Formal Verification

How to check if a produced program meets the formal specification?

Testing/Typing are not sufficient

- Easy to argue that a given input will produce a given output (though the halting problem is already undecidable).
- Easy to argue that a property always holds at a single program point
- Also easy to argue that all constructs in the language will preserve some property (like when we proved type soundness).
- Much harder to prove general properties of the behavior of a program on all inputs.

Undecidability of Program Verification

Rice's Theorem (1951): Every *nontrivial* semantic property of recursively enumerable languages is *undecidable*.

- Recursively enumerable languages are equivalent to Turing machines (and almost all languages you program).

Proof: Reduce from the halting problem of Turing machines.

Program Verification

1949

A. M. Turing

Friday, 24th June [1949]

Checking a large routine by Dr A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and

1967

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Robert W. Floyd

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established

1979

DeMillo, Lipton and Perlis

Reports and Articles

Social Processes and Proofs of Theorems and Programs

Richard A. De Millo
Georgia Institute of Technology

Richard J. Lipton and Alan J. Perlis
Yale University

2009

Hoare, Misra, Leavens, Shankar

The Verified Software Initiative: A Manifesto

C.A.R. HOARE
Microsoft Research

JAYADEV MISRA
The University of Texas at Austin

GARY T. LEAVENS
Iowa State University

and

NATARAJAN SHANKAR
SRI International Computer Science Laboratory

1. INTRODUCTION

We propose an ambitious and long-term research program toward the construction of error-free software systems. Our manifesto represents a consensus position that has emerged from a series of national and international meetings, workshops, and conferences held from 2004 to 2007. The research project, the Verified Software Initiative,

Success Stories

Infrastructure: Z3 CVC4 Yices2 Coq Isabelle LEAN

Verifiers: VCC Dafny Astrée Boogie Move Bedrock ESC/Java2 CPA

Success Stories: Hyper-V seL4 IronFleet AIRBUS Verve OS ExpressOS diem

Axiomatic Semantics (AKA program logics)

A system for proving properties about programs

Key idea:

- We can define the semantics of a construct by describing its effect on **assertions** about the program state.

Two components

- A language for stating assertions (“the assertion logic”)
- Can be First-Order Logic (FOL), a specialized logic such as separation logic, or Higher-Order Logic (HOL), which can encode the others.
- Many specialized languages developed over the years:
  - Z, Larch, JML, Spec#
- Deductive rules (“the program logic”) for establishing the truth of such assertions

The Basics

```
{A} stmt {B}
```

Precondition   Postcondition

Hoare triple

- If the program state *before* execution satisfies A, and the execution of stmt *terminates*, the program state *after* execution satisfies B
- This is a partial correctness assertion.

- We sometimes use the notation

```
[A] stmt [B]
```

to denote a total correctness assertion

which means you also have to prove termination.

What do assertions mean?

The language of assertions:

- $A \coloneqq \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 \leq e_2 \mid A_1 \wedge A_2 \mid \neg A \mid \forall x. A$
- $e \coloneqq 0 \mid 1 \mid \cdots \mid x \mid y \mid \cdots \mid e_1 + e_2 \mid e_1 \cdot e_2$

Notation $\sigma \vDash A$ means that the assertion holds on state σ .

- A is interpreted inductively over state σ as a FO structure.
- Ex. $\sigma \vDash A \wedge B$ iff. $\sigma \vDash A$ and $\sigma \vDash B$

Derivation Rules

Derivation rules for each language construct

$$\frac{}{\vdash \{A[x \to e]\} x := e \, \{A\}}$$

$$\frac{\vdash \{A \wedge b\} c_1 \, \{B\} \qquad \vdash \{A \wedge not \ b\} c_2 \, \{B\}}{\vdash \{A\} if \ b \ then \ c_1 \, else \ c_2 \, \{B\}}$$

$$\frac{\vdash \{A \wedge b\} c \, \{A\}}{\vdash \{A\} while \ b \ do \ c \, \{A \wedge not \ b\}}$$

$$\frac{\vdash \{A\} c_1 \, \{C\} \qquad \vdash \{C\} c_2 \, \{B\}}{\vdash \{A\} c_1 ; c_2 \, \{B\}}$$

Can be combined with the rule of consequence

$$\frac{\vdash A' \Rightarrow A \vdash \{A\} c \, \{B\} \vdash B \Rightarrow B'}{\vdash \{A'\} c \, \{B'\}}$$

Soundness and Completeness

What does it mean for our derivation rules to be sound?

What does it mean for them to be complete?

So, are they complete?

