Abstract Interpretation: From Theory to Tools

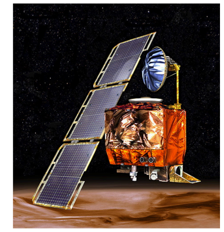
Patrick Cousot

cims.nyu.edu/~pcousot/
pcousot@cims.nyu.edu

Bugs everywhere!

Ariane 5.01 failure
(overflow error)

Patriot failure
(float rounding error)



Mars orbiter loss
(unit error)

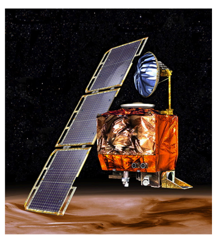
Russian Proton-M/DM-03 rocket carrying 3 Glonass-M satellites
(unknown programming error :)

Heartbleed
(buffer overrun)

Bugs everywhere!

Ariane 5.01 failure
(overflow error)

Patriot failure
(float rounding error)



Mars orbiter loss
(unit error)

Russian Proton-M/DM-03 rocket carrying 3 Glonass-M satellites
(unknown programming error :)

Heartbleed
(buffer overrun)

- These are great proofs of the presence of bugs!

On the limits of bug finding

- Giant software manufacturers can rely on gentle end-users to find myriads of bugs;
- But what about:

can passengers really help?

- Is dynamic/static bug finding always enough?
- Proving the absence of bugs is much better!

Formal Methods

Formal Methods

- Mathematical and engineering principles applied to the specification, design, construction, verification, maintenance, and evolution of very high quality software
- Strongly promoted by Harlan D. Mills since the 70's e.g.
  - Harlan D. Mills: The New Math of Computer Programming. Commun. ACM 18(1): 43-48 (1975)
  - Harlan D. Mills: Software Development. IEEE Trans. Software Eng. 2(4): 265-273 (1976)
  - Harlan D. Mills: Function Semantics for Sequential Programs. IFIP Congress 1980: 241-250
  - ...

Main formal methods for verification

- Objective: prove automatically that a program does satisfy a specification given either explicitly or implicitly (*e.g.* absence of runtime errors)
  - Deductive methods: use a theorem prover/proof assistant to check a user-provided proof argument
  - Enumerative, symbolic, bounded, solver(*e.g.* Z3)-based, interpolation, statistical, *etc* model-checking: check the specification by enumerating *finitely many* possibilities
  - Abstract interpretation: use approximation ideas to consider *infinitely many* possiblilities

Fundamental limitations

- By Gödel's undecidability, no perfect solution is and will ever be possible:
  - Deductive methods: the burden is on the end-user and the proofs are exponential in the size of programs
  - Model-checking: severe unsolved scalability problem
  - Abstract interpretation: may produce false alarms (but no false negative)
  - Unsound methods (Coverity, Klocwork, Purify, etc): no correctness guarantee at all.

The Evolution of Formal Methods

Change of Scale

- 1993: **IBM Flight Control.** A HH60 helicopter avionics component was developed on schedule in three increments comprising 33 KLOC of JOVIAL [6]. A total of 79 corrections were required during statistical certification for an error rate of 2.3 errors per KLOC for verified software with no prior execution or debugging.
- 2013: Astrée checks automatically the absence of any runtime error in the control/command software of the A380 and A400M by abstract interpretation *i.e.* > 1000 KLOC of C

Harlan D. Mills: Zero Defect Software: Cleanroom Engineering. Advances in Computers 36: 1-41 (1993)

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival: Why does Astrée scale up? Formal Methods in System Design 35(3): 229-264 (2009)

Proliferation

WCET, Axiomatic semantics, Confidentiality analysis, Program synthesis, Grammar analysis, Statistical model-checking, Invariance proof, Probabilistic verification, Parsing, Security protocole verification, Partial evaluation, Effect systems, Trace semantics, Symbolic execution, Quantum entanglement detection, Type theory, Dataflow analysis, Obfuscation, Denotational semantics, Theories combination, Code contracts, Steganography, Model checking, Systems biology analysis, Database query, Dependence analysis, CEGAR, Program transformation, Interpolants, Integrity analysis, Bisimulation, SMT solvers, Tautology testers, Abstract model checking, Operational semantics, Abstraction refinement, Type inference, Separation logic, Termination proof, Shape analysis, Malware detection, Code refactoring

The *Theory* of Abstract Interpretation: Unifies Formal Methods

The need for a unified account of formal methods

WCET
Axiomatic semantics
Security protocole verification
Systems biology analysis
Operational semantics
Confidentiality analysis
Dataflow analysis
Model checking
Database query
Abstraction refinement
Partial evaluation
Type inference
Program synthesis
Obfuscation
Dependence analysis
Effect systems
Denotational semantics
Separation logic
Grammar analysis
CEGAR
Termination proof
Theories combination
Program transformation
Trace semantics
Statistical model-checking
Code contracts
Interpolants
Abstract model checking
Shape analysis
Invariance proof
Symbolic execution
Integrity analysis
Malware detection
Probabilistic verification
Quantum entanglement detection
Bisimulation
Code refactoring
SMT solvers
Parsing
Type theory
Steganography
Tautology testers

