Abstractions and Decision Procedures for Effective Software Model Checking

Prof. Natasha Sharygina
The University of Lugano,
Carnegie Mellon University

Microsoft Summer School, Moscow, July 2011
Lecture 1
Traditional Approaches

• **Testing**: Run the system on select inputs

• **Simulation**: Simulate a model of the system on select (often symbolic) inputs

• Code *review* and *auditing*
What are the Problems?

• not exhaustive (missed behaviors)

• not all are automated (manual reviews, manual testing)

• do not scale (large programs are hard to handle)

• no guarantee of results (no mathematical proofs)

• concurrency problems (non-determinism)
What is Formal Verification?

• Build a **mathematical model** of the system:
  – what are possible behaviors?

• Write **correctness requirement** in a specification language:
  – what are desirable behaviors?

• Analysis: (Automatically) **check** that model satisfies specification
What is Formal Verification (2)?

- **Formal** - Correctness claim is a precise mathematical statement

- **Verification** - Analysis either proves or disproves the correctness claim
Algorithmic Analysis by Model Checking

• Analysis is performed by an algorithm (tool)

• Analysis gives counterexamples for debugging

• Typically requires exhaustive search of state-space

• Limited by high computational complexity
Temporal Logic Model Checking

[Clarke, Emerson 81][Queille, Sifakis 82]

$M \models P$

“implementation” (system model) \iff “specification” (system property)

“satisfies”, “implements”, “refines” (satisfaction relation)
Temporal Logic Model Checking

more detailed  \[ M \models P \]  more abstract

“implementation” (system model)  \[ \text{"satisfies"}, \text{"implements"}, \text{"refines"}, \text{"confirms"}, \text{(satisfaction relation)} \]

“specification” (system property)
Temporal Logic Model Checking

\[ M \models P \]

- **system model**
- **system specification**
- **satisfaction relation**
Decisions when choosing a system model:

- variable-based vs. event-based
- interleaving vs. true concurrency
- synchronous vs. asynchronous interaction
- clocked vs. speed-independent progress
- etc.
Characteristics of system models

which favor model checking over other verification techniques:

- ongoing input/output behavior
  (not: single input, single result)

- concurrency
  (not: single control flow)

- control intensive
  (not: lots of data manipulation)
Decisions when choosing a system model:

While the choice of system model is important for ease of modeling in a given situation, the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.
Finite State Machine (FSM)

- Specify *state-transition* behavior
- Transitions depict *observable* behavior

Acceptable sequences of acquiring and releasing a lock
High-level View

Linux Kernel (C)

Conformance Check

Spec (FSM)
High-level View

Linux Kernel (C)

Finite State Model (FSM)

Model Checking

Spec (FSM)

By Construction
Low-level View

State-transition graph

\[ S \quad \text{set of states} \]
\[ I \quad \text{set of initial states} \]
\[ AP \quad \text{set of atomic observation} \]
\[ R \subseteq S \times S \quad \text{transition relation} \]
\[ L: S \rightarrow 2^{AP} \quad \text{observation (labeling) function} \]
Run: \( s_1 \rightarrow s_3 \rightarrow s_1 \rightarrow s_3 \rightarrow s_1 \rightarrow \) state sequence

Trace: \( a \rightarrow b \rightarrow a \rightarrow b \rightarrow a \rightarrow \) observation sequence
Model of Computation

State Transition Graph

Infinite Computation Tree

Unwind State Graph to obtain Infinite Tree.

A trace is an infinite sequence of state observations.
The semantics of a FSM is a set of traces.
Where is the model?

- Need to extract automatically
- Easier to construct from hardware
- Fundamental challenge for software

Linux Kernel
- ~1,000,000 LOC
- Recursion and data structures
- Pointers and Dynamic memory
- Processes and threads

Finite State Model
Mutual-exclusion protocol

\[
\begin{align*}
\text{loop} & \quad \text{||} \quad \text{loop} \\
\text{out:} & \quad x_1 := 1; \text{last} := 1 \\
\text{req:} & \quad \text{await } x_2 = 0 \text{ or last} = 2 \\
\text{in:} & \quad x_1 := 0 \\
\text{end loop.} & \quad \text{||} \quad \text{out:} & \quad x_2 := 1; \text{last} := 2 \\
\text{req:} & \quad \text{await } x_1 = 0 \text{ or last} = 1 \\
\text{in:} & \quad x_2 := 0 \\
\text{end loop.}
\end{align*}
\]
\[ 3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 = 72 \text{ states} \]
The translation from a system description to a state-transition graph usually involves an exponential blow-up!!!

