SMT-based Model Checking

Cesare Tinelli

The University of Iowa.

Formal Techniques Summer School

Modeling Computational Systems

Software or hardware systems can be often represented as a *state transition system* $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ where

- $\mathcal{S}$ is a set of *states*
- $\mathcal{I} \subseteq \mathcal{S}$ is a set of *initial states*
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a (right-total) *transition relation*
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{Pr}$ is a *labeling function* where Pr is a set of *base predicates* in some logic

Typically, the base predicates denote variable-value pairs $x = v$

Model Checking

Software or hardware systems can be often represented as a *state transition system*, or *model*, $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$

$\mathcal{M}$ is a model both in

1. an engineering sense: a mock-up of the real system

   and

2. a mathematical logic sense: a Kripke structure in some modal logic

Model Checking

Software or hardware systems can be often represented as a *state transition system*, or *model*, $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$

$\mathcal{M}$ is a model both in

1. an engineering sense: we can analyze and check $\mathcal{M}$ instead of the real system

   and

2. a mathematical logic sense: we can make the analysis formal and rely on (semi)automated tools

Model Checking

The functional properties of a computational system can be expressed as *temporal* properties

- for a suitable model $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ of the system
- in a suitable temporal logic

Model Checking

The functional properties of a computational system can be expressed as *temporal* properties

- for a suitable model $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ of the system
- in a suitable temporal logic

Two main classes of properties:

- *Safety properties*: nothing bad ever happens
- *Liveness properties*: something good eventually happens

Model Checking

The functional properties of a computational system can be expressed as *temporal* properties

- for a suitable model $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ of the system
- in a suitable temporal logic

Two main classes of properties:

- *Safety properties*: nothing bad ever happens
- *Liveness properties*: something good eventually happens

We will focus on checking safety in this talk

Talk Roadmap

- Checking safety properties
- Logic-based model checking
- Satisfiability Modulo Theories
  - theories
  - solvers
- SMT-based model checking
  - main approaches
  - k-Induction
    - basic method
    - enhancements

Safety Properties

Let $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ be a transition system

The set $\mathcal{R}$ of *reachable states (of $\mathcal{M}$)* is the smallest subset of $\mathcal{S}$ satisfying the following constraints

1. $\mathcal{I} \subseteq \mathcal{R}$ (initial states are reachable)
2. $\mathcal{R} \bowtie \mathcal{T} \subseteq \mathcal{R}$ ($\mathcal{T}$-successors of reachable states are reachable)

$\mathcal{M}$ is *safe* wrt a *state property* $\mathcal{P} \subseteq \mathcal{S}$ iff $\mathcal{P} \cap \mathcal{R} = \emptyset$

A state property $\mathcal{P}$ is *invariant (for $\mathcal{M}$)* iff $\mathcal{R} \subseteq \mathcal{P}$

Note: $\mathcal{M}$ is safe wrt $\mathcal{P}$ iff $\overline{\mathcal{P}} = \mathcal{S} \setminus \mathcal{P}$ is invariant

Example: Resettable Counter

Vars

input bool r

int c, n

Initialization

c := 1

n := 3

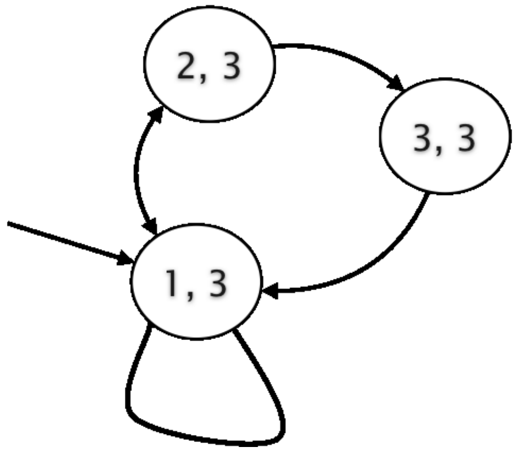
Transitions

n':= n

c' := if (r' or c = n)

then 1

else c + 1



$\mathcal{S} := \mathbb{Z} \times \mathbb{Z}$

$\mathcal{I} := \{(1,3)\}$

$\mathcal{T} := \{((1,3),(1,3)),((1,3),(2,3)),\ldots\}$

$\mathcal{R} := \{(1,3),(2,3),(3,3)\}$

$\mathcal{P} := \{(5,3)\}$ (safety)

Checking Safety

In principle, to check that $\mathcal{M}$ is safe wrt $\mathcal{P}$ it suffices to

1. compute $\mathcal{R}$ and
2. check that $\mathcal{P} \cap \mathcal{R} = \emptyset$

This can be done explicitly only if $\mathcal{S}$ is finite, and relatively small ($< 10M$ states)

Alternatively, we can represent $\mathcal{M}$ symbolically and use

- BDD-based methods, if $\mathcal{S}$ is finite,
- automata-based methods, or
- logic-based methods

Logic-based Symbolic Model Checking

Applicable if we can encode $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ in some (classical) logic $\mathbb{L}$ with decidable entailment $\models_{\mathbb{L}}$

Logic-based Symbolic Model Checking

Applicable if we can encode $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ in some (classical) logic $\mathbb{L}$ with decidable entailment $\models_{\mathbb{L}}$

($\varphi \models_{\mathbb{L}} \psi$ iff every $\mathbb{L}$-model of φ is a model of ψ)

Examples of $\mathbb{L}$:

- Propositional logic
- Quantified Boolean Formulas
- Bernay-Schönfinkel logic
- Quantifier-free real (or linear integer) arithmetic with arrays and uninterpreted functions
- ...