{true} x:=x {p}

{true} c {false}

Relative Completeness in the sense of Cook (1974)

Expressible enough to express intermediate assertions, e.g., loop invariants

Example

The following program purports to compute the square of a given integer n (not necessarily positive).

```
int i, j;
i := 1;
j := 1;
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
return i;
```

Example

```
{true}
int i, j;
i := 1;
j := 1;
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{??}
i := 1;
{??}
j := 1;
{??}
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{??}
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{??}  //loop invariant
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{i = n*n} //weakest precondition
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{i = j*j}  //loop invariant
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
{i = n*n} //weakest postcondition
return i;
{i = n*n}
```

Example

```
{true}
int i, j;
{true}  //strongest postcondition
i := 1;
{i=1}  //strongest postcondition
j := 1;
{i=1 ∧ j=1}  //strongest postcondition
{i = j*j}  //loop invariant
while (j != n) {
    {i = j*j ∧ j != n}
    i := i + 2*j + 1;
    {i = (j+1)*(j+1) ∧ j != n}
    j := j+1;
    {i = j*j ∧ j-1 != n}
}
{i = n*n} //weakest postcondition
return i;
{i = n*n}
```

Total Correctness

```
[A] stmt [B]
```

Hoare triple

- If A holds before stmt, stmt terminates and B will hold afterward.

Total Correctness

Definition: a well-ordered set is a set S with a total order $>$ such that every non-empty subset of S has a least element.

E.g., $(\mathbb{N}, >)$ is a w.o. set, $(\mathbb{Z}, >)$ is not

$(\mathbb{N}^2, >)$ where $(a, b) > (a', b')$ if $a > a'$, or $a = a'$ and $b > b'$

Total Correctness

Termination:

1. find a ranking function $rank: ProgStates \rightarrow (\mathbb{N}, >)$

2. find a set of cutpoints (program points) to cut the program

3. prove for any cutpoint pc, and any two program states S_1, S_2, if (S_1, pc) reaches (S_2, pc) in an execution sequence, then $rank(S_1) > rank(S_2)$

Example: while (x>5) x:=x-1;

Total Correctness

Example:

```
int i, j;
i := 1;
j := 1;
while (j != n) {
    i := i + 2*j + 1;
    j := j+1;
}
return i;
```

Try Dafny!

Verification and synthesis put together

Oracle-Guided Synthesis (OGIS)

Formal Specification

Synthesizer

candidate

Verifier (not quite an oracle)

inductive invariants
ranking functions
etc.

counterexample

Provably-Correct Program

Impossible Trilemma

Expressive properties

General-purpose programs

Automaticity

From the perspective of synthesis:

- A synthesizer usually needs to verify many candidate programs
- The verifier should serve as an oracle
- automation and efficiency are most important
- The goal is to synthesize program that can *be automatically verified*
- Automated reasoning is possible in some domains!

Logical Reasoning for Verification

```
x=1;
y=1;
while (*) {
   x=x+2;
   y=y+1;
}
```

$$\forall x, y: (x = 1 \wedge y = 1) \rightarrow x + y \geq 2$$
$$\forall x, y, x'y'(x + y \geq 2 \wedge x' = x + 2 \wedge y' = y + 2) \rightarrow x' + y' \geq 2$$

Q: is x+y>=2 always true?

Q: Are these formulae valid in arithmetic?

Satisfiability Modulo Theories

First-Order Theories

Q: Which statements are true in arithmetic/set-theory/groups/fields?

A **theory** is a set of FOL sentences in a FO language

- Fix a language for arithmetic: $(\leq, +, \cdot, 0, 1)$ (why no $-, <$?)

How to define a theory?

- Fix a standard model: $\mathbb{N}$ (or $\mathbb{Z}$?)
- Peano Arithmetic: $PA = (\mathbb{N}, \leq, +, \cdot, 0, 1)$
- Theory of PA: $Th(PA) = \{\varphi \mid \varphi \text{ is a sentence in } (\leq, +, \cdot, 0, 1) \text{ and } \mathbb{N} \vDash \varphi\}$

Another way to define a theory

- Fix a set of axioms Σ, then $Th(\Sigma) = \{\varphi \mid \Sigma \vdash \varphi\}$

Common Theories

- Presburger Arithmetic: $PrA = (\mathbb{N}, +, 0,1)$
- Integers: $Int = (\mathbb{Z}, +, -, <, \dots, -1,0,1, \dots)$
- Reals: $Real = (\mathbb{R}, +, -, \cdot, 0,1)$
- Rationals: $RA = (\mathbb{Q}, +, -, \cdot, 0,1)$
- Arrays: $Arr = \left(\text{AllArrays}, read(\cdot,\cdot), write(\cdot,\cdot,\cdot)\right)$
- Strings (many variants): $Str = (\text{AllStrings}, +, len, in_{re}, replaceAll, \dots)$

What Theories are Decidable?

