Deductive Verification, the Inductive Way

Daniel Neider

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

ForMaI Spring School

6 June 2019

1,715,430,778,504

WHITE PAPERS

Software Fail Watch: 5th Edition

White Paper

The Software Fail Watch is an analysis of software failures found in a year's worth of English language news articles. The result is an extraordinary reminder of the role software plays in our daily lives, the necessity of software testing in every industry, and the far-reaching impacts of its failure.

The 5th Edition of the Software Fail Watch identified 606 recorded software failures, impacting half of the world's population (3.7 billion people), $1.7 trillion in assets, and 314 companies. And this is just scratching the surface—there are far more software defects in the world than we will likely ever know about.

LO$$E$ FROM SOFTWARE FAILURES (USD)

1,715,430,778,504

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EIGHTTHOUSANDFIVEHUNDREDFOUR

Download the report for a detailed analysis of the year's software failures, including:

Tricentis

The Need for Software Verification

WHITE PAPERS

Software Fail Watch: 5th Edition

White Paper

The Software Fail Watch is an analysis of software failures found in a year's worth of English language news articles. The result is an extraordinary reminder of the role software plays in our daily lives, the necessity of software testing in every industry, and the far-reaching impacts of its failure.

The 5th Edition of the Software Fail Watch identified 606 recorded software failures, impacting half of the world's population (3.7 billion people), $1.7 trillion in assets, and 314 companies. And this is just scratching the surface—there are far more software defects in the world than we will likely ever know about.

LO$$E$ FROM SOFTWARE FAILURES (USD)

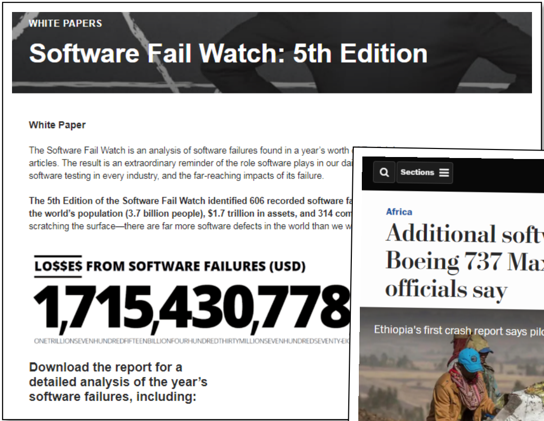
1,715,430,778,504

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EIGHTTHOUSANDFIVEHUNDREDFOUR

Download the report for a detailed analysis of the year's software failures, including:

Tricentis

The Need for Software Verification


WHITE PAPERS

Software Fail Watch: 5th Edition

White Paper

The Software Fail Watch is an analysis of software failures found in a year's worth
articles. The result is an extraordinary reminder of the role software plays in our da
software testing in every industry, and the far-reaching impacts of its failure.

The 5th Edition of the Software Fail Watch identified 606 recorded software fa
the world's population (3.7 billion people), $1.7 trillion in assets, and 314 com
scratching the surface—there are far more software defects in the world than we w

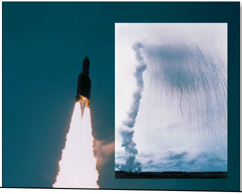
LO$$E$ FROM SOFTWARE FAILURES (USD)

1,715,430,778

ONETRILLIONSEVENHUNDREDFIFTEENBILLIONFOURHUNDREDTHIRTYMILLIONSEVENHUNDREDSEVENTY-EI

Download the report for a detailed analysis of the year's software failures, including:

Tricentis




Sections

The Washington Post

Democracy Dies in Darkness

Sign In

Get this offer

Africa

Additional software problem detected in Boeing 737 Max flight control system, officials say

Ethiopia's first crash report says pilots followed Boeing's recommendations

How does one develop high-quality software?

- Correct and complete specifications/design
- Good software development process
- Testing
- Formal verification
- Runtime monitoring
- ...

How does one develop high-quality software?

- Correct and complete specifications/design
- Good software development process
- Testing
- Formal verification
- Runtime monitoring
- ...

Combine logical reasoning and machine learning

Alan Turing, Computing Machinery and Intelligence (1950)

"Machine learning studies algorithms that can learn from data and make predictions on data without being explicitly programmed"

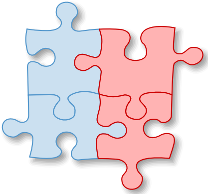
Arthur Samuel (1959)

Logical Reasoning Meets Machine Learning

Combine logical reasoning and machine learning

Alan Turing, Computing Machinery and Intelligence (1950)

Deductive Reasoning
(Formal Verification)



Inductive Reasoning
(Machine Learning)

Infer conclusions by applying logical rules

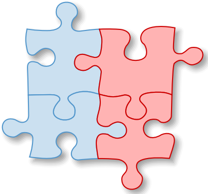
Infer conclusions by generalizing from data

Logical Reasoning Meets Machine Learning

Combine logical reasoning and machine learning

Alan Turing, Computing Machinery and Intelligence (1950)

Deductive Reasoning (Formal Verification)



Inductive Reasoning (Machine Learning)

Goal: Improve the verification process by incorporating knowledge that has been learned from the program

1. A crash course on deductive software verification

2. Inductive inference of annotations

1. Crash Course on Deductive Software Verification

Program Correctness

If the proper condition to run a program holds, and the program is run, then the program will halt, and when it halts, the desired result follows

- The proper condition to run the program is called precondition
- The desired result is called postcondition

It is often convenient to prove termination and correctness separately

- Precondition implies termination (termination)
- Precondition and termination imply postcondition (partial correctness)

Running Example

```
var i, n: int;
assume (i == 0 && n >= 0);
while (i < n)
{
    i := i + 1;
}
assert (i == n);
```

Running Example

```
var i, n: int;

assume (i == 0 && n >= 0);

while (i < n)
{
    i := i + 1;
}

assert (i == n);
```

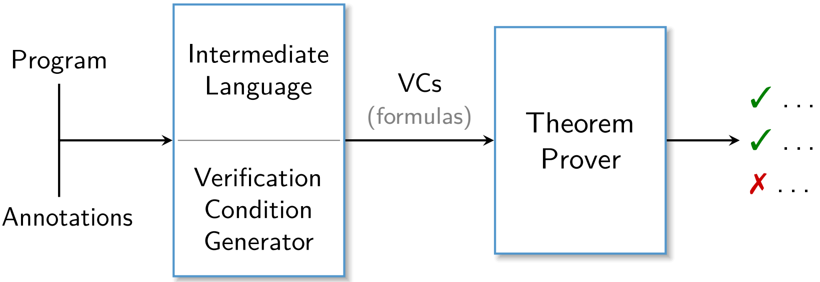
precondition

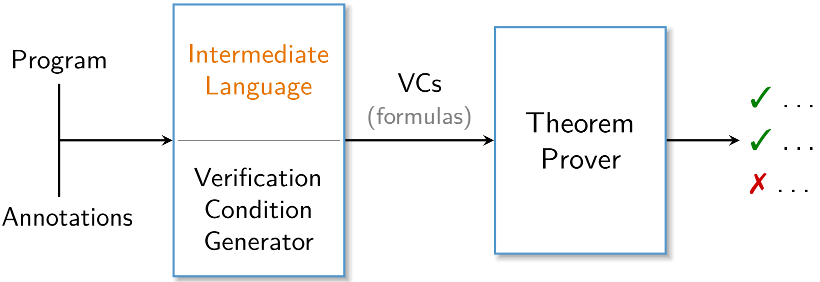
postcondition

```
var i, n: int;
assume (i == 0 && n >= 0);        precondition
while (i < n)
{
    i := i + 1;
}
assert (i == n);                  postcondition
```

Deductive Program Verification

- Express the correctness of a program as a set of mathematical statements (i.e., formulas), called verification conditions
- Then, check their validity using either automated or interactive theorem provers


Program
Annotations
Intermediate Language
Verification Condition Generator
VCs
(formulas)
Theorem Prover
✓ ...
✓ ...
✗ ...


Program
Annotations
Intermediate Language
Verification Condition Generator
VCs
(formulas)
Theorem Prover
✓ ...
✓ ...
✗ ...