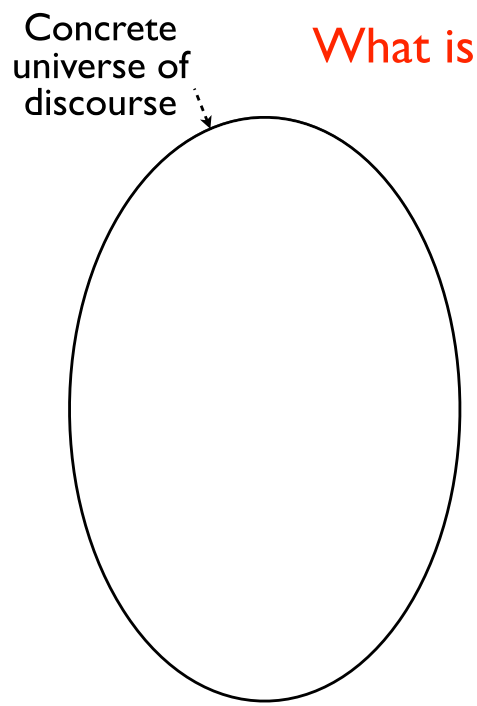
Underlying unity of formal methods

Abstract interpretation

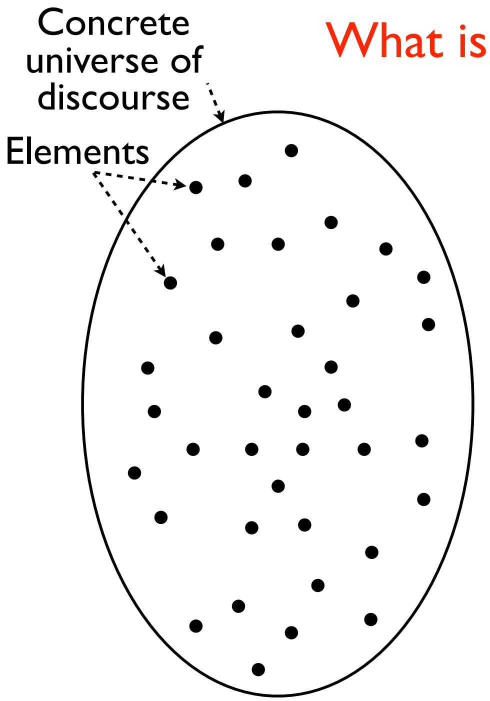
WCET
Axiomatic semantics
Security protocole verification
Systems biology analysis
Operational semantics
Confidentiality analysis
Dataflow analysis
Model checking
Database query
Abstraction refinement
Partial evaluation
Type inference
Program synthesis
Obfuscation
Dependence analysis
Effect systems
Denotational semantics
Separation logic
Grammar analysis
CEGAR
Termination proof
Theories combination
Program transformation
Trace semantics
Statistical model-checking
Code contracts
Interpolants
Abstract model checking
Shape analysis
Invariance proof
Symbolic execution
Integrity analysis
Malware detection
Probabilistic verification
Quantum entanglement detection
Bisimulation
Code refactoring
SMT solvers
Parsing
Type theory
Steganography
Tautology testers

Principle of Abstract Interpretation

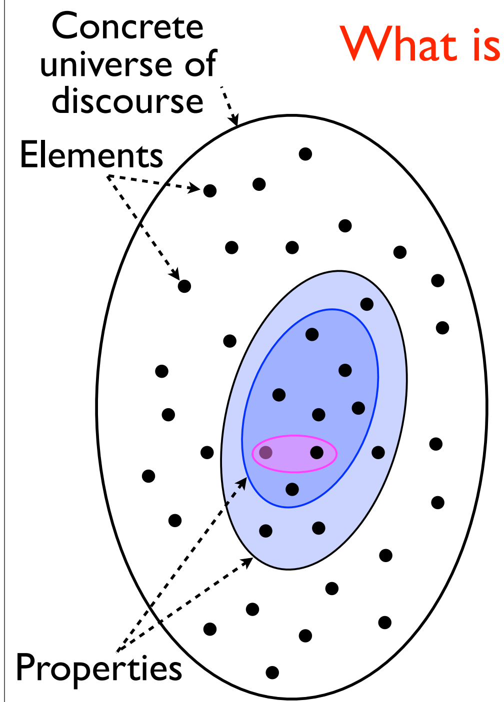
What is abstraction in AI?



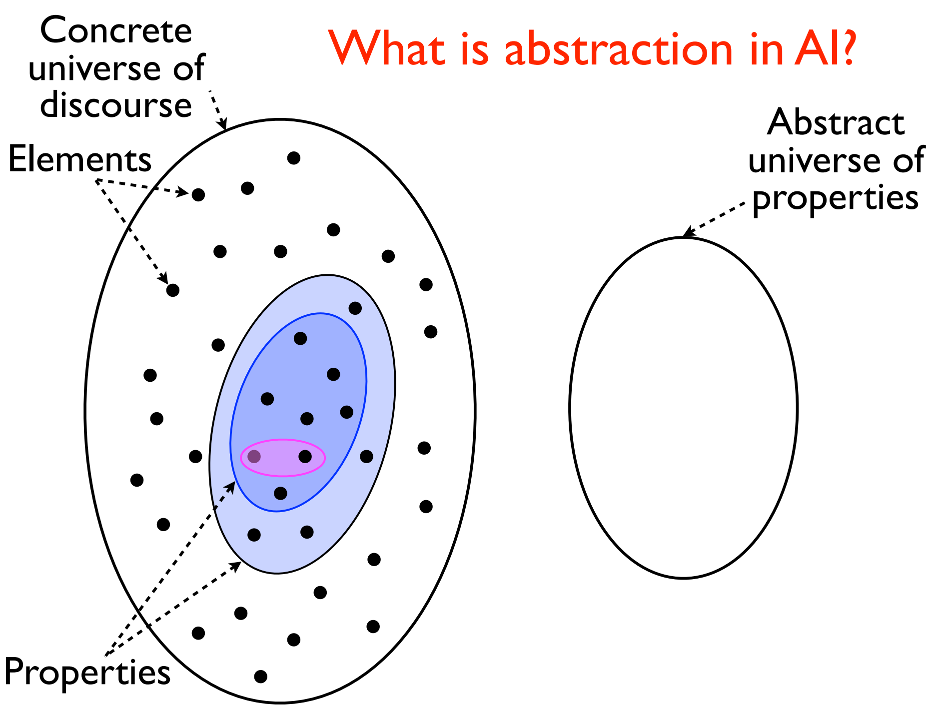
What is abstraction in AI?



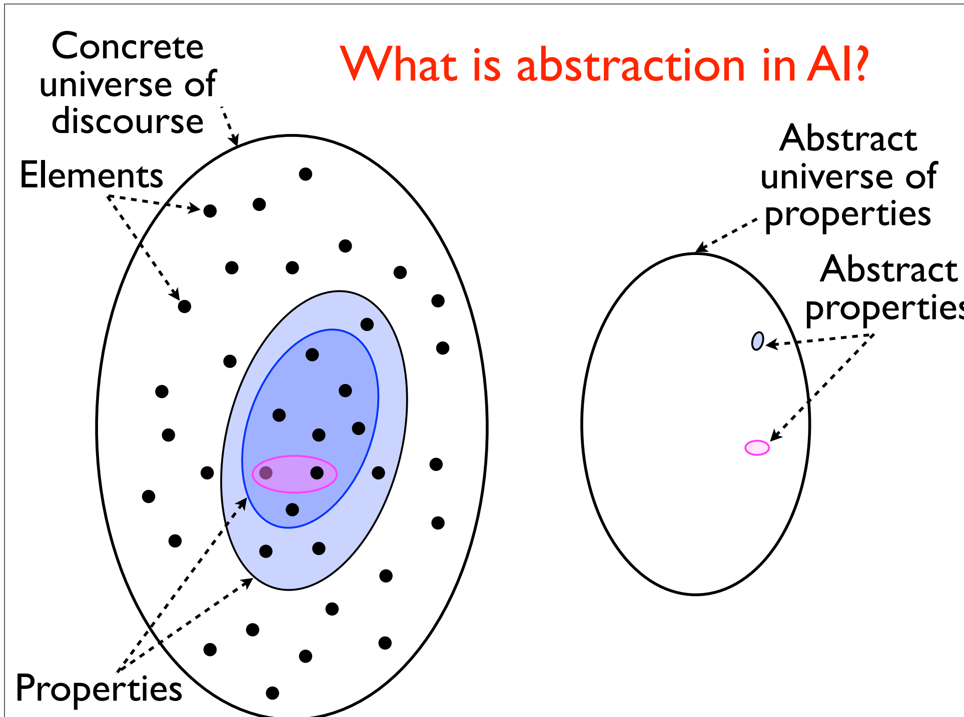
What is abstraction in AI?

