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 Model Checking

Lecture 4: State/Event-based software model checking

Lecture 5: Deadlock Detection and Component Substitutability

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

Actual Goal

- Deadlock for concurrent blocking message-passing C programs
- Tackle complexity using automated abstraction and compositional reasoning
- Obtain precise answers using automated iterative abstraction refinement

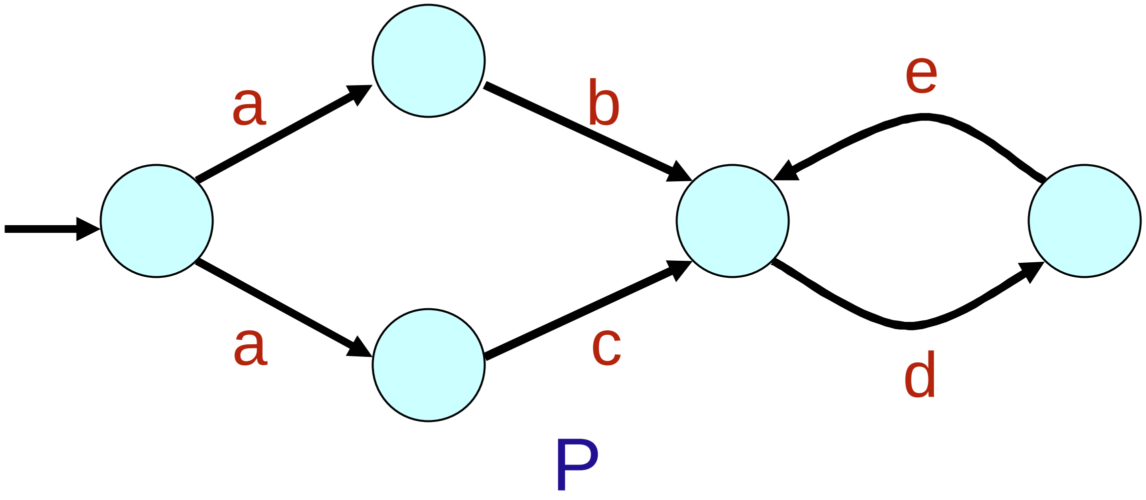
For this talk

- Focus on finite state machines
  - Labeled transition systems (LTSs)

- Parallel composition of state machines
  - Synchronous communication
  - Asynchronous execution
  - Natural for modeling blocking message-passing C programs

Finite LTS

- $P = (Q , I , \Sigma , T)$
  - $Q \equiv$ non-empty set of states
  - $I \in Q \equiv$ initial state
  - $\Sigma \equiv$ set of actions $\equiv$ alphabet
  - $T \subseteq Q \times \Sigma \times Q \equiv$ transition relation



$\Sigma(P) = \{a,b,c,d,e,f\}$

Concurrency

- Components communicate by handshaking (synchronizing) over shared actions
- Else proceed independently (asynchronously)
- Essentially CSP semantics

- Composition of A_1 & $A_2 \equiv A_1 \,||\, A_2$

Operational Semantics

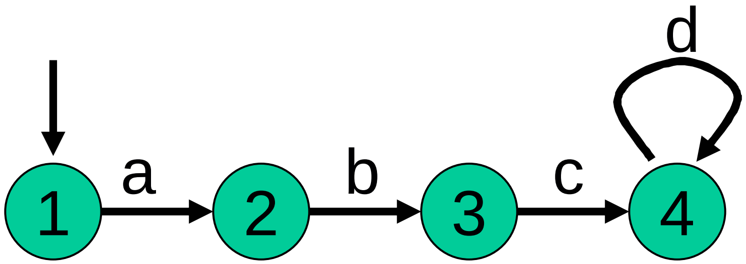
- State of $M_1 \parallel M_2$ is of the form (s_1,s_2) where s_i is a state of M_i

$$\frac{s_1 \xrightarrow{a} s'_1 \quad a \notin \Sigma(M_2)}{(s_1,s_2) \xrightarrow{a} (s'_1,s_2)} \qquad \frac{s_2 \xrightarrow{a} s'_2 \quad a \notin \Sigma(M_1)}{(s_1,s_2) \xrightarrow{a} (s_1,s'_2)}$$

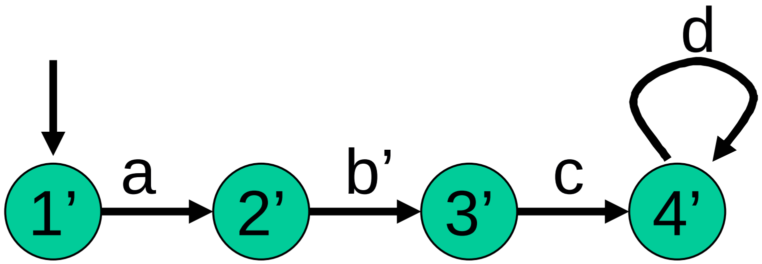
$$\frac{s_1 \xrightarrow{a} s'_1 \qquad s_2 \xrightarrow{a} s'_2}{(s_1,s_2) \xrightarrow{a} (s'_1,s'_2)}$$

