Static Program Analysis for Verification - an Introduction -

Reinhard Wilhelm

Universität des Saarlandes

UNIVERSITÄT
DES
SAARLANDES

Reinhard Wilhelm

- studied math, physics and mathematical logic at Westfälische Wilhelms-Universität Münster and Informatics at Technische Universität München and Stanford University
- 1977 Dr. rer. nat TU Munich
- 1978 - professor at Saarland University
- 1990 – Scientific Director of the Leibniz Center for Informatics at Schloss Dagstuhl
- 1998 cofounder of AbsInt
- 2000 Fellow of the ACM
- 2006 Alwin-Walther medal from TU Darmstadt and Fraunhofer-Institut für Graphische Datenverarbeitung
- 2007 Prix Gay-Lussac-Humboldt from the French Minister of Education and Research
- 2008 Member of the European Academy of Sciences (Academia Europaea)
- 2008 Honorary doctorates from RWTH Aachen and Tartu University
- 2009 Konrad Zuse Medal from Gesellschaft für Informatik
- 2010 Bundesverdienstkreuz am Bande
- 2010 ACM Distinguished Service Award
- 2013 Member of the German National Academy of Sciences Leopoldina

Verification

Given the task to verify some **correctness statement** about a program

- the **functional correctness** of a program,
- the absence of **run-time errors**,
- the satisfaction of **space constraints**, or
- the satisfaction of **timing constraints**

non-functional correctness properties

Verification is used here in a strong sense!

We look for **guarantees**.

No doubts about verified correctness claims!

- What are the alternatives?
- What are you doing in practice?
- What would you like to have available?

A Short History of Static Program Analysis

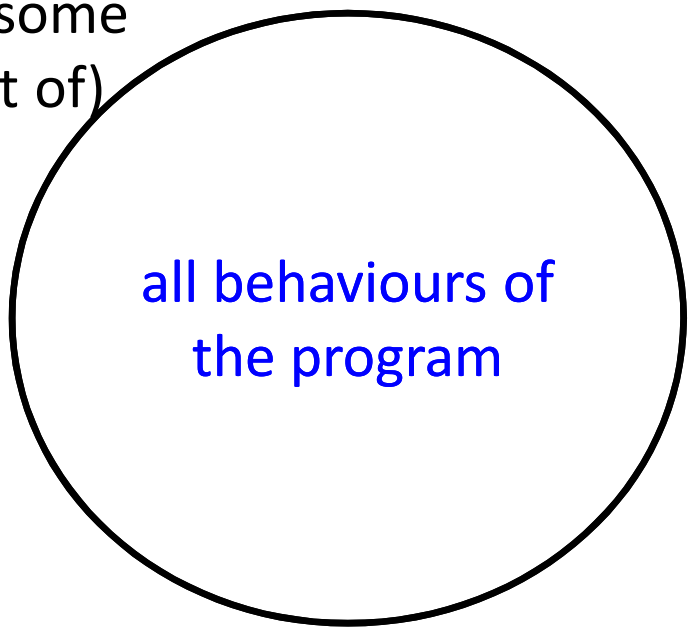
- Early high-level programming languages were implemented on very small and very slow machines
- Compilers needed to generate executables that were extremely efficient in space and time
- Compiler writers invented efficiency-increasing program transformations, wrongly called **optimizing transformations**
- Transformations must **not change the semantics of programs**
  - **Enabling conditions** guaranteed semantics preservation
  - Enabling conditions were checked by static analysis of programs (data-flow analysis)

Theoretical Foundations of Static Program Analysis

- Theoretical foundations for the solution of recursive equations: Kleene (1930s), Tarski (1955)
- Gary Kildall (1972) clarified the lattice-theoretic foundation of data-flow analysis.
- Patrick Cousot (1974) established the relation to the programming-language semantics.