SMACK

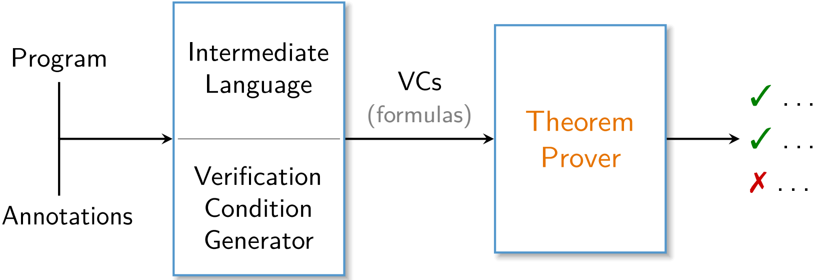
VIPER

GRASShopper

Boogie: An Intermediate Verification Language

Why3

- Support for many programming languages (e.g., via LLVM IR)
- Support for many theorem provers (e.g., via SMTLib2)
- Many industry applications (e.g., Microsoft SDV, Facebook INFER)


Program
Annotations
Intermediate Language
Verification Condition Generator
VCs
(formulas)
Theorem Prover
✓ ...
✓ ...
✗ ...

Satisfiability Modulo Theories (SMT)

Satisfiability problem for logical formulas with respect to combinations of background theories expressed in first-order logic with equality

- theory of real numbers, the theory of integers
- theory of bit vectors (useful for modeling machine-level data types)
- theory of various data structures such as lists and arrays

Usually, one considers quantifier-free theories

Satisfiability Modulo Theories (SMT)

Satisfiability problem for logical formulas with respect to combinations of background theories expressed in first-order logic with equality

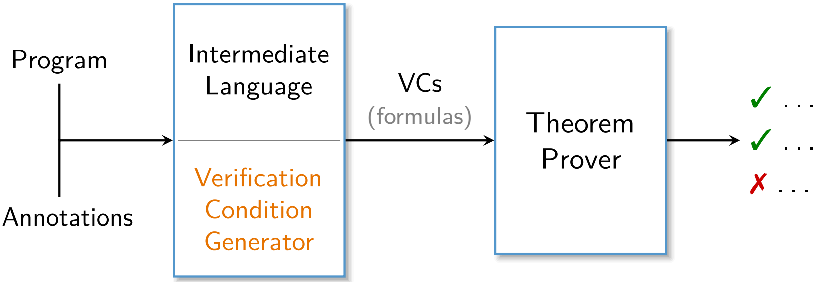
- theory of real numbers, the theory of integers
- theory of bit vectors (useful for modeling machine-level data types)
- theory of various data structures such as lists and arrays

Usually, one considers quantifier-free theories

SMT Solvers

Numerous highly-optimized solvers are available

- Z3, CVC4, OpenSMT, ...
- `http://smtcomp.sourceforge.net/2018/`


Program
Annotations
Intermediate Language
Verification Condition Generator
VCs
(formulas)
Theorem Prover
✓ ...
✓ ...
✗ ...

Verification Conditions (VCs)

Verification conditions are logic formulas derived from the program's source code

- If the VC is valid: the program is correct
- If the VC is invalid: there are errors in the program

Verification Conditions (VCs)

Verification conditions are logic formulas derived from the program's source code

- If the VC is valid: the program is correct
- If the VC is invalid: there are errors in the program

Floyd-Hoare-style Verification

- Hoare triples $\{P\}S\{Q\}$ formalize the semantics of software for the purpose of deductive verification
- Verification conditions can be generated automatically using the concept of weakest preconditions

Verification Conditions: Straight-Line Code

Proving a Program Correct

```
var x: int;
assume x >= 1;
x := x + 2;
assert x >= 3;
```

Proving a Program Correct

```
var x: int;
assume x >= 1;
x := x + 2;
assert x >= 3;
```

$$x_2 \geq 1 \Rightarrow \Big[x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3] \Big]$$

Verification Conditions: Straight-Line Code

Proving a Program Correct

```
var x: int;
assume x >= 1;
x := x + 2;
assert x >= 3;
```

$$x_2 \geq 1 \Rightarrow \Big[x_3 = x_2 + 2 \Rightarrow \big[x_3 \geq 3 \big] \Big]$$

Detecting Assertion Violations

```
var x: int;
assume x >= 1;
x := x + 2;
assert x < 3;
```

Proving a Program Correct

```
var x: int;
assume x >= 1;
x := x + 2;
assert x >= 3;
```

$$x_2 \geq 1 \Rightarrow \left[x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3]\right]$$

Detecting Assertion Violations

```
var x: int;
assume x >= 1;
x := x + 2;
assert x < 3;
```

$$x_2 \geq 1 \Rightarrow \left[x_3 = x_2 + 2 \Rightarrow [x_3 < 3]\right]$$

Proving a Program Correct

```
1: var x: int;
2: assume x >= 1;
3: x := x + 2;
4: assert x >= 3;
```

$$x_2 \geq 1 \Rightarrow \left[x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3] \right]$$

Detecting Assertion Violations

```
1: var x: int;
2: assume x >= 1;
3: x := x + 2;
4: assert x < 3;
```

$$x_2 \geq 1 \Rightarrow \left[x_3 = x_2 + 2 \Rightarrow [x_3 < 3] \right]$$

- A satisfying assignment for the negation of the VC provides input to the program that violates the assertion

Verification Conditions: Branching

```
var x, y: int;
if (x < 0) {
   y := -x;
} else {
   y := x;
}
assert y >= 0;
```

Verification Conditions: Branching

```
var x, y: int;
if (x < 0) {
   y := -x;
} else {
   y := x;
}
assert y >= 0;
```

$$\Big[x_2 < 0 \Rightarrow \big[y_3 = -x_2 \Rightarrow [y_3 \geq 0]\big]\Big]$$
$$\wedge \Big[\neg(x_2 < 0) \Rightarrow \big[y_5 = x_2 \Rightarrow [y_5 \geq 0]\big]\Big]$$

Design by Contract or Assume-Guarantee Reasoning

Developers annotate software components with contracts (i.e., formal specifications)

- Contracts document the developer's intent
- Verification is broken down into compositional verification of individual components

Typical Contracts

- Annotations on procedure boundaries (preconditions and postconditions)
- Annotations on loop boundaries (loop invariants)

Example

How can we verify the following program?

```
foo() { ... }
bar() { ... foo(); ... }
```

Example

How can we verify the following program?

```
foo() { ... }
bar() { ... foo(); ... }
```

First Solution

Inline `foo`

Example

How can we verify the following program?

```
foo() { ... }
bar() { ... foo(); ... }
```

Second Solution

Write contract/specification P of `foo`

- Assume P when checking `bar`

```
bar() { ... assume P; ... }
```

- Guarantee P when checking foo

```
foo() { ... assert P; }
```

Method Contracts – Details

```
procedure M(x, y)
returns (r, s)
requires P
ensures Q
{
    S;
}

call a, b := M(c, d);
```

```
procedure M(x, y)
returns (r, s)
requires P
ensures Q
{
    S;
}

call a, b := M(c, d);
```

```
assume P;
S;
assert Q;
```

```
procedure M(x, y)
returns (r, s)
requires P
ensures Q
{
    S;
}
```

```
assume P;
S;
assert Q;
```

```
call a, b := M(c, d);
```

```
x' := c;   y' := d;
assert P';
assume Q';
a := r';   b := s';
```

where `x'`, `y'`, `r'`, `s'` are fresh variables, `P'` is `P` with `x'`, `y'` for `x`, `y`, and `Q'` is `Q` with `x'`, `y'`, `r'`, `s'` for `x`, `y`, `r`, `s`

Loops and Loop Unrolling

```
var i, n: int;
assume (i == 0 && n >= 0);
while (i < n)
{
    i := i + 1;
}
assert (i == n);
```

```
var i, n: int;
assume (i == 0 && n >= 0);
while (i < n)
{
    i := i + 1;
}
assert (i == n);
```

Loop Unrolling

```
assume (i == 0 && n >= 0);
if (i < n) {
    i := i + 1;
    if (i < n) {
        ...     (assume false;)
    }
}
assert (i == n);
```

N unrollings

Loops and Loop Invariants

```
var i, n: int;
assume (i == 0 && n >= 0);
while (i < n)
{
    i := i + 1;
}
assert (i == n);
```

Loop Invariants

A loop invariant is a statement I over the variables of the program (i.e., a predicate) such that

- I holds before the loop and is implied by the precondition
- I holds on every iteration of the loop (is inductive)
- I holds after the final iteration and implies the postcondition

```
var i, n: int;
assume (i == 0 && n >= 0);
while (i < n)
{
    i := i + 1;
}
assert (i == n);
```

Loop Invariants

An adequate invariant is $i \leq n$

- Precondition: $(i = 0 \land n \geq 0) \;\Rightarrow\; i \leq n$
- Inductivity: $(i \leq n \;\land\; i < n \;\land\; i' = i + 1 \;\land\; n' = n) \Rightarrow i' \leq n'$
- Postcondition: $(i \leq n \;\land\; \neg(i < n)) \;\Rightarrow\; i = n$

Quiz: Can You Prove This Program Correct?

