## *Formal Verification – Robust and Efficient Code*
## *Lecture **1***

# Introduction to FV

**Kim Albertsson**

**Luleå University of Technology**

**Inverted CERN School of Computing, 29 February – 2 March 2016**

# Introduction

- **Kim Albertsson**
  - M.Sc. Engineering Physics and Electrical Engineering
  - Currently studying for M.Sc. Computer Science
- **Research interests**
  - Automation
  - Machine learning
- **Formal methods**
  - Great way to automate tedious tasks

Image from Wikimedia Commons, the free media repository

# Formal Methods

- **Managing complexity**
- **“…mathematically based languages, techniques, and tools for specifying and verifying … systems.”**
  Edmund M. Clarke et al.
- **Reveals inconsistencies and ambiguities**

# Formal Methods

- **Specification + Model + Implementation enables verification**
- **Proving properties for a system**
  - Will my algorithm sort the input?
  - Will my soda pop be delivered in time?
  - Can the LHC interlock system end up in an inconsistent state?

# Overview Series

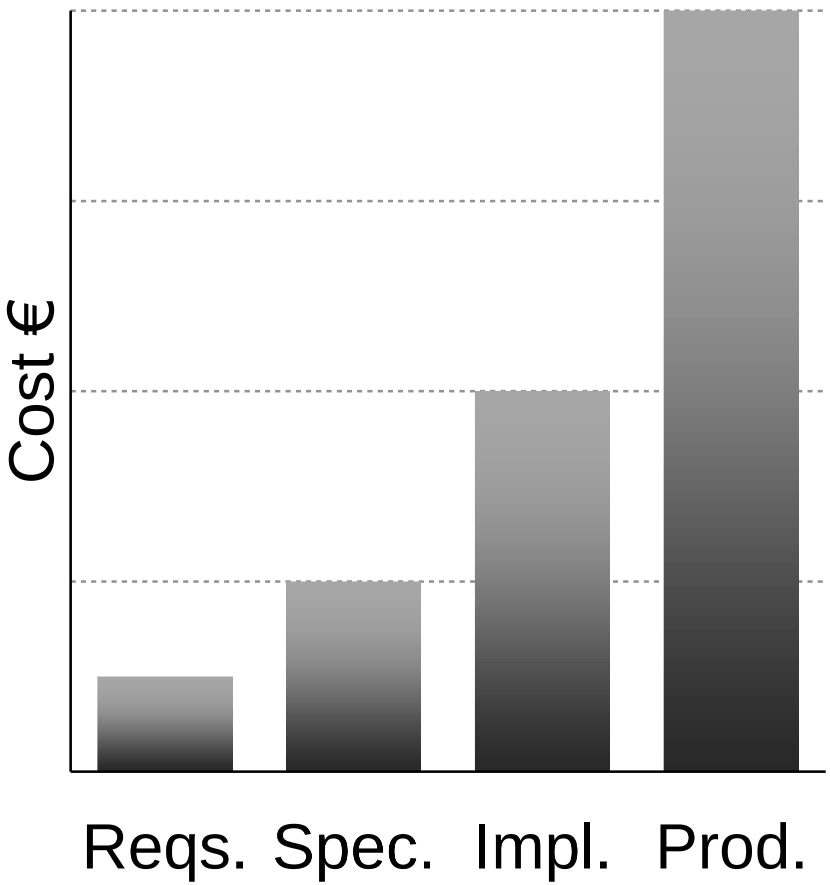
- **Approaches**
    - Model Checking and Theorem Proving
    - Logic and automation
    - How to build an ecosystem
- **Application**
    - Robust code
    - Robust and Efficient code

# Introduction Lecture 1

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving
- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

# Introduction Lecture 1

- **Errors discovered late**
  - Expensive
- **FV requires**
  - More effort for specification
- **FV gives**
  - Feedback on inconsistencies
  - Reduces late errors
  - Reduces total time



# Overview

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving
- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

# Practical Examples

- **Space flight, Ariane 5 - 501**
- **Avionics, Lockheed**

# Ariane 5 - 501

- **Ariane 5**
  - Reused inertial reference system of Ariane 4
  - Greater horizontal acceleration
  - 64-bit float cast to 16-bit integer overflowed
- **Expensive bug discovery**

Image from Wikimedia Commons, the free media repository

# Lockheed C130J

- **C130J Hercules**
  - Secure low-level flight control code
- **Review concluded**
  - Improved quality of product
  - Decreased cost of development

Image from Wikimedia Commons, the free media repository

# Specification

- **Concise, high level description of behaviour**
- **Varying degree of rigour**
  - Informal; written English
  - Formal; using standardised syntax
- **Required for verification**

A **verified program** has demonstrated that specification is consistent with implementation

