Abstraction and Refinement in Model Checking

Orna Grumberg
Technion, Haifa, Israel

FMCO, Amsterdam
November 2005
Model Checking

An efficient procedure that receives:
- A finite-state model describing a system
- A temporal logic formula describing a property

It returns
yes, if the system has the property
no + Counterexample, otherwise
Model of a system
Kripke structure / transition system
Model of systems

\( M = \langle S, I, R, L \rangle \)

- **S** - Finite set of states.
- **I \( \subseteq S \)** - Initial states.
- **R \( \subseteq S \times S \)** - Total transition relation.
- **L: S\( \rightarrow 2^{AP} \)** - Labeling function.

\( AP \) - Set of atomic propositions
Propositional temporal logic

In Negation Normal Form

AP - a set of atomic propositions

Temporal operators:

- $Gp$ - always true
- $Fp$ - eventually true
- $Xp$ - next proposition
- $pUq$ - proposition $p$ until proposition $q$

Path quantifiers: $A$ for all path $E$ there exists a path
CTL/CTL*

- **CTL** - Allows any combination of temporal operators and path quantifiers
- **CTL** - a useful sub-logic of **CTL**

**ACTL / ACTL**

The **universal** fragments of CTL/CTL* with only universal path quantifiers
Model Checking

- Emerging as an industrial standard tool for hardware design: Intel, IBM, Cadence, Motorola,…

- Recently applied successfully also for software verification: SPIN (Bell Labs), Java PathFinder (NASA), Zing (Microsoft), Bandera (Kansas University), Wolf (IBM, Haifa), TVLA (Tel Aviv University),…
Main limitation:

- **The state explosion problem:** Model checking is often efficient in time but suffers from high space requirements.

- The number of states in the model must be **finite**.
Solutions to the state explosion problem

• Symbolic model checking
  - BDD-based
  - SAT-based
• Abstraction
• Modular verification
• Partial order reduction
• Symmetry reduction
• Distributed model checking
• And more
Abstraction

• Removes or simplifies details
• Removes entire components that are irrelevant to the property under consideration

Can reduce the number of states
• from large to small
• from infinite to finite
• Manual abstraction requires great creativity

• **Goal:**
  Automatically construct an abstract model that will preserve the required property
Outline (for abstraction)

- Define an abstract model that preserves the checked property
- Consider different types of abstractions
- Automatically construct an abstract model
- Automatically refine it, if the abstraction is not detailed enough
• We first define an abstract model $M_h$ based on a concrete (full) model $M$ of the system

• Goal: constructing $M_h$ directly from the program text
Abstraction preserving ACTL/ACTL*

We use **Existential Abstraction** in which the abstract model is an **over-approximation** of the concrete model:

- The abstract model has **more behaviors**
- But no concrete behavior is lost

- Every ACTL/ACTL* property true in the abstract model is also true in the concrete model
Given an abstraction function $h : S \rightarrow S_h$, the concrete states are grouped and mapped into abstract states:

**Existential Abstraction**
How to define an abstract model:

Given $M$ and $\varphi$, choose

- $S_h$ - a set of abstract states
- $AP$ - a set of atomic propositions that label concrete and abstract states
- $h : S \rightarrow S_h$ - a mapping from $S$ on $S_h$ that satisfies:
  \[ h(s) = h(t) \text{ only if } L(s) = L(t) \]
The abstract model $M_h = (S_h, I_h, R_h, L_h)$

- $s_h \in I_h \iff \exists s \in I : h(s) = s_h$

- $(s_h, t_h) \in R_h \iff \exists s, t \ [h(s) = s_h \land h(t) = t_h \land (s, t) \in R]$

- $L_h(s_h) = L(s)$ for some $s$ where $h(s) = s_h$

This is an exact abstraction
An approximated abstraction
(an approximation)

• \( s_h \in I_h \iff \exists s \in I : h(s) = s_h \)

• \((s_h, t_h) \in R_h \iff \exists s, t \ [(h(s) = s_h \land h(t) = t_h \land (s, t) \in R)]\]