State-space exponential in # of components

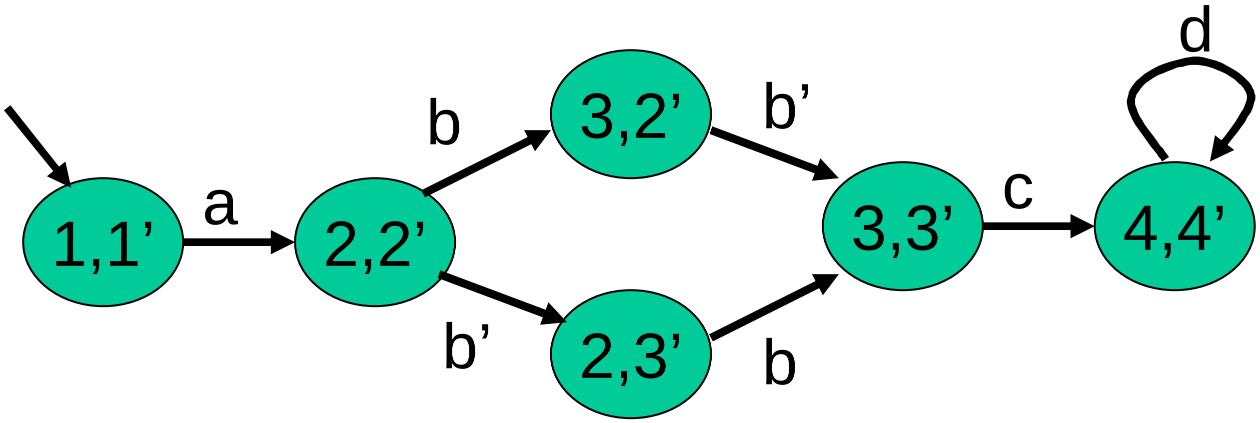
Example



M_1 $\Sigma = \{a,b,c,d\}$

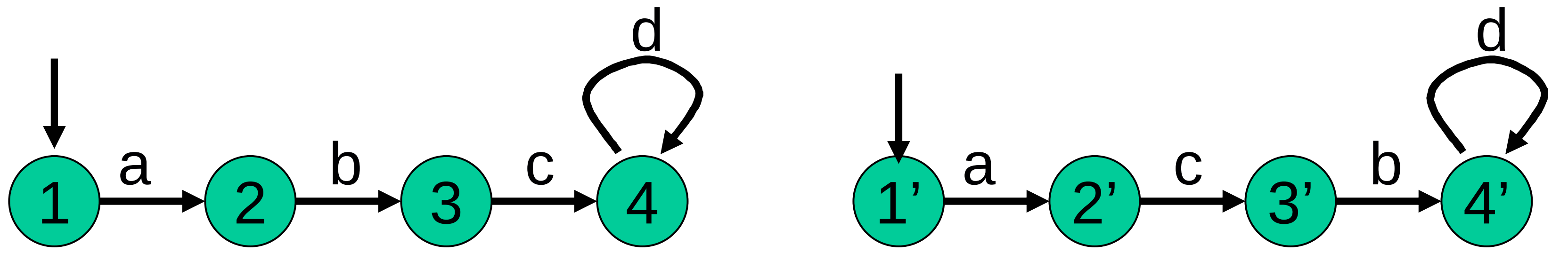


M_2 $\Sigma = \{a,b',c,d\}$

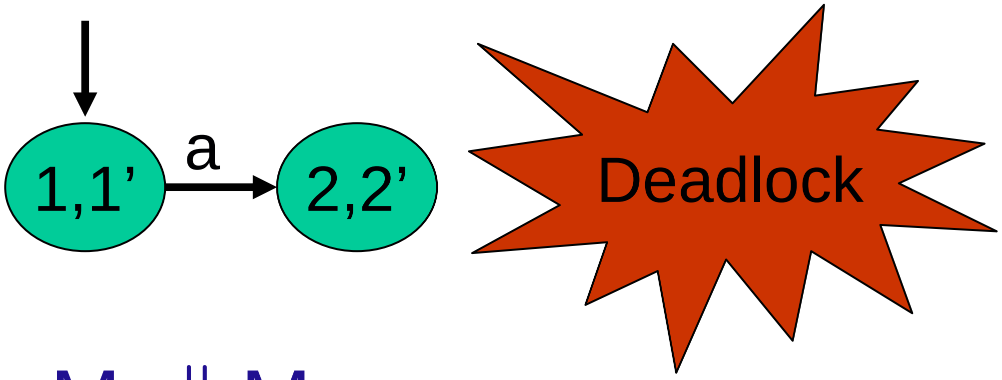


$M_1 \parallel M_2$

Deadlock



M_1 $\Sigma = \{a,b,c,d\}$ M_2 $\Sigma = \{a,b,c,d\}$

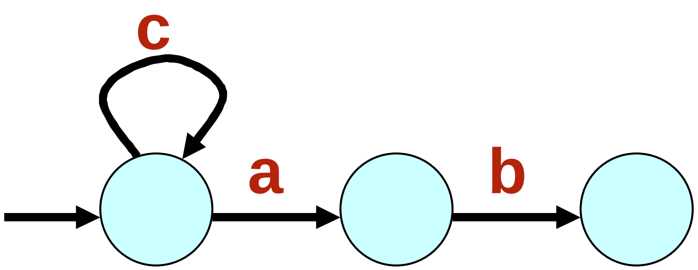


$M_1 \parallel M_2$

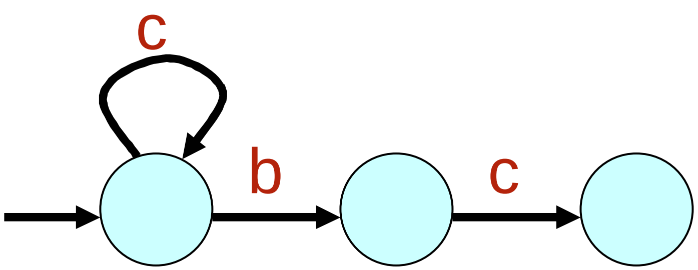
Deadlock ⇔ a reachable state cannot perform any actions at all

