Checking Correctness – Approaches

- Physical prototypes
  - e.g. testing

- Models (Virtual SW Prototypes)
  - Ad hoc models
    - e.g. SystemC simulation
  - Formal models
    - (well-defined notion of state and transition)

- Verification
  - Algorithmic Verification
  - Axiomatic Verification
Three essential ingredients

- **Models** describe a transition relation on the system states

- **Requirements** describe the expected behavior
  - either as a set of formulas f of a logic (model m satisfies f)
  - or as a set abstract models m’ (model m is observationally equivalent to m’)

- **Methods** for checking that the models satisfy the requirements
Correctness-by-checking – Verification

Model

Requirements

Should be:

- **faithful** e.g. whatever property is satisfied for the model holds for the real system
- generated **automatically** from system descriptions

Verification Method

Should be:

- **consistent** e.g. there exists some model satisfying them
- **complete** e.g. they tightly characterize the system’s behavior

YES, NO, DON’T KNOW

- As a rule, for infinite state models all non trivial properties are undecidable e.g. $x<100$
- Intrinsically high complexity for finite state models (state explosion problem)
### Algorithmic Verification

- **Focus on automation**
- Verification tools use semi-algorithms to check that a **global model** \( m \) (hardware, software, system) meets given requirements
- Theoretical limitations are overcome by using adequate **abstractions**
- Applied mainly to hardware, and abstract software and system models

### Axiomatic Verification

- Semi-automatic compositional verification process based on the use of rules formalizing the semantics of the language
- Relies on the use of Theorem Provers driven by skilled engineers/researchers – efficiency largely depends on their ingenuity
- Applied mainly to hardware or software components e.g. operating systems, compilers
- Invariants
- Axiomatic Verification
- Model-checking
- Behavioral Verification
- Using Abstraction
- Discussion
Models can be considered as transition systems. These are the basic model for studying discrete dynamic systems and their properties.

A transition system

- $Q$: set of states
- $\rightarrow \subseteq Q \times Q$: transition relation

Notation: $q \rightarrow q'$ for $(q, q') \in \rightarrow$

- $q_0 \ q_1 \ldots \ q_i \ldots$ such that $q_i \rightarrow q_{i+1}$ is an execution sequence
- A run is a maximal execution sequence
For a transition system TS=(Q,→), a state predicate P is a function P:Q →{true,false} (subset of Q)

- \( \text{pre}(P)(q) = \exists q' \in Q. \ q \rightarrow q' \) and \( P(q') \) – may-predecessors of P

- \( \text{pre}(P)(q) = \forall q' \in Q. \ q \rightarrow q' \Rightarrow P(q') \) – must-predecessors of P
Invariants – Predicate Transformers

For a transition system $TS=(Q,\rightarrow)$, a state predicate $P$ is a function $P:Q \rightarrow \{\text{true},\text{false}\}$ (subset of $Q$)

- $\text{post}(P)(q) = \exists q' \in Q. \ q' \rightarrow q \text{ and } P(q')$ – may-successors of $P$

- $\widehat{\text{post}}(P)(q) = \neg \text{post}(\neg P)(q) = \forall q' \in Q. \ q' \rightarrow q \Rightarrow P(q')$ – must-successors of $P$
Invariants – Definition

For a transition system $TS=(Q,\to)$, an invariant is a state predicate $P$ that is closed for the transition relation that is

$P(q) \Rightarrow (\forall q'. q \rightarrow q' \Rightarrow P(q'))$

Why invariants are useful? To establish that a property $R$ always holds it is sufficient to find an invariant $P$ such that

$P \Rightarrow R$ and $\text{init} \Rightarrow P$
Fixpoint Characterization:

P is an invariant iff

\[ P \Rightarrow \text{pre}(P) \]

iff

\[ \text{post}(P) \Rightarrow P. \]

That is invariants are solutions of the equations

\[ P = P \land \text{pre}(P) \quad \text{or} \quad P = P \lor \text{post}(P) \]

