

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 :)

```
unsigned int payload = 18; /* Sequence number + random bytes */
unsigned int padding = 16; /* Use minimum padding */

/* Check if padding is too long, payload and padding
 * must not exceed 2^14 - 3 = 16381 bytes in total.
 */
OPENSSL_assert(payload + padding <= 16381);

/* Create HeartBeat message, we just use a sequence number
 * as payload to distinguish different messages and add
 * some random stuff.
 * - Message Type, 1 byte
 * - Payload Length, 2 bytes (unsigned int)
 * - Payload, the sequence number (2 bytes uint)
 * - Payload, random bytes (16 bytes uint)
 * - Padding
 */

buf = OPENSSL_malloc(1 + 2 + payload + padding);
p = buf;
/* Message Type */
*p++ = TLS1_HB_REQUEST;
/* Payload length (16 bytes here) */
s2n(payload, p);
/* Sequence number */
s2n(s->tlsext hb_seq, p);
/* 16 random bytes */
RAND_pseudo_bytes(p, 16);
p += 16;
/* Random padding */
RAND_pseudo_bytes(p, padding);

ret = dtls1_write_bytes(s, TLS1_HB_HEARTBEAT, buf, 3 + payload + padding);
```

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 :)

```
unsigned int payload = 18; /* Sequence number + random bytes */
unsigned int padding = 16; /* Use minimum padding */

/* check if padding is too long, payload and padding
 * must not exceed 2^14 - 3 = 16381 bytes in total.
 */
OPENSSL_assert(payload + padding <= 16381);

/* Create HeartBeat message, we just use a sequence number
 * as payload to distinguish different messages and add
 * some random stuff.
 * - Message Type, 1 byte
 * - Payload Length, 2 bytes (unsigned int)
 * - Payload, the sequence number (2 bytes uint)
 * - Payload, random bytes (16 bytes uint)
 * - Padding
 */

buf = OPENSSL_malloc(1 + 2 + payload + padding);
p = buf;
/* Message Type */
*p++ = TLS1_HB_REQUEST;
/* Payload length (16 bytes here) */
s2n(payload, p);
/* Sequence number */
s2n(s->txseq, p);
/* 16 random bytes */
RAND_pseudo_bytes(p, 16);
p += 16;
/* Random padding */
RAND_pseudo_bytes(p, padding);

ret = dtls1_write_bytes(s, TLS1_HB_HEARTBEAT, buf, 1 + payload + padding);
```

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* possibilities

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 protocols verification
Dataflow analysis
Obfuscation
Denotational semantics
Theories combination
Code contracts
Quantum entanglement detection
Steganography

Systems biology analysis
Model checking
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Bisimulation
SMT solvers
Tautology testers

Operational semantics
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Quantum entanglement detection

The *Theory* of Abstract Interpretation: Unifies Formal Methods

The need for a unified account of formal methods

WCET
Axiomatic semantics
Confidentiality analysis
Program synthesis
Grammar analysis
Statistical model-checking
Invariance proof
Probabilistic verification
Parsing

Security protocols verification
Dataflow analysis
Obfuscation
Denotational semantics
Theories combination
Code contracts
Quantum entanglement detection
Steganography

Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Type theory

Systems biology analysis
Model checking
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Bisimulation
SMT solvers
Tautology testers

Database query
Operational semantics
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

Underlying unity of formal methods

Abstract interpretation

WCET
Axiomatic semantics
Confidentiality analysis
Program synthesis
Grammar analysis
Statistical model-checking
Invariance proof
Probabilistic verification
Parsing

Security protocols verification
Dataflow analysis
Partial evaluation
Effect systems
Trace semantics
Symbolic execution
Quantum entanglement detection
Type theory

Obfuscation
Denotational semantics
Theories combination
Code contracts
Steganography

Systems biology analysis
Model checking
Dependence analysis
CEGAR
Program transformation
Interpolants
Integrity analysis
Bisimulation
SMT solvers
Tautology testers

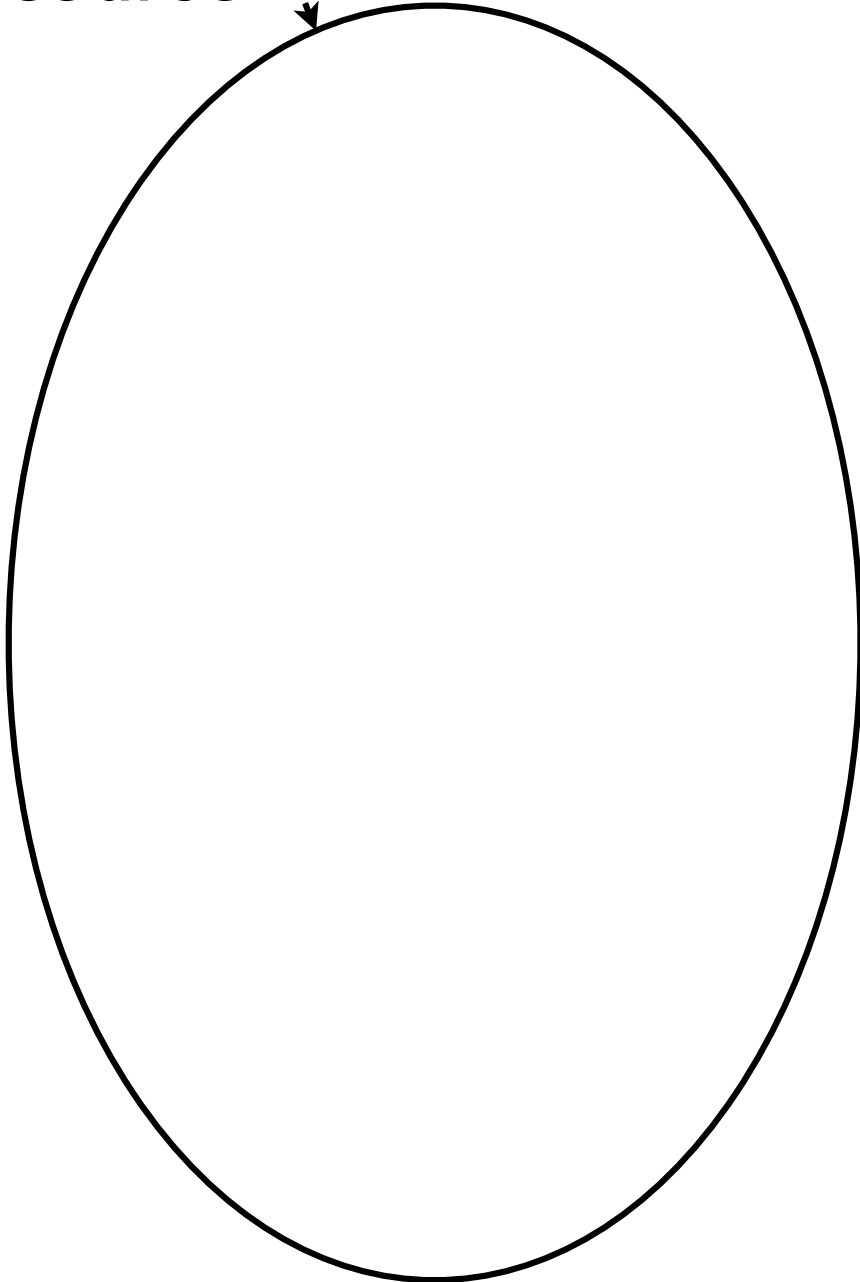
Database query
Abstraction refinement
Type inference
Separation logic
Termination proof
Shape analysis
Malware detection
Code refactoring

Operational semantics

Principle of Abstract Interpretation

Concrete
universe of
discourse

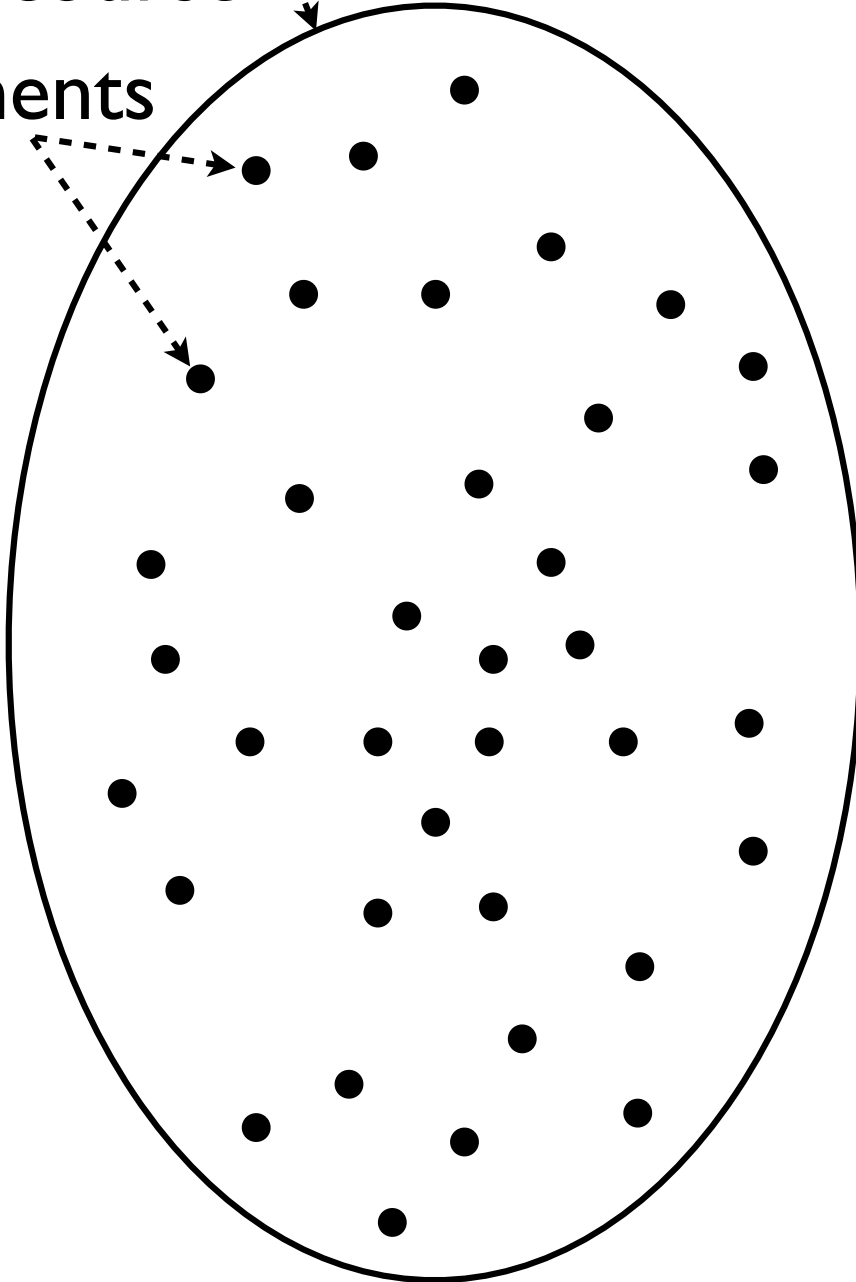
What is abstraction in AI?



What is abstraction in AI?

Concrete universe of discourse

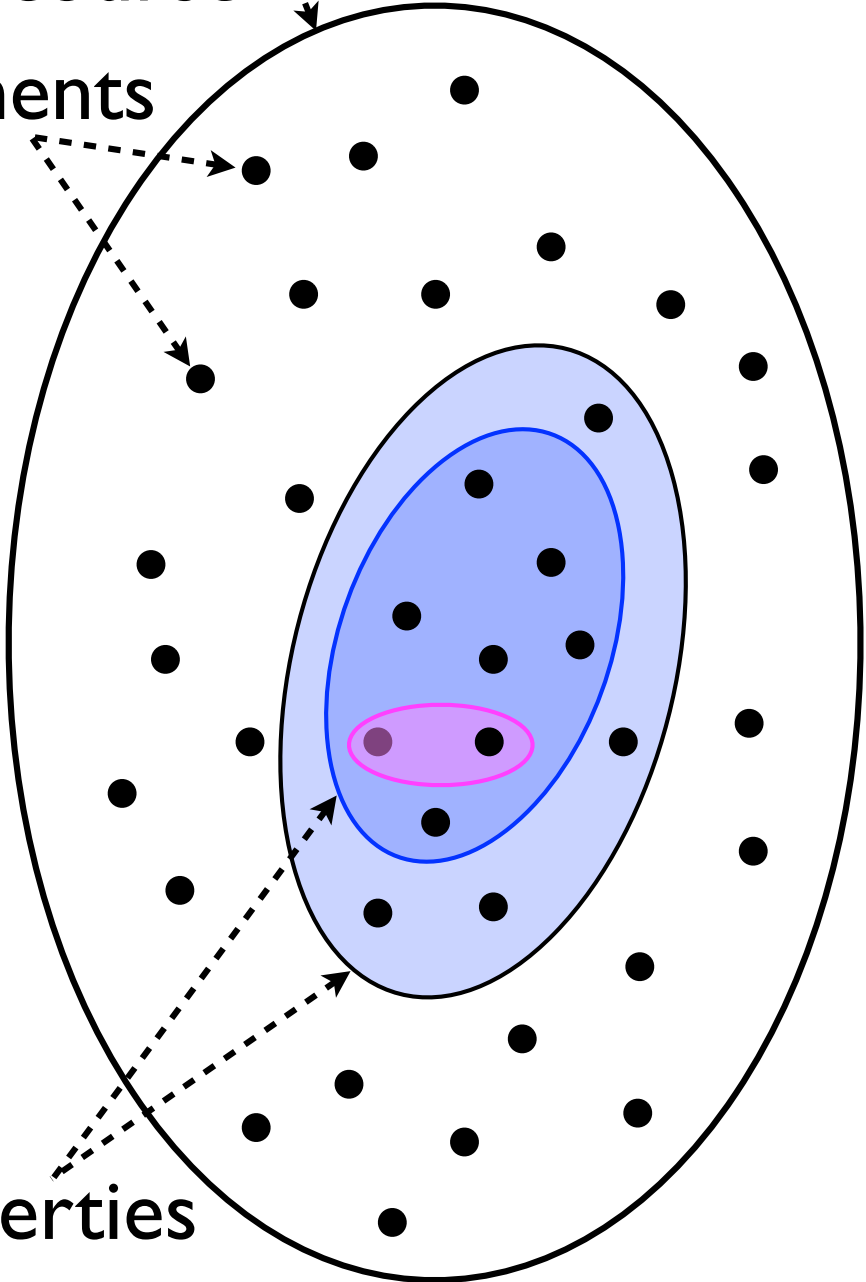
Elements



What is abstraction in AI?

