# Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

*Guest Lectures at the Analysis of Software Artifacts Class, Spring 2005*



## Outline

*Lecture 1:* Overview of Model Checking

***Lecture 2:*** Complexity Reduction Techniques

*Lecture 3:* Software Verification

*Lecture 4:* State/Event-based software model checking

*Lecture 5:* Component Substitutability

*Lecture 6:* Model Checking Practicum (Student Reports on the Lab exercises)





## Today's Lecture

– Model checking basics (review from the last class)

– Problems with model checking

– Combating state space explosion





## Temporal Logic Model Checking

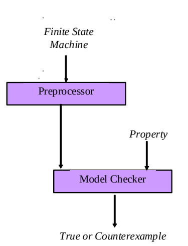
[Clarke,Emerson 81][Queille,Sifakis 82]

- Systems are modeled by finite state machines
- Properties are written in propositional temporal logic
- Verification procedure is an exhaustive search of the state space of the design
- Diagnostic counterexamples





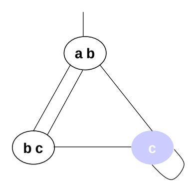
## Temporal Logic Model Checking



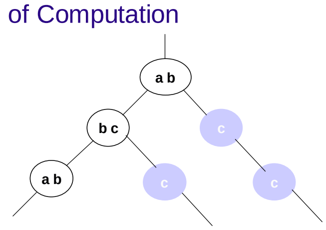





## Model of Computation




**State Transition Graph**




**Infinite Computation Tree**

Unwind State Graph to obtain Infinite Tree.

*A trace* is an infinite sequence of states.

## What is “M”?

$M = \langle W, I, R, L, \Gamma \rangle$

W - set of states
$I \subseteq W$ – set of initial states
$R \subseteq W \times W$ - set of arcs
L - set of atomic propositions
$\Gamma : W \rightarrow 2^L$ – mapping from states to colors





## Computation Tree Logics

***Examples:*** ***Safety*** (mutual exclusion): no two processes can be at a critical section at the same time
***Liveness*** (absence of starvation): every request will be eventually granted

In a linear temporal logic (LTL), operators are provided for describing system behavior along a single computation path.

In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state.





## 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*)

$M = M_1 \mid\mid \ldots \mid\mid M_n$

If each $M_i$ has just 2 local states, potentially $2^n$ global states

*Research Directions:* State space reduction





## Principal Approaches to State Space Reduction

Compositional reasoning (reasoning about parts of the system)

Abstraction (elimination of details irrelevant to verification of a property)

Symbolic Verification (BDDs represent state transition diagrams more efficiently)

Partial Order Reduction (reduction of number of states that must be enumerated)

Domain specific reductions (syntactic program transformations)

Other (symmetry, cone of influence reduction, ….)





## Divide and Conquer

Compositional reasoning reduces reasoning about entire system to reasoning about individual parts

Decompose the model: M = M1 || M2

Partition global properties into local properties: $P = P1 \wedge P2$

Show that M1 |= P1 and M2 |= P2

Limitation: It does not work in practice for properties that are constrained by behavior of the environment





## Example

**Process M1**
**var x: Boolean;**
**init x = 0;**
**x := y;**

x
y

**Process M2**
**var y: Boolean;**
**init y = 0;**
**y := 1;**

**Property P1: Always(x);** **Property P2: Always(y);**

M2 |= P2 can be checked but M1 |= P1 can not be checked since *y* is not constrained

## Assume-Guarantee Compositional Reasoning

In the assume-guarantee paradigm: each component guarantees properties based on assumptions about other components via proof rules

Constraining environment

**M1 || P2 |= P1**
**M2 || P1 |= P2**
———————————
**M1 || M2 |= P1 || P2***

*Circularity during verification of different blocks is broken by induction over time

[AbadiLamport95][AlurHenzinger96][HenzingerQadeerRajamani99][Kurshan9][McMillan98][Stark85]





## Scalability of Multi-Process Model Checking

***Limitation:*** Properties do not give details about the verifiable processes

***Approach:*** Add abstraction constraints

*Abstraction constraints* (cf. Henzinger 98) are added to the property specifications:

***M1*** |= ***P2 || P1**$^{abs}$* **|| …|| *Pn**$^{abs}$*

P1$^{abs}$ , …, Pn$^{abs}$ – abstraction constraints

Abstraction constraints specification:
- temporal logic formulae constraining the external variables of the verifiable processes





## Abstractions in Model Checking

Data Abstraction (abstraction of details irrelevant to verification of a property)

**Approach:** $M_{abs} \models P \Rightarrow M_{conc} \models P$
**To Prove:** **soundness and completeness**

Systematic Construction of Abstractions (Predicate Abstraction) [Saidi,Graf 97]

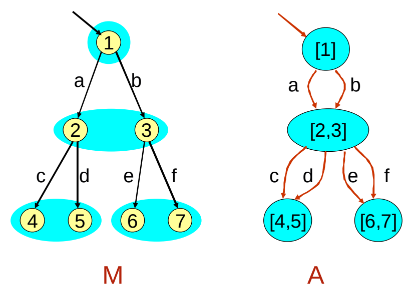
- Define an abstraction function as a predicate over concrete data
- Specify decision procedures to compute a set of abstraction predicates
- Demonstrate the soundness and completeness of the abstraction

Refinement–based abstraction [MSR SLAM Project],[Clarke et. al. '00], [Saidi 00], [Visser et. al 00]





## Conservative Abstraction







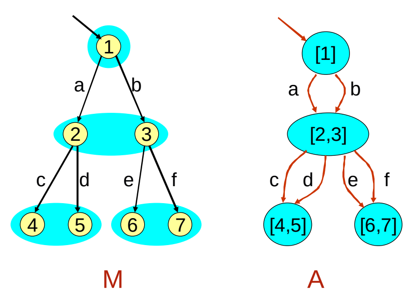

## Conservative Abstraction

- Every trace of M is a trace of A
  - A over-approximates what M can do (Preserves safety properties!): A ⊨ ϕ ⇒ M ⊨ ϕ
- 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

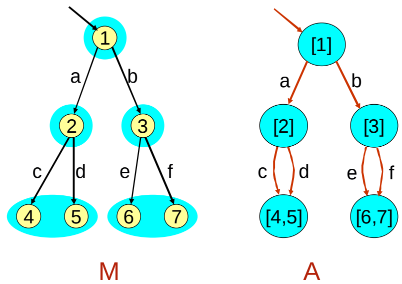




## Original Abstraction



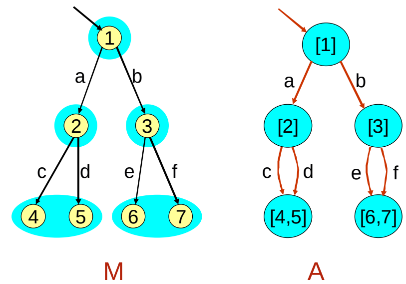
## Refined Abstraction








## Abstractions in Model Checking: Limitations








## Predicate Abstraction

[Graf/Saïdi 97]

- Idea: Only keep track of predicates on data

$$p_1(s), \ldots, p_n(s)$$

- Abstraction function:

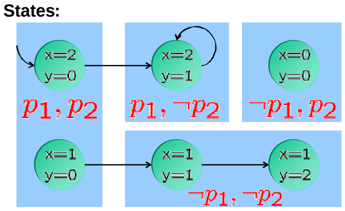
$$h(s) \;=\; (p_1(s), p_2(s), \ldots, p_n(s))$$





## Predicate Abstraction

**Concrete States:**




**Predicates:**

$p_1(s) = (s.x > s.y)$

$p_2(s) = (s.y = 0)$

**Abstract transitions?**





## Under- vs. Overapproximation

- How to abstract the transitions?
  - Depends on the property we want to show
  - Typically done in a conservative manner

$$\hat{I}(\hat{s}) \;:\Longleftrightarrow\; \exists s : I(s)$$

$$\hat{R}(\hat{s}, \hat{s}') \;:\Longleftrightarrow\; \exists s, s' : R(s, s') \wedge h(s) = \hat{s} \wedge h(s') = \hat{s}'$$

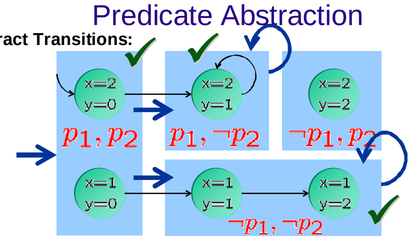
$\Rightarrow$ Preserves safety properties





