JPMorgan Chase Distinguished Lecture Series

Abstract interpretation: from principles to applications

Patrick Cousot

NYU, New York

pcousot@cs.nyu.edu cs.nyu.edu/~pcousot

Wednesday, June 30th, 2021, 11:00 AM EST.

What is program semantics,
program verification,
program dynamic or static analysis,
and abstract interpretation?

Program semantics

- syntax: a representation of a program of a language (e.g. character file, syntax tree, etc.)
- semantics: a formal description $\mathcal{S}[\![\mathtt{P}]\!]$ of the executions of a program $\mathtt{P}$ of a programming language (e.g. set of execution traces, set of reachable states at each program point)

Program semantics

- syntax: a representation of a program of a language (e.g. character file, syntax tree, etc.)
- semantics: a formal description $\mathcal{S}[\![\mathtt{P}]\!]$ of the executions of a program $\mathtt{P}$ of a programming language (e.g. set of execution traces, set of reachable states at each program point)

Verification of a Specification

- specification: a desired property of the program semantics (e.g. all executions are finite, no runtime errors)
- verification: a mathematical proof that a program semantics satisfies a specification
- induction: proofs are by induction/recurrence to handle loops/recursions
- inductive argument: the induction hypothesis in proof by induction/recurrence to handle loops/recursions

Undecidability

- finite mechanical proofs must fail on infinitely many programs

Undecidability

- finite mechanical proofs must fail on infinitely many programs

Verification Methods

- the proof is incorrect (e.g. Coverity)
- the proof is restricted to decidable cases (e.g. termination of linear arithmetic loop with no inner test or loop)
- the proof goes out of memory/time resources (e.g. model-checking)
- the proof requires human interaction (e.g. deductive methods)
- the proof is correct, always terminate, but may be inconclusive (static analysis).

Dynamic analysis

- the proof is done by monitoring execution at runtime
- one execution at a time (cannot handle accurately e.g. dependency/non-interference)

Dynamic analysis

- the proof is done by monitoring execution at runtime
- one execution at a time (cannot handle accurately e.g. dependency/non-interference)

Symbolic execution

- give symbolic names to values (of variables, inputs, array elements, etc.)
- not all paths can be explored (e.g. non-termination)

Dynamic analysis

- the proof is done by monitoring execution at runtime
- one execution at a time (cannot handle accurately e.g. dependency/non-interference)

Symbolic execution

- give symbolic names to values (of variables, inputs, array elements, etc.)
- not all paths can be explored (e.g. non-termination)

Bug Finding

- specify a program path in the program (e.g. to a potential bug)
- prove its [un]feasibility by a SMT solver

Dynamic analysis

- the proof is done by monitoring execution at runtime
- one execution at a time (cannot handle accurately e.g. dependency/non-interference)

Symbolic execution

- give symbolic names to values (of variables, inputs, array elements, etc.)
- not all paths can be explored (e.g. non-termination)

Bug Finding

- specify a program path in the program (e.g. to a potential bug)
- prove its [un]feasibility by a SMT solver

These are not verification methods!

Static analysis

- the proof is done by considering the program text only
- valid for all executions

Static analysis

- the proof is done by considering the program text only
- valid for all executions

Abstract Interpretation

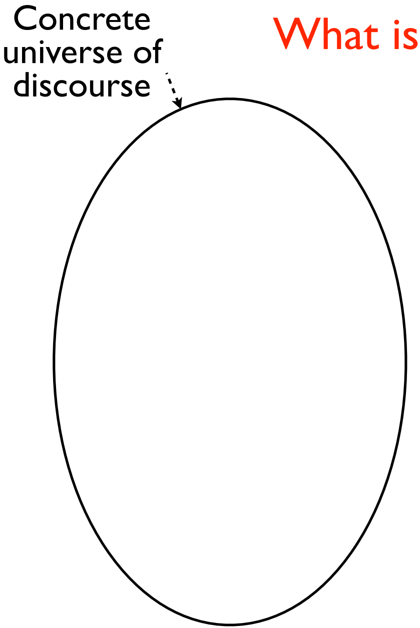
- a theory of abstraction (of the semantics of programming languages)
- applied to the design of semantics, verification methods, and analysis methods

Main objectives of abstract interpretation

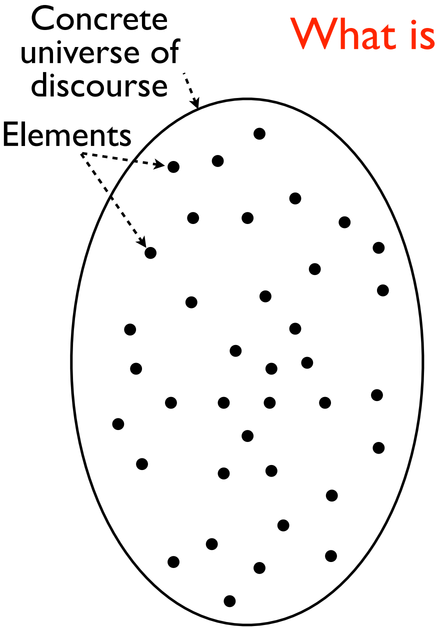
- soundness: what is proved is true
- completeness: what is true can be proved (e.g. for manual verification methods)
- incompleteness: what is true may not be provable due to approximations (for static analysis methods)
- constructive design: by calculus, guided by the theory, machine checkable.

An informal introduction to abstract interpretation

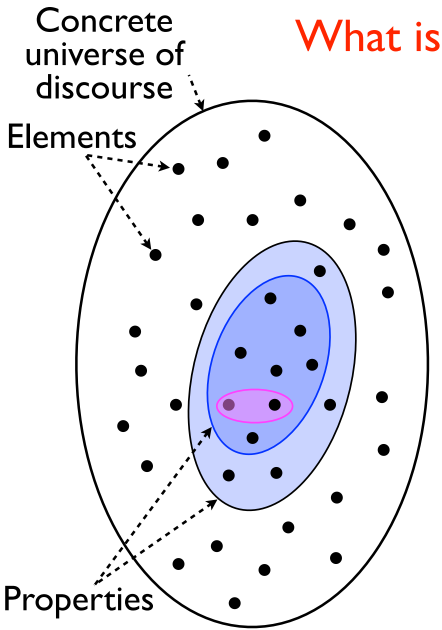
What is abstraction in AI?



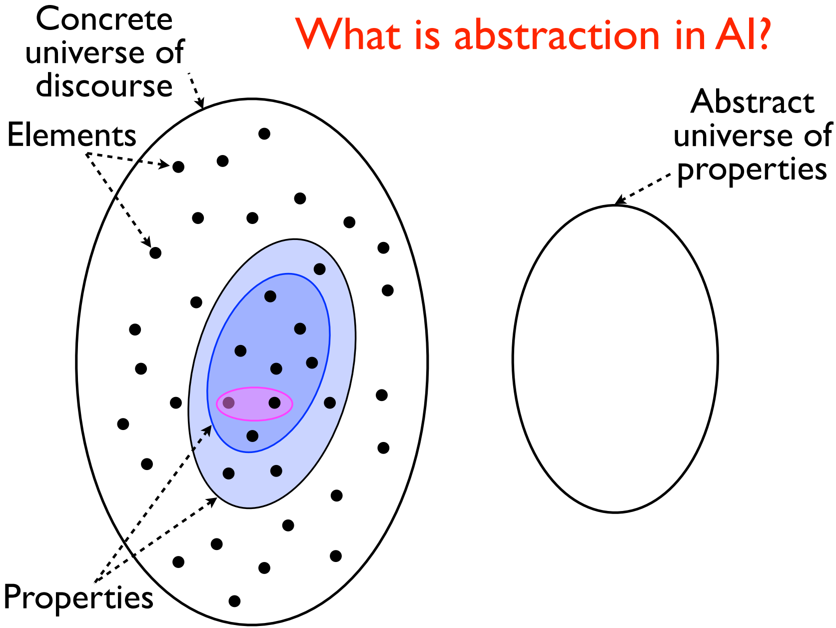
What is abstraction in AI?