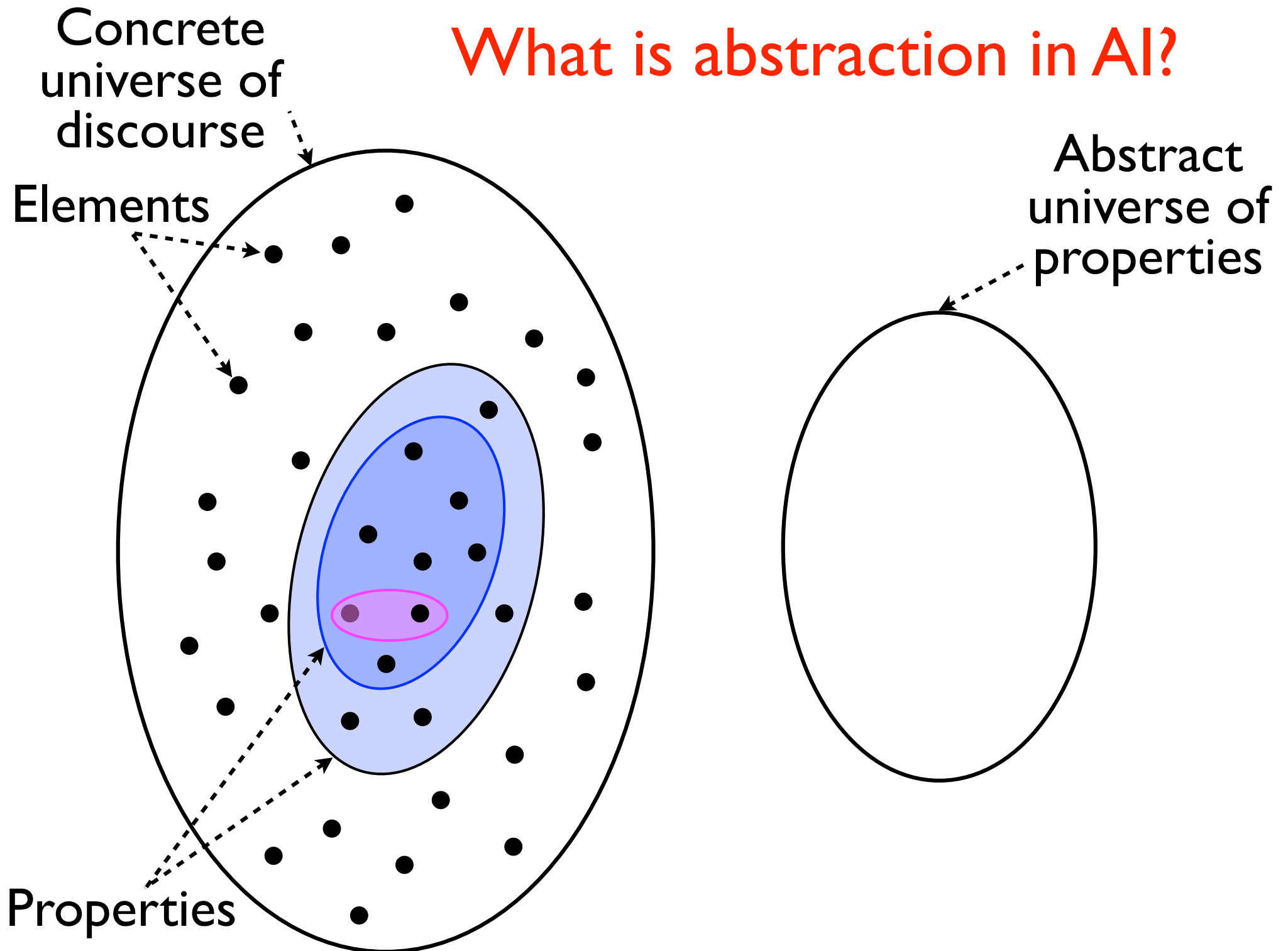
Concrete universe of discourse

Elements

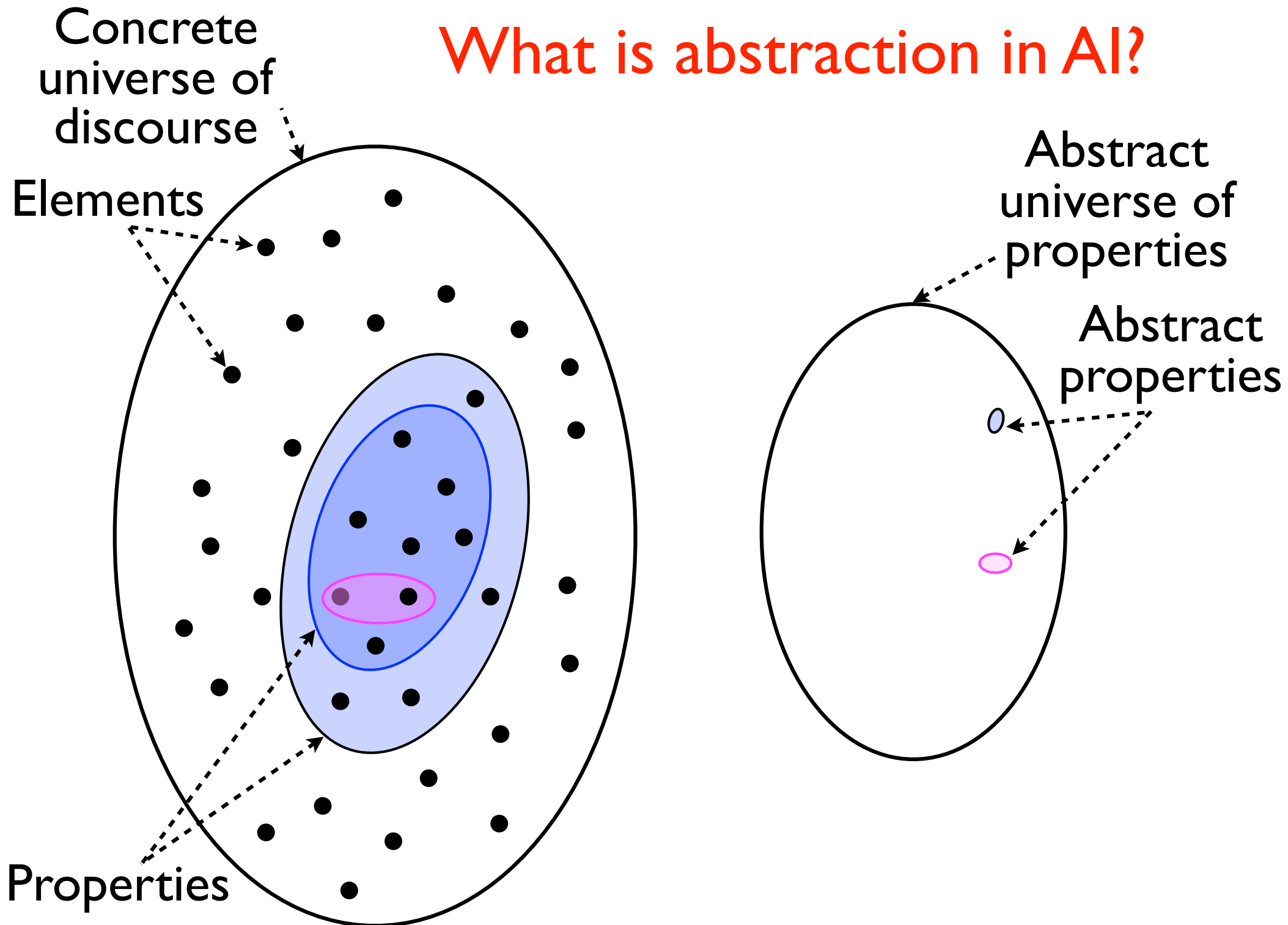


Properties

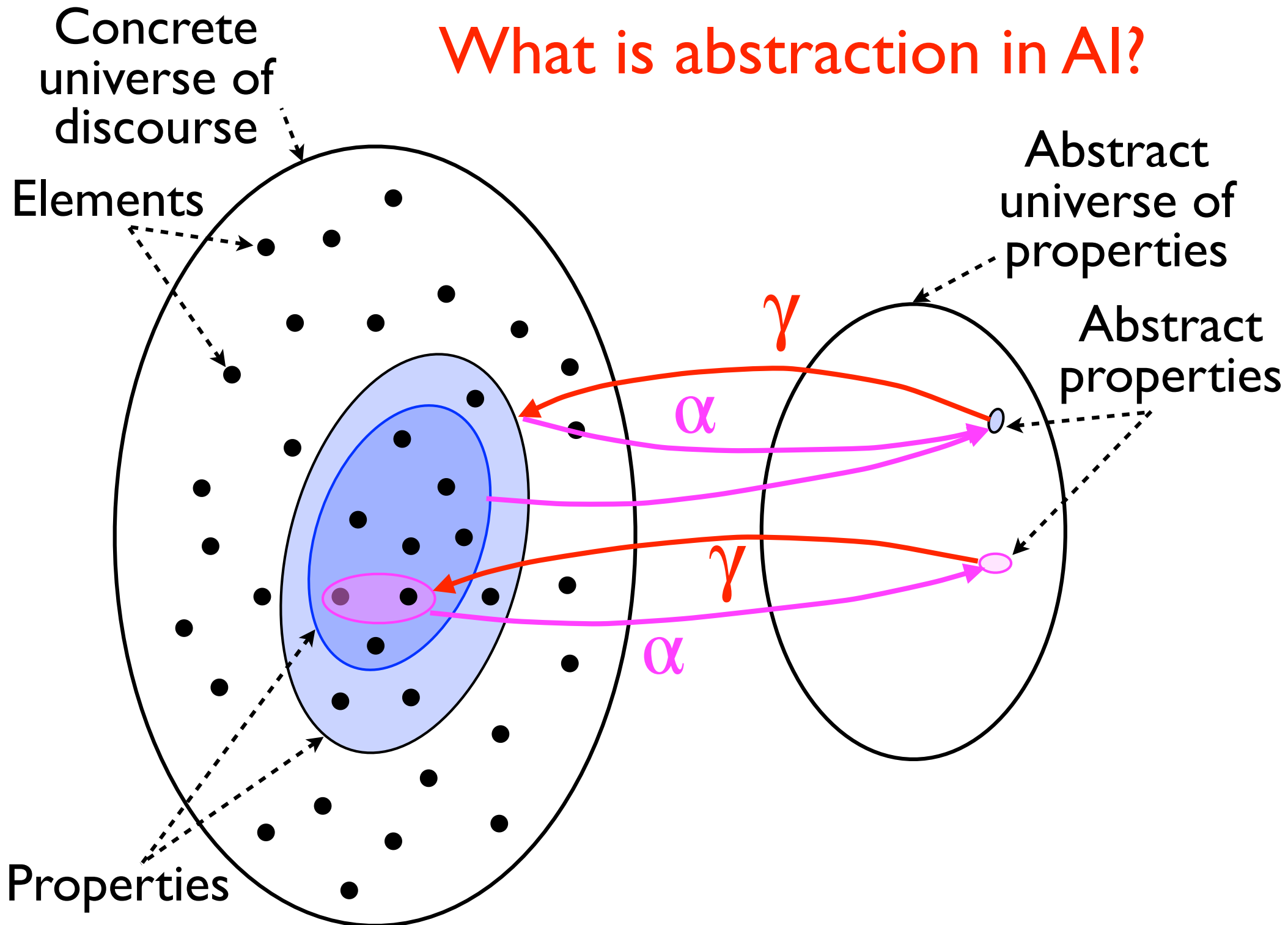
What is abstraction in AI?