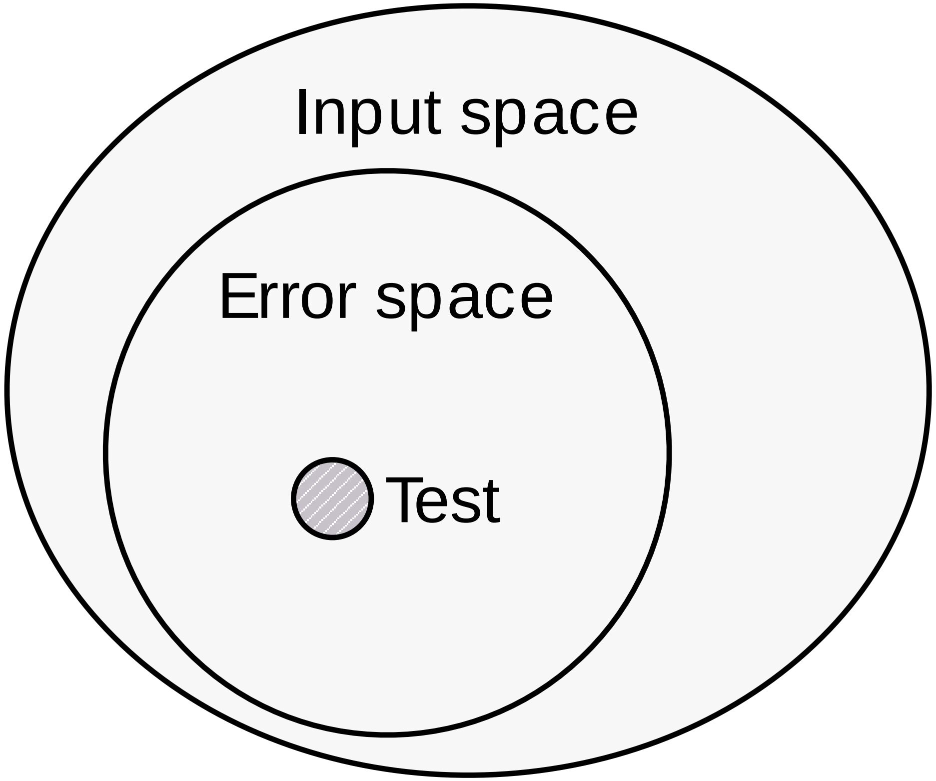
# Specification

- **Example in whyML**

```
let bubblesort (a: array int) =
   requires { … }
   ensures { sorted a }
   …
```

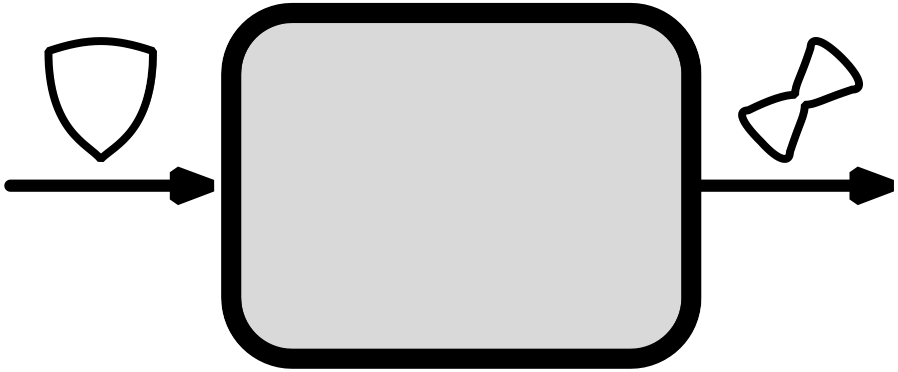
# Testing vs. Proving

- **Testing**
  - Limited to particular point in input space
- **Proving**
  - Reason about complete input space

