

Formal Verification (CS-550)

Viktor Kuncak, EPFL

<https://lara.epfl.ch/w/fv>

Formal Verification

Goal: rigorously prove that computer systems “do what they should do”

“do what they should do” = satisfy a specification

Define a mathematically rigorous notion of a system satisfying a specification

Use combination of automated tools and human effort to construct the proof

We will learn how to use and build tools for computer-aided formal verification

Comparison to Testing

We test computer systems and we should.

In formal verification, we go beyond: build models and prove that the systems work.

Comparison to Testing

We test computer systems and we should.

In formal verification, we go beyond: build models and prove that the systems work.

(But, I have a **really fast** fuzz tester!)

Comparison to Testing

We test computer systems and we should.

In formal verification, we go beyond: build models and prove that the systems work.

(But, I have a **really fast** fuzz tester!)

Suppose we want to test that addition of two Long integer values is commutative by trying all possible values, and we can run 10 tests every nanosecond (10GHz rate).

How much time would it take to cover all cases?

Comparison to Testing

We test computer systems and we should.

In formal verification, we go beyond: build models and prove that the systems work.

(But, I have a **really fast** fuzz tester!)

Suppose we want to test that addition of two Long integer values is commutative by trying all possible values, and we can run 10 tests every nanosecond (10GHz rate).

How much time would it take to cover all cases?

Number of tests: $2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$

Seconds: 10^{28}

Days: $1.15 \cdot 10^{23}$

Years: $3.15 \cdot 10^{20}$

Billions of years: $3.15 \cdot 10^{11}$

Comparison to Testing

We test computer systems and we should.

In formal verification, we go beyond: build models and prove that the systems work.

(But, I have a **really fast** fuzz tester!)

Suppose we want to test that addition of two Long integer values is commutative by trying all possible values, and we can run 10 tests every nanosecond (10GHz rate).

How much time would it take to cover all cases?

Number of tests: $2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$

Seconds: 10^{28}

Days: $1.15 \cdot 10^{23}$

Years: $3.15 \cdot 10^{20}$

Billions of years: $3.15 \cdot 10^{11}$

Wait for the universe to “big bang” ten billion times (and $x + (y + z) == (x + y) + z$?)

Using Software Verifier and Satisfiability Modulo Theory (SMT) Solver

```
def addCom0(x: Long, y: Long): Unit = {  
} ensuring(_  $\Rightarrow$  x + y = y + x)
```

```
> stainless-scalac AddCom.scala
```

```
addCom0 postcondition valid U:smt-z3 (0.2 seconds)
```


Proof Assistant Approach: Prove for n Bits

```
type Digits = List[Boolean]; val zero = Nil[Boolean]()  
def add(x: Digits, y: Digits, carry: Boolean): Digits = {  
  require(x.length == y.length)  
  (x,y) match {  
    case (Nil(), Nil()) => if (carry) true::zero else zero  
    case (x1::xs, y1::ys) => {  
      val z = x1 ^ y1 ^ carry  
      val carry1 = (x1 && y1) || (x1 && carry) || (y1 && carry)  
      z :: add(xs, ys, carry1) } }  
def addCom(x: Digits, y: Digits, carry: Boolean): Unit = {  
  require(x.length == y.length) // proof by induction:  
  (x,y) match {  
    case (x1::xs, y1::ys) => {  
      val carry1 = (x1 && y1) || (x1 && carry) || (y1 && carry)  
      addCom(xs, ys, carry1) }  
    case _ => () }  
} ensuring(_ => add(x, y, carry) == add(y, x, carry)) // property
```

Modeling and Semantics

We will see how to verify recursive functions using SMT solvers

But how do we verify

- ▶ higher-order functions
- ▶ dynamically allocated data structures (mutable trees, graphs)
- ▶ distributed systems

We show how we can represent them as functional programs

- ▶ see already the talk by J Moore “Machines Reasoning about Machines”

Overview of the Course

part	application area	tools	automation
I	finite-state systems	model checkers, SAT solvers	high
II	all systems, math	proof assistants	low but can invoke others
III	software	verifiers, SMT solvers	medium
IV	modeling	intermediate language verifiers	N/A

