Deductive Verification, the Inductive Way

Daniel Neider

MAX PLANCK INSTITUTE
FOR SOFTWARE SYSTEMS

ForMal Spring School
6 June 2019
The Need for Software Verification

1,715,430,778,504
The Need for Software Verification

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.

**Losses from Software Failures (USD)**

1,715,430,778,504

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

Tricentis

Daniel Neider: Deductive Verification, the Inductive Way
The Need for Software Verification

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.

**LOSSES FROM SOFTWARE FAILURES (USD)**

1,715,430,778,504

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

Tricentis

Daniel Neider: Deductive Verification, the Inductive Way
The Need for Software Verification

White Paper

The Software Fail Watch is an analysis of software failures found in a year’s worth of major articles. The result is an extraordinary reminder of the role software plays in our daily lives. 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 that affected the world’s population (3.7 billion people), $1.7 trillion in assets, and 314 countries. Scratching the surface—there are far more software defects in the world than we will ever be able to identify.

** LOSSES FROM SOFTWARE FAILURES (USD) **

1,715,430,778

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

---

**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
- ...
Logical Reasoning Meets Machine Learning

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)
Combine logical reasoning and machine learning

Alan Turing, Computing Machinery and Intelligence (1950)

**Deductive Reasoning** (Formal Verification)
- Infer conclusions by applying logical rules

**Inductive Reasoning** (Machine Learning)
- Infer conclusions by generalizing from data

Daniel Neider: Deductive Verification, the Inductive Way
Logical Reasoning Meets Machine Learning

Combine logical reasoning and machine learning

Alan Turing, Computing Machinery and Intelligence (1950)

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

Deductive Reasoning (Formal Verification)

Inductive Reasoning (Machine Learning)
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**)
1: var i, n: int;
2: assume (i == 0 && n >= 0);
3: while (i < n)
4: {
5:     i := i + 1;
6: }
7: assert (i == n);
1: var i, n: int;

2: assume (i == 0 && n >= 0); \hline precondition

3: while (i < n)
4: {
5:     i := i + 1;
6: }

7: assert (i == n); \hline postcondition
Running Example

1: var i, n: int;

2: assume (i == 0 && n >= 0); \textbf{precondition}

3: while (i < n)
4: { 
5: \hspace{1em} i := i + 1;
6: } 

7: assert (i == n); \textbf{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
Deductive Software Verification

The process involves:

1. Program
2. Annotations
3. Intermediate Language
4. Verification Condition Generator
5. VCs (formulas)
6. Theorem Prover

Outcome:
- ✓ ...
- ✓ ...
- ✗ ...

Daniel Neider: Deductive Verification, the Inductive Way
Deductive Software Verification

Program → Intermediate Language → Verification Condition Generator → Theorem Prover

Annotations → VCs (formulas) → Theorem Prover

Daniel Neider: Deductive Verification, the Inductive Way
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)
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, ...
Deductive Program Verification

Program Annotations

Intermediate Language

Verification Condition Generator

VCs (formulas)

Theorem Prover

✓ ... ✓ ... X ...
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
Proving a Program Correct

1: var x: int;
2: assume x >= 1;
3: x := x + 2;
4: assert x >= 3;
Proving a Program Correct

1: var x: int;
2: assume x >= 1; \quad x_2 \geq 1 \Rightarrow [x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3]]
3: x := x + 2;
4: assert x >= 3;
Verification Conditions: Straight-Line Code

Proving a Program Correct

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

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

Detecting Assertion Violations

1: var x: int;
2: assume x >= 1;
3: x := x + 2;
4: assert x < 3;
Verifying Conditions: Straight-Line Code

Proving a Program Correct

1. var x: int;
2. assume x >= 1;  \( x_2 \geq 1 \Rightarrow [x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3]] \)
3. x := x + 2;
4. assert x >= 3;

Detecting Assertion Violations

1. var x: int;
2. assume x >= 1;  \( x_2 \geq 1 \Rightarrow [x_3 = x_2 + 2 \Rightarrow [x_3 < 3]] \)
3. x := x + 2;
4. assert x < 3;
Verification Conditions: Straight-Line Code