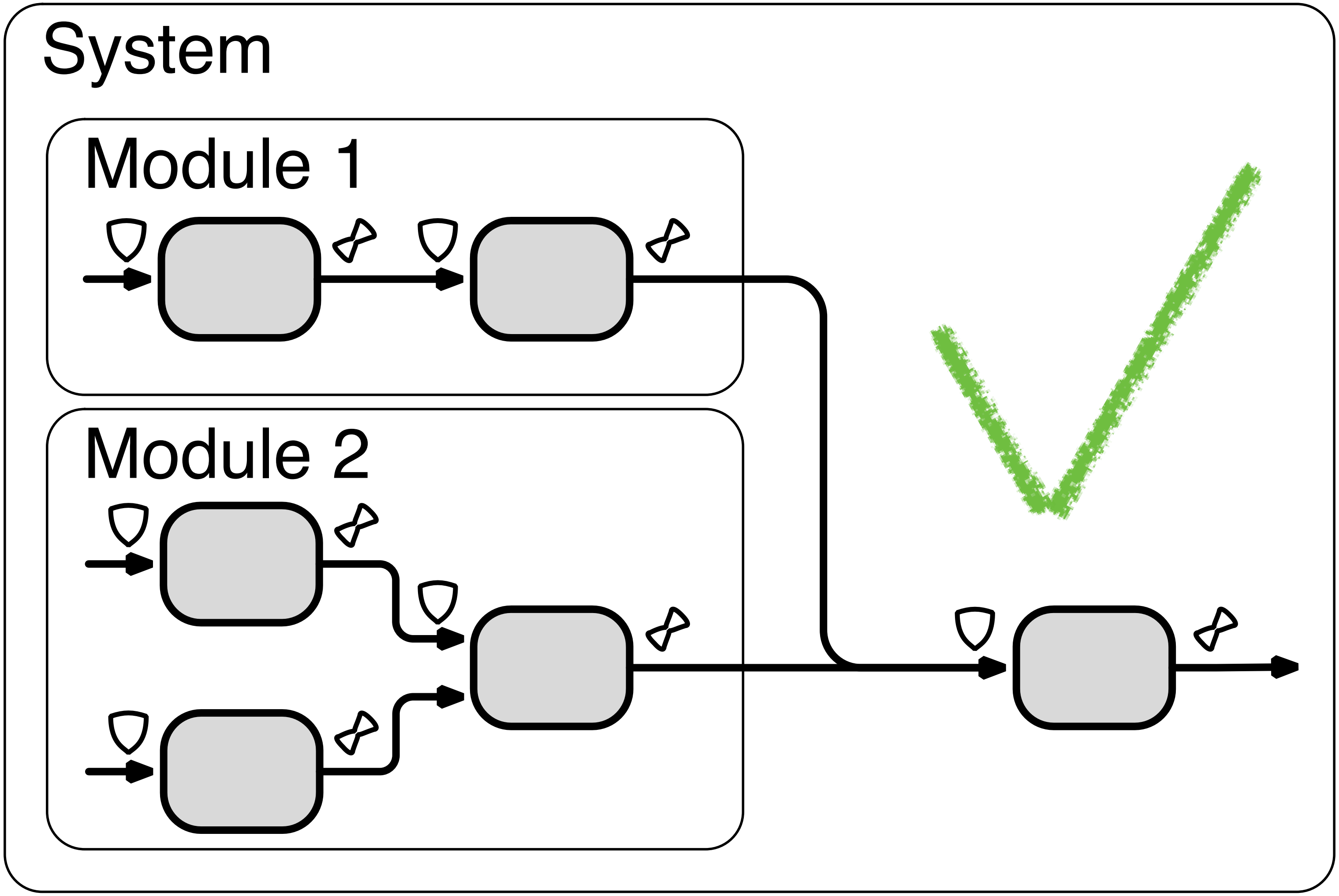


# Contract

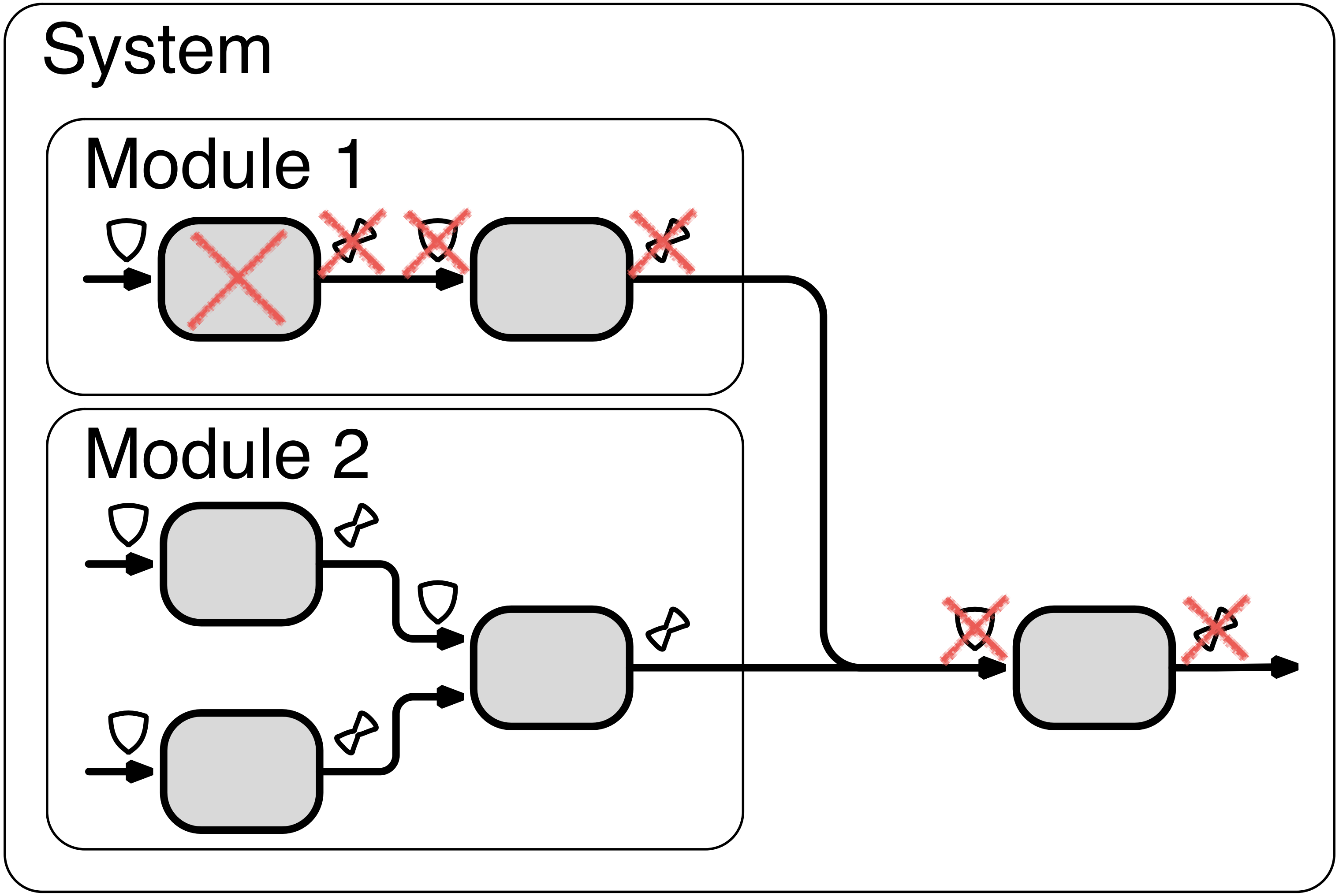
- **A contract consists of**
  - pre-condition, that must hold before execution
  - post-condition, that must hold after execution
- **Weakest precondition**