Deadlock and Composition

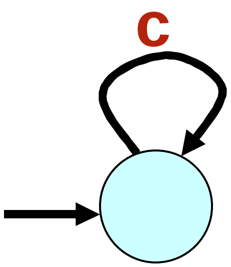
Deadlock



M_1



M_2

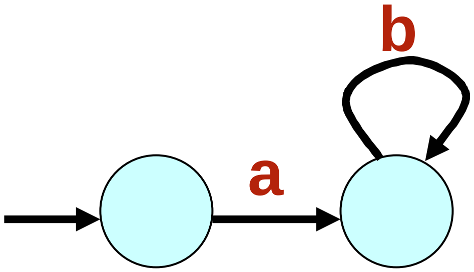


$M_1 || M_2$

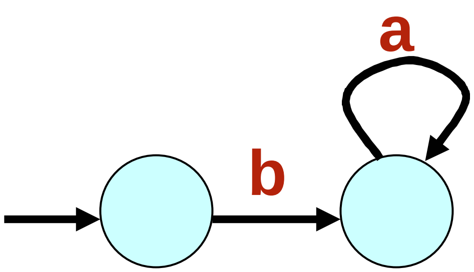
No Deadlock

Deadlock and Composition

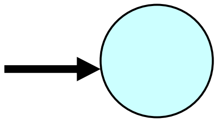
No Deadlock



M_1



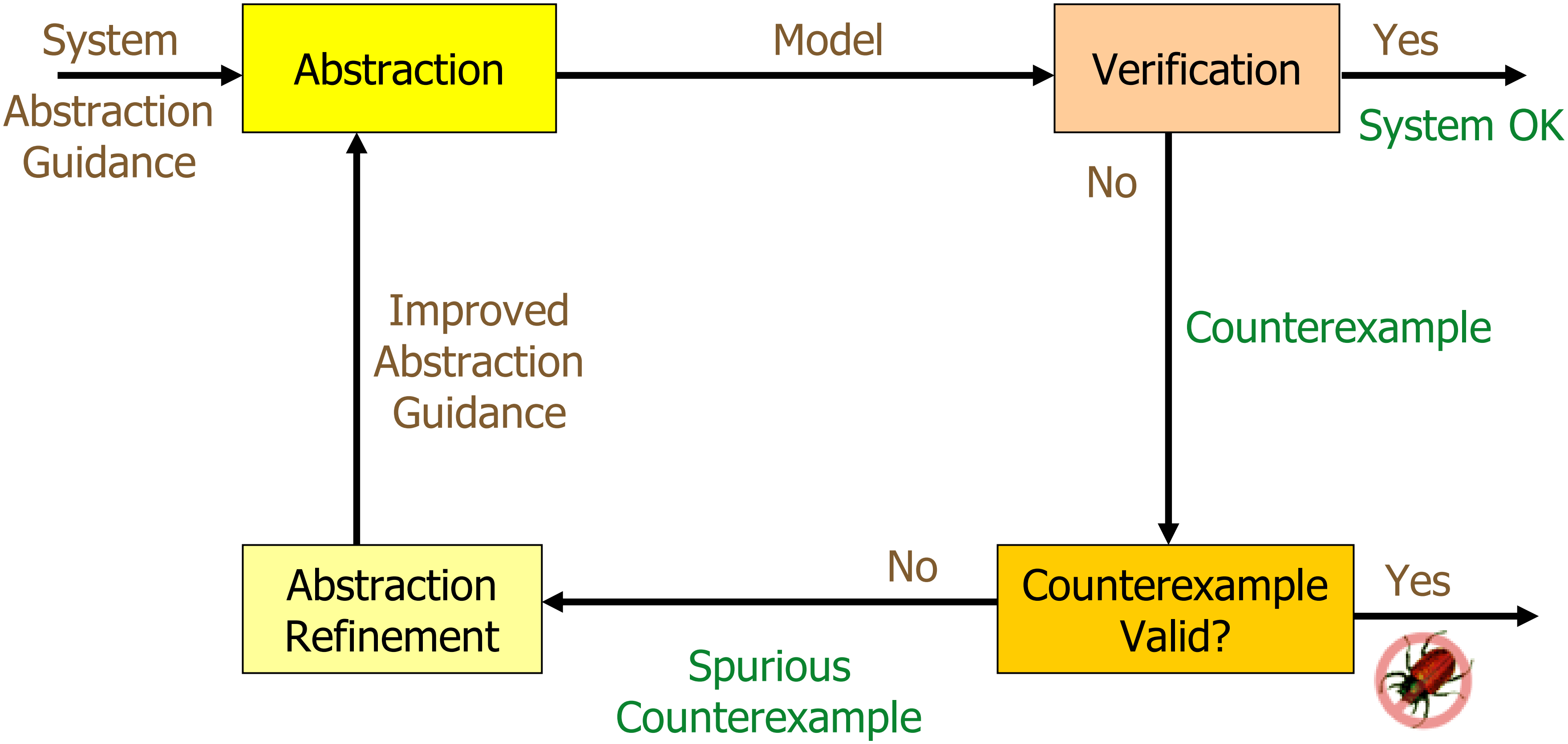
M_1



$M_1 || M_2$

Deadlock

Iterative Refinement



System

Abstraction

Abstraction Guidance

Model

Verification

Yes

System OK

No

Counterexample

Improved Abstraction Guidance

Abstraction Refinement

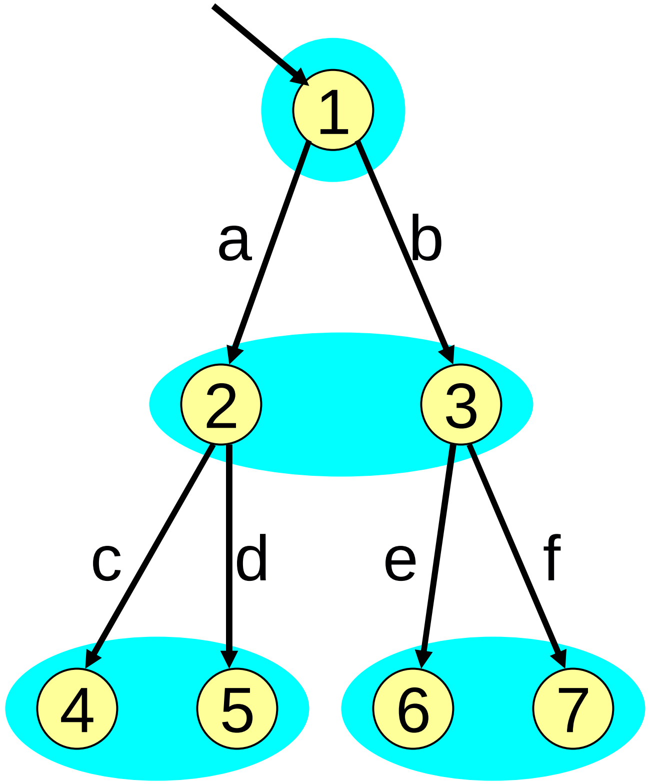
No

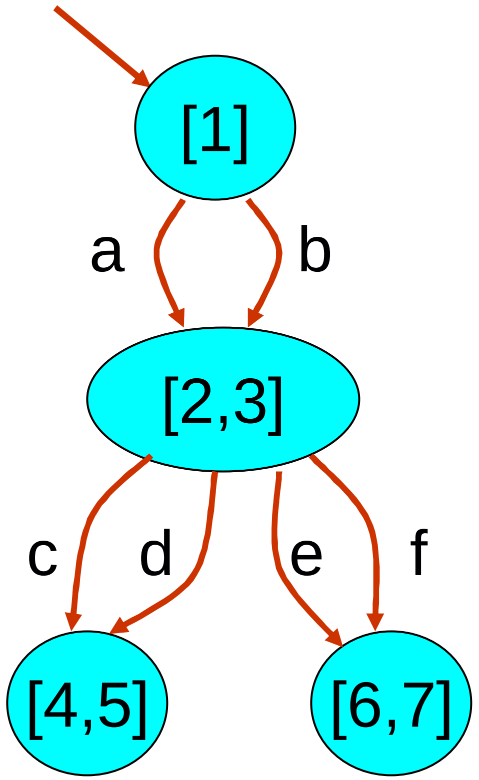
Counterexample Valid?

Yes

Spurious Counterexample

Conservative Abstraction





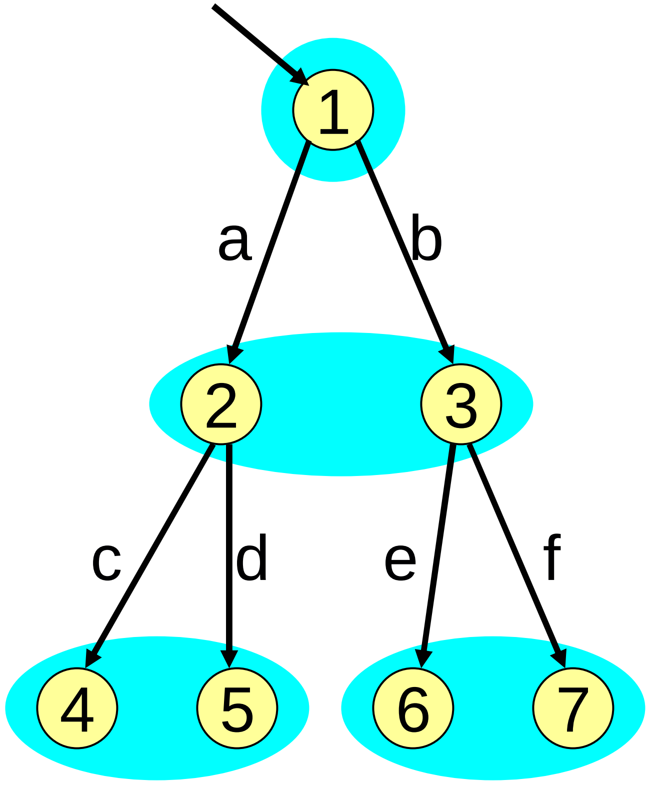
P

A

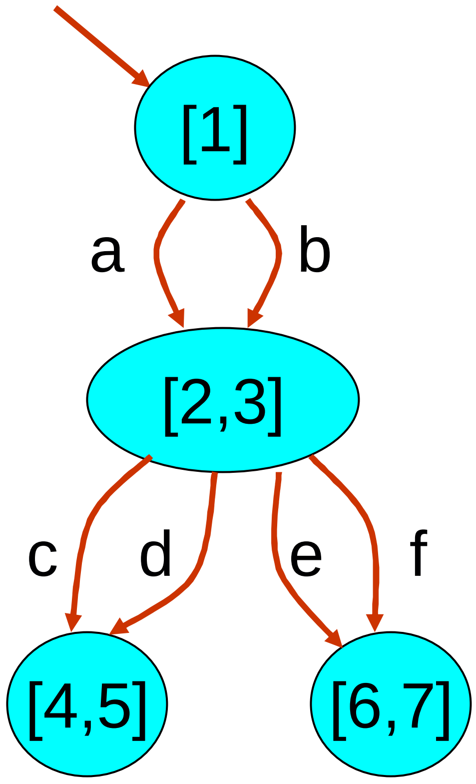
Conservative Abstraction

- Every trace of P is a trace of A
  - Preserves safety properties: $A \models \phi \Rightarrow P \models \phi$
  - A over-approximates what P can do
- Some traces of A may not be traces of P
  - May yield spurious counterexamples - $\langle$ a, e $\rangle$
- Eliminated via abstraction refinement
  - Splitting some clusters in smaller ones
  - Refinement can be automated