• \(L_h\) is as before

Notation:
\(M_r\) – reduced (exact)  \(M_h\) – approximated
Logic preservation

- $M_h$ (exact or approximated) has less states but more behaviors, therefore:

- **Theorem** If $\varphi$ is an ACTL/ACTL* specification over AP, then

$$M_h \models \varphi \Rightarrow M \models \varphi$$
Types of Abstraction \((S_h, h)\)

- Localization reduction: each variable either keeps its concrete behavior or is fully abstracted (has free behavior) [Kurshan94]
Types of Abstraction \((S_h, h)\)

- Localization reduction: each variable either keeps its concrete behavior or is fully abstracted (has free behavior) \([\text{Kurshan94}]\)

- Predicate abstraction: concrete states are grouped together according to the set of predicates they satisfy \([\text{GS97, SS99}]\)
Types of Abstraction \((S_h, h)\)

- **Localization reduction**: each variable either keeps its concrete behavior or is fully abstracted (has free behavior) \([\text{Kurshan94}]\)

- **Predicate abstraction**: concrete states are grouped together according to the set of predicates they satisfy \([\text{GS97, SS99}]\)

- **Data abstraction**: the domain of each variable is abstracted into a small abstract domain \([\text{CGL94, LONG94}]\)
Depending on $h$ and the size of $M$, $M_h$ (i.e. $I_h, R_h$) can be built using:

- BDDs or
- SAT solver or
- Theorem prover

We later demonstrate such constructions for specific types of abstractions.
Predicate abstraction

• Given a program over variables $V$
• **Predicate** $P_i$ is a first-order atomic formula over $V$
  Examples: $x+y < z^2$, $x=5$

• Choose: $AP = \{ P_1, \ldots, P_k \}$ that includes
  - the atomic formulas in the property $\phi$ and
  - conditions in *if*, *while* statements of the program
• Labeling of concrete states:

\[ L(s) = \{ P_i \mid s \models P_i \} \]
Abstract model

- Abstract states are defined over Boolean variables \{B_1,\ldots,B_k\}:
  \[ S_h \subseteq \{0,1\}^k \]

- \( h(s) = s_h \iff \forall 1 \leq j \leq k : [s |= P_j \iff s_h |= B_j] \)

- \( L_h(s_h) = \{ P_j \mid s_h |= B_j \} \)
Example

Program over natural variables $x$, $y$

$AP = \{ P_1, P_2, P_3 \}$ where

$P_1 = x \leq 1$, $P_2 = x > y$, $P_3 = y = 2$

$AP = \{ x \leq 1, x > y, y = 2 \}$

For state $s$ where $s(x) = s(y) = 0$:  
$L(s) = \{ P_1 \}$

For state $t$ where $t(x) = 1, t(y) = 2$:  
$L(t) = \{ P_1, P_3 \}$
\[ S_h \subseteq \{ 0,1 \}^3 \]

\[ h(s) = (1,0,0) \]

\[ h(\dagger) = (1,0,1) \]

\[ L_h((1,0,0)) = \{ P_1 \} \]

\[ L_h((1,0,1)) = \{ P_1, P_3 \} \]

The concrete state and its abstract state are labeled identically.
Computing $R_h$ (same example)

$$(s_h, t_h) \in R_h \iff \exists s, t \ [ h(s) = s_h \land h(t) = t_h \land (s, t) \in R ]$$

- Program with one statement: $x := x+1$
\[(b_1, b_2, b_3), (b_1', b_2', b_3') \in R_h \iff
\exists xyx'y' \left[ \begin{array}{l}
P_1(x, y) \iff b_1 \land \\
P_2(x, y) \iff b_2 \land \\
P_3(x, y) \iff b_3 \land \\
x' = x + 1 \land y' = y \land \\
P_1(x', y') \iff b_1' \land \\
P_2(x', y') \iff b_2' \land \\
P_3(x', y') \iff b_3' \end{array} \right] \]
Computing $R_h$ using BDDs

