Abstract Interpretation for Software Verification

Patrick COUSOT
École Normale Supérieure, 45 rue d'Ulm
75230 Paris cedex 05, France

mailto:cousot@ens.fr
http://www.di.ens.fr/~cousot

Workshop on Formal Design of Safety Critical Embedded Systems,
21–23 March 2001

Abstract

From a mathematical point of view, abstract interpretation formalizes the idea of approximation of sets and set operations as considered in set (or category) theory. From a computer science point of view, abstract interpretation is a theory of approximation of the semantics of programming languages. Its main application has been the formal design of static program analyzers aiming at inferring general runtime properties of programs statically, that is without executing them. Such fully automatic analyzers have been used successfully to track bugs in safety critical embedded systems and to verify the absence of runtime errors.

The purpose of this talk is to explain the basic principles of abstract interpretation and to explain, in an informal way, how the concept of approximation, formalized by this theory, is central to all software verification methods (from debugging, to typing, model-checking and deductive methods). In abstract interpretation we consider safe approximations which provide a full coverage of all possible cases at run-time.

Then, we sketch the application to the formal design of static program analyzers by compositional abstraction of a programming language semantics. The price to be paid for the effective static safe approximation of run-time properties is that of uncertainty. Practical analyzers have therefore to provide a good compromise between the cost of the analysis and its precision. We discuss various methods which can be used to obtain an acceptable balance.

Finally, we explain an application of static program analysis to abstract testing, a version of interactive program debugging where the execution of programs on test values are replaced by the static checking of user provided program properties.

Introductory Motivations on Software Reliability

The Software Reliability Problem

- The evolution of hardware by a factor of 10^6 over the past 25 years has lead to the explosion of the program sizes;
- The scope of application of very large software is likely to widen rapidly in the next decade;
- These big programs will have to be modified and maintained during their lifetime (often over 20 years);
- The size and efficiency of the programming and maintenance teams in charge of their design and follow-up cannot grow up in similar proportions;

The Software Reliability Problem (Cont'd)

- At a not so uncommon (and often optimistic) rate of one bug per thousand lines such huge programs might rapidly become hardly manageable in particular for safety critical systems;
- Therefore in the next 10 years, the *software reliability problem* is likely to become a major concern and challenge to modern highly computer-dependent societies.

Software Verification Costs

In avionics software:

- Currently, 50% of the software cost;
- Can be significantly exceeded (depending on criticality level);
- Will increase dramatically in the near future without radical improvements of verification cost-effectiveness.

What Can We Do About It?

- Use our intelligence (thinking/intellectual tools: abstract interpretation);
- Use our computer (mechanical tools : static program analysis/checking/testing).

Software Verification and Validation

Verification versus Validation

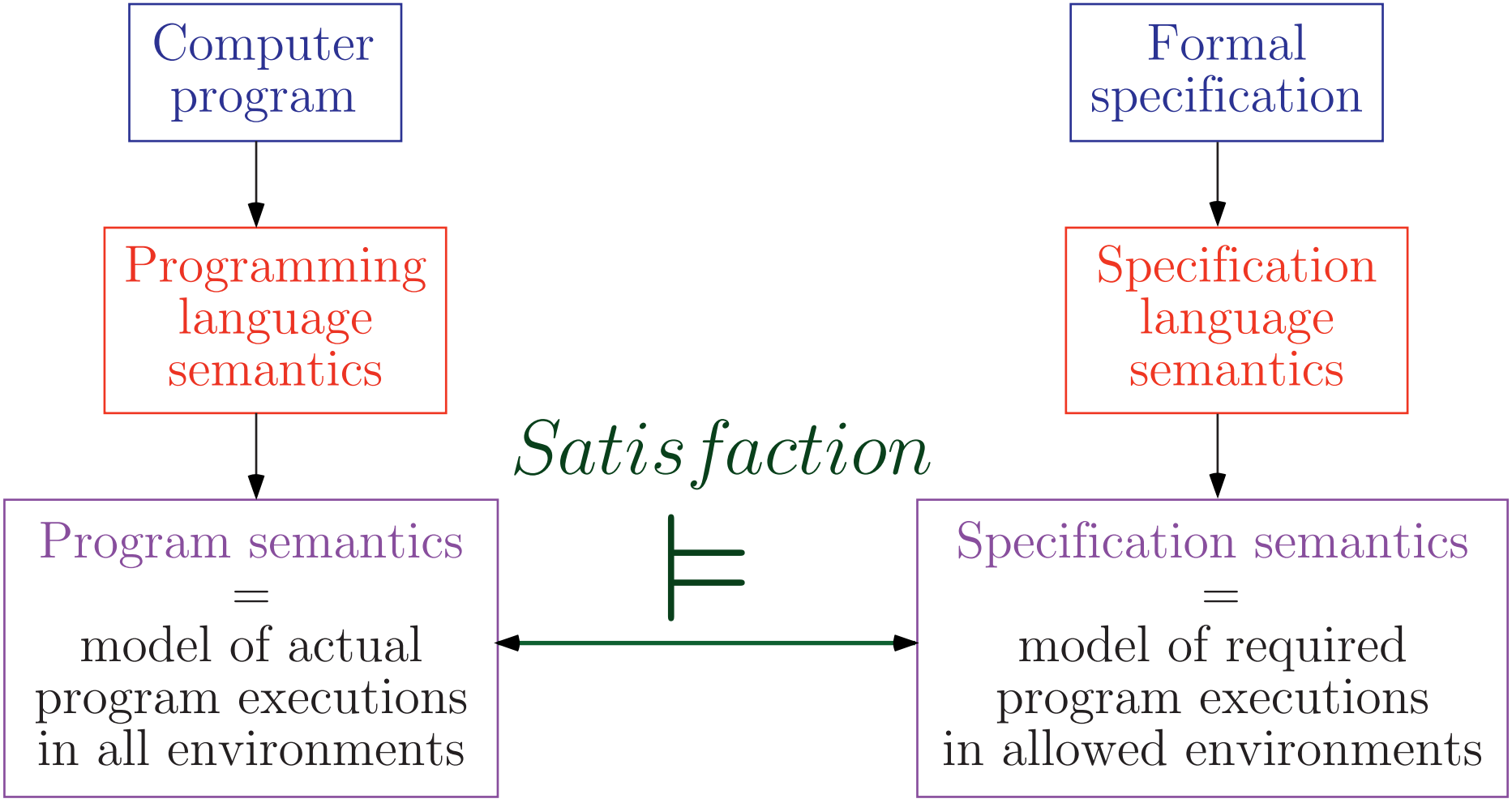
One definition (among others):

Validation: absence of failures and satisfaction of the formal and proper usage requirements of a fully developed software;

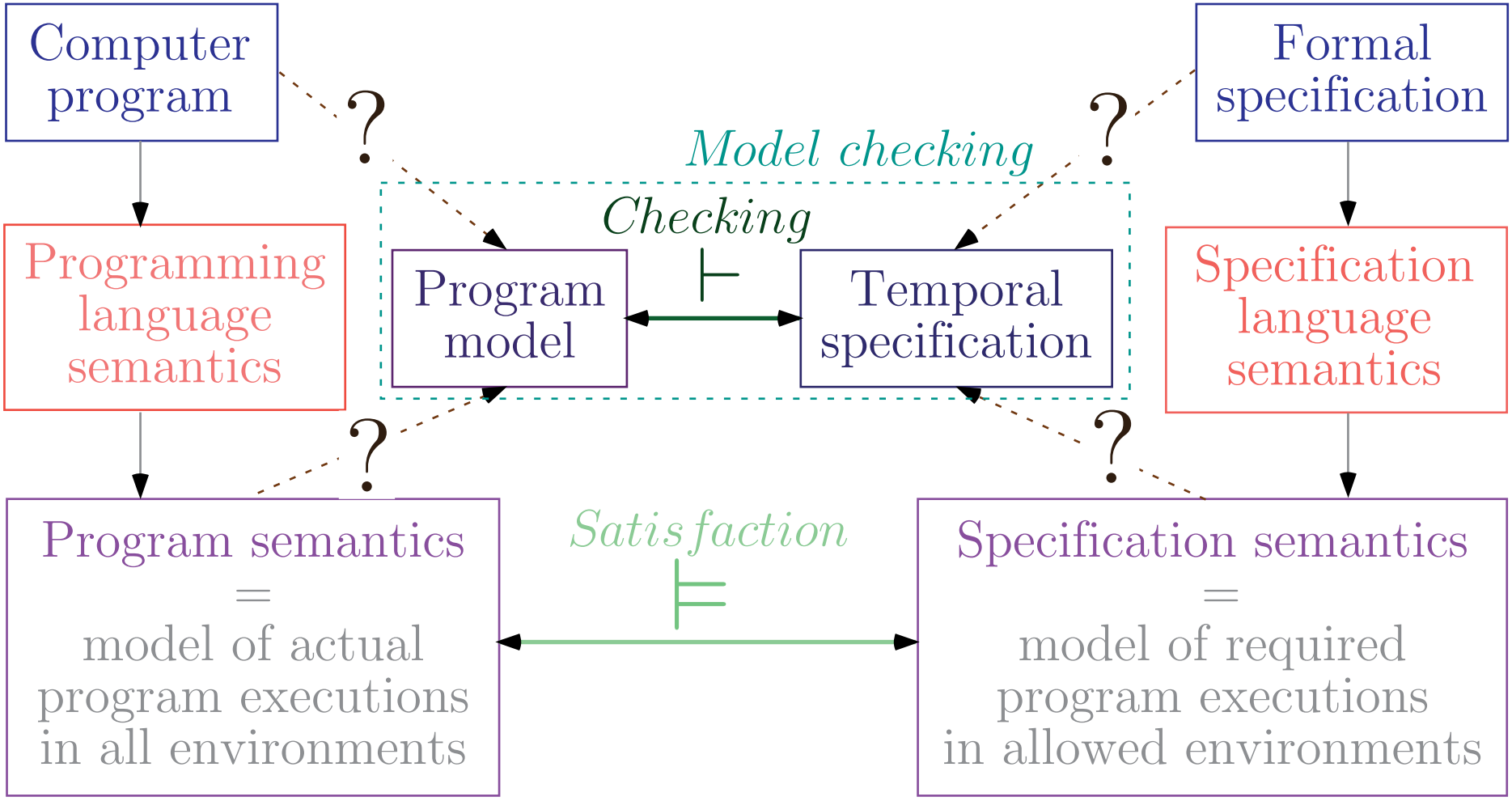
Verification: satisfaction of the formal requirements during software development.

As far as static analysis is concerned, not fundamentally different (a useful software is permanently evolving).

The Verification/Validation Problem



Example: Model Checking



Other Examples of Software Verification/Validation Techniques

- Software testing;
- Simulation and prototyping;
- Technical reviews;
- Requirements tracing;
- Formal correctness proofs;
- Etc.

Practical Limitations

- Testing:
  - Testing all data on all paths is impossible;
- Formal methods:
  - No formal specification perfectly reflects informal human expectations;
  - Proofs grow exponentially in the size of programs/specifications which is incompatible with friendly user interaction and full automation;
- etc.

Fundamental Theoretical Limitations

- Undecidability: full automation of software verification/validation is impossible;

Undecidability and Approximation

- Since program verification is undecidable, computer aided program verification methods are all partial/incomplete;
- They all involve some form of approximation:
  - practical complexity limitations,
  - require user interaction,
  - semi-algorithms or finiteness hypotheses,
  - restricted specifications or programs;
- Most of these approximations are formalized by **Abstract Interpretation**.

Examples of approximations

- Testing: coverage is partial (so errors are frequently found until the end of the software lifetime);
- Proofs: specifications are often partial, debugging proofs is often harder that testing programs (so only parts of very large software can be formally proved correct);
- Model checking: the model must fit machine limitations (so some facets of program execution must be left out) and be redesigned after program modifications;
- Typing: types are weak program properties (so type verification cannot be generalized to complex specifications).

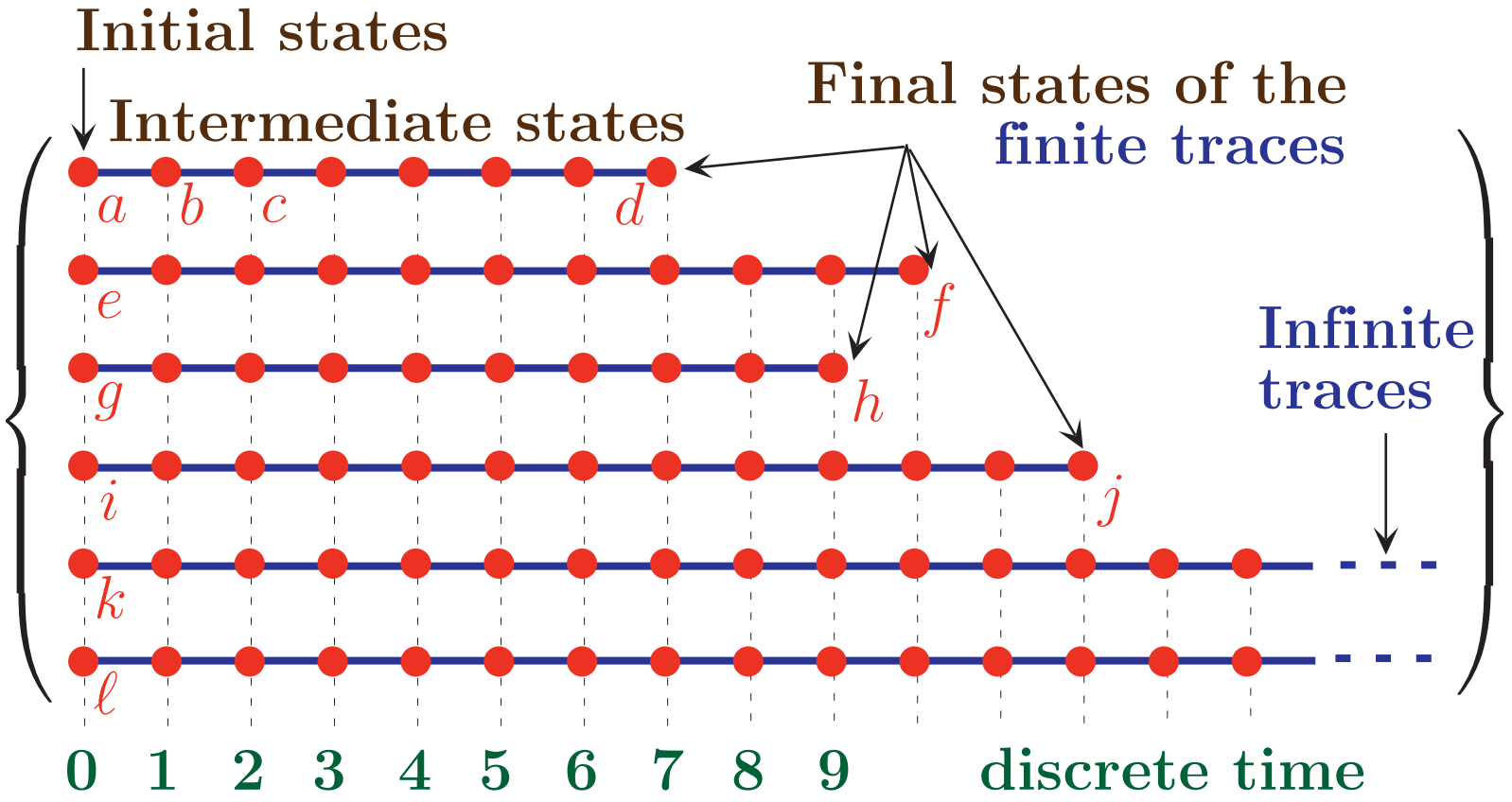
Content

Semantics

Semantics: intuition

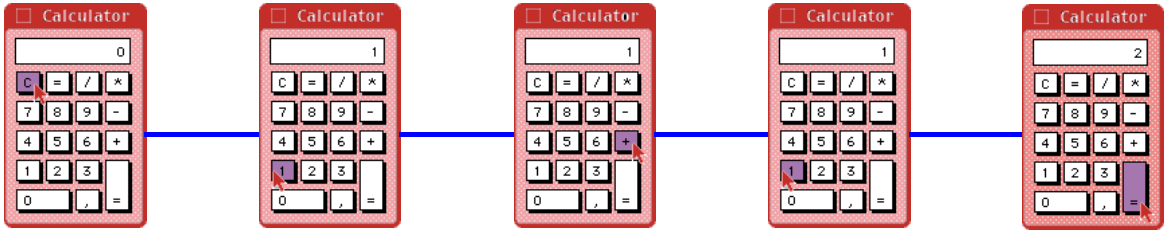
- The semantics of a language defines the semantics of any program written in this language;
- The semantics of a program provides a formal mathematical model of all possible behaviors of a computer system executing this program (interacting with any possible environment);
- Any semantics of a program can be defined as the solution of a fixpoint equation;
- All semantics of a program can be organized in a hierarchy by abstraction.