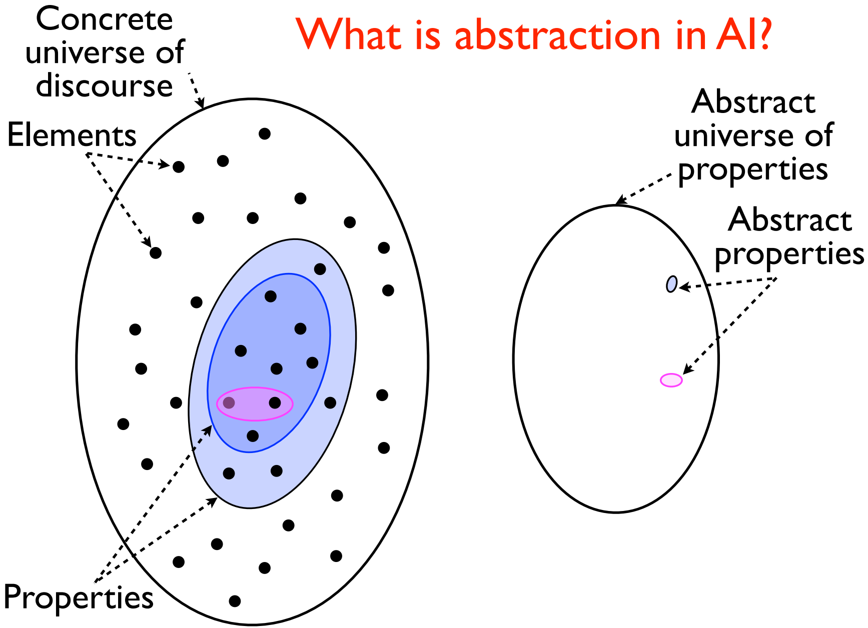
What is abstraction in AI?



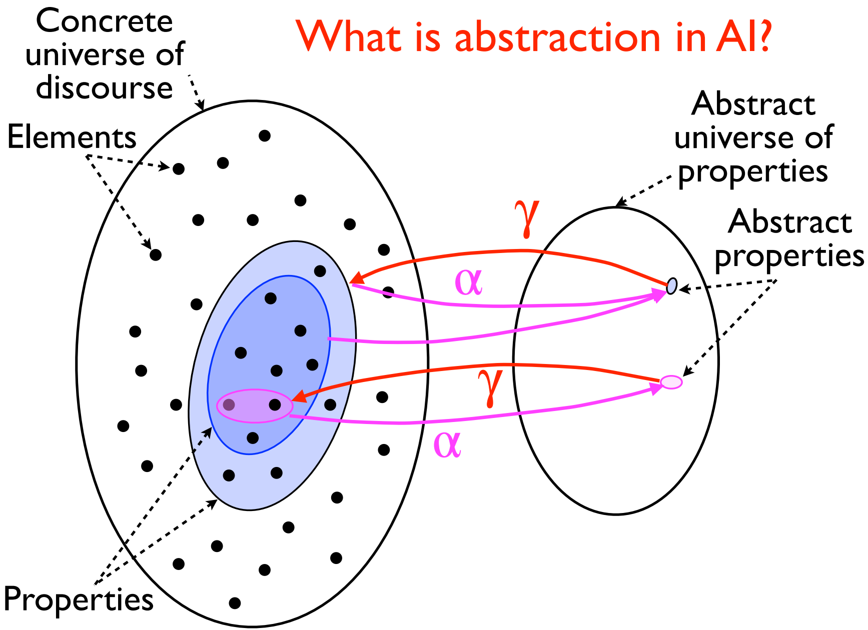
What is abstraction in AI?



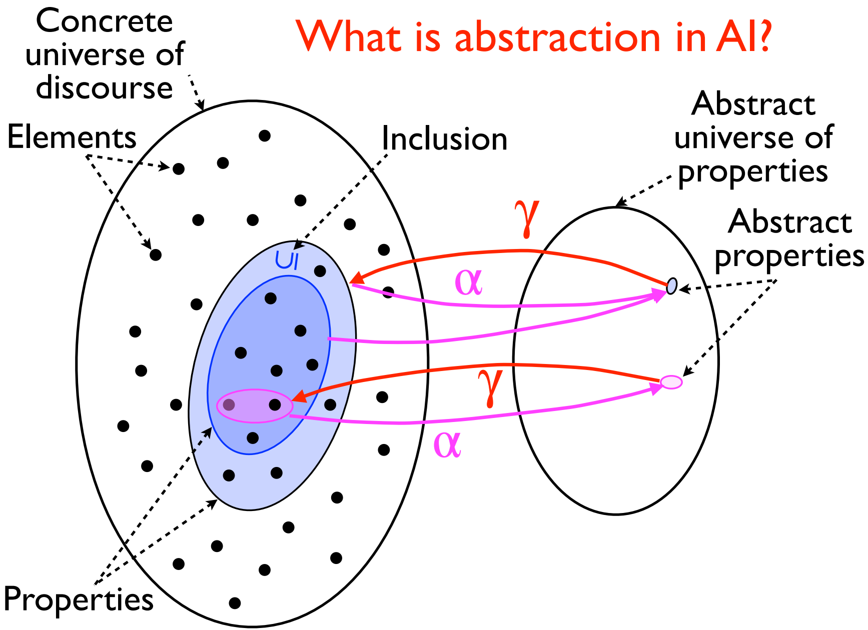
What is abstraction in AI?



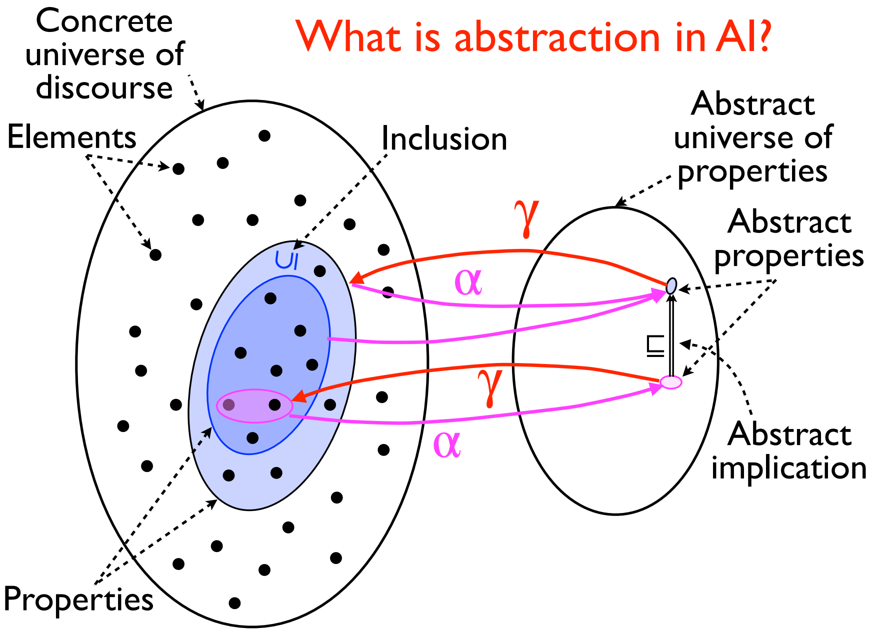
What is abstraction in AI?



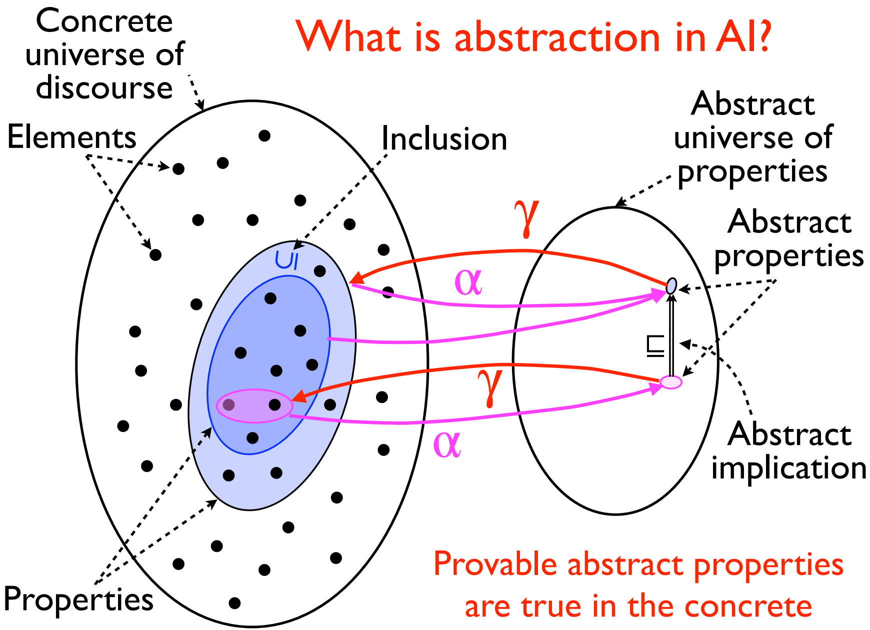
What is abstraction in AI?