Proving a Program Correct

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

\text{\[ x_2 \geq 1 \Rightarrow [x_3 = x_2 + 2 \Rightarrow [x_3 \geq 3]] \]}

Detecting Assertion Violations

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

\text{\[ x_2 \geq 1 \Rightarrow [x_3 = x_2 + 2 \Rightarrow [x_3 < 3]] \]}

▶ A satisfying assignment for the negation of the VC provides input to the program that violates the assertion
1: var x, y: int;
2: if (x < 0) {
3:     y := -x;
4: } else {
5:     y := x;
6: }
7: assert y >= 0;
1: var x, y: int;
2: if (x < 0) {
3:     y := -x;
4: } else {
5:     y := x;
6: }
7: assert y >= 0;

\[
x_2 < 0 \Rightarrow \left[ y_3 = -x_2 \Rightarrow [y_3 \geq 0] \right]
\land 
\left[ \neg(x_2 < 0) \Rightarrow \left[ y_5 = x_2 \Rightarrow [y_5 \geq 0] \right] \right]
\]
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?

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

How can we verify the following program?

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

First Solution

Inline foo
Method Contracts – Idea

Example

How can we verify the following program?

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

Second Solution

Write contract/specification $P$ of foo

- Assume $P$ when checking bar
  ```java
  bar() {
      ...
      assume P;
      ...
  }
  ```

- Guarantee $P$ when checking foo
  ```java
  foo() {
      ...
      assert P;
  }
  ```
1: procedure M(x, y)  
   returns (r, s)  
   requires P  
   ensures Q
2: {  
3:     S;  
4: }  
5: call a, b := M(c, d);
1: procedure M(x, y) 1: assume P;
  returns (r, s) 2: S;
 requires P 3: assert Q;
 ensures Q

2: {
3:   S;
4: }

5: call a, b := M(c, d);
Method Contracts – Details

1: procedure M(x, y)  
   returns (r, s)  
   requires P  
   ensures Q

2: {  
3:     S;  
4: }

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

1: assume P;  
2: S;  
3: assert Q;

4: x’ := c;   y’ := d;  
5: assert P’;  
6: assume Q’;  
7: 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

1: var i, n: int;
2: assume (i == 0 && n >= 0);
3: while (i < n)
4: {
5:   i := i + 1;
6: }
7: assert (i == n);
Loops and Loop Unrolling

1: var i, n: int;
2: assume (i == 0 && n >= 0);
3: while (i < n)
4: {
5: i := i + 1;
6: }
7: assert (i == n);

Loop Unrolling

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

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

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

1: var i, n: int;
2: assume (i == 0 && n >= 0);
3: while (i < n)
4: {
5:     i := i + 1;
6: }
7: 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

1: var x, y: int;
2: x := -1;
3: while (x < 0)
4: invariant ???;
5: {
6: x := x + y;
7: y := y + 1;
8: }
9: assert (y > 0);

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

Daniel Neider: Deductive Verification, the Inductive Way
Example

1: var x, y: int;
2: x := -1;
3: while (x < 0)
4:   invariant x >= 0 ==> y > 0;
5: {
6:     x := x + y;
7:     y := y + 1;
8: }
9: assert (y > 0);

https://horn-ice.mpi-sws.org
2. Inductive Inference of Annotations
Deductive Program Verification

Program

Annotations

Intermediate Language

Verification Condition Generator

VCs (formulas)

Theorem Prover

✓ . . .

✓ . . .

✗ . . .

Daniel Neider: Deductive Verification, the Inductive Way
$\begin{align*}
loc &= l_{13} \\
x &= 2 \\
y &= 17 \\
\vdots
\end{align*}$

$x = x + 1$

$\begin{align*}
loc &= l_{14} \\
x &= 3 \\
y &= 17 \\
\vdots
\end{align*}$
\( \phi_{\text{pre}} \)  

\( \neg \phi_{\text{post}} \)  

\( \text{Init} \)  

\( \text{Bad} \)
Daniel Neider: Deductive Verification, the Inductive Way
Invariant