Logic-based Symbolic Model Checking

Applicable if we can encode $\mathcal{M} = (\mathcal{S}, \mathcal{I}, \mathcal{T}, \mathcal{L})$ in some (classical) logic $\mathbb{L}$ with decidable entailment $\models_{\mathbb{L}}$

Given a set X of variables and a set V of values in $\mathbb{L}$,

- states $\sigma \in \mathcal{S}$ are identified with their label $\mathcal{L}(s)$ and represented as n-tuples in V^n
- $\mathcal{I}$ is encoded by a formula $I[\mathbf{x}]$ with free variables $\mathbf{x}$ s.t.
  $$\sigma \in I \quad \text{iff} \quad \models_{\mathbb{L}} I[\sigma]$$
- $\mathcal{T}$ is encoded by a formula $T[\mathbf{x}, \mathbf{x}']$ s.t.
  $$\models_{\mathbb{L}} T[\sigma, \sigma'] \text{ for all } (\sigma, \sigma') \in \mathcal{T}.$$
- State properties are encoded by formulas $P[\mathbf{x}]$

Notation: if $\mathbf{x} = (x_1, \ldots, x_n)$ and $\sigma = (v_1, \ldots, v_n)$, then

$$\phi[\sigma] := \phi[v_1/x_1, \ldots, v_n/x_n]$$

Strongest Inductive Invariant

The *strongest inductive invariant (for $\mathcal{M}$ in $\mathbb{L}$)* is a formula $R[\mathbf{x}]$ s.t. $\models_{\mathbb{L}}\ R[\sigma]$ iff $\sigma \in \mathcal{R}$

Strongest Inductive Invariant

The *strongest inductive invariant (for $\mathcal{M}$ in $\mathbb{L}$)* is a formula $R[\mathbf{x}]$ s.t. $\models_{\mathbb{L}} R[\sigma]$ iff $\sigma \in \mathcal{R}$

Suppose we can compute R from I and T

Then, checking that $\mathcal{M}$ is safe wrt a property $P[\mathbf{x}]$ reduces to checking that $R[\mathbf{x}] \models_{\mathbb{L}} \neg P[\mathbf{x}]$

Strongest Inductive Invariant

The *strongest inductive invariant (for $\mathcal{M}$ in $\mathbb{L}$)* is a formula $R[\mathbf{x}]$ s.t. $\models_{\mathbb{L}} R[\sigma]$ iff $\sigma \in \mathcal{R}$

Suppose we can compute R from I and T

Then, checking that $\mathcal{M}$ is safe wrt a property $P[\mathbf{x}]$ reduces to checking that $R[\mathbf{x}] \models_{\mathbb{L}} \neg P[\mathbf{x}]$

Problem: R may be very expensive or impossible to compute, or not even representable in $\mathbb{L}$

Strongest Inductive Invariant

The *strongest inductive invariant (for $\mathcal{M}$ in $\mathbb{L}$)* is a formula $R[\mathbf{x}]$ s.t. $\models_{\mathbb{L}} R[\sigma]$ iff $\sigma \in \mathcal{R}$

Suppose we can compute R from I and T

Then, checking that $\mathcal{M}$ is safe wrt a property $P[\mathbf{x}]$ reduces to checking that $R[\mathbf{x}] \models_{\mathbb{L}} \neg P[\mathbf{x}]$

Problem: R may be very expensive or impossible to compute, or not even representable in $\mathbb{L}$

Logic-based model checking is about approximating R as efficiently as possible and as precisely as needed

Main Logic-based Approaches

- Bounded model checking [CBRZ01, AMP06, BHvMW09]
- Interpolation-based model checking [McM03, McM05]
- Model checking without unrolling [BM07, Bra10]
- Temporal induction [SSS00, dMRS03, HT08]
- Backward reachability [ACJT96, GR10]
- ...

Past accomplishments: mostly based on propositional logic, with SAT solvers as reasoning engines

Next frontier: based on SMT logics, with SMT solvers as reasoning engines [Seb07, BSST09]

Model Checking Modulo Theories

We invariably reason about transition systems in the context of some theory of their data types

Examples

- Pipelined microprocessors: theory of equality, atoms like $f(g(a, b), c) = g(c, a)$
- Timed automata: theory of integers/reals, atoms like $x - y < 2$
- General software: combination of theories, atoms like $a[2 * j + 1] + x \geq car(l) - f(x)$

Such reasoning can be reduced to checking the satisfiability of certain formulas in (or *modulo*) the theory.