What is abstraction in AI?



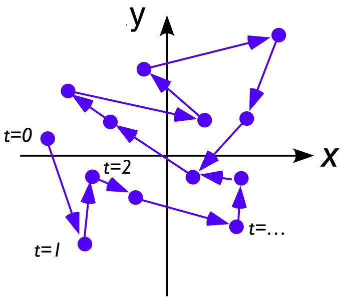
What is abstraction in AI?



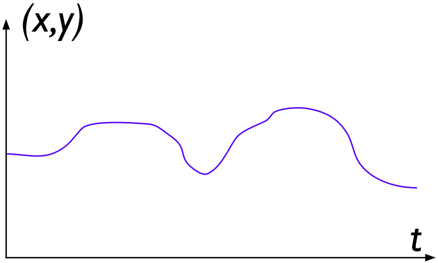
Provable abstract properties are true in the concrete

1) Define the programming language semantics

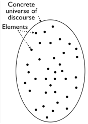
*Formalize the concrete **executions** of programs (e.g. transition system)*



Trajectory
in state space

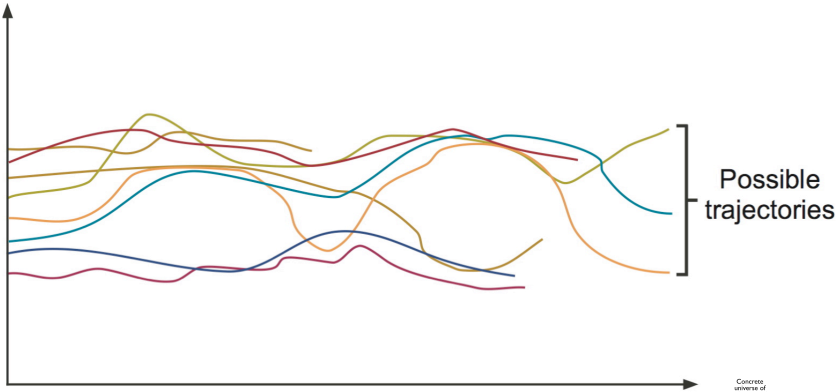


Space/time trajectory

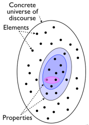


II) Define the program properties of interest

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

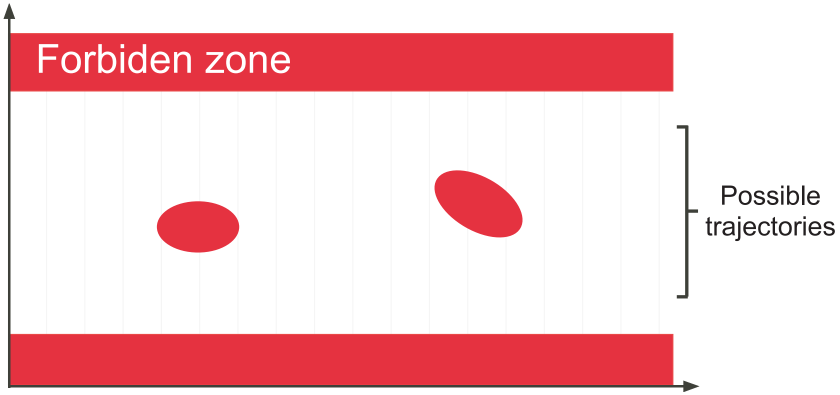


We are interested in the set of possible trajectories



III) Define which specification must be checked

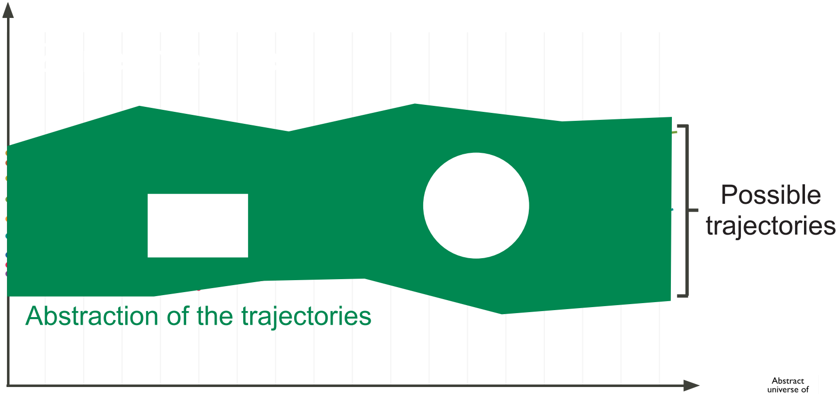
Formalize what you are interested to ***prove*** *about program behaviors*



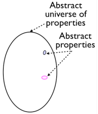
No trajectory should hit the forbidden zone

IV) Choose the appropriate abstraction

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

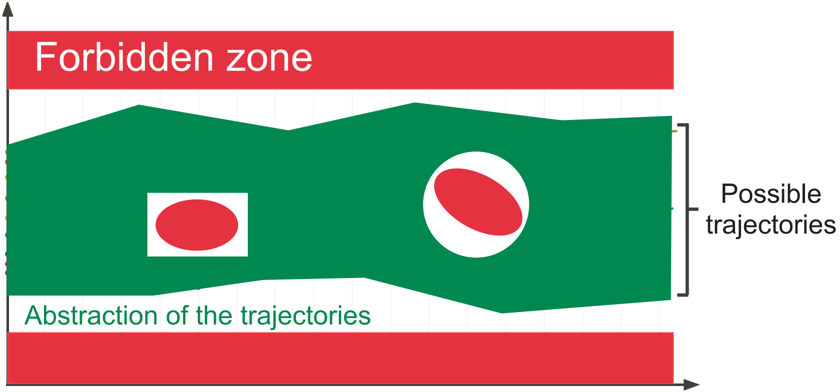


Abstraction by geometric forms (rectangles, polyh
ellipsoids, abstraction by parts, etc)



V) Mechanically verify in the abstract

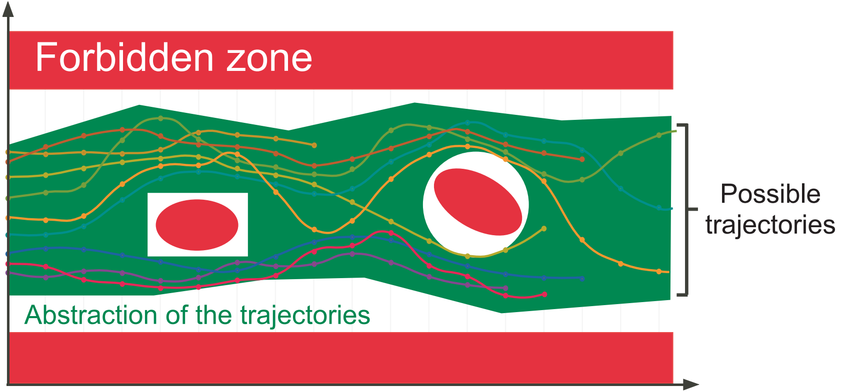
The proof is fully ***automatic***

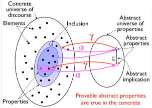


Provable abstract properties
are true in the concrete

Soundness of the abstract verification

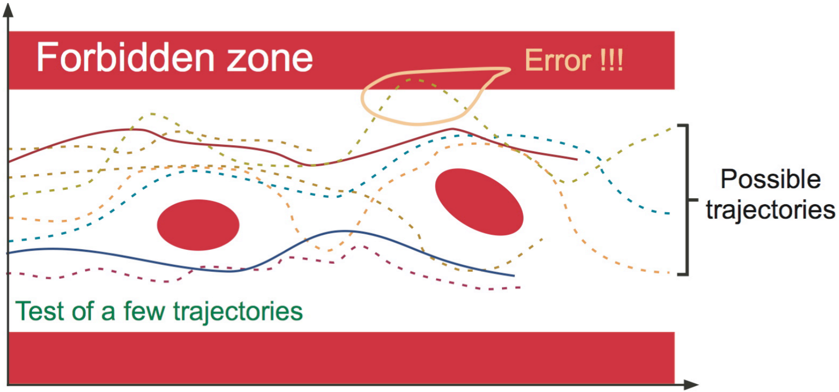
Never forget any possible case so the ***abstract proof is correct in the concrete***