If the program is over finite, relatively small domains, use BDDs

We need BDDs for:
- the concrete transition relation $R$ (possibly partitioned)
- the abstraction mapping $h$

$R_h$ is computed according to the formula above, using BDD operations
Computing $R_h$ using SAT or Theorem Prover

- If the program is over finite, relatively large domains, use SAT

- If the program is over infinite domains, use theorem prover

- Theorem prover and SAT solve the problem for each pair of states separately: $|S_h| \times |S_h|$ applications
Data Abstraction

- Given a program over variables $v_1, \ldots, v_n$
  where $v_i$ is over a domain $D_i$
- Program states (concrete) $S = D_1 \times \ldots \times D_n$

Choose for each $v_i$:
- an abstract domain $A_i$
- $h_i : D_i \rightarrow A_i$

$S_h = A_1 \times \ldots \times A_n$
$\mathsf{AP} = \{ \mathsf{v}_i^A = a \mid a \in A_i , \ i=1,\ldots,n \}$

where $\mathsf{v}_i^A$ refers to the abstract value of $\mathsf{v}_i$

- $h : \mathcal{S} \rightarrow \mathcal{S}_h$ is defined:
  
  $h((d_1,\ldots,d_n)) = (a_1,\ldots,a_n)$ where $a_i = h(d_i)$

- $L((d_1,\ldots,d_n)) = \{ \mathsf{v}_i^A = a_i \mid h(d_i) = a_i , \ i=1,\ldots,n \}$

- $L_h((a_1,\ldots,a_n)) = \{ \mathsf{v}_i^A = a_i \mid i=1,\ldots,n \}$

As before: $L(s) = L_h(h(s))$
Constructing $M_h$ from program text

we assume that the program is given by first order formulas

- $I(V)$ - describing the initial states
- $R(V, V')$ - describing the transition relation

where

$V = (v_1, \ldots, v_n)$ and $V' = (v_1', \ldots, v_n')$
Representing a program by formulas:

example

program:
k: if x=0 then k₁: t:=x; x:=y; y:=t
else k₂: x:=x-1  k'

Formula R (x,y,t, pc, x’,y’,t’, pc’):
( pc=k ∧ x=0 ∧ x’=x ∧ pc’ = k₁ ) ∨
( pc=k ∧ x \neq 0 ∧ x’=x ∧ pc’ = k₂ ) ∨
( pc=k₁ ∧ t’=x ∧ x’=y ∧ y’=t ∧ pc’=k’ ) ∨
( pc=k₂ ∧ x’=x-1 ∧ pc’=k’ )
Notation:
For a formula $\Phi$ over variables $V=(v_1, \ldots, v_n)$ and predicates $P_1, \ldots, P_k$:

$$[\Phi] (B_1, \ldots, B_k) = \exists v_1, \ldots, v_n ( P_1(V) \iff B_1 \land \ldots \land P_k(V) \iff B_k \land \Phi(v_1, \ldots, v_n) )$$
Notation:
For a formula $\Phi$ over variables $V=(v_1, \ldots, v_n)$ and predicates $P_1, \ldots, P_k$:

$[\Phi] (B_1, \ldots, B_k) =$

$\exists v_1, \ldots, v_n (P_1(V) \iff B_1 \land \ldots \land P_k(V) \iff B_k \land \Phi(v_1, \ldots, v_n))$

Note: $[I(V)]$ and $[R(V, V')]$ are formulas over abstract variables.
Notation:
For a formula $\Phi$ over variables $V=(v_1, \ldots, v_n)$ and predicates $P_1, \ldots, P_k$:

$$[\Phi] (B_1, \ldots, B_k) = \exists v_1, \ldots, v_n \left( P_1(V) \iff B_1 \land \ldots \land P_k(V) \iff B_k \land \Phi(v_1, \ldots, v_n) \right)$$

Note: $[I(V)]$ and $[R(V, V')]$ are formulas over abstract variables

$[I(V)]$ and $[R(V, V')]$ represent $M_r$. 
Given first order formulas $I(V)$ and $R(V, V')$, representing the “program text”, we can construct $[I(V)]$ and $[R(V, V')]$, representing $M_r$. 
Problem:

Given \([I(V)]\) and \([R(V, V')]\), in order to determine if \(s_h \in I_r\), we need to find a state \(s \in I\) (a satisfying assignment for \(I(V)\)) so that \(h(s) = s_r\).