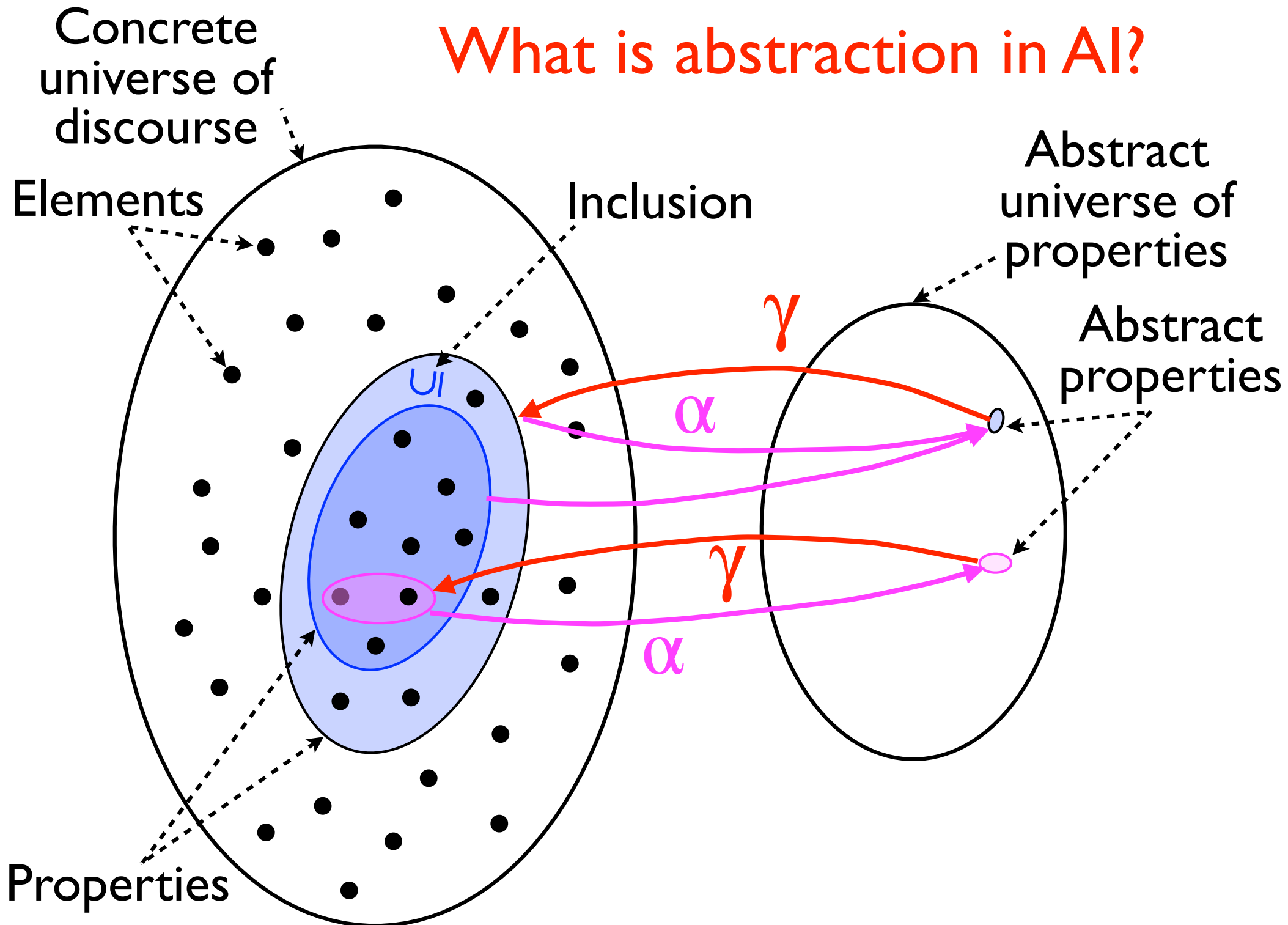
What is abstraction in AI?



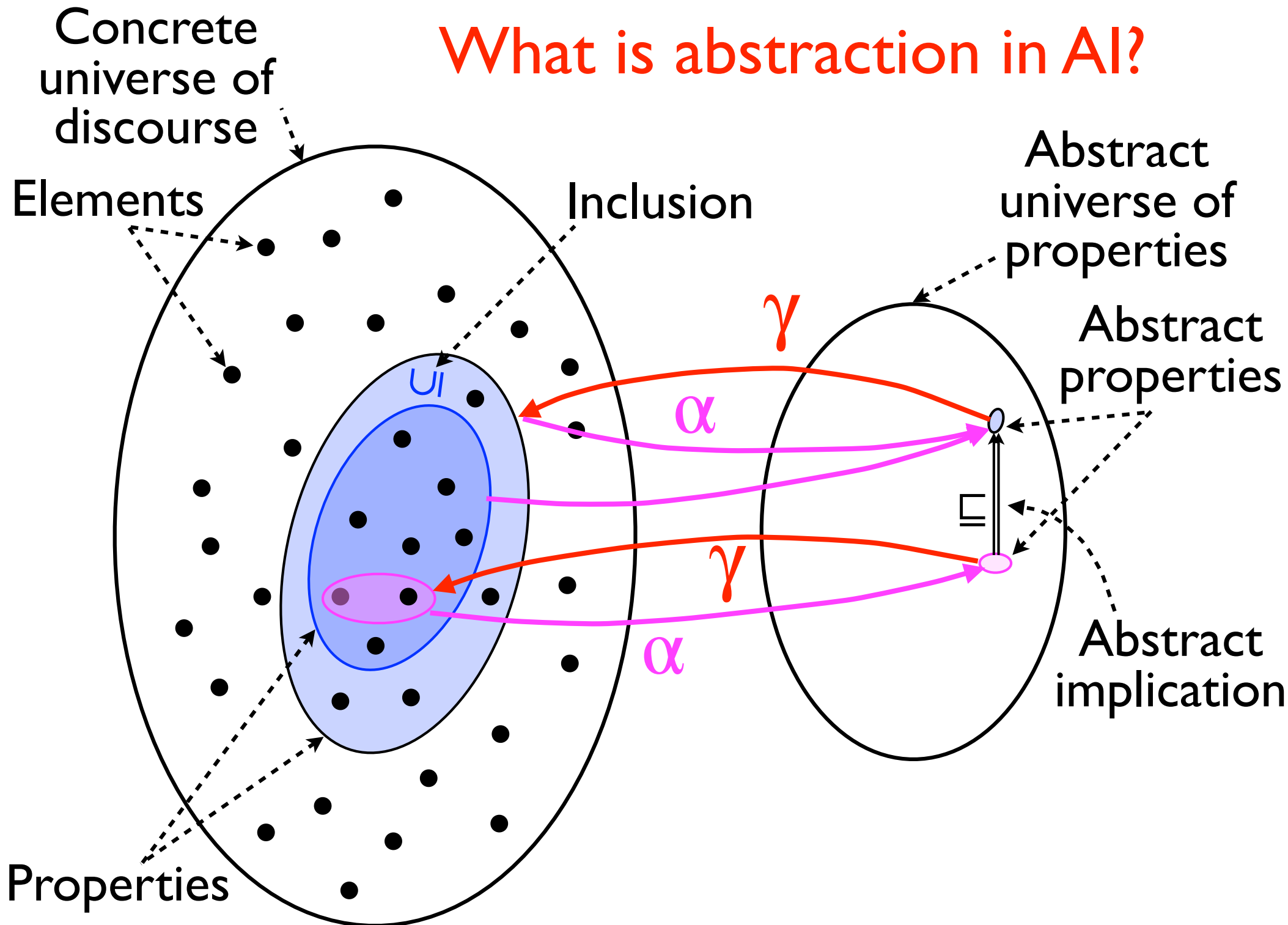
What is abstraction in AI?



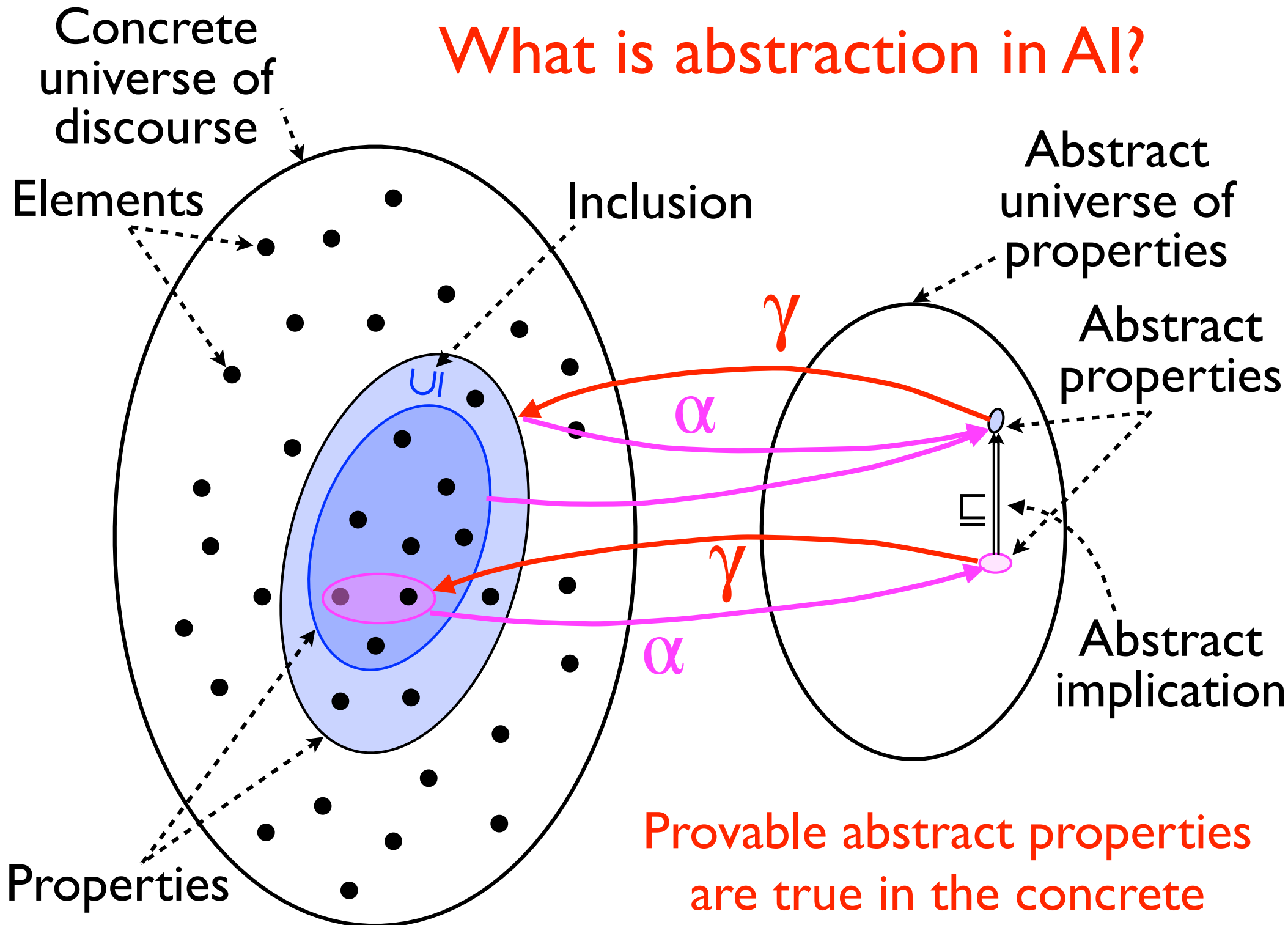
What is abstraction in AI?



What is abstraction in AI?



What is abstraction in AI?



Provable abstract properties
are true in the concrete

Abstract interpretation: example

Theory:

Galois Connections We recall from [11] that a Galois connection $\langle C, \preceq \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \sqsubseteq \rangle$ 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 $\forall x \in C : \forall y \in A : \alpha(x) \sqsubseteq y \iff x \preceq \gamma(y)$. We write $\langle C, \preceq \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \sqsubseteq \rangle$ 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 $\langle C, \preceq \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \sqsubseteq \rangle$.

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

$$P \implies P' \triangleq \forall \vec{v} \in \vec{V}[\vec{v}] : P(\vec{v}) \implies P'(\vec{v}).$$

For example $\lambda x \bullet x = 0 \implies \lambda x \bullet x \geq 0$. Predicates with partial order \implies form a complete Boolean lattice:

$$\langle \mathcal{P}[\vec{v}], \implies, \text{false}, \text{true}, \dot{\vee}, \dot{\wedge}, \dot{\neg} \rangle$$

where false is the infimum, 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 \implies on the set $\mathcal{P}[\vec{v}]$.