Satisfiability Modulo Theories

Let T be a first-order theory of signature Σ

The T-satisfiability problem for a class $\mathcal{C}^{\Sigma}$ of Σ-formulas consists in deciding for any formula $\varphi[\mathbf{x}] \in \mathcal{C}^{\Sigma}$ whether $T \cup \{\exists \mathbf{x}.\, \varphi\}$ is satisfiable

Satisfiability Modulo Theories

Let T be a first-order theory of signature Σ

The T-satisfiability problem for a class $\mathcal{C}^\Sigma$ of Σ-formulas consists in deciding for any formula $\varphi[\mathbf{x}] \in \mathcal{C}^\Sigma$ whether $T \cup \{\exists \mathbf{x}.\, \varphi\}$ is satisfiable

Fact: the T-satisfiability of quantifier-free formulas is decidable for many theories T of interest in model checking

Satisfiability Modulo Theories

Let T be a first-order theory of signature Σ

The T-satisfiability problem for a class $\mathcal{C}^{\Sigma}$ of Σ-formulas consists in deciding for any formula $\varphi[\mathbf{x}] \in \mathcal{C}^{\Sigma}$ whether $T \cup \{\exists \mathbf{x}.\, \varphi\}$ is satisfiable

Fact: the T-satisfiability of quantifier-free formulas is decidable for many theories T of interest in model checking

- Equality with "Uninterpreted Function Symbols"
- Linear Arithmetic (Real and Integer)
- Arrays (i.e., updatable maps)
- Finite sets and multisets
- Inductive data types (enumerations, lists, trees, ...)
- ...

Satisfiability Modulo Theories

Let T be a first-order theory of signature Σ

The T-satisfiability problem for a class $\mathcal{C}^\Sigma$ of Σ-formulas consists in deciding for any formula $\varphi[\mathbf{x}] \in \mathcal{C}^\Sigma$ whether $T \cup \{\exists \mathbf{x}.\, \varphi\}$ is satisfiable

Fact: the T-satisfiability of quantifier-free formulas is decidable for many theories T of interest in model checking

Thanks to advances in SAT and in decision procedures, this can be done very efficiently in practice by current SMT solvers

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

Are typically built to be embeddable in larger systems: they are on-line, incremental, restartable, . . .

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

Are typically built to be embeddable in larger systems: they are on-line, incremental, restartable, . . .

Provide additional functionalities besides satisfiability checking

- satisfying assignments
- unsatisfiable cores
- explanations
- interpolants
- proof objects

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

Are typically built to be embeddable in larger systems: they are on-line, incremental, restartable, . . .

Provide additional functionalities besides satisfiability checking

Are being extended to reason efficiently, if incompletely, with quantified formulas as well

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

Are typically built to be embeddable in larger systems: they are on-line, incremental, restartable, ...

Provide additional functionalities besides satisfiability checking

Are being extended to reason efficiently, if incompletely, with quantified formulas as well

Increasingly conform to a standard I/O language: the SMT-LIB format

SMT Solvers

Differ from traditional theorem provers for having built-in theories, and using specialized methods to reason about them

Are typically built to be embeddable in larger systems: they are on-line, incremental, restartable, ...

Provide additional functionalities besides satisfiability checking

Are being extended to reason efficiently, if incompletely, with quantified formulas as well

Increasingly conform to a standard I/O language: the SMT-LIB format

Are now incorporated into a variety of FM tools

Model Checking: SMT or SAT?

SMT encodings in model checking provide several advantages over SAT encodings

- Boolean formulas $\longrightarrow$ (unquantified) first-order formulas
- more powerful language
- satisfiability still efficiently decidabile
- more natural and compact encodings
- greater scalability
- similar high level of automation
- work indifferently for finite and infinite state systems

Model Checking: SMT or SAT?

SMT encodings in model checking provide several advantages over SAT encodings

- Boolean formulas $\longrightarrow$ (unquantified) first-order formulas
- more powerful language
- satisfiability still efficiently decidabile
- more natural and compact encodings
- greater scalability
- similar high level of automation
- work indifferently for finite and infinite state systems

SMT-based model checking techniques blur the line between traditional model checking and deductive verification

Talk Roadmap

- √ Checking safety properties
- √ Logic-based model checking
- √ Satisfiability Modulo Theories
  - √ theories
  - √ solvers
- SMT-based model checking
  - main approaches
  - k-Induction
    - basic method
    - enhancements

SMT-based Model Checking

A few approaches:

- Predicate abstraction + finite model checking
- Bounded model checking
- Interpolation-based model checking
- Backward reachability
- Temporal induction (aka k-induction)

SMT-based Model Checking

A few approaches:

- Predicate abstraction + finite model checking
- Bounded model checking
- Interpolation-based model checking
- Backward reachability
- Temporal induction (aka k-induction)

I will focus on temporal induction

SMT-based Model Checking

A few approaches:

- Predicate abstraction + finite model checking
- Bounded model checking
- Interpolation-based model checking
- Backward reachability
- Temporal induction (aka k-induction)

I will focus on temporal induction

Reasons:

- it does not need advanced SMT features (such as interpolation, quantifier elimination), and ...