Example: trace semantics [?, 26]

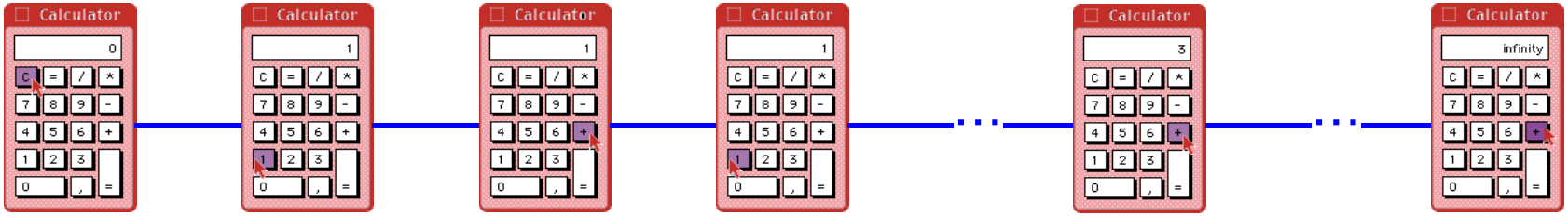


Examples of computation traces

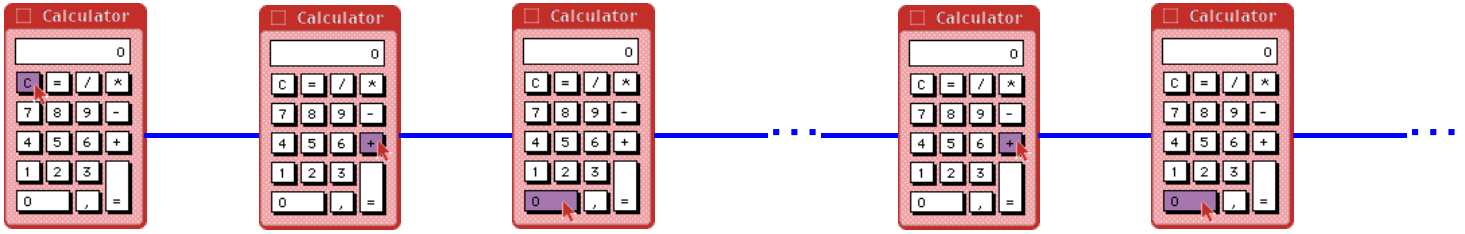
- Finite (C1+1=):



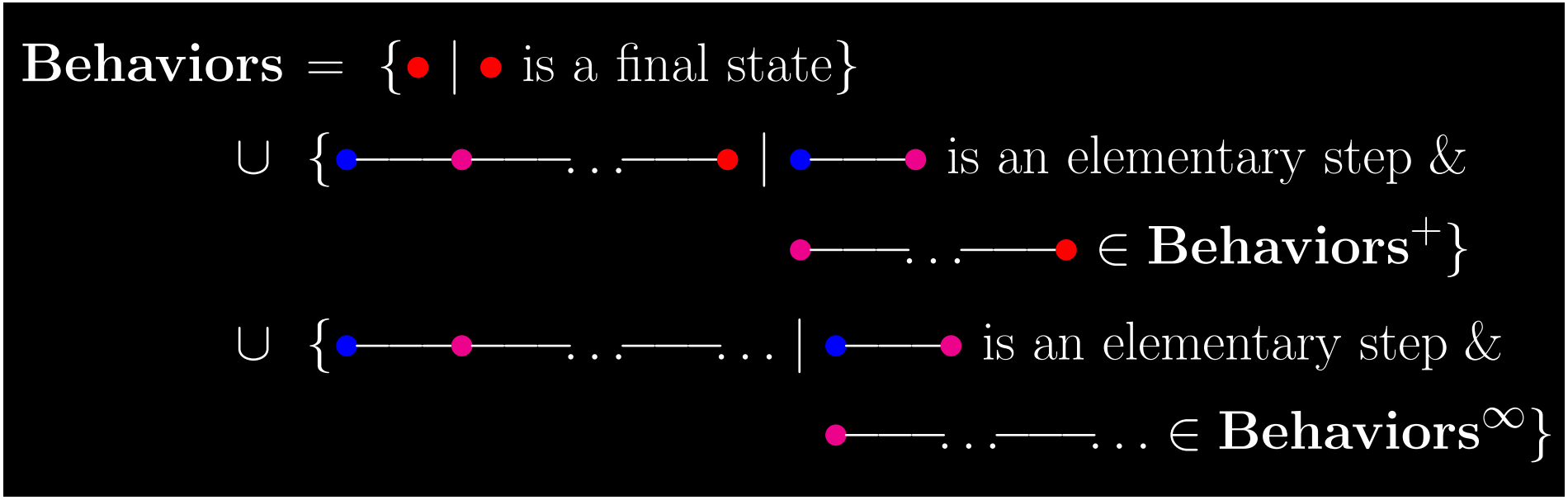
- Erroneous  (C1+1+1+1...):



- Infinite (C+0+0+0...):



Least Fixpoints: intuition [26]



- In general, the equation has multiple solutions.
- Choose the least one for the partial ordering:

« more finite traces & less infinite traces ».

Abstract Interpretation

Abstract Interpretation [1]

- Formalizes the idea of approximation of sets and set operations as considered in set (or category) theory;
- A theory of approximation of the semantics of programming languages;
- Main application: formal method for inferring general runtime properties of programs.

Reference

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages POPL'77*, Los Angeles, CA, 1977. ACM Press, pp. 238–252.

Examples of Abstract Interpretations

- Data flow analysis [2, 4];
- Set based analysis [3];

References

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[3] P. Cousot and R. Cousot. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In 7^{th} *FPCA*, pages 170–181, La Jolla, CA, 25–28 June 1995. ACM Press.

[4] P. Cousot and R. Cousot. Temporal abstract interpretation. In 27^{th} *POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.

Examples of Abstract Interpretations (Cont'd)

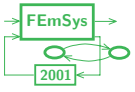
- Polymorphic type inference [5];
- Model checking [6, 7].

References

[5] P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, FR, Jan. 1997. ACM Press.

[6] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Aut . Soft . Eng.*, 6:69–95, 1999.

[7] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, MA, Jan. 2000. ACM Press.



Abstraction

Abstraction: intuition

- Abstract interpretation formalizes the intuitive idea that a semantics is more or less precise according to the considered observation level of the program executions;

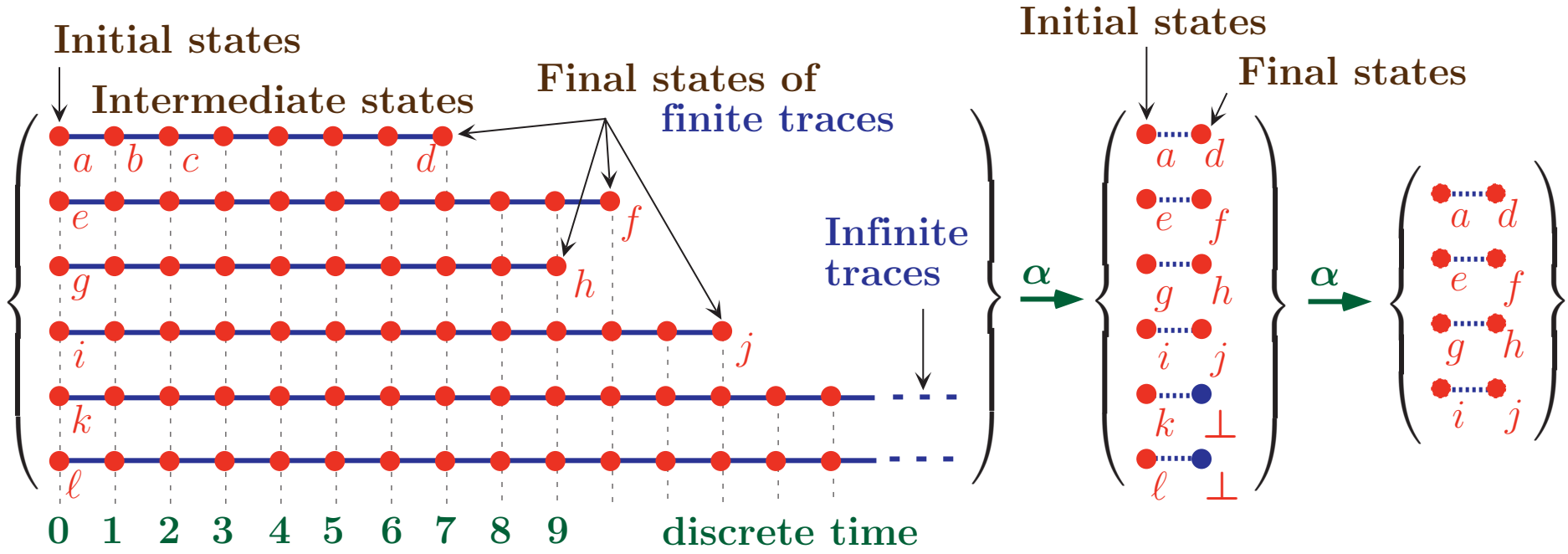
- Abstract interpretation theory formalizes this notion of approximation/abstraction in a mathematical setting which is independent of particular applications.

Abstractions of Semantics [8]

Reference

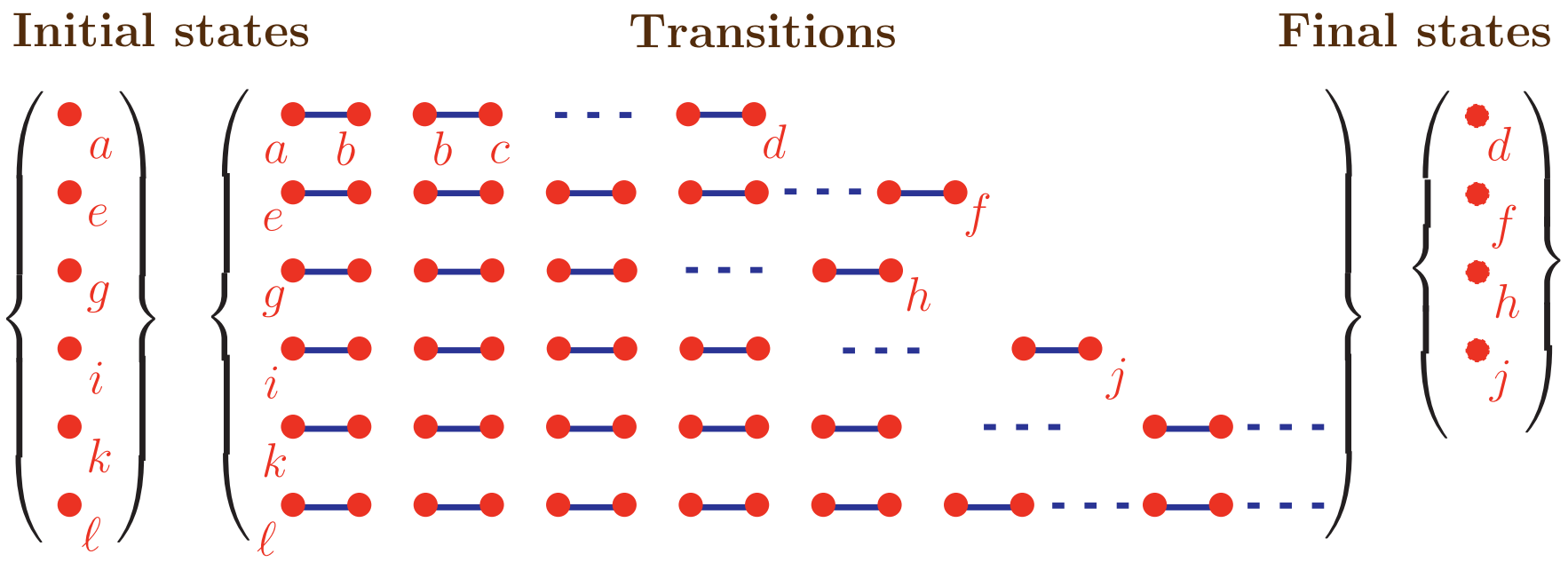
[8] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. To appear in *Theoretical Computer Science*, (2001).

Example 1 of Abstraction



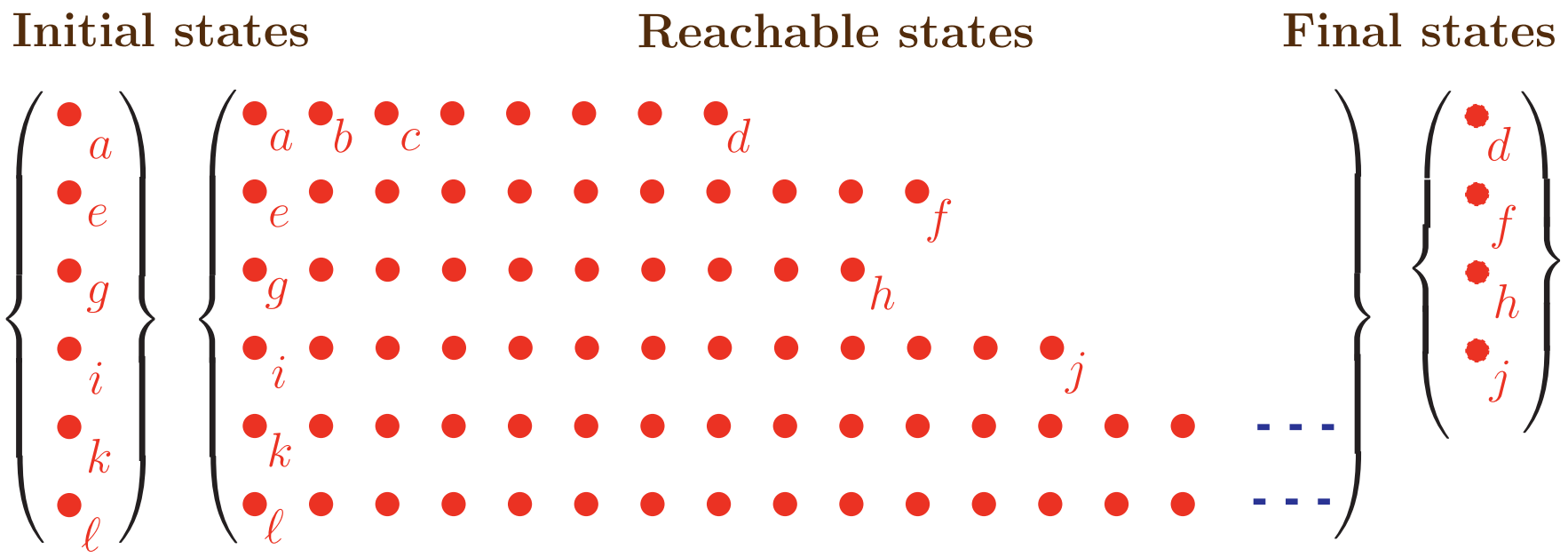
Trace semantics **Denotational semantics** **Natural semantics**

Example 2 of Abstraction



(Small-Step) Operational Semantics

Example 3 of Abstraction



Partial Correctness / Invariance Semantics

Effective Abstractions

Reference

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77*, ACM Press, pp. 238–252.

