# Mathematics of Program Construction

Paritosh Pandya

Tata Institute of Fundamental Research, Mumbai

15 May, 2017

# Program Correctness

## Engineering Design

- Performance Functionality, Efficiency, Portability, Modifiability.
- Correctness "Product functions reliably as intended."

## Correctness in Software

Improving Reliability of Programs (Software Metrics).

- Testing
- Code walkthrough
- Managing Software Development Process (CMM Maturity Levels)
- n-version programming

Software Crisis Nato Software Engineering Conference in 1960s.

# Correctness in Engineering

## Basis of Correctness

- Mathematical Modelling of Designs
- Analysis and Synthesis Techniques
- Rigorous Mathematical Specification of Products

# Correctness in Engineering

## Basis of Correctness

- Mathematical Modelling of Designs
- Analysis and Synthesis Techniques
- Rigorous Mathematical Specification of Products

## Software: Engineering or Craft?

- No rigorous specification of the intended behaviour of the system.
- No mathematical analysis of designs for correctness. Validation is by testing.
- Correctness is implicit and uncertain. Products are unreliable.

# State of Affairs

## Disasters:

- Intel Pentium SRT Division Bug (1995)
  (Cost 1 Billion $).
- Arian 5 Launch Failure (1996)
  (Cost 850 Million $).
- Mars Polar lander (1999) Incomplete requirements.
- Indian PSLV Failed Launch

## Changing Face of Hardware and Software

- Multi core processors,new memory models.
- New architectures: clusters, Grids, Multi-agent systems,
  Service oriented computing
- Concurrent asynchronous modules with mediated interaction

# Formal Methods in Programming

- Mathematical Models for program behaviours.
- Logical notations for specifying properties of programs.
- Methods for checking that program meets its desired specification.

**Verification Problem**

To check whether $M \models \phi$

- $M$ system model in programming/modelling language.
- $\phi$ property in specification notation.

Must check that all behaviours of $M$ satisfy $\phi$.

# Programs and Assertions

**Programs (e expressions, b boolean expr.)**

x:=e
S1;S2
if b then S1 else S2 fi
while b do S od

A State assigns a value to each variable.

A program starts in an initial state. It ends in a final state or does not terminate.

# Assertions

## Assertions

Conditions on state. They specify a subset of states. E.g. $x > y$.
Formally, assertions are formulae of first-order logic.

Assertions use logical connectives.

| | |
|---|---|
| $P \wedge Q$ | $P$ and $Q$ |
| $P \vee Q$ | $P$ or $Q$ |
| $\neg P$ | not $P$ |
| $P \Rightarrow Q$ | whenever $P$ is true so is $Q$ |

## Reasoning

$AXIOMS \models P \Rightarrow Q$

# A Simple Program

## Problem

Compute quotient $q$ and reminder $r$ of integers $x$ divided by $y$.

```
r:=x; q:=0;
while r  >  y do
        r:=r-y;  q:=q+1
od
```

# A Simple Program

## Problem

Compute quotient $q$ and reminder $r$ of integers $x$ divided by $y$.

```
r:=x; q:=0;
while r  >  y do
              r:=r-y;  q:=q+1
od
```

| x | y | q | r |
|---|---|---|---|
| 8 | 3 | 2 | 2 |
| 8 | 0 | | |
| -8 | 3 | 0 | -8 |
| 6 | 3 | 1 | 3 |

# A Simple Program

## Problem

Compute quotient $q$ and reminder $r$ of integers $x$ divided by $y$.

$\{0 < y \quad \wedge \quad 0 \leq x\}$ Precondition

```
r:=x; q:=0;
while r > y do
          r:=r-y;  q:=q+1
od
```

| x | y | q | r |
|---|---|---|---|
| 8 | 3 | 2 | 2 |
| 8 | 0 | | |
| -8 | 3 | 0 | -8 |
| 6 | 3 | 1 | 3 |

# A Simple Program

## Problem

Compute quotient $q$ and reminder $r$ of integers $x$ divided by $y$.

$\{0 < y \quad \wedge \quad 0 \leq x\}$ Precondition

```
r:=x; q:=0;
while r > y do
        r:=r-y;  q:=q+1
od
```

$\{x = y * q + r \quad \wedge \quad 0 \leq r < y\}$ Postcondition

| x | y | q | r |
|---|---|---|---|
| 8 | 3 | 2 | 2 |
| 8 | 0 | | |
| -8 | 3 | 0 | -8 |
| 6 | 3 | 1 | 3 |

# Program specification

$\{ P \} \quad S \quad \{ Q \}$

- $S$ Program (fragment)
- $P$ Precondition

  Assumed to be true when $S$ starts.
- $Q$ Postcondition

  Required to be true when $S$ terminates.