Example

```
var x, y: int;
x := -1;
while (x < 0)
invariant ???;
{
    x := x + y;
    y := y + 1;
}
assert (y > 0);
```

https://horn-ice.mpi-sws.org

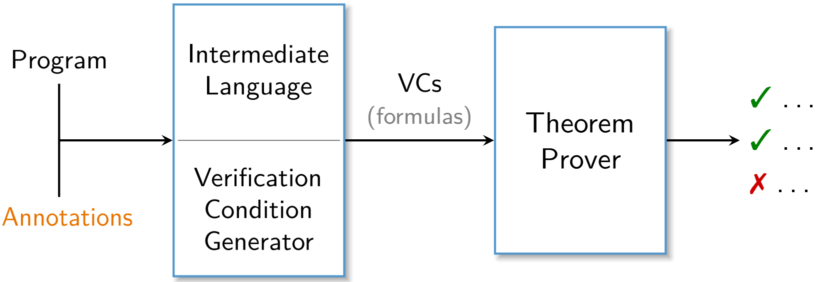
Quiz: Can You Prove This Program Correct?

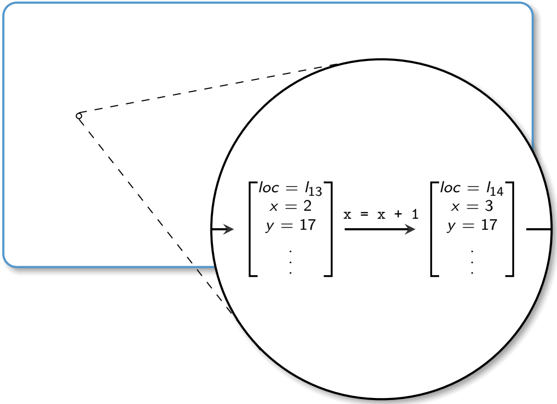
Example

```
var x, y: int;
x := -1;
while (x < 0)
invariant x >= 0 ==> y > 0;
{
      x := x + y;
      y := y + 1;
}
assert (y > 0);
```

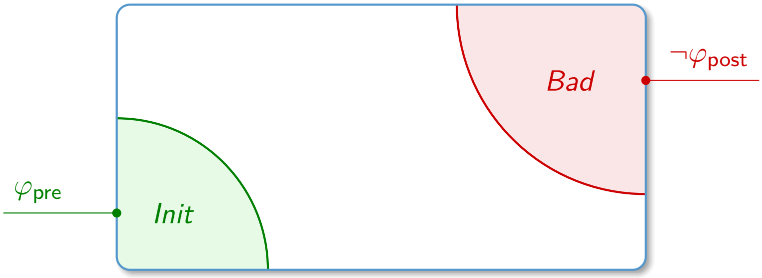
https://horn-ice.mpi-sws.org

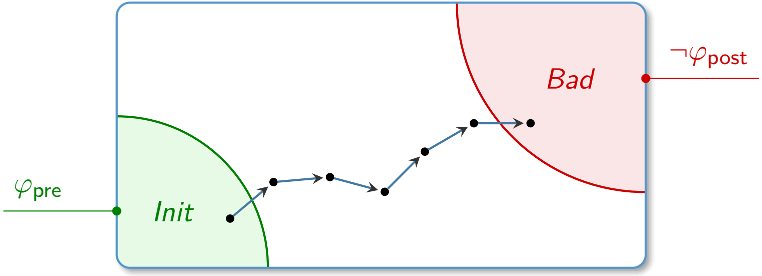
2. Inductive Inference of Annotations

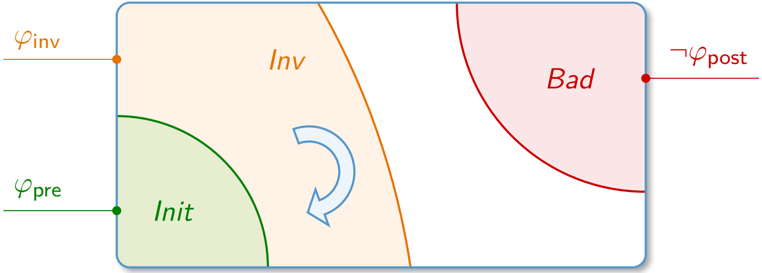

Program
Annotations
Intermediate Language
Verification Condition Generator
VCs
(formulas)
Theorem Prover
✓ ...
✓ ...
✗ ...



$\begin{bmatrix} loc = l_{13} \\ x = 2 \\ y = 17 \\ \vdots \end{bmatrix} \xrightarrow{\texttt{x = x + 1}} \begin{bmatrix} loc = l_{14} \\ x = 3 \\ y = 17 \\ \vdots \end{bmatrix}$

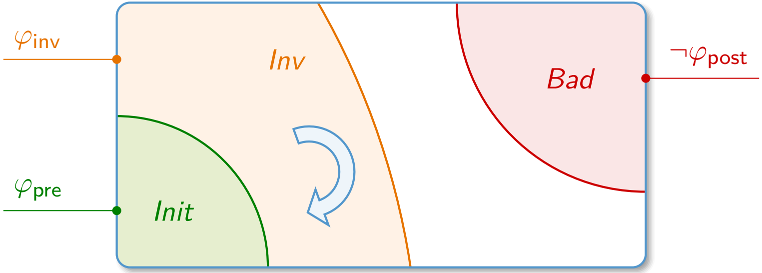

$\neg\varphi_{\text{post}}$
Bad
φ_{pre}
Init


$\neg\varphi_{\text{post}}$
Bad
φ_{pre}
Init



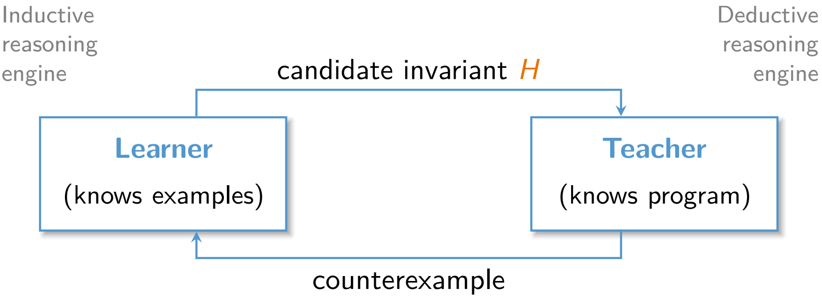
Invariant

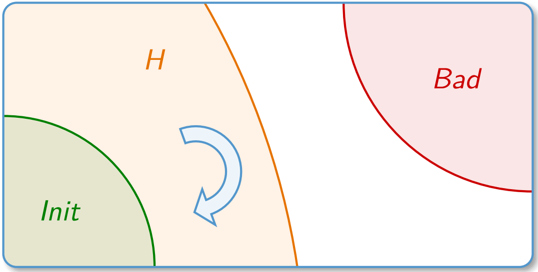
1. $\mathit{Init} \subseteq \mathit{Inv}$ (includes initial configurations)
2. $\mathit{Bad} \cap \mathit{Inv} = \emptyset$ (excludes bad configurations)
3. $\mathit{Step}(\mathit{Inv}) \subseteq \mathit{Inv}$ (is inductive)