Effective Abstractions of Semantics

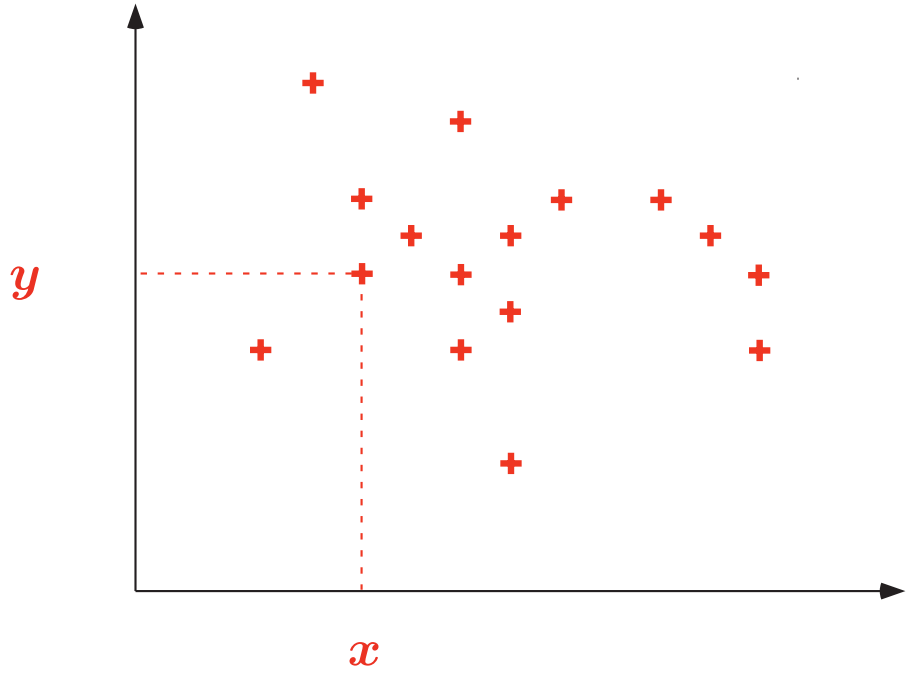
- If the approximation is rough enough, the abstraction of a semantics can lead to a version which is less precise but is effectively computable by a computer;

- The computation of this abstract semantics amounts to the effective iterative resolution of fixpoint equations;

- By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them.

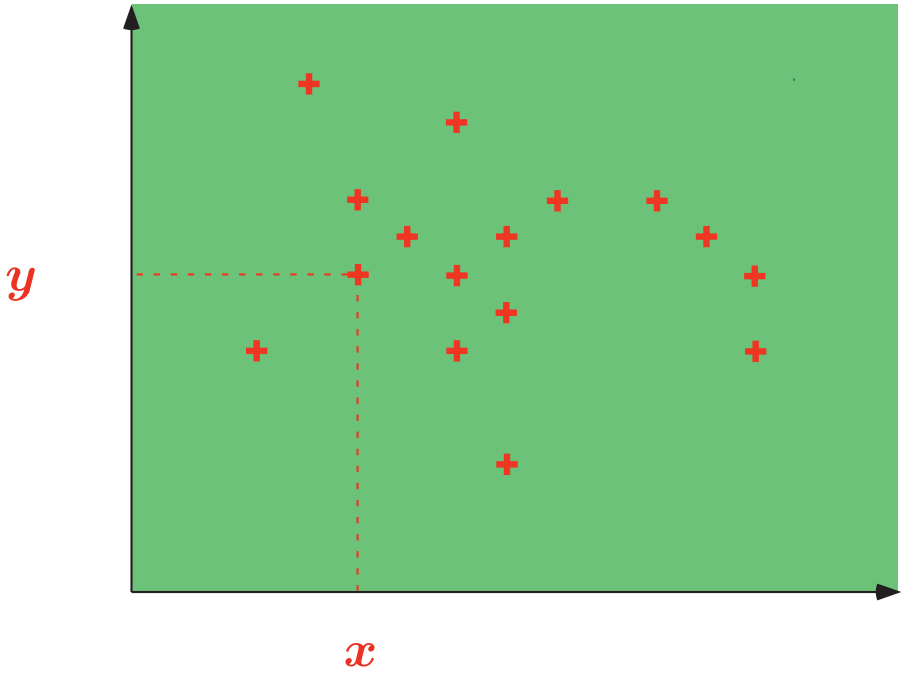
Effective Numerical Abstractions

Effective Abstractions of an [In]finite Set of Points;



$$\{\ldots, \langle 19,\ 88\rangle, \ldots, \langle 19,\ 99\rangle, \ldots\}$$

Effective Abstractions of an [In]finite Set of Points; Example 1: Signs [10]

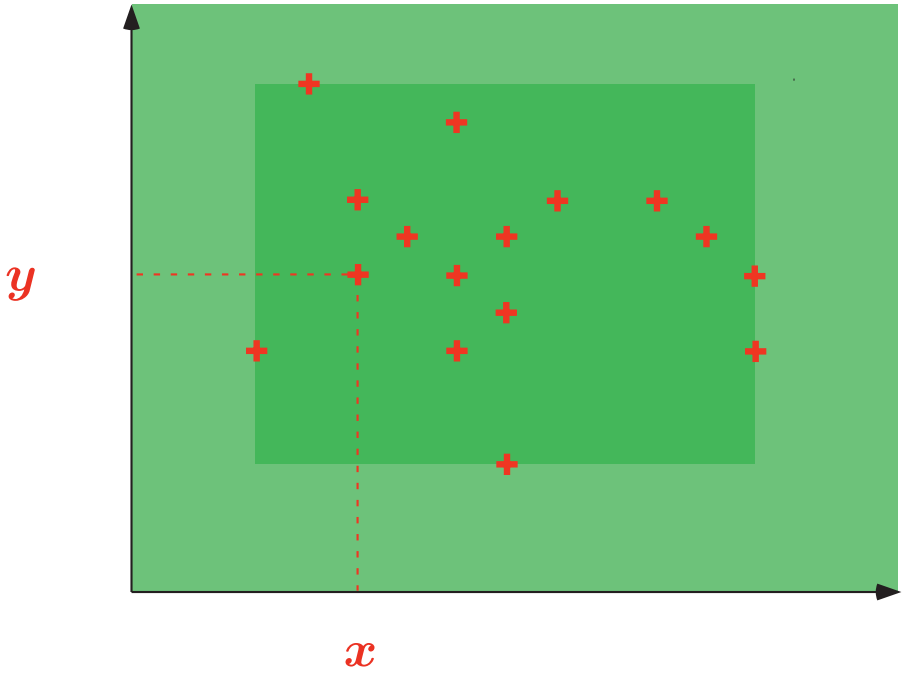


$$\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$$

Reference

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

Effective Abstractions of an [In]finite Set of Points; Example 2: Intervals [11]

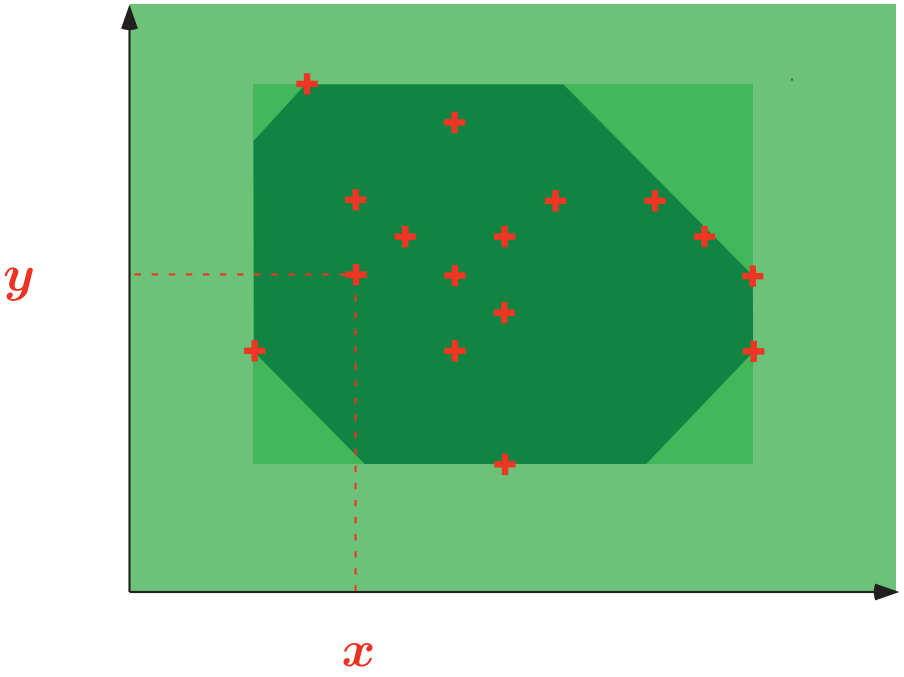


$$\begin{cases} x \in [19,\ 88] \\ y \in [19,\ 99] \end{cases}$$

Reference

[11] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

Effective Abstractions of an [In]finite Set of Points; Example 3: Octagons [12]

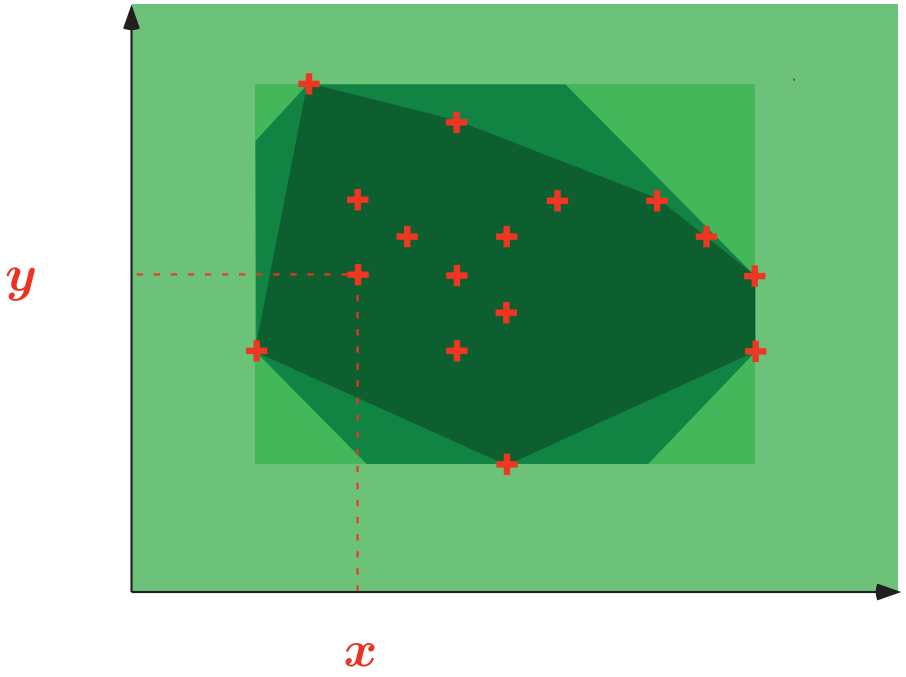


$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 88 \\ 1 \leq y \leq 9 \\ x - y \leq 99 \end{cases}$$

Reference

[12] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *2nd Symposium on Programs as Data Objects, PADO'2001*, To appear in LNCS, Springer, 2001.

Effective Abstractions of an [In]finite Set of Points; Example 4: Polyhedra [13]

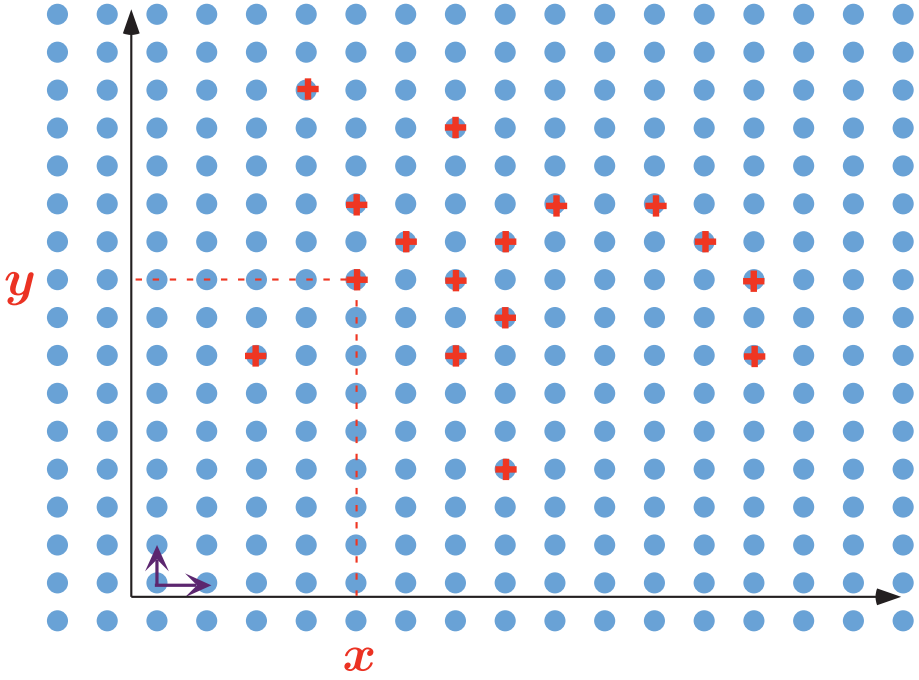


$$\begin{cases} 19x + 88y \leq 2000 \\ 19x + 99y \geq 0 \end{cases}$$

Reference

[13] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In 5^{th} *POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

Effective Abstractions of an [In]finite Set of Points; Example 5: Simple Congruences [14]

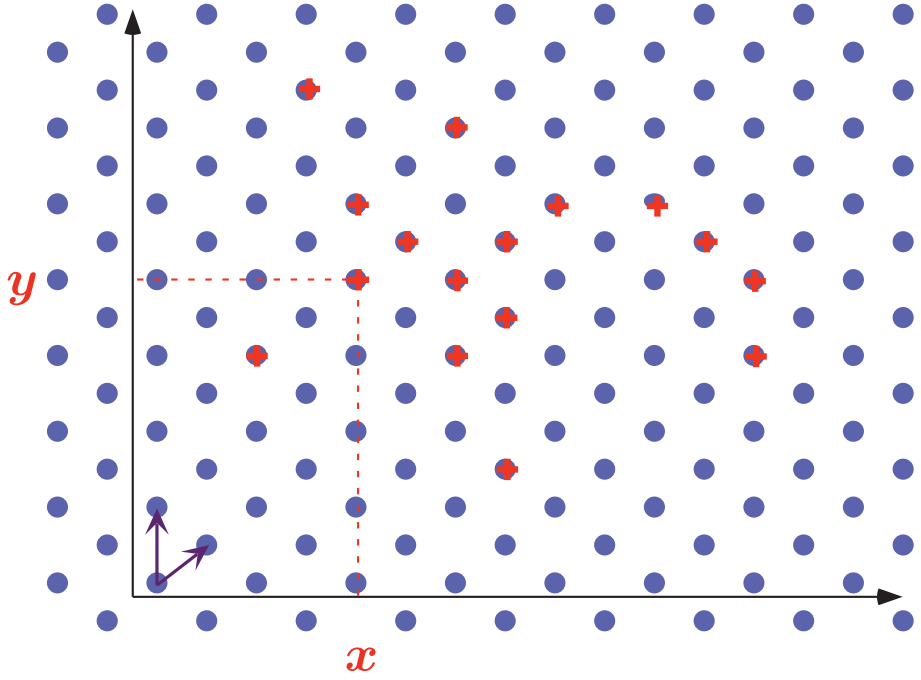


$$\begin{cases} x = 19 \bmod 88 \\ y = 19 \bmod 99 \end{cases}$$

Reference

[14] P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.