Advantages

- Clear and Unambiguous articulation of what program must do.
- Separation of concern: User versus developer.

  interface specification.
- Can be formally verified.

# Annotated Program

$\{0 < y \ \wedge\ 0 \leq x\}$ (1)

r:=x; q:=0; (2)

$\{0 < y \ \wedge\ 0 \leq x \ \wedge\ r = x \ \wedge\ q = 0\}$ (3)

$\{inv : 0 \leq r \ \wedge\ 0 < y \ \wedge\ x = y * q + r\}$ (4)

while $\uparrow$ r≥y do (6)

    $\{0 \leq r \ \wedge\ 0 < y \ \wedge\ x = y * q + r \ \wedge\ y \leq r\}$ (7)

    r:=r-y; q:=q+1 (8)

    $\{0 \leq r \ \wedge\ 0 < y \ \wedge\ x = y * q + r\}$ (9)

od (10)

$\{r < y \ \wedge\ 0 \leq r \ \wedge\ 0 < y \ \wedge\ x = y * q + r\}$ (11)

$\{x = y * q + r \ \wedge\ 0 \leq r < y\}$ (12)

# Hoare Logic

Given predicate $Q$
$Q[e/x]$ denotes $Q$ with $x$ substituted by $e$
E.g. $x < 0[x+1/x]$ gives $x+1 < 0$.

Assignment $\{Q[e/x]\} \quad x := e \quad \{Q\}$

Example: $\{x+1 < 0\}$ x:=x+1 $\{x < 0\}$

Sequential Composition

$$\frac{\{P\}\ S_1\ \{Q_1\}, \quad Q_1 \Rightarrow Q_2, \quad \{Q_2\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

David Gries, The Science of Programming, Springer-Verlag.

# Principle of Compositionality

Assembly of Components: $S_1; S_2$

$$\frac{\{P\}\ S_1\ \{Q_1\}, \quad Q_1 \Rightarrow Q_2, \quad \{Q_2\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Meaning of the assembly is derived from the meaning of the components (Frege, Hoare)

Separation of Concerns For any component $\{P\}\ S\ \{Q\}$

- Component user only relies on its specification.
- component developer need not know how it is used.
- Meaning of the assembly is unaffected by changes within the component structure till their meaning is unchanged.

# Proof Carrying Code

[Necule, Lee, Taylor, Morriset]

- Problem domain: extensible programs incorporating externally supplied modules.
- Code carries specification and annotations
- Easy to verify $\mathit{Code} \models \mathit{Spec}$ using annotations. (It is hard to find the annotations.)
- Compilers can add annotations about type safety and memory safety to low level code.

Security Automata Access Control Policies, Information Flow Properties.

"No Send after a Read".

- First ideas: (Turing), Floyd , Hoare, Dijkstra
  - Assigning Meaning to Programs (Floyd,1967)
  - On an axiomatic basic for program correctness (Hoare,1969)
  - The discipline of programming, (Dijkstra 1975).
- Extended in 70s and 80' to
  Procedure calls, Module and Objects,
  Data and Action Refinement,
  Concurrent and Distributed Programs
  Structured Specification Notations: Z, B, RAISE.
- Extended to Reactive Programs using Temporal logic (Pnueli 1977).

Very powerful Methods.
Manual proofs are prohibitively large.

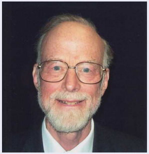
# Founders of Formal Verification

## First Order Logic for Assertions

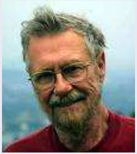
Alan Turing

Bob Floyd

Tony Hoare



Edsgar Dijkstra



## Temporal Logic for specifying reactive systems

Amir Pnueli

- Acceptance has been low in practice.
  Cumbersome formulae.
  Long tedious proofs
  Lack of training.
- Resurgence of Interest.
  Key Factor Tool Support using
  - Theorem Provers
  - Model Checkers

# Automatic and Interactive Theorem Proving

Theorem Provers are programs which find proof of validity (implications) of logic formulae. E.g.
$TH(\mathcal{R}) \vdash x > 15 \wedge x - y < 3 \quad \Rightarrow \quad y > 12$.

Morphology:

- Proof Checker User gives the proof. Machine checks that rules are correctly applied.
- Automatic Theorem Prover Proof is generated by the Machine.
- Interactive Theorem Prover Simple steps are derived by the machine. User makes important steps.
- Computer assisted proofs Proof is given by user. Large but mechanical symbolic or numerical calculations done by machine.