Properties:

- \[ \text{pre}(P_1 \lor P_2) = \text{pre}(P_1) \lor \text{pre}(P_2) \]
- \[ \text{pre}(P_1 \land P_2) = \text{pre}(P_1) \land \text{pre}(P_2) \]
- \[ \text{post}(P_1 \lor P_2) = \text{post}(P_1) \lor \text{post}(P_2) \]
- If \( P_1, P_2 \) are invariants then \( P_1 \lor P_2 \) and \( P_1 \land P_2 \) are invariants

Computing invariants amounts to solving fixpoint equations \( P = F(P) \)
where \( F \) is a monotonic predicate transformer (\( P_1 \Rightarrow P_2 \) implies \( F(P_1) \Rightarrow F(P_2) \))
Invariants – Solving Fixpoint Equations $P = F(P)$

- **Tarski’s Theorem:** There exists a least and a greatest fixpoint for $F$

- If $F(P_0) \Rightarrow P_0$ then $\text{gfp}(F, P_0) \Rightarrow \bigwedge_{i=0}^{\infty} F^i(P_0)$ and $\text{gfp}(F, P_0) = \bigwedge_{i=0}^{\infty} F^i(P_0)$ for finite state transition systems

- If $P_0 \Rightarrow F(P_0)$ then $\bigvee_{i=0}^{\infty} F^i (P_0) \Rightarrow \text{lfp}(F, P_0)$ and $\text{lfp}(F, P_0) = \bigvee_{i=0}^{\infty} F^i (P_0)$ for finite state transition systems
Invariants – Computing Invariants for Finite-state Systems

P = P \land \pre(P) \quad \text{can be used to compute the greatest invariant implying } P_0

- \quad P_0 \land \pre(P_0) \Rightarrow P_0
- \quad \text{The greatest invariant implying } P_0 \text{ is}
  \quad P_0 \land \pre(P_0) \land \pre^2(P_0) \land \pre^3(P_0) \land \pre^4(P_0) \land \ldots

P = P \lor \post(P) \quad \text{can be used to compute the least invariant implied by } P_0

- \quad P_0 \Rightarrow P_0 \lor \post(P_0)
- \quad \text{The least invariant implied by } P_0 \text{ is}
  \quad P_0 \lor \post(P_0) \lor \post^2(P_0) \lor \post^3(P_0) \lor \post^4(P_0) \lor \ldots
do c_1(X) → X:=f_1(X)|| c_2(X) → X:=f_2(X)|| ..... || c_n(X) → X:=f_n(X) od

Examples:

- do x>y → x:=x-y|| y>x → y:=y-x od
  
  GCD(x,y)=GCD(x_0,y_0) is an invariant

- do x>y → y:=y+1|| y>z → z:=z+1|| z>x → x:=x+1 od
  
  max(x,y,z)=max(x_0,y_0,z_0) is an invariant

- Readers/Writers
  
  do w=0 → r:=r+1|| true → r:=r-1|| w=0∧(r=0) → w:=w+1|| true → w:=w -1od
  
  Mutex = (w=0)∨(r=0) ∧(w≤1) is an invariant

Folk Theorem:

Any system with finite state control can be modeled as a loop guarded command e.g. iterative programs, Petri nets, automata
do \ c_1(X) \rightarrow X:=f_1(X) \parallel c_2(X) \rightarrow X:=f_2(X) \parallel \ldots \parallel c_n(X) \rightarrow X:=f_n(X) \ od

\ pre(P)(X) = c_1(X) \land P[f_1(X)/X] \lor c_2(X) \land P[f_2(X)/X] \lor \ldots \lor c_n(X) \land P[f_n(X)/X]

= \lor_{i=1}^{n} c_i(X) \land P[f_i(X)/X]