1. $\text{Init} \subseteq \text{Inv}$
2. $\text{Bad} \cap \text{Inv} = \emptyset$
3. $\text{Step}(\text{Inv}) \subseteq \text{Inv}$

(includes initial configurations)
(excludes bad configurations)
(is inductive)
Invariant Synthesis

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

Inductive reasoning engine

candidate invariant $H$

Deductive reasoning engine

Learner
(knows examples)

Teacher
(knows program)

counterexample

Daniel Neider: Deductive Verification, the Inductive Way
1. \( \text{Init} \subseteq H \) (includes initial configurations)
2. \( \text{Bad} \cap H = \emptyset \) (excludes bad configurations)
3. \( \text{Step}(H) \subseteq H \) (is inductive)
Refuting Non-Invariants

1. Positive counterexample: if \( \text{Init} \not\subseteq H \), report \( c \in \text{Init} \setminus H \)
Refuting Non-Invariants

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

2. Negative counterexample: if $\text{Bad} \cap H \neq \emptyset$, report $c \in \text{Bad} \cap H$
Counterexamples

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

Teacher (knows program)

candidate invariant $H$

positive, negative, or implication counterexample

Learner

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

- $c \in H$ for each $c \in Pos$
- $c \notin H$ for each $c \in Neg$
- $c \in H$ implies $c' \in H$ for each $c \Rightarrow c' \in Impl$
Teacher / Tool Architecture

C/C++/Java program

SMACK

Boogie program

SMT solver

Teacher (Boogie)

Counterexample

ICE Learner

Verified

Failed

Model

VC

Hypothesis
We assume that a finite set $P$ of predicates is given that allows separating any pair of program configurations in the ICE sample. We will relax this restriction later.
Implementing a Learner

\[
x + y \leq -2
\]

\[
x \leq -10
\]

We assume that a finite set \( P \) of predicates is given that allows separating any pair of program configurations in the ICE sample. We will relax this restriction later.

Daniel Neider: Deductive Verification, the Inductive Way
Implementing a Learner

Simplification

- We assume that a finite set $P$ 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
The Houdini Algorithm

Example

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

- \((1, 0, 1, 1, 0), +\); \((1, 1, 1, 0, 1), +\)
- \((1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)\)
- \((0, 1, 0, 1, 1), -\)
The Houdini Algorithm

Example

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

\[ \begin{align*}
\triangleright (1, 0, 1, 1, 0), & \quad +; (1, 1, 1, 0, 1), & \quad + \\
\triangleright (1, 1, 1, 0, 0) & \rightarrow (0, 1, 1, 1, 1) \\
\triangleright (0, 1, 0, 1, 1), & \quad -
\end{align*} \]

Algorithm 1: The Houdini algorithm

1. \( X \leftarrow P \) (i.e., \( \varphi_X = p_1 \land \ldots \land p_n \))
2. while \( X \) is not consistent with \( Pos \) do
3. \hspace{1em} Remove predicates \( p_i \) from \( X \) that “occur as 0” in a positive example
4. \hspace{1em} if the left-hand-side of an implication in \( Impl \) is satisfied then
5. \hspace{2em} mark the right-hand-side as positive
6. 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. $(1, 0, 1, 1, 0), +; (1, 1, 1, 0, 1), +$
2. $(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)$
3. $(0, 1, 0, 1, 1), −$

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

Algorithm 1: The Houdini algorithm

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

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

- $\text{(1, 0, 1, 1, 0), +; (1, 1, 1, 0, 1), +}$
- $\text{(1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1)}$
- $\text{(0, 1, 0, 1, 1), -}$

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

Algorithm 1: The Houdini algorithm

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

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

\[
\begin{align*}
\text{Example: } & (1, 0, 1, 1, 0), +; (1, 1, 1, 0, 1), + \\
\text{Example: } & (1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1) \\
\text{Example: } & (0, 1, 0, 1, 1), -
\end{align*}
\]

\[p_1 \land p_2 \land p_3 \land p_4 \land p_5\]

