost)

$$\alpha_3(\lambda c \cdot \{ \rho_i | i \in \Delta_c \}) = \lambda c \cdot \lambda x \cdot \{ \rho_i(x) | i \in \Delta_c \}$$

Partitionned cartesian interval of reachable states: take min and max of the values of the variables

$$\alpha_4(\lambda c \cdot \lambda x \cdot \{ v_i | i \in \Delta_{c,x} \}) = \lambda c \cdot \lambda x \cdot \{ \min \{ v_i | i \in \Delta_{c,x} \}, \max \{ v_i | i \in \Delta_{c,x} \} \}$$

$$\alpha_0, \alpha_1, \alpha_2, \alpha_3 \text{ and } \alpha_4, \text{ whence } \alpha_4 \circ \alpha_3 \circ \alpha_2 \circ \alpha_1 \circ \alpha_0 \text{ are lower-adjoints of Galois connections}$$

Example 3: Reduced Product of Abstract Domains

To combine abstractions

$$\langle D, \subseteq \rangle \xleftarrow{\gamma_1} \langle D_1^\parallel, \subseteq_1 \rangle \text{ and } \langle D, \subseteq \rangle \xleftarrow{\gamma_2} \langle D_2^\parallel, \subseteq_2 \rangle$$

the reduced product is

$$\alpha(X) \overset{\text{def}}{=} \bigcap \{ \langle x, y \rangle | X \subseteq \gamma_1(x) \land X \subseteq \gamma_2(y) \}$$

such that $\subseteq \overset{\text{def}}{=} \subseteq_1 \times \subseteq_2$ and

$$\langle D, \subseteq \rangle \xleftarrow{\gamma_1 \times \gamma_2} \langle \alpha(D), \subseteq \rangle$$

Example: $x \in [1, 9] \land x \bmod 2 = 0$ reduces to $x \in [2, 8] \land x \bmod 2 = 0$

---

\[ \text{assuming these values to be totally ordered.} \]
Abstract Reachability Semantics of Programs

\[ S^\parallel[X = E; R] \overset{\text{def}}{=} \alpha(\{\rho | X \leftarrow E[R] \mid \rho \in \gamma(R) \cap \text{dom}(E)\}) \]

\[ S^\parallel[\text{if } B C' \text{ else } C''] R \overset{\text{def}}{=} S^\parallel[C'](B^\parallel[B]R) \cup S^\parallel[C''](B^\parallel[-B]R) \]

\[ S^\parallel[\text{while } B C'] R \overset{\text{def}}{=} \text{let } \mathcal{W} = \text{lfp } ^\parallel \lambda \chi. R \cup S^\parallel[C'](B^\parallel[B]\chi) \]

\[ \text{in } (B^\parallel[-B]\mathcal{W}) \]

\[ S^\parallel[\text{if } B C' \text{ else } C''] R \overset{\text{def}}{=} S^\parallel[C'](B^\parallel[B]R) \cup S^\parallel[C''](B^\parallel[-B]R) \]

\[ S^\parallel[\text{while } B C'] R \overset{\text{def}}{=} \text{let } \mathcal{F}^\parallel = \lambda \chi. \text{let } \mathcal{Y} = R \cup S^\parallel[C'](B^\parallel[B]\chi) \]

\[ \text{in } \mathcal{F}^\parallel \mathcal{Y} \text{ and } \mathcal{W} = \text{lfp } ^\parallel \mathcal{F}^\parallel \text{ in } (B^\parallel[-B]\mathcal{W}) \]

\[ S^\parallel[\text{while } B C'] R \overset{\text{def}}{=} \text{let } \mathcal{W} = \text{lfp } ^\parallel \lambda \chi. R \cup S^\parallel[C'](B^\parallel[B]\chi) \]

\[ \text{in } \mathcal{W} \]

\[ S^\parallel[\text{if } B C' \text{ else } C''] R \overset{\text{def}}{=} S^\parallel[C'](B^\parallel[B]R) \cup S^\parallel[C''](B^\parallel[-B]R) \]

\[ S^\parallel[\text{while } B C'] R \overset{\text{def}}{=} \text{let } \mathcal{F}^\parallel = \lambda \chi. \text{let } \mathcal{Y} = R \cup S^\parallel[C'](B^\parallel[B]\chi) \]

\[ \text{in } \mathcal{F}^\parallel \mathcal{Y} \text{ and } \mathcal{W} = \text{lfp } ^\parallel \mathcal{F}^\parallel \text{ in } (B^\parallel[-B]\mathcal{W}) \]

3 Note: \( \mathcal{F}^\parallel \) not monotonic!
A few applications of Abstract Interpretation (Cont’d)

- (Abstract) Model Checking [POPL ’00]
- Program Transformation [POPL ’02]
- Software Watermarking [POPL ’04]
- Bisimulations [RT-ESOP ’04]
- ...