Approaches to Functional Verification

Starting Point:
Correctness statements
concern all (or some specified subset of) behaviours of the program

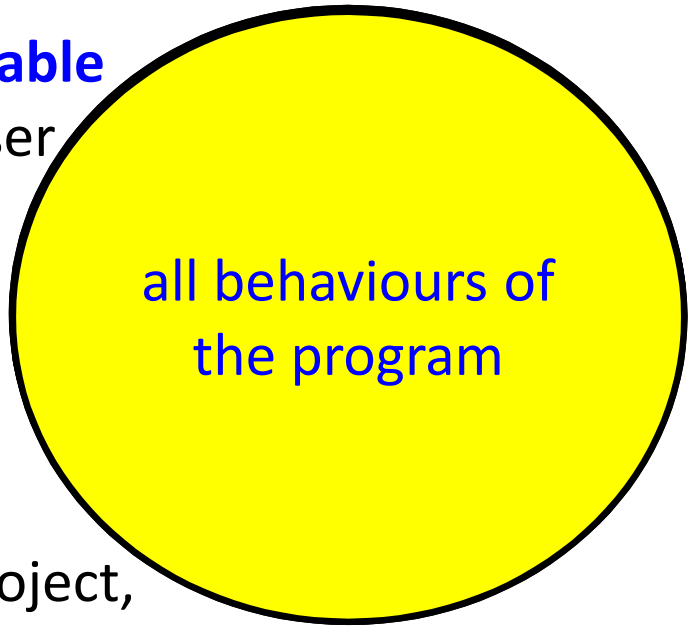


Deductive Verification

Functional correctness proved by **Theorem Proving**

Problem:

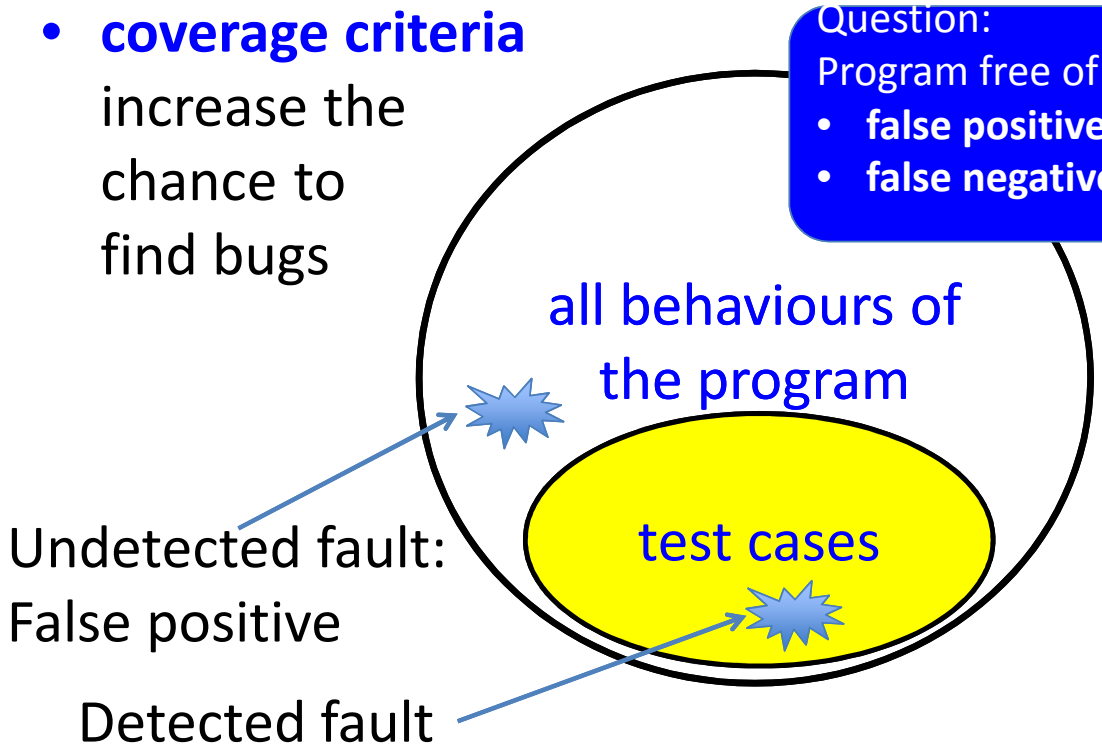
- **not automatable**
- non-trivial user interaction
- realistic only in **academic** setting
- Examples: CompCert project, verified C compiler, Verisoft, great effort by highly skilled academics