## Predicate Abstraction

**Abstract Transitions:**



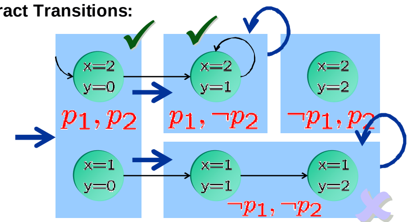

**Property:**

$$p_1 \vee \neg p_2 \quad \Longleftrightarrow \quad (s.x > s.y) \vee (s.y \neq 0)$$

**Property holds. Ok.**

## Predicate Abstraction

**Abstract Transitions:**




**Property:**

$p_1 \iff (s.x > s.y)$

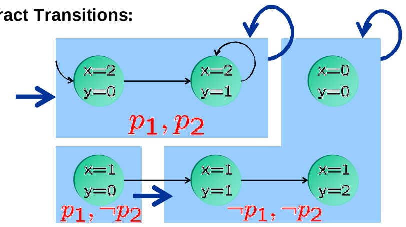
**This trace is spurious!**





## Predicate Abstraction

**Abstract Transitions:**




**Property:**

$p_1 \iff (s.x > s.y)$ ✓

**New Predicates:**

$p_1(s) = (s.x > s.y)$
$p_2(s) = (s.x = 2)$





## Predicate Abstraction for Software

- Let's take existential abstraction seriously
- Basic idea: with n predicates, there are $2^n$ x $2^n$ possible abstract transitions
- Let's just check them!





## Existential Abstraction

Predicates

$p_1 \iff i = 1$
$p_2 \iff i = 2$
$p_3 \iff \text{even}(i)$

Basic Block

```
i++;
```

Formula

$i' = i + 1$

Current Abstract State

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Next Abstract State

| $p'_1$ | $p'_2$ | $p'_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Query

$i \neq 1 \wedge i \neq 2 \wedge \text{even}(i) \wedge$
$i' = i + 1 \wedge$
$i' \neq 1 \wedge i' \neq 2 \wedge \text{even}(i')$





## Existential Abstraction

Predicates

$p_1 \iff i = 1$
$p_2 \iff i = 2$
$p_3 \iff \text{even}(i)$

Basic Block

```
i++;
```

Formula

$i' = i + 1$

Current Abstract State

| $p_1$ | $p_2$ | $p_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Next Abstract State

| $p'_1$ | $p'_2$ | $p'_3$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Query

$i \neq 1 \wedge i \neq 2 \wedge \text{even}(i) \wedge$
$i' = i + 1 \wedge$
$i' \neq 1 \wedge i' \neq 2 \wedge \text{even}(i')$

… and so on …





## Predicate Abstraction for Software

- A precise existential abstraction can be way too slow
- Use an over-approximation instead
  - Fast to compute
  - But has additional transitions
- E.g.:
  - SLAM (FastAbs) - Microsoft
  - Predicate partitioning