\sim \ pre(P)(X) = \land_{i=1}^{n} (\neg c_i(X) \lor P[f_i(X)/X] )
do $c_1(X) \rightarrow X:=f_1(X) || c_2(X) \rightarrow X:=f_2(X) || \ldots || c_n(X) \rightarrow X:=f_n(X)$ od

post(P)(X) =
$\exists X'. P(x') \land [c_1(X') \land (X=f_1(X')) \lor c_2(X') \land (X=f_2(X')) \lor \ldots \lor c_n(X') \land (X=f_n(X'))]$  
$= \exists X'. P(x') \land \bigvee_{i=1}^{n} c_i(X') \land (X=f_i(X'))$
- Invariants

- Axiomatic Verification

- Model-checking

- Behavioral Verification

- Using Abstraction

- Discussion
The correctness of a transformational program S is characterized by triples

\{P\} S \{R\}

where P, R are respectively a precondition and a postcondition, formulas in a predicate logic

- **Partial correctness**: when P holds before the execution of S and S terminates, then R will hold afterwards.
- **Total correctness**: when P holds before the execution of S, then S terminates and R will hold afterwards.

Note that if S is represented as a relation between initial and final states then

- \(\sim\text{pre}(R)\) is the weakest precondition for partial correctness
- \(\text{pre}(R) \land \sim\text{pre} (R)\) is the weakest precondition for total correctness

Axiomatic verification methods provide a set of rules for inferring \{P\}S\{R\} by reasoning on the structure of program S.
Axiomatic Verification

Syntax

\[ S ::= \text{skip} | x := E | \text{if } C \text{ then } S \text{ else } S | \text{while } C \text{ do } S | S ; S \]

An operational semantics defines a transition relation \(<P,v> \rightarrow <P',v'>\) by a set of rules where \(P,P'\) are programs and \(v, v'\) are valuations of program variables:

- \(<\text{skip},v> \rightarrow <\text{skip},v>\)
- \(<x:=E,v> \rightarrow <\text{skip},v[E/x]>\)

\(C(v) = \text{true}\):

- \(<\text{if } C \text{ then } S \text{ else } S',v> \rightarrow <S,v>\)
- \(<\text{while } C \text{ do } S,v> \rightarrow <S;\text{while } C \text{ do } S,v>\)

\(C(v) = \text{false}\):

- \(<\text{if } C \text{ then } S \text{ else } S',v> \rightarrow <S',v>\)
- \(<\text{while } C \text{ do } S,v> \rightarrow <\text{skip},v>\)

- \(<S,v> \rightarrow <\text{skip},v'>\)
- \(<S;S',v> \rightarrow <S',v'>\)

- \(<S,v> \rightarrow <S'',v'>\)
- \(<S;S',v> \rightarrow <S'';S',v'>\)
Axiomatic Verification – Hoare Logic

Rules for partial correctness

\( \{P\} \text{ skip } \{P\} \quad \{P[E/x]\} \ x:=E \ \{P\} \)

\( \{P\} \ S \ \{R\} \quad \{R\} \ S' \ \{T\} \quad \text{Sequential Composition Rule} \)

\( \{P\} \ S;S' \ \{T\} \)

\( \{P \land C\} \ S \ \{R\} \quad \{P \land \neg C\} \ S' \ \{R\} \quad \text{Conditional Rule} \)

\( \{P\} \ \text{if } C \ \text{then } S \ \text{else } S' \ \{R\} \)

\( \{P \land C\} \ S \ \{P\} \quad \text{While Rule (loop invariant)} \)

\( \{P\} \ \text{while } C \ \text{do } S \ \{P \land \neg C\} \)

\( P' \Rightarrow P \quad \{P\} \ S \ \{R\} \quad R \Rightarrow R' \quad \text{Consequence Rule} \)

Result: Hoare Logic is sound and complete provided the underlying logic is
Instead of proving \{P\} S \{R\}
compute wp[S](R), the weakest precondition of R and check that
\[ P \Rightarrow wp[S](R) \]