SMT-based Model Checking

A few approaches:

- Predicate abstraction + finite model checking
- Bounded model checking
- Interpolation-based model checking
- Backward reachability
- Temporal induction (aka k-induction)

I will focus on temporal induction

Reasons:

- it does not need advanced SMT features (such as interpolation, quantifier elimination), and ...
- I have more experience with it $\ddot\smile$

Technical Preliminaries

Let's fix

- $\mathbb{L}$, a logic whose quantifier-free (QF) fragment is decided by an SMT solver

  (e.g., linear arithmetic and EUF)

- $S = (I[\mathbf{x}], T[\mathbf{x}, \mathbf{x}'])$, a QF encoding of a transition system in $\mathbb{L}$

- $P[x]$, a QF state property to be proven invariant for S

Example: Parametric Resettable Counter

Vars

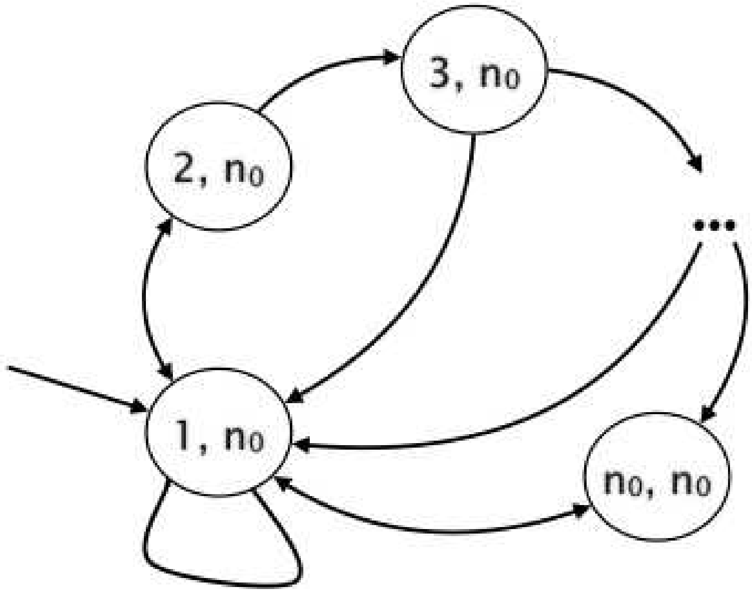
input pos int n_0
input bool r
int c, n

Initialization

c := 1
n := n_0

Transitions

n' := n
c' := if (r' or c = n)
then 1
else c + 1



The transition relation contains infinitely many instances of the schema above, one for each $n_0 > 0$

Example: Parametric Resettable Counter

Vars

input pos int n_0
input bool r
int c, n

Initialization

c := 1
n := n_0

Transitions

n' := n
c' := if (r' or c = n)
then 1
else c + 1

$$\mathbf{x} := (c, n, r, n_0)$$

$$I[\mathbf{x}] := (c = 1) \wedge (n = n_0)$$

$$\begin{aligned} T[\mathbf{x}, \mathbf{x}'] := & \ (n' = n) \\ & \wedge (r' \vee (c = n) \to (c' = 1)) \\ & \wedge (\neg r' \wedge (c \neq n) \to (c' = c + 1)) \end{aligned}$$

$$P[\mathbf{x}] := c < n + 1$$

Inductive Reasoning

Let $S = (I[\mathbf{x}], T[\mathbf{x}, \mathbf{x}'])$

To prove $P[x]$ invariant for S it suffices to show that it is *inductive* for S, i.e.,

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ (base case)

   and

2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$ (inductive step)

Inductive Reasoning

Let $S = (I[\mathbf{x}], T[\mathbf{x}, \mathbf{x}'])$

To prove $P[x]$ invariant for S it suffices to show that it is *inductive* for S, i.e.,

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ (base case)

   and

2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$ (inductive step)

An SMT solver can check both entailments above
($\varphi \models_{\mathbb{L}} \psi$ iff $\varphi \wedge \neg\psi$ is unsatisfiable in $\mathbb{L}$)

Inductive Reasoning

Let $S = (I[\mathbf{x}], T[\mathbf{x}, \mathbf{x}'])$

To prove $P[x]$ invariant for S it suffices to show that it is *inductive* for S, i.e.,

1. $I[\mathbf{x}] \ \models_{\mathbb{L}} \ P[\mathbf{x}]$ (base case)

   and

2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \ \models_{\mathbb{L}} \ P[\mathbf{x}']$ (inductive step)

Problem: Not all invariants are inductive

Example: In the parametric resettable counter, $P = c \leq n + 1$ is invariant but (2) above is falsifiable, e.g., by $(c, n, r) = (4, 3, \mathit{false})$ and $(c, n, r)' = (5, 3, \mathit{false})$

Induction: Sound but Imprecise

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ (base case)

   and

2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$ (inductive step)

Cases:

base case	ind. step	P invariant
holds	holds	yes
fails	*	no
holds	fails	?