Impact of Verification: Software Disasters

- ▶ Ariane 5 rocket maiden flight explosion: http://www.inf.ed.ac.uk/teaching/courses/seoc/2008_2009/resources/ariane5.pdf
- ▶ Mars Polar orbiter loss:
https://en.wikipedia.org/wiki/Mars_Polar_Lander "most likely cause of the mishap was a software error that incorrectly identified vibrations"
- ▶ Accidents in various Boeing models (before: 777, in recent year others)
- ▶ Northeast blackout of 2003:
https://en.wikipedia.org/wiki/Northeast_blackout_of_2003
(race condition)
- ▶ Radio therapy machine Therac-25:
<https://en.wikipedia.org/wiki/Therac-25>

Successful Companies and Startups

- ▶ AbsInt products, many originally from academia:
<https://www.absint.com/products.htm>
 - ▶ Verified control software of Airbus 340, 380 using ASTRÉE static analyzer
 - ▶ Formally proven correct C compiler CompCert (originally by Xavier Leroy)
 - ▶ worst-case execution time analysis, ...
- ▶ Formally verified microkernel seL4 and stack built on top by Data61 (formerly Nicta), used Isabelle
- ▶ Coverity static analysis company prevent acquired for USD 380M by Synopsis
- ▶ Jasper Design Automation acquired by Cadence
- ▶ Semmle datalog analysis: just acquired by GitHub
- ▶ Monoidics: acquired by Facebook, running analysis on facebook phone client
- ▶ Microsoft Static Driver Verifier: shipped in 2000-s as part of driver validation

Transition System

They are similar to finite-state machines

Define transition system as (S, I, r, A) :

- ▶ S - the set containing all states of the system.
If S is finite, we have a *finite-state system*
- ▶ $I \subseteq S$ is the set of possible initial states of the system
- ▶ $r \subseteq S \times A \times S$ - transition relation; $(s, a, s') \in r$ means:
with the environment signal a , system can move in one step from state s to s'
 - ▶ we mostly assume that a is the input to the system
 - ▶ in the special case that $r : S \times A \rightarrow S$, we say the system is *deterministic*
- ▶ A - set of signals with which the system communicates with the environment

To establish that a system is well behaved we often introduce a set of error states $E \subseteq S$ that we never want the system to reach, as well as its complement, the set $G \subseteq S$ of good states.

A Trace of the System $M = (S, I, r, A)$

A finite or infinite sequence $s_0, a_0, s_1, a_1, s_2, \dots$ starting from $s_0 \in I$ with steps given by r :

$$\begin{array}{l} \boxed{s_0} \quad s_0 \in I \\ \downarrow a_0 \quad a_0 \in A \\ \boxed{s_1} \quad (s_0, a_0, s_1) \in r \\ \downarrow a_1 \quad a_1 \in A \\ \boxed{s_2} \quad (s_1, a_1, s_2) \in r \\ \dots \end{array}$$

In general, we require $(s_i, a_i, s_{i+1}) \in r$ for all i in the length of the sequence.

If the trace is finite, we assume it ends with a state s_n and call n its length.

$Traces(M)$ is the set of all traces of M

Reachable states $Reach(M)$: states s_n for which there exists a trace that ends in s_n ,

$$Reach(M) = \{s_n \mid \exists n. \exists (s_0, a_0, s_1, a_1, \dots, s_n) \in Traces(M)\}$$

Algorithm: Explicit-State Reachability Checking

- ▶ Input: $M = (S, I, r, A)$ where S is **finite**, $E \subseteq S$ (error states)
- ▶ Output: either a $(s_0, a_0, s_1, a_1, \dots, s_n) \in \text{Traces}(M)$ where $s_n \in E$, or the answer “Safe” if no such trace exists
- ▶ Idea: graph reachability from nodes in I , following edges in $(s, a, s') \in r$ as long as we have not seen s' before
- ▶ To be able to report the trace, build a directed reachability graph of explored edges (never create cycles or duplicate nodes)
- ▶ If no edge in r leads to a previously unexplored node, we stop (this must eventually happen because S is finite)

Explicit-State Reachability Checking Algorithm: Graph Search

Graph reachability using a work list

- ▶ Input: $M = (S, l, r, A)$ where S is **finite**, $E \subseteq S$ (error states)
- ▶ Output: either a $(s_0, a_0, s_1, a_1, \dots, s_n) \in \text{Traces}(M)$ where $s_n \in E$, or For “safe” if no such trace exists

efficiency, differentiate three sets of nodes in a graph:

- ▶ set of all nodes
- ▶ explored nodes: whose all successors we have explored
- ▶ frontier nodes (worklist): we have explored them but not their successors

Key operation: take a frontier node s , add all of its unexplored non-frontier successors to the frontier, move s to explored.