Similarly, for \((s_h, t_h) \in R_r\) we look for a satisfying assignment for \(R(V, V')\).

This is a difficult task due to the size of the two formulas.
Simplifying the formulas

For $\Phi$ in negation normal form over atomic formulas $a_i$ and $\neg a_i$, $T(\Phi)$ simplifies $[\Phi]$ by “pushing” the existential quantifiers inward:

$$T(a_i(v_1,\ldots,v_n)) = [a_i](B_1, \ldots, B_k)$$
$$T(\neg a_i(v_1,\ldots,v_n)) = [-a_i](B_1, \ldots, B_k)$$
$$T(\Phi_1 \land \Phi_2) = T(\Phi_1) \land T(\Phi_2) \quad (*)$$
$$T(\Phi_1 \lor \Phi_2) = T(\Phi_1) \lor T(\Phi_2)$$

(*) does not preserve equivalence
Approximated model

**Theorem:**

\[ \Phi \] \Rightarrow T(\Phi)

In particular, \[ I \] \Rightarrow T(I) and \[ R \] \Rightarrow T(R)

**Corollary:**

The approximation model \( M_h \), defined by \( T(I) \) and \( T(R) \) satisfies:

\( M_h \geq M_r \geq M \) by the simulation preorder
Approximated model (cont.)

- Defined over the same set of abstract states as $M_r$
- Easier to compute since existential quantifiers are applied to simpler formulas
- Less precise: has more initial states and more transitions than $M_r$
Computing approximated model from the text

• No need to construct formulas. The approximated model can be constructed directly from the program text
Logic preservation Theorem (a reminder)

- **Theorem** If $\phi$ is an ACTL/ACTL* specification over AP, then
  \[ M_h \models \phi \Rightarrow M \models \phi \]
- However, the reverse may not be valid.
Traffic Light Example

Property:
\( \varphi = \text{AG AF} \neg (\text{state}=\text{red}) \)

Abstraction function \( h \) maps green, yellow to go.

\[ M \models \varphi \iff M_h \models \varphi \]
Traffic Light Example (Cont)

If the abstract model invalidates a specification, the actual model may still satisfy the specification.

- **Property:**
  \[ \varphi = \text{AG AF (state=red)} \]

- \( M \models \varphi \) but \( M_h \not\models \varphi \)

- **Spurious Counterexample:**
  \[ \langle \text{red,go,go, ...} \rangle \]
CEGAR: The CounterExample-Guided Abstraction-Refinement methodology
The CEGAR Methodology

1. **Model Check**
   - \( M_h \models \varphi \)

2. **Generate Initial Abstraction**
   - \( M_h \)

3. **Refinement**
   - \( T_h \)

4. **Generate Counterexample**
   - \( T_h \)

5. **Check Spurious Counterexample**
   - \( T_h \) is spurious

6. **Stop**
   - \( M_h \) is not spurious

7. **Spurious Check**
   - The process is not spurious
Refinement versus approximation

Remark:

**Refinement** builds new \( S_h \) and \( h \), then recomputes \( R_h \) and \( I_h \) (exact or approximated)

**Approximation** uses the same \( S_h \) and \( h \) and just adds more transitions and initial states
Generating the Initial Abstraction

- If we use predicate abstraction then predicates are extracted from the program’s control flow and the checked property.