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

$$\gamma_1 \in \langle A[\vec{v}], \sqsubseteq \rangle \rightarrow \langle \mathcal{P}[\vec{v}], \implies \rangle$$

is increasing (i.e., $\bar{P} \sqsubseteq \bar{P}'$ implies $\gamma_1(\bar{P}) \implies \gamma_1(\bar{P}')$).

Applications:

RefactorContract($\bar{P}_S, S, \bar{p}, \bar{g}, \bar{Q}_S$) {

use $\langle A[\bar{p}], \sqcap, \Delta_1 \rangle$ // precondition abstract domain
 $\langle B[\bar{p}, \bar{p}], \sqcap, \Delta_2 \rangle$ // postcondition abstract domain
 post // forward analyser with widening/narrowing
 pre // backward analyser with widening/narrowing

// abstract projection on potentially used variables \bar{p}
 $\langle \bar{P}_S^V, \bar{Q}_S^V \rangle = \langle \downarrow_{\bar{p}\bar{g}}(\bar{P}_S), \downarrow_{\bar{p}\bar{g}}(\bar{Q}_S) \rangle$;

// infer a correct safety abstract contract

Let \bar{P}_m be the abstract safety pre-condition for S computed by the static analysis [18];

$\bar{Q}_m = \text{post}[\mathcal{S}[\bar{p}]]\bar{P}_m$; // forward abstract static analysis
 // $\{ \bar{P}_m \} \mathcal{S}[\bar{p}\bar{g}] \{ \bar{Q}_m \}$ holds

$$\langle \bar{P}_R, \bar{Q}_R \rangle = \langle \bar{P}_S^V, \bar{Q}_S^V \rangle$$

do

// compute $\langle X, Y \rangle = \bar{F}_R[\mathcal{S}](\langle \bar{P}_R, \bar{Q}_R \rangle)$

$X = \bar{P}_m \sqcap \bar{P}_R \sqcap \text{pre}[\mathcal{S}[\bar{p}]]\bar{Q}_R$; // backward analysis

$Y = \bar{Q}_m \sqcap \bar{Q}_R \sqcap \text{post}[\mathcal{S}[\bar{p}]]\bar{P}_R$; // forward analysis

$\langle \bar{P}_R, \bar{Q}_R \rangle = \langle \bar{P}_R \Delta_1 X, \bar{Q}_R \Delta_2 Y \rangle$; // narrowing

while $\langle \bar{P}_R, \bar{Q}_R \rangle \neq \langle X, Y \rangle$;

// $\text{gfp}_{\langle \bar{P}_S^V, \bar{Q}_S^V \rangle} \bar{F}_R[\mathcal{S}] \sqsubseteq \langle \bar{P}_R, \bar{Q}_R \rangle \sqsubseteq \langle \bar{P}_S^V, \bar{Q}_S^V \rangle$ holds

return $\langle \bar{P}_R, \bar{Q}_R \rangle$; // (a) validity & (b) safety hold

}

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

Practice:

The screenshot shows an IDE with a code editor. The code is for a `Decrement` method. A dialog box titled "Extract Method" is open, with the option "Extract method with Contracts" selected. The code in the editor is as follows:

```

public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);

    while (x != 0) x--;
}

```

The dialog box shows the selected code block and the option to extract it with contracts. The resulting code in the editor is:

```

...
public int Decrement(int x)
{
    Contract.Requires(x >= 5);
    Contract.Ensures(Contract.Result<int>() >= 0);
    x = NewMethod(x);
    return x;
}
int NewMethod(int x)
{
    Contract.Requires(0 <= x);
    Contract.Ensures(Contract.Result<System.Int32>() == 0);

    while (x != 0) x--;
    return x;
}
...

```

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

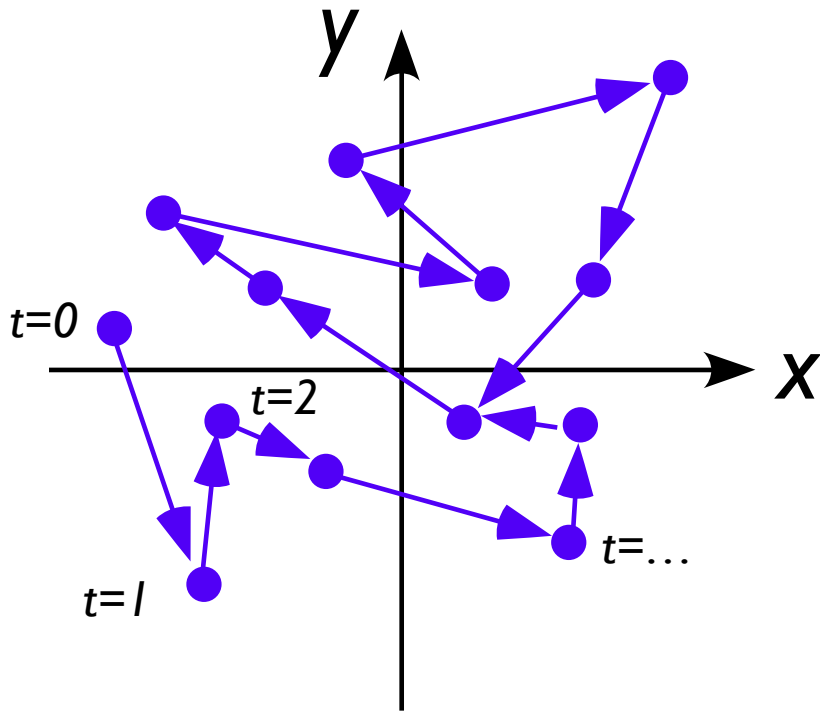
A very informal introduction to abstract interpretation

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

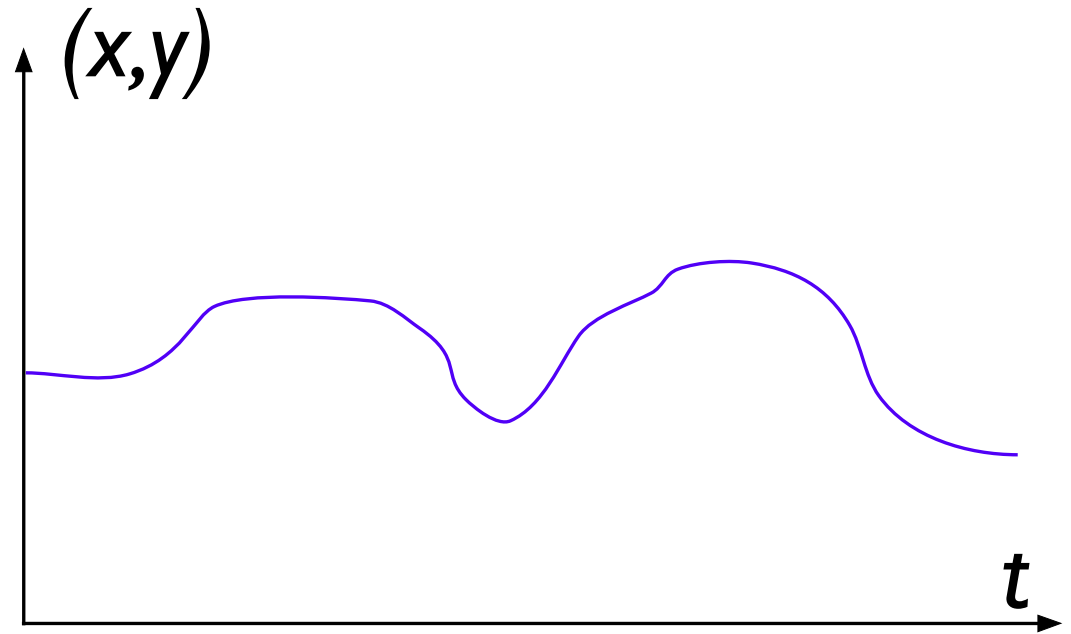
Patrick Cousot, Radhia Cousot: Systematic Design of Program Analysis Frameworks. POPL 1979: 269-282

I) Define the programming language semantics

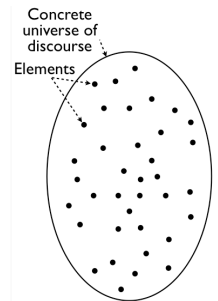
Formalize the concrete **executions** of programs (e.g. transition system)