Algorithm 1: The Houdini algorithm

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

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

\[ (1, 0, 1, 1, 0), +; (1, 1, 1, 0, 1), + \]
\[ (1, 1, 1, 0, 0) \rightarrow (0, 1, 1, 1, 1) \]
\[ (0, 1, 0, 1, 1), - \]

\[ p_1 \land p_2 \land p_3 \land p_4 \land p_5 \]

Algorithm 1: The Houdini algorithm

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

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

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

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

Algorithm 1: The Houdini algorithm

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

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

- $\langle 1, 0, 1, 1, 0 \rangle$, $+$; $\langle 1, 1, 1, 0, 1 \rangle$, $+$
- $\langle 1, 1, 1, 0, 0 \rangle \rightarrow \langle 0, 1, 1, 1, 1 \rangle$
- $\langle 0, 1, 0, 1, 1 \rangle$, $-$

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

Algorithm 1: The Houdini algorithm

1. $X \leftarrow \mathcal{P}$ (i.e., $\varphi_X = p_1 \land \ldots \land p_n$)
2. while $X$ is not consistent with Pos do
3. Remove predicates $p_i$ from $X$ that "occur as 0" in a positive example
4. if the left-hand-side of an implication in Impl is satisfied then
5. mark the right-hand-side as positive
6. 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 $|S| \cdot |\mathcal{P}|$. 
The Houdini Algorithm

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 $|S| \cdot |\mathcal{P}|$.

▶ 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 $|S| \cdot |\mathcal{P}|$.

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

- **Disadvantage:**
  Houdini is independent of negative examples/the post-condition
B. Sorcar
The Sorcar Algorithm

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

1. static \( R \leftarrow \emptyset \)

2. Procedure Sorcar\((S, P, R)\):

3. \( X \leftarrow \text{Houdini}(S, P) \)  // Takes care of positive examples in \( Pos \)

4. while \( X \cap R \) is not consistent with \( S \) do

5. \hspace{1em} foreach negative example in \( Neg \) not consistent with \( X \cap R \) do

6. \hspace{2em} Add “relevant” predicate from \( X \setminus R \) to \( R \)

7. \hspace{1em} foreach implication in \( Impl \) not consistent with \( X \cap R \) do

8. \hspace{2em} Mark the left-hand-side as negative

9. return \( X \)
Theorem (Madhusudan, N., Saha)

Sorcar learns an inductive invariant in at most $2 \cdot |P|$ rounds if one is expressible as a conjunction over $P$. The time spend in each round is proportional to $|S| \cdot f(|P|)$, 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 \(|S| \cdot f(|\mathcal{P}|)\), 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: \(f(n) \in \mathcal{O}(n^3)\)
Quiz: Can You Prove This Program Correct?

Example (Houdini/Sorcar)

1: var x, y, z: int;
2: assume (x < y);
3: z := y;
4: while (z > x)
5:  invariant ???;
6:  {
7:    z := z - 1;
8:  }
9: assert (x <= z);

https://horn-ice.mpi-sws.org
Example (Houdini/Sorcar)

1: `var x, y, z: int;`
2: `assume (x < y);`
3: `z := y;`
4: `while (z > x)`
5: `invariant x <= z && y > x;`
6: `{`
7: `z := z - 1;`
8: `}`
9: `assert (x <= z);`

https://horn-ice.mpi-sws.org
C. ICE Learning Using Decision Trees
### Decision Trees

<table>
<thead>
<tr>
<th>Sunny?</th>
<th>Hot?</th>
<th>Windy?</th>
<th>Play Tennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>

**Sunny?**

- **+**

**Windy?**

- **-**
- **+**
### Decision Trees

<table>
<thead>
<tr>
<th>$x \geq 0$</th>
<th>$\text{res} &lt; 0$</th>
<th>$x \leq \text{res}$</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
x & \geq 0 \lor \neg (x \geq 0) \land \neg (x \leq \text{res})
\end{align*}\]
### Decision Trees

<table>
<thead>
<tr>
<th>$x \geq 0$</th>
<th>$res &lt; 0$</th>
<th>$x \leq res$</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>+</td>
</tr>
</tbody>
</table>