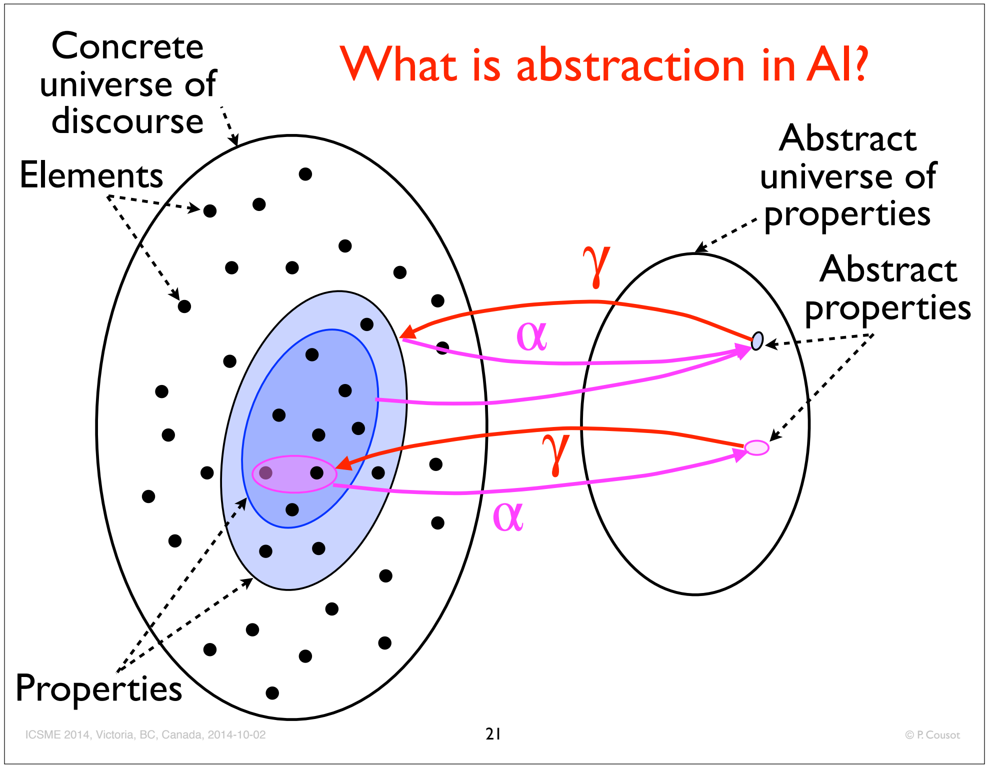


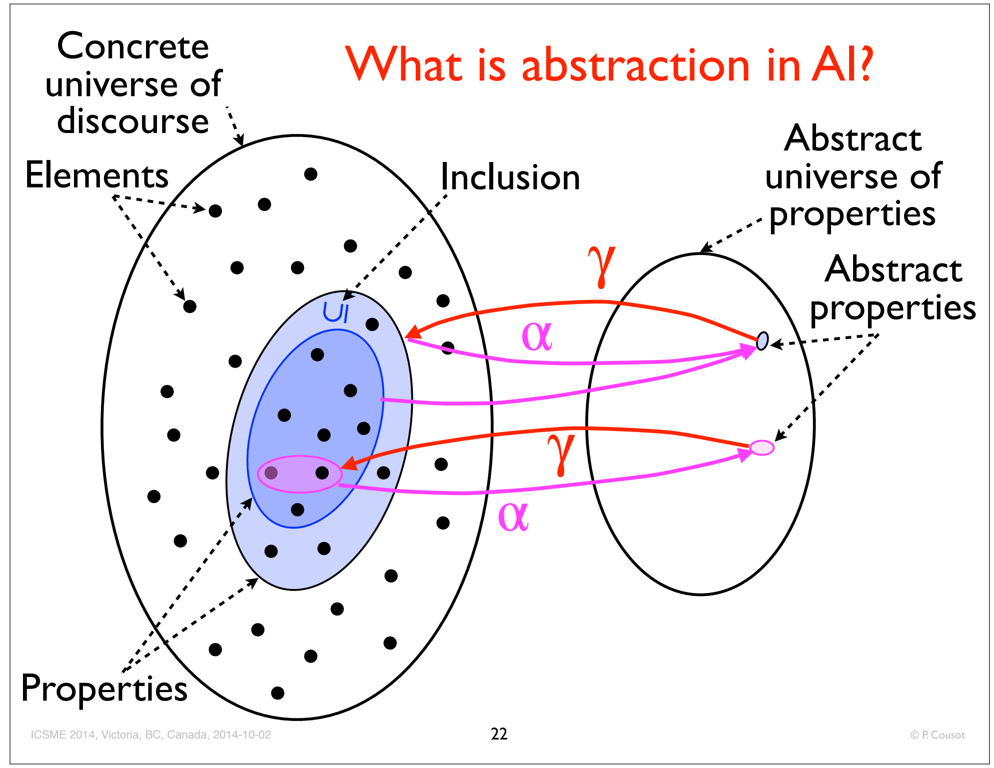
What is abstraction in AI?