Effective Abstractions of an [In]finite Set of Points; Example 6: Linear Congruences [15]

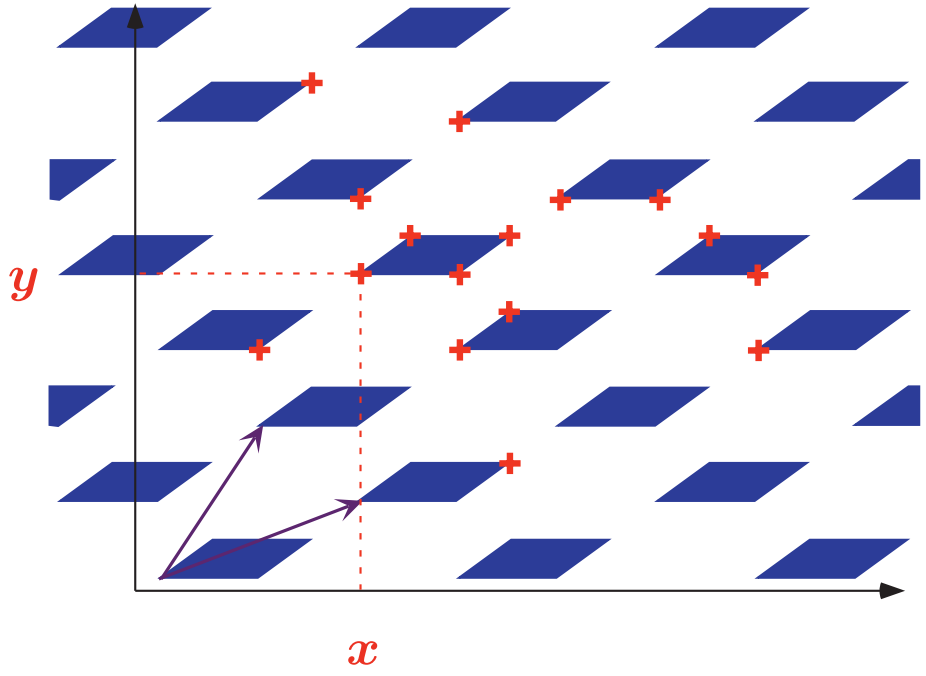


$$\begin{cases} 1x + 9y = 8 \bmod 8 \\ 1x - 9y = 9 \bmod 9 \end{cases}$$

Reference

[15] P. Granger. Static analysis of linear congruence equalities among variables of a program. *CAAP '91*, LNCS 493, pp. 169–192. Springer, 1991.

Effective Abstractions of an [In]finite Set of Points; Example 7: Trapezoidal Linear Congruences [16]



$$\begin{cases} 1x + 9y \in [0, 88] \bmod 10 \\ 1x - 9y \in [0, 99] \bmod 11 \end{cases}$$

Reference

[16] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *ACM Int. Conf. on Supercomputing, ICS '92*, pages 226–235, 1992.

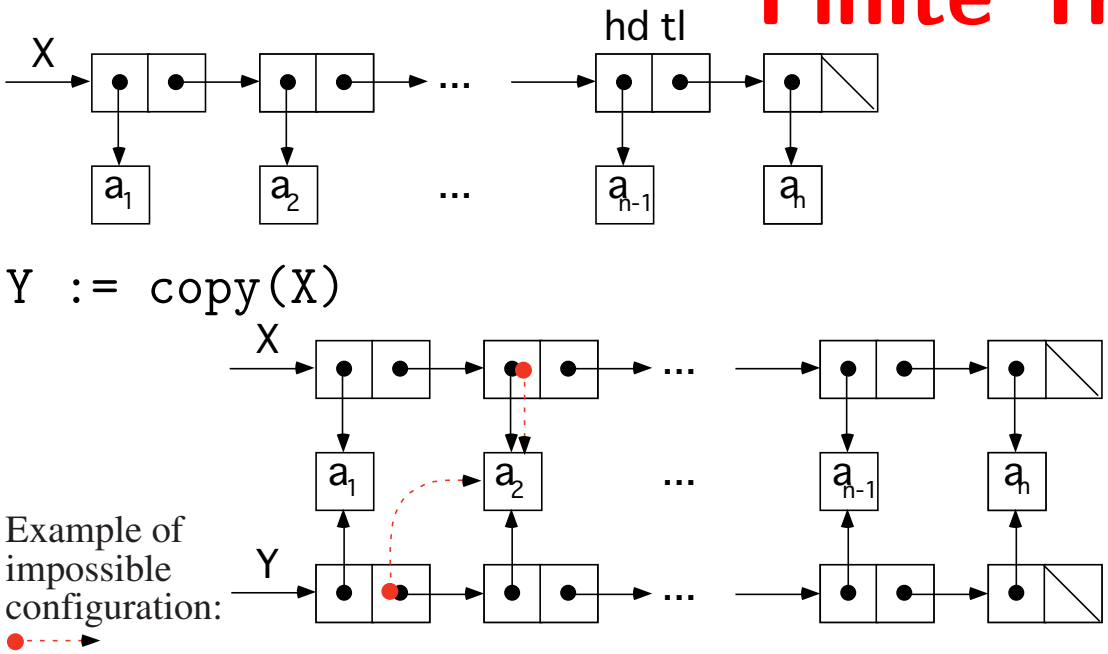
Effective Symbolic Abstractions

Effective Abstractions of Symbolic Structures

- Most structures manipulated by programs are *symbolic structures* such as control structures (call graphs), data structures (search trees), communication structures (distributed & mobile programs), etc;
- It is very difficult to find compact and expressive abstractions of such sets of objects (languages, automata, trees, graphs, etc.).

Example of Abstractions of Infinite Sets of Finite Trees

- Program :



- Alias analysis:

$\emptyset$

`Y := copy(X)`

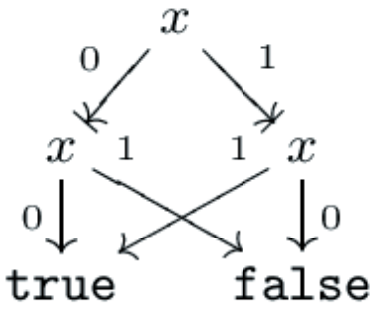
$\{(\mathtt{X} \mapsto (\mathtt{tl} \mapsto)^i \mapsto \mathtt{hd}, \mathtt{Y} \mapsto (\mathtt{tl} \mapsto)^j \mapsto \mathtt{hd}) \mid i = j\}$

Reference

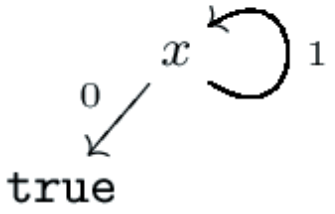
[17] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI'94*, pp. 230–241, 1994.

Example of Abstractions of Infinite Sets of Infinite Trees

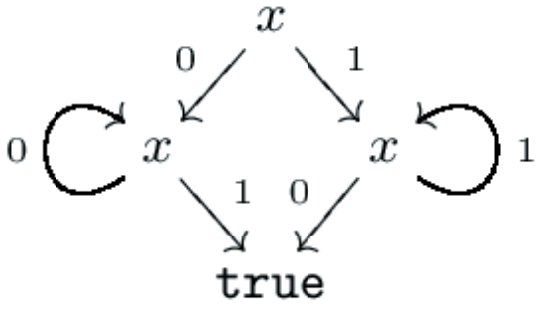
Binary Decision Graphs: [18]



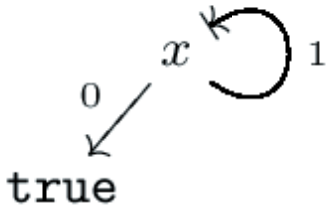
$\{0^\omega, 1^\omega\}$



infinite number of 0's



fair vectors



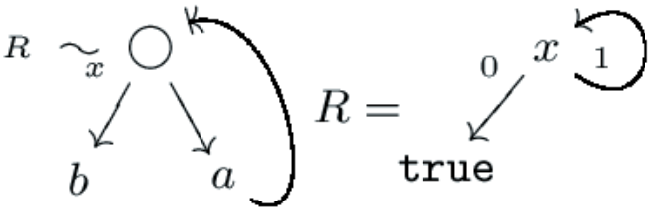
ends by 0^ω

Reference

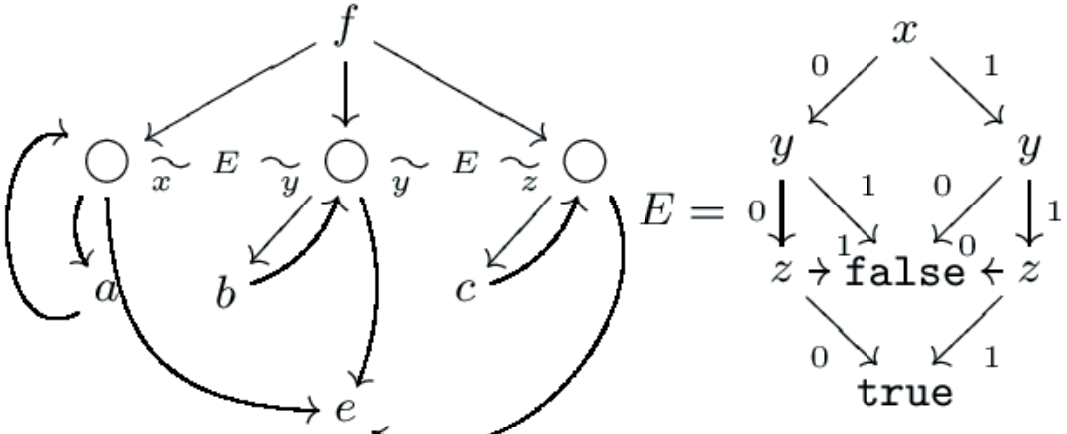
[18] L. Mauborgne. Binary decision graphs. *SAS '99*, LNCS 1694, pp. 101–116. Springer, 1999.

Example of Abstractions of Infinite Sets of Infinite Trees (Cont'd)

Tree Schemata: [19, 20]



$\{a^n b | n \in \mathbb{N}\}$



$\{f(a^n e, b^n e, c^n e) | n \in \mathbb{N}\}$

Note that E is the equality relation.

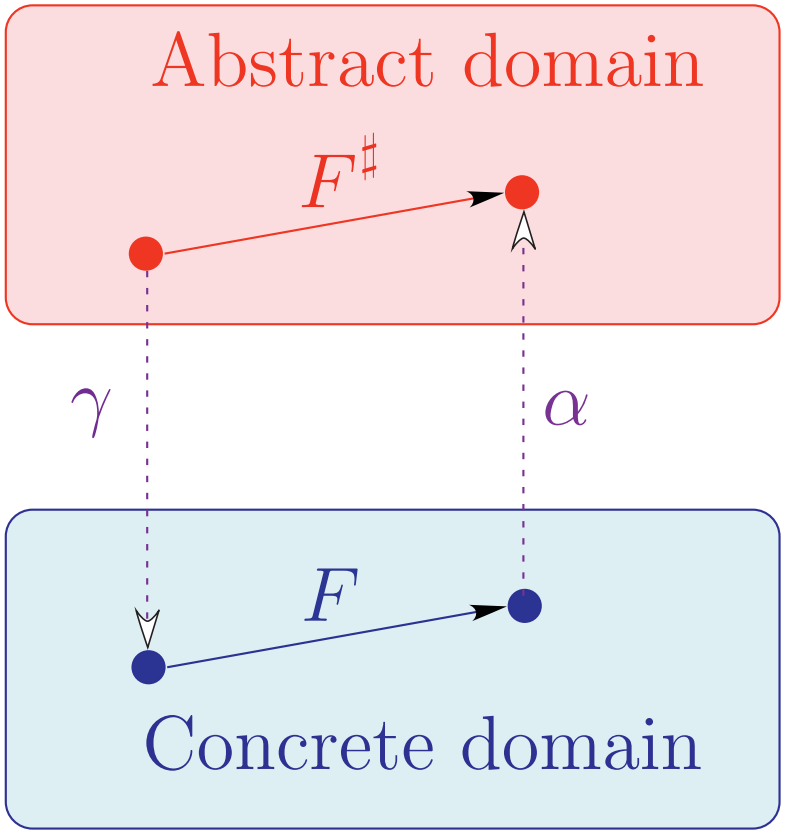
Reference

[19] L. Mauborgne. Improving the representation of infinite trees to deal with sets of trees. *ESOP '2000*, LNCS 1782, pp. 275–289. Springer, 2000.

[20] L. Mauborgne. Tree schemata and fair termination. *SAS '2000*, LNCS 1824, pp. 302–321. Springer, 2000.