Trajectory
in state space

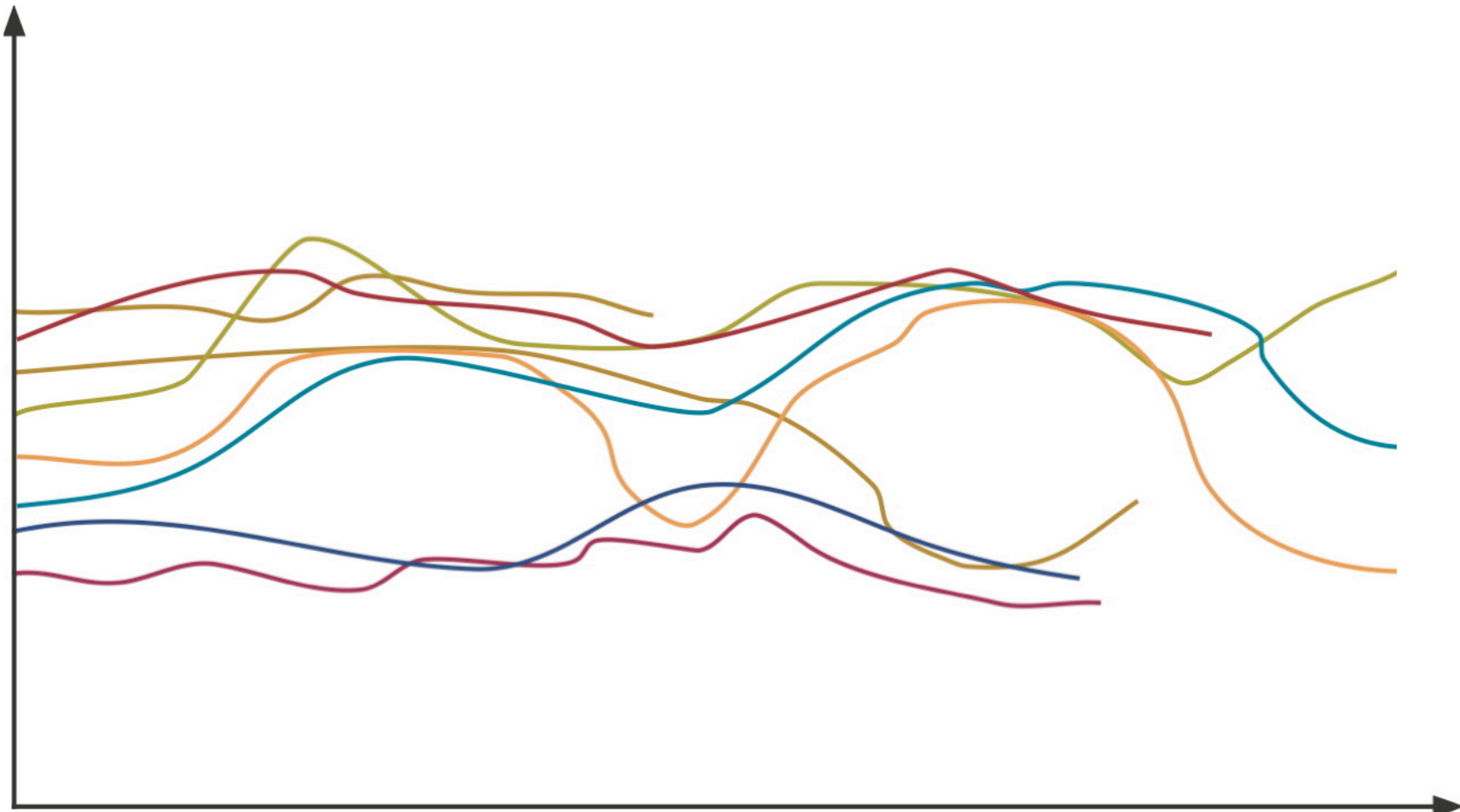


Space/time trajectory

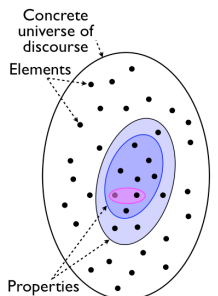


II) Define the program properties of interest

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

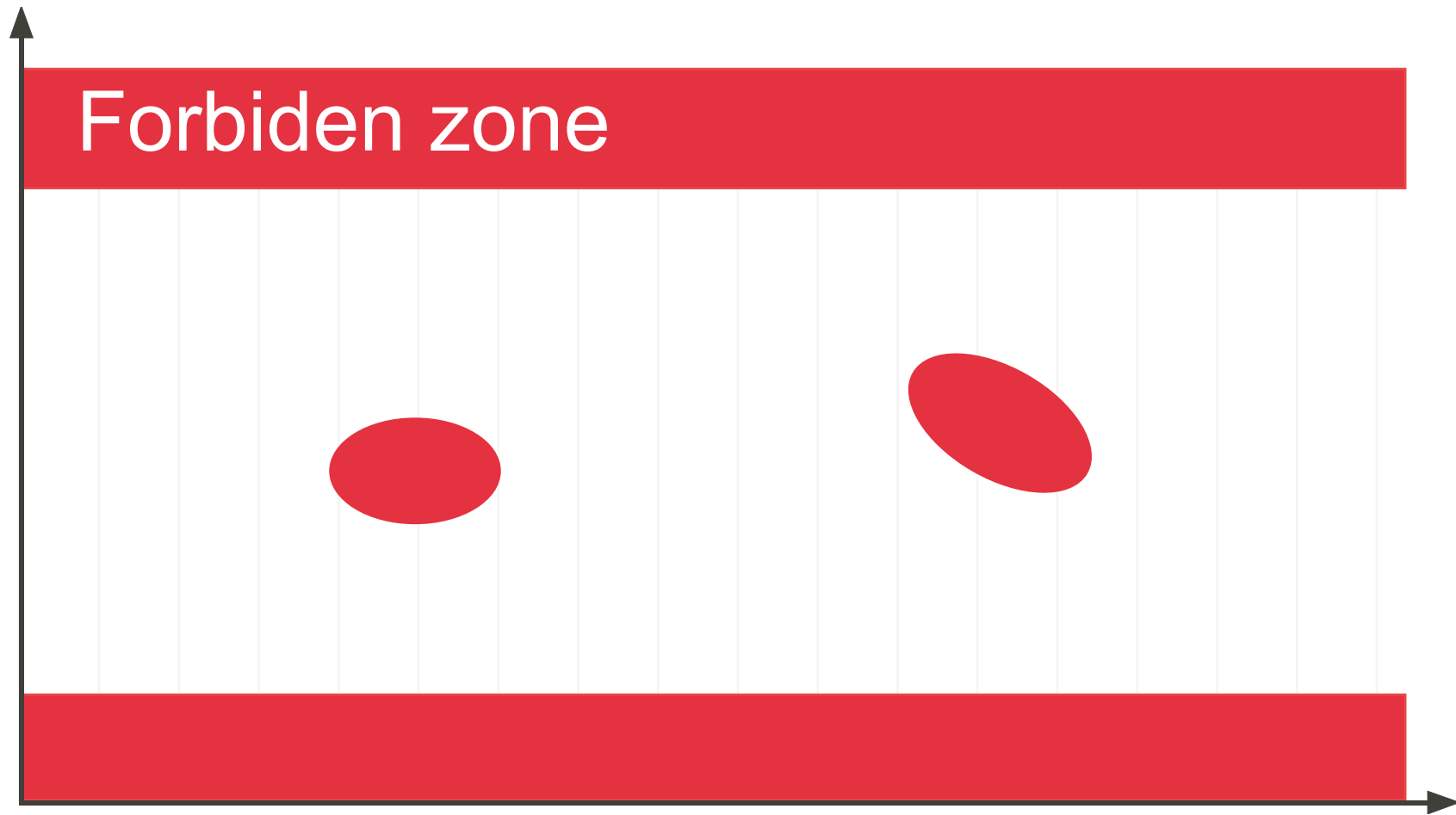


We are interested in the set of possible trajectories



III) Define which specification must be checked

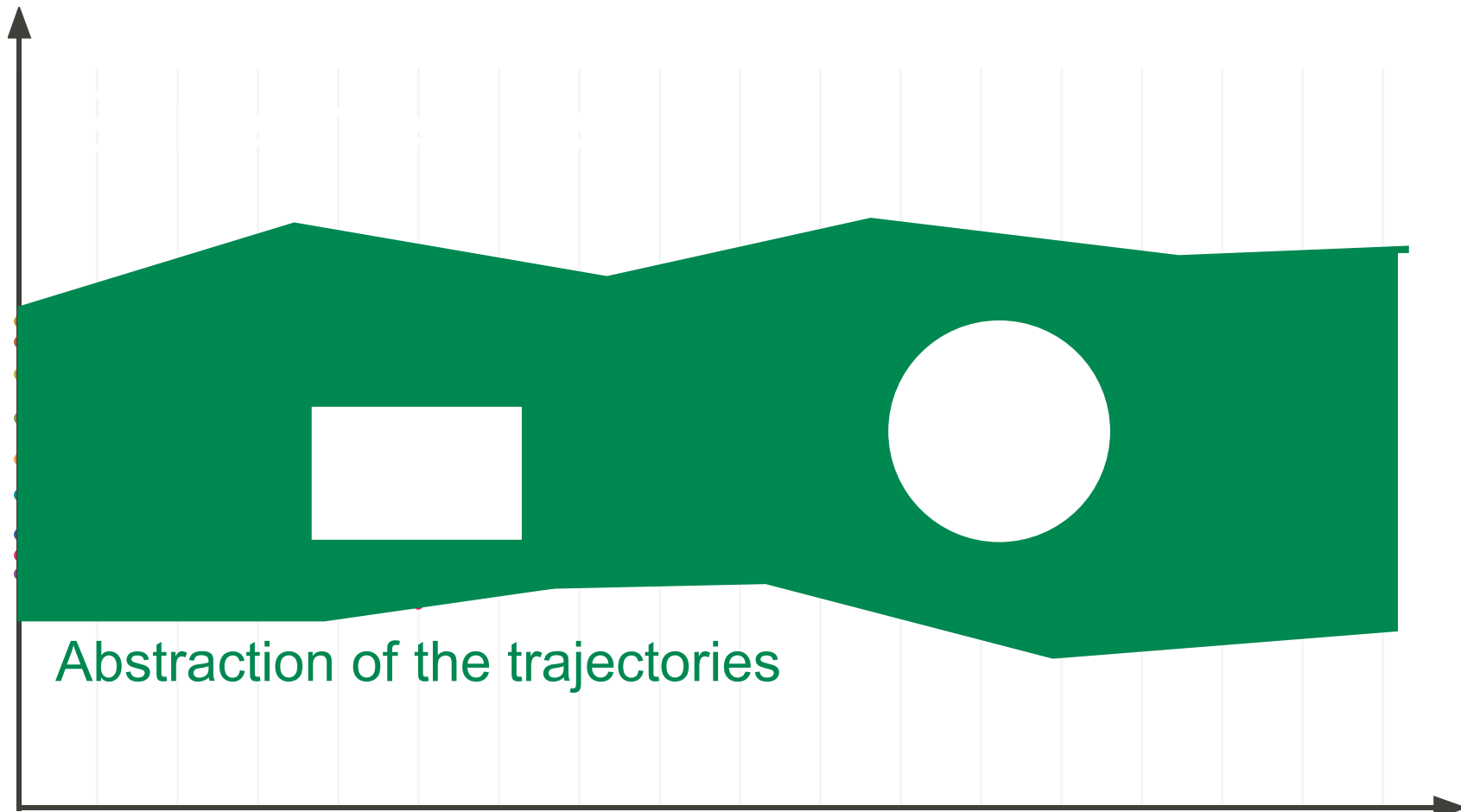
Formalize what you are interested to **prove** about program behaviors



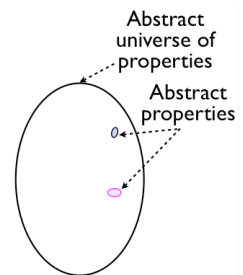
No trajectory should hit the forbidden zone

IV) Choose the appropriate abstraction

Abstract away all information on program behaviors irrelevant to the proof

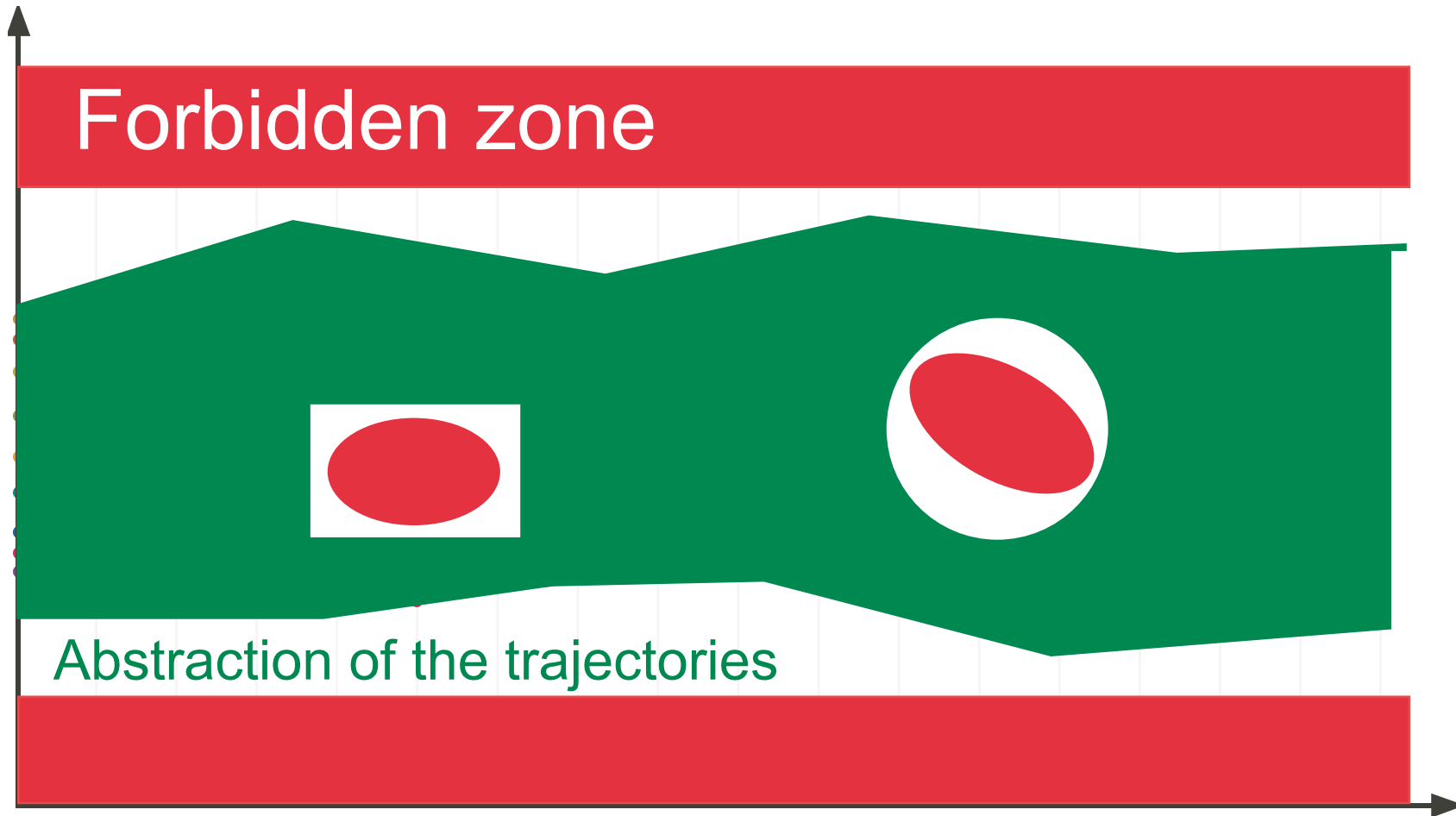


Abstraction by geometric forms (rectangles, polyhedra, ellipsoids, abstraction by parts, etc)