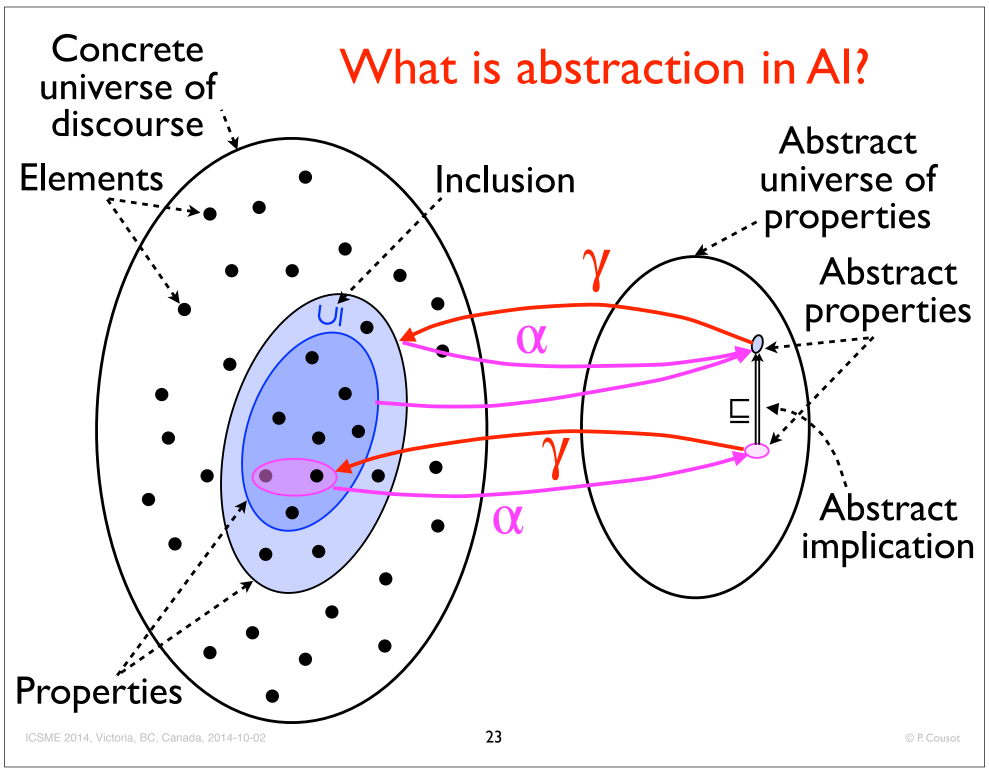


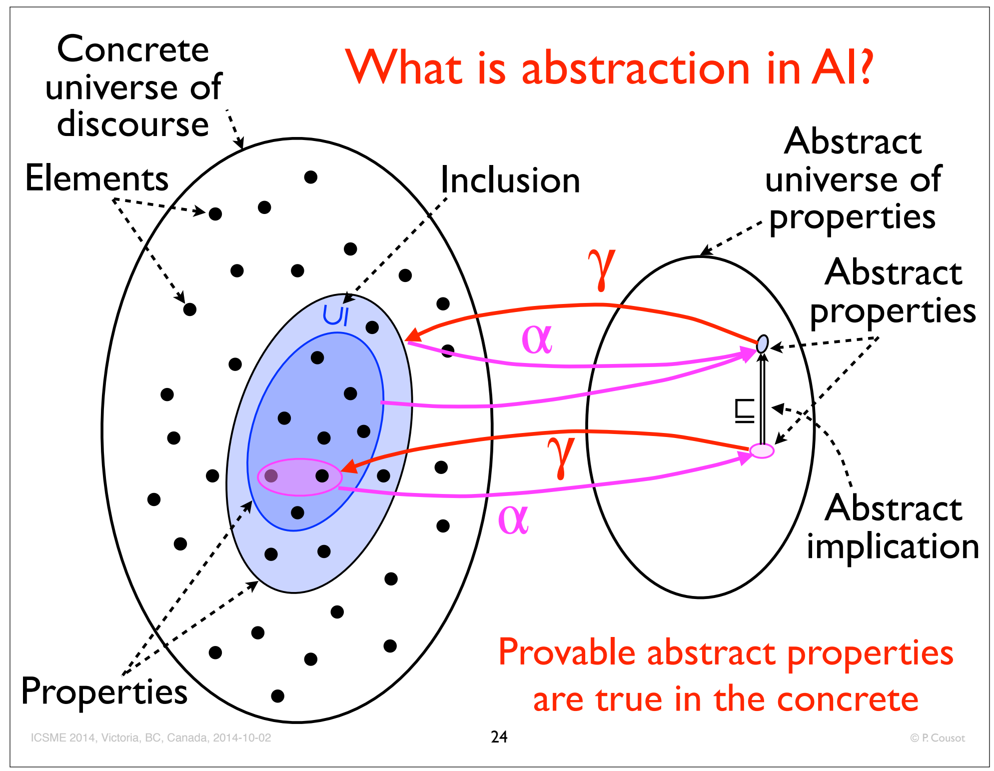
What is abstraction in AI?




What is abstraction in AI?
Concrete universe of discourse
Elements
Abstract universe of properties
Abstract properties
γ
α
γ
α
Properties
ICSME 2014, Victoria, BC, Canada, 2014-10-02
21
© P. Cousot


What is abstraction in AI?
Concrete universe of discourse
Elements
Inclusion
⊆
Abstract universe of properties
Abstract properties
γ
α
γ
α
Properties
ICSME 2014, Victoria, BC, Canada, 2014-10-02
22
© P. Cousot


What is abstraction in AI?
Concrete universe of discourse
Elements
Inclusion
⊆
Abstract universe of properties
Abstract properties
γ
α
⊑
γ
α
Abstract implication
Properties
ICSME 2014, Victoria, BC, Canada, 2014-10-02
23
© P. Cousot


What is abstraction in AI?
Concrete universe of discourse
Elements
Inclusion
⊆
Abstract universe of properties
Abstract properties
γ
α
⊑
γ
α
Abstract implication
Properties
Provable abstract properties are true in the concrete
ICSME 2014, Victoria, BC, Canada, 2014-10-02
24
© P. Cousot

Abstract interpretation: example

Theory:

Galois Connections We recall from [11] that a Galois connection $\langle C, \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq\rangle$ is such that $\langle C, \preceq\rangle$ and $\langle A, \sqsubseteq\rangle$ are partial orders, $\alpha \in C \rightarrow A$ and $\gamma \in C \rightarrow A$ satisfy $\forall x \in C : \forall y \in A : \alpha(x) \sqsubseteq y \Longleftrightarrow x \preceq \gamma(y)$. We write $\langle C, \preceq\rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\!\twoheadrightarrow}} \langle A, \sqsubseteq\rangle$ to denote that the abstraction function α is surjective, and hence that there are no multiple representations for the same concrete property in the abstract. If the C and A are complete lattices, and α is join-preserving, then it exists a unique γ such that $\langle C, \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq\rangle$.

Abstract domains We let $\mathtt{S} \in \mathbb{S}[\![\vec{\mathtt{v}}]\!]$ be a statement with visible variables $\vec{\mathtt{v}}$ and $\mathcal{P}[\![\vec{\mathtt{v}}]\!]$ be the set of unary predicates on variables $\vec{\mathtt{v}}$. Predicates can be isomorphically represented as Boolean functions $P \in \mathcal{P}[\![\vec{\mathtt{v}}]\!] \triangleq \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!] \rightarrow \mathbb{B}$ mapping values $\vec{v} \in \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!]$ of vector values of variables $\vec{\mathtt{v}}$ to Booleans: $P(\vec{v}) \in \mathbb{B} \triangleq \{\mathtt{true}, \mathtt{false}\}$. Predicates are ordered according to $\dot{\Longrightarrow}$, *i.e.*, the pointwise lifting of logical implication to functions:

$$P \dot{\Longrightarrow} P' \triangleq \forall \vec{v} \in \vec{\mathcal{V}}[\![\vec{\mathtt{v}}]\!] : P(\vec{\mathtt{v}}) \Longrightarrow P'(\vec{\mathtt{v}}).$$

For example $\boldsymbol{\lambda} x \bullet x = 0 \dot{\Longrightarrow} \boldsymbol{\lambda} x \bullet x \geqslant 0$. Predicates with partial order $\dot{\Longrightarrow}$ form a complete Boolean lattice:

$$\langle \mathcal{P}[\![\vec{\mathtt{v}}]\!], \dot{\Longrightarrow}, \dot{\mathtt{false}}, \dot{\mathtt{true}}, \dot{\vee}, \dot{\wedge}, \dot{\neg}\rangle$$

where $\dot{\mathtt{false}}$ is the infimum, $\dot{\mathtt{true}}$ is the supremum, $\dot{\vee}$ is the least upper bound (lub), $\dot{\wedge}$ is the greatest lower bound (glb), and $\dot{\neg}$ is the unique complement for the partial order $\dot{\Longrightarrow}$ on the set $\mathcal{P}[\![\vec{\mathtt{v}}]\!]$.