\[ \text{e.g., } n \text{ boolean variables } \Rightarrow 2^n \text{ states} \]

This is called the “state-explosion problem.”
Temporal Logic Model Checking

$M \models P$

system model

system specification

satisfaction relation
Decisions when choosing system properties:

- operational vs. declarative: automata vs. logic
- may vs. must: branching vs. linear time
- prohibiting bad vs. desiring good behavior: safety vs. liveness
System Properties/Specifications

- Atomic propositions: properties of states
- (Linear) Temporal Logic Specifications: properties of traces.
Specification (Property)

**Examples:**

- **Safety** (mutual exclusion): no two processes can be at the critical section at the same time

- **Liveness** (absence of starvation): every request will be eventually granted

Linear Time Logic (LTL) [Pnueli 77]: logic of temporal sequences.

- **next** ($\alpha$): $\alpha$ holds in the next state
- **eventually**($\gamma$): $\gamma$ holds eventually
- **always**($\lambda$): $\lambda$ holds from now on
- **$\alpha$ until $\beta$**: $\alpha$ holds until $\beta$ holds
Operational Software Components

Real-Time Control Components

Performance

Kinematics

Criteria

Compliance

Dynamics

Operational Software Components

Resource Allocation

To Simulation

Operator Priority Setting

Actuator Control
Examples of the Robot Control Properties

- **Configuration Validity Check:** If an instance of *EndEffector* is in the “FollowingDesiredTrajectory” state, then the instance of the corresponding *Arm* class is in the ‘Valid’ state.

  \[
  \text{Always}((\text{ee\_reference}=1) \implies (\text{arm\_status}=1))
  \]

- **Control Termination:** Eventually the robot control terminates.

  \[
  \text{Eventually}(\text{abort\_var}=1)
  \]
What is “satisfy”?

*M* satisfies *S* if *all* the reachable states satisfy *P*

Different Algorithms to check if *M* |= *P*.

- Explicit State Space Exploration

For example: Invariant checking Algorithm.

1. Start at the initial states and explore the states of *M* using DFS or BFS.
2. In any state, if *P* is violated then print an “error trace”.
3. If all reachable states have been visited then say “yes”.
State Space Explosion

**Problem:** Size of the state graph can be exponential in size of the program (both in the number of the program *variables* and the number of program *components*)

\[
M = M_1 \| \ldots \| M_n
\]

If each \(M_i\) has just 2 local states, potentially \(2^n\) global states

**Research Directions:** State space reduction
Abstractions

• They are one of the most useful ways to fight the state explosion problem

• They should preserve properties of interest: properties that hold for the abstract model should hold for the concrete model

• Abstractions should be constructed directly from the program
Abstractions

• Why do we need to abstract?
  – To reduce a number of states
  – To represent (in a sound manner) infinite state systems as finite state systems
Abstractions

• Why we need to abstract?
  – To reduce a number of states
  – To represent (in a sound manner) infinite state systems as finite state systems

• How do we abstract?
  – By removing irrelevant to verification details
Data Abstraction

Given a program $P$ with variables $x_1,...,x_n$, each over domain $D$, the **concrete model** of $P$ is defined over states $(d_1,...,d_n) \in D \times ... \times D$

**Choosing**

- Abstract domain $A$
- Abstraction mapping (surjection) $h: D \rightarrow A$

we get an **abstract model** over abstract states $(a_1,...,a_n) \in A \times ... \times A$
Example

Given a program $P$ with variable $x$ over the integers

Abstraction 1:

$A_1 = \{ a_{-}, a_{0}, a_{+} \}$

\[
    h_1(d) = \begin{cases} 
    a_{+} & \text{if } d > 0 \\
    a_{0} & \text{if } d = 0 \\
    a_{-} & \text{if } d < 0 
    \end{cases}
\]

Abstraction 2:

$A_2 = \{ a_{\text{even}}, a_{\text{odd}} \}$

\[
    h_2(d) = \text{if even}( |d| ) \text{ then } a_{\text{even}} \text{ else } a_{\text{odd}}
\]
Existential Abstraction
Existential Abstraction

\[ M \]

\[ A \]
Existential Abstraction

- **Every** trace of $M$ is a trace of $A$
  - $A$ over-approximates what $M$ can do
    (Preserves safety properties!): $A$ satisfies $\phi \Rightarrow M$ satisfies $\phi$