Original Abstraction

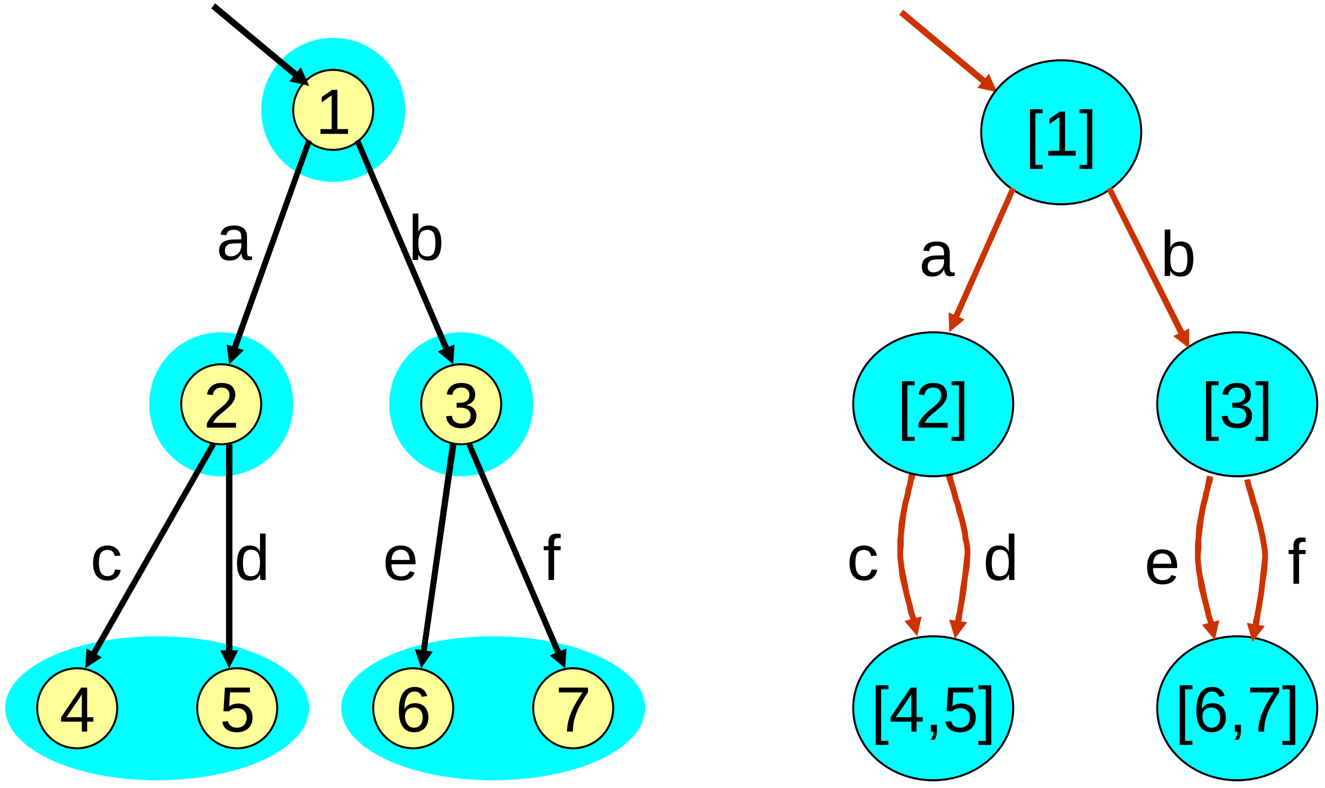


P



A

Refined Abstraction

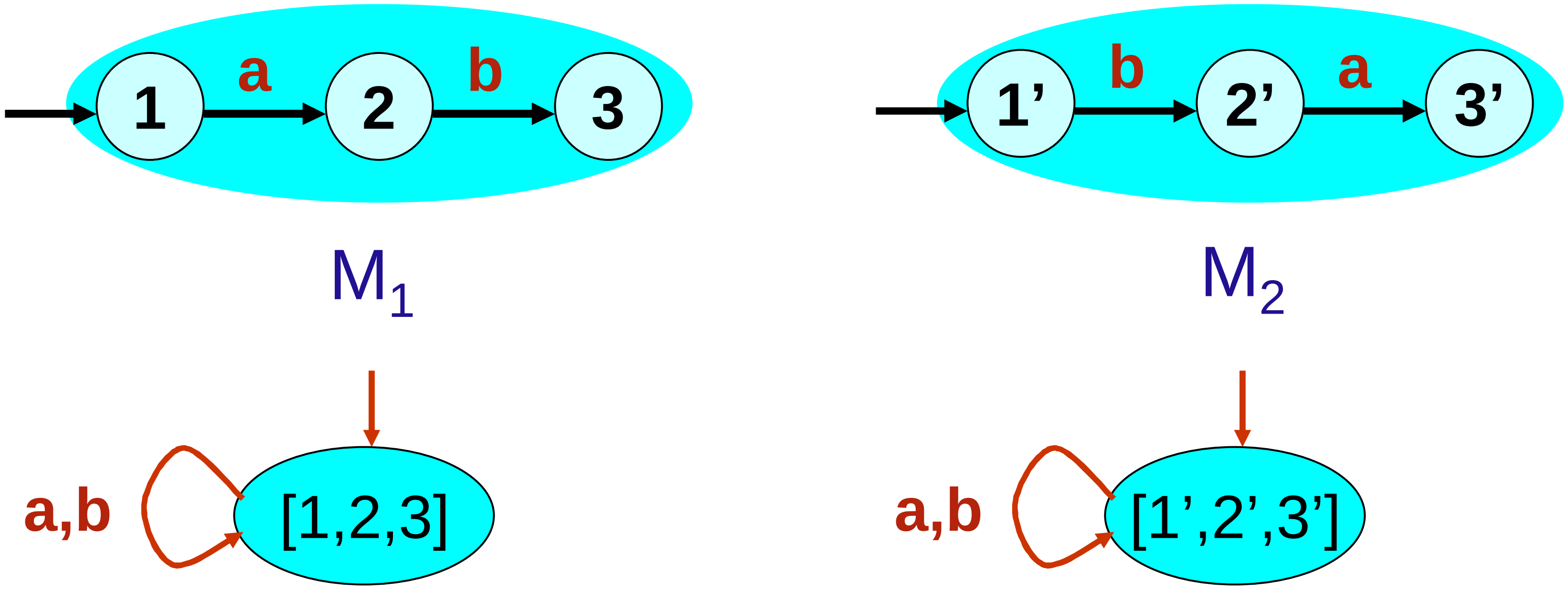


P

A

Deadlock : Problem

- Deadlock is not preserved by abstraction



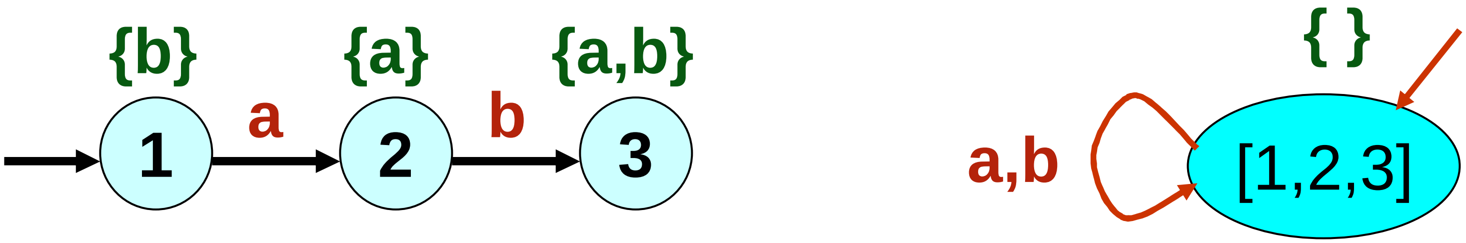
Deadlock Detection : Insight

- Deadlock ⇔ a reachable state cannot perform any actions at all
  - Deadlock depends on the set of actions that a reachable state cannot perform