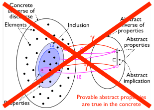




Unsound validation: testing

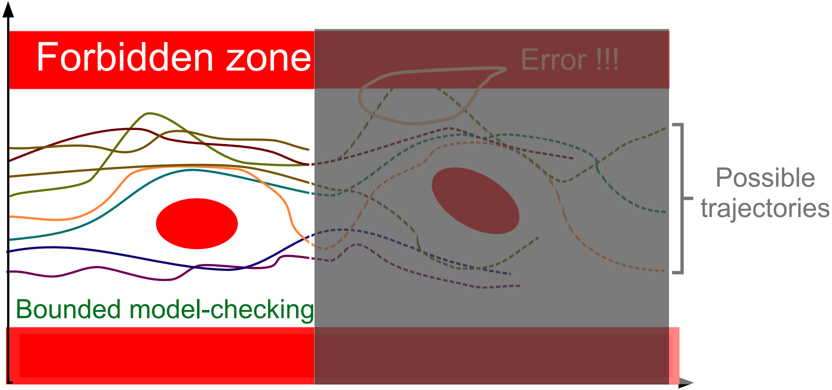
Try a few cases

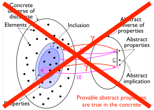




Unsound validation: bounded model-checking

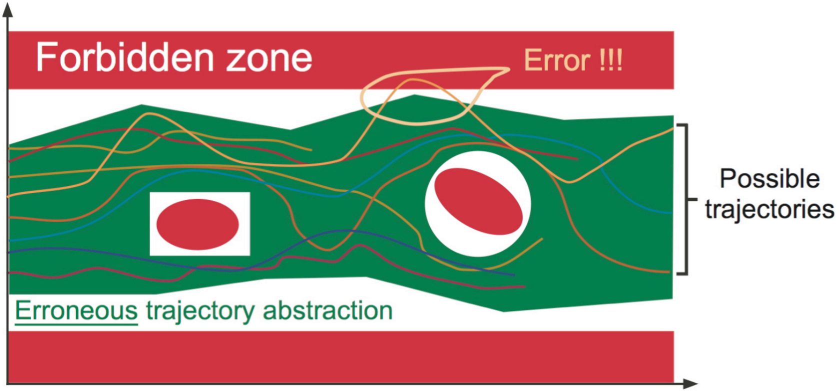
Simulate the beginning of all executions

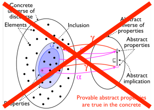




Unsound validation: static analysis

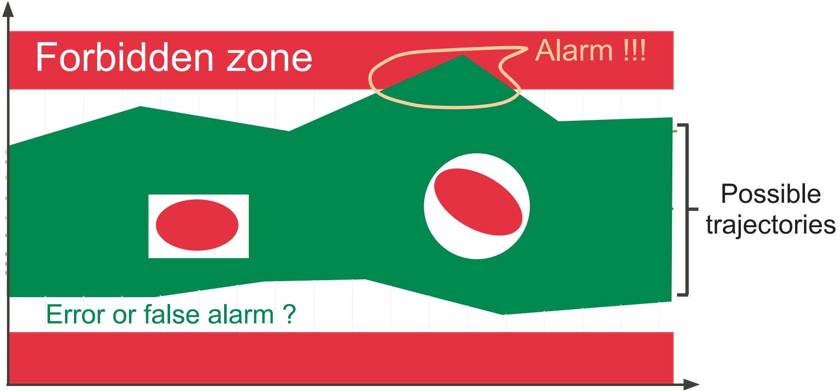
*Many static analysis tools are **unsound** (e.g. Coverity, etc.) so inconclusive*



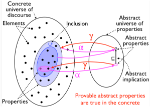


Incompleteness

When abstract proofs may fail while concrete proofs would succeed

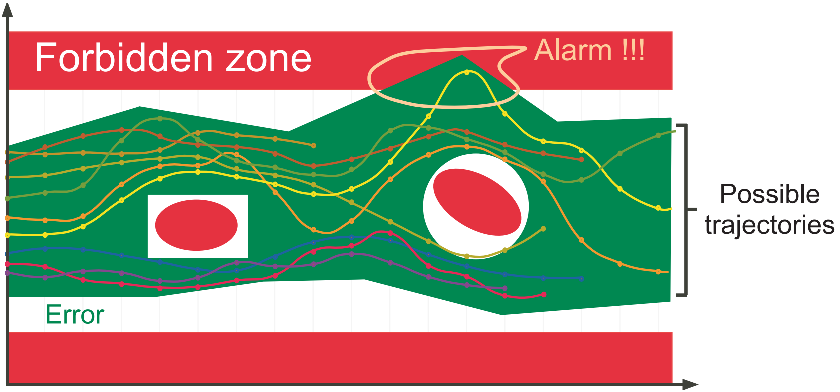


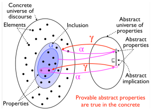
By soundness an alarm must be raised for this overapproximation!



True error

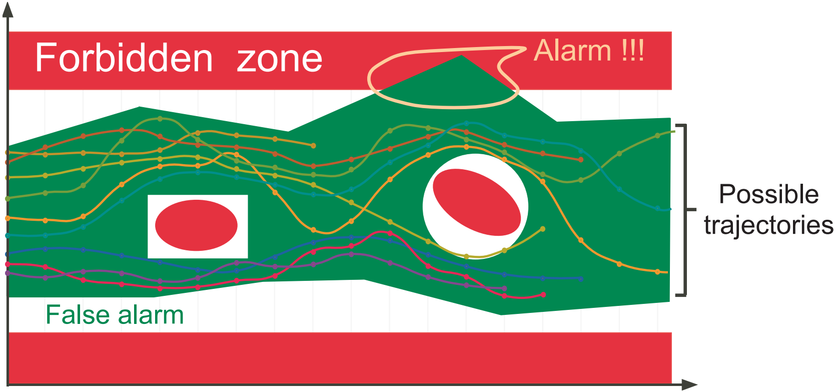
The abstract alarm may correspond to a concrete error



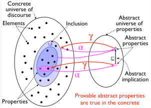


False alarm

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)!



A few basic concepts in abstract interpretation

Example of semantics

Trace semantics

- A trace semantics is a (finite or infinite) set of traces
- A trace is a finite or infinite sequence of states
- A state is a pair or a control state and a memory state
- A control state records all calls to methods leading to a program point
- A memory state records the values of variables, allocated memory, inputs, etc.

Example of prefix trace semantics

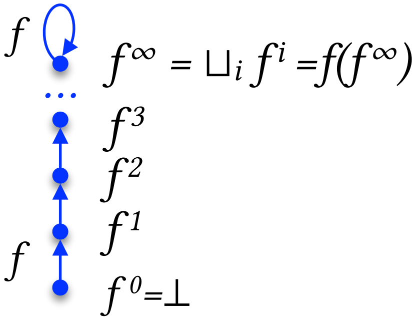
- a simple while language
- a state $\langle \ell, \rho \rangle$ is a pair *program point* $\times$ *environments* (assigning values of variables)
- defined by structural induction (induction on the syntax of programs)
- prefix traces of an assignment

Prefix state traces of an assignment statement $\mathsf{S} ::= {}^{\ell}\ \mathsf{x} = \mathsf{A}\ ;$

$$\widehat{\boldsymbol{\mathcal{S}}}^{*}_{\mathbb{S}}[\![\mathsf{S}]\!] = \{\langle \ell, \rho \rangle \mid \rho \in \mathbb{E}\mathrm{v}\} \cup \{\langle \ell, \rho \rangle \langle \mathsf{after}[\![\mathsf{S}]\!], \rho[\mathsf{x} \leftarrow \nu] \rangle \mid \rho \in \mathbb{E}\mathrm{v} \wedge \nu = \mathscr{A}[\![\mathsf{A}]\!]\rho\} \quad (42.4)$$