V) Mechanically verify in the abstract

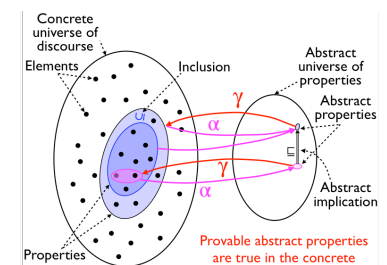
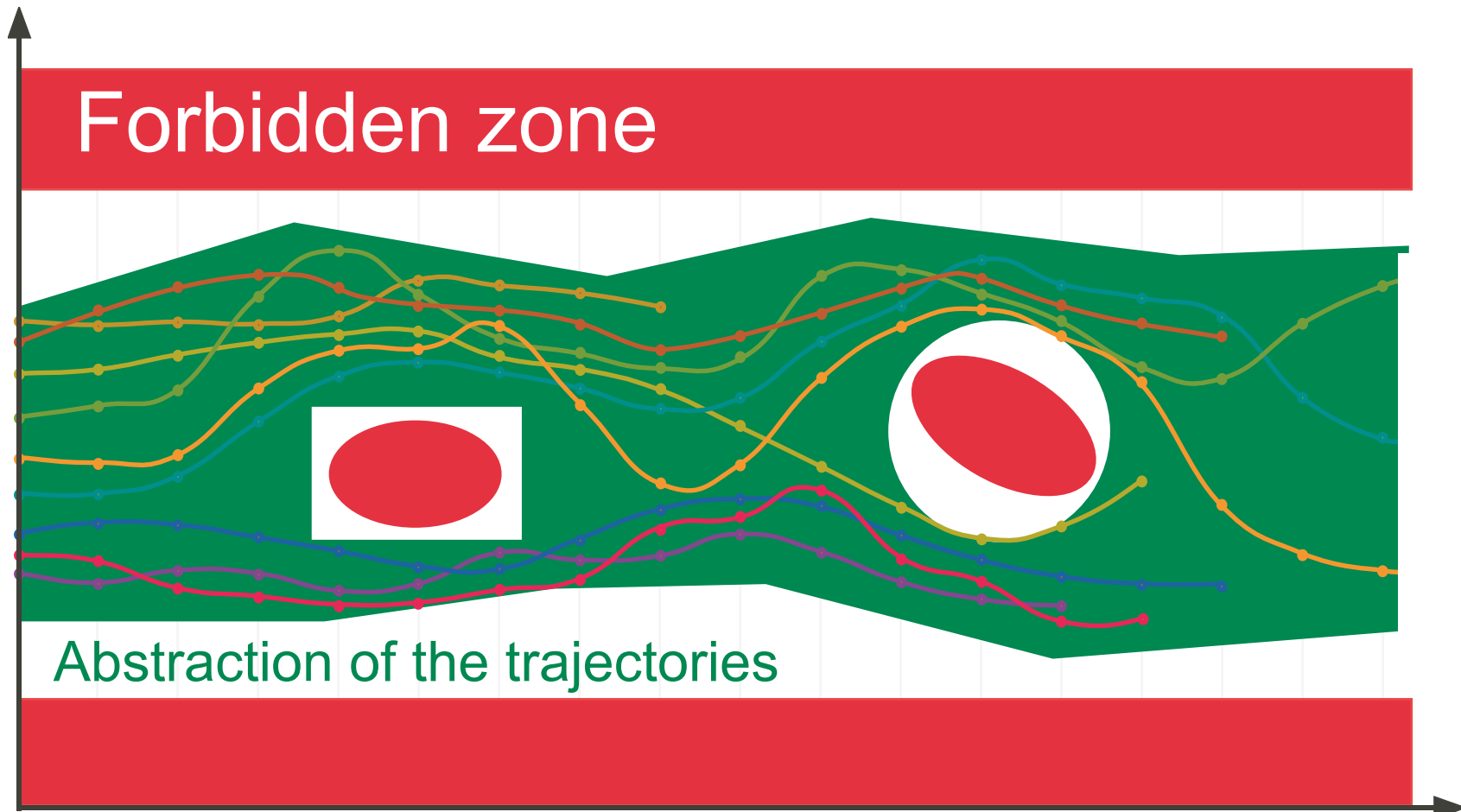
The proof is fully *automatic*



Provable abstract properties
are true in the concrete

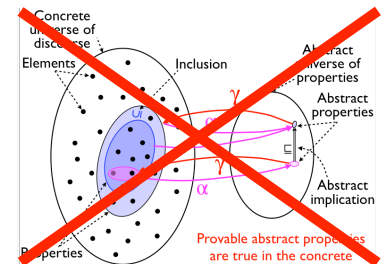
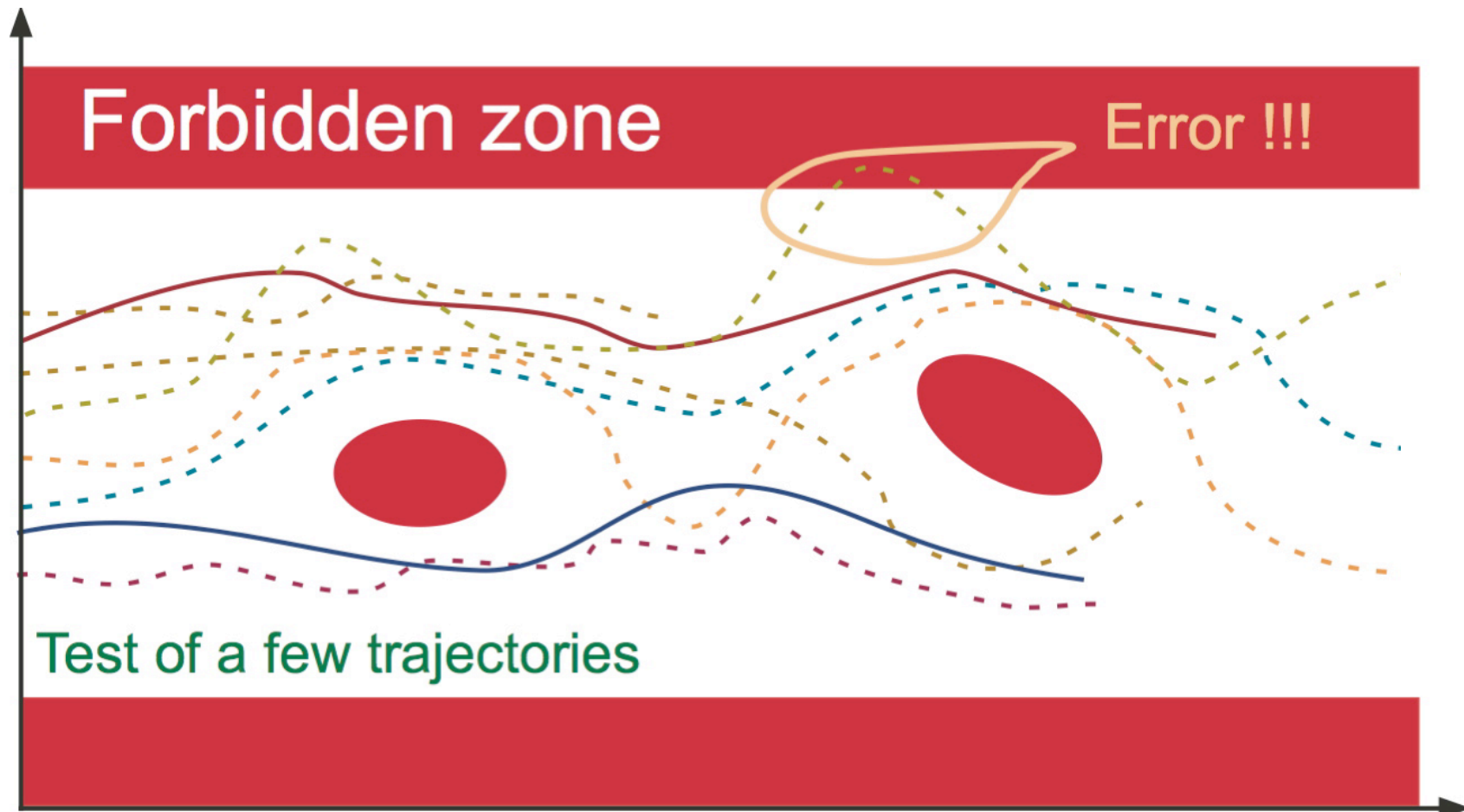
Soundness of the abstract verification

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



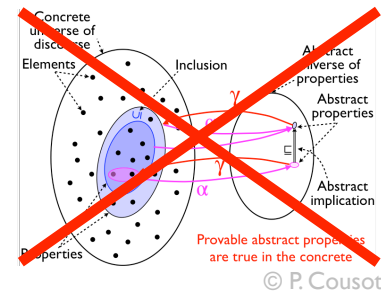
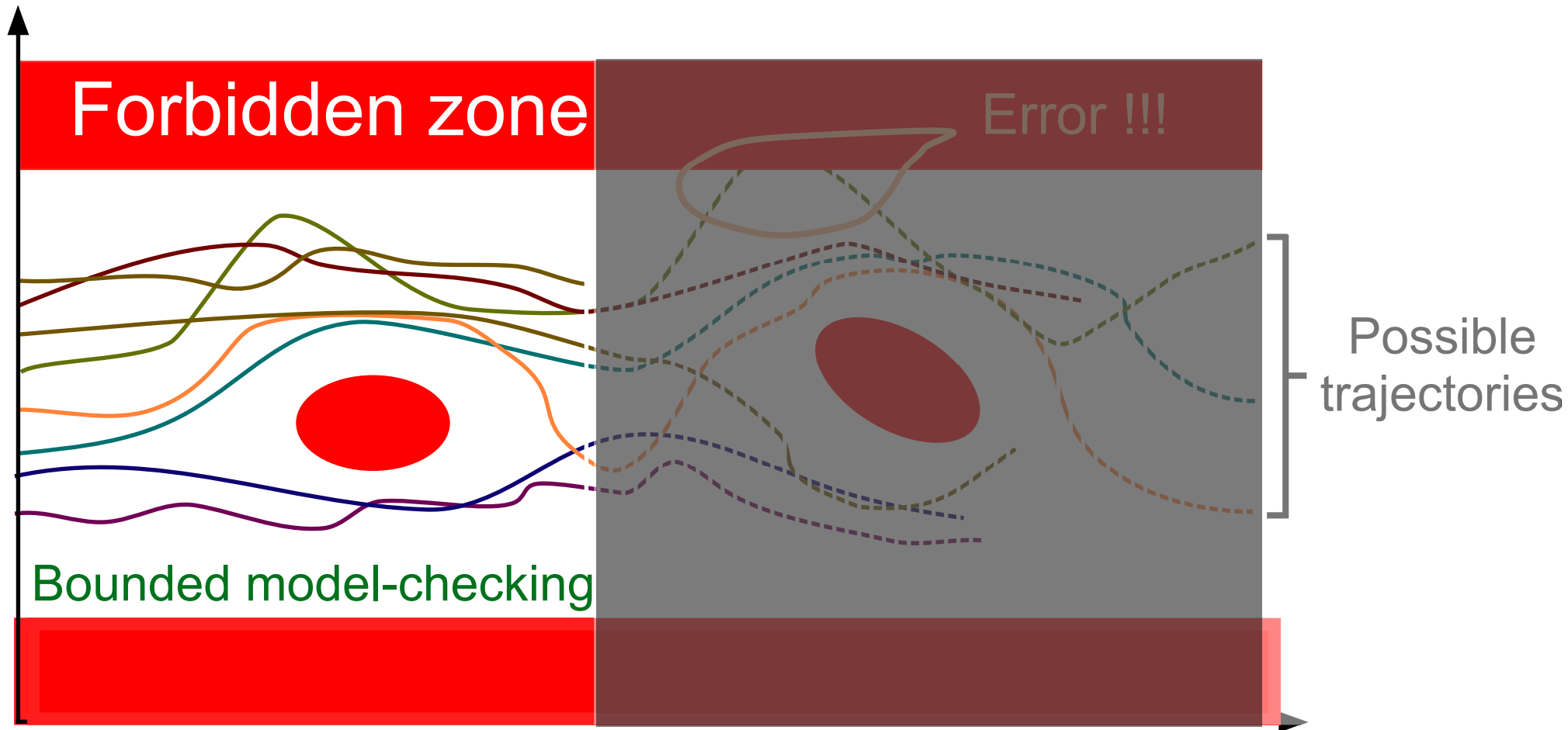
Unsound validation: testing