Testing

(for functional and non-functional properties)

- Test the program on a number of inputs
- **coverage criteria** increase the chance to find bugs



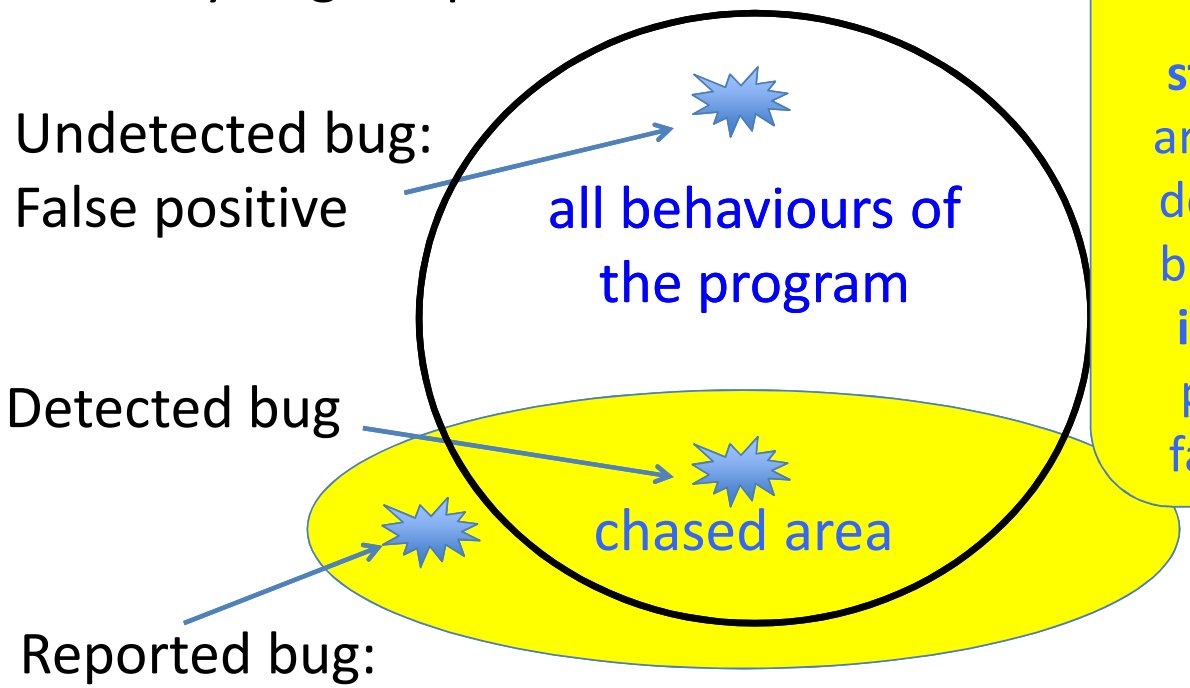
Question:
Program free of a certain type of error?
- **false positive**: answer wrongly "Yes"
- **false negative**: answer wrongly "No"

Program testing can be used to show the presence of bugs, but never to show their absence!
Dijkstra (1970)

Bug Chasing

Question: Is program free of bugs?