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

A Practical Application of Abstract Interpretation to the ASTRÉÉ Static Analyzer

Reference
Programs analysed by ASTRÉE

Applicatıon Domain: large safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

C programs:
- with
  - basic numeric datatypes, structures and arrays
  - pointers (including on functions),
  - floating point computations
  - tests, loops and function calls
  - limited branching (forward goto, break, continue)

- without
  - union (new memory model in progress)
  - dynamic memory allocation
  - recursive function calls
  - backward branching
  - conflicting side effects
  - C libraries, system calls (parallelism)

Concrete Operational Semantics

- International norm of C (ISO/IEC 9899:1999)
- restricted by implementation-specific behaviors depending upon the machine and compiler (e.g. encoding of integers, IEEE 754-1985 norm for floats and doubles)
- 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 (e.g. volatile environment specified by a trusted configuration file, assert, execution stops on first runtime error,)

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, no float 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).

4 Thanks A. Miné

5 semantics of C unclear after an error, equivalent if no alarm
Abstraction

- Set of traces of relational state abstractions of subtraces for the concrete trace operational semantics

Example of Industrial applications

- 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
- A380: $\times 3/7$ (up to 1.000.000 LOCs)

The Class of Considered Periodic Synchronous Programs

declare volatile input, state and output variables;
initialize state and output variables;
loop forever
  - read volatile input variables,
  - compute output and state variables,
  - write to output variables;
  __ASTREE_wait_for_clock ();
end loop

Task scheduling is static:

- Requirements: the only interrupts are clock ticks;
- Execution time of loop body less than a clock tick [EMSOFT ’01].

Requirements on the Abstract Semantics

- Soundness: absolutely essential for verification
- Precision: few or no false alarm $^6$ (full certification)
- Efficiency: rapid analyses and fixes during development

---

$^6$ Potential runtime error signaled by the analyzer due to overapproximation but impossible in any actual program run compatible with the configuration file.
Challenging aspects

- **Size:** > 100 kLOC, > 10,000 variables
- **Floating point computations**
  including interconnected networks of filters, non linear control with feedback, interpolations...
- **Interdependencies among variables:**
  - Stability of computations should be established
  - Complex relations should be inferred among numerical and boolean data
  - Very long data paths from input to outputs

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 VeriSoft, Bandera, Java PathFinder)

**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)

**Sound:** covers the whole state space (≠ MAGIC, CBMC) so never omit potential errors (≠ UNO, CMC from coverity.com) or sort most probable ones (≠ Splint)

**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 module implementing an abstract domain

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

---

**Benchmarks (Airbus A340 Primary Flight Control Software)**

- 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
  → A world première!

---

**Example of Analysis Session**

---

**Benchmarks (Airbus A380 Primary Flight Control Software)**

- 350,000 lines
- 0 alarms (Nov. 2004),
  - 7h on 2.8 GHz PC,
  - 1 Gigabyte
  → A world grand première!

---

\[\text{We are still in a phase where we favour precision rather than computation costs, and this should go down. For example, the A340 analysis went up to 5 h, before being reduced by requiring less precision while still getting no false alarm.} \]
Examples of Abstractions

General-Purpose Abstract Domains: Intervals and Octagons

Intervals:
\[
1 \leq x \leq 9 \\
1 \leq y \leq 20
\]

Octagons [10]:
\[
1 \leq x \leq 9 \\
x + y \leq 77 \\
1 \leq y \leq 20 \\
x - y \leq 04
\]

Difficulties: many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program and analyzer) [POPL '77, 10, 11]

Floating-Point Computations

```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
```

```c
/* double-error.c */
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
```

Explanation of the huge rounding error

\[(x + a) - (x - a) \neq 2a\]
Floating-point linearization [11, 12]

- Approximate arbitrary expressions in the form
  \[ [a_0, b_0] + \sum_k ([a_k, b_k] \times V_k) \]

- Example:
  \[ Z = X - (0.25 \times X) \]
  is linearized as
  \[ Z = ([0.749 \cdots, 0.750 \cdots] \times X) + (2.35 \cdots 10^{-38} \times [-1,1]) \]

- Allows simplification even in the interval domain
  if \( X \in [-1,1] \), we get \( |Z| \leq 0.750 \cdots \) instead of \( |Z| \leq 1.25 \cdots \)

- Allows using a relational abstract domain (octagons)
- Example of good compromise between cost and precision

Symbolic abstract domain [11, 12]

- Interval analysis: if \( x \in [a, b] \) and \( y \in [c, d] \) then \( x - y \in [a - d, b - c] \) so if \( x \in [0, 100] \) then \( x - x \in [-100, 100] \)!!!