- In order to preserve deadlock A must over-approximate not just what P can do but also what P refuses

Refusal & Deadlock

- Ref(s) = set of actions s cannot perform

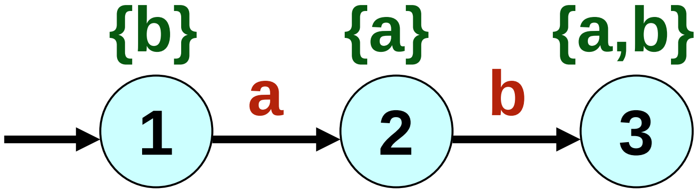


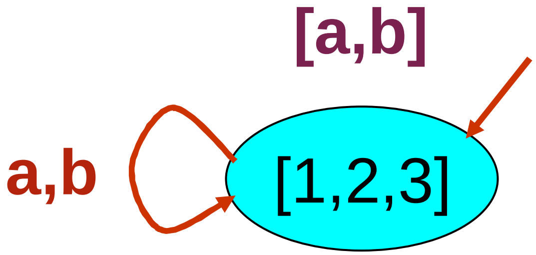
- M deadlocks iff there is a reachable state s such that Ref(s) = Σ
  - Denote by DLock(M)

- $\mathrm{Ref}([s_1 \,..\, s_n]) = \mathrm{Ref}(s_1) \cap \,..\, \cap \mathrm{Ref}(s_n)$

Abstract Refusal

- $AR([s_1 .. s_n]) = Ref(s_1) \cup .. \cup Ref(s_n)$





- $AR([M_1] .. [M_n]) = AR([M_1]) \cup .. \cup AR([M_n])$

Abstract Deadlock

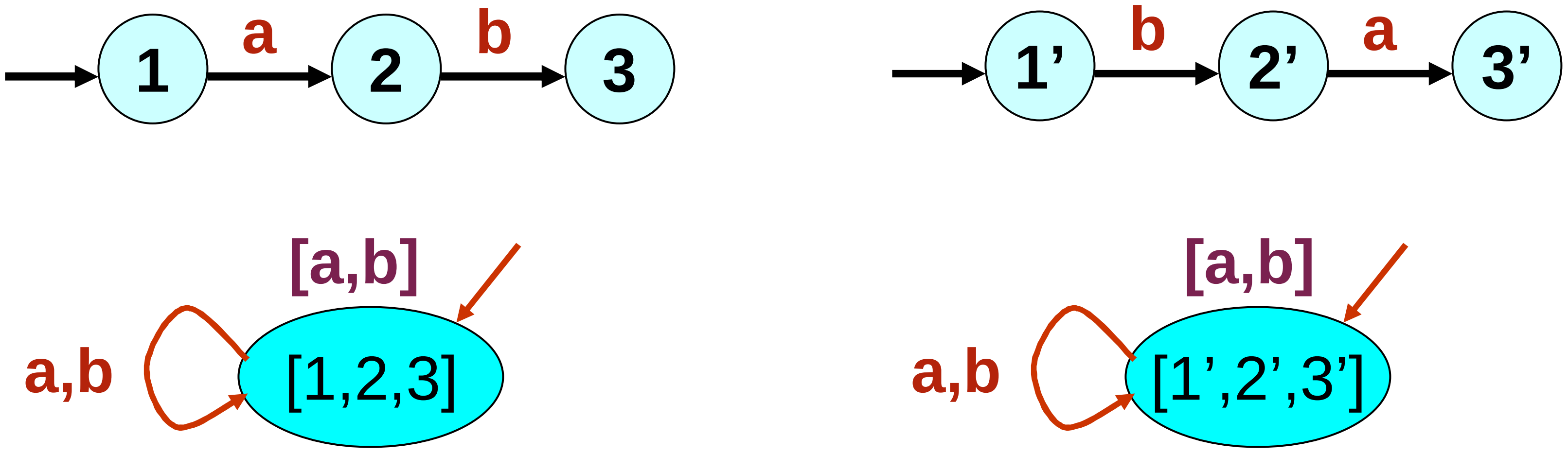
- M abstractly deadlocks iff there is a reachable state s such that $AR(s) = \Sigma$
  - Denote by ADLock(M)

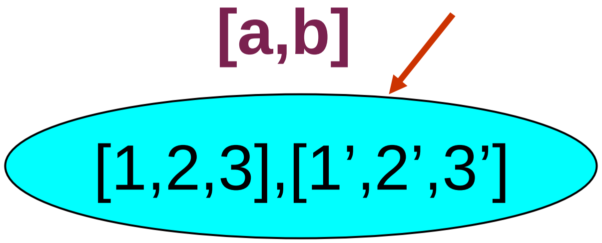
$$\neg\ \mathrm{ADLock}([M_1] \,||\, .. \,||\, [M_n])$$