Fixpoints

- solutions to equations $x = f(x)$
- may have 0, one, or many solutions
- Tarski's fixpoint theorem ensures that there is a unique least solution $\text{lfp}^{\sqsubseteq} f$ for some order $\sqsubseteq$
- Can be calculated iteratively (as the limit of infinite iterations)



Example of prefix trace semantics (cont'd)

- prefix traces of an iteration

Prefix state traces of an iteration statement $\mathsf{S} ::= \mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b$

$$\widehat{\mathcal{S}}^*_{\mathbb{S}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!] \quad = \quad \mathsf{lfp}^{\dot\subseteq}\ \mathcal{F}^*_{\mathbb{S}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!] \tag{42.6}$$

$$\begin{aligned}
\mathcal{F}^*_{\mathbb{S}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!]\ X \quad \triangleq \quad & \{\langle \ell,\ \rho\rangle \mid \rho \in \mathbb{Ev}\} & \text{(a)}\\
& \cup \{\pi_2\langle \ell',\ \rho\rangle\langle \mathsf{after}[\![\mathsf{S}]\!],\ \rho\rangle \mid \pi_2\langle \ell',\ \rho\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\,\rho = \mathrm{ff} \wedge \ell' = \ell\} & \text{(b)}\\
& \cup \{\pi_2\langle \ell',\ \rho\rangle\langle \mathsf{at}[\![\mathsf{S}_b]\!],\ \rho\rangle \cdot \pi_3 \mid \pi_2\langle \ell',\ \rho\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\,\rho = \mathrm{tt}\ \wedge & \text{(c)}\\
& \qquad \langle \mathsf{at}[\![\mathsf{S}_b]\!],\ \rho\rangle \cdot \pi_3 \in \widehat{\mathcal{S}}^*_{\mathbb{S}}[\![\mathsf{S}_b]\!] \wedge \ell' = \ell\}
\end{aligned}$$

Maximal trace semantics

- Maximal trace semantics

$$\mathcal{S}^{+}_{\mathbb{V}}[\![\mathsf{S}]\!] \triangleq \{\pi\langle\ell,\ \rho\rangle \in \mathcal{S}^{*}_{\mathbb{V}}[\![\mathsf{S}]\!] \mid (\ell = \mathsf{after}[\![\mathsf{S}]\!]) \vee (\mathsf{escape}[\![\mathsf{S}]\!] \wedge \ell = \mathsf{break\text{-}to}[\![\mathsf{S}]\!])\}$$

$$\mathcal{S}^{\infty}_{\mathbb{V}}[\![\mathsf{S}]\!] \triangleq \lim(\mathcal{S}^{*}_{\mathbb{V}}[\![\mathsf{S}]\!])$$

- Limit

$$\lim \mathcal{T} \triangleq \{\pi \in \mathbb{T}^{\infty} \mid \forall n \in \mathbb{N} \,.\, \pi[0..n] \in \mathcal{T}\}.$$

Example of abstractions

Reachability semantics (invariance)

- Collects reachable states at each program point
- $\mathcal{S}^{\vec{r}}[\![\mathrm{P}]\!] = \alpha(\mathcal{S}[\![\mathrm{P}]\!]) = \mathcal{P}_0 \mapsto \ell \mapsto \{\rho \mid \exists \sigma\sigma'.\sigma\langle\ell,\ \rho\rangle\sigma' \in \mathcal{S}[\![\mathrm{P}]\!]\}$
- By calculational design we get

Reachability of an iteration statement $\mathsf{S} ::= \mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b$

$$\widehat{\mathcal{S}}^{\vec{e}}[\![\mathsf{S}]\!]\,\mathscr{P}_0\,\ell' \quad = \quad (\mathsf{lfp}^{\dot{\subseteq}}\,\mathscr{F}^{\vec{e}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!]\,\mathscr{P}_0)\,\ell' \tag{19.16}$$

$$\mathscr{F}^{\vec{e}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!]\,\mathscr{P}_0 \quad \in \quad (\mathbb{L} \to \wp(\mathbb{E}\mathbb{v}^{\vec{e}})) \xrightarrow{\nearrow} (\mathbb{L} \to \wp(\mathbb{E}\mathbb{v}^{\vec{e}}))$$

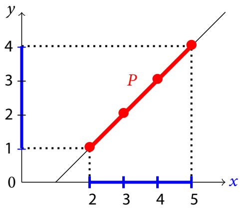
$$\begin{aligned}
&\mathscr{F}^{\vec{e}}[\![\mathtt{while}\ \ell\ (\mathsf{B})\ \mathsf{S}_b]\!]\,\mathscr{P}_0\,X\,\ell' \quad = \\
&\quad (\!\![\ \ell' = \ell\ ?\ \mathscr{P}_0 \cup \widehat{\mathcal{S}}^{\vec{e}}[\![\mathsf{S}_b]\!]\,(\mathsf{test}^{\vec{e}}[\![\mathsf{B}]\!]X(\ell))\,\ell \\
&\quad \|\ \ell' \in \mathsf{in}[\![\mathsf{S}_b]\!] \setminus \{\ell\}\ ?\ \widehat{\mathcal{S}}^{\vec{e}}[\![\mathsf{S}_b]\!]\,(\mathsf{test}^{\vec{e}}[\![\mathsf{B}]\!]X(\ell))\,\ell' \\
&\quad \|\ \ell' = \mathsf{after}[\![\mathsf{S}]\!]\ ?\ \overline{\mathsf{test}}^{\vec{e}}[\![\mathsf{B}]\!](X(\ell)) \cup \bigcup_{\ell'' \in \mathsf{breaks\text{-}of}[\![\mathsf{S}_b]\!]} \widehat{\mathcal{S}}^{\vec{e}}[\![\mathsf{S}_b]\!]\,(\mathsf{test}^{\vec{e}}[\![\mathsf{B}]\!]X(\ell))\,\ell'' \\
&\quad :\ \varnothing\ ]\!\!)
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{test}^{\vec{r}}[\![\mathsf{B}]\!]\mathscr{P} &\triangleq \{\rho \in \mathscr{P} \mid \mathscr{B}[\![\mathsf{B}]\!]\rho = \mathsf{tt}\} \\
\mathsf{test}^{\vec{R}}[\![\mathsf{B}]\!]\mathscr{P} &\triangleq \{\langle\rho_0,\ \rho\rangle \in \mathscr{P} \mid \mathscr{B}[\![\mathsf{B}]\!]\rho = \mathsf{tt}\} \\
\overline{\mathsf{test}}^{\vec{r}}[\![\mathsf{B}]\!]\mathscr{P} &\triangleq \{\rho \in \mathscr{P} \mid \mathscr{B}[\![\mathsf{B}]\!]\rho = \mathsf{ff}\} \\
\overline{\mathsf{test}}^{\vec{R}}[\![\mathsf{B}]\!]\mathscr{P} &\triangleq \{\langle\rho_0,\ \rho\rangle \in \mathscr{P} \mid \mathscr{B}[\![\mathsf{B}]\!]\rho = \mathsf{ff}\}
\end{aligned}$$

Cartesian abstraction

- We are left with sets of environments mapping variables to their values
- Cartesian abstraction



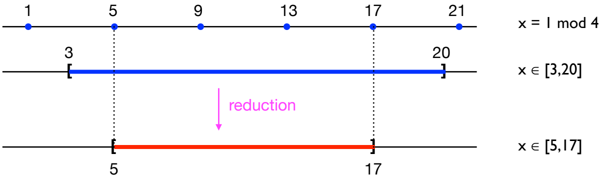
- $\alpha(R) = \mathtt{x} \mapsto \{\rho(\mathtt{x}) \mid \rho \in R\}$

Interval abstraction

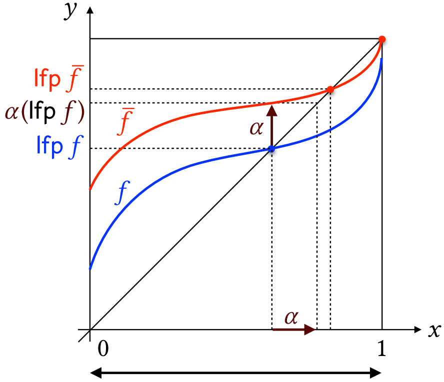
- We are left with sets of values
- For totally ordered sets, the interval abstraction records the minimum (or $-infty$) and maximum (or $+\infty$)
- $\alpha(V) = [\min V, \max V]$