Invariant Synthesis

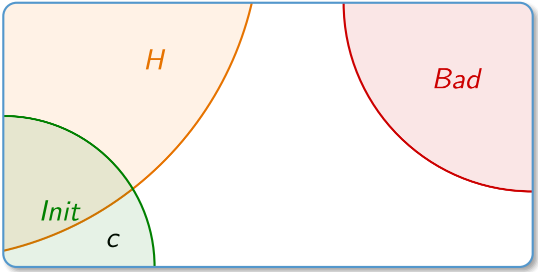
- Abstract interpretation, predicate abstraction, Craig's interpolation, IC3, etc.
- Inductive techniques from machine learning


Inductive reasoning engine
Deductive reasoning engine
candidate invariant H
Learner
(knows examples)
Teacher
(knows program)
counterexample



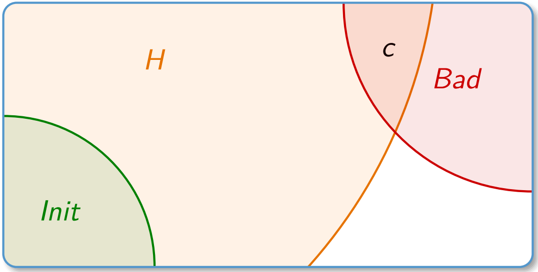
Invariant

1. $Init \subseteq H$ (includes initial configurations)
2. $Bad \cap H = \emptyset$ (excludes bad configurations)
3. $Step(H) \subseteq H$ (is inductive)



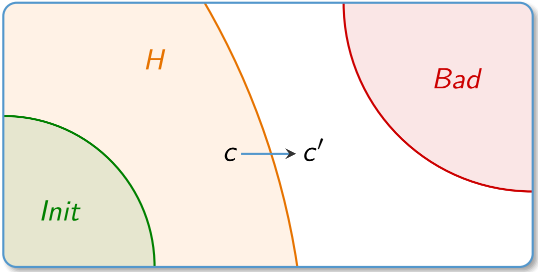
Refuting Non-Invariants

1. Positive counterexample: if $\mathit{Init} \not\subseteq H$, report $c \in \mathit{Init} \setminus H$



Refuting Non-Invariants

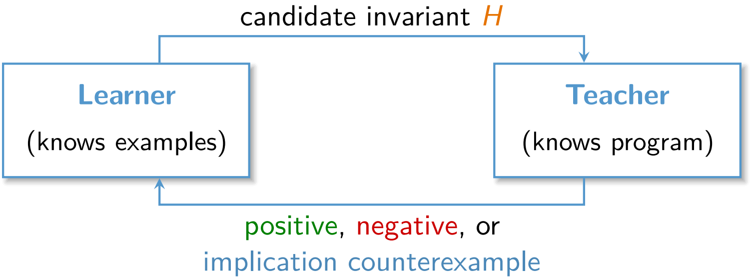
1. Positive counterexample: if $Init \not\subseteq H$, report $c \in Init \setminus H$
2. Negative counterexample: if $Bad \cap H \neq \emptyset$, report $c \in Bad \cap H$



Refuting Non-Invariants

1. Positive counterexample: if $Init \not\subseteq H$, report $c \in Init \setminus H$
2. Negative counterexample: if $Bad \cap H \neq \emptyset$, report $c \in Bad \cap H$
3. Implication counterexample: if $Step(H) \not\subseteq H$, report $c \Rightarrow c'$ with $Step(c, c')$, $c \in H$, and $c' \notin H$

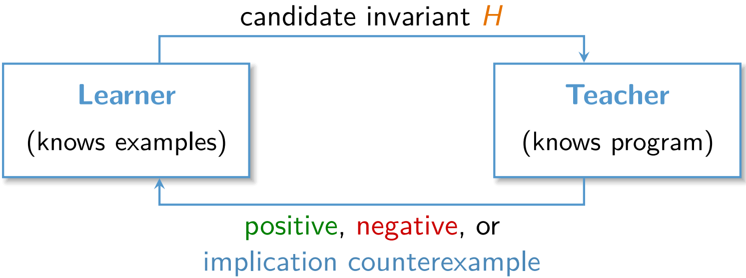
ICE Learning [CAV '14]



Teacher

1. Given a hypothesis H, check Conditions 1, 2, and 3
2. If H is not an invariant, return a positive, negative, or implication counterexample

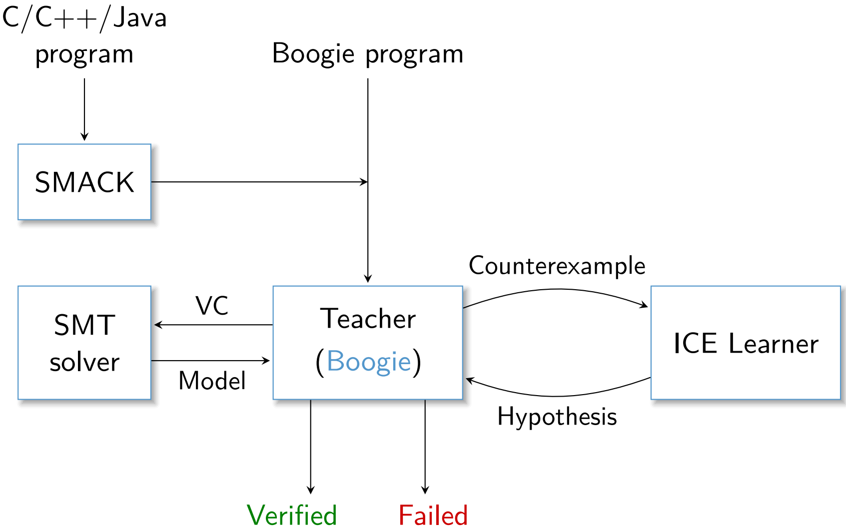
ICE Learning [CAV '14]

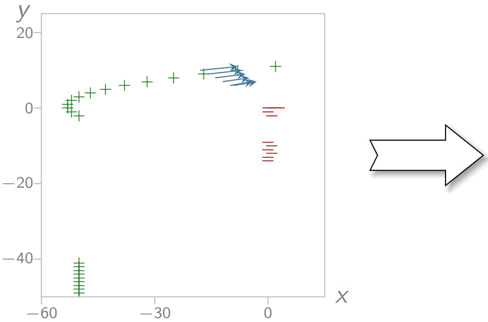


Learner

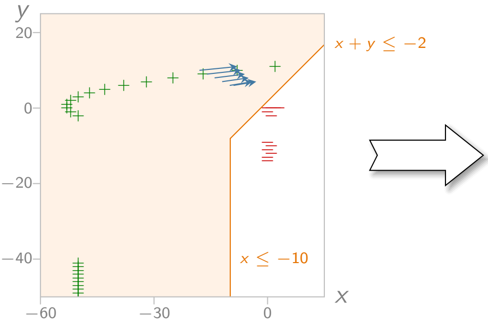
Maintains a sample $\mathcal{S} = (Pos, Neg, Impl)$ and constructs a hypothesis H that is consistent with $\mathcal{S}$:

- $c \in H$ for each $c \in Pos$
- $c \notin H$ for each $c \in Neg$
- $c \in H$ implies $c' \in H$ for each $c \Rightarrow c' \in Impl$

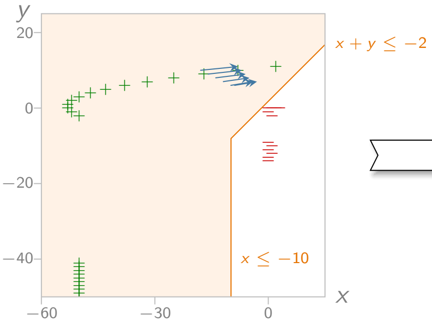

C/C++/Java
program
Boogie program
SMACK
Counterexample
SMT
solver
VC
Teacher
(Boogie)
ICE Learner
Model
Hypothesis
Verified
Failed



Hypothesis H



$$x + y \leq -2$$
$$\vee$$
$$x \leq -10$$



$$x + y \leq -2 \\ \vee \\ x \leq -10$$

Simplification

- We assume that a finite set $\mathcal{P}$ of predicates is given that allows separating any pair of program configurations in the ICE sample
- We will relax this restriction later