Exercise 1: Bounded Counter

Consider a system with $S = \{0, 1, 2, \dots, 6\}$ that takes signals $A = \{+, -\}$ with initial state 0 and counts up by 2 on $+$ and down by 2 on $-$ but never goes below 0 or above 6 (stays in the state if needed). Write down the transition system definition and prove that the state $E = \{3\}$ is not reachable using explicit-state reachability algorithm. Draw the reachability graph.

Simplified Transition Relation and Reachable States

Let $M = (S, I, r, A)$ be a transition system.

Define $\bar{r} = \{(s, s') \mid \exists a \in A. (s, a, s') \in r\}$

Note: even if r is deterministic, \bar{r} can become non-deterministic

Composition of relations: $r_1 \circ r_2 = \{(x, z) \mid \exists y. (x, y) \in r_1 \wedge (y, z) \in r_2\}$

Iteration (paths of length n): $r_1^0 = \Delta = \{(x, x) \mid x \in A\}$, $r_1^{n+1} = r_1 \circ r_1^n$

Transitive closure of r_1 :

$r_1^* = \bigcup_{n \geq 0} r_1^n$ relates endpoints of all finite paths in graph given by r_1

Image of a set under relation: $r_1[X] = \{y \mid \exists x \in X. (x, y) \in r_1\}$

Theorem

$Reach(M) = (\bar{r})^*[I]$ (end points of all finite paths starting in I)

Reachable States Using post

$$M = (S, I, r, A)$$

If $X \subseteq S$, define $post(X) = \bar{r}[X]$

Define $post^0(X) = X$, $post^{n+1}(X) = post(post^n(X))$

Theorem

$$\bigcup_{n \geq 0} post^n(I) = Reach(M)$$

Proof (by swapping existential quantifiers in definitions of image, composition, and \bigcup):

$$\bigcup_{n \geq 0} post^n(I) = \bigcup_{n \geq 0} \bar{r}[\dots \bar{r}[I] \dots] = \bigcup_{n \geq 0} \bar{r}^n[I] = \left(\bigcup_{n \geq 0} \bar{r}^n \right) [I] = \bar{r}^*[I]$$

Invariant and Inductive Invariant

Invariant P of the system M is any superset of reachable states: $Reach(M) \subseteq P$.

- ▶ P is a property satisfied by all reachable states (though not all states in P need to be reachable).
- ▶ In every trace, by definition $s_i \in Reach(M) \subseteq P$. So the property $s_i \in P$ remains in-variant (does not change) as the system makes a step from i to $i + 1$

Inductive invariant Ind is a set $Ind \subseteq S$ that satisfies the following:

- ▶ $I \subseteq Ind$ (holds initially)
- ▶ if $s \in Ind$ and $(s, a, s') \in r$, then $s' \in Ind$

Exercise: prove that every **inductive invariant** is an **invariant**.

For invariant I , Ind is an **inductive strengthening** of I if Ind is an inductive invariant and $Ind \subseteq I$ (Ind is an inductive hypothesis that proves $Reach(M) \subseteq Ind \subseteq I$)

Invariants in Bounded Counter

Consider again the bounded counter system $M = (S, l, r, A)$ with $S = \{0, 1, 2, \dots, 6\}$ and $A = \{+, -\}$.

Let $G_1 = S \setminus \{3\} = \{0, 1, 2, 4, 5, 6\}$

- ▶ Is G_1 an invariant? Prove or disprove.
- ▶ Is G_1 an inductive invariant? Prove or disprove.

Same question for $G_2 = \{4, 5, 6\}$

Same question for $G_3 = \{0, 2, 4, 6\}$

set	invariant?	inductive invariant?
G_1	yes	no
G_2	no	no
G_3	yes	yes

Checking Inductive Invariants in Stainless

```
def check(inputs: List[Input]) = {  
  var remains: List[Input] = inputs  
  var left = true  
  var state: Int = 0  
  (while (left) {  
    remains match {  
      case Nil() => left = false  
      case input::_ => {  
        input match {  
          case Plus if (state <= 4) => state += 2  
          case Minus if (state >= 2) => state -= 2  
          case _ => ()  
        }  
        remains = remains.tail  
      }  
    }  
  }) invariant (G3.contains(state)) // try with G1, G2, G3  
}
```