- \( wp[\text{skip}](R) = R \)
- \( wp[x:=E](R) = R[E/x] \)
- \( wp[S;S'](R) = wp[S](wp[S'](R)) \)
- \( wp[\text{if } C \text{ then } S \text{ else } S'](R) = C \Rightarrow wp[S](R) \land \neg C \Rightarrow wp[S'](R) \)
- \( wp[\text{while } C \text{ do } S](R) = ???? \)

Put WHILE = while C do S
- We have WHILE = if C then S;WHILE else skip
- Thus, wp[WHILE](R) = (C \Rightarrow (wp[S](wp[WHILE](R)) \land \neg C \Rightarrow R )
- wp[WHILE](R) is the least solution of the fixpoint equation

\[ P = C \land wp[S](P) \lor \neg C \land R \]
- Invariants
- Axiomatic Verification
- Model-checking
- Behavioral Verification
- Using Abstraction
- Discussion
Temporal logics

- are used to specify ongoing behavior - non-terminating programs and systems - rather than input/output relation
- are modal logics where modalities express how the truth of statements can vary in time: possibly (\(\text{possibly}(f)\)), inevitably (\(\text{inevitably}(f)\)), always (\(\text{always}(f)\))

**Linear time** (behavior is the set of traces)

**Branching time** (behavior is a tree)
### Syntax

\[ f ::= \text{true} \mid p \mid f \land f \mid \neg f \mid \text{poss}[f](f) \mid \text{inev}[f](f) \]

### Semantics

Given a transition system \( TS=(Q, \rightarrow) \), the meaning of \( f \) is \( |f| \) defined by a function \( | \cdot | \) such that:

- \( |\text{true}| = Q \)
- \( |p| = \text{P state predicate on } Q \)
- \( |f_1 \land f_2| = |f_1| \cap |f_2| \)
- \( |\neg f| = Q - |f| \)
- \( |\text{poss}[f_1](f_2)| = \{q \in Q | \exists s \in \text{EX}(q) \exists k \in \mathbb{N} \forall k' < k \ s(k') \in |f_1| \text{ and } s(k) \in |f_2|\} \)
- \( |\text{inev}[f_1](f_2)| = \{q \in Q | \forall s \in \text{EX}(q) \exists k \in \mathbb{N} \forall k' < k \ s(k') \in |f_1| \text{ and } s(k) \in |f_2|\} \)

A state \( q \) satisfies \( f \) if \( q \in |f| \)

### Abbreviations:

- \( \text{poss}(f) = \text{poss}[	ext{true}](f) \)
- \( \text{inev}(f) = \text{inev}[	ext{true}](f) \)
- \( \text{alw}[f_1](f_2) = \neg \text{poss}[f_1](\neg f_2) \)
- \( \text{alw}(f) = \text{alw}[\text{true}](f) \)
Model-Checking – Expressing Properties

- \((x=n) \land (y=m) \Rightarrow \text{alw}(\text{halt} \Rightarrow x=y=\text{GCD}(n,m))\)  partial correctness
- \((x=n) \land (y=m) \Rightarrow \text{inev}(\text{halt} \land (x=y=\text{GCD}(n,m)))\)  total correctness

- \(\text{alw}(\neg \text{CS1} \lor \neg \text{CS2})\)  mutual exclusion
- \(\text{alw}(\text{poss}(\text{execute}))\)  deadlock-freedom
- \(\text{alw}(\text{inev}(\text{execute}))\)  no starvation

Alternation of send and receive

\[
\begin{align*}
\text{send} & \Rightarrow \text{inev(receive)} \\
\text{receive} & \Rightarrow \text{inev(send)} \\
\text{alw}(\text{send} & \Rightarrow \text{inev(receive)}) \\
\text{alw}(\text{receive} & \Rightarrow \text{inev(send)}) \\
\text{alw}(\text{send} & \Rightarrow \text{inev}[^{\neg} \text{send}](\text{receive})) \\
\text{alw}(\text{receive} & \Rightarrow \text{inev}[^{\neg} \text{receive}](\text{send}))
\end{align*}
\]
Model-Checking – Fixpoint Characterization

- \( \text{poss}[f_1](f_2) = \text{lfpX. } f_2 \lor f_1 \land \text{pre}(X) \)
- \( \text{inev}[f_1](f_2) = \text{lfpX. } f_2 \lor f_1 \land \text{pre}(X) \land \sim \text{pre}(X) \)

These define a semi-algorithmic verification method that can be either enumerative or symbolic.