Try a few cases



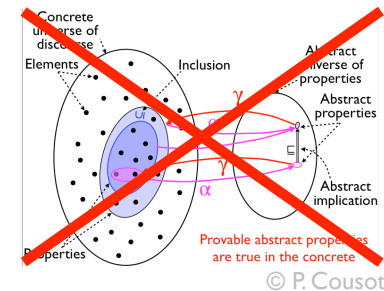
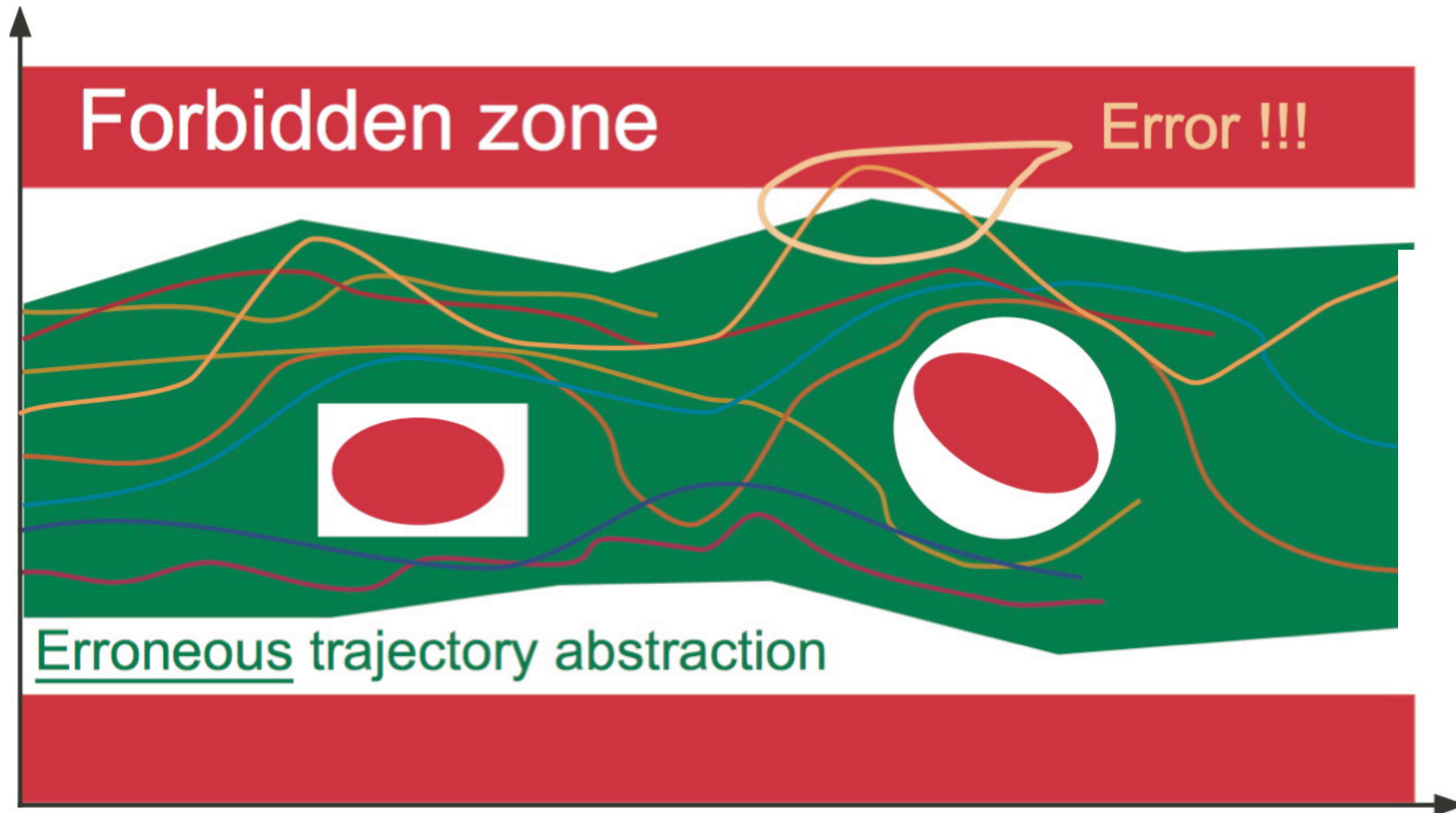
Unsound validation: bounded model-checking

Simulate the beginning of all executions



Unsound validation: static analysis

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

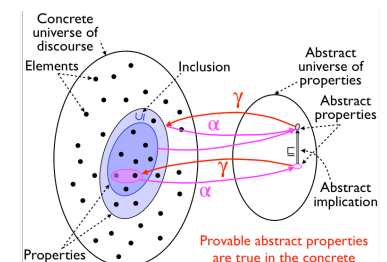


Incompleteness

When abstract proofs may fail while concrete proofs would succeed

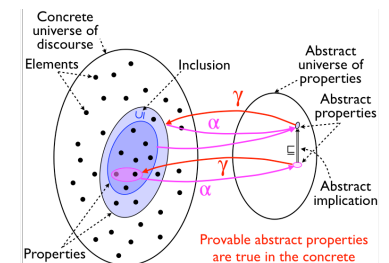
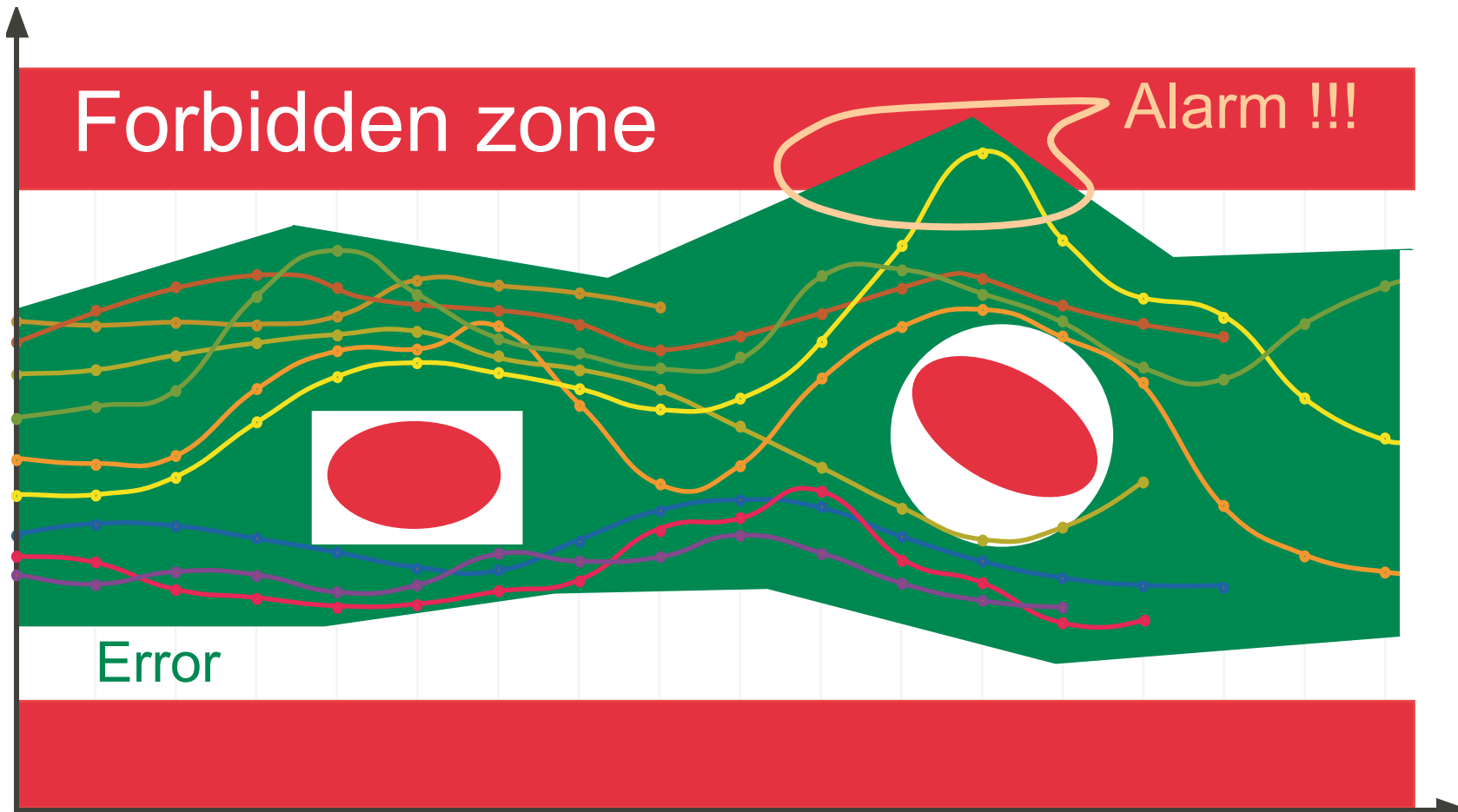


By soundness an alarm must be raised for this overapproximation!



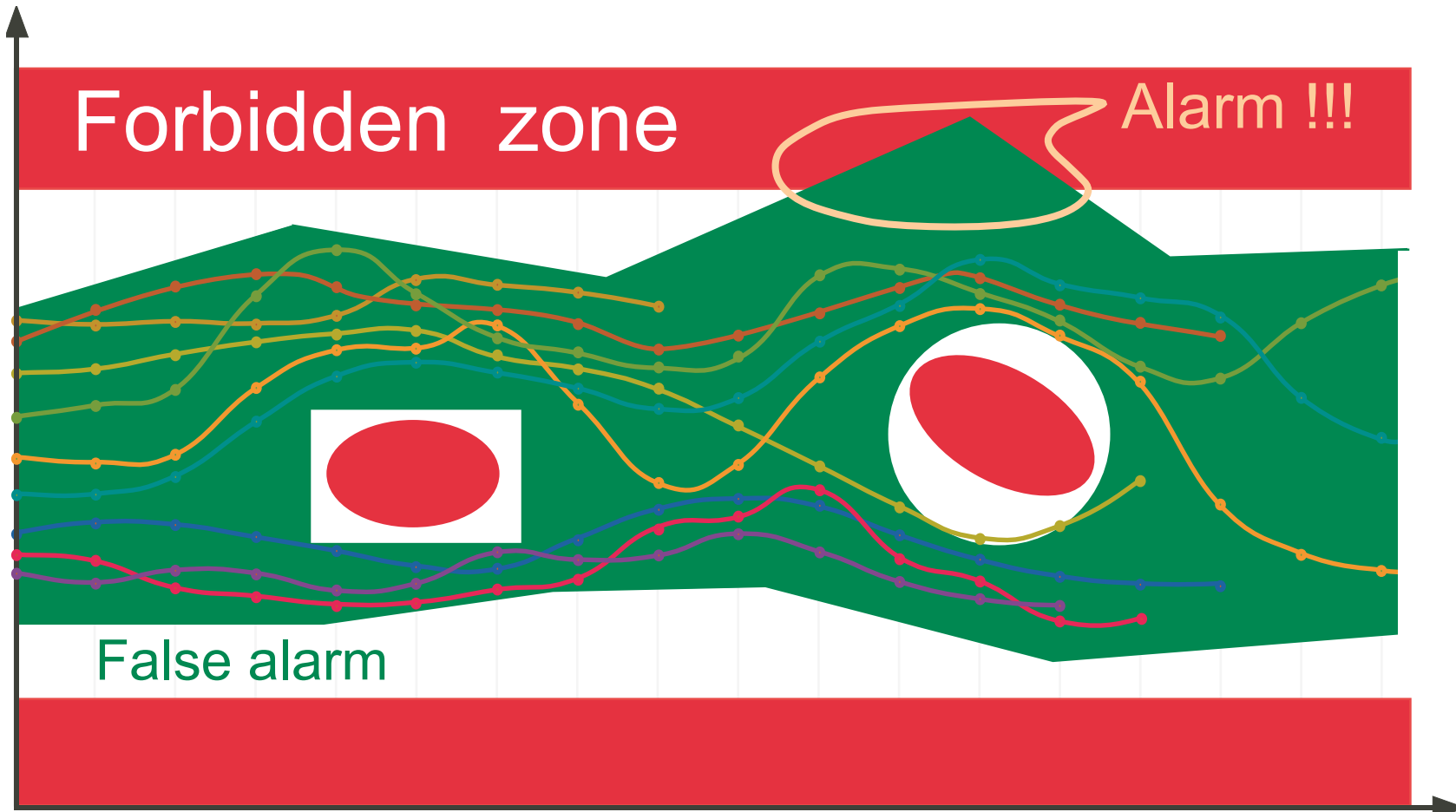
True error

The abstract alarm may correspond to a concrete error

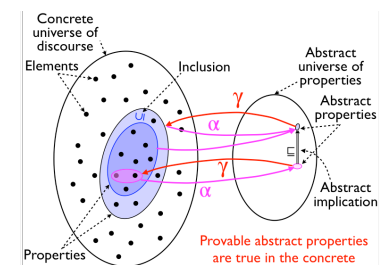


False alarm

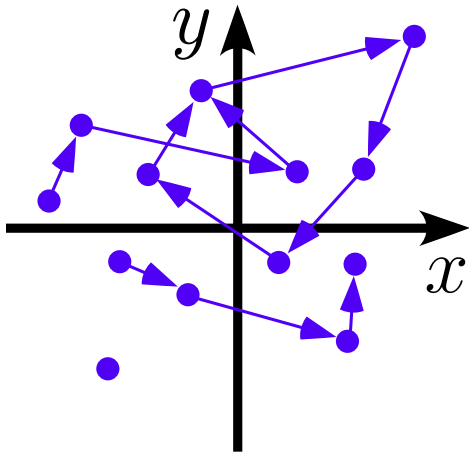
The abstract alarm may correspond to no concrete error (false negative)



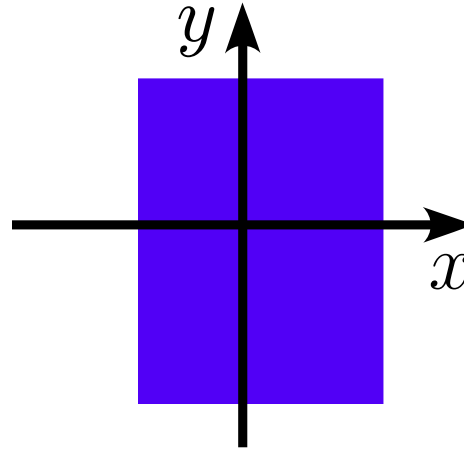
The only solution is to refine the analysis to take more properties into account (e.g. specifically for a domain of application)!