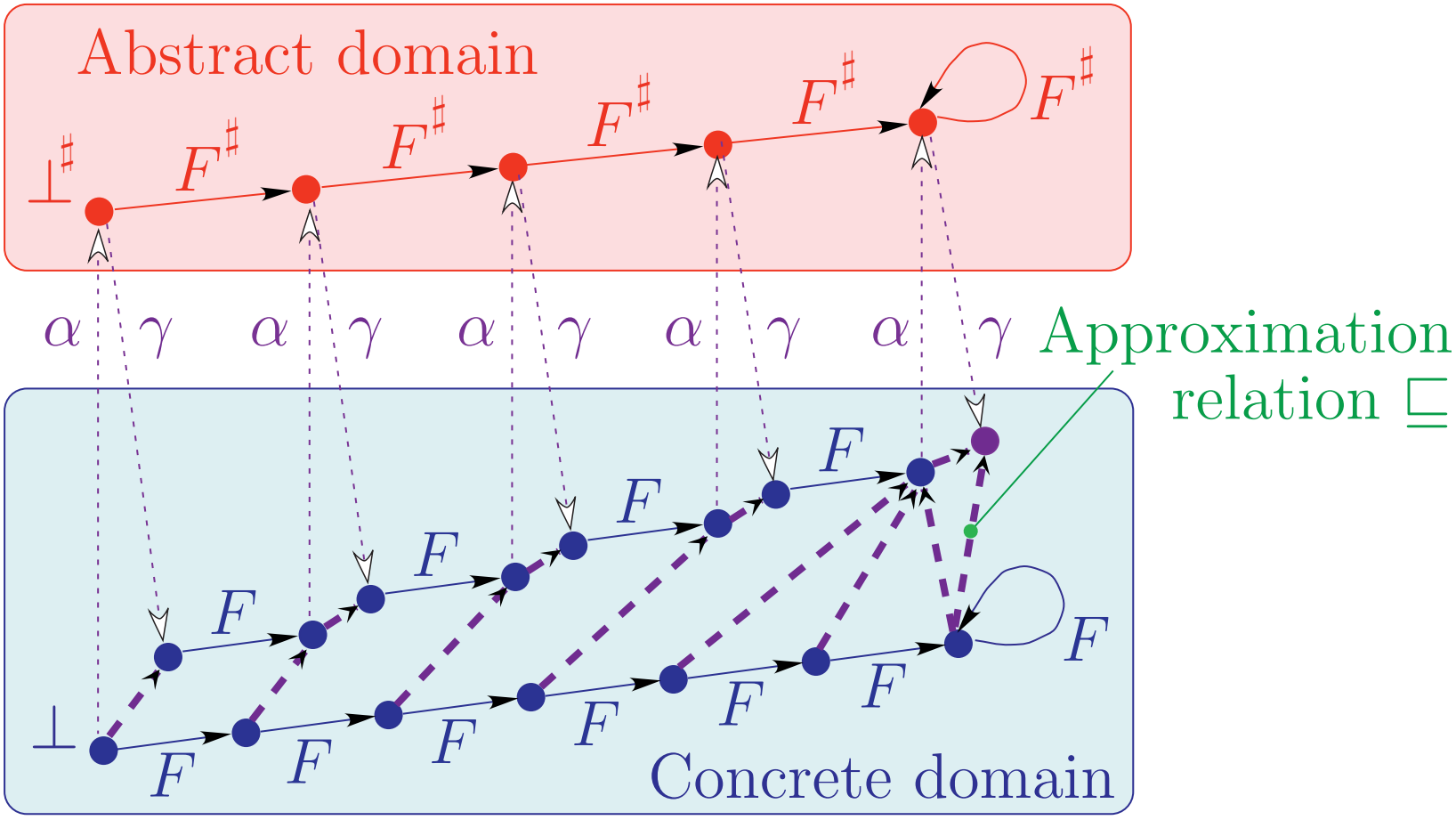
Effective Fixpoint Abstraction

Function Abstraction



$$F^\sharp = \alpha \circ F \circ \gamma$$

Fixpoint Abstraction



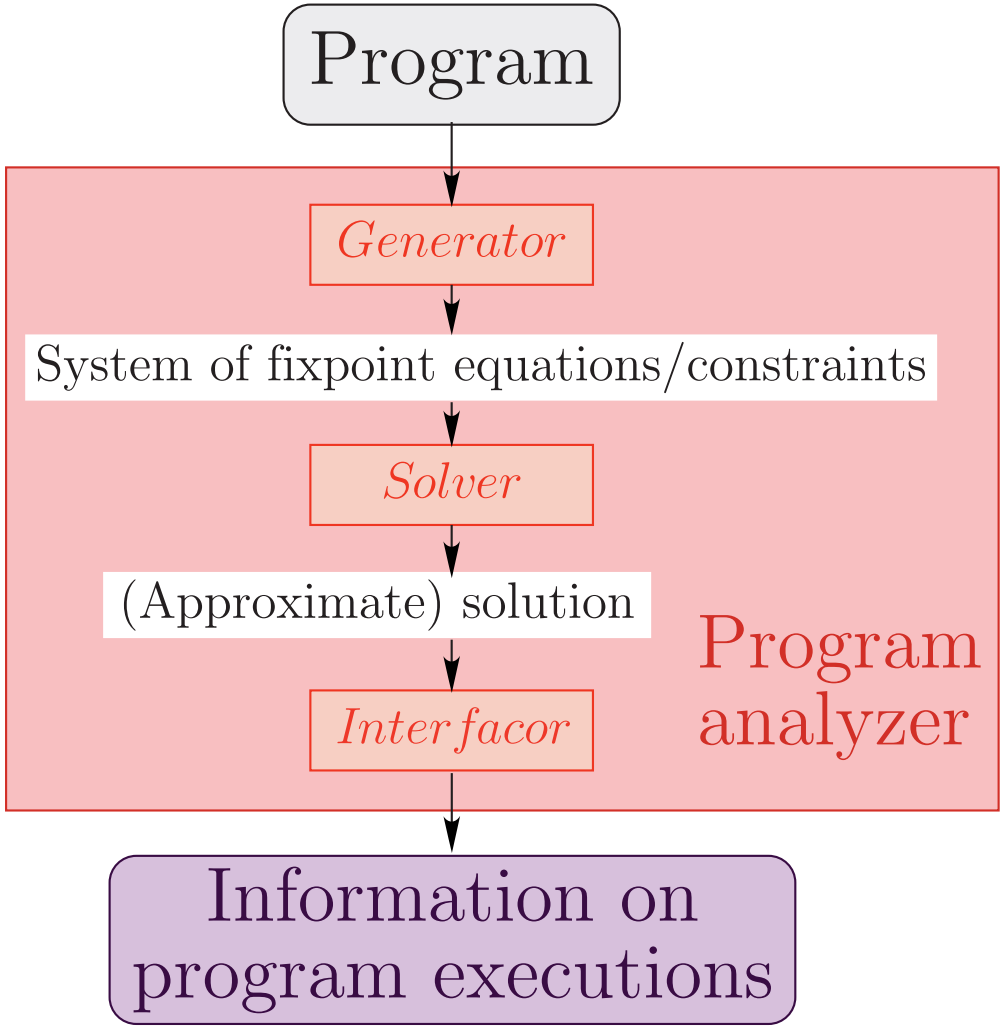
$$lfp\, F \sqsubseteq \gamma\left(lfp\, F^\sharp\right)$$

Abstract Static Program Analysis

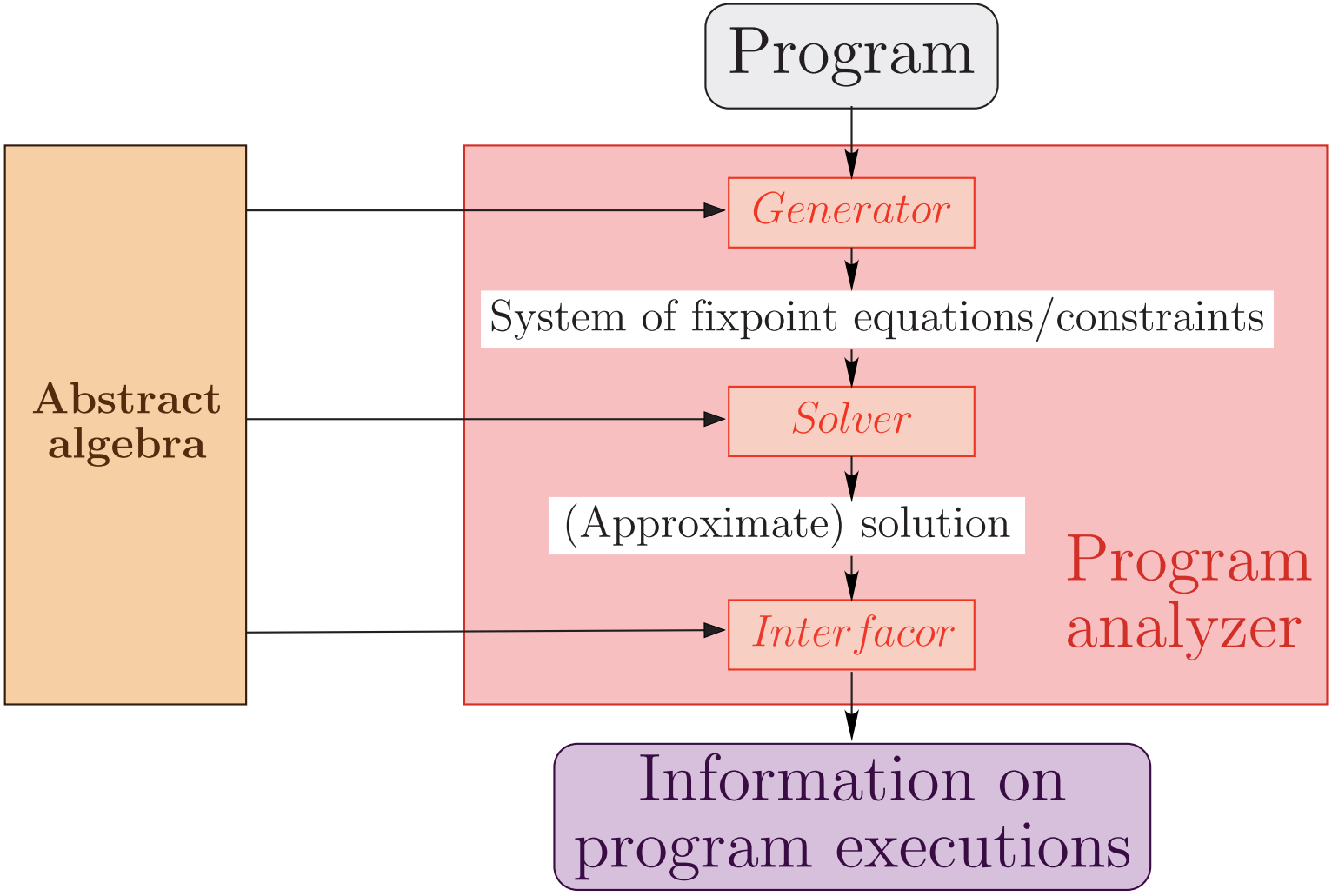
Objective of Abstract Static Program Analysis

- Program analysis is the automatic static determination of dynamic run-time properties of programs;
- The principle is to compute an approximate semantics of the program to check a given specification;
- *Abstract interpretation* is used to derive, from a standard semantics, the approximate and computable abstract semantics;
- This derivation is itself not (fully) mechanizable.

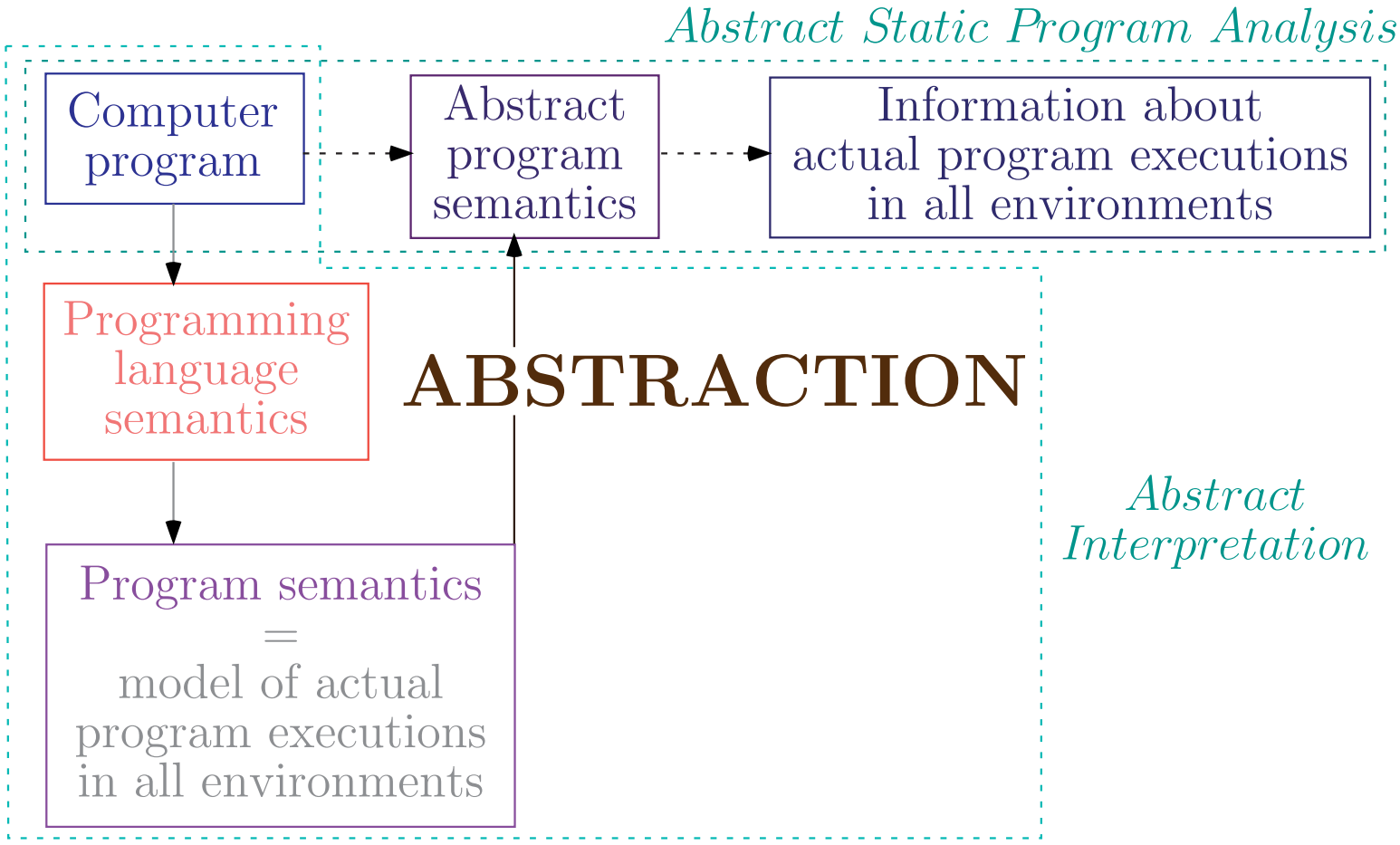
Principle of an Abstract Static Program Analyzer



Generic of an Abstract Static Program Analyzer



Design of a Static Program Analyzer by Abstract Interpretation



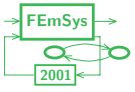
A Classical Example: Interval Analysis

Example: interval analysis (1975) [1]

Program to be analyzed:

```
   x := 1;
1:
   while x < 10000 do
2:
      x := x + 1
3:
   od;
4:
```

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.



Example: interval analysis (1975) [1]

Equations (abstract interpretation of the semantics):

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

The analyzer reads the program text and produces (a representation of) the above equations and then solve them iteratively. The equations are an abstraction of the trace semantics of the program. The formal derivation of the algorithm producing the equation by abstract interpretation of the program trace semantics is (mainly) manual.

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Constraints (abstract interpretation of the semantics):

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 \supseteq [1,1] \\ X_2 \supseteq (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 \supseteq X_2 \oplus [1,1] \\ X_4 \supseteq (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

The analyzer reads the program text and produces (a representation of) the above constraints and then solve them iteratively. The constraints are an abstraction of the trace semantics of the program. The formal derivation of the algorithm producing the constraints by abstract interpretation of the program trace semantics is (mainly) manual.

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration, initialization:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = \emptyset \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = \emptyset \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = \emptyset \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,1] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,2] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence?**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,2] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence??**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,3] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence???**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,3] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence????**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,4] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence?????**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,4] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence??????**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,5] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Increasing chaotic iteration: **convergence???????**

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,5] \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Convergence speed-up by extrapolation:

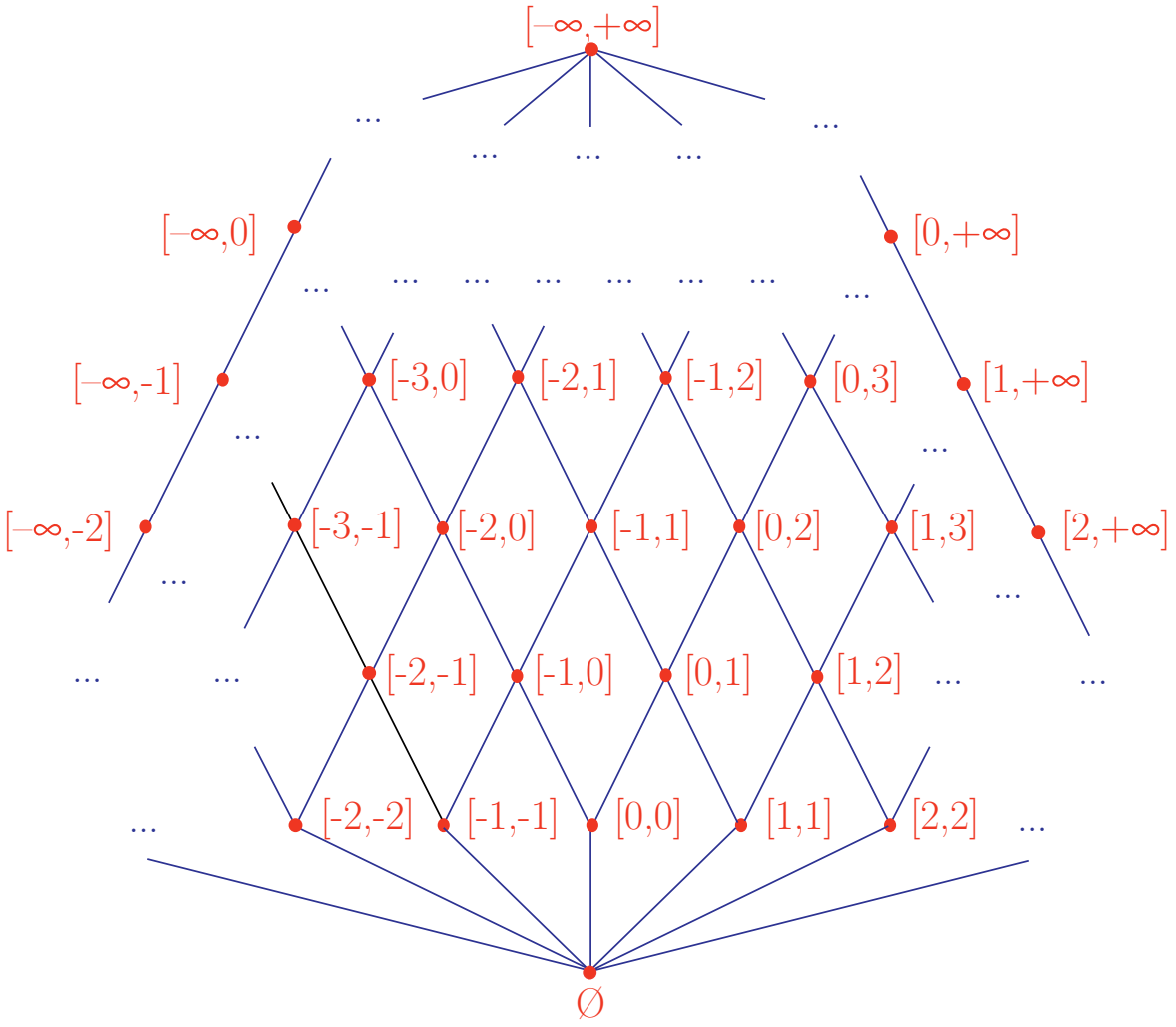
```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,+\infty] \quad \Leftarrow \text{widening} \\ X_3 = [2,6] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Widening



Example: interval analysis (1975) [1]

Decreasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1, +\infty] \\ X_3 = [2, +\infty] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Decreasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,9999] \\ X_3 = [2,+\infty] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Decreasing chaotic iteration:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,9999] \\ X_3 = [2,10000] \\ X_4 = \emptyset \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Final solution:

```
    x := 1;
1:
    while x < 10000 do
2:
        x := x + 1
3:
    od;
4:
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,9999] \\ X_3 = [2,10000] \\ X_4 = [10000,10000] \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Example: interval analysis (1975) [1]

Result of the interval analysis:

```
    x := 1;
1: {x = 1}
    while x < 10000 do
2: {x ∈ [1, 9999]}
        x := x + 1
3: {x ∈ [2, 10000]}
    od;
4: {x = 10000}
```

$$\begin{cases} X_1 = [1,1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1,1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$$

$$\begin{cases} X_1 = [1,1] \\ X_2 = [1,9999] \\ X_3 = [2,10000] \\ X_4 = [10000,10000] \end{cases}$$

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

A More Intriguing Example

```
program Variant_of_McCarthy_91_function;
    var X, Y : integer;
    function F(X : integer) : integer;
    begin
        if X > 100 then F := X - 10
        else F := F(F(F(F(F(F(F(F(F(X + 90)))))))));
    end;
begin
    readln(X);
    Y := F(X);
    { Y ∈ [91, +∞] }
end.
```

Reference

[21] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. FMPA*, LNCS 735, pages 128–141. Springer, 1993.

Probabilistic Program Analysis [3]

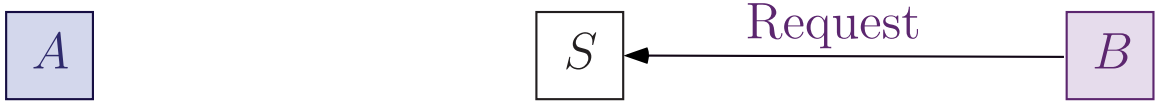
```
double x, i;
assume (-1.0 < x < 0.0);
i = 0.0;
while (i < 3.0) {
  x += uniform();
  i += 1.0;
};
assert (x < 1.0);
```

With 99% safety:

- the probability of the outcome $(x < 1)$ is less than 0.859,
- assuming:
  - worst-case nondeterministic choices of the precondition $(-1.0 < x < 0.0)$,
  - random choices `uniform()` chosen in $[0, 1]$ with the Lebesgue uniform distribution.

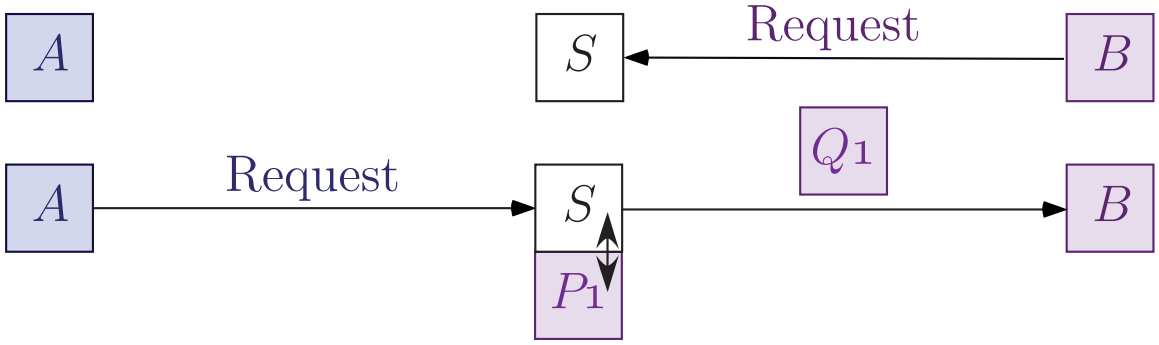
[1] D. Monniaux, SAS'00, POPL'01

Communication Topology of Mobile Processes [1]



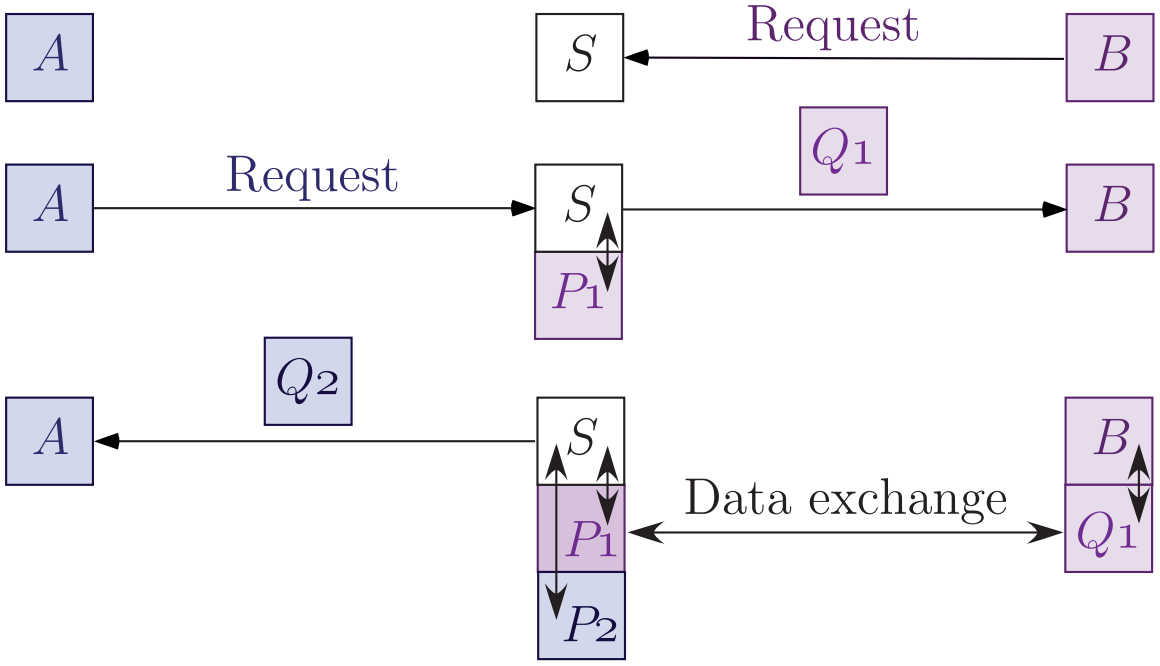
[1] J. Feret, SAS'00, ENTCS Vol. 39

Communication Topology of Mobile Processes[1]



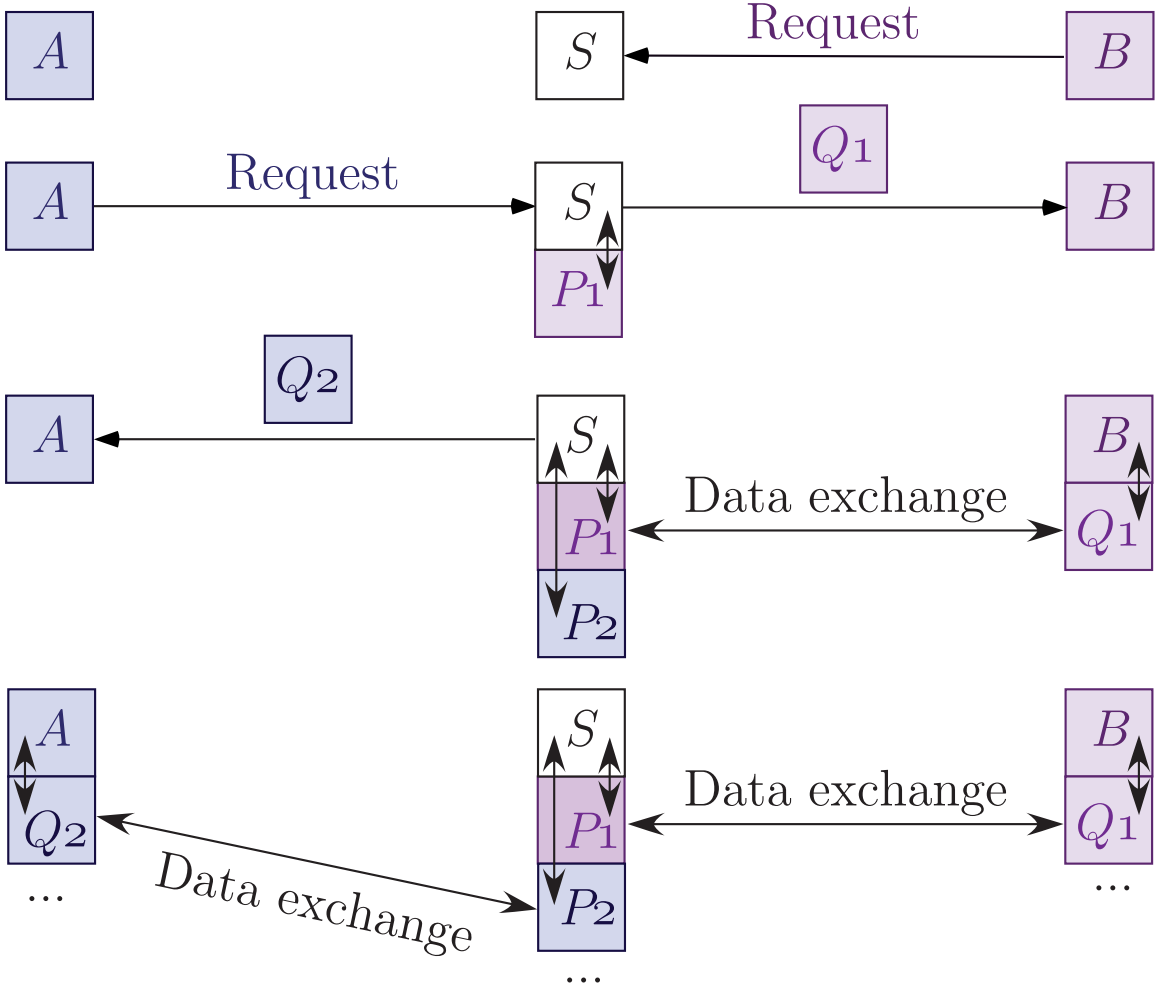
[1] J. Feret, SAS'00, ENTCS Vol. 39

Communication Topology of Mobile Processes[1]



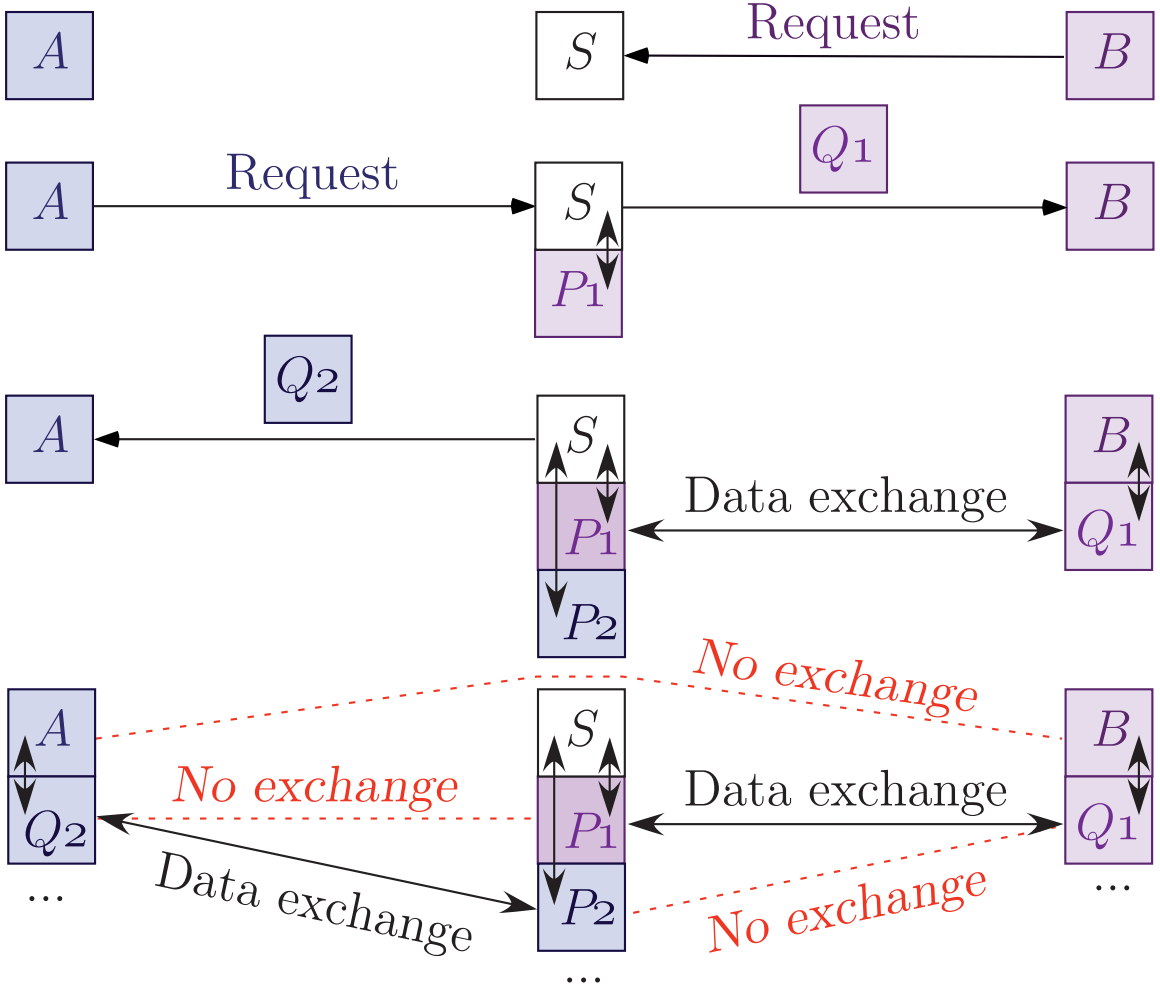
[1] J. Feret, SAS'00, ENTCS Vol. 39

Communication Topology of Mobile Processes[1]



[1] J. Feret, SAS'00, ENTCS Vol. 39

Communication Topology of Mobile Processes [1]



[1] J. Feret, SAS'00, ENTCS Vol. 39

Other Applications of Abstract Static Program Analysis

- Escape analysis for functional [22] and object oriented languages [23];
- Security analysis of cryptographic protocoles in hostile environments [24]; .../...

References

[22] B. Blanchet. – Escape analysis: correctness proof, implementation and experimental results. *25th POPL '98*, pp. 25–37. ACM, 1998

[23] B. Blanchet. – Escape Analysis for Object-Oriented Languages: Application to Java. *OOPSLA 99*, pp. 20–34. ACM, 1999.

[24] D. Monniaux. – Abstracting cryptographic protocols with tree automata. *SAS '99*. LNCS, pp. 149–163. Springer, 1999.

Other Applications of Abstract Static Program Analysis (Cont'd)

- Liveness proofs under weak fairness [25];
- Design of hierarchies of semantics of programming languages [26];
- Precision analysis of floating point computations;
- Software source watermarking/tatooing;
- Etc.