- \( \text{poss}[f_1](f_2) = \bigvee_{i=0}^{\infty} X_i \) where
  \( X_0 = f_2 \) and \( X_{i+1} = X_i \lor (f_1 \land \text{pre}(X_i)) \)

- \( \text{inev}[f_1](f_2) = \bigvee_{i=0}^{\infty} X_i \) where
  \( X_0 = f_2 \) and \( X_{i+1} = X_i \lor (f_1 \land \text{pre}(X_i) \land \sim \text{pre}(X_i)) \)
Consider the stability property $\text{inve}(\text{alw}(\text{GOOD}))$ for the model.

The property is not satisfied for branching time logic.

The property is satisfied for linear time logic.
- Invariants
- Axiomatic Verification
- Model-checking
- Behavioral Verification
- Using Abstraction
- Discussion
Behavioral Verification – Process Algebra

Processes are defined as terms of an algebra:

\[ P ::= \text{Nil} \mid K \mid \alpha P \mid P + P \mid P \parallel P \mid P / \alpha \mid K = P \]

where,

- Nil is a constant
- K is a set of process names
- \( \alpha \in A \cup A \cup \{\tau\} \) is an action
- \( + \) is the non-deterministic choice operator (associative, commutative, idempotent, Nil is the neutral element)
- \( \parallel \) is the parallel composition operator (associative, commutative, Nil is the neutral element)

Operational semantics:

- **true**
- \( \alpha . P \rightarrow P \)
- \( P - \alpha \rightarrow P' \quad \alpha \neq a, a \)
- \( P / a - \alpha \rightarrow P' \)
- \( P - \alpha \rightarrow P' \quad K = P \)
- \( K - \alpha \rightarrow P' \)
- \( P_1 - \alpha \rightarrow P'_1 \)
- \( P_1 + P_2 - \alpha \rightarrow P'_1 \)
- \( P_2 - \alpha \rightarrow P'_2 \)
- \( P_1 + P_2 - \alpha \rightarrow P'_2 \)
- \( P_1 \parallel P_2 - \alpha \rightarrow P'_1 \parallel P_2 \)
- \( P_1 \parallel P_2 - \alpha \rightarrow P_1 \parallel P'_2 \)
- \( P_1 \parallel P_2 - \tau \rightarrow P'_1 \parallel P'_2 \)
Behavioral Verification

- Behavioral verification algorithms allow comparison of two processes modulo some observational equivalence.
- Observational equivalence \( \cong \) relates states exhibiting the same behavior abstracted to a set of observable actions.
Behavioral Verification – Observational Equivalence

Observational equivalence is usually parameterized by an observation criterion $[a] = \tau^*a\tau^*$ or $[a] = \tau^*a$

An observational equivalence is a bisimulation, that is a relation which is stable for the transition relation induced by the chosen observation criterion.
Behavioral Verification – Observational Equivalence

Verification method:
- Compute the transition relation for the chosen observation criterion
- Compute $\equiv$ iteratively as the limit $\bigcap_{i=0}^{\infty} \equiv_k$ where
  - $q \equiv_0 q'$ for all $q, q'$
  - $q \equiv_{k+1} q'$ if $\forall q_1 \forall a \ q-[a] \to q_1$ implies $\exists q'_1 \ q'_-[a] \to q'_1$ and $q_1 \equiv_k q'_1$
    $\forall q'_1 \forall a \ q'_-[a] \to q'_1$ implies $\exists q_1 \ q-[a] \to q_1$ and $q_1 \equiv_k q'_1$