$$\Rightarrow$$

$$\neg\ \mathrm{DLock}(M_1 \,||\, .. \,||\, M_n)$$

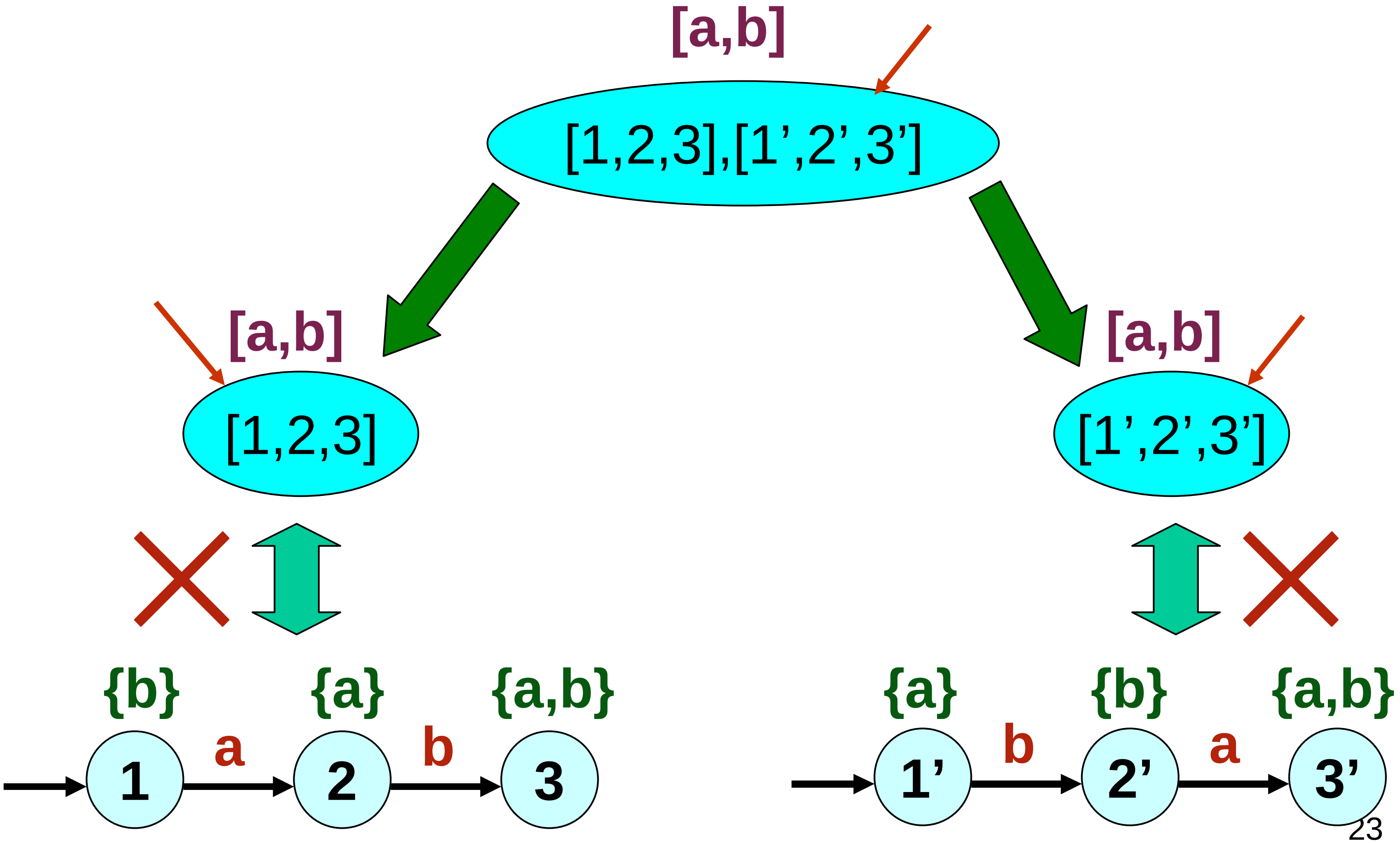
Iterative Deadlock Detection



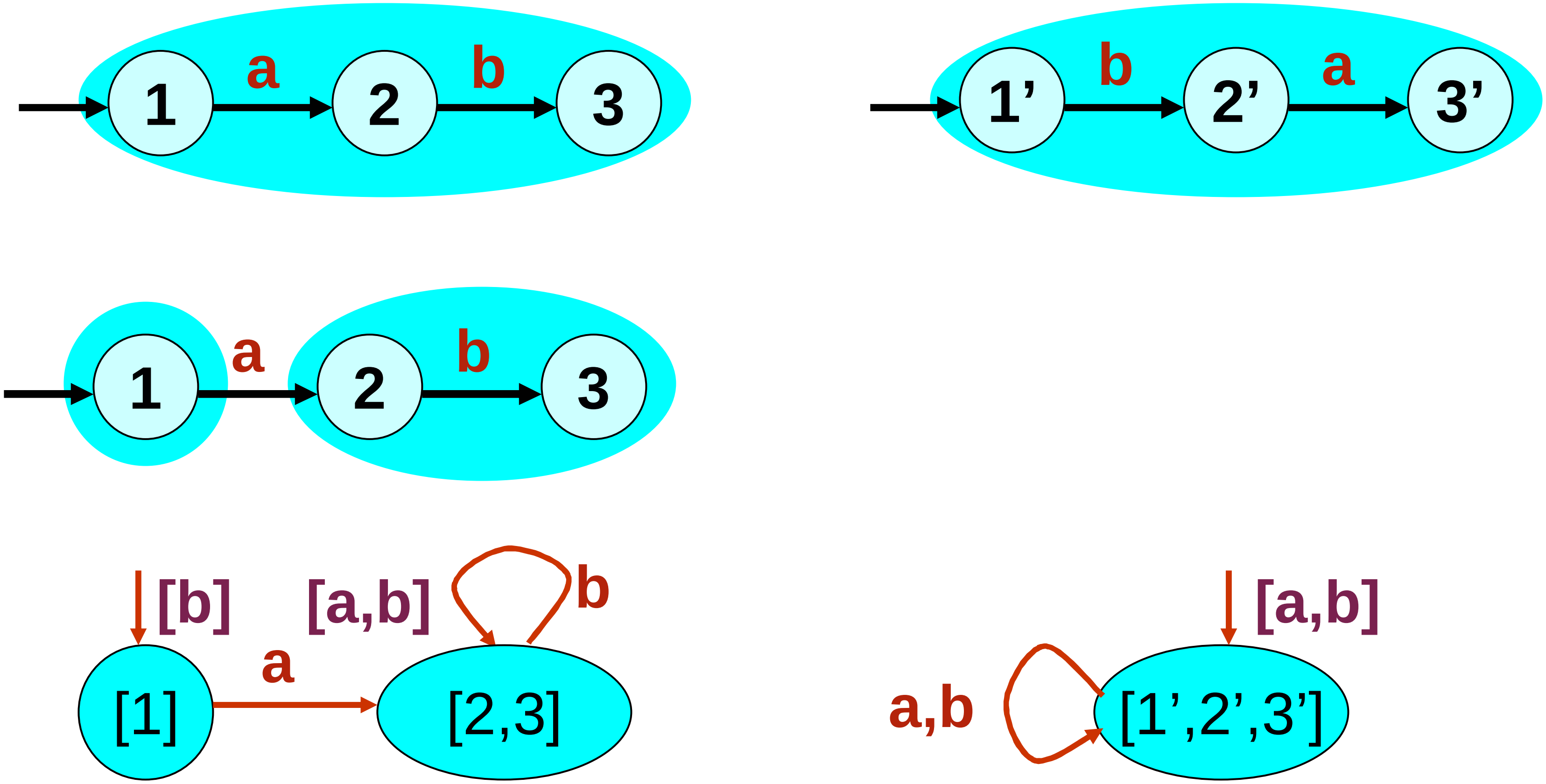


Counterexample to Abstract Deadlock

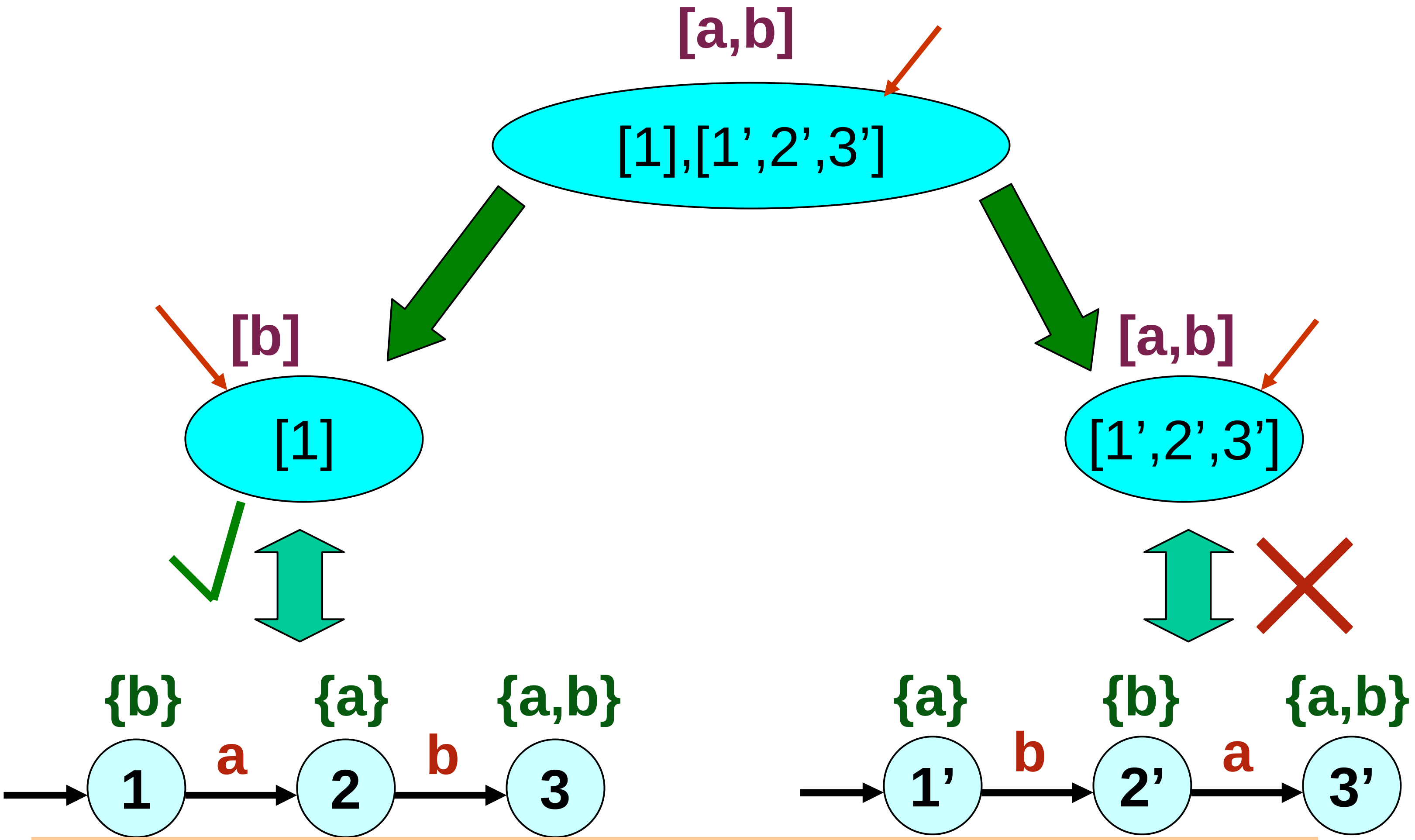
Counterexample Validation



Refinement

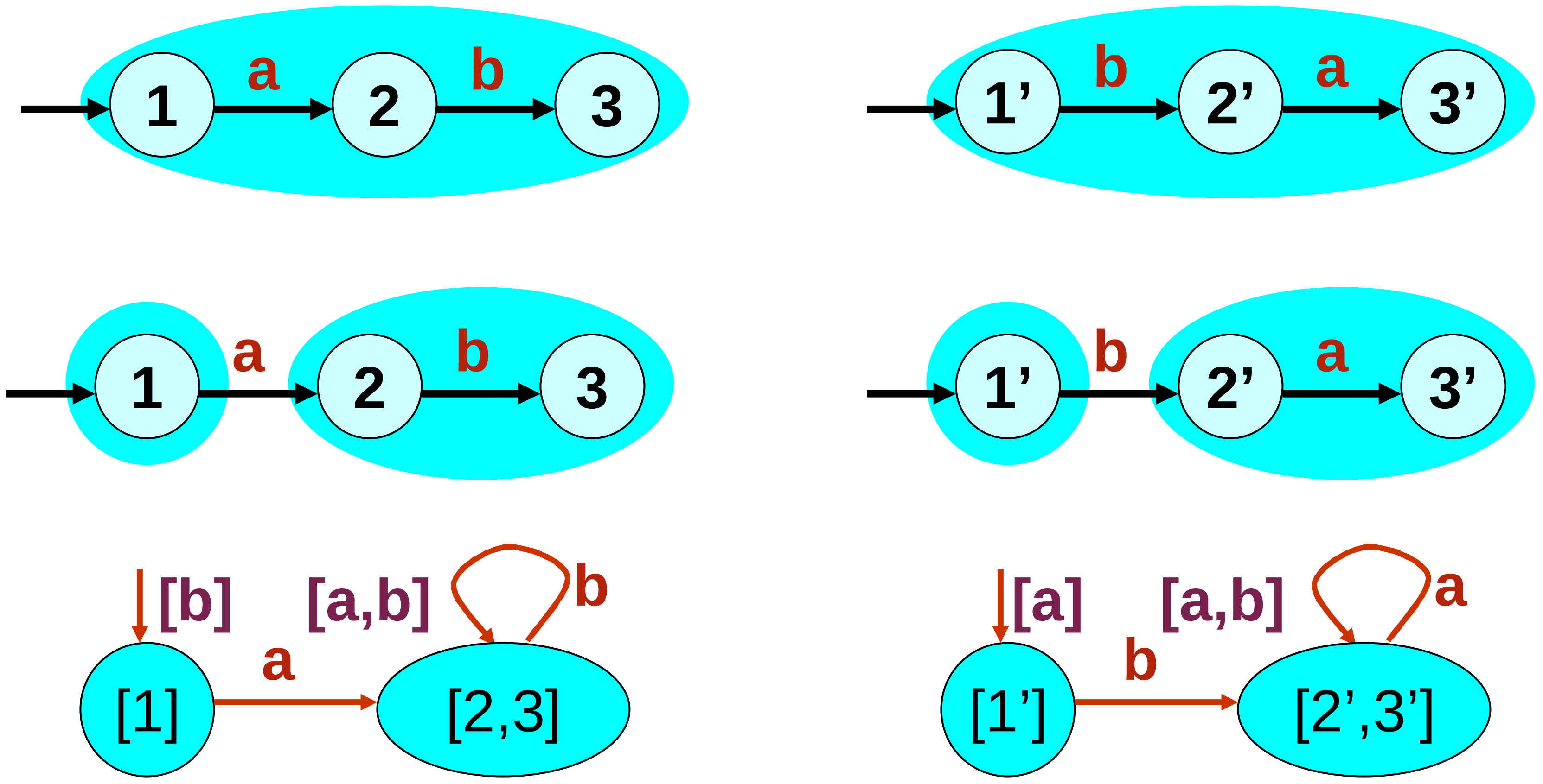


Counterexample Validation

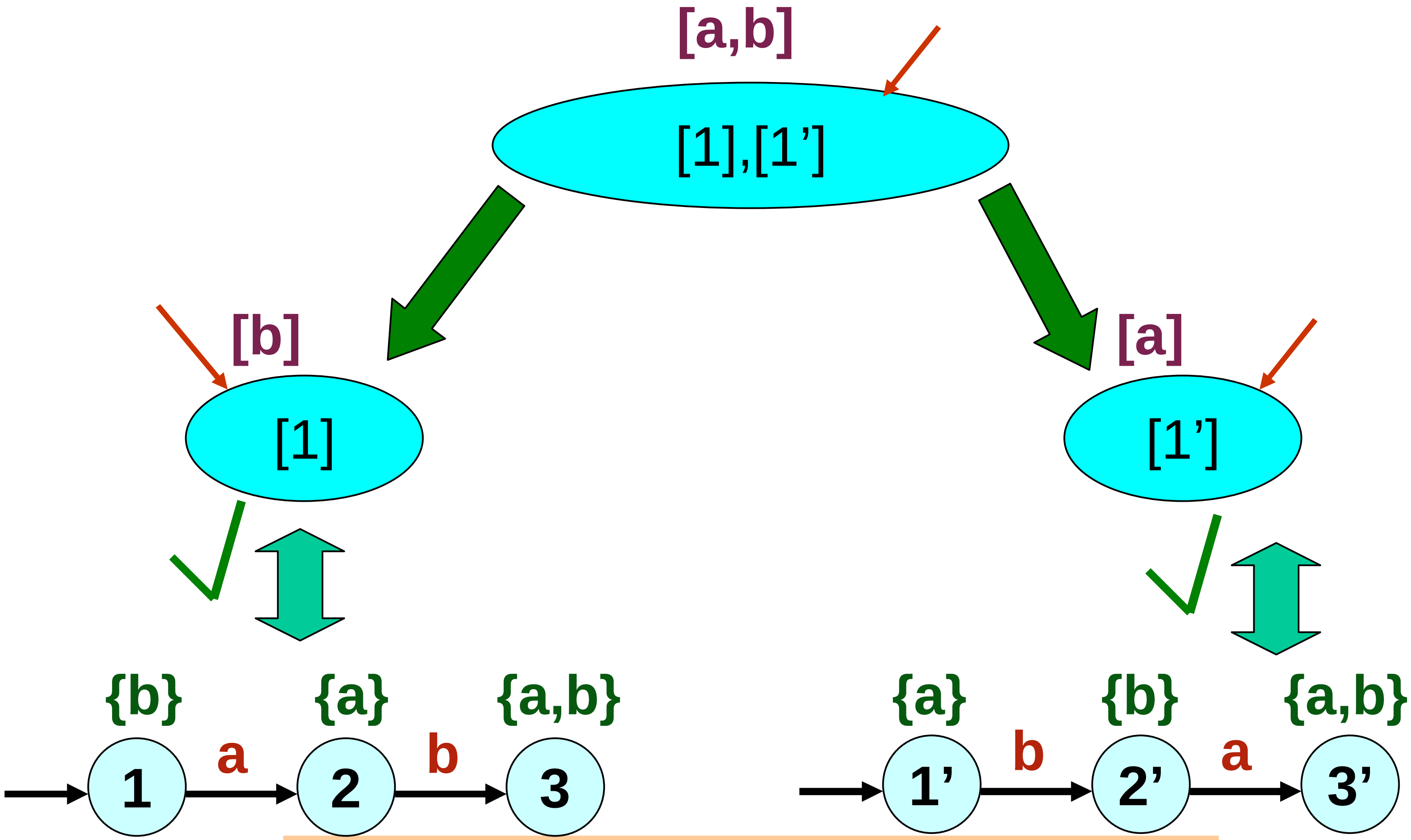


Another spurious counterexample

Refinement

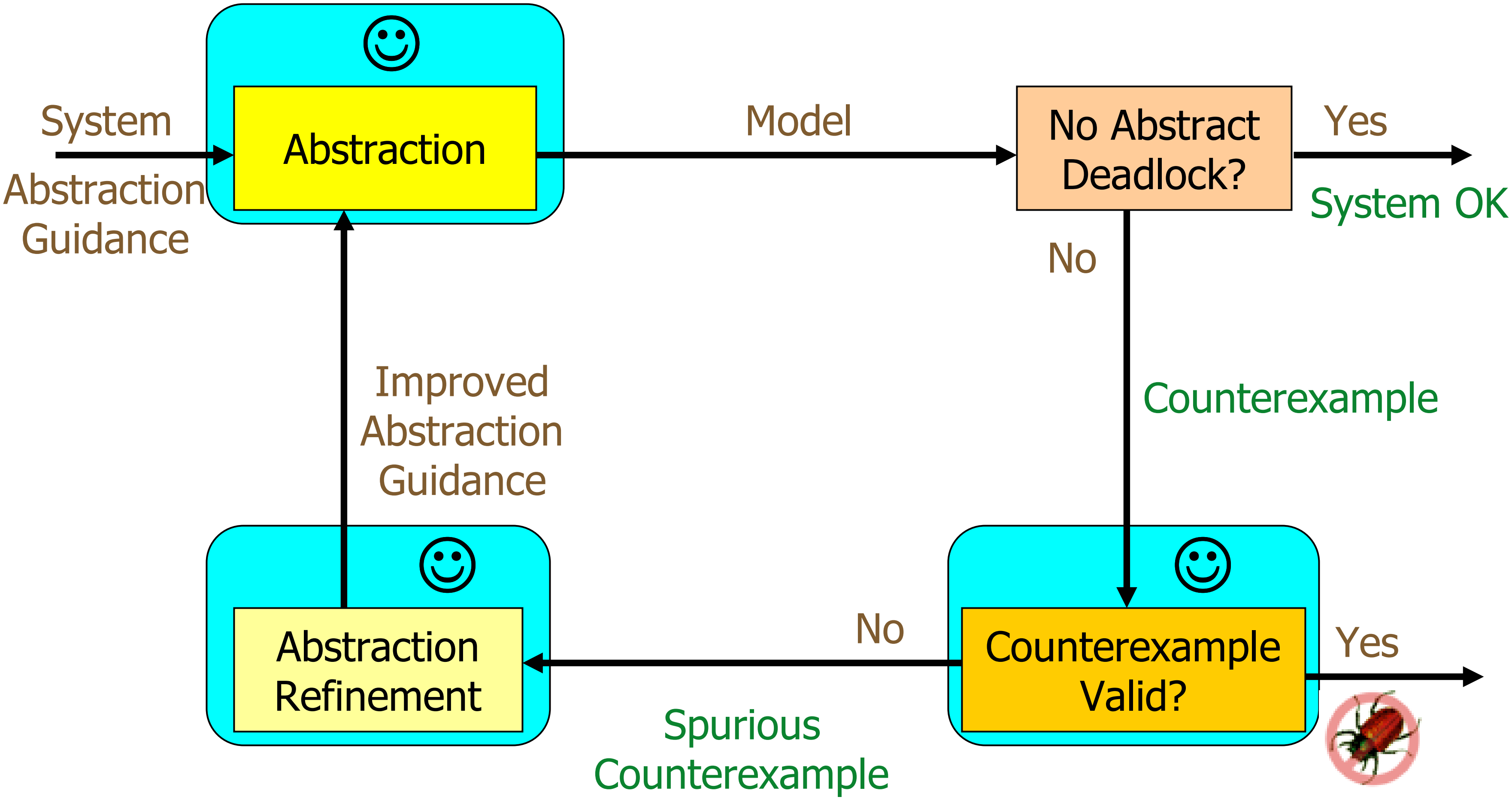


Counterexample Validation



Real Deadlock Detected

Iterative Deadlock



Case Studies

- MicroC/OS-II
  - Real-time OS for embedded applications
  - Widely used (cell phones, medical devices, routers, washing machines…)