  - The set of least restrictive of pre-conditions that still ensure the post-condition



# System Composition



# System Composition

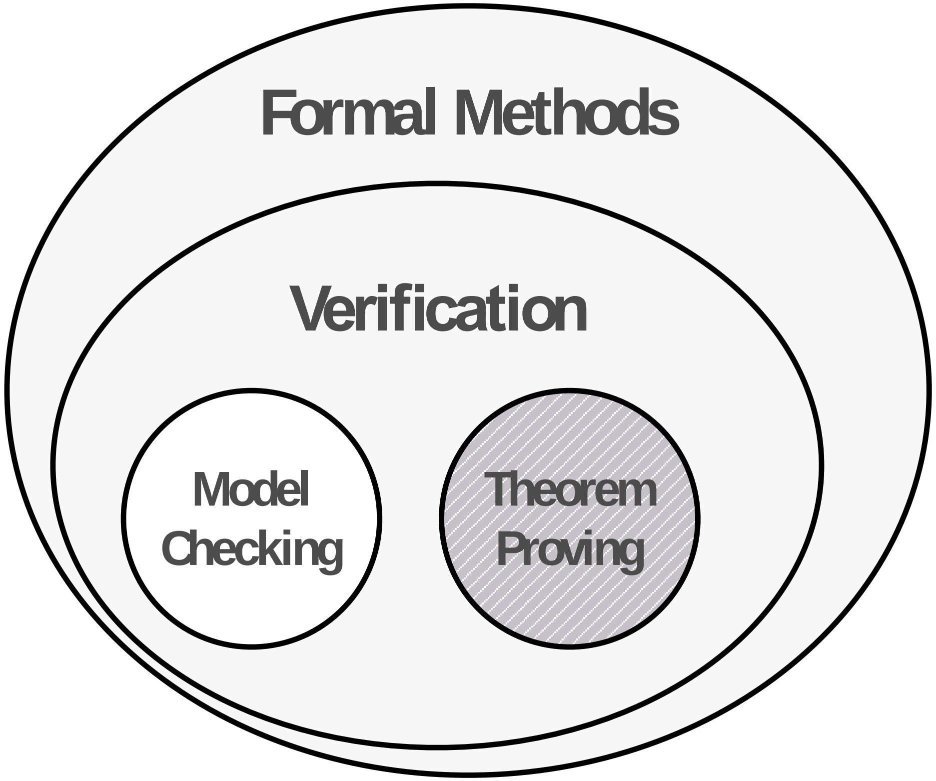


# Overview

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving
- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

