Scaling up in static verification by abstract interpretation

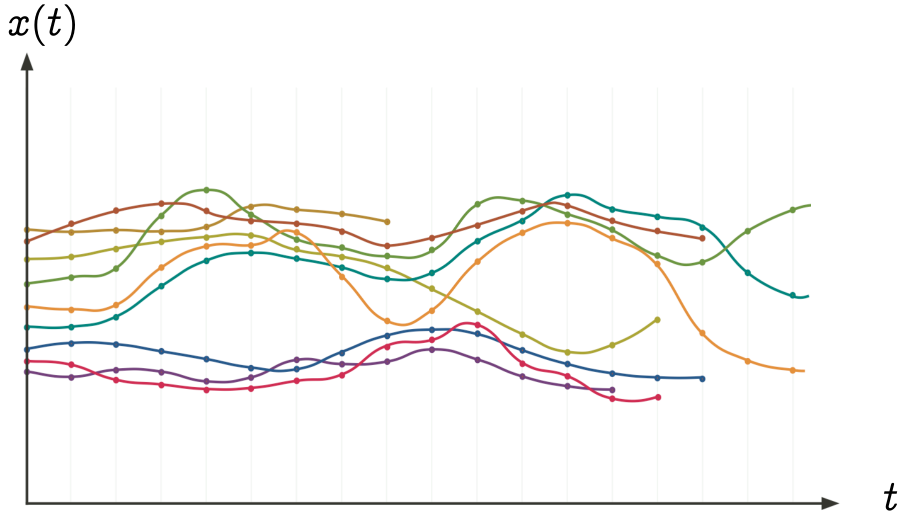
Patrick Cousot

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

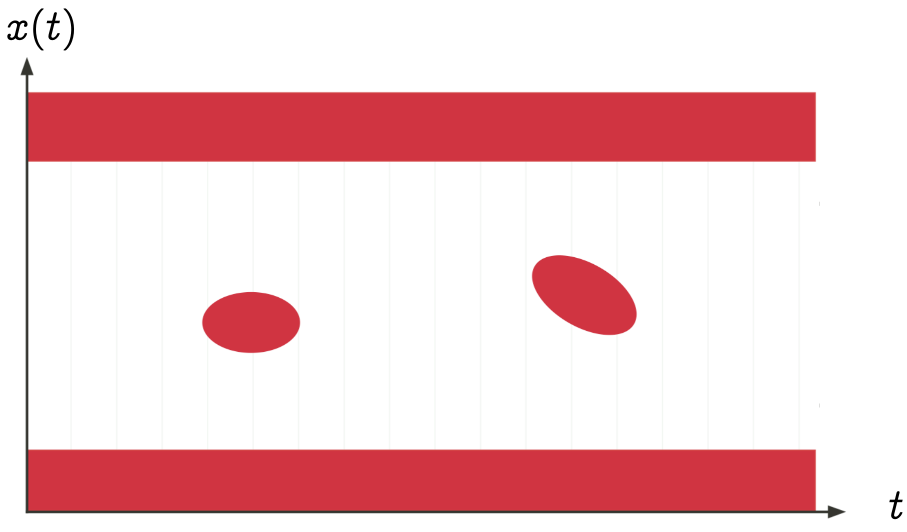
October 7, 2009