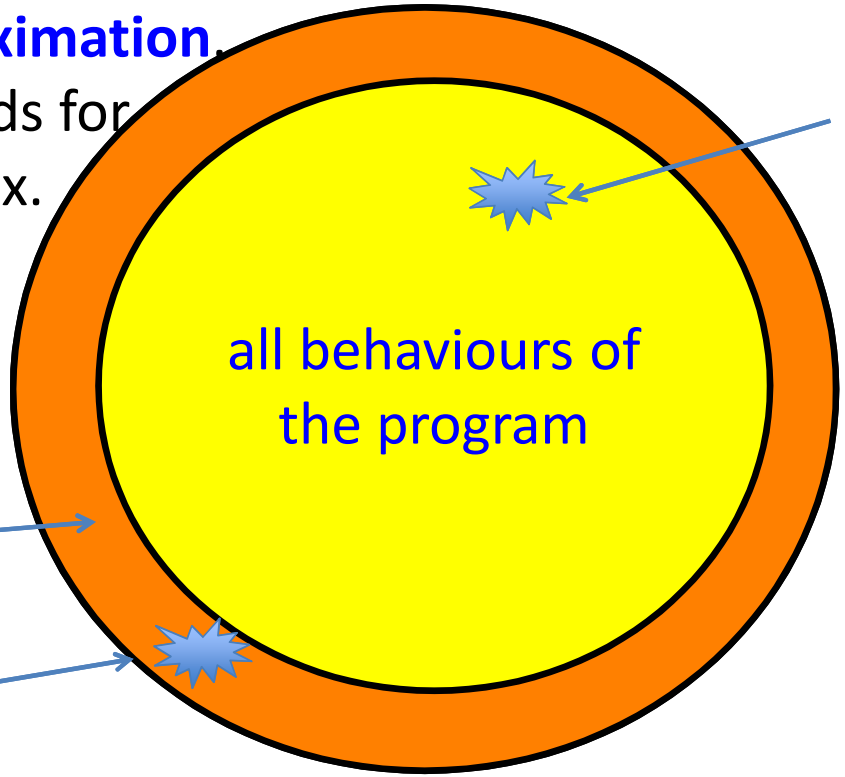
Attempt to (quickly) find
as many bugs as possible



Bug chasing tools, often called **static analysers** are **unsound**, i.e. do not detect all bugs, (and often **imprecise**, i.e., produce many false negatives)

Sound Static Program Analysis (Abstract Interpretation)

Verify **safety properties**, something bad cannot happen,
by **over-approximation**.
If property holds for the over-approx. it holds for all behaviours.



All violations detected

Infeasible behaviours

Warning: False Negative

Some (Non-Functional) Safety Properties

Absence of **run-time errors** (at program points)

- division never by 0
- sqrt never of a negative number
- arithmetic operation never causes overflow/underflow
- array-index is never out of bounds
- dereference never applied to null-pointer

Absence of **timing accidents** (for instructions) for WCET analysis

- memory access is a cache hit
- pipeline unit is available for dispatch of operation
- bus access is not blocked

Derived global program properties

- stacks do not overflow
- program terminates within a given deadline

The Cost

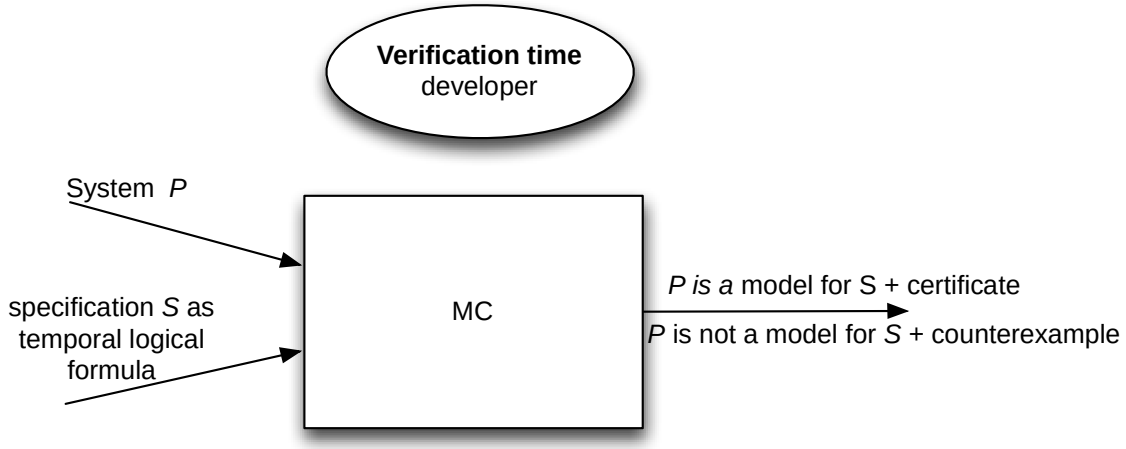
- Abstract interpretation is a **powerful**, **automatic** method
- However, it needs **educated users**
  - less so than other verification technologies, i.e. deductive verification, model checking
  - only unsound methods are really push-button technologies
  - distributes required competences better between the different roles