In last case, $P[\sigma] \wedge T[\sigma, \sigma'] \wedge \neg P[\sigma']$ is sat for some σ, σ'
Then, σ could be

- reachable in $k > 0$ steps (making P non-invariant) or
- unreachable

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$
2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$
2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$

2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

  Difficult to automate

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$
2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

  Difficult to automate

- Strengthen T: Find another invariant $Q[\mathbf{x}]$ and do induction with $Q[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge Q[\mathbf{x}']$ instead of $T[\mathbf{x}, \mathbf{x}']$

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ 2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

  Difficult to automate

- Strengthen T: Find another invariant $Q[\mathbf{x}]$ and do induction with $Q[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge Q[\mathbf{x}']$ instead of $T[\mathbf{x}, \mathbf{x}']$

  Difficult to automate (but lots of recent progress)

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ 2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

  Difficult to automate

- Strengthen T: Find another invariant $Q[\mathbf{x}]$ and do induction with $Q[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge Q[\mathbf{x}']$ instead of $T[\mathbf{x}, \mathbf{x}']$

  Difficult to automate (but lots of recent progress)

- Consider longer T-paths: k-induction

Improving Induction's Precision

1. $I[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$ 2. $P[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \models_{\mathbb{L}} P[\mathbf{x}']$

A few options:

- Strengthen P: Find a property Q s.t. $Q[\mathbf{x}] \models_{\mathbb{L}} P[\mathbf{x}]$, and prove Q inductive

  Difficult to automate

- Strengthen T: Find another invariant $Q[\mathbf{x}]$ and do induction with $Q[\mathbf{x}] \wedge T[\mathbf{x}, \mathbf{x}'] \wedge Q[\mathbf{x}']$ instead of $T[\mathbf{x}, \mathbf{x}']$

  Difficult to automate (but lots of recent progress)

- Consider longer T-paths: k-induction

  Easy to automate (but fairly weak in its basic form)

Basic k-Induction (Naive Algorithm)

Notation: $I^i := I[\mathbf{x}^{(i)}]$, $P^i := P[\mathbf{x}^{(i)}]$, $T^i := T[\mathbf{x}^{(i-1)}, \mathbf{x}^{(i)}]$

for $i = 0$ **to** ∞ **do**
  if not $(I^0 \wedge T^1 \wedge \cdots \wedge T^i \models_{\mathbb{L}} P^i)$ **then**
    return fail
  if $(P^0 \wedge \cdots \wedge P^i \wedge T^1 \wedge \cdots \wedge T^{i+1} \models_{\mathbb{L}} P^{i+1})$ **then**
    return success

P is *k-inductive* for some $k \geq 0$, if the first entailment holds for all $i = 0, \ldots, k$ and the second entailment holds for $i = k$

Example: In the parametric resettable counter,

$$P := c \leq n + 1$$

is 1-inductive, but not 0-inductive

Basic k-Induction (Naive Algorithm)

for $i = 0$ **to** ∞ **do**
  if not $(I^0 \wedge T^1 \wedge \cdots \wedge T^i \models_{\mathbb{L}} P^i)$ **then**
    return fail
  if $(P^0 \wedge \cdots \wedge P^i \wedge T^1 \wedge \cdots \wedge T^{i+1} \models_{\mathbb{L}} P^{i+1})$ **then**
    return success

P is *k-inductive* for some $k \geq 0$, if the first entailment holds for all $i = 0, \ldots, k$ and the second entailment holds for $i = k$

Note:

- inductive $=$ 0-inductive
- k-inductive $\Rightarrow$ $(k+1)$-inductive $\Rightarrow$ invariant
- some properties are invariant but not k-inductive for any k

Basic k-Induction with SMT Solvers

Solver maintains current set of *asserted* formulas

Two solver instances: b, i

```
assert_b(I_0)
k := 0
loop
    assert_b(T_k)    // T_0 = true by convention
    if not entailed_b(P_k) then return cex_b()
    assert_i(P_k)
    assert_i(T_{k+1})
    if entailed_i(P_{k+1}) then return success
    k := k + 1
```

assert$_s(\varphi)$:	add formula φ to asserted formulas
entailed$_s(\varphi)$:	check if φ is entailed by asserted formulas
cex$_s()$:	return counterexample after failed entailment

Enhancements to k-Induction

- Path compression
- Termination checks
- Property strengthening
- Invariant generation
- Multiple property checking

Path compression (simplified)