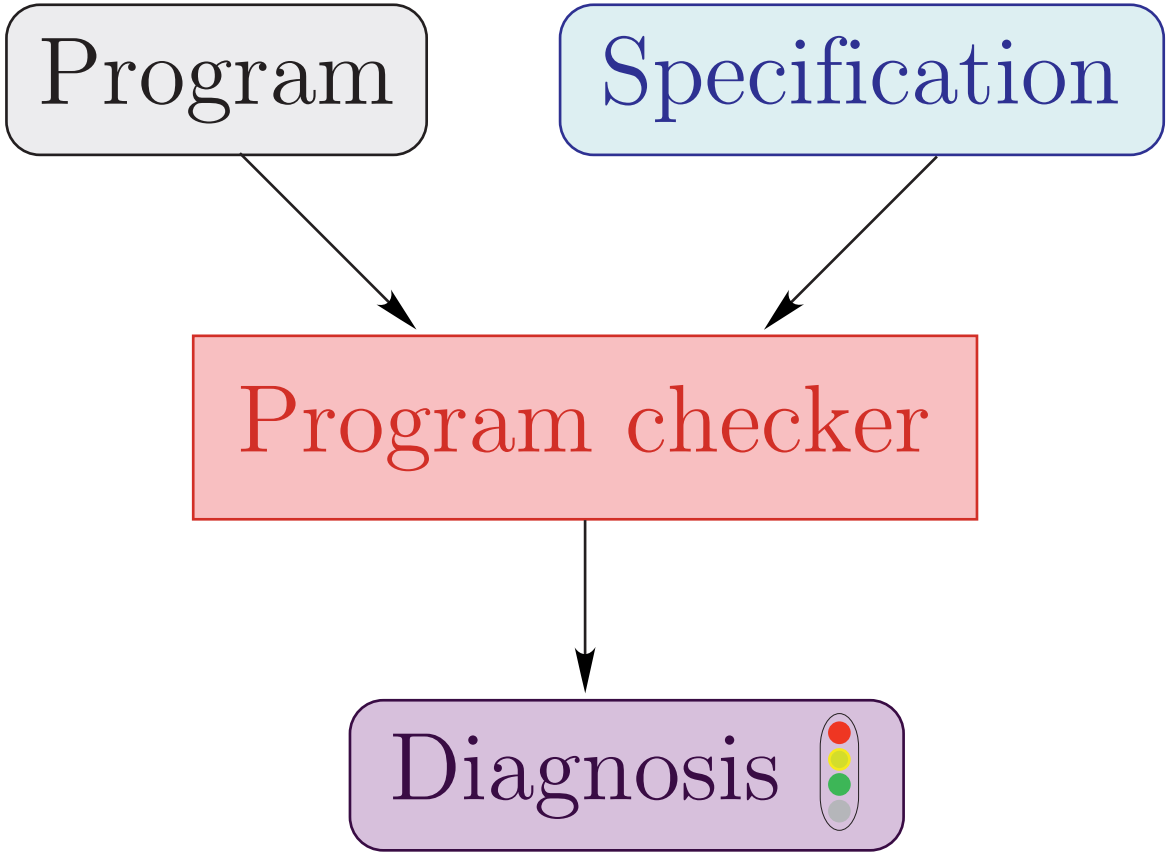
References

[25] L. Mauborgne. – Tree schemata and fair termination. *SAS '00*, LNCS 1824, pp. 302–320. Springer, 2000.

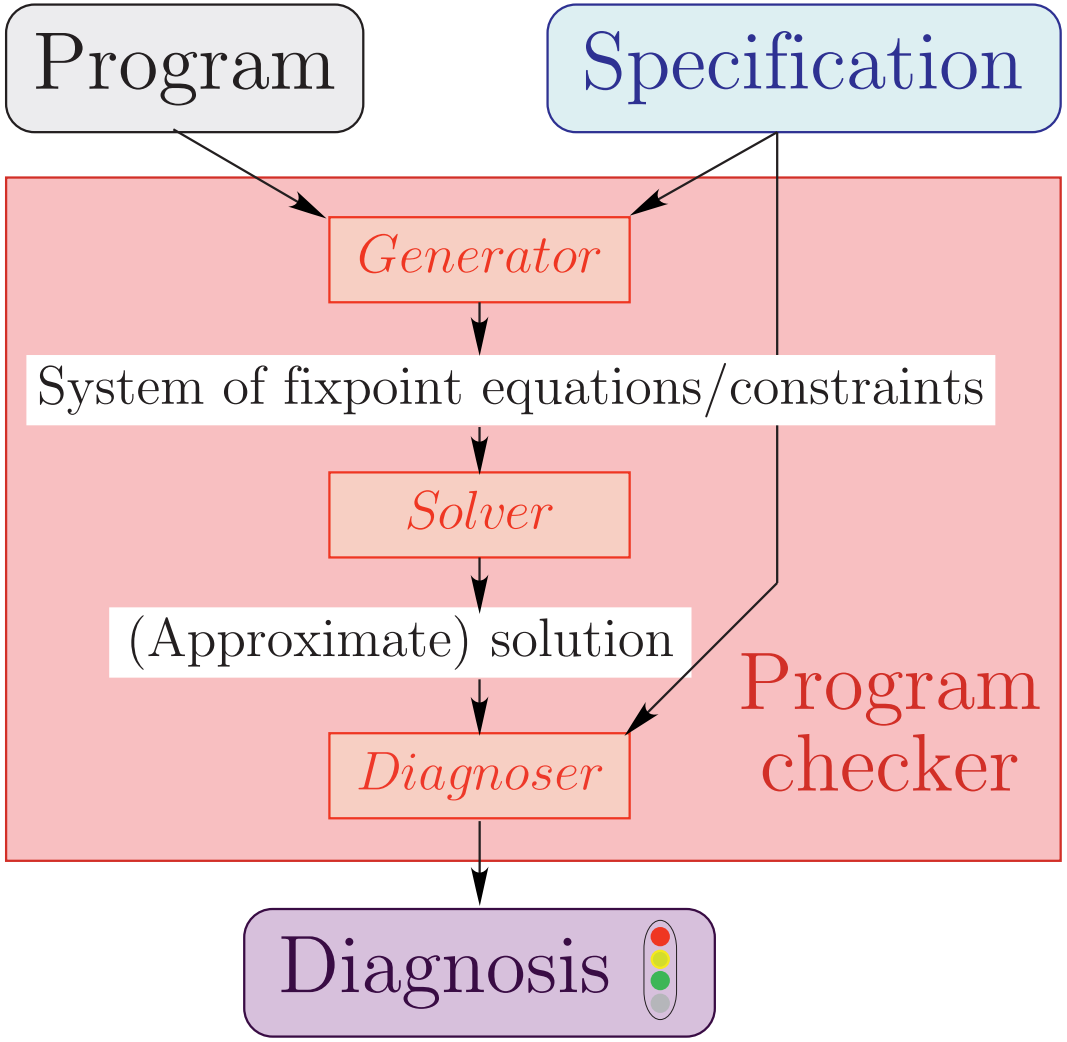
[26] P. Cousot. – Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, to appear in 2001.

Abstract Static Program Checking

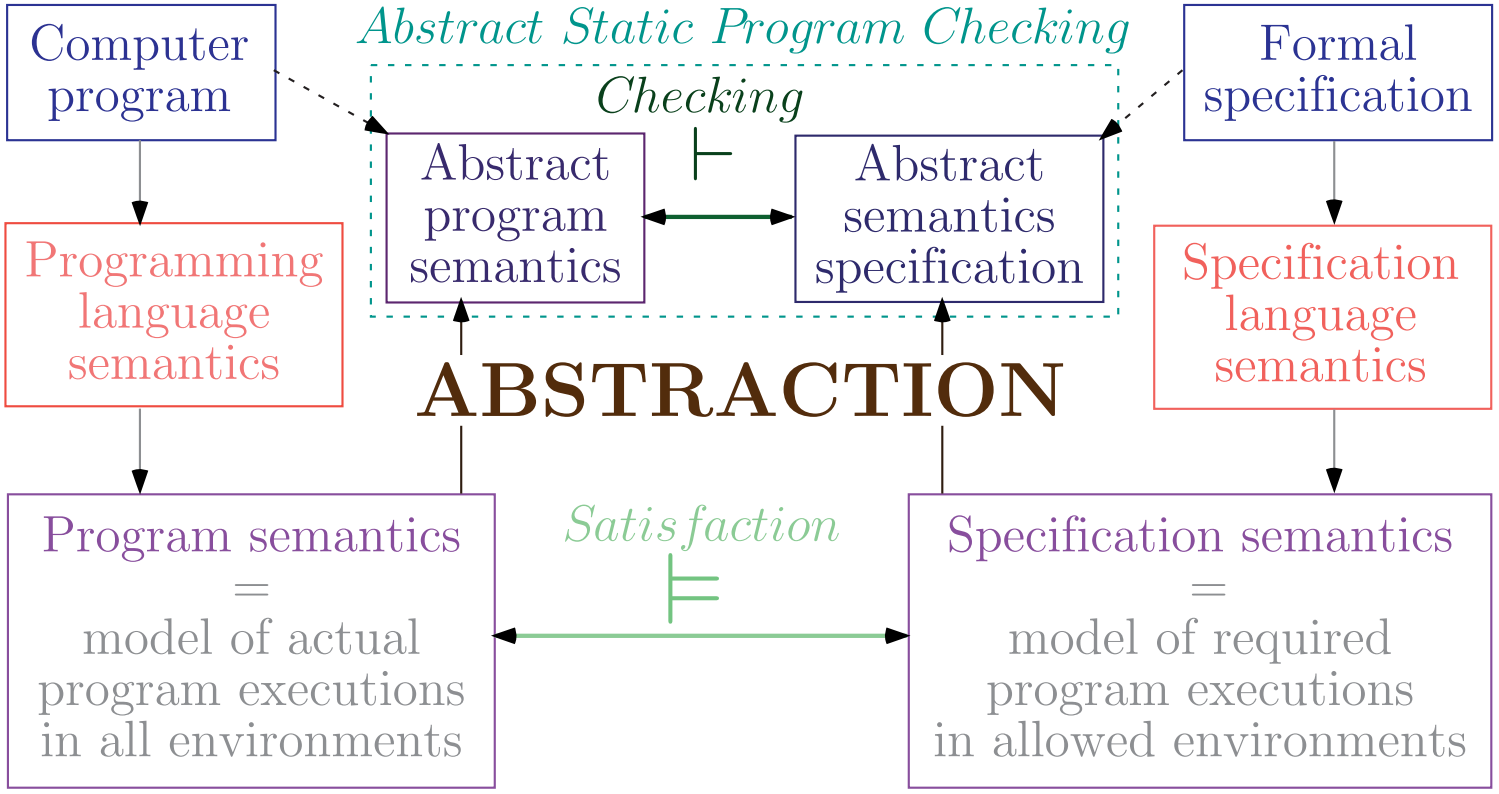
Objective of Abstract Static Program Checking



Principle of an Abstract Static Program Checker



Design of a Static Program Checker by Abstract Interpretation



Example: interval analysis (1975) [1]

Exploitation of the result of the interval analysis:

```
    x := 1;
1: {x = 1}
    while x < 10000 do
2: {x ∈ [1, 9999]}
        x := x + 1          ⟵ no overflow
3: {x ∈ [2, 10000]}
    od;
4: {x = 10000}
```

[1] P. Cousot & R. Cousot, ISOP'1976, POPL'77.

Other Examples of Faultless Execution Checks

- Absence of runtime errors (array bounds violations, arithmetic overflow, erroneous data accesses, etc.),
- Absence of memory leaks (dangling pointers, uninitialized variables, etc.),
- Handling of all possible runtime exceptions (failures of I/O and system calls, etc.),
- No resource contention and race conditions in concurrent programs (deadlocks & livelocks),
- Termination / non termination conditions,
- Etc.

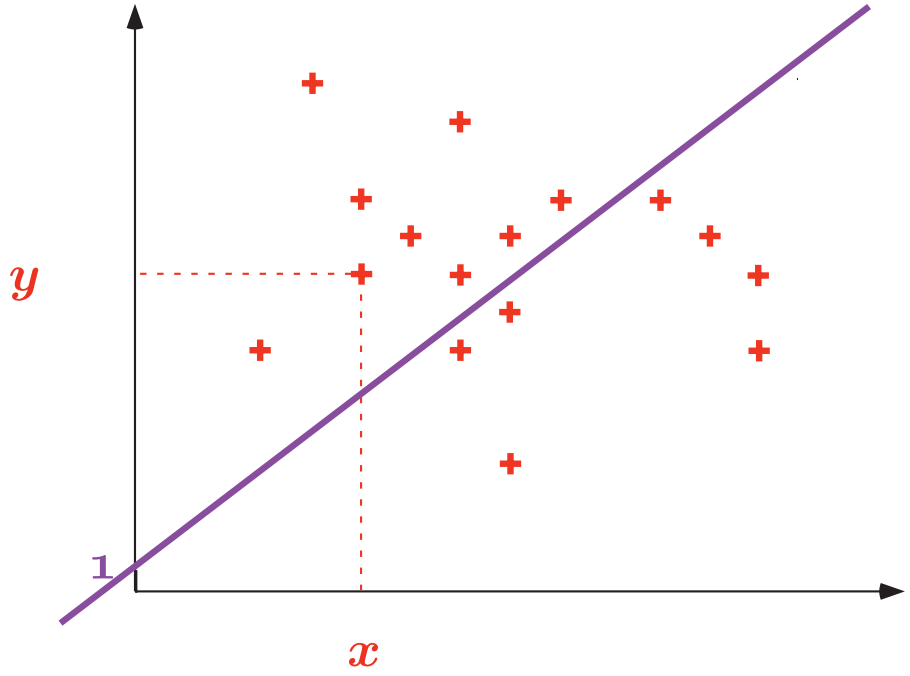
Information Loss

Information Loss

- All answers given by the abstract semantics are always correct with respect to the concrete semantics;
- Because of the information loss, not all questions can be definitely answered with the abstract semantics;
- The more concrete semantics can answer more questions;
- The more abstract semantics are more simple.