The *precondition abstract domain* $\langle A[\![\vec{\mathtt{v}}]\!], \sqsubseteq_1\rangle$ is an abstract domain expressing properties of the variables $\vec{\mathtt{v}}$ where the partial order $\sqsubseteq_1$ abstracts logical implication. The meaning of an abstract property $\overline{P} \in A[\![\vec{\mathtt{v}}]\!]$ is a concrete property $\gamma_1(\overline{P}) \in \mathcal{P}[\![\vec{\mathtt{v}}]\!]$ where the concretization

$$\gamma_1 \in \langle A[\![\vec{\mathtt{v}}]\!], \sqsubseteq_1\rangle \rightarrow \langle \mathcal{P}[\![\vec{\mathtt{v}}]\!], \dot{\Longrightarrow}\rangle$$

is increasing (*i.e.*, $\overline{P} \sqsubseteq_1 \overline{P}'$ implies $\gamma_1(\overline{P}) \dot{\Longrightarrow} \gamma_1(\overline{P}')$).

Applications:

RefactorContract($\overline{P}_{\mathtt{S}}$, $\mathtt{S}$, $\vec{\mathtt{p}}$, $\vec{\mathtt{g}}$, $\overline{Q}_{\mathtt{S}}$) {

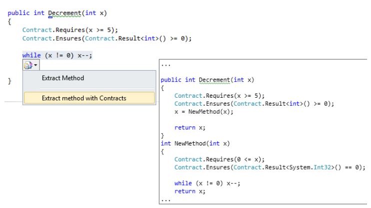
use $\langle A[\![\vec{\mathtt{p}}]\!], \sqcap_1, \Delta_1\rangle$ // precondition abstract domain
$\langle B[\![\vec{\mathtt{p}}, \vec{\mathtt{p}}]\!], \sqcap_2, \Delta_2\rangle$ // postcondition abstract domain
$\overline{\mathrm{post}}$ // forward analyser with widening/narrowing
$\overline{\widetilde{\mathrm{pre}}}$ // backward analyser with widening/narrowing

// abstract projection on potentially used variables $\vec{\mathtt{p}}$
$\langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle = \langle \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{P}_{\mathtt{S}}), \downarrow_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}(\overline{Q}_{\mathtt{S}})\rangle$;
// infer a correct safety abstract contract
Let $\overline{P}_{\mathtt{m}}$ be the abstract safety pre-condition for $\mathtt{S}$ computed by the static analysis [18];
$\overline{Q}_{\mathtt{m}} = \overline{\mathrm{post}}[\![\mathtt{S}\lceil_{\vec{\mathtt{p}}}]\!]\overline{P}_{\mathtt{m}}$; // forward abstract static analysis
// $\bar{\{}\, \overline{P}_{\mathtt{m}} \,\bar{\}}\, \mathtt{S}|_{\vec{\mathtt{p}}\setminus\vec{\mathtt{g}}}\, \bar{\{}\, \overline{Q}_{\mathtt{m}} \,\bar{\}}$ holds

$\langle \overline{P}_R, \overline{Q}_R\rangle = \langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$;
do
// compute $\langle X, Y\rangle = \overline{F}_R[\![\mathtt{S}]\!](\langle \overline{P}_R, \overline{Q}_R\rangle)$
$X = \overline{P}_{\mathtt{m}} \sqcap_1 \overline{P}_R \sqcap_1 \overline{\widetilde{\mathrm{pre}}}[\![\mathtt{S}\lceil_{\vec{\mathtt{p}}}]\!]\overline{Q}_R$; // backward analysis
$Y = \overline{Q}_{\mathtt{m}} \sqcap_2 \overline{Q}_R \sqcap_2 \overline{\mathrm{post}}[\![\mathtt{S}\lceil_{\vec{\mathtt{p}}}]\!]\overline{P}_R$; // forward analysis
$\langle \overline{P}_R, \overline{Q}_R\rangle = \langle \overline{P}_R \,\Delta_1\, X, \overline{Q}_R \,\Delta_2\, Y\rangle$; // narrowing
while $\langle \overline{P}_R, \overline{Q}_R\rangle \neq \langle X, Y\rangle$;
// $\mathrm{gfp}^{\overset{cc}{\sqsubseteq}}_{\langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle}\, \overline{F}_R[\![\mathtt{S}]\!] \overset{cc}{\sqsubseteq} \langle \overline{P}_R, \overline{Q}_R\rangle \overset{cc}{\sqsubseteq} \langle \overline{P}^{\curlyvee}_{\mathtt{S}}, \overline{Q}^{\curlyvee}_{\mathtt{S}}\rangle$ holds
return $\langle \overline{P}_R, \overline{Q}_R\rangle$; // $(\bar{\mathrm{a}})$ $\overline{\mathrm{validity}}$ & $(\bar{\mathrm{b}})$ $\overline{\mathrm{safety}}$ hold
}

Algorithm 5. Algorithm $\overline{\mathrm{EMC}}$ (Extract Methods with Abstract Contracts) computing an approximation of a greatest fixpoint with convergence acceleration.

Practice:



Patrick Cousot, Radhia Cousot, Francesco Logozzo, Michael Barnett: An abstract interpretation framework for refactoring with application to extract methods with contracts. OOPSLA 2012: 213-232

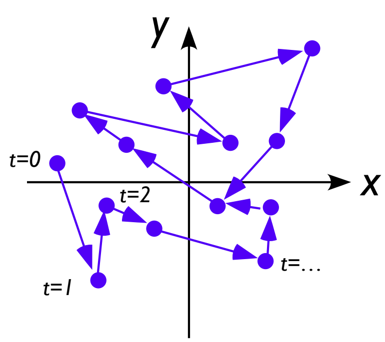
A very informal introduction to abstract interpretation

Patrick Cousot, Radhia Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252

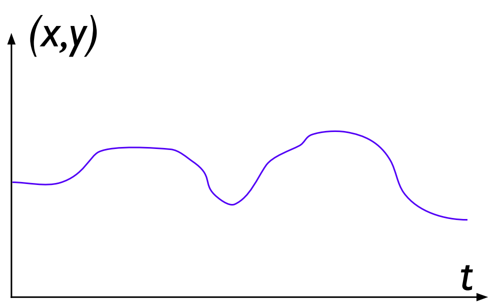
Patrick Cousot, Radhia Cousot: Systematic Design of Program Analysis Frameworks. POPL 1979: 269-282

I) Define the programming language semantics

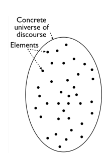
Formalize the concrete ***executions*** *of programs (e.g. transition system)*