Let $E[\mathbf{x},\mathbf{y}]$ be a qff s.t. $E[\mathbf{x},\mathbf{y}] \models_{\mathbb{L}} \forall\mathbf{z}\,(T[\mathbf{x},\mathbf{z}] \Leftrightarrow T[\mathbf{y},\mathbf{z}])$

(**Ex:** $E[\mathbf{x},\mathbf{y}] := \mathbf{x} = \mathbf{y}$)

Path compression (simplified)

Let $E[\mathbf{x}, \mathbf{y}]$ be a qff s.t. $E[\mathbf{x}, \mathbf{y}] \models_{\mathbb{L}} \forall \mathbf{z}\, (T[\mathbf{x}, \mathbf{z}] \Leftrightarrow T[\mathbf{y}, \mathbf{z}])$
(**Ex:** $E[\mathbf{x}, \mathbf{y}] := \ \mathbf{x} = \mathbf{y}$)

Can strengthen the premise of the inductive step as follows

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge C^k \models_{\mathbb{L}} P^{k+1}$$

where $C^k := \bigwedge_{0 \leq i < j \leq k} \neg E[\mathbf{x}_i, \mathbf{x}_j]$

Path compression (simplified)

Let $E[\mathbf{x}, \mathbf{y}]$ be a qff s.t. $E[\mathbf{x}, \mathbf{y}] \models_{\mathbb{L}} \forall \mathbf{z}\, (T[\mathbf{x}, \mathbf{z}] \Leftrightarrow T[\mathbf{y}, \mathbf{z}])$
(**Ex:** $E[\mathbf{x}, \mathbf{y}] := \mathbf{x} = \mathbf{y}$)

Can strengthen the premise of the inductive step as follows

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge C^k \models_{\mathbb{L}} P^{k+1}$$

where $C^k := \bigwedge_{0 \le i < j \le k} \neg E[\mathbf{x}_i, \mathbf{x}_j]$

Rationale: Let $\pi := \sigma_0, \ldots, \sigma_i, \sigma_{i+1}, \ldots, \sigma_j, \sigma_{j+1}, \ldots, \sigma_{k+1}$ be a path that breaks (2), with $E[\sigma_i, \sigma_j]$ and $i < j$
If π is part of an actual execution of S, so is the shorter path $\sigma_0, \ldots, \sigma_i, \sigma_{j+1}, \ldots, \sigma_{k+1}$

Path compression (simplified)

Let $E[\mathbf{x},\mathbf{y}]$ be a qff s.t. $E[\mathbf{x},\mathbf{y}] \models_{\mathbb{L}} \forall \mathbf{z}\,(T[\mathbf{x},\mathbf{z}] \Leftrightarrow T[\mathbf{y},\mathbf{z}])$
(**Ex:** $E[\mathbf{x},\mathbf{y}] := \mathbf{x} = \mathbf{y}$)

Can further strengthen the premise of the inductive step with

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge C^k \wedge N^k \models_{\mathbb{L}} P^{k+1}$$

where $N^k := \bigwedge_{1 \le i \le k+1} \neg I[\mathbf{x}_i]$

Path compression (simplified)

Let $E[\mathbf{x},\mathbf{y}]$ be a qff s.t. $E[\mathbf{x},\mathbf{y}] \models_{\mathbb{L}} \forall \mathbf{z}\,(T[\mathbf{x},\mathbf{z}] \Leftrightarrow T[\mathbf{y},\mathbf{z}])$
(**Ex:** $E[\mathbf{x},\mathbf{y}] := \mathbf{x} = \mathbf{y}$)

Can further strengthen the premise of the inductive step with

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge C^k \wedge N^k \models_{\mathbb{L}} P^{k+1}$$

where $N^k := \bigwedge_{1 \leq i \leq k+1} \neg I[\mathbf{x}_i]$

Rationale: if
$\sigma_0, \ldots, \sigma_i, \ldots, \sigma_{k+1}$ breaks (2) and $I[\sigma_i]$, then
$\sigma_i, \ldots, \sigma_{k+1}$ breaks the base case in the first place

Path compression (simplified)

Let $E[\mathbf{x}, \mathbf{y}]$ be a qff s.t. $E[\mathbf{x}, \mathbf{y}] \models_{\mathbb{L}} \forall \mathbf{z} \, (T[\mathbf{x}, \mathbf{z}] \Leftrightarrow T[\mathbf{y}, \mathbf{z}])$
(**Ex:** $E[\mathbf{x}, \mathbf{y}] := \mathbf{x} = \mathbf{y}$)

Can further strengthen the premise of the inductive step with

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge C^k \wedge N^k \models_{\mathbb{L}} P^{k+1}$$

where $N^k := \bigwedge_{1 \leq i \leq k+1} \neg I[\mathbf{x}_i]$

Better E's than $\mathbf{x} = \mathbf{y}$ can be generated by an analysis of S

More sophisticated notions of compressions have been proposed [dMRS03]