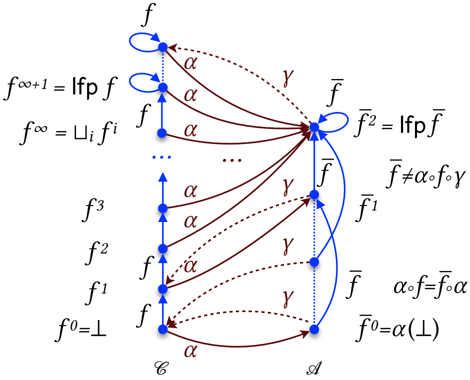
Reduced product

- Static analyzers use many abstractions
- The static analyzer can be refined by new abstractions
- The are also used to infer new properties and reduce the previous abstractions
- Example of reduction for cartesian congruence and interval analysis

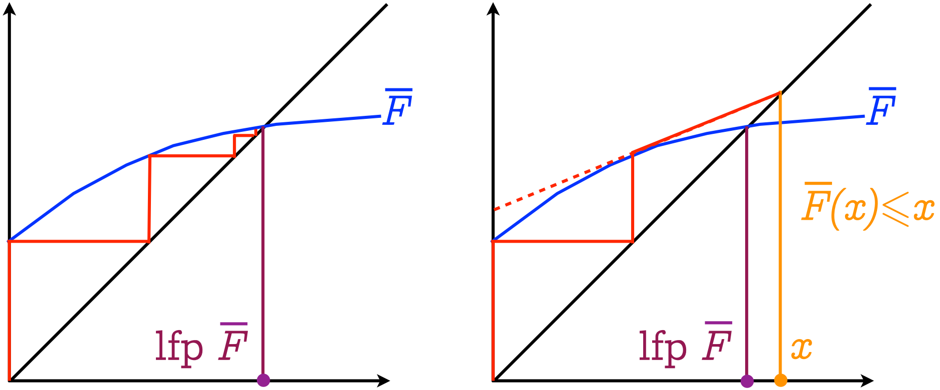


Fixpoint abstraction

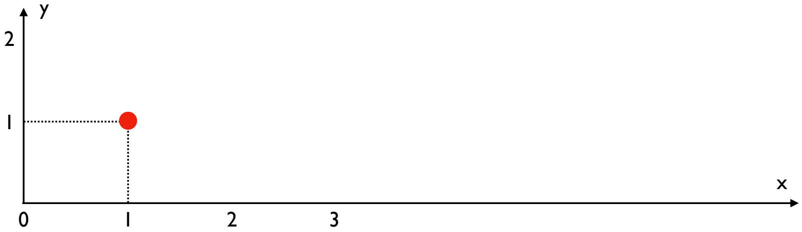




Fixpoint iteration acceleration (with widening)

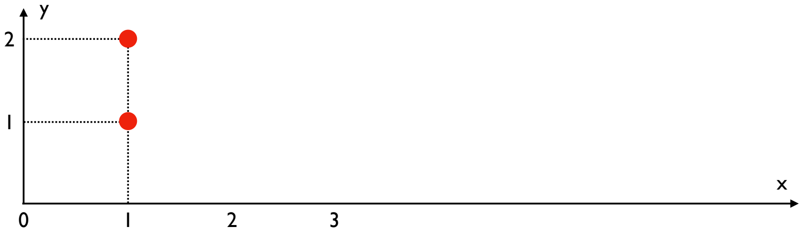


Example of widening (intervals)



$\alpha(x = 1 \wedge y = 1) = x \in [1, 1] \wedge y \in [1, 1]$

Example of widening (intervals)

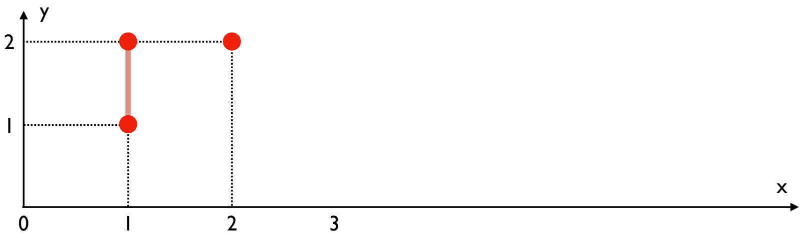


$$\alpha((x \in [1,1] \wedge y \in [1,1]) \vee (x = 1 \wedge y = 2))$$

Example of widening (intervals)

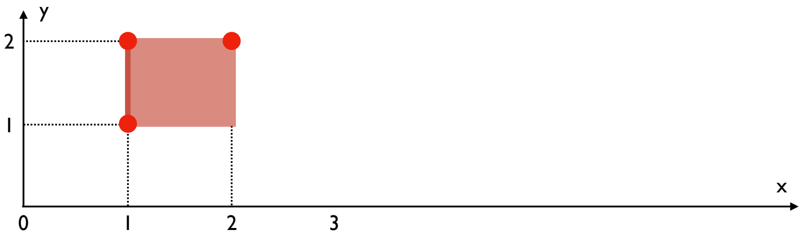
$x \in [1, 1] \wedge y \in [1, 2]$

Example of widening (intervals)



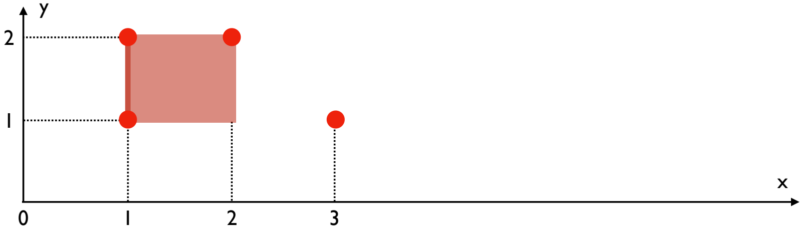
$\alpha((x \in [1,1] \wedge y \in [1,2]) \vee (x = 2 \wedge y = 2))$

Example of widening (intervals)



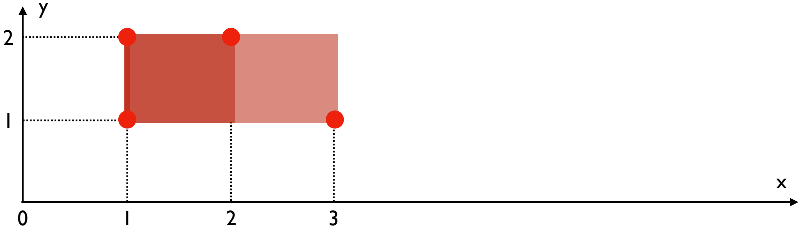
$x \in [1, 2] \wedge y \in [1, 2]$

Example of widening (intervals)



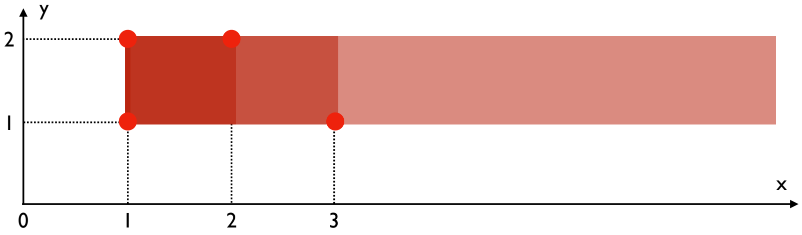
$\alpha((x \in [1,2] \wedge y \in [1,2]) \vee (x = 3 \wedge y = 2)$

Example of widening (intervals)



$(x \in [1, 3] \land y \in [1, 2])$

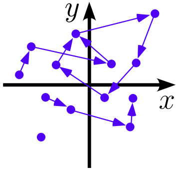
Example of widening (intervals)



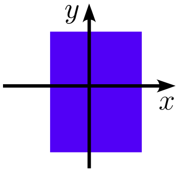
$$(x \in [1,2] \wedge y \in [1,2]) \text{ widening } (x \in [1,3] \wedge y \in [1,2]) = (x \in [1,+\infty] \wedge y \in [1,2])$$

Examples of static analyzes

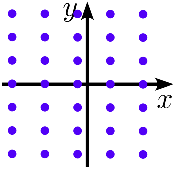
Numerical properties



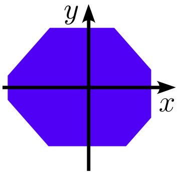
Collecting semantics:
partial traces



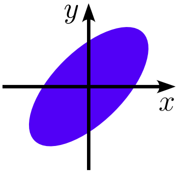
Intervals:
$\mathtt{x} \in [a, b]$



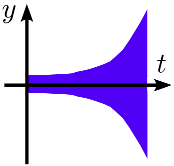
Simple congruences:
$\mathtt{x} \equiv a[b]$