Three Types of ICE Learning Algorithms

A. Houdini
(Flanagan and Leino, FME '01)

B. Sorcar
(Madhusudan, N., and Saha)

C. ICE Learning Using Decision Trees
(Garg, Madhusudan, N., Roth, POPL '16)

A. Houdini

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \to (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \land \ldots \land p_n$)
while *X is not consistent with Pos* **do**
  Remove predicates p_i from X that "occur as 0" in a positive example
  if *the left-hand-side of an implication in Impl is satisfied* **then**
    mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

The Houdini Algorithm

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1,0,1,1,0)$, $+$; $(1,1,1,0,1)$, $+$
- $(1,1,1,0,0) \to (0,1,1,1,1)$
- $(0,1,0,1,1)$, $-$

$$p_1 \land p_2 \land p_3 \land p_4 \land p_5$$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \land \ldots \land p_n$)
while *X is not consistent with Pos* **do**
  Remove predicates p_i from X that "occur as 0" in a positive example
  if *the left-hand-side of an implication in Impl is satisfied* **then**
    mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \wedge \ldots \wedge p_n$)
while *X is not consistent with Pos* **do**
    Remove predicates p_i from X that "occur as 0" in a positive example
    if *the left-hand-side of an implication in Impl is satisfied* **then**
        mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1,0,1,1,0)$, $+$; $(1,1,1,0,1)$, $+$
- $(1,1,1,0,0) \rightarrow (0,1,1,1,1)$
- $(0,1,0,1,1)$, $-$

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \wedge \ldots \wedge p_n$)
while X *is not consistent with Pos* **do**
    Remove predicates p_i from X that "occur as 0" in a positive example
    if *the left-hand-side of an implication in Impl is satisfied* **then**
        mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

The Houdini Algorithm

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

Algorithm 1: The Houdini algorithm

```
X ← P (i.e., φ_X = p_1 ∧ ... ∧ p_n)
while X is not consistent with Pos do
    Remove predicates p_i from X that "occur as 0" in a positive example
    if the left-hand-side of an implication in Impl is satisfied then
        mark the right-hand-side as positive
return X if no negative example in Neg is satisfied
```

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \wedge \ldots \wedge p_n$)
while *X is not consistent with Pos* **do**
  Remove predicates p_i from X that "occur as 0" in a positive example
  if *the left-hand-side of an implication in Impl is satisfied* **then**
    mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

Example

Let $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$ be a set of predicates

- $(1, 0, 1, 1, 0)$, $+$; $(1, 1, 1, 0, 1)$, $+$
- $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
- $(0, 1, 0, 1, 1)$, $-$

$$p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5$$

Algorithm 1: The Houdini algorithm

$X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \wedge \ldots \wedge p_n$)
while *X is not consistent with Pos* **do**
  Remove predicates p_i from X that "occur as 0" in a positive example
  if *the left-hand-side of an implication in Impl is satisfied* **then**
    mark the right-hand-side as positive
return X if no negative example in *Neg* is satisfied

Theorem (Flanagan and Leino, [FME '01])

Houdini learns the semantically smallest inductive invariant expressible as a conjunction over $\mathcal{P}$ in at most $|\mathcal{P}|$ rounds (if one exists). The time spend in each round is proportional to $|\mathcal{S}| \cdot |\mathcal{P}|$.

Theorem (Flanagan and Leino, [FME '01])

Houdini learns the semantically smallest inductive invariant expressible as a conjunction over $\mathcal{P}$ in at most $|\mathcal{P}|$ rounds (if one exists). The time spend in each round is proportional to $|\mathcal{S}| \cdot |\mathcal{P}|$.

- Advantage:
  Houdini is independent of negative examples/the post-condition

Theorem (Flanagan and Leino, [FME '01])

Houdini learns the semantically smallest inductive invariant expressible as a conjunction over $\mathcal{P}$ in at most $|\mathcal{P}|$ rounds (if one exists). The time spend in each round is proportional to $|\mathcal{S}| \cdot |\mathcal{P}|$.

- Advantage:
  Houdini is independent of negative examples/the post-condition
- Disadvantage:
  Houdini is independent of negative examples/the post-condition

B. Sorcar

Idea: Relevant Predicates

A predicate is *relevant* if it has shown evidence to be useful for refuting negative examples (i.e., occurs as “0” in a negative example)

Idea: Relevant Predicates

A predicate is *relevant* if it has shown evidence to be useful for refuting negative examples (i.e., occurs as “0” in a negative example)

Algorithm 2: The Sorcar algorithm

```
static R ← ∅
Procedure Sorcar(𝒮, 𝒫, R):
    X ← Houdini(𝒮, 𝒫)        // Takes care of positive examples in Pos
    while X ∩ R is not consistent with 𝒮 do
        foreach negative example in Neg not consistent with X ∩ R do
            Add “relevant” predicate from X \ R to R
        foreach implication in Impl not consistent with X ∩ R do
            Mark the left-hand-side as negative
    return X
```

Theorem (Madhusudan, N., Saha)

Sorcar learns an inductive invariant in at most $2 \cdot |\mathcal{P}|$ rounds if one is expressible as a conjunction over $\mathcal{P}$. The time spend in each round is proportional to $|\mathcal{S}| \cdot f(|\mathcal{P}|)$, where f is a function capturing the complexity of finding relevant predicates.

- *The conjunction Sorcar computes is always a subset of the one Houdini computes*

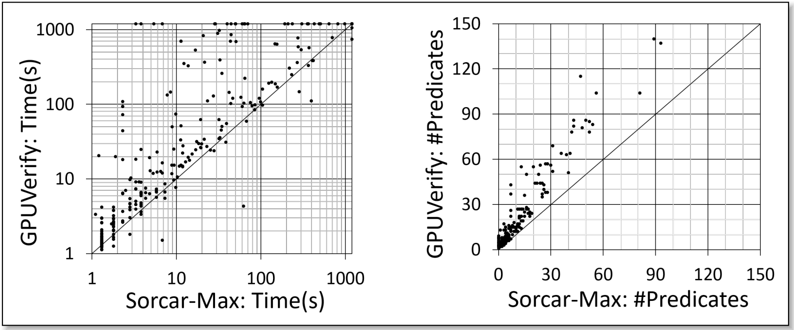
Theorem (Madhusudan, N., Saha)

Sorcar learns an inductive invariant in at most $2 \cdot |\mathcal{P}|$ rounds if one is expressible as a conjunction over $\mathcal{P}$. The time spend in each round is proportional to $|\mathcal{S}| \cdot f(|\mathcal{P}|)$, where f is a function capturing the complexity of finding relevant predicates.

- *The conjunction Sorcar computes is always a subset of the one Houdini computes*

1. `Sorcar-Max`: $f(n) \in \mathcal{O}(n)$
2. `Sorcar-First`: $f(n) \in \mathcal{O}(n)$
3. `Sorcar-Min`: $f(n) \in \mathcal{O}(2^n)$
4. `Sorcar-Greedy`: $f(n) \in \mathcal{O}(n^3)$

Houdini vs. Sorcar



Quiz: Can You Prove This Program Correct?

Example (Houdini/Sorcar)

```
var x, y, z: int;
assume (x < y);
z := y;
while (z > x)
invariant ???;
{
    z := z - 1;
}
assert (x <= z);
```

https://horn-ice.mpi-sws.org

Quiz: Can You Prove This Program Correct?