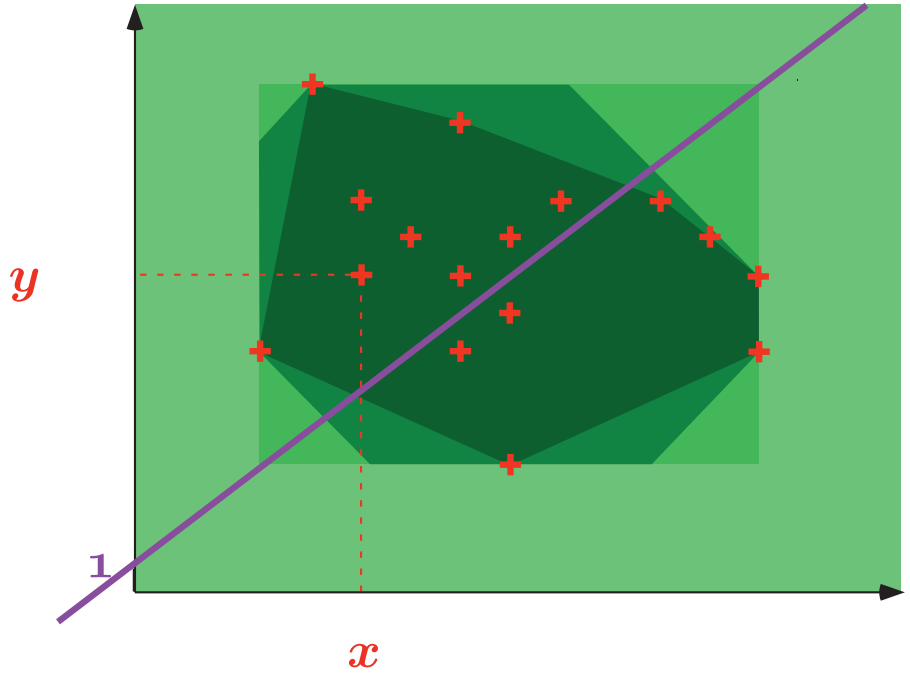
Example of Information Loss

- Is the operation `1/(x+1-y)` well defined at run-time?
- Concrete semantics: **yes**



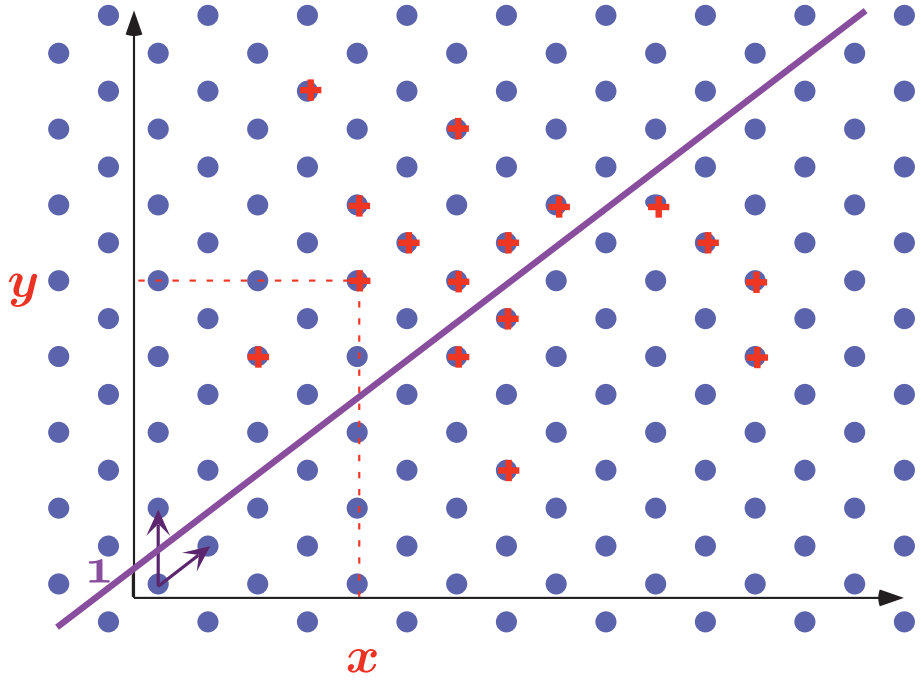
Example of Information Loss

- Is the operation `1/(x+1-y)` well defined at run-time?
- Abstract semantics 1: **I don't know**



Example of Information Loss

- Is the operation `1/(x+1-y)` well defined at run-time?
- Abstract semantics 2: **yes**



What Can We Do When Too Much Information Is Lost?

- Automatic inference/refinement of abstractions:
  - Mathematically, a most abstract sound and complete abstraction always exists for proving correctness;
  - Finding such a sound and complete abstraction is logically equivalent to the discovery of an inductive argument and checking of a proof obligation [27];
  - In practice: I have no hope!

Reference

[27] P. Cousot. Partial Completeness of Abstract Fixpoint Checking. In *Proc. 4th Int. Symp. SARA'2000*, LNAI 1864, pp. 1–25, Springer, 2000.

What Can We Do When Too Much Information Is Lost? (Cont'd)

- A modest alternative:
  - Interactive choice of abstractions within a predefined broad spectrum;
  - Manual decomposition of complex specifications into partial requirements;
  - User-interaction in the form of abstract testing.

Abstract Static Program Testing

Combining Empirical and Formal Methods

- The user provides local formal abstractions of the program specifications using predefined abstractions [2];
- The program is evaluated by abstract interpretation of the formal semantics of the program [3];
- If the local abstract specification cannot be proved correct, a more precise abstract domain must be considered [4];
- The process is repeated until appropriate coverage of the specification.

---

2 thus replacing infinitely many test data.

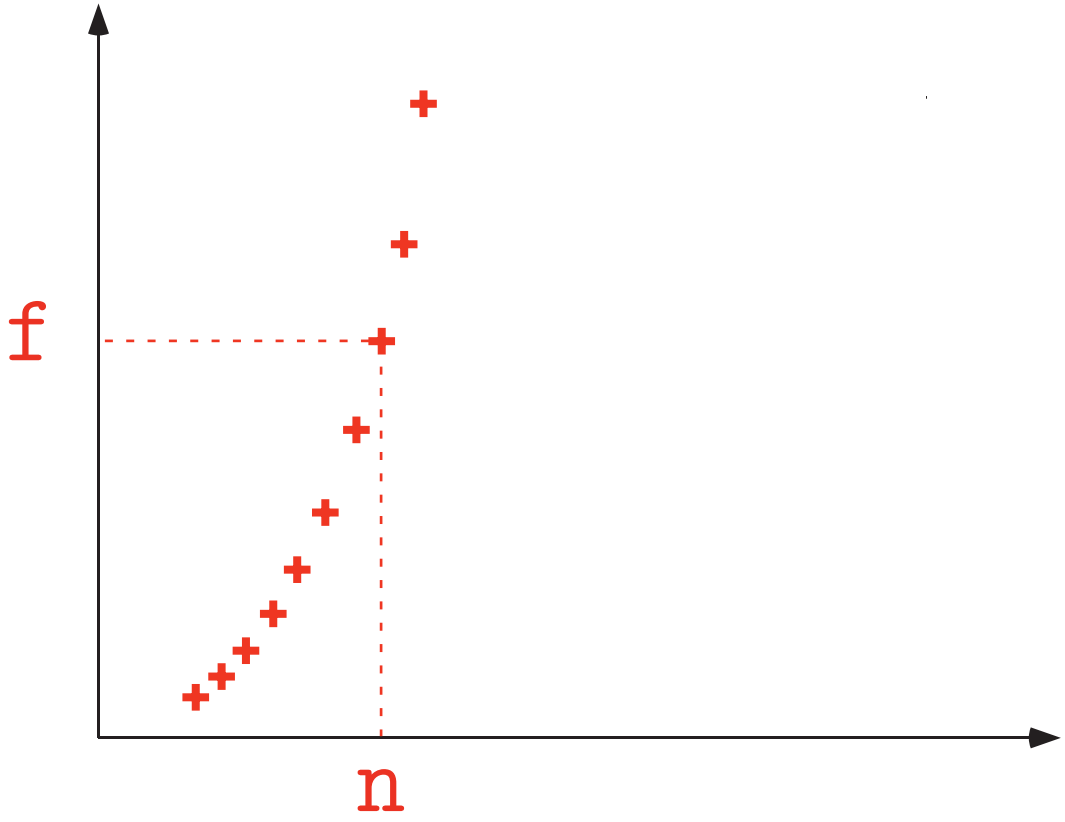
3 thus replacing program execution on the test data.

4 similarly to different test data.

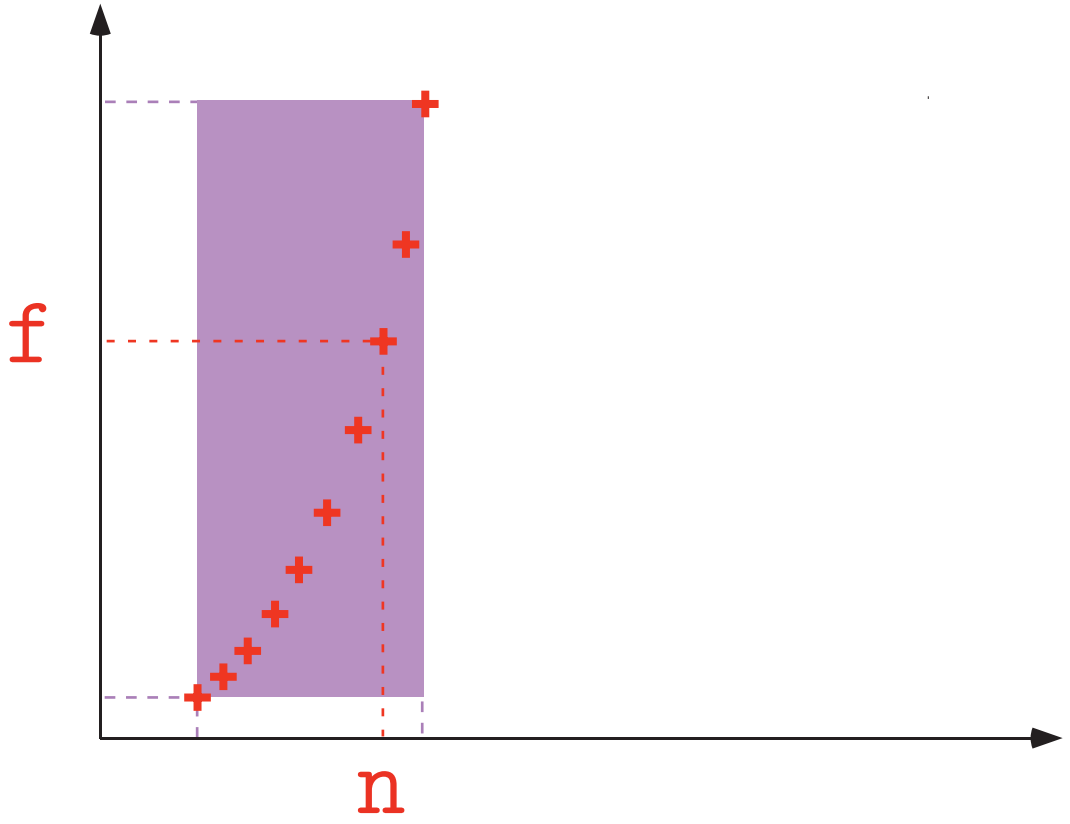
Abstract Program Testing

Debugging	Abstract testing
Run the program	Compute the abstract semantics
On test data	Choosing a predefined abstraction
Checking if all right	Checking user-provided abstract assertions
Providing more tests	With more refined abstractions
Until coverage	Until enough assertions proved or no predefined abstraction can do.

Example of predefined abstraction



Example of predefined abstraction: intervals



A Tiny Example

```
0: { n:[−∞,+∞]?; f:[−∞,+∞]? }
  read(n);
1: { n:[0,+∞]; f:[−∞,+∞]? }
  f := 1;
2: { n:[0,+∞]; f:[1,+∞] }
  while (n <> 0) do
    3: { n:[1,+∞]; f:[1,+∞] }
      f := (f * n);
    4: { n:[1,+∞]; f:[1,+∞] }
      n := (n - 1)
    5: { n:[0,+∞]; f:[1,+∞] }
  od;
6: { n:[0,0]; f:[1,+∞] }
sometime true;;
```

■ static analyzer inference

diagnosis: ■ definite error

■ no error

■ potential error

■ user program

■ user specification

A Tiny Example (Cont'd)

```
0: { n:⊥; f:⊥ }
  initial (n < 0);
1: { n:[−∞,-1]; f:[−∞,+∞]? }
  f := 1;
2: { n:[−∞,-1]; f:[−∞,+∞] }
  while (n <> 0) do
    3: { n:[−∞,-1]; f:[−∞,+∞] }
      f := (f * n);
    4: { n:[−∞,-1]; f:[−∞,+∞] }
      n := (n - 1)
    5: { n:[−∞,-2]; f:[−∞,+∞] }
  od
6: { n:⊥; f:⊥ }
```

- static analyzer inference
- user specification
- user program

diagnosis:
- no error
- potential error
- potential error
- ⊥ unreachable code

A More Intriguing Example

```
program Variant_of_McCarthy_91_function;
    var X, Y : integer;
    function F(X : integer) : integer;
    begin
        if X > 100 then F := X - 10    91
        else F := F(F(F(F(F(F(F(F(F(X + 90)))))))));
    end;
begin
    readln(X);
     {% X > 100 %}
    Y := F(X);
    {% sometime true %}

end.
```

Example of cycle: $\mathtt{F}(100) \rightarrow \mathtt{F}(190) \rightarrow \mathtt{F}(180) \rightarrow \mathtt{F}(170) \rightarrow \mathtt{F}(160) \rightarrow \mathtt{F}(150) \rightarrow \mathtt{F}(140) \rightarrow \mathtt{F}(130) \rightarrow \mathtt{F}(120) \rightarrow \mathtt{F}(110) \rightarrow \mathtt{F}(100) \rightarrow \ldots$

Comparing with program debugging

- Similarity: user interaction, on the source code;
- Essential differences:
  - user provided test data are replaced by abstract specifications;
  - evaluation of an abstract semantics instead of program execution/simulation;
  - one can prove the absence of (some categories of) bugs, not only their presence;
  - abstract evaluation can be forward and/or backward (reverse execution).

Comparing with abstract model-checking

- Similarities:
  - use of specifications instead of test data sets;
  - ability to automatically produce counter-examples [2];

[2] or specifications of infinitely many such counter-examples in the case of abstract program testing.

Comparing with abstract model-checking (cont'd)

- Essential differences:
  - reasoning on the concrete program (not on a program model);
  - no attempt to make a one-shot complete formal proof of the specification;
  - interaction with user repeatedly providing partial specifications in a form close to conventional debugging;
  - predefined abstractions (not user defined);
  - finite and infinite abstract domains are allowed.

Examples of Functional Specifications for Abstract Testing

- Worst-case execution/response time in real-time systems running on a computer with pipelines and caches;
- Periodicity of some action over time/with respect to some clock;
- Possible reactions to real-time event/message sequences;
- Compatibility with state/transition/sequence diagrams/charts;
- Absence of deadlock/livelock with different scheduling policies;

Conclusion

Concluding Remarks

- Program debugging is still the prominent industrial program "verification" method. Complementary program verification methods are needed;
- Fully mechanized program verification by formal methods is either impossible (e.g. typing/program analysis) or extremely costly since it ultimately requires user interaction (e.g. abstract model checking/deductive methods for large programs);
- For program verification, semantic abstraction is mandatory but difficult whence hardly automatizable, even with the help of programmers;

Concluding Suggestions

- Abstract interpretation introduces the idea of safe approximation within formal methods;
- So you might think to use it for partial verification of the source specification/program code:
  - Abstract checking (fully automatic and exhaustive diagnosis on run-time safety properties),
  - Abstract testing (interactive/planned diagnosis on functional, behavioural and resources-usage requirements),

  using tools providing predefined abstractions. .../...

- Does apply to any computer-related language with a well-specified semantics describing computations (e.g. specification languages, data base languages, sequential, concurrent, distributed, mobile, logical, functional, object oriented, ... programming languages, etc.);
- Does apply to any property and combinations of properties (such as safety, liveness, timing, event preconditions, ...);
- Can follow up program modifications over time;
- Very cost effective, especially in early phases of program development (see e.g. next talk).

Industrialization of Static Analysis/Checking by Abstract Interpretation

-  Connected Components Corporation (U.S.A.), L. Harrison, 1993 [3];
-  AbsInt Angewandte Informatik GmbH (Germany), R. Wilhelm & C. Ferdinand, 1998;
- PolySpace TECHNOLOGIES Polyspace Technologies (France), A. Deutsch & D. Pilaud, 1999.

[3] Internal use for compiler design.

DAEDALUS European project on the verification of critical real-time avionic software (oct. 2000 — sep. 2002):

- P. Cousot (ENS, France), scientific coordinator;
- R. Cousot (École polytechnique, France);
- A. Deutsch & D. Pilaud (Polyspace Technologies, France);
- C. Ferdinand (AbsInt, Germany);
- É. Goubault (CEA, France);
- N. Jones (DIKU, Denmark);
- F. Randimbivololona & J. Souyris (EADS Airbus, France), coord.;
- M. Sagiv (Univ. Tel Aviv, Israel);
- H. Seidel (Univ. Trier, Germany);
- R. Wilhelm (Univ. Sarrebrücken, Germany);

A reference (with a large bibliography)

P. Cousot.

Abstract interpretation based formal methods and future challenges.

In R. Wilhelm (editor), « *Informatics — 10 Years Back, 10 Years Ahead* ».

Volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

An extended electroning version is also available on Springer-Verlag web site together with a very long electroning version with a complete bibliography.

THE END