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



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



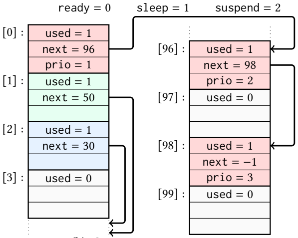
Exponentials:
$-a^{bt} \leqslant \mathtt{y}(t) \leqslant a^{bt}$

Symbolic properties

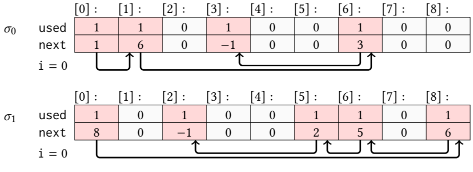
- Numerous abstractions to handle symbolic properties (arrays, pointers, memory allocation, etc.)
- example: process tables of an OS

```
struct {
  int used;
  int next;
  int prio;
} a[100];
int ready;
int sleep;
int suspend;
```

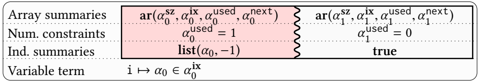
(a) Declaration



(b) Concrete memory state



(a) Concrete states



(b) Coalesced abstract state

Jiangchao Liu, Liqian Chen, Xavier Rival: Automatic Verification of Embedded System Code Manipulating Dynamic Structures Stored in Contiguous Regions. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 37(11): 2311-2322 (2018)

Static specification checking

- Examples of specifications: datalog, regular expressions to specify sequences of invariants
  - $\texttt{(?:x>=0)}^*$ states that the value of $\texttt{x}$ is always positive or zero during program execution.
  - $\texttt{(?:x>='x)}^*$ states that the value of $\texttt{x}$ is always greater than or equal to its initial value $\texttt{'x}$ during execution.
  - $(\neg\ell\texttt{:x>=0})^* \cdot \ell : \texttt{x==0} \cdot \texttt{(?:x<0)}^*$ states that
    - the value of $\texttt{x}$ should be positive or zero, and next
    - if program point ℓ is ever reached then $\texttt{x}$ should be 0, and next
    - if computations go on after program point ℓ then $\texttt{x}$ should be negative afterwards.
- In the literature: Fred Schneider's security monitors: monitor the actions of a program, checks the behavior of the program against a given safety specification (and initiate remedial actions)[1,2]

Patrick Cousot: Calculational design of a regular model checker by abstract interpretation. Theor. Comput. Sci. 869: 62-84 (2021)
Fred B. Schneider: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1): 30-50 (2000)

[1] use automata equivalent to regular expressions

[2] use actions instead of program labels.

Soundness

Soundness is difficult

- Languages have machine-dependent and undefined behaviors that must be taken into account by sound static analyzers
- Astrée for C: 3 types of errors
  1. the erroneous behavior is perfectly defined for the machine (e.g. integer overflow) $\rightarrow$ sound
  2. the erroneous behavior can be over approximated (e.g. integer division by zero is always an integer on some machines) $\rightarrow$ sound but imprecise
  3. the erroneous behavior is undefined $\rightarrow$
     - Astrée signals the error and goes on as if the error did not occur
     - the analysis is sound for executions up to the point where this error might occur, if ever, and inconclusive afterwards
     - allows for discovering other errors afterwards
- Static analysis is harder than verification

Patrick Cousot, Roberto Giacobazzi, Francesco Ranzato: Program Analysis Is Harder Than Verification: A Computability Perspective. CAV (2) 2018: 75-95

Examples of static analyzers

Andromeda

- Static analyzer for security of Web applications written in Java, .NET and JavaScript
- Developed by Marco Pistoia and his team at IBM Yorktown Heights
- Sound demand-driven abstract interpretation-based static dependency/taint analysis
- Precise and scalable
- Checks for cross-site scripting (XSS), SQL injection (SQLi), log forging, etc.

ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications

Omer Tripp[1], Marco Pistoia[2], Patrick Cousot[3], Radhia Cousot[4], and Salvatore Guarnieri[5]

Astrée (https://www.absint.com/astree)

Timing	Stack usage	Runtime errors	Rule checking	Compilation →

Astrée automatically proves the absence of runtime errors and invalid concurrent behavior in C/C++ applications. It is sound for floating-point computations, very fast, and exceptionally precise. The analyzer also checks for MISRA coding rules and supports qualification for ISO 26262, DO-178C level A, and other safety standards. Jenkins and Eclipse plugins are available.

Who uses Astrée?

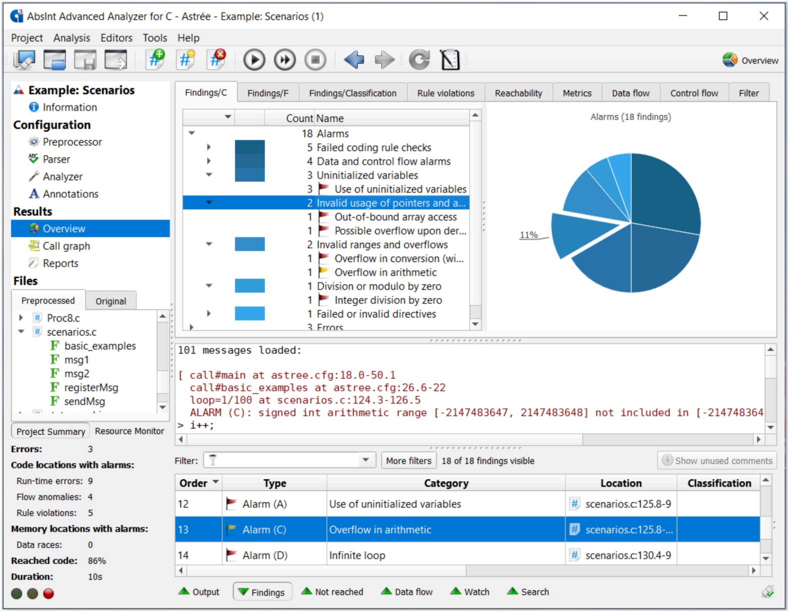
- » **Automotive**
- » Aviation
- » Power plants
- » Space flight
- » Ventilation

The global automotive supplier Helbako in Germany is using Astrée to guarantee that no runtime errors can occur in their electronic control software and to demonstrate MISRA compliance of the code.

HELBAKO

Bosch Automotive Steering replaced their legacy tools with Astrée and RuleChecker, resulting in significant savings thanks to faster analyses, higher accuracy, and optimized licensing and support costs.

BOSCH


AbsInt Advanced Analyzer for C - Astrée - Example: Scenarios (1)
Project Analysis Editors Tools Help
Overview
Example: Scenarios
Information
Configuration
Preprocessor
Parser
Analyzer
Annotations
Results
Overview
Call graph
Reports
Files
Preprocessed
Original
Proc8.c
scenarios.c
basic_examples
msg1
msg2
registerMsg
sendMsg
Project Summary
Resource Monitor
Errors: 3
Code locations with alarms:
Run-time errors: 9
Flow anomalies: 4
Rule violations: 5
Memory locations with alarms:
Data races: 0
Reached code: 86%
Duration: 10s
Findings/C
Findings/F
Findings/Classification
Rule violations
Reachability
Metrics
Data flow
Control flow
Filter
Count Name
18 Alarms
5 Failed coding rule checks
4 Data and control flow alarms
3 Uninitialized variables
3 Use of uninitialized variables
2 Invalid usage of pointers and a...
1 Out-of-bound array access
1 Possible overflow upon der...
2 Invalid ranges and overflows
1 Overflow in conversion (wi...
1 Overflow in arithmetic
1 Division or modulo by zero
1 Integer division by zero
1 Failed or invalid directives
3 Errors
Alarms (18 findings)
11%
101 messages loaded:
[call#main at astree.cfg:18.0-50.1
call#basic_examples at astree.cfg:26.6-22
loop=1/100 at scenarios.c:124.3-126.5
ALARM (C): signed int arithmetic range [-2147483647, 2147483648] not included in [-214748364
> i++;
Filter:
More filters
18 of 18 findings visible
Show unused comments
Order Type Category Location Classification
12 Alarm (A) Use of uninitialized variables scenarios.c:125.8-9
13 Alarm (C) Overflow in arithmetic scenarios.c:125.8-...
14 Alarm (D) Infinite loop scenarios.c:130.4-9
Output
Findings
Not reached
Data flow
Watch
Search

Which runtime properties are analyzed by Astrée?