Trajectory
in state space

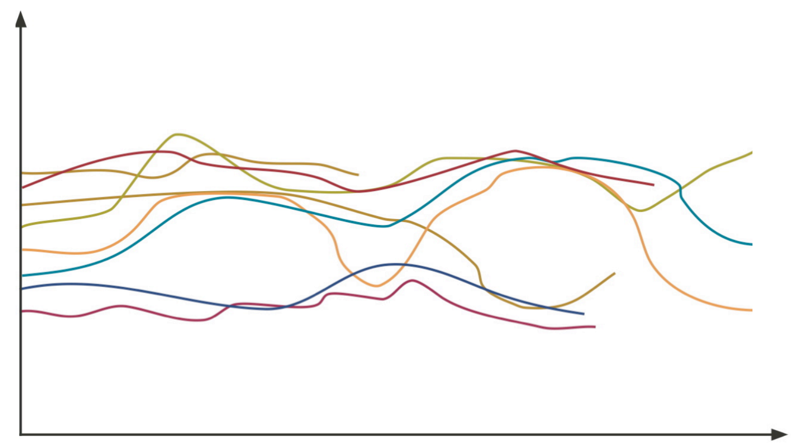


Space/time trajectory

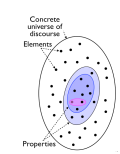


II) Define the program properties of interest

Formalize what you are interested to ***know*** *about program behaviors*

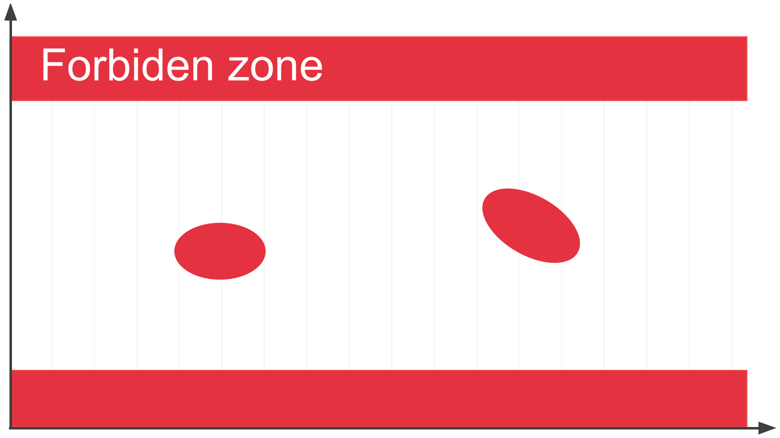


We are interested in the set of possible trajectories



III) Define which specification must be checked

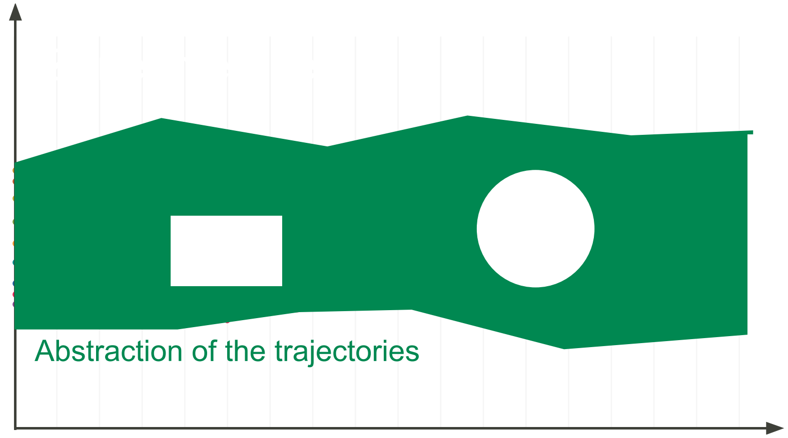
*Formalize what you are interested to **prove** about program behaviors*



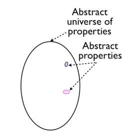
No trajectory should hit the forbidden zone

IV) Choose the appropriate abstraction

***Abstract** away all information on program behaviors irrelevant to the proof*

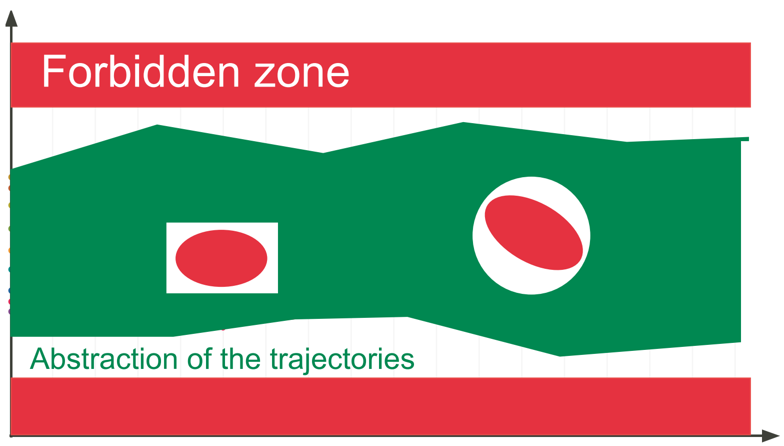


Abstraction by geometric forms (rectangles, polyhedra, ellipsoids, abstraction by parts, etc)



V) Mechanically verify in the abstract

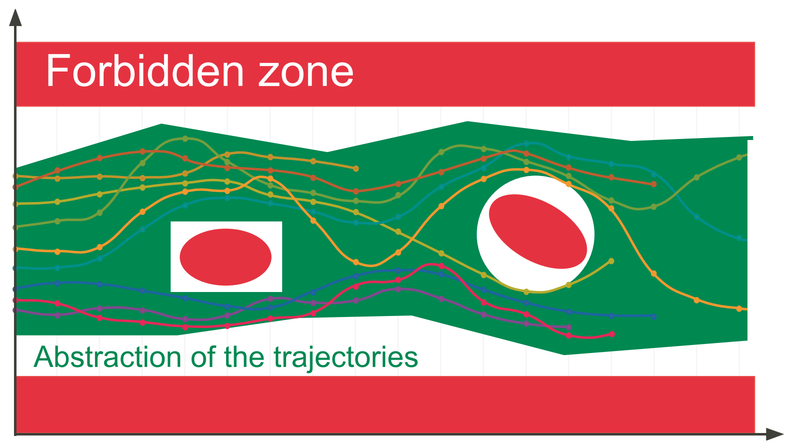
*The proof is fully **automatic***

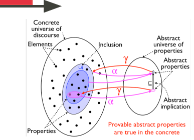


Provable abstract properties are true in the concrete

Soundness of the abstract verification

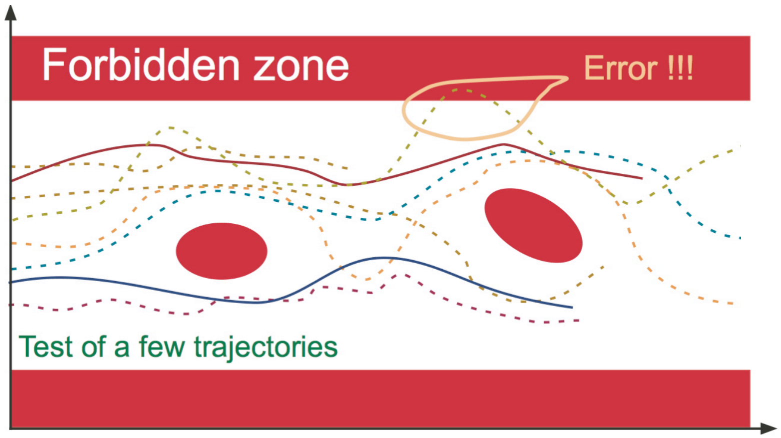
*Never forget any possible case so the **abstract proof is correct in the concrete***