Carnegie Mellon

## Example for Predicate Abstraction

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

\+ $p_1 \Leftrightarrow i=0$, $p_2 \Leftrightarrow even(i)$ =

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2)
  {
    p1=p1?FALSE:nondet();
    p2=!p2;
  }
}
```

**C program** **Predicates** **Boolean program**

[Graf, Saidi '97]
[Ball, Rajamani '00]

31

Carnegie Mellon

## Fast Syntactic Abstraction

- Problem: Computing precise existential abstraction is exponential
- Even moderate-sized software needs >500 predicates
- ➢ Won't scale
- Syntactic abstraction is polynomial
  - ➢ Will scale!
- But: May result in additional spurious behavior and more non-determinism

32

Carnegie Mellon

## Predicate Abstraction for Software

- How do we get the predicates?
- Automatic abstraction refinement!

[Clarke *et al.* '00]

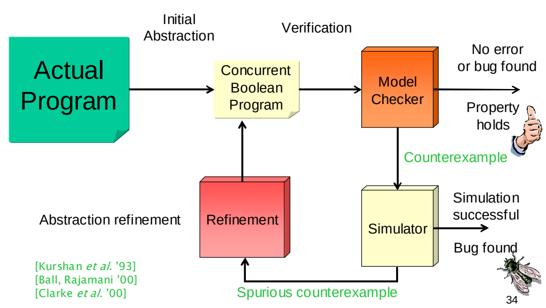
[Kurshan *et al.* '93]

[Ball, Rajamani '00]

33

Carnegie Mellon

## Abstraction Refinement Loop




34

Carnegie Mellon

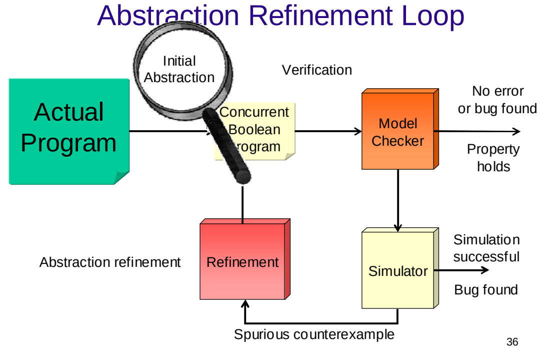
## SLAM

- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- To be shipped with Device Driver Development Kit
- Full detail (and all the slides) available at http://research.microsoft.com/slam/

35

Carnegie Mellon

## Abstraction Refinement Loop




36

## Problem with Existing Tools

- Existing tools (BLAST, SLAM, MAGIC) use a Theorem Prover like Simplify
- Theorem prover works on real or natural numbers, but C uses bit-vectors ➔ false positives
- Most theorem provers support only few operators (+, -, <, ≤, …), no bitwise operators
- Idea: Use SAT solver to do bit-vector!





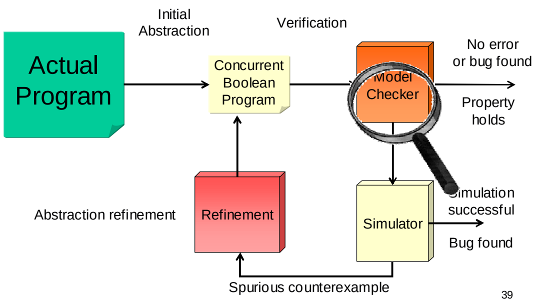
## Abstraction with SAT

- Successfully used for abstraction of C programs
  (Clarke, Kroening, Sharygina, Yorav '03 – SAT-base predicate abstraction)
- There is now a version of SLAM that has it
  - Found previously unknown Windows bug
- Create a SAT instance which relates initial value of predicates, basic block, and the values of predicates after the execution of basic block
- SAT also used for simulation and refinement





## Abstraction Refinement Loop







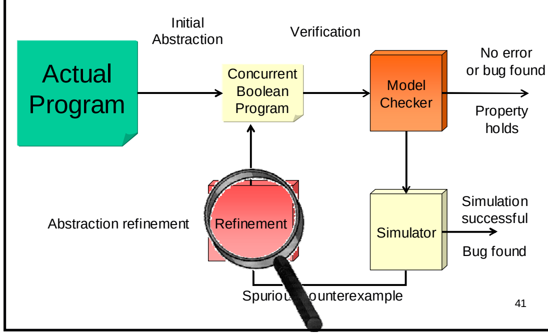

## Model Checkers for Boolean Programs

- Explicit State
  - Zing
  - SPIN
- Symbolic
  - Moped
  - Bebop
  - SMV





## Abstraction Refinement Loop








## Refinement

- Need to distinguish two sources of spurious behavior
  1. Too few predicates
  2. Laziness during abstraction
- SLAM:
  - First tries to find new predicates (NEWTON) using weakest preconditions
  - If this fails, second case is assumed.
    Transitions are refined (CONSTRAIN)
- Refine transitions using UNSAT cores (Clarke, Kroening, Sharygina, Yorav'05)

## Experimental Results

| Model checking result | SLAM/Zapato | SLAM/SAT |
| --- | --- | --- |
| Property passes | 243 | 264 |
| Time threshold exceeded | 39 | 17 |
| Property violations found | 17 | 19 |
| Cases of abstraction-refinement failure | 9 | 8 |

- Comparison of SLAM with Integer-based theorem prover against SAT-based SLAM
- 308 device drivers
- Timeout: 1200s





## Questions?