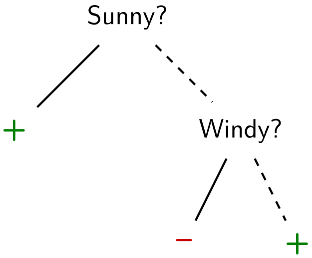
Example (Houdini/Sorcar)

```
var x, y, z: int;
assume (x < y);
z := y;
while (z > x)
invariant x <= z && y > x;
{
    z := z - 1;
}
assert (x <= z);
```

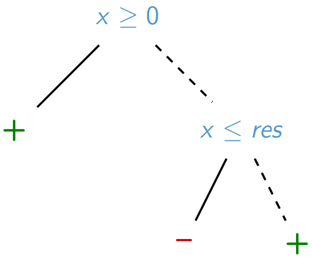
https://horn-ice.mpi-sws.org

C. ICE Learning Using Decision Trees

Sunny?	Hot?	Windy?	Play Tennis
0	1	1	−
0	0	1	−
1	0	1	+
0	1	0	+

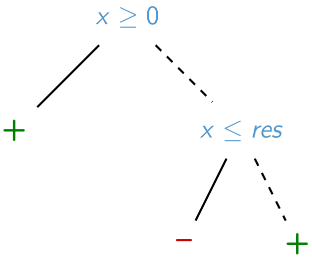


$x \geq 0$	$res < 0$	$x \leq res$	Class
0	1	1	−
0	0	1	−
1	0	1	+
0	1	0	+



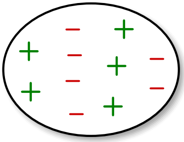
Decision Trees

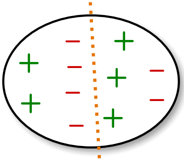
$x \geq 0$	$res < 0$	$x \leq res$	Class
0	1	1	$-$
0	0	1	$-$
1	0	1	$+$
0	1	0	$+$



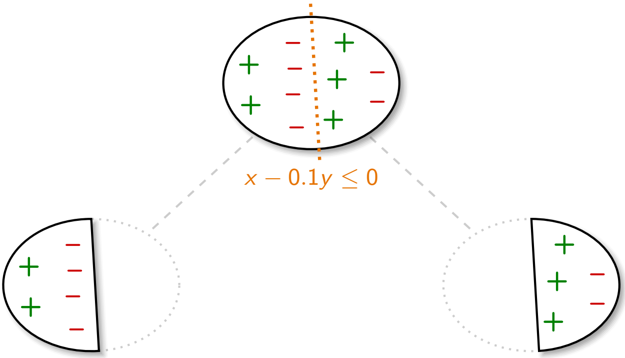
$$x \geq 0 \lor (\neg(x \geq 0) \land \neg(x \leq res))$$