$x \geq 0$ and $x \leq res$ are the conditions for the decision tree. The expression $x \geq 0 \lor \neg(x \geq 0) \land \neg(x \leq res)$ captures the logic of the tree.
Learning Decision Trees
Learning Decision Trees

\[ x - 0.1y \leq 0 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]

\[ x + y \leq 2 \]
Learning Decision Trees

\[
x - 0.1y \leq 0
\]

\[
x \leq -4
\]

\[
x + y \leq 2
\]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]
Learning Decision Trees

$x - 0.1y \leq 0$

$x \leq -4$
Learning Decision Trees

$x - 0.1y \leq 0$

$x \leq -4$
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]
Learning Decision Trees

\[ x - 0.1y \leq 0 \]

\[ x \leq -4 \]

\[ x + y \leq 2 \]
Goal: Split such that the resulting subsamples are as “pure” as possible

Entropy (Shannon, 1948)

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

$$H(S) = -\left[ P(+) \log_2 P(+) + P(–) \log_2 P(–) \right],$$

where $P(+) = \frac{|\text{Pos}|}{|\text{Pos}| + |\text{Neg}|}$ and $P(–) = \frac{|\text{Neg}|}{|\text{Pos}| + |\text{Neg}|}$.

Information Gain

A split of $S$ into $S_1$ and $S_2$ results in an information gain of

$$G(S, S_1, S_2) = H(S) - (H(S_1) + H(S_2)).$$
How to Split?

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

Entropy (Shannon, 1948)
Let \( S = (\text{Pos}, \text{Neg}) \) be a sample. Then, entropy is defined as

\[
H(S) = - \left[ P(+) \log_2 P(+) + P(-) \log_2 P(-) \right],
\]

where \( P(+) = \frac{|\text{Pos}|}{|\text{Pos}| + |\text{Neg}|} \) and \( P(-) = \frac{|\text{Neg}|}{|\text{Pos}| + |\text{Neg}|} \).
How to Split?

**Goal:** Split such that the resulting subsamples are as “pure” as possible

**Entropy (Shannon, 1948)**

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

$$H(S) = - \left[ P(+) \log_2 P(+) + P(-) \log_2 P(-) \right],$$

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

**Information Gain**

A split of $S$ into $S_1$ and $S_2$ results in an information gain of

$$G(S, S_1, S_2) = H(S) - (H(S_1) + H(S_2))$$
How to Split in the Presence of Implications

Let $c \in \mathbb{R}$ be a constant and $\gamma$ be the number of implications that go from $S_1$ to $S_2$ or vice versa.

$$\text{Penalty}(S, S_1, S_2) = \text{Penalty}(S, S_1, S_2) - c \cdot \gamma$$

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 \( \gamma \) be the number of implications that go from \( S_1 \) to \( S_2 \) or vice versa

\[
G_{\text{penalty}}(S, S_1, S_2) = G(S, S_1, S_2) - c \cdot \gamma
\]

Daniel Neider: Deductive Verification, the Inductive Way
A fixed set of predicates might be insufficient to construct a consistent decision tree.

\[ x \leq 3 \]

Diagram showing a line segment from 0 to 4 with a dotted line at 3, indicating the predicate \( x \leq 3 \).
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.
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

Imperative programs
SV Comp 2016

Recursive programs
SV Comp 2018

Time CPAChecker in s

Time ICE in s

Time Horn-ICE in s

Time Automizer in s

Daniel Neider: Deductive Verification, the Inductive Way
Quiz: Can You Prove This Program Correct?

Example (Decision trees)

1: var s, x, y: int;
2: assume (x >= 0);
3: s := 0;
4: while (s < x)
5: invariant ???;
6: {
7:     s := s + 1;
8: }
9: y := 0;
10: while (y < s)
11: invariant ???;
12: {
13:     y := y + 1;
14: }
15: assert (y == x);
Quiz: Can You Prove This Program Correct?

Example (Decision trees)

1: var s, x, y: int;
2: assume (x >= 0);
3: s := 0;
4: while (s < x)
5: invariant s <= x;
6: {
7: s := s + 1;
8: }
9: y := 0;
10: while (y < s)
11: invariant y <= s;
12: {
13: y := y + 1;
14: }
15: 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