Decidable theories

- $PrA = (\mathbb{N}, +, 0,1)$: double exponential
- $Int = (\mathbb{Z}, +, -, <, \dots, -1,0,1, \dots)$: triple exponential
- $Real = (\mathbb{R}, +, -, \cdot, 0,1)$: double exponential
- $RA = (\mathbb{Q}, +, -, \cdot, 0,1)$: double exponential (P if quantifier-free)
- Quantifier-free $Arr = (\text{AllArrays}, \ read(\cdot,\cdot), write(\cdot,\cdot,\cdot))$: NP-complete
- Quantifier-free Equality (plain FOL): NP-complete
- Quantifier-free String Equations: PSPACE-complete

Undecidable theories

- $PA = (\mathbb{N}, \leq, +, \cdot, 0,1)$ (Gödel’s Incompleteness Theorem, 1931)
- $(\mathbb{Z}, +, \cdot, 1, -1,0)$ (Tarski-Mostowski, 1949)
- $Arr = (\text{AllArrays}, \ read(\cdot,\cdot), write(\cdot,\cdot,\cdot))$
- Theory of Rings RI (Mal'cev, 1961)
- Set Theory ZF (Tarski, 1949)
- Theory of String Equations (Quine, 1946)

Deciding Rational Arithmetic

Definition: A set of formulae Σ admits quantifier elimination if for any formula $\exists\bar{x}\varphi(\bar{x},\bar{y}) \in \Sigma$, there is a quantifier free $\varphi'(\bar{y}) \in \Sigma$ such that $\exists\bar{x}\varphi(\bar{x},\bar{y}) \equiv \varphi'(\bar{y})$.

Theorem: RA admits quantifier elimination.

Rational Arithmetic QE

Step 1: Normalization

- Convert φ to Negation Normal Form (NNF)

Step 2: Remove Negation

- $\neg(s > t) \Rightarrow t > s \vee t = s$
- $\neg(s = t) \Rightarrow s > t \vee t > s$

Step 3: Solve for x in $\exists x \varphi$

- $3x > 7y \Rightarrow x > \frac{7}{3}y$
- Collect all terms t_i compared to x, e.g., $x > t_1, t_2 > x, x = t_3, \ldots$
- Instantiate x in $\exists x \varphi$ with all possible $\frac{t_i + t_j}{2}$, ∞ and $-\infty$

Example

- $\exists x(2x = y)$
- $\exists x(3x + 1 = 10 \land 7x - 6 > 7)$

Solving QF Rational Arithmetic

Solve satisfiability of $\exists \bar{x} \varphi(\bar{x})$

- Each conjunction is $\bigwedge_j a_{1,j} x_1 + \cdots + a_{k,j} x_k > c_j$
- Just linear programming!
- LP is solvable in (weakly) polynomial time

Theorem: $\mathrm{Th}(RA)$ is decidable in double exponential time.

Automated reasoning focuses on QF theories

- Many theories are only QF-decidable
- Quantified theories are usually too expensive, even if they are decidable
- QF theories are *compositional* (under some conditions)

How to combine decidable theories?

How to combine decidable theories?

$L_1 = (R_1, F_1, C_1)$
Th_1 is a decidable theory over L_1
D_1 is a decision procedure for Th_1

$L_2 = (R_2, F_2, C_2)$
Th_2 is a decidable theory over L_2
D_2 is a decision procedure for Th_2

$L_1 \cup L_2 = (R_1 \cup R_2, F_1 \cup F_2, C_1 \cup C_2)$
$Th_1 \cup Th_2 = \{\varphi \mid Th_1 \cup Th_2 \vdash \varphi\}$
Can we build a decision procedure for $Th_1 \cup Th_2$ from D_1 and D_2?

Example

PrA is decidable

Arr is QF-decidable

Is $a[x + x] > 2 \land a[4] = 1 \land x > 1 \land x < 3$ satisfiable in $PrA \cup Arr$?

The combined theory is undecidable in general!

Nelson-Oppen Combination

Theorem (1979): If

- Th_1 is a QF-decidable theory over L_1
- Th_2 is a QF-decidable theory over L_2
- $L_1 \cap L_2 = \emptyset$
- Both Th_1 and Th_2 are stably infinite (intuitively, both theories have infinite models)

then $Th_1 \cup Th_2$ is QF-decidable!