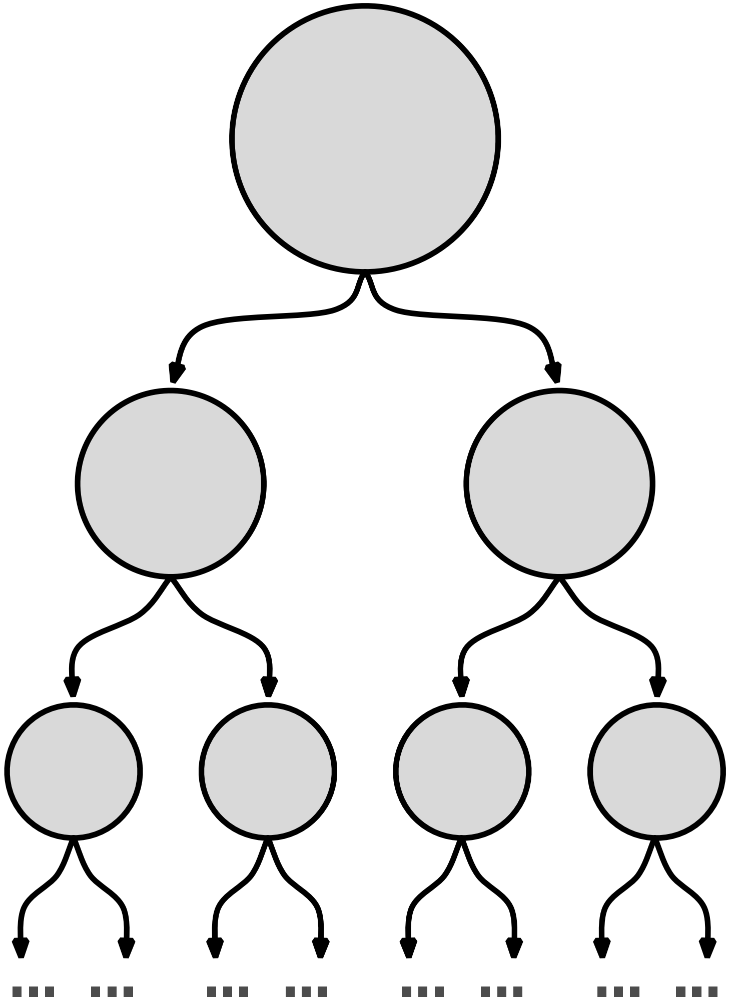
# Methods of FV

- **Model Checking**
  - General tool
  - Tricky problem
- **Theorem Proving**
  - General tool
  - Tricky problem



# Model Checking

- **Verifies that property holds for all states**
  - Explores all model states
  - Suitable for finite state models
- **Program verification**
  - Model program as a graph



# Model Checking

- **Pros**
  - Easy start-up with when applicable
- **Cons**
  - Requires clever algorithms
  - Must start anew each time

# Theorem Proving

- **Define a formal logic system**
  - With inference rules
- **Proofs are derived by applying rules**
  - By hand or by machine
- **Turns out full automation is a tricky problem**
  - Interactive provers, or proof assistants

# Overview

- **Introduction**
  - Practical examples
  - Specification
  - Testing vs. Proving
  - Contracts
- **Methods of Formal Verification**
  - Model Checking
  - Theorem Proving
- **In-depth Theorem Proving**
  - First Order Logic
  - Satisfiability Modulo Theories
  - Verification Platforms

# First Order Logic

- **Object variables**
  - Refers to a unique object
  - Names like x, jim, 1
- **Predicates**
  - Relations like =, <, AtHome
- **Connectives**
  - $\wedge, \vee, \rightarrow, =$
- **Functions**
  - Complex names
  - mother(jim)
  - head(x)
  - 1+1
- **Sentence**
  - Combination of the above
  - mother(jim) = kimberly
  - head(x) < 1

# First Order Logic

**Implication elimination**

$$\begin{array}{l} a \to b,\ a \\ \hline \therefore b \end{array}$$

**Negation Introduction**

$$\begin{array}{l} a \vdash b,\ a \vdash \neg b \\ \hline \therefore \neg a \end{array}$$

**Conjugation Introduction**

$$\begin{array}{l} a\ ,\ b \\ \hline \therefore a \land b \end{array}$$

**Disjunctive Syllogism**

$$\begin{array}{l} a \lor b,\ \neg a \\ \hline \therefore b \end{array}$$

$$\begin{array}{r|lr}
(1) & x \land \neg y \to y \lor z & \\
(2) & y \to \neg x & \\
(3) & x & \\ \hline
(4) & \qquad \left|\; y \right. & \text{assump.} \\
(5) & \qquad \left|\; \underline{\neg x\;\;} \right. & \text{impl. elim. (2, 3)} \\
(6) & \qquad \left|\; \neg y \right. & \text{neg. intro. (3, 5)} \\
(7) & x \land \neg y & \text{conj. intro. (3, 7)} \\
(8) & y \lor z & \text{impl. elim. (1, 7)} \\
(9) & z & \text{dis. syll. (6, 8)}
\end{array}$$