A very short intuitive and informal introduction to abstract interpretation

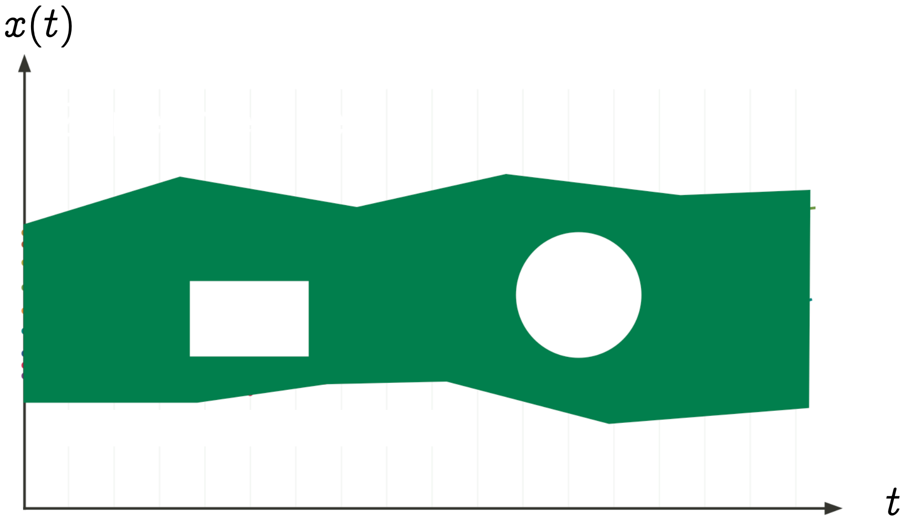
Semantics



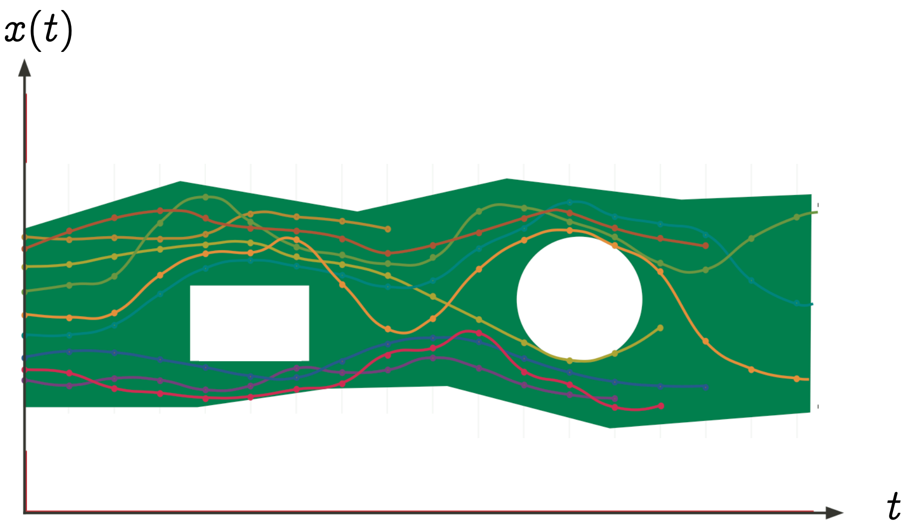
Safety specification



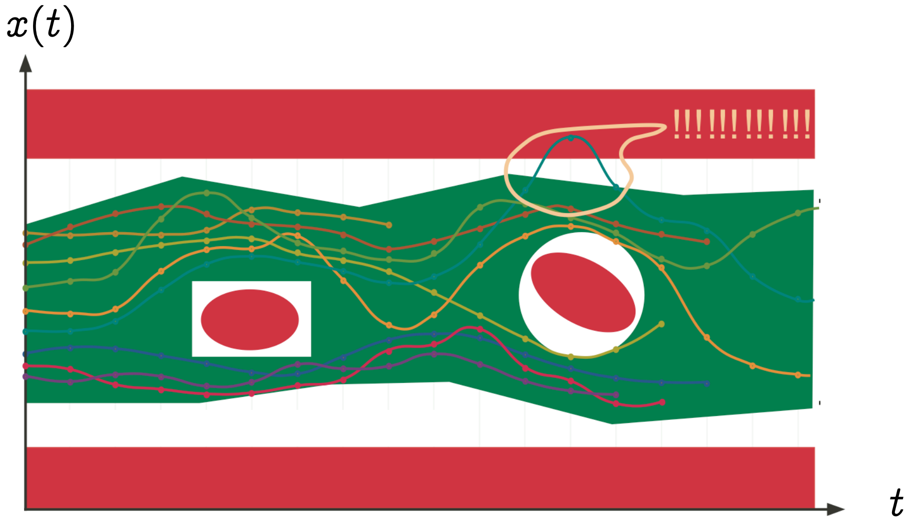
Abstraction



Soundness of the abstraction