Roles in Formal Methods

- the **FM researcher**: enjoys academic life ☺
- the **tool developer**: has to realize the ideas of the researcher ☹
- the **system modeller/abstractor**: speaks automata
- the **property specifier**: speaks temporal logic
- the **verification engineer:** runs the tool on (a model of) the system

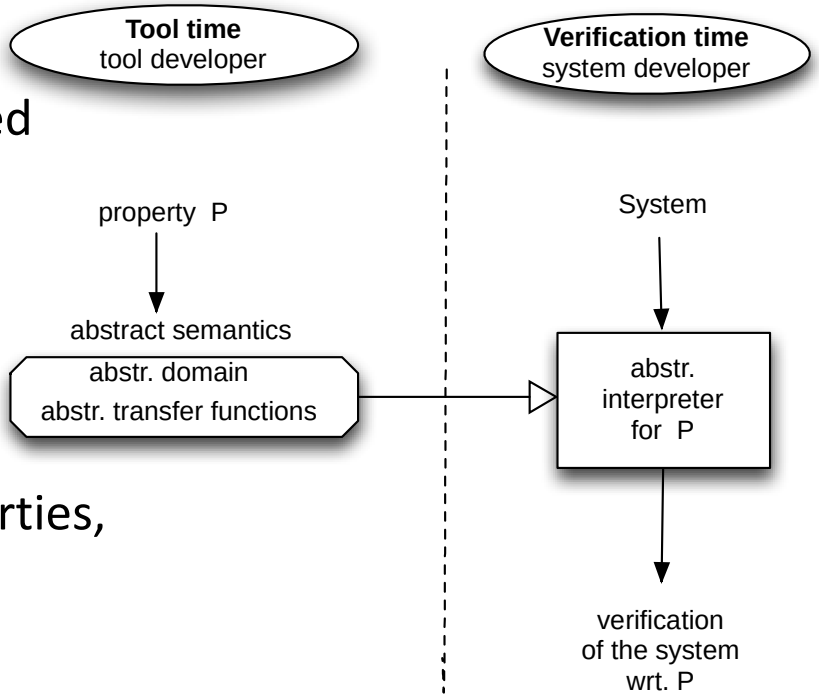
Roles in Model Checking

System modeller, property specifier, verification engineer typically coincide:
the verification engineer runs a universal tool on an abstract model of the system and a specification of a correctness property



Work Distribution in AI

- The design of abstract interpreters is difficult
- should be left to specialists,
  i.e. **tool developers**
- **verification engineer** may need
  to provide some properties
  of the system
- abstract interpreters
  are dedicated to a set of
  correctness properties,
  it can be tailored to the properties,
  good for scalability, usability



Admitted Methods for Automotive Systems: according to ISO-26262

Table 9 — Methods for the verification of software unit design and implementation

	Methods	ASIL A	ASIL B	ASIL C	ASIL D	sound, i.e., can give a guarantee
1a	Walk-through[a]	++	+	o	o	-
1b	Inspection[a]	+	++	++	++	-
1c	Semi-formal verification	+	+	++	++	-
1d	Formal verification	o	o	+	+	+
1e	Control flow analysis[bc]	+	+	++	++	-
1f	Data flow analysis[bc]	+	+	++	++	-
1g	Static code analysis	+	++	++	++	-
1h	Semantic code analysis[d]	+	+	+	+	+

[a] In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.

[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

Excerpt from:
Final Draft ISO 26262-6 Road vehicles - Functional safety – Part 6: Product development: Software Level. Version ISO/FDIS 26262-6:2011(E), 2011.