# First Order Logic

- **Quantifiers**
  - ∀ — for all
  - ∃ — existence
- **Usage**
  - ∀x. Programmer(x) → Happy(x)
  - ∃x. Programmer(x) → ProducesBugs(x)
  - ∃x∀y. Loves(x, y)

# Hoare Logic

- P,  Precondition
- Q,  Postcondition
- C, Command
- When C is executed under P, Q is guaranteed
- Rules required for each action of a language

$$\{P\}\, C\, \{Q\}$$

$$\{x + 1 < N\}\, x := x + 1\, \{x \leq N\}$$

# Hoare Logic

- **Composition**
  - Allows commands to be executed in sequence

$$\frac{\{P\}\,S\,\{Q\}\quad,\quad\{Q\}\,T\,\{R\}}{\{P\}\,S\ ,\ T\,\{R\}}$$

- **While rule**
  - Models while-statement
  - P is called loop invariant
  - Can be extended to prove termination

$$\frac{\{P\wedge B\}\,S\,\{Q\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\,\{\neg B\wedge P\}}$$

CERN School of Computing

# Theories

- **Theory**
  - Set of sentences that are assumed to be true, T
- **Axiom**
  - Each element in T
- **Theorem**
  - Any sentence that can be concluded from the theory
- **Example**
  - Peano arithmetic, theory of lists…

# Satisfiability Modulo Theories

- **Hoare logic**
  - To reason about programs

$$\{P\}\,C\,\{Q\}$$

- **Reasoning expressed**
  - First order logic

$$\{x+1<N\}\,x:=x+1\,\{x\leq N\}$$

- **Verification conditions (VC)**

# Satisfiability Modulo Theories

$$(x \wedge \neg y) \wedge (y \vee z)$$

- **Similar to Binary Satisfiability problem, SAT**
  - One of the first problems to be shown to be NP-complete
  - Find assignment of x, y, z … so that the expression is satisfied
- **SAT variables are boolean**
  - SMTs are extended to handle FOL constructs
- **Verifying a proof is easy**
  - Check each step for validity assuming our logic system

# Satisfiability Modulo Theories

- **Formulas are considered w.r.t background theory**
  - For formula F, assume ¬F
  - F is valid when ¬F is not satisfiable under T
- **Modern solvers use heuristics**
  - Improves performance for specific theories
  - At the cost of general performance

# Satisfiability Modulo Theories

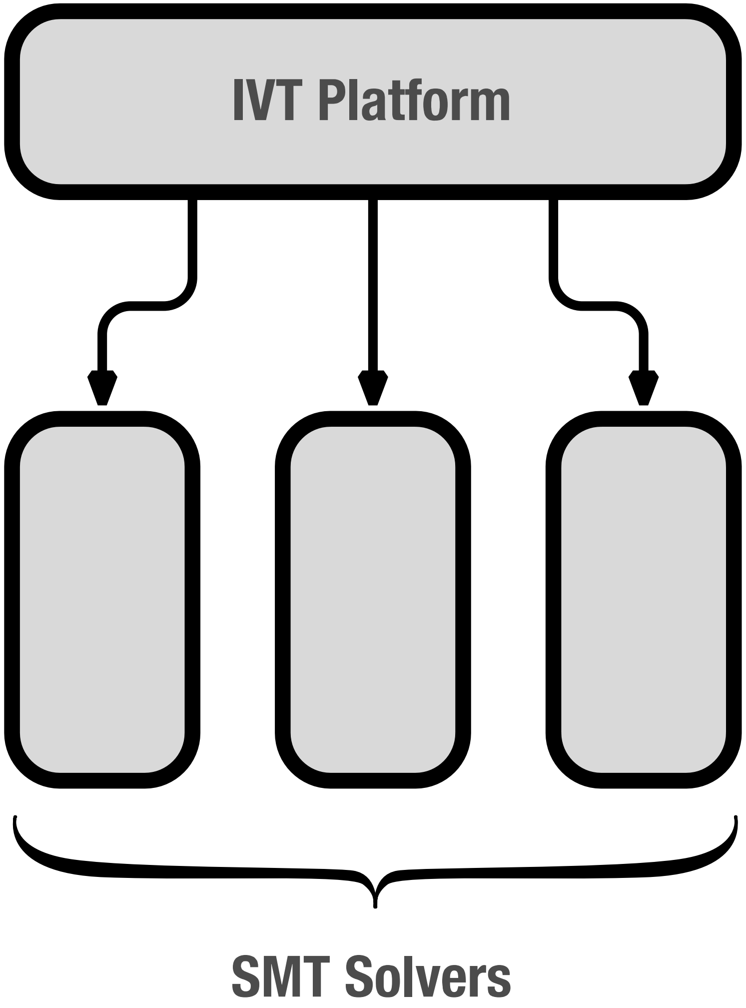
- **Example in alt-ergo**
  - Try for yourself https://alt-ergo.ocamlpro.com/try.php

```
logic x, y, z: prop
axiom a_1:
   (x and not y) -> (y or z)
axiom a_2:
   y  -> not x
axiom a_3:
   x
goal g_1: z

# [answer] Valid (0.0060 seconds) (3 steps)
```