Combinable theories: $\{PrA, Int, Real, RA\}$ + Equality + Arr

Nelson-Oppen Combination

Step 1: Purification

- Split an $L_1 \cup L_2$-formula φ into an L_1-formula φ_1 and an L_2-formula φ_2 such that φ and $\varphi_1 \wedge \varphi_2$ are equisatisfiable

- Example: $f\big(x + g(y)\big) < g(a) + f(b)$

$$\Downarrow$$

$$\begin{array}{c} t_1 = g(y) \\ t_3 = f(t_2) \\ t_4 = g(a) \\ t_5 = f(b) \end{array} \quad \wedge \quad \begin{array}{c} t_2 = x + t_1 \\ t_5 < t_4 + t_5 \end{array}$$

Nelson-Oppen Combination

Step 2: Guess and Check

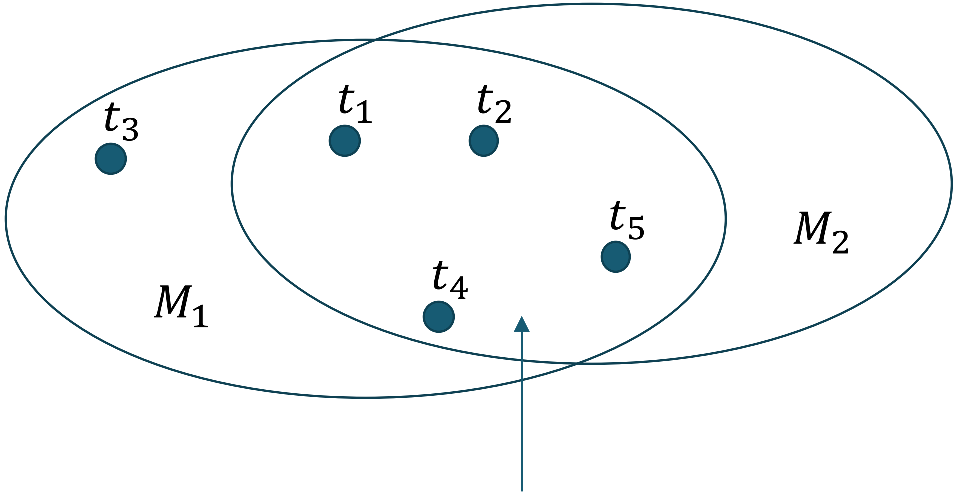
$$t_1 = g(y)$$
$$t_3 = f(t_2)$$
$$t_4 = g(a)$$
$$t_5 = f(b)$$

$$\wedge$$

$$t_2 = x + t_1$$
$$t_5 < t_4 + t_5$$

Guess an equality:

$$t_1 = t_2$$
$$t_1 = t_4$$
$$t_2 = t_4$$
$$t_1 \neq t_5$$
$$t_2 \neq t_5$$
$$t_4 \neq t_5$$



M_1 and M_2 should agree on the equality between shared variables!

Solve the two theories separately!

(if both theories are in NP, so is the combined procedure)

Satisfiability Modulo Theories

Nelson-Oppen Method + DPLL Procedure (solving propositional constraints using backtracking)

Standard Interchange Format

Supports arithmetic, bit-vectors, uninterpreted functions, arrays, data types, …

A plethora of well-engineered solvers (Z3, CVC4, etc.)

Try Z3-play

Example

Is $a[x + x] > 2 \wedge a[4] = 1 \wedge x > 1 \wedge x < 3$ satisfiable in $PrA \cup Arr$?

```
(declare-fun x () Int)
(declare-const a (Array Int Int))
(assert (> (select a (+ x x)) 2))
(assert (= (select a 4) 1))
(assert (> x 1))
(assert (< x 3))
(check-sat)
(get-model)
(exit)
```

```
unsat
(error "line 8 column 10: model is not available")
```

Example

Is $a[x+x] > 2 \wedge a[4] = 1 \wedge x > 1 \wedge x < 3$ satisfiable in $Real \cup Arr$?

```
(declare-fun x () Real)
(declare-const a (Array Real Real))
(assert (> (select a (+ x x)) 2))
(assert (= (select a 4) 1))
(assert (> x 1))
(assert (< x 3))
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun a () (Array Real Real)
    (store ((as const (Array Real Real)) 1.0) 3.0 (/ 5.0 2.0)))
  (define-fun x () Real
    (/ 3.0 2.0))
)
```