Termination check

Recall $C^k := \bigwedge_{0 \le i < j \le k} \neg E[\mathbf{x}_i, \mathbf{x}_j]$

(0) **for** $k = 0$ **to** ∞ **do**
(0)   **if not** $(I^0 \wedge T^1 \wedge \cdots \wedge T^k \models_{\mathbb{L}} P^k)$ **then**
(0)   **return** fail
(0)   **if** $(P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1})$ **then**
(0)   **return** success
(0)   **if** $(I^0 \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} \neg C^{k+1})$ **then**
(0)   **return** success

Termination check

Recall $C^k := \bigwedge_{0 \le i < j \le k} \neg E[\mathbf{x}_i, \mathbf{x}_j]$

$$
\begin{array}{l}
\textbf{for } k = 0 \textbf{ to } \infty \textbf{ do} \\
\quad \textbf{if not } (I^0 \wedge T^1 \wedge \cdots \wedge T^k \models_{\mathbb{L}} P^k) \textbf{ then} \\
\quad\quad \textbf{return } \mathsf{fail} \\
\quad \textbf{if } (P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1}) \textbf{ then} \\
\quad\quad \textbf{return } \mathsf{success} \\
\quad \textbf{if } (I^0 \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} \neg C^{k+1}) \textbf{ then} \\
\quad\quad \textbf{return } \mathsf{success}
\end{array}
$$

Rationale: If the last test succeeds, every execution of length $k+1$ is compressible to a shorter one.

Hence, the whole reachable state space has been covered without finding counterexamples for P

Termination check

Recall $C^k := \bigwedge_{0 \le i < j \le k} \neg E[\mathbf{x}_i, \mathbf{x}_j]$

(0) **for** $k = 0$ **to** ∞ **do**
(0)   **if not** $(I^0 \wedge T^1 \wedge \cdots \wedge T^k \models_{\mathbb{L}} P^k)$ **then**
(0)   **return** fail
(0)   **if** $(P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1})$ **then**
(0)   **return** success
(0)   **if** $(I^0 \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} \neg C^{k+1})$ **then**
(0)   **return** success

Note: The termination check may slow down the process but increases precision in some cases

It makes k-induction complete for finite states systems, and some classes of infinite state ones (e.g., timed automata)

Property Strengthening

Suppose in the k-induction loop the SMT solver finds a counterexample $\sigma_0, \ldots, \sigma_{k+1}$ for

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1}$$

Property Strengthening

Suppose in the k-induction loop the SMT solver finds a counterexample $\sigma_0, \ldots, \sigma_{k+1}$ for

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1}$$

Then this property is satisfied by σ_0:

$$F[\mathbf{x}_0] := \exists x_1, \ldots, x_{k+1}(P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge \neg P^{k+1})$$

Property Strengthening

Suppose in the k-induction loop the SMT solver finds a counterexample $\sigma_0, \ldots, \sigma_{k+1}$ for

$$2. \quad P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \models_{\mathbb{L}} P^{k+1}$$

Then this property is satisfied by σ_0:

$$F[\mathbf{x}_0] := \exists x_1, \ldots, x_{k+1} (P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge \neg P^{k+1})$$

(Naive) Algorithm:

1. find a QFF $B[\mathbf{x}]$ satisfied by σ_0 s.t. $B[\mathbf{x}] \models_{\mathbb{L}} F[\mathbf{x}]$,
2. restart the process with $P[\mathbf{x}] \wedge \neg B[\mathbf{x}]$ in place of $P[\mathbf{x}]$

Correctness of Property Strengthening

$$F[\mathbf{x}_0] := \exists x_1, \ldots, x_{k+1}\, (P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge \neg P^{k+1})$$

When F is satisfied by some σ_0, we

1. find a QFF $B[\mathbf{x}]$ satisfied by σ_0 s.t. $B[\mathbf{x}] \models_{\mathbb{L}} F[\mathbf{x}]$,
2. replace $P[\mathbf{x}]$ with $Q[\mathbf{x}] := P[\mathbf{x}] \wedge \neg B[\mathbf{x}]$,
3. restart the process

- If all states satisfying B are unreachable, we can remove them from consideration in the inductive step
- Otherwise, P is not invariant and the base case is guaranteed to fail with Q

Viability of Property Strengthening

$$F[\mathbf{x}_0] := \exists x_1, \ldots, x_{k+1}\,(P^0 \wedge \cdots \wedge P^k \wedge T^1 \wedge \cdots \wedge T^{k+1} \wedge \neg P^{k+1})$$

When F is satisfied by some σ_0, we

1. find a QFF $B[\mathbf{x}]$ satisfied by σ_0 s.t. $B[\mathbf{x}] \models_{\mathbb{L}} F[\mathbf{x}]$,
2. replace $P[\mathbf{x}]$ with $Q[\mathbf{x}] := P[\mathbf{x}] \wedge \neg B[\mathbf{x}]$,
3. restart the process

- Computing a B equivalent to F requires QE, which may be impossible or very expensive
- Under-approximating F might be cheaper but less effective in pruning unreachable states.