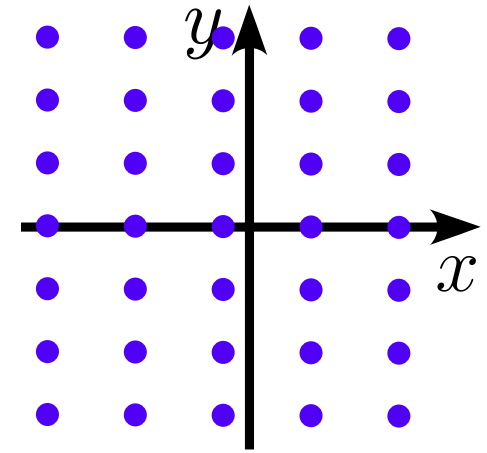
Combination of abstractions in Astrée



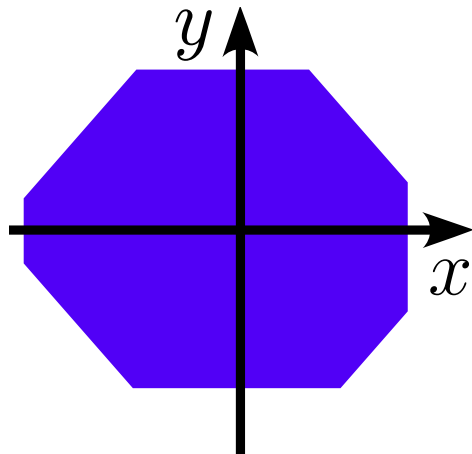
Collecting semantics:¹
partial traces



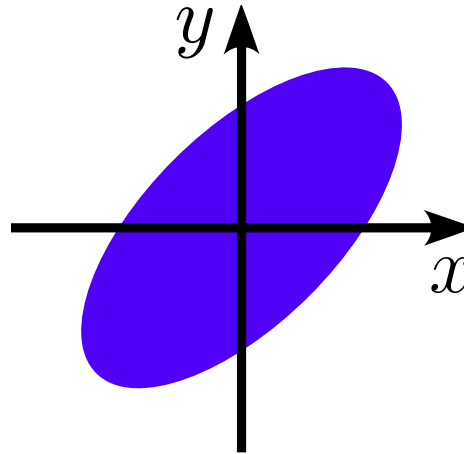
Intervals:
 $x \in [a, b]$



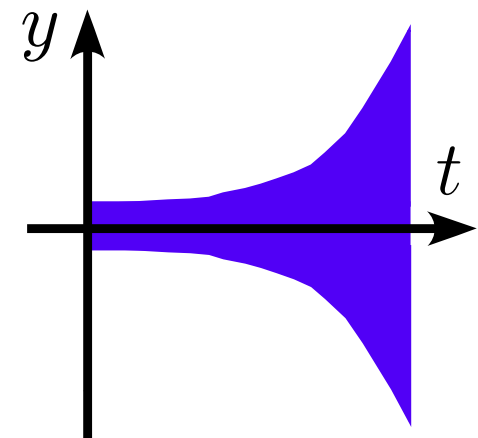
Simple congruences:
 $x \equiv a[b]$



Octagons:
 $\pm x \pm y \leq a$



Ellipses:
 $x^2 + by^2 - axy \leq d$



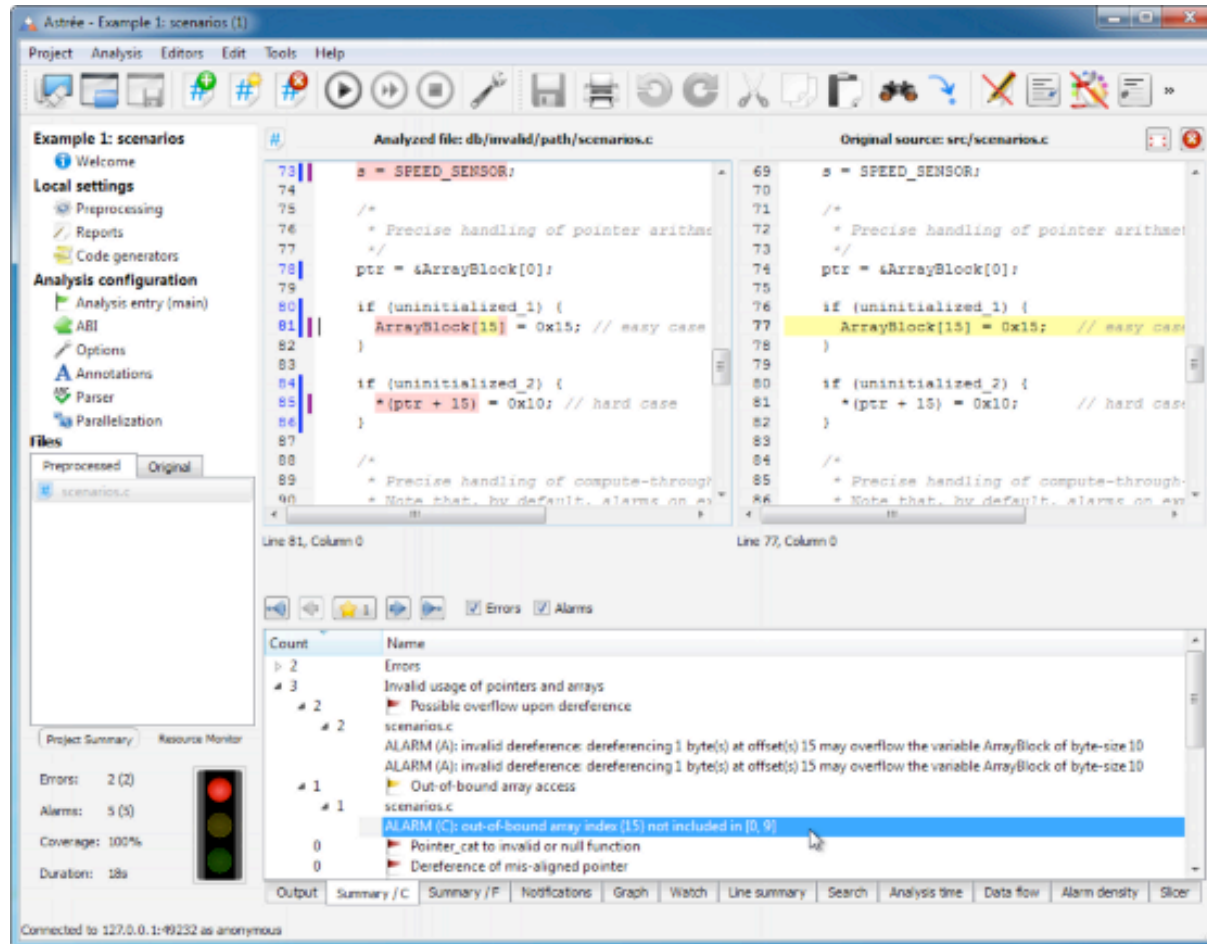
Exponentials:
 $-a^{bt} \leq y(t) \leq a^{bt}$

Examples of abstract interpretation-based program verification tools

Example 1: Astrée

Astrée

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



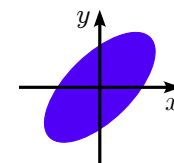
- Effectively used in production to qualify truly large and complex software in transportation, communications, medicine, etc

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

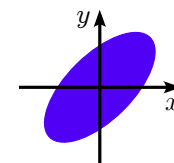
- Abstract interpretation is the only formal method able to **automatically infer program properties**
- All others can **only check your assertions**

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

Example of domain-specific abstraction: ellipses

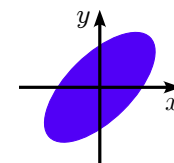
```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
    static float E[2], S[2];
    if (INIT) { S[0] = X; P = X; E[0] = X; }
    else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
                + (S[0] * 1.5)) - (S[1] * 0.7)); }
    E[1] = E[0]; E[0] =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

The screenshot displays the Microsoft Visual Studio interface. The main window shows a C# code file named `Program.cs` with a `BinarySearch` method. The code includes contract annotations such as `Contract.Ensures` and `Contract.Requires`. The `Code Contracts` settings window is open, showing the configuration for the static checker. The `Perform Static Contract Checking` checkbox is checked, and various options like `Check in Background`, `Implicit Non-Null Obligations`, and `Implicit Array Bounds Obligations` are also checked. The `Warning Level` is set to `low`. The `Error List` window at the bottom shows four items:

Description	File	Line	Column	Project
CodeContracts: Suggested requires: <code>Contract.Requires(array != null);</code>	Program.cs	22	7	Example
CodeContracts: Suggested Code fix: Consider replacing the expression <code>(inf + sup) / 2</code> with an equivalent, yet not overflowing expression. Fix: <code>inf + (sup - inf) / 2</code>	Program.cs	27	9	Example
CodeContracts: Possible use of a null array 'array'	Program.cs	23	7	Example
CodeContracts: Checked 14 assertions: 12 correct 2 unknown	Example.exe	1	1	Example

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...

Abstract Interpretation: Past, Present and Future

Patrick Cousot
CIMS*, NYU, USA
pcousot@cims.nyu.edu

Radhia Cousot
CNRS Emeritus, ENS**, France
rcousot@ens.fr

- 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

Abstract

Abstract interpretation is a theory of abstraction and constructive approximation of the mathematical structures used in the formal description of complex or infinite systems. The inference of verification of their computational independence properties has been studied since several years in various situations, either implicitly or explicitly, to many aspects of computer science (such as static analysis and verification, contract inference, type inference, termination inference, model-checking, abstraction/refinement, program transformation (including watermarking, obfuscation, etc), compilation, detection of redundancies during software testing, data-flow analysis, etc.) and more recently in the context of SAT/SMT solvers. High-quality verification tools based on abstract interpretation are available and used in the advanced software, hardware, transportation, communication, and medical industries.

The talk will consist in an introduction to the basic notions of abstract interpretation and the induced methodology for the systematic development of sound abstract interpretation-based tools. Examples of abstractions will be provided, from semantics to typing, grammars to safety, reachability to potential/definite termination (numerical to protein-protein abstractions, as well as applications to hardware and software verification). This paper is a general discussion of abstract interpretation, with selected publications, which unfortunately are far from exhaustive both in the current research and the corresponding references. *Categories and Subject Descriptors*: D.2.4 [Software Program Verification]; F.3.1 [Formal Verification and Theory]; F.3.3 [Verification and Reasoning about Programs].

General Terms: Algorithms, Languages, Reliability, Security, Verification, Static Analysis.

1. Introduction

Non-trivial complex systems (including complex systems) can be done without abstracting the behavior, i.e. the semantics, of

the system. Because reasoning on a system involves determining or proving its properties, the central concept is the abstraction of properties of the system, starting from the strongest one, as specified by the system semantics (and called the collecting semantics¹). This is the purpose of abstract interpretation (where “interpretation” stands for “reasoning” or “execution”). A few gentle introductions (e.g. Cousot and Cousot [4, 79]) can be consulted for a first approach, including some publicly available on the web (e.g. web.mit.edu/16.399/www/).

2. Soundness

Abstract interpretation comprehends undecidable problems (hence is also applicable to decidable but complex ones). This implies that any tool (prover, checker, analyzer) designed by abstract interpretation will fail on infinitely many counter-examples. For example a finiteness or decidability hypothesis will only be applicable to a very restricted class of programs with finite behavior, hence will fail on infinitely many other ones. This is inherent to undecidable problems hence inescapable. By failure we understand being unsound/incorrect, non-terminating, using a human oracle to assist the computation. Although abstract interpretation also applies to these (e.g. Cousot and Cousot [4, 79]), and fully automatic program analysis verification (including inferences of sound inductive arguments (like invariants) to deal with infinite recurrences for unbounded/non-terminating executions, which makes the problem particularly difficult, with a very high complexity).

3. Static analysis

The origin of abstract interpretation is in static program analysis [30, 31] where reachable states are abstracted by local intervals (numerical intervals) under a fixed point calculation. The inference of the static analysis (with respect to the abstract semantics) is formalized by a Galois insertion and convergence acceleration of the iterates by widening, later improved by narrowing. The main innovations at the time were to consider infinite non-coherent abstractions of infinite systems and to prove the correctness of the static analysis with respect to the original semantics (see more details in footnote ⁶).

The collecting static semantics is a strong property (standard semantics).

² For example, some commercial products do consider only two iterations in loops without widening, which is an under-approximation of an over-approximation (abstraction of program executions) and can be formalized by abstract interpretation theory. The authors do not have beyond doubt the time to claim that the result must be claimed to be “sound” (over-approximation of program behavior, a conclusion which is not always stated clearly enough for practitioners to have a precise understanding of the scope of commercial static analyzers).

¹ This work is supported in part by the European NSF award 0976166.
² We are supported in part by the European ARTIMIS project MBAL grant agreement N° 269333.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third parties may not be reproduced.
For all other use, contact the publisher/author.
Copyright is held by the author(s).
CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.
Copyright © 2014 ACM ACM 978-1-4503-2886-9/14/07...\$15.00.
<http://dx.doi.org/10.1145/2603088.2603165>

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