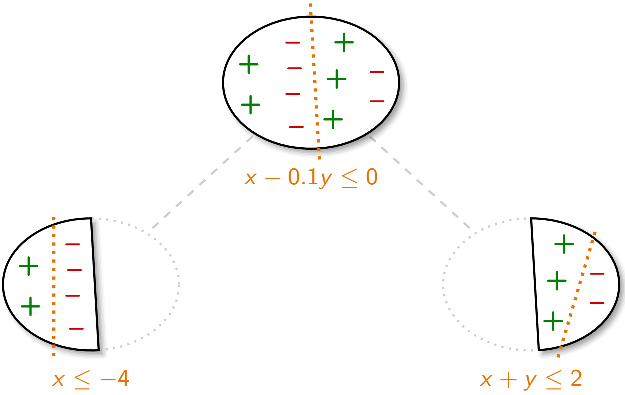
Learning Decision Trees

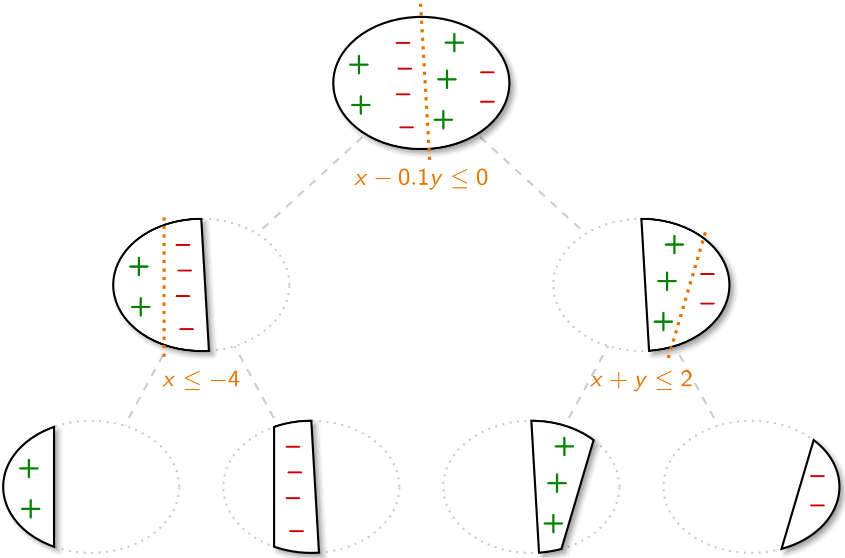




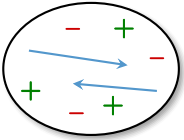
$x - 0.1y \leq 0$

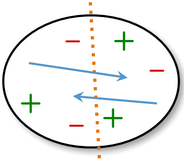

$x - 0.1y \leq 0$


$x - 0.1y \leq 0$
$x \leq -4$
$x + y \leq 2$

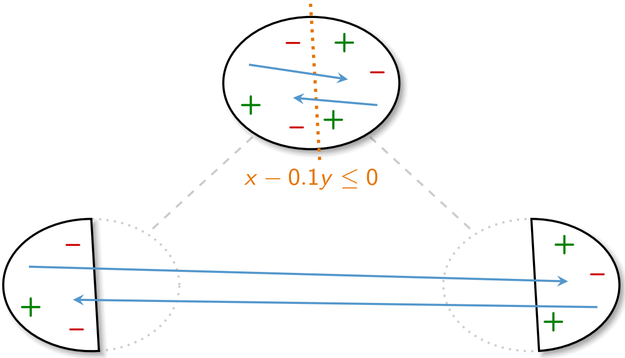

$x - 0.1y \leq 0$
$x \leq -4$
$x + y \leq 2$

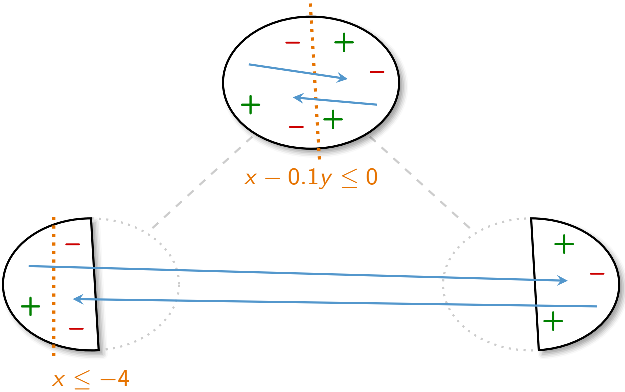
Learning Decision Trees

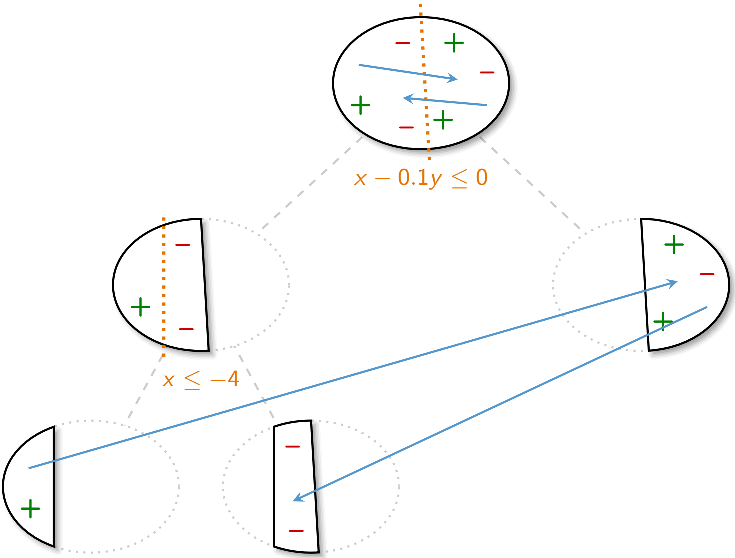




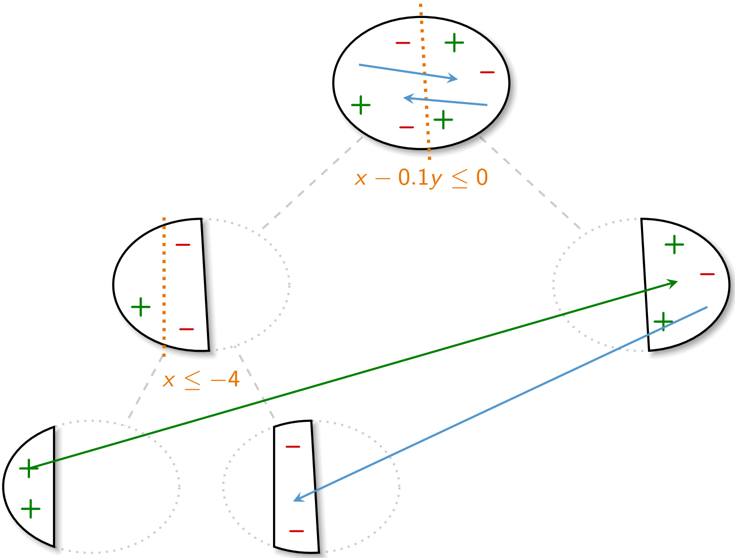
$x - 0.1y \leq 0$


$x - 0.1y \leq 0$


$x - 0.1y \leq 0$
$x \leq -4$

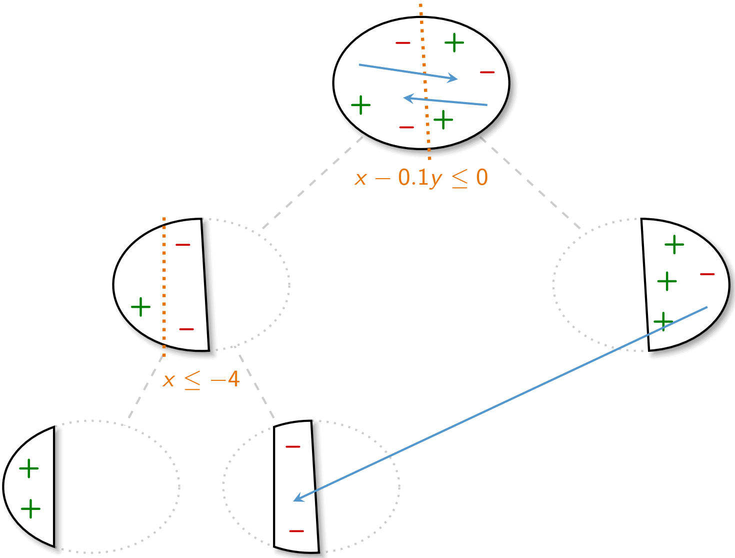


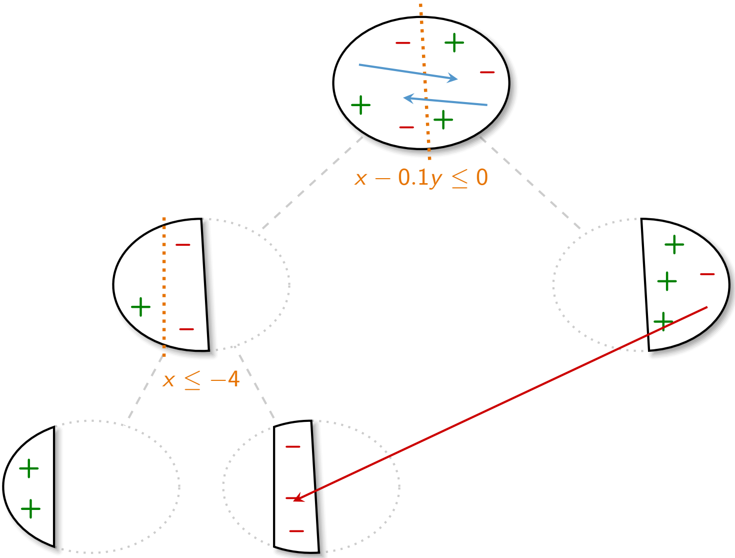
$x - 0.1y \leq 0$
$x \leq -4$



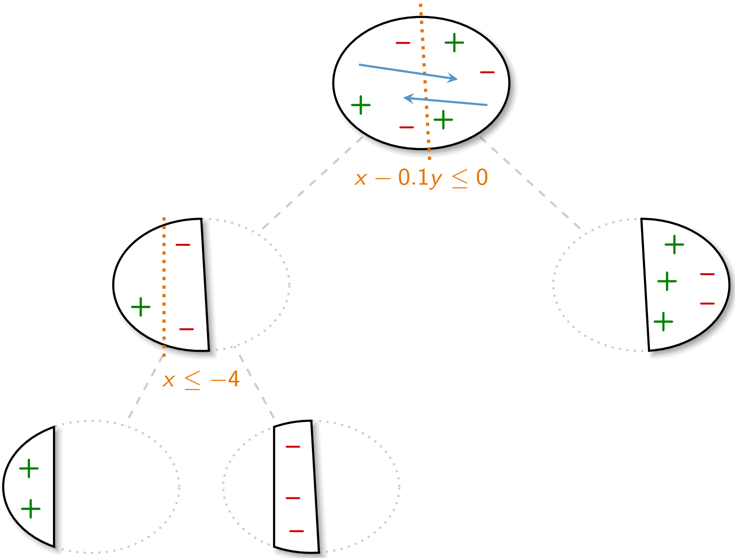
$x - 0.1y \leq 0$
$x \leq -4$

Learning Decision Trees

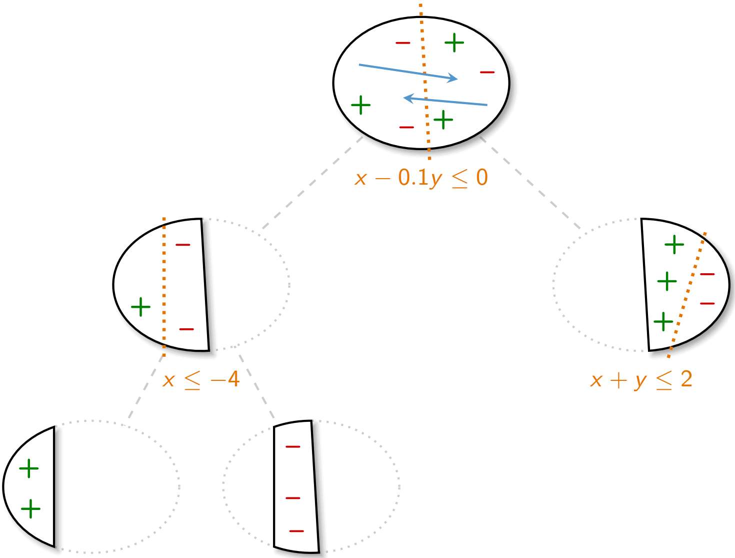




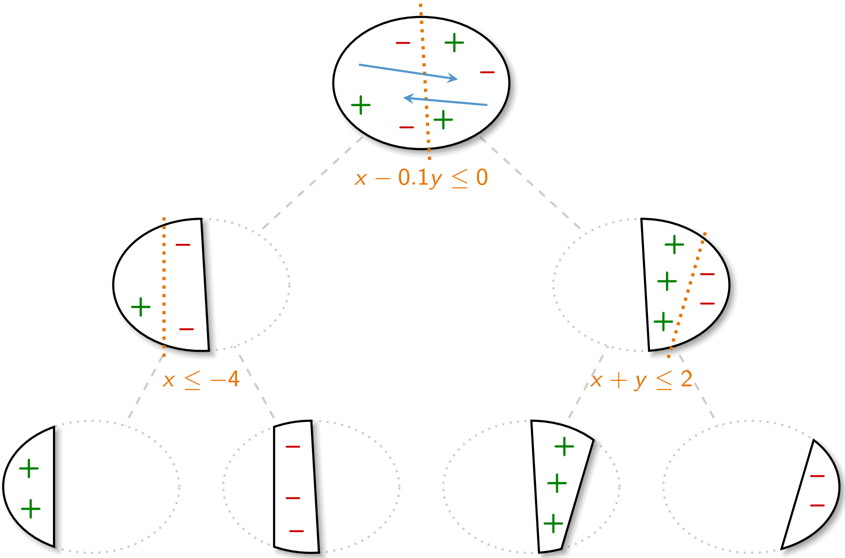
−
+
−
+
−
+
$x - 0.1y \leq 0$
−
+
−
+
+
−
+
$x \leq -4$
+
+
−
−
−