# Interactive Theorem Provers and Demonstrators

Some leading academic Theorem Provers:

- ACL2 (Boyer-Moore, Univ. of Texas)
- PVS (N. Shankar, SRI International, Stanford)
- HOL (Mike Gordon, Cambridge, U.K.)
- COQ (Xavier Leroy, France)

Success stories:

- Pentium SRT Division Algorithm checked using PVS.
- AMD floating point processor verification in ACL2.
  (20 million lines of lemmas and their proofs.)
- Verification of Pipelined processor in PVS
- CompCert: Verification of Optimizing C Compiler in Coq.
- Verification of Micro Kernel in Coq.

# Theorem Proving in Mathematics

- Four color theorem.
- Jordan Curve Theorem
- Godels first and second completeness theorems.
- Robbin's Conjecture Open for 60 years.
  Automatically proved by the EQP theorem prover at Argonne Natiaonal Lab generating human readable proof (published).
- MISAR Mathematical Library. Collection of over 50,000 theorems with checked proofs.

# SAT and SMT Solving

## SAT Solving

Given a propositional formula, such as $(a \wedge \neg b \quad \vee \quad \neg a \wedge c) \wedge b$, find a truth assignment, such as $(a = f, b = t, c = t)$, which makes the formula true.

- SAT is NP-complete in theory.
- Often doable in practice.
  CDCL algorithms solve many million formulae.

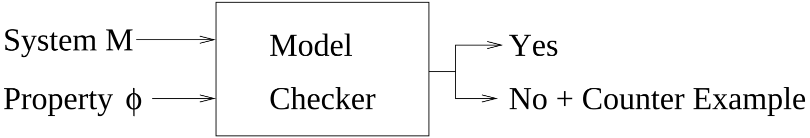
## SMT Solving: Satisfiability Modulo Theories

Theory clauses instead of propositions.
Quantifier free formula from efficiently decidable theory.

- Linear arithmetic: $x_1 + 2 \cdot x_3 \leq y$ over Reals and Integers.
- Bit Vectors and finite precision numbers.
- Uninterpreted functions with equality.

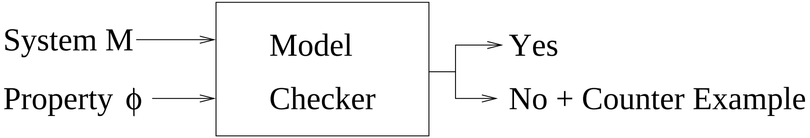
# Model Checking

- Model checker is an algorithm which, given a system model $M$ and property $\phi$, determines whether $M \models \phi$.
- A good model checker will produce a counter example if $M \not\models \phi$.



# Model Checking

- Model checker is an algorithm which, given a system model $M$ and property $\phi$, determines whether $M \models \phi$.
- A good model checker will produce a counter example if $M \not\models \phi$.




## Applicability

Hardware verification
Verification of Embedded systems controllers
Protocol verification (networks, mobile telephony)

# Reactive Systems

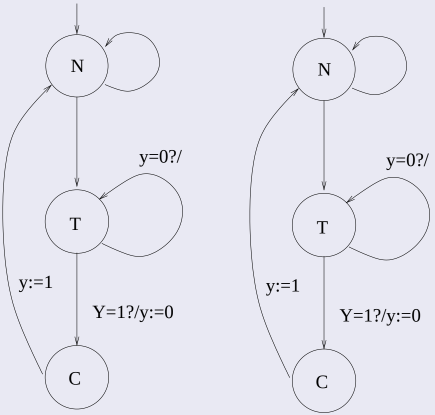
Reactive systems are complex requiring verification.

- Reactive: Output depends on interaction with environment.
- Temporal: Current output depends on past *sequence* of inputs.
- Global: Output from a component may depend on all other components due to interaction.
- Safety Critical Often used in applications requiring high degree of reliability.
- Difficult to test.
  - Failure is not repeatable.
  - Cannot collect enough test data.

Verification question: Do all behaviours of $M$ satisfy $\phi$ ?

## Mutual Exclusion

Initially $y := 0$




- Asynchronous parallelism
- Guarded assignments.

# Properties

- Mutual exclusion: System will not reach a state where both processes are in critical region.
- In *each execution*, whenever either process is in critical region the value of $y = 0$.
- Starvation-freedom: If process 1 is trying to enter the critical region, it will eventually succeed.