- **Some** traces of $A$ may not be traces of $M$
  - May yield spurious counterexamples - $<a, e>$

- **Eliminated** via abstraction refinement
  - Splitting some clusters in smaller ones
  - Refinement can be automated
Original Abstraction

\[ M \]

\[ A \]
Refined Abstraction

\[ M \]

\[ A \]

\[ \begin{align*}
1 & \rightarrow a & 2 & \rightarrow c & 4 & \rightarrow 5 \\
1 & \rightarrow b & 3 & \rightarrow d & 6 & \rightarrow 7 \\
4 & \rightarrow 5 & 6 & \rightarrow 7
\end{align*} \]

\[ \begin{align*}
\end{align*} \]
How to define an abstract model

Given $M$ (model) and $\phi$ (spec), 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:
  
  $h(s) = h(t)$ only if $L(s) = L(t)$
Abstraction

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
Predicate Abstraction

[Graf/Saïdi 97]

- Idea: Only keep track of predicates on data

\[ p_1(s), \ldots, p_n(s) \]

- Abstraction function:

\[ h(s) = (p_1(s), p_2(s), \ldots, p_n(s)) \]
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
Predicate Abstraction

Labeling of concrete states:

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

- Abstract states are defined over Boolean variables \( \{ B_1, \ldots, B_k \} \):

\[ S_h \subseteq \{0,1 \}^k \]

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

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

Program over natural variables $x, y$

$AP = \{ P1, P2, P3 \}$, where

$P1 = x \leq 1$, $P2 = x > y$, $P3 = y = 2$

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

For state $s$, where $s(x) = s(y) = 0$: $L(s) = \{ P1 \}$
For state $t$, where $t(x) = 1, t(y) = 2$: $L(t) = \{ P1, P3 \}$
Example

\[ S_h \subseteq \{ 0,1 \}^3 \]

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

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

The concrete state and its abstract state are labeled identically
Computing abstract transition relation

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

- Program with one statement: \( x := x+1 \)

\[
( (b_1, b_2, b_3), (b'_1, b'_2, b'_3) ) \in R_h \iff
\exists xyx'y' \ [ \ P_1(x, y) \iff b_1 \land \\
 P_2(x, y) \iff b_2 \land \\
 P_3(x, y) \iff b_3 \land \\
 x' = x+1 \land y' = y \land \\
 P_1(x', y') \iff b'_1 \land \\
 P_2(x', y') \iff b'_2 \land \\
 P_3(x', y') \iff b'_3 \ ]
\]
Example

Concrete States:

Predicates:

\[ p_1(s) = (s.x > s.y) \]
\[ p_2(s) = (s.y = 0) \]
How to get abstract transitions?

• Typically done in a conservative manner
• Existential abstraction:

\[ \hat{I}(\hat{s}) \iff \exists s : I(s) \]
\[ \hat{R}(\hat{s}, \hat{s}') \iff \exists s, s' : R(s, s') \]
\[ \land h(s) = \hat{s} \land h(s') = \hat{s}' \]
Predicate Abstraction

Abstract Transitions:

$p_1, p_2$

$p_1, \neg p_2$

$\neg p_1, p_2$

$x=0$
$y=0$

$x=1$
$y=0$

$x=1$
$y=1$

$x=1$
$y=2$

$x=2$
$y=0$

$x=2$
$y=1$

$\neg p_1, \neg p_2$

Property:

$p_1 \lor \neg p_2 \iff (s.x > s.y) \lor (s.y \neq 0)$

Property holds. Ok.
Abstract Transitions:

Predicate Abstraction

Abstract Transitions:

Property:

\[ p_1 \iff (s.x > s.y) \]

This trace is spurious!
New Predicates:

\[ p_1 \iff (s.x > s.y) \]

\[ p_1(s) = (s.x > s.y) \]

\[ p_2(s) = (s.x = 2) \]
CEGAR

Counter Example Guided Abstraction Refinement
CEGAR approach
CEGAR

Original Model

Refinement

Validation or Counterexample

Spurious Counterexample

M

M_\alpha

Initial Abstraction

Correct!

Refinement

Original Model