$x - 0.1y \leq 0$
$x \leq -4$



$x - 0.1y \leq 0$
$x \leq -4$
$x + y \leq 2$



$x - 0.1y \leq 0$
$x \leq -4$
$x + y \leq 2$

Goal: Split such that the resulting subsamples are as "pure" as possible

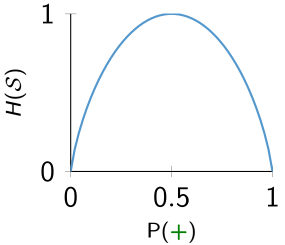
Goal: Split such that the resulting subsamples are as "pure" as possible

Entropy (Shannon, 1948)

Let $\mathcal{S} = (Pos, Neg)$ be a sample. Then, entropy is defined as

$$H(\mathcal{S}) = -\Big[P(+)\log_2 P(+) + P(-)\log_2 P(-)\Big],$$

where $P(+) = \frac{|Pos|}{|Pos|+|Neg|}$ and $P(-) = \frac{|Neg|}{|Pos|+|Neg|}$



Goal: Split such that the resulting subsamples are as "pure" as possible

Entropy (Shannon, 1948)

Let $\mathcal{S} = (Pos, Neg)$ be a sample. Then, entropy is defined as

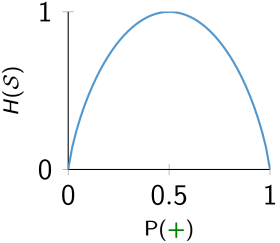
$$H(\mathcal{S}) = -\Big[P(+)\log_2 P(+) + P(-)\log_2 P(-)\Big],$$

where $P(+) = \frac{|Pos|}{|Pos|+|Neg|}$ and $P(-) = \frac{|Neg|}{|Pos|+|Neg|}$

Information Gain

A split of $\mathcal{S}$ into $\mathcal{S}_1$ and $\mathcal{S}_2$ results in an information gain of

$$G(\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2) = H(\mathcal{S}) - \big(H(\mathcal{S}_1) + H(\mathcal{S}_2)\big)$$



How to Split in the Presence of Implications

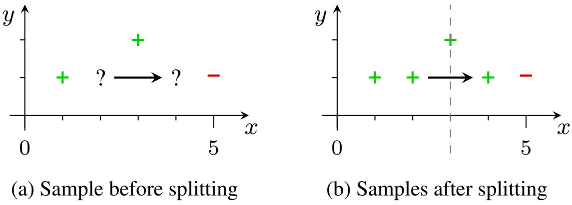
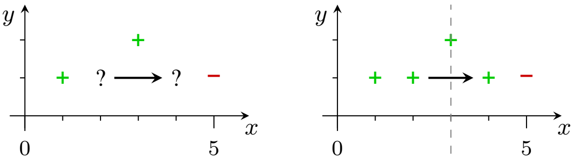


Figure 1: The samples discussed in Example 1.



(a) Sample before splitting (b) Samples after splitting

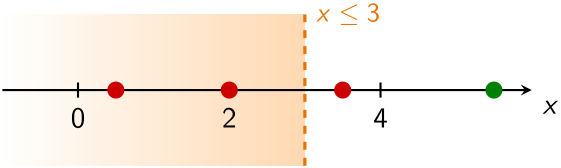
Figure 1: The samples discussed in Example 1.

Penalize Implications That Are Cut by a Split

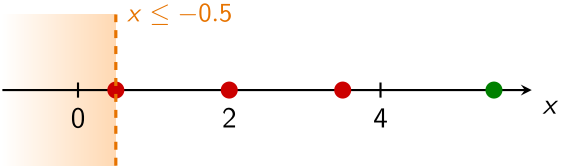
Let $c \in \mathbb{R}$ be a constant and γ be the number of implications that go from $\mathcal{S}_1$ to $\mathcal{S}_2$ or vice versa

$$G_{\text{penalty}}(\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2) = G(\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2) - c \cdot \gamma$$

A fixed set of predicates might be insufficient to construct a consistent decision tree



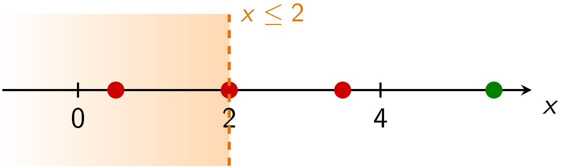
A fixed set of predicates might be insufficient to construct a consistent decision tree



Solution

Let the learner choose the best split based on the data, which allows separating any pair of program configurations

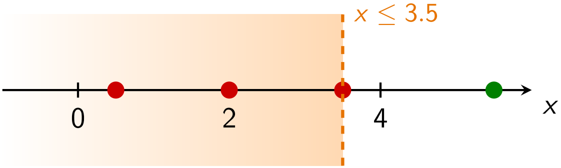
A fixed set of predicates might be insufficient to construct a consistent decision tree



Solution

Let the learner choose the best split based on the data, which allows separating any pair of program configurations

A fixed set of predicates might be insufficient to construct a consistent decision tree



Solution

Let the learner choose the best split based on the data, which allows separating any pair of program configurations

Correctness of the Decision Tree Learner

Theorem (Garg, Madhusudan, N., Roth [POPL '16])

Let $\mathcal{P}$ be a finite set of predicates that allows separating any two data points in a sample. If the sample is non-contradictory, the presented learner always produces a decision tree over $\mathcal{P}$ that is consistent with the given ICE sample (independent of the strategy used to split).

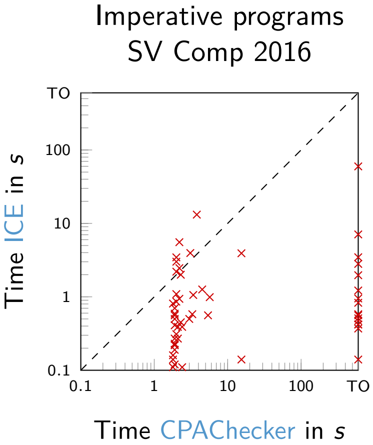
Theorem (Garg, Madhusudan, N., Roth [POPL '16])

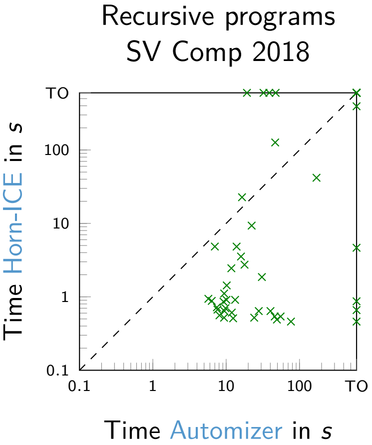
Let $\mathcal{P}$ be a finite set of predicates that allows separating any two data points in a sample. If the sample is non-contradictory, the presented learner always produces a decision tree over $\mathcal{P}$ that is consistent with the given ICE sample (independent of the strategy used to split).

Theorem (Garg, Madhusudan, N., Roth [POPL '16])

By only allowing splits with values in a range $[-c; c]$ and increasing c only if necessary, one obtains a decision tree learner that is guaranteed to find an inductive invariant if one can be expressed as a decision tree.

Performance of the Decision Tree Learner





Quiz: Can You Prove This Program Correct?

Example (Decision trees)

```
var s, x, y: int;
assume (x >= 0);
s := 0;
while (s < x)
invariant ???;
{
      s := s + 1;
}
y := 0;
while (y < s)
invariant ???;
{
      y := y + 1;
}
assert (y == x);
```

https://horn-ice.mpi-sws.org

Quiz: Can You Prove This Program Correct?

Example (Decision trees)

```
var s, x, y: int;
assume (x >= 0);
s := 0;
while (s < x)
invariant s <= x;
{
      s := s + 1;
}
y := 0;
while (y < s)
invariant y <= s;
{
      y := y + 1;
}
assert (y == x);
```

https://horn-ice.mpi-sws.org

Conclusion

Summary

Deductive software verification using inductive learning has been applied in practice with great success:

- GPUVerify (Houdini)
- Microsoft's Static Driver Verifier (Corral, Houdini)

Future Research

- Synthesizing predicates
- Learning from symbolic counterexamples
- Learning termination proofs
- Beyond software: verification of cyber-physical and AI-driven systems
- Beyond verification: program synthesis

The Max Planck Institute for Software Systems is offering opportunities:

- Internships
- PhD positions
- PostDoc positions

https://www.mpi-sws.org