# Properties

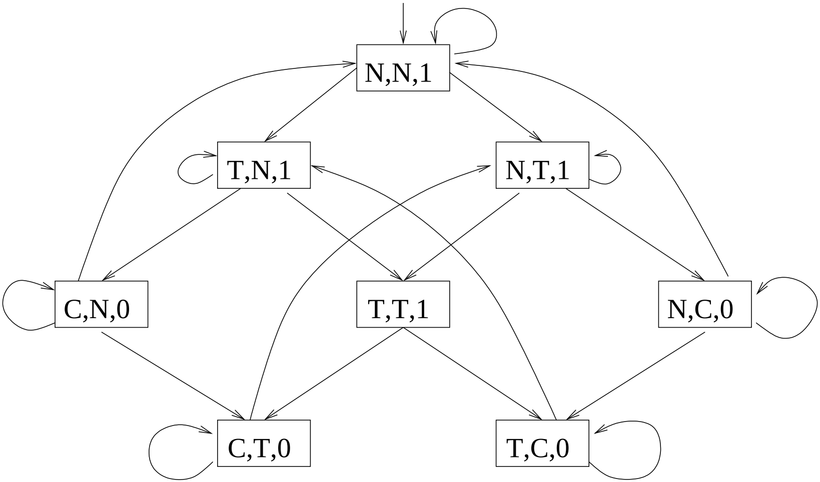
- Mutual exclusion: System will not reach a state where both processes are in critical region.

  $\Box\ \neg(pc1 = C \wedge pc2 = C)$
- In *each execution*, whenever either process is in critical region the value of $y = 0$.

  $\Box\ ((pc1 = C \vee pc2 = C)\ \Rightarrow\ y = 0)$
- Starvation-freedom: If process 1 is trying to enter the critical region, it will eventually succeed.

  $\Box\ (pc1 = T\ \Rightarrow\ (\Diamond\ pc1 = C))$

# Global Automaton for Mutual Exclusion Problem



# Interacting Automata

Models of Reactive Real-time Systems

- Finite State Automata with Hieararchy, Concurrency, Synchronization
- High Level Programming languages: Esterel, SCADE/Lustre.
- Timed and Hybrid Automata for Embedded Systems Modelling
- Petri Nets, Push Down Automata, Multi Counter Machines ...

# SPIN (Bell Labs)

Efficient representation and exploration of state graph.

hashing: one bit per state!!

cycle detection

on-the-fly construction of state graphs

partial-order reductions

symmetry reduction

Can typically explore systems with $10^6$ to $10^9$ states.

Given ACM systems software award for year 2001.

- for developing system software that has lasting influence reflected in concepts, commercial aspects or both.
- Other winners: Unix (1983), System R and Ingress (1988), TCP/IP (1991) and Java (2002).

# Symbolic Model Checking

Can very often explore systems with $10^{120}$ states.

Technique: Represent set of states by boolean formulae.

Iteratively compute the set of reachable states.

$S_i$ set of states reachable from *Init* in $i$ or less steps.

$$S_{i+1} \;=\; S_i \cup F(S_i)$$

Chain $S_0 = Init \subset S_1 \subset \ldots \subset S_m = S_{m+1}$

$S_m$ is the set of reachable states:

$$M \models invariantly(P) \;\text{ iff }\; S_m \subset P.$$

# State of the Art

Initial Idea: [Bryant, MacMillan] BDDS, CTL model checking by iterative computing of fixed points.
SAT solving [Mallik(Princeton)].

Advanced techniques:

- Partitioning and image computation
- Variable ordering (BDD Heuristics)
- Bounded Model Checking
- SAT solving.
- Abstraction and Induction
- Rich specification languages

Leading Symbolic Model Checkers from Universities

Z3 *(Microsoft)*, SMV *(CMU)*, VIS *(UC Berkeley, UC Colorado)*, NuSMV *(IRST,Trento)*, UCLID *(CMU)*, ICS *(SRI,Stanford)*

# Industrial Use

Hardware and Computer Architecture Companies have inhouse model checking tools.
Microsoft, Intel, IBM, Motorola, Siemens, Cadence, Synopsis.

- Intel announced formal verification of floating point unit of Pentium Pro processor (1999). Possibly used formal verification to check parts of design of Pentium IV processor.
- Siemens used Equivalence Checking in validation of ASICs with upto a Million Gates.
- Microsoft provides driver development kit where every third party driver is model checked for health.
- Intel, Motorola, IBM routinely use Model Checking in their design process.

Some model checkers from India

DCVALID *(TIFR)* (Released since 1997),
Word Level ABC verifier *(IITB)* (Not released).
Bebop and SLAM *Microsoft Research India*
DCSYNTH *(TIFR)*, Releasing next week!

# Road Ahead

- Software Model Checking with numerical data.
- Handling systems with complex non-linear dynamics.
- Probabilistic and statistical physics techniques.

We need to interact with other disciplines working on complex systems!!

Thank You.