# Verification Platforms

- **Unified interface for provers**
- **Generates verification conditions (VCs)**
  - Discharged by any compliant prover, interactive or automatic
- **Examples**
  - why3, boogie



# Example

```
let insertion_sort (a: array int) =
  for i = 1 to length a - 1 do
    let v = a[i]  in
    let j = ref i in
    while !j > 0 && a[!j - 1] > v do
      a[!j] <- a[!j - 1];
      j := !j - 1
    done;
    a[!j] <- v
  done
end
```

# Example

```
let insertion_sort (a: array int) =
   ensures { sorted a }
   for i = 1 to length a - 1 do
      invariant { sorted_sub a 0 i }
      let v = a[i] in
         let j = ref i in
         while !j > 0 && a[!j - 1] > v do
            invariant { 0 <= !j <= i }
            invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                      -> k1 <> !j
                      -> k2 <> !j -> a[k1] <= a[k2] }
            invariant { forall k: int. !j+1 <= k <= i
                     -> v < a[k] }
            a[!j] <- a[!j - 1];
            j := !j - 1
         done;
         a[!j] <- v
      done
end
```

# Example

```
let insertion_sort (a: array int) =
    ensures { sorted a }
    for i = 1 to length a - 1 do
        invariant { sorted_sub a 0 i }
        let v = a[i] in
            let j = ref i in
            while !j > 0 && a[!j - 1] > v do
                invariant { 0 <= !j <= i }
                invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                            -> k1 <> !j
                            -> k2 <> !j -> a[k1] <= a[k2] }
                invariant { forall k: int. !j+1 <= k <= i
                           -> v < a[k] }
                a[!j] <- a[!j - 1];
                j := !j - 1
            done;
            a[!j] <- v
        done
end
```

# Example

```
let insertion_sort (a: array int) =
    ensures { sorted a }
    for i = 1 to length a - 1 do
        invariant { sorted_sub a 0 i }
        let v = a[i] in
            let j = ref i in
            while !j > 0 && a[!j - 1] > v do
                invariant { 0 <= !j <= i }
                invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                          -> k1 <> !j
                          -> k2 <> !j -> a[k1] <= a[k2] }
                invariant { forall k: int. !j+1 <= k <= i
                         -> v < a[k] }
                a[!j] <- a[!j - 1];
                j := !j - 1
            done;
            a[!j] <- v
        done
end
```

# Example

- i, j are loop variables
- v holds the current element
- We are sorting subsegment [0, i]

$$0 \leq j \leq i$$

$$\begin{aligned} &\forall k_1 \forall k_2 . \\ &\quad 0 \leq k_1 \leq k_2 \leq i \\ &\quad \rightarrow k_1 \neq j \\ &\quad \rightarrow k_2 \neq j \\ &\quad a\left[k_1\right] \leq a\left[k_2\right] \end{aligned}$$

$$\begin{aligned} &\forall k .\ j+1 \leq k \leq i \\ &\quad \rightarrow v < a\left[k\right] \end{aligned}$$

# Example

```
let insertion_sort (a: array int) =
   ensures { sorted a }
   for i = 1 to length a - 1 do
       invariant { sorted_sub a 0 i }
       let v = a[i] in
          let j = ref i in
          while !j > 0 && a[!j - 1] > v do
             invariant { 0 <= !j <= i }
             invariant { forall k1 k2: int. 0 <= k1 <= k2 <= i
                       -> k1 <> !j
                       -> k2 <> !j -> a[k1] <= a[k2] }
             invariant { forall k: int. !j+1 <= k <= i
                      -> v < a[k] }
             a[!j] <- a[!j - 1];
             j := !j - 1
          done;
          a[!j] <- v
       done
end
```