Astrée statically analyzes whether the programming language is used correctly and whether there can be any runtime errors during any execution in any environment. This covers any use of C or C++ that, according to the corresponding language standard, has undefined behavior or violates hardware-specific aspects.

Additionally, Astrée reports invalid concurrent behavior, violations of user-specified programming guidelines, and various program properties relevant for functional safety.

Astrée detects any:

- division by zero,
- out-of-bounds array indexing,
- erroneous pointer manipulation and dereferencing (NULL, uninitialized and dangling pointers),
- integer and floating-point arithmetic overflow,
- read access to uninitialized variables,
- data races (read/write or write/write concurrent accesses by two threads to the same memory location without proper mutex locking),
- inconsistent locking (lock/unlock problems),
- invalid calls to operating system services (e.g. OSEK calls to `TerminateTask` on a task with unreleased resources),
- violation of optional user-defined assertions to prove additional runtime properties (similar to assert diagnostics),
- code it can prove to be unreachable under any circumstances.

https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8304.pdf

The NIST Software Assurance Metrics And Tool Evaluation project, or SAMATE for short, is dedicated to improving software assurance by developing methods for evaluating software tools, measuring their effectiveness, and identifying gaps in methods and techniques.

The SAMATE project recognizes the value and importance of sound static code analyzers. During the 6th Static Analysis Tool Exposition (SATE VI), the NIST team evaluated static analyzers with respect to the SATE VI Ockham Sound Analysis Criteria.

At least four tools were considered, only two of which satisfied the SATE VI criteria: Astrée and Frama-C.

Astrée was run on 28 sets of test cases from the Juliet 1.3 C test suite, containing a total of 18,954 buggy sites. All 18,954 were reported by Astrée.

https://frama-c.com

Soundness

Astrée was run on 28 sets of test cases from the Juliet 1.3 C test suite, containing a total of 18,954 buggy sites. All 18,954 were reported by Astrée.

These included test cases for buffer overflows/underflows, invalid pointer dereferences, integer overflows/underflows, divisions by zero, use of uninitialized variables, dead code, infinite loops, double free and use after free.

Additionally, Astrée discovered thousands of unintended defects in the Juliet 1.3 benchmark set.

"Alarms from **Astrée led us to find and fix thousands of mistakes** in what was intended as the Juliet known-bug list, manifest.xml.
Because **Astrée analyzes code very precisely** and we checked meticulously, details of modeling that otherwise would be inconsequential showed up and had to be resolved."

Choosing a static analyzer

Irresponsibility: avoid static analysis

- Programmers are never held responsible for their errors, even when the human and economic consequences are huge[3];
- Software engineers are guaranteed qualified immunity under the argument that verification is beyond best practice;
- If best practice would include the mandatory use of standards and qualified tools, programmers and their hierarchy could be held accountable at least for definite bugs automatically found be static analysis tools.

DOI:10.1145/2631185 Vinton G. Cerf

Responsible Programming

[3] e.g. 2009–11 Toyota vehicle recalls, Boeing 737 MAX groundings.

Academic versus industry

Benchmarking Software Model Checkers on Automotive Code

Lukas Westhofen[1], Philipp Berger[2], and Joost-Pieter Katoen[2]

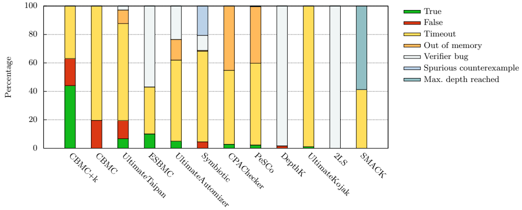
[1] OFFIS e.V., Oldenburg, Germany
lukas.westhofen@offis.de
[2] RWTH Aachen University, Aachen, Germany
{berger, katoen}@cs.rwth-aachen.de

Metric	DSR	ECC
Complexity		
Source lines of code	1,354	2,517
Cyclomatic complexity	213	268

Requirement Characteristics.

Invariant properties are assertions that are supposed to hold for all reachable states. Bounded-response properties request that a certain assertion holds within a given number of computational steps whenever a given, second assertion holds.

Coverage. Fig. 2 shows the verification results of running the open-source verifiers on the two case studies, omitting the results of the witness validation.



Lukas Westhofen, Philipp Berger, Joost-Pieter Katoen: Benchmarking Software Model Checkers on Automotive Code. NFM 2020: 133-150

Commerce is not science

SYNOPSYS®

WHITE PAPER

Coverity: Risk Mitigation for DO-178C

Gordon M. Uchenick, Lead Aerospace/Defense Sales Engineer

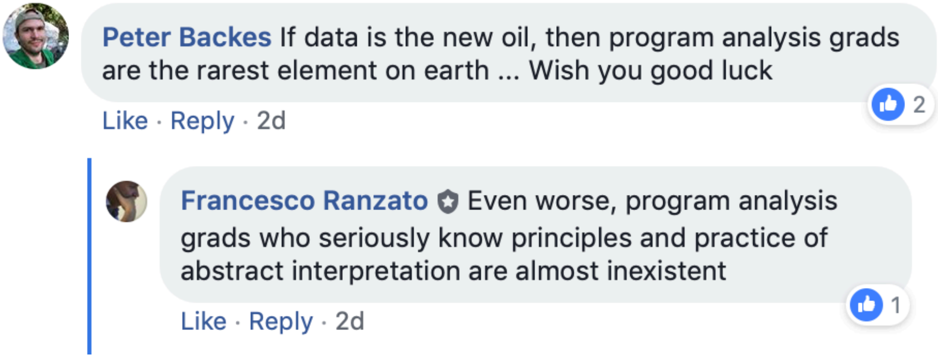
...

Don'ts

- Don't overestimate the limited value of standard test suites such as Juliet.†† These suites often exercise language features that are not appropriate for safety-critical code. Historically, the overlap between findings of different tools that were run over the same Juliet test suite has been surprisingly small.

†† Juliet Test Suites are available at https://samate.nist.gov/SRD/testsuite.php.

Competence is very rare



Some hot topics in abstract interpretation

- Blockchain

  Víctor Pérez, Maximiliano Klemen, Pedro López-García, José Francisco Morales, Manuel V. Hermenegildo: *Cost Analysis of Smart Contracts Via Parametric Resource Analysis*. SAS 2020: 7-31

- Fairness in neural networks

  Caterina Urban, Antoine Miné: *A Review of Formal Methods applied to Machine Learning*. CoRR abs/2104.02466 (2021)

  Caterina Urban, Maria Christakis, Valentin Wüstholz, Fuyuan Zhang: *Perfectly parallel fairness certification of neural networks.* Proc. ACM Program. Lang. 4(OOPSLA): 185:1-185:30 (2020)

  Caterina Urban: *Static Analysis of Data Science Software.* SAS 2019: 17-23

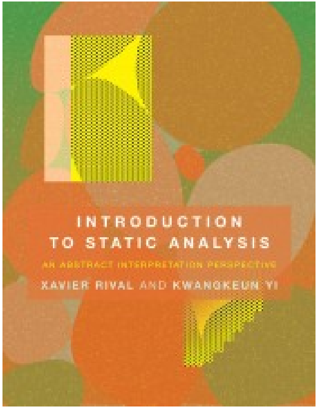
- Quantum computing

  Nengkun Yu and Jens Palsberg: *Quantum Abstract Interpretation.* PLDI 2021.

References

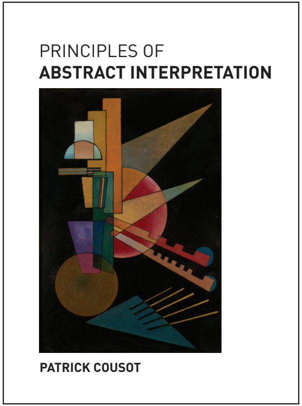
For engineers

Introduction to Static Analysis
Xavier Rival and Kwangkeun Yi
MIT Press, 2020



For researchers

Principles of Abstract Interpretation
Patrick Cousot
MIT Press, September 21st, 2021



The End, Thank you