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction

2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction

2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

Correctness: states not satisfying J are definitely unreachable and so can be pruned

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction

2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

Correctness: states not satisfying J are definitely unreachable and so can be pruned

Viability: can use any non-property-driven method for invariant generation (abstract interpr., template-based, ...)

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction

2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

Correctness: states not satisfying J are definitely unreachable and so can be pruned

Viability: can use any non-property-driven method for invariant generation (abstract interpr., template-based, ...)

Effectiveness: when P is invariant, can substantially improve

- speed, by making P k-inductive for a smaller k, and
- precision, by turning P from k-inductive for no k to k-inductive for some k

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction
2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

Shortcomings: Invariants are computed independently from P and so may not prune the right unreachable states

(Undirected) Invariant Generation

1. Generate QF invariants for S independently from P, either before or in parallel with k-induction

2. For each (proven) invariant $J[\mathbf{x}]$, add $J^0 \wedge \cdots \wedge J^{k+1}$ to the induction step

Shortcomings: Invariants are computed independently from P and so may not prune the right unreachable states

Adding too many invariants may swamp the SMT solver

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Proving them separately is time consuming and ineffective

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Proving them separately is time consuming and ineffective

Proving them together as $P := P^1 \wedge \cdots \wedge P^n$ is inadequate if

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Proving them separately is time consuming and ineffective

Proving them together as $P := P^1 \wedge \cdots \wedge P^n$ is inadequate if

- some are invariants and some are not:
  then the whole P is not invariant

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Proving them separately is time consuming and ineffective

Proving them together as $P := P^1 \wedge \cdots \wedge P^n$ is inadequate if

- some are invariants and some are not:
  then the whole P is not invariant
- they are k-inductive for different k's:
  then P is k-inductive only for the largest k

Multiple Property Checking

Often one wants to prove several properties $P^1, \ldots, P^n$

Proving them separately is time consuming and ineffective

Proving them together as $P := P^1 \wedge \cdots \wedge P^n$ is inadequate if

- some are invariants and some are not:
  then the whole P is not invariant
- they are k-inductive for different k's:
  then P is k-inductive only for the largest k

Solution: Incremental multi-property k-induction

Incremental Multi-Property k-Induction

Main idea:

Incremental Multi-Property k-Induction

Main idea:

- Use $P^1 \wedge \cdots \wedge P^n$ but be aware of its components
- When basic case fails,
  1. identify falsified properties
  2. remove them from the problem
  3. repeat the step

Incremental Multi-Property k-Induction

Main idea:

- Use $P^1 \wedge \cdots \wedge P^n$ but be aware of its components
- When basic case fails,
  1. identify falsified properties
  2. remove them from the problem
  3. repeat the step
- When inductive step fails,
  1. set falsified properties aside for next iteration (with increased k)
  2. repeat step and (1) until success or no more properties
  3. add proven properties as invariants for next iteration

Incremental Multi-Property k-Induction

Pros:

- Much better from an HCI point of view
- Proving multiple invariants in conjunction is easier than proving them separately
- adding proven properties as invariants often obviates the need for external invariants

Incremental Multi-Property k-Induction

Pros:

- Much better from an HCI point of view
- Proving multiple invariants in conjunction is easier than proving them separately
- adding proven properties as invariants often obviates the need for external invariants

Cons:

- More complex implementation
- Having several unrelated properties can diminish the effectiveness of simplifications based on the *cone of influence*.

Next Directions for SMT-based MC

- Quantifiers are often needed to encode
  - parametrized model checking problems
    (coming, e.g., from multi-process systems)
  - problems with arrays

- New SMT techniques are needed to generate/work with transition relations, interpolants, invariants, etc., with quantifiers

- We are starting to see some promising work in this direction, but much is left to do

References

[**AMP06**] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In *Proceedings of the 13th International SPIN Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *LNCS*, pages 146–162. Springer, 2006

[**ACJT96**] P. A. Abdulla, K. Cerans, B. Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 313–321. IEEE Computer Society, 1996

[**Bie09**] A. Biere. Bounded model checking. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 14, pages 455–481. IOS Press, February 2009

[**BM07**] A. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design*, pages 173–180, 2007

[**Bra10**] A. Bradley. Sat-based model checking without unrolling. In *In Proc. Verification, Model-Checking, and Abstract-Interpretation (VMCAI)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 2010

References

[**BSST09**] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009

[**CBRZ01**] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001

[**GR10**] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010

[**HT08**] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAV'08), Portland, Oregon*, pages 109–117. IEEE, 2008

[**McM05**] K. McMillan. Applications of Craig interpolants in model checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Edinburgh, UK)*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005

References

[McM03] K. McMillan. Interpolation and SAT-based model checking. In *Proceedings of the 15th International Conference on Computer Aided Verification, (Boston, Massachusetts)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003

[dMRS03] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003

[Seb07] R. Sebastiani. Lazy satisability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, London, UK, 2000. Springer-Verlag