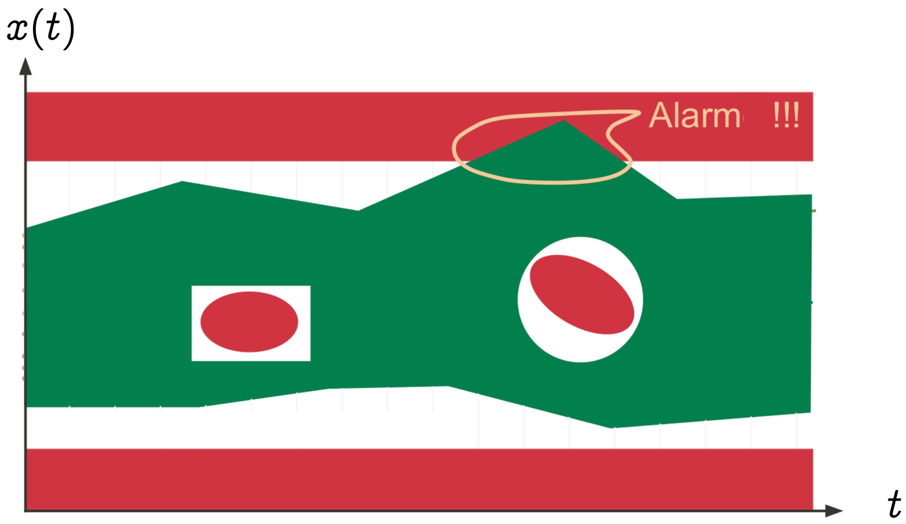


Example of unsound abstraction [(1)]

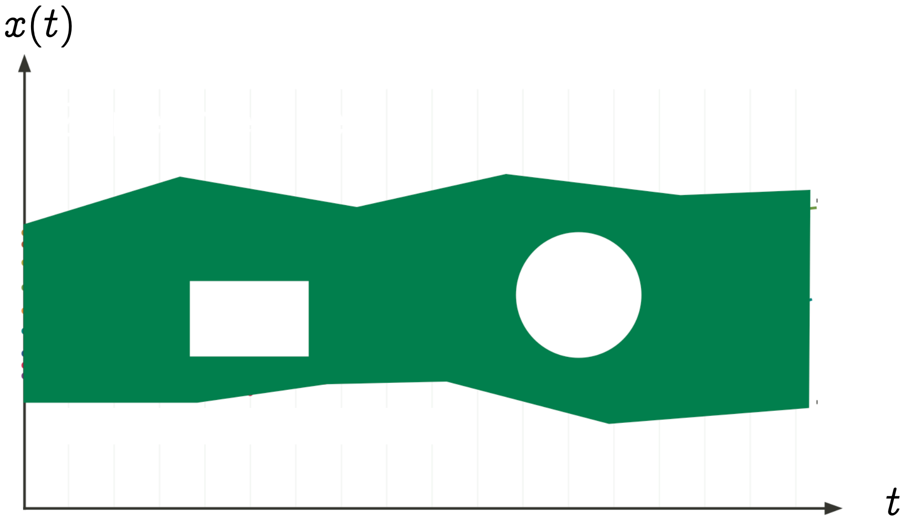


(1) excluded by abstract interpretation theory

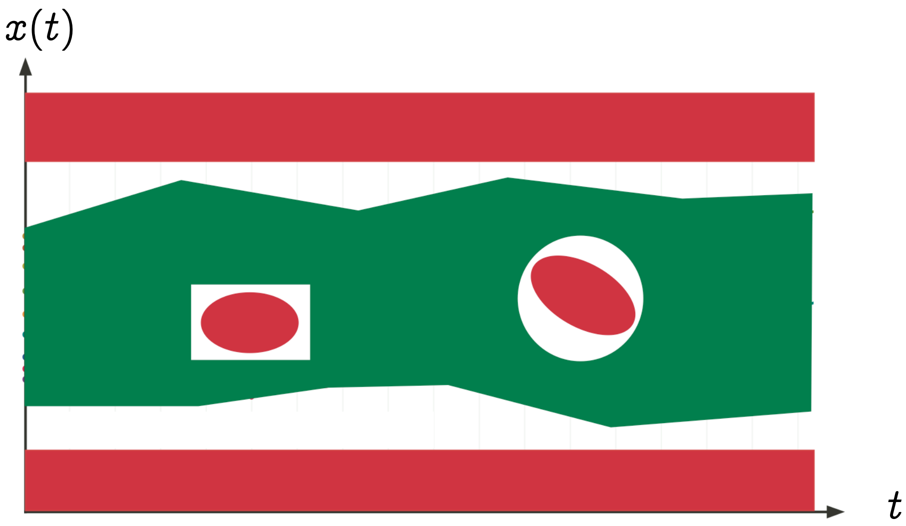
True or false alarm?