- The reduced model $TS/\equiv$ defined by the transition relation
  $[q] - a \to [q']$ iff $q - [a] \to q'$ should be isomorphic to the requirements
Behavioral Verification – Observational Equivalence

- Composability of reduction - observational equivalence should be a congruence

\[ TS_1 \cong TS_2 \cong TS_1/\cong \cong TS_2/\cong \]

- Observational equivalences should be stronger than trace equivalence by taking into account the branching structure

- Non composability is a very common manifestation of lack of modularity for action refinement in concurrent systems
Behavioral Verification – Observational Equivalence

Logics characterize properties of infinite execution sequences while observational equivalences capture similarity of finite behavior.

- Operational semantics reduce parallelism to non-determinism that is interleaving of finite sequences: \((a \cup b)^*\)
- Logic-based semantics can characterize fair (infinite) execution sequences: 
  \[(a \cup b)^\omega - (a \cup b)^*(a^\omega \cup b^\omega)\]

Some issues:
- Compatibility between logical and observational equivalences
- Expressing fairness in temporal logics – the concepts differ for branching time and linear time
 Invariants

 Axiomatic Verification

 Model-checking

 Behavioral Verification

 Using Abstraction

 Discussion
Abstraction

if….
while valid do
  if x<0 then z=x
  else z=-x;
while …

PROGRAM

if x<0 then z:=x
else z:=-x;

SEMANTIC MODEL

x<0
z:=x

x>=0
z:=-x

SEMANTIC MODEL

¬ valid

VALID

¬ valid

¬ valid

ABSTRACT MODEL

valid

¬ valid

¬ valid

¬ valid

b
z:=b

¬ b
z:=¬b

¬ valid

valid

¬ b
z:=¬b

¬ valid

¬ valid

abstraction

semantics
Abstraction

Homomorphisms between transition systems define abstractions

- The set of the traces of the concrete system is contained in the set of the traces of the abstract system
- **Property preservation**: if a property involving universal quantification over execution paths holds for the abstract system, then it holds for the concrete system e.g. $\text{alw}(P)$
- Idea: compute tractable abstractions of infinite state systems and check preserved properties on these abstractions
Abstract interpretation is a general framework for computing abstractions based on the use of Galois connections [Cousot&Cousot 79].

\[ \begin{align*}
\alpha, \gamma & \text{ are monotonic} \\
\text{Id} & \subseteq \gamma \alpha \\
\alpha \gamma & \subseteq \text{Id}
\end{align*} \]

\[ \alpha \mathcal{F} \gamma \] is the best upper approximation of \( \mathcal{F} \) in the abstract lattice.

\[
\begin{array}{c}
\text{The lattice of concrete properties} \\
\gamma \alpha (P) \\
P \\
\text{true} \\
\text{false}
\end{array}
\quad
\begin{array}{c}
\text{The lattice of abstract properties} \\
\alpha (P) \\
\text{true} \\
\text{false}
\end{array}
\quad
\begin{array}{c}
\alpha \mathcal{F} \gamma \\
\text{true} \\
\text{false}
\end{array}
\]
Abstraction – Abstract Interpretation

Abstraction lattices

Computing abstractions

- $\alpha(P_1 \lor P_2) = \text{lub}\{\alpha(P_1), \alpha(P_2)\}$
- $\alpha(P_1 \land P_2) = \text{glb}\{\alpha(P_1), \alpha(P_2)\}$

Examples of operations in abstract domains

Intervals:

- lub$\{[l_1,u_1], [l_2,u_2]\} = [\min\{l_1,l_2\}, \max\{u_1,u_2\}]$
- glb$\{[l_1,u_1], [l_2,u_2]\} = [\min\{u_1,u_2\}, \max\{l_1,l_2\}]$

Convex Polyhedra:

- lub$\{CP_1, CP_2\} = \text{Convex\_Hull}\{CP_1, CP_2\}$
- glb$\{CP_1, CP_2\} = CP_1 \cap CP_2$
Sufficient conditions for the verification of $\text{alw}(P)$

$\alpha_{\text{post}}^{*}(\text{init}) \Rightarrow P$

$\alpha_{\text{pre}}^{*}(\neg P) \wedge \text{init} = \text{false}$
- Invariants
- Axiomatic Verification
- Model-checking
- Behavioral Verification
- Using Abstraction
- Discussion
Discussion – A Historical Overview


A. Pnueli “The Temporal Logic of Programs” FOCS 1977: 46-57

L. Lamport and S. Owicki “Proving Liveness Properties of Concurrent Programs” ACM Transactions on Programming Languages and Systems 4, 3 (July 1982), 455-495.
Discussion – A Historical Overview

P. Cousot, R.Cousot “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints” POPL 1977: 238-252


Discussion – Requirements

- Behavioral requirements are good for characterizing causal dependencies e.g. abstract execution sequences

- Logic-based requirements
  - are good for expressing global properties such as mutual exclusion, termination, fairness
  - allow a deep understanding and formalization of properties of concurrent systems.

Nonetheless, writing rigorous specifications in temporal logic is not trivial. The declarative and dense style is not always easy to master and understand.

- In practice, we need a combination of both logic-based and behavioral styles, e.g. PSL

- We lack adequate formalisms for extra-functional requirements
  - security and privacy properties
  - reconfigurability (e.g. non interference of features),
  - quality of service (e.g. jitter).
**Discussion – Verification Techniques**

**Algorithmic verification** (model-checking, abstract interpretation, static analysis)

- Widely applied to hardware verification – tools: nuSMV, proprietary industrial tools
- Verification of abstract software models – tools: SLAM, Blast, ASTREE, Polyspace, SPIN
- Behavioral verification of security protocols – tools: FDR2
- Verification of simple models – tools for timed automata e.g. UPPAAL and hybrid automata e.g. Hy Tech

**Axiomatic Verification**

- Verification technology based on provers such as Coq, Isabelle, PVS, ACL2, HOL
- Applied mainly to intensively used sequential software components e.g. the seL4 kernel, CompCert C verified compiler.
Correctness by checking is a relative judgment “Are we building the system right?” It would be an answer to the question “Are we building the right system?” if

1. requirements could be correctly formalized, sound and complete

*Effective use of rigorous requirement specification languages for real-life systems is problematic*

2. system models could faithfully represent the system behavior interacting with its environment

*Generating models even for very simple systems, such as the node of a wireless sensor network, requires understanding intricate interaction between application software and the underlying execution platform*

Correctness-by-checking contributes to trustworthiness but it is restricted to requirements that can be formalized and checked efficiently

For optimization requirements, a more natural approach for their satisfaction is by enforcing or synthesis rather than by checking
Some overwhelming obstacles

1. Undecidability $\Rightarrow$ non converging computation of fixpoints
   - To prove properties of infinite state systems we need induction e.g.
     invariants are a natural induction principle
   - Induction principles should be invented!
   - Algorithmic verification is symbolic execution guided by the property
     to be verified – It can only compute approximations that provide
     necessary or sufficient conditions

2. Lack of compositional verification methods – proving a global system
   property from properties of its constituents

3. Combination of automated and manual verification proves to be ineffective
Formal methods are mathematically based techniques for the specification, development and verification of software and hardware systems.

They have mobilized vast communities of researchers and have been a very active research area for decades on topics including logic, semantics, proof techniques, verification, type theory, theory of concurrency, etc.

They are applied when it is possible to make automated proofs or when the cost of faults is high.

Existing theory:
- allows a deep insight on hard and still unsolved problems raised by correct system development rather than practicable and scalable techniques
- is mainly based on transition systems and can be badly lifted from semantic to syntactic level

The alternative: move from a posteriori correctness-by-checking to correctness-by-construction
Thank You