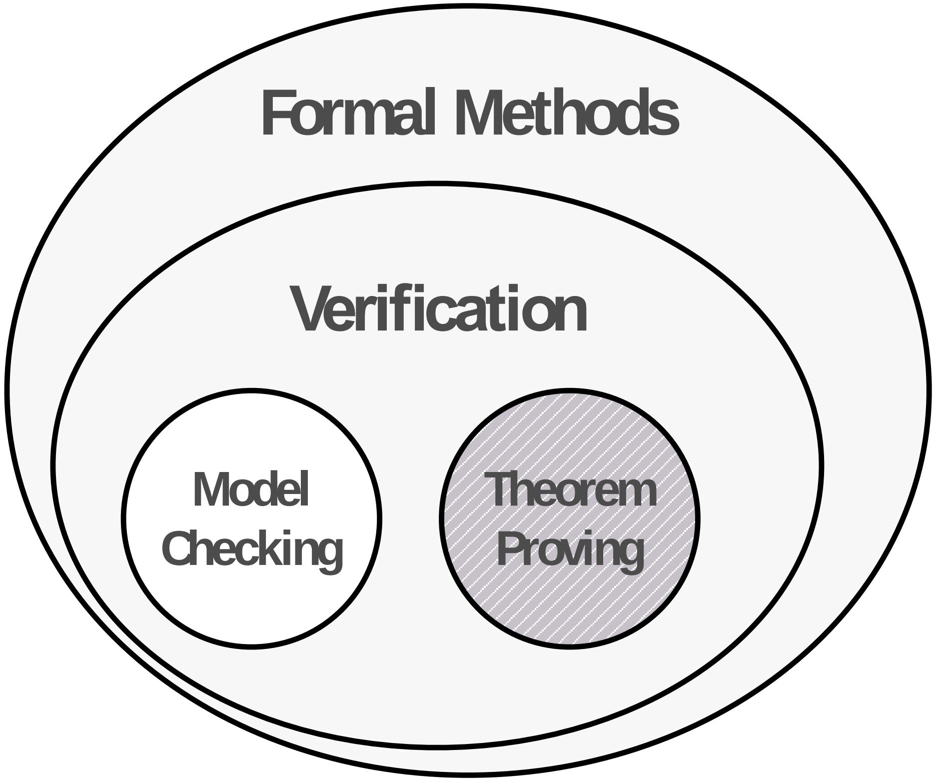
# Recap of Lecture 1

- **FV manages complexity**
- **Finds errors early**
  - Save money, time and possibly life
- **Proof is a demonstration**
  - Consistency of specification and implementation

Image from Wikimedia Commons, the free media repository

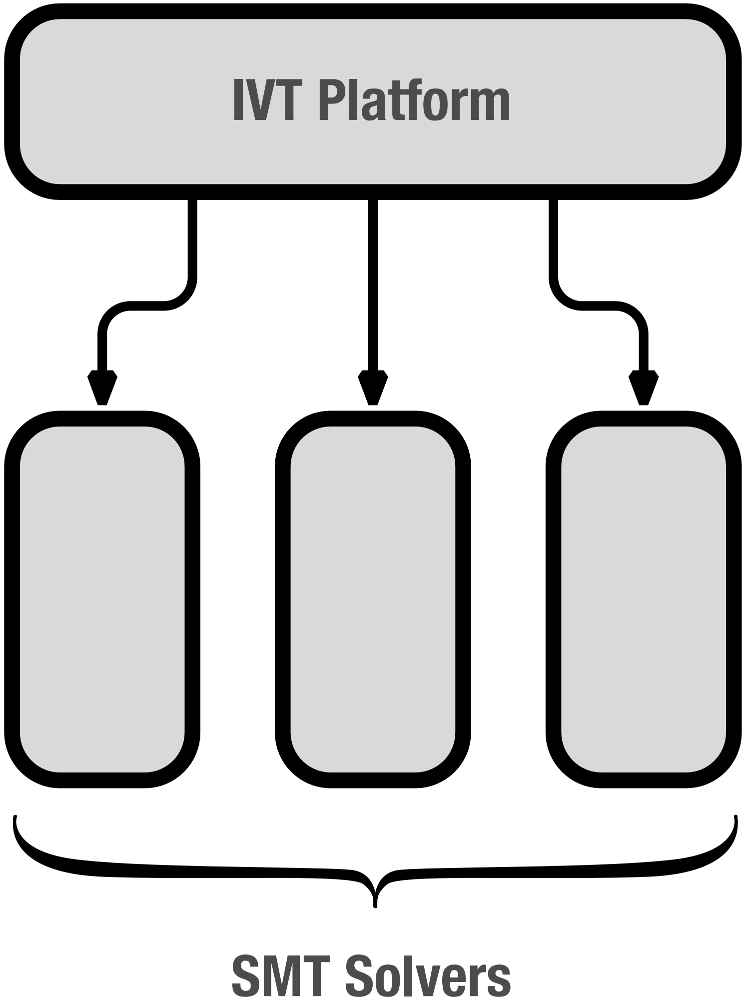
# Recap of Lecture 1

- **Proving properties about your system**
- **Model Checking**
  - Exhaustive search of state space
- **Theorem Proving**
  - Deductive approach



# Recap of Lecture 1

- **Theorem Proving**
  - Based on First Order Logic
  - Satisfiability Modulo Theories (SMT) solvers to find application of rules
  - IVT platforms front-end with many SMT backends



# Thank you