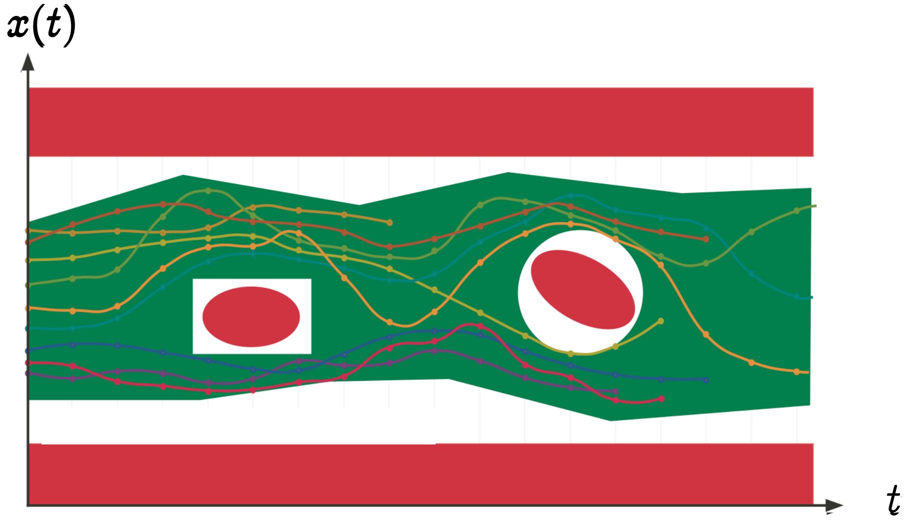
Refinement



Formal proof in the abstract



Formal proof in the abstract is sound



Example: inference of an invariant

$\{y \geqslant 0\}$ $\leftarrow$ hypothesis
`x := y`
$\{I(x,y)\}$ $\leftarrow$ loop invariant
`while` $(\texttt{x} > 0)$ `do`
  `x := x - 1;`
`od`

Abstract fixpoint equation:

$$I(x,y) = x \geqslant 0 \wedge (x = y \vee I(x+1,y)) \qquad (i.e.\ I = \mathbf{F}^\sharp(I)^{(1)})$$

Equivalent Floyd-Naur-Hoare verification conditions:

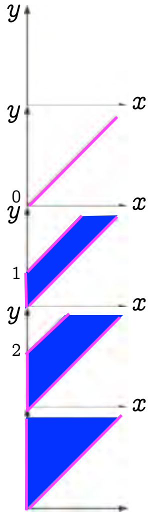
$$(y \geqslant 0 \wedge x = y) \Longrightarrow I(x,y) \qquad \textit{initialisation}$$
$$(I(x,y) \wedge x > 0 \wedge x' = x - 1) \Longrightarrow I(x',y) \qquad \textit{iteration}$$

(1) We look for the most precise invariant I, implying all others, that is $\mathsf{lfp}^{\Rightarrow}\ \mathbf{F}^\sharp$.

Iterates with widening

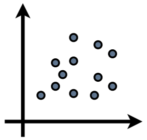
$$\boxed{I = \bigsqcup_{n\to\infty} \mathbf{F}^{\sharp n}(\text{false})}$$

$$\begin{aligned}
I^0(x,y) &= \text{false}\\
I^1(x,y) &= x \geqslant 0 \wedge (x = y \vee I^0(x+1,y))\\
&= 0 \leqslant x = y\\
I^2(x,y) &= x \geqslant 0 \wedge (x = y \vee I^1(x+1,y))\\
&= 0 \leqslant x \leqslant y \leqslant x+1\\
I^3(x,y) &= x \geqslant 0 \wedge (x = y \vee I^2(x+1,y))\\
&= 0 \leqslant x \leqslant y \leqslant x+2\\
I^4(x,y) &= I^2(x,y) \mathbin{\nabla} I^3(x,y) \leftarrow \text{widening}\\
&= 0 \leqslant x \leqslant y\\
I^5(x,y) &= x \geqslant 0 \wedge (x = y \vee I^4(x+1,y))\\
&= I^4(x,y) \quad \text{fixed point!}
\end{aligned}$$

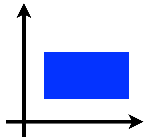


The invariants are computer representable with octagons!