17

Examples

What we need to know

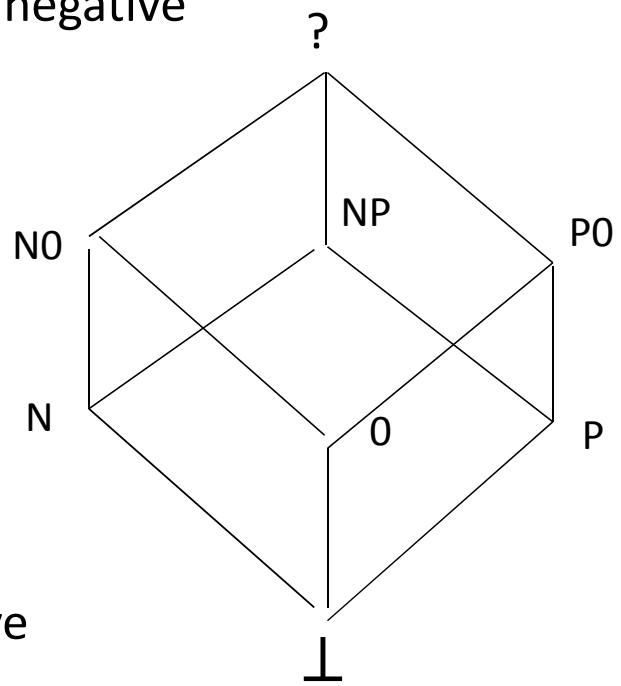
safety property	need to know
division not by 0	operand values positive or negative
sqrt not of a negative umber	operand values positive or 0
index not out of bounds	set of potential index values
no overflow/underflow	set of potential operand values

We will now meet abstractions providing these information

A Simple Abstraction of Variable Values

Strong enough to answer

- whether value may be 0
- whether value may be negative



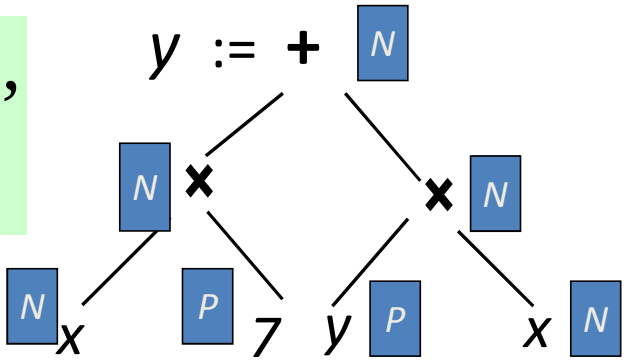
Read:

N	as	Negative
P	as	Positive
0	as	0
N0	as	Negative or 0
P0	as	Positive or 0
NP	as	Positive or Negative
?	as	don't know
⊥	as	haven't tried to find out (initial value)

The abstract effect of an assignment $y := x \times 7 + y \times x$

Before:

$\{x \mapsto N,$
$y \mapsto P\}$



$+^{\#}$	0	P	N	?
0	0	P	N	?
P	P	P	?	?
N	N	?	N	?
?	?	?	?	?

After:

$\{x \mapsto N,$
$y \mapsto N\}$

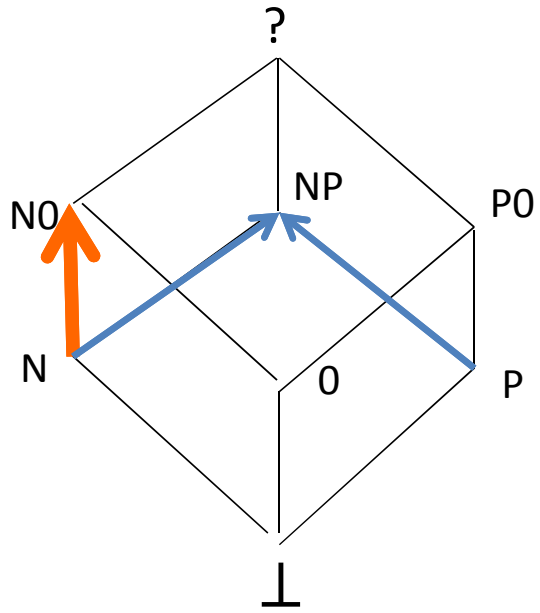
$\times^{\#}$	0	P	N	?
0	0	0	0	0
P	0	P	N	?
N	0	N	P	?
?	0	?	?	?

