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

How?
1. Define a mathematically rigorous notion of a system satisfying a specification

2. 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. But in formal verification, we go beyond: make mathematical models and prove that the systems work.

Comparison to Testing

We test computer systems and we should. But in formal verification, we go beyond: make mathematical models and prove that the systems work. This is hard work.

Comparison to Testing

We test computer systems and we should. But in formal verification, we go beyond: make mathematical models and prove that the systems work. This is hard work.
Why bother? Maybe we can use a **really fast** fuzz tester?
Suppose we want to **test** that addition of two **Long** integer values is commutative by trying all possible values.

assert(x + y == y + x)

(The values x and y are arbitrary, they may come, e.g., from input.)
Suppose we can run 10 tests every **nanosecond**. How long to test **all** cases?

Comparison to Testing

We test computer systems and we should. But in formal verification, we go beyond: make mathematical models and prove that the systems work. This is hard work.
Why bother? Maybe we can use a **really fast** fuzz tester?
Suppose we want to **test** that addition of two **Long** integer values is commutative by trying all possible values.

assert$(x + y == y + x)$

(The values x and y are arbitrary, they may come, e.g., from input.)
Suppose we can run 10 tests every **nanosecond**. How long to test **all** cases?

Number of tests:	$2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$
Seconds:	10^{28}
Days:	$1.15 \cdot 10^{23}$
Years:	$3.15 \cdot 10^{20}$

Comparison to Testing

We test computer systems and we should. But in formal verification, we go beyond: make mathematical models and prove that the systems work. This is hard work.
Why bother? Maybe we can use a **really fast** fuzz tester?
Suppose we want to **test** that addition of two **Long** integer values is commutative by trying all possible values.

assert$(x + y == y + x)$

(The values x and y are arbitrary, they may come, e.g., from input.)
Suppose we can run 10 tests every **nanosecond**. How long to test **all** cases?

Number of tests:	$2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$
Seconds:	10^{28}
Days:	$1.15 \cdot 10^{23}$
Years:	$3.15 \cdot 10^{20}$

Ten billion times since “big bang”.

Comparison to Testing

We test computer systems and we should. But in formal verification, we go beyond: make mathematical models and prove that the systems work. This is hard work.
Why bother? Maybe we can use a **really fast** fuzz tester?
Suppose we want to **test** that addition of two **Long** integer values is commutative by trying all possible values.

assert$(x + y == y + x)$

(The values x and y are arbitrary, they may come, e.g., from input.)
Suppose we can run 10 tests every **nanosecond**. How long to test **all** cases?

Number of tests:	$2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$
Seconds:	10^{28}
Days:	$1.15 \cdot 10^{23}$
Years:	$3.15 \cdot 10^{20}$

Ten billion times since "big bang". Don't even ask about $x + (y + z) == (x + y) + z$

Feasible Alternative: Automated Theorem Proving

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- we can also prove theorems about unbounded integers (need infinitely many tests)

A verifier also uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x+y=y+x$, and derive new ones, like $x+(a+1)=(a+1)+x$.

Feasible Alternative: Automated Theorem Proving

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- we can also prove theorems about unbounded integers (need infinitely many tests)

A verifier also uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x+y=y+x$, and derive new ones, like $x+(a+1)=(a+1)+x$. Verifiers also make us of **automated theorem proving procedures**, which can discover an infinite number of facts using known theorems and rules.

Feasible Alternative: Automated Theorem Proving

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- we can also prove theorems about unbounded integers (need infinitely many tests)

A verifier also uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x+y=y+x$, and derive new ones, like $x+(a+1)=(a+1)+x$.
Verifiers also make us of **automated theorem proving procedures**, which can discover an infinite number of facts using known theorems and rules.
Different kinds of proving procedures, depending on success guarantees:

- decision procedures (linear arithmetic, proof checkers, SAT, bitvectors): algorithms upper bounds on time by which they terminate with yes/no answer

Feasible Alternative: Automated Theorem Proving

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- we can also prove theorems about unbounded integers (need infinitely many tests)

A verifier also uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x+y=y+x$, and derive new ones, like $x+(a+1)=(a+1)+x$.
Verifiers also make us of **automated theorem proving procedures**, which can discover an infinite number of facts using known theorems and rules.
Different kinds of proving procedures, depending on success guarantees:

- decision procedures (linear arithmetic, proof checkers, SAT, bitvectors): algorithms upper bounds on time by which they terminate with yes/no answer
- semi-decision procedures (first-order logic provers): no upper bound, but for every theorem there exists a time by which they discover it is true (complete)

Feasible Alternative: Automated Theorem Proving

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- we can also prove theorems about unbounded integers (need infinitely many tests)

A verifier also uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x+y=y+x$, and derive new ones, like $x+(a+1)=(a+1)+x$.
Verifiers also make us of **automated theorem proving procedures**, which can discover an infinite number of facts using known theorems and rules.
Different kinds of proving procedures, depending on success guarantees:

- decision procedures (linear arithmetic, proof checkers, SAT, bitvectors): algorithms upper bounds on time by which they terminate with yes/no answer
- semi-decision procedures (first-order logic provers): no upper bound, but for every theorem there exists a time by which they discover it is true (complete)
- heuristics (simplifiers, induction heuristics, type inference): even if formula is valid, they may run forever, or say "do not know"

Compiling to Formulas

How do we go from program correctness statements to theorems?

Compiling to Formulas

How do we go from program correctness statements to theorems?

Compile programs and properties to formulas!
We call these formulas **verification conditions**.

If verification condition is valid formula, then program satisfies specification.

Analogy:

programming language compiler	verification-condition generator
program $\longrightarrow$ machine code	(program + specification) $\longrightarrow$ formula

Compiling to Formulas

How do we go from program correctness statements to theorems?

Compile programs and properties to formulas!
We call these formulas **verification conditions**.

If verification condition is valid formula, then program satisfies specification.

Analogy:

programming language compiler	verification-condition generator
program $\longrightarrow$ machine code	(program + specification) $\longrightarrow$ formula

$$\boxed{\text{program verifier}} = \boxed{\text{verification condition generator}} \oplus \boxed{\text{theorem prover}}$$