- The symbolic abstract domain propagates the symbolic values of variables and performs simplifications;

- Must maintain the maximal possible rounding error for float computations (overestimated with intervals);
### Boolean Relations for Boolean Control

- **Code Sample:**
  ```c
  /* boolean.c */
  typedef enum {F=0,T=1} BOOL;
  BOOL B;
  void main () {
    unsigned int X, Y;
    while (1) {
      ...
      B = (X == 0);
      ...
      if (!B) {
        Y = 1 / X;
      }
      ...
    }
  }
  ```

  The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leafs.

### Control Partitionning for Case Analysis

- **Code Sample:**
  ```c
  /* trace_partitionning.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;
    int i = 0;
    ...
    found invariant -100 ≤ x ≤ 100 ...
    while ((i < 3) && (x ≥ t[i+1])) {
      i = i + 1;
    }
    r = (x - t[i]) * c[i] + d[i];
  }
  ```

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

### Ellipsoid Abstract Domain for Filters

- **2nd Order Digital Filter:**
  ```
  \[ X_n = \alpha X_{n-1} + \beta X_{n-2} + Y_n \]
  ```

  - Computes \( X_n \) 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 [7]**
  ```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 = (((((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;
            } /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
  }
  ```

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

---

Kestrel Technology, Palo Alto, CA
© P. & R. Cousot Nov. 7th, 2005
Arithmetic-geometric progressions\footnote{ here in $\mathbb{R}$ } \cite{8}

- Abstract domain: $(\mathbb{R}^+)^5$
- Concretization:
  \[ \gamma \in (\mathbb{R}^+)^5 \implies \varphi(\mathbb{N} \mapsto \mathbb{R}) \]
  \[ \gamma(M, a, b, a', b') = \{ f \mid \forall k \in \mathbb{N} : |f(k)| \leq (\lambda x. ax + b \circ (\lambda x.a'x + b')^k)(M) \} \]
  i.e. any function bounded by the arithmetic-geometric progression.

---

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) {
    // potential overflow!
    __ASTREE_log_vars((R));
    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} BOOL;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;
void dev( ){
  X=E;
  if (FIRST) { P = X; }
  else {
    P = (P - ((((2.0 * P) - A) - B) * 4.491048e-03));
    B = A;
    if (SWITCH) {A = P;}
    else {A = X;}
  }
} 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 <= 23.039326881

---

(Automatic) Parameterization

- All abstract domains of ASTRÉE are parameterized, e.g.
  - variable packing for octagones and decision trees,
  - partition/merge program points,
  - loop unrollings,
  - thresholds in widenings, ...;
- End-users can either parameterize by hand (analyzer options, directives in the code), or
- choose the automatic parameterization (default options, directives for pattern-matched predefined program schemata).
The main loop invariant for the A340
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 ...
involving 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 \(^9\);
- 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).

\(^9\) 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.
- Exhaustive (contrary to current simulations)
- The plant model discretization errors are similar to those of simulation methods (but for the use of the actual control program instead of a model!)
- In general, polyhedral abstractions are unstable or of very high complexity
- New abstractions have to be studied (e.g. ellipsoidal abstractions)

- The control-theoretic ‘static analysis’ is easier on the plant/controller model using continuous optimization methods
- The in/variant hypotheses on the controlled plant are assumed to be true in the analysis of the plant control program
- It is now sufficient to perform the analysis analysis control program under these in/variant hypotheses
- The results can then be checked on the whole system (plant simulation + control program)

**System analysis & verification, Avenue 2**

Abstractions: program $\rightarrow$ precise, system $\rightarrow$ precise

**System analysis & verification, Avenue 3**

Abstractions: program $\rightarrow$ precise, system $\rightarrow$ precise
– The translated in/variants can be checked for the plant simulator/control program (easier than in/variant discovery)
– Should scale up (since these complex in/variants are relevant to a small part of the control program only\textsuperscript{10})

\begin{center}
Conclusion
\end{center}

1. On soundness and completeness:
   - Software checking (e.g. [abstract] testing): unsound
   - Software static analysis (for a language): sound but unprecise
   - Software verification (for a well-defined family of programs): theoretically possible [SARA’00], practically feasible [PLDI’03]

\textbf{Conclusion (cont’d)}

2. On specifications for static verification:
   - Implicit: e.g. from a language semantics (e.g. RTE) → extremely easy for engineers
   - Explicit:
     - By a logic → very hard for engineers
     - By a model → easy for engineers / hard for static analysis
     - By a program automatically generated from a model → easy for engineers / easy for static analysis

\textsuperscript{10} e.g. the plant model assumes perfect sensors/actuators/computers whereas the control program must be made dependable by using redundant failing sensors/actuators/computers
References

[2] www.astree.ens.fr [4, 5, 6, 7, 8, 9, 10, 11, 12]