- If we use localization reduction then the unabstractual variables are those appearing in the predicates above.
Model Check The Abstract Model

Given the abstract model $M_h$

- If $M_h \not \models \phi$, then the model checker generates a counterexample trace ($T_h$)
- Most current model checkers generate paths or loops
- Question: is $T_h$ spurious?
Path Counterexample

Assume that we have four abstract states
\{1,2,3\} \leftrightarrow \alpha \quad \{4,5,6\} \leftrightarrow \beta \\
\{7,8,9\} \leftrightarrow \gamma \quad \{10,11,12\} \leftrightarrow \delta \\

Abstract counterexample \( T_h = \langle \alpha, \beta, \gamma, \delta \rangle \)

\( T_h \) is not spurious, therefore, \( M \models \varphi \)
Spurious Path Counterexample

The concrete states mapped to the failure state are partitioned into 3 sets.

<table>
<thead>
<tr>
<th>states</th>
<th>dead-end</th>
<th>bad</th>
<th>irrelevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>reachable</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>out edges</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

$T_h$ is spurious
Refining The Abstraction

- **Goal**: refine $h$ so that the dead-end states and bad states do not belong to the same abstract state.

- For this example, two possible solutions.
Computing a Concrete Counterexample from an Abstract one

Let $T = (a_1, \ldots, a_n)$ be a path abstract counterexample

$h^{-1}(a) = \{ s \mid h(s) = a \}$

The set of concrete counterexamples corresponding to $T$:

$h^{-1}(T) = \{ (s_1, \ldots, s_n) \mid \bigwedge_i h(s_i) = a_i \land I(s_1) \land \bigwedge_i R(s_i, s_{i+1}) \}$
BDD-based computation of $h^{-1}(a_1), \ldots, h^{-1}(a_n)$

$S_1 = h^{-1}(a_1) \cap I$

For $i = 2, \ldots, n$ do

$S_i = \text{Image}(S_{i-1}) \cap h^{-1}(a_i)$

if $S_i = \emptyset$ then

$\text{dead-end} := S_{i-1}$

return $(i-1, \text{dead-end})$

print ("counterexample exists")

Return $(S_1, \ldots, S_n)$
Computing a concrete counterexample from $S_1, \ldots, S_n$

\[ t_n = \text{choose } (S_n) \]

For $i = n-1$ to 1
\[ t_i = \text{choose } (\text{predecessors}(t_{i+1}) \cap S_i) \]

Return \((t_1, \ldots, t_n)\)
Remark:

• If $h(s_i) = a_i$ then for every atomic formula $f$, $s_i \models f \iff a \models f$

• In particular, if $a_n \models \neg p$ then $s_n \models \neg p$

• Therefore, if $(a_1, \ldots, a_n)$ is an abstract counterexample then $(s_1, \ldots, s_n)$ is a concrete counterexample
Refining the abstraction

- Refinement separates dead-end states from bad states, thus, eliminating the spurious transition from $S_{i-1}$ to $S_i$
Implementing CEGAR

With BDDs:
• The concrete model \( M \) is finite but too big to directly apply model checking on
• \( R \) and \( I \) can be held as BDDs in memory, \( R \) possibly partitioned:
  \[ R(V, V') = \bigwedge_i R_i (V, v_i') \]
• \( h \) is held as BDD over concrete and abstract states

Can also be implemented with SAT or Theorem Prover
Conclusion

• We defined several types of abstractions

• We showed how to extract them from the concrete model or from the program text

• We presented the CEGAR framework for abstraction-refinement

• We considered several possible implementations
More on abstraction

- Abstract models (KMTSs) with may and must transitions that preserve full CTL
- 3-valued and even multi-valued abstraction
- Static analysis for computing abstract models directly from program text
- Abstract interpretation:
  
  Instead of \( h \):
  
  \[ \alpha : 2^S \rightarrow S_h \]
  
  \[ \gamma : S_h \rightarrow 2^S \]
THE END
• **BDDs:**

• **BDD-based model checking:**

• **SAT-based Bounded model checking:**
  Symbolic model checking using SAT procedures instead of BDDs, A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, DAC'99
• **Existential abstraction + data abstraction:**

• **Localization reduction:**
• **Predicate abstraction:**
  S. Graf and H. Saidi, *Construction of abstract state graphs with PVS*, CAV'97


• **BDD-based CEGAR:**