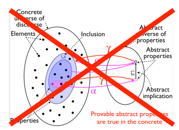




Unsound validation: testing

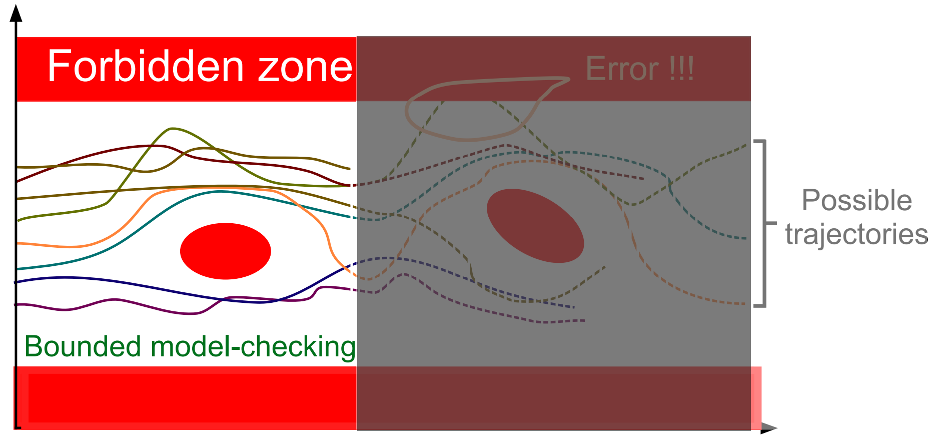
Try a few cases

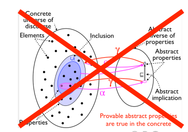




Unsound validation: bounded model-checking

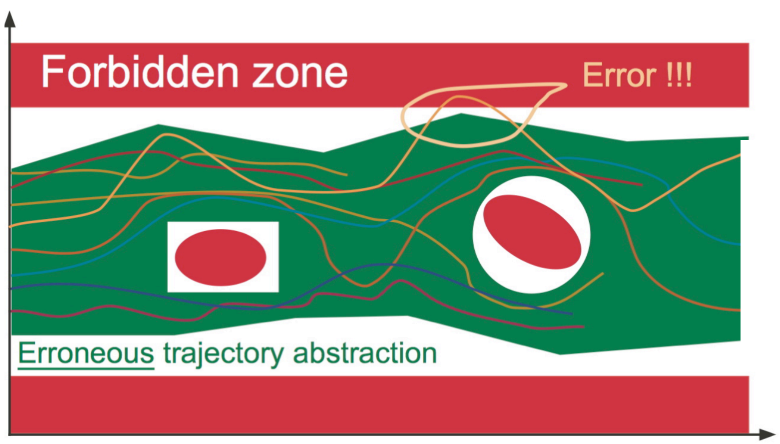
Simulate the beginning of all executions

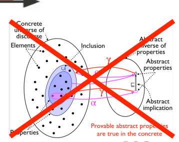




Unsound validation: static analysis

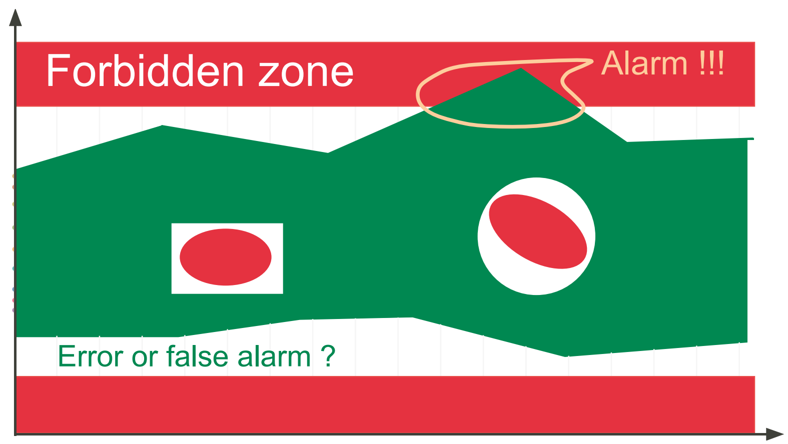
*Many static analysis tools are **unsound** (e.g. Coverity, etc.) so inconclusive*



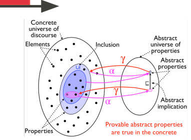


Incompleteness

When abstract proofs may fail while concrete proofs would succeed

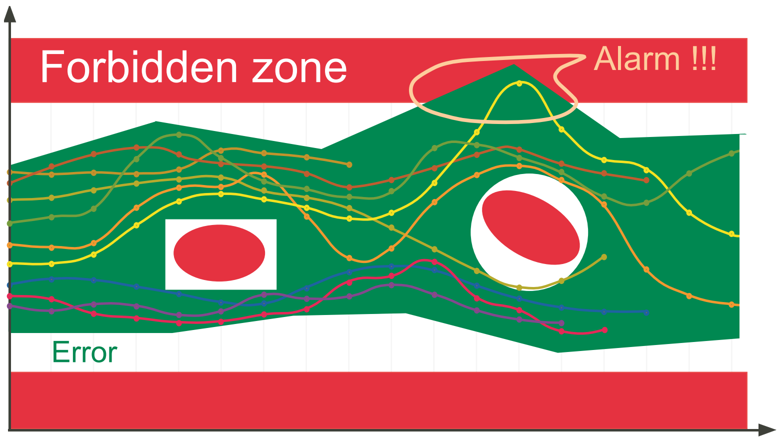


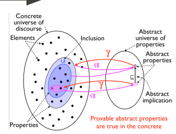
By soundness an alarm must be raised for this overapproximation!



True error

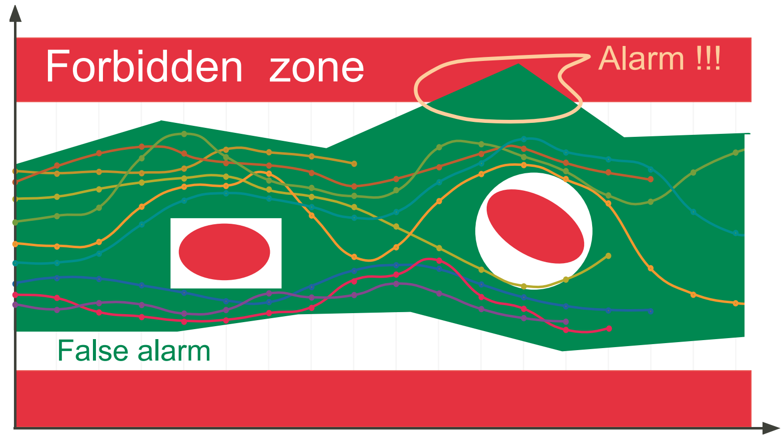
The abstract alarm may correspond to a concrete error



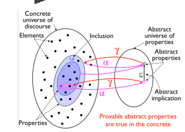


False alarm

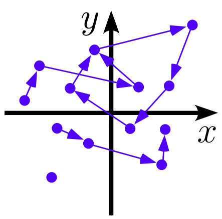
The abstract alarm may correspond to no concrete error (false negative)



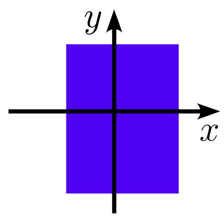
The only solution is to refine the analysis to take more properties into account (e.g. specifically for a domain of application)!