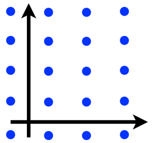
Examples of abstractions used by Astrée



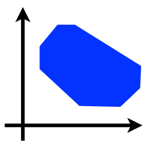
semantics
set of points



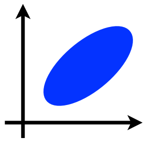
intervals
$x \in [a, b]$



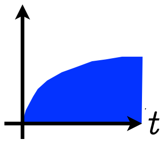
simple congruences
$x \equiv a[b]$



octagons
$\pm x \pm y \leqslant a$

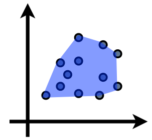


ellipsoids
$ax^2 + by^2 + cxy \leqslant d$

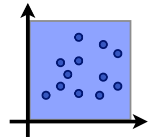


exponentials
$x(t) \leqslant a^{bt}$

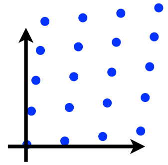
Examples of abstractions not used by Astrée



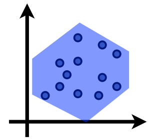
Polyhedra (too expensive)



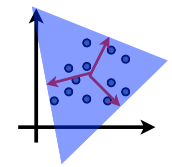
Signs (too imprecise)



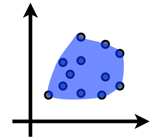
Linear congruences (too expensive)



Zonotopes (inclusion?)



Support functions (widening?)



Convex sets (algorithmics?)

Example of analysis by Astrée

```
% cat retro.c
typedef enum {FALSE=0, TRUE=1} BOOL;
BOOL FIRST;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;

void dev( )
{ X=E;
  if (FIRST) { P = X; }
  else
    { P =  (P - ((((2.0 * P) - A) - B)
              * 5.0e-03)); };
  B = A;
  if (SWITCH) {A = P;}
  else {A = X;}
}
```

```
void main()
{ FIRST = TRUE;
  while (TRUE) {
    dev( );
    FIRST = FALSE;
    __ASTREE_wait_for_clock(());
  }}
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));

astree -exec-fn main -config-sem retro.config
retro.c |& grep "|P|" | tail -n 1
|P| <=1.0000002*((15.  +
5.8774718e-39/(1.0000002-1))*(1.0000002)^clock -
5.8774718e-39/(1.0000002-1)) + 5.8774718e-39 <=
23.039353
```

Scaling up in static verification by abstract interpretation

Scalability in verification

Many different measures of scalability of an automatic verifier:

- size scalability : size of the systems to be verified
- design scalability : required efforts from the verifier designer
- use scalability : required efforts from the verifier end-user
- re-use scalability : reproducibility under small changes (maintenance)

Scalability → can only be measured with respect to well-specified and often incomparable objectives

Implicit versus explicit specifications

Implicit specifications:

- e.g. absence of runtime errors
- no extra burden on the end-user, cost effective

Explicit specifications:

- e.g. temporal specification
- available specification may not be exploitable (Simulink model)
- additional cost to maintain both the system, its specification and a formal specification

Size scalability through specification restriction:

- e.g. verification → bug-finding

Universal versus domain specific systems

Universal systems:

- e.g. programs in a universal programming language

Domain specific systems:

- e.g. synchronous real-time control command programs
- excludes
  - recursive function calls,
  - dynamic memory allocations,
  - backward gotos,
  - long jumps,
  - concurrency,
  - very complicated data structures,
  - or highly nested loops.

Size scalability through application domain restriction

Precision versus incompleteness

Precise abstractions:

- always exist for a given system (but not an infinite family of systems)
- very hard to find

General abstractions:

- universal use (e.g. intervals)
- hard to avoid false alarms

Size-scalability through abstraction → how to get rid of false alarms?

System modelization

Abstract model generation:

- System ⟶ Model
  e.g. C program — transition system

Abstract model generator:

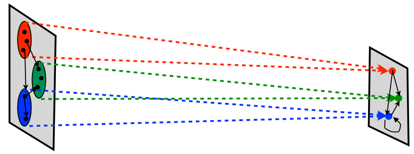
- System generator ⟶ Model generator
  e.g. C language — C program ⟶ abstract model

Use-scalability through automatic model generation → more burden is put on the static analyzer designer

Models versus properties (cont'd)

Model abstraction:

- Concrete model ⟶ Abstract model
  e.g. transition system — abstract transition system



The same tools can be used in the concrete and the abstract

.../...