  - 6000+ LOC

- ABB IPC Module
  - Deployed by a world leader in robotics
  - 15000+ LOC
  - 4 components
  - Over 30 billion states after predicate abstraction

Results

	Plain			IterDeadlock			
Name	**St**	**T**	**Mem**	**St**	**It**	**T**	**Mem**
ABB	*	*	162	1973	861	1446	33.3
SSL	25731	44	43.5	16	16	31.9	40.8
μCD-3	*	*	58.6	4930	120	221.8	15
μCN-6	*	*	219.3	71875	44	813	30.8
DPN-6	*	*	203	62426	48	831	26.1
DPD-10	38268	87.6	17.3	44493	51	755	18.4

* indicates out of time limit (1500s)

Ongoing and Future Work

- Shared memory
- Assume-Guarantee reasoning
- Industrial size examples
- Symbolic implementation
- Branching-time state/event logic (completed)

Component Substitutability: Motivation

- Model checking is a highly time consuming, labor intensive effort

- For example, a system of 25 components (~20K LOC) and 100+ properties might take up to a month of verification effort

- It discourages practitioner use when system evolves

- Can model checking be used to automatically determine if previously established properties will hold for the *evolved* system without repeating each of the individual checks

What’s The Problem

- Software evolution is inevitable in any real system:
  - Changing requirements
  - Bug fixes
  - Product changes (underlying platform, third-party,etc.)
  - Incremental verification during the design process