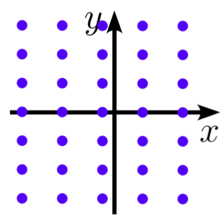
Combination of abstractions in Astrée



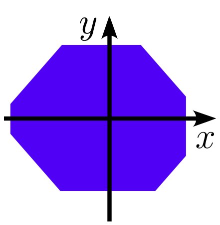
Collecting semantics:[1] partial traces



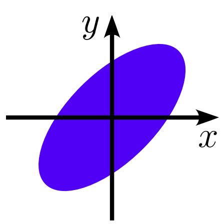
Intervals: $\mathtt{x} \in [a, b]$



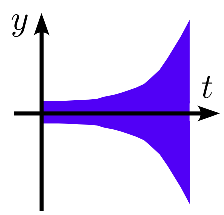
Simple congruences: $\mathtt{x} \equiv a[b]$



Octagons: $\pm\mathtt{x} \pm \mathtt{y} \leqslant a$



Ellipses: $\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{x}\mathtt{y} \leqslant d$



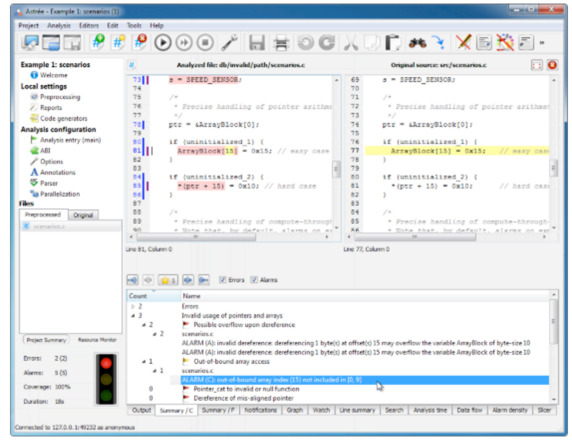
Exponentials: $-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

Examples of abstract interpretation-based program verification tools

Example 1: Astrée

Astrée

- Commercially available: `www.absint.com/astree/`



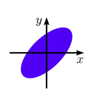
- Effectively used in production to qualify truly large and complex software in transportation, communications, medicine, *etc*

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival: **A static analyzer for large safety-critical software.** *PLDI 2003*: 196-207

Example of domain-specific abstraction: ellipses

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

To be inferred, not tested, checked, or verified



Abstract interpretation

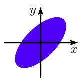
- Abstract interpretation is the only formal method able to automatically infer program properties
- All others can only check your assertions

Types are abstract interpretations, see Patrick Cousot: Types as Abstract Interpretations. POPL 1997: 316-331

Example of domain-specific abstraction: ellipses

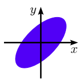
```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
             + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [?????, ?????]                    */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```

To be inferred, not tested, checked, or verified



Example of domain-specific abstraction: ellipses

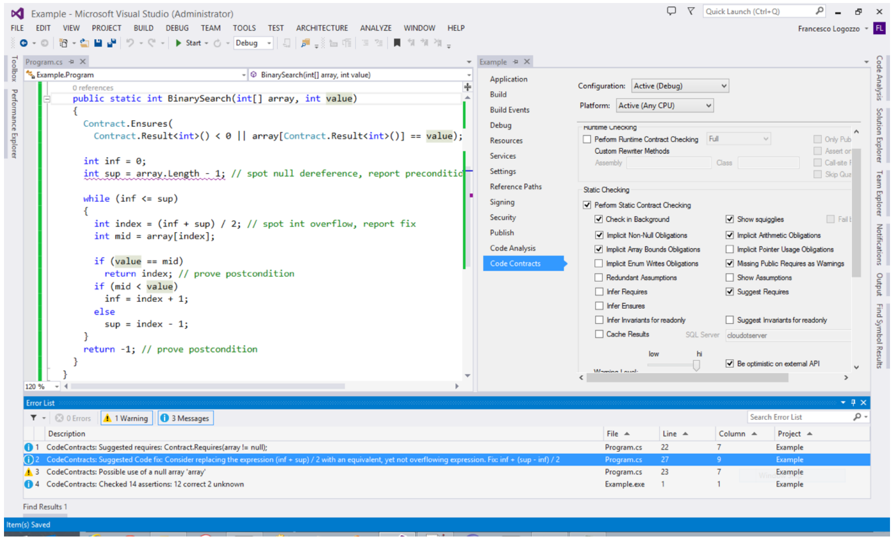
```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
             + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [-1418.3753, 1418.3753]        */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```



Example II: cccheck

Code Contract Static Checker (cccheck)

- Available within MS Visual Studio



Manuel Fähndrich, Francesco Logozzo: Static Contract Checking with Abstract Interpretation. FoVeOOS 2010: 10-30

Comments on screenshot (courtesy Francesco Logozzo)

- A screenshot from Clousot/`cccheck` on the classic binary search.
- The screenshot shows from left to right and top to bottom
  1. C# code + CodeContracts with a buggy BinarySearch
  2. `cccheck` integration in VS (right pane with all the options integrated in the VS project system)
  3. `cccheck` messages in the VS error list
- The features of `cccheck` that it shows are:
  1. basic abstract interpretation:
     a. the loop invariant to prove the array access correct and that the arithmetic operation may overflow is inferred fully automatically
     b. different from deductive methods as e.g. ESC/Java or Boogie or Dafny where the loop invariant must be provided by the end-user
  2. inference of necessary preconditions:
     a. Clousot finds that array may be null (message 3)
     b. Clousot suggests and propagates a necessary precondition invariant (message 1)
  3. array analysis (+ disjunctive reasoning):
     a. to prove the postcondition one must infer properties of the content of the array
     b. please note that the postcondition is true even if there is no precondition requiring the array to be sorted.
  4. verified code repairs:
     a. from the inferred loop invariant does not follow that index computation does not overflow
     b. suggest a code fix for it (message 2)

Conclusion

To explore abstract interpretation...

- A good starting point:

Patrick Cousot and Radhia Cousot:
Abstract interpretation: past, present and future.

In:
Thomas A. Henzinger, Dale Miller (Eds.): Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. ACM 2014, ISBN 978-1-4503-2886-9

Conclusion

- 40 years after Harlan D. Mills pioneer ideas, abstract interpretation-based formal methods have made considerable progress both in *theory* and *practice*
- May become *indispensable* as
  - safety and security become central to computer science
  - programmers are held responsible for their errors
  - machines hence programming becomes more and more complicated (if not intractable, e.g. parallelism, cloud, *etc*)

The End, Thank You