Models versus properties (cont'd)

Property abstraction:

- Concrete properties generator — Abstract properties generator
  e.g. trace collecting semantics — polyhedral analysis

Use-scalability through more general abstractions → more burden is put on the static analyzer designer

Universal versus domain specific abstractions

Universal abstractions:

- e.g. invariance
- hard to automatize

Domain-specific abstractions:

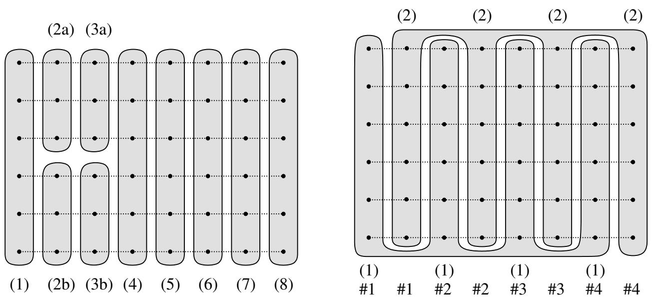
- e.g. filters, integrators in system control
- domain-specific knowledge can be incorporated in the verification process

Scalability by domain-specific abstractions → helpful (essential?), more burden on the static analyzer designer

Global versus local abstractions

Global abstractions:

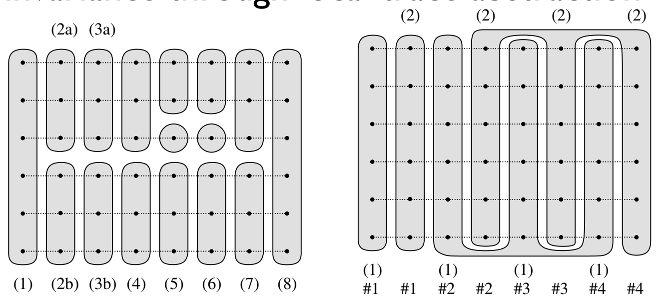
- e.g. invariance through reachability
- the same abstraction everywhere in time and space
- provably sound and complete (Cook proved relative completeness of Hoare logic)



Global versus local abstractions (cont'd)

Local abstractions:

- e.g. invariance through local trace abstraction



Scalability through local abstraction → much more powerful in the abstract, more burden of the static analyzer designer

Finite versus infinite abstractions

Finite abstractions:

- Universal representations/algorithms
- No approximation in the abstract

Infinite abstractions:

- No universal representations/algorithms
- Require further approximation in the abstract to ensure termination (widening/narrowing)

Equivalent for the static analysis of a given system, provably more powerful for an infinite family of systems

Scalability through infinite abstractions → more burden of the static analyzer designer

Unique versus multiple abstraction encoding

Universal representation of abstractions:

- e.g. terms for provers, BDDs, etc

Abstraction-specific representations:

- e.g. no efficient universal representation of geometric objects

Scalability through multiple abstraction representations → more burden on the analyzer designer but very efficient algorithms require specific data structures

Handling disjunction with multiple abstractions

Unique abstraction representation:

- embeds conjunction

Multiple abstraction-specific representations:

- conjunction of multiple abstractions with different abstractions is not immediate
- reduced product (optimal in precision)
- partial reduction is cheaper (although sub-optimal)

e.g. Astrée:

- fixed reduction order among abstract domains
- some domains broadcast information to all domains
- some domains ask questions to others, in turn

Refinement

- Refinement is necessary (very hard to find good abstractions)
- Automatic abstraction-refinement is costly and may not terminate
- Manual refinement by end-user &| static analyzer designer

e.g. in Astrée:

- Parameterized abstract domains
- Parameterized widenings
- (Automated) analysis directives
- Local improvements of the abstract predicate transformers
- Inclusion of new abstract domains

Scalability through manual refinement → more burden of the analyzer designer (& user)

Conclusion

- Finding the appropriate abstraction is undecidable and extremely difficult
- Universal/all purpose tools with automatic refinement only have the program as guidelines for automatic refinement, this is hardly sufficient (e.g. linear program → ellipsoïdal abstraction for programs)
- Human guided parametrization/directives/choice of combinations of domain specific abstractions is an alternative which proved successfull but ...
- Ultimately new abstractions (hence computer property representations & manipulation algorithms) must be designed manually but are reusable