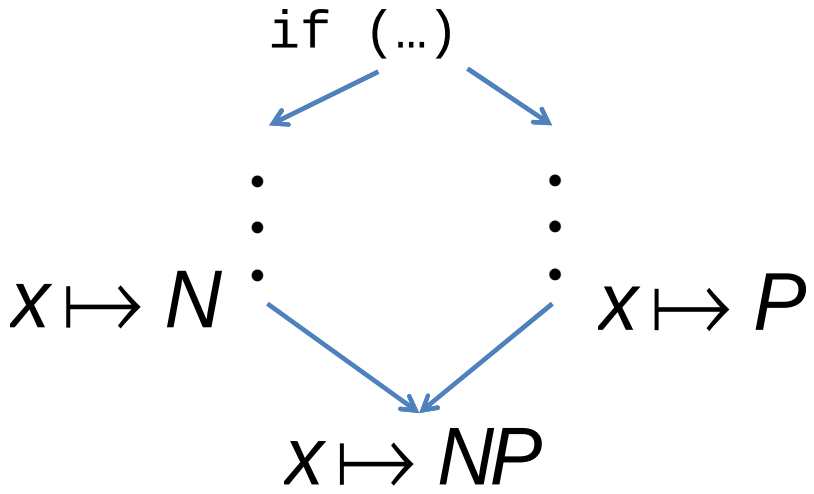
What have we seen?

- Different **abstract values** representing sets of **concrete values**:
  - P represents {1, 2, 3, …}
  - N represents {-1, -2, -3, …}
  - 0 represents {0},
  - ? represents the set of all integers,
  - ⊥ the empty set of integers.
- **Variables** have abstract values instead of concrete values
- **Abstract operations** for concrete operations
  - $+^{\#}$ for +, $\times^{\#}$ for ×
    were used to define **abstract semantics of statements**
  - assignment statement updates abstract value binding

There is more!

- Abstract values form a **lattice**
- going up means losing precision
- joining control flow means going up to the least common ancestor





Arrays

- Consider a particular data structure, **arrays**
- Correctness claim: all accesses are legal
- potential violations:
  - a[e], value of e outside bounds of a: *index-out-of bound*
  - strcopy(a,pStr), string pointed to by pStr longer than size of a: *buffer overflow*

Index-out-of Bound Errors

- How can one exclude index-out-of-bounds errors?
  - run-time check – prescribed by civilized languages
  - static verification
- Consider `a[e]`
- Claim: all potential values of `e` within the bounds of `a`
- Question: Which value may a variable have at a program point?
- Answers:
  - in general, several!
  - may depend on (the unknown) input

Intervals (Cousot&Cousot'76)

- Approach: determine intervals enclosing all potential values a variable may take on at each program point
- Access a[e] legal?
  Check whether interval for e is fully contained within bounds of a

- How to determine intervals for program variables?
- Execute the program with intervals of integers instead of integers—another abstraction of values

Arithmetic on Intervals

Assume: Intervals [0,1] for y, [2,4] for z ,
i.e., y may have value 0 or 1, z may have values 2, 3, or 4.

What is the interval for x after x = y + z ?

[2,5] = [0+2, 1+4]

What is the interval for x after x = y - z ?

[-4, -1] = [0 - 4, 1 – 2]

General rules:

Given intervals $[L_x, H_x]$ for x and $[L_y, H_y]$ for y

Interval for x+y is $[L_x+L_y, H_x+H_y]$

Interval for $x-y$ is $[L_x-H_y, H_x-L_y]$

Similar rules for * and /

Intervals – the Abstract Domain

Intervals:

$[l,u]\ with\ \ l \in N \cup \{-\infty\} \quad u \in N \cup \{\infty\} \quad l \leq u$

$[-\infty, u],\ u \neq \infty$	unknown lower bound
$[l, \infty],\ l \neq -\infty$	unknown upper bound
$[l, u],\ l \neq -\infty,\ u \neq \infty$ [*l,u*] contained in bounds of array	no index-out-of-bounds error

Intervals, some Properties

- intervals are not only relevant for checking index-out-of-bounds errors
- they are also used to prepare a data-cache analysis, see talk by Chr. Ferdinand

Buffer Overflows

Buffer overflows are the source of most security vulnerabilites:

```
char A[8];
unsigned short B;
strcpy(A, “too_long_string");
void foo()
{
}
```

- will overwrite B and an initial part of foo
- can be caught by static analysis since length of string is known

```
void myMethod(char * pStr) {
char A[8];
int nCount = 0;
strcpy(A, pStr);
}
void foo()
{
}
```

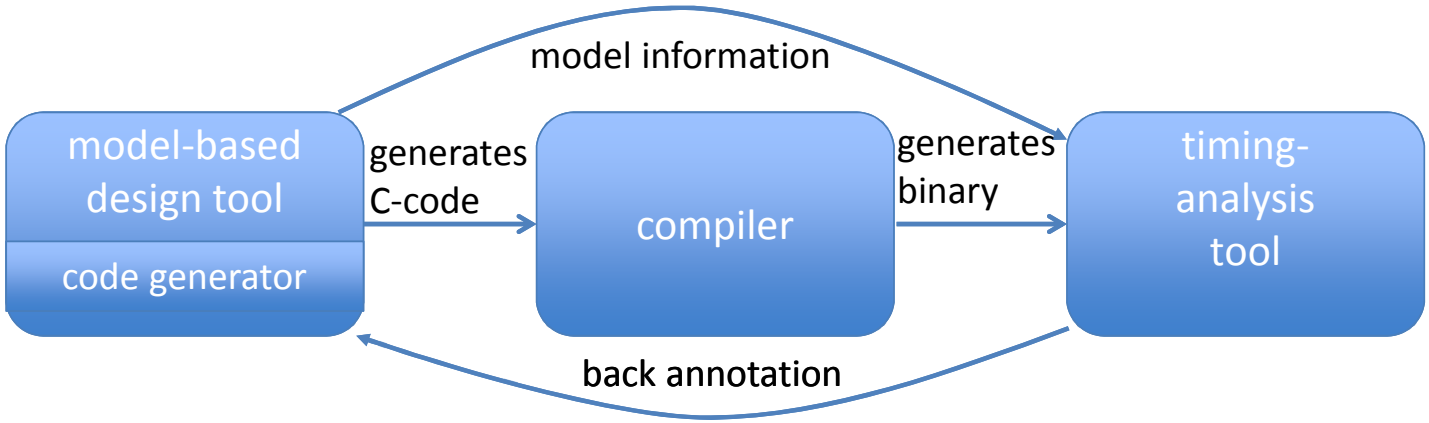
- length of string pointed by pStr is unknown
- static analysis will issue a warning for buffer overflow

Comparison

	sound/precise	automatic	scalable	
deductive verification	✔	–	–	
model checking	✔ / ✔	✔	–	
bug chasing	– / ?	✔	✔	
testing	– / ✔	?	?	
static analysis	✔ / adaptable	✔ annotation may help	adaptable	

Tool Integration

- the more information available, the more precise the results
- annotations
  - provided by developers
  - provided by environment (other tools)
- Example:



Conclusions

Sound static analysis is a viable automatic verification method:

- it has a good distribution of **competences**
- it allows to **tailor tools to applications**, i.e. types of software, cf Astrée
- it provides a trade-off between **efficiency** and **precision**
- it has a strong **theoretical foundation** in Abstract Interpretation, which makes **tool qualification** easier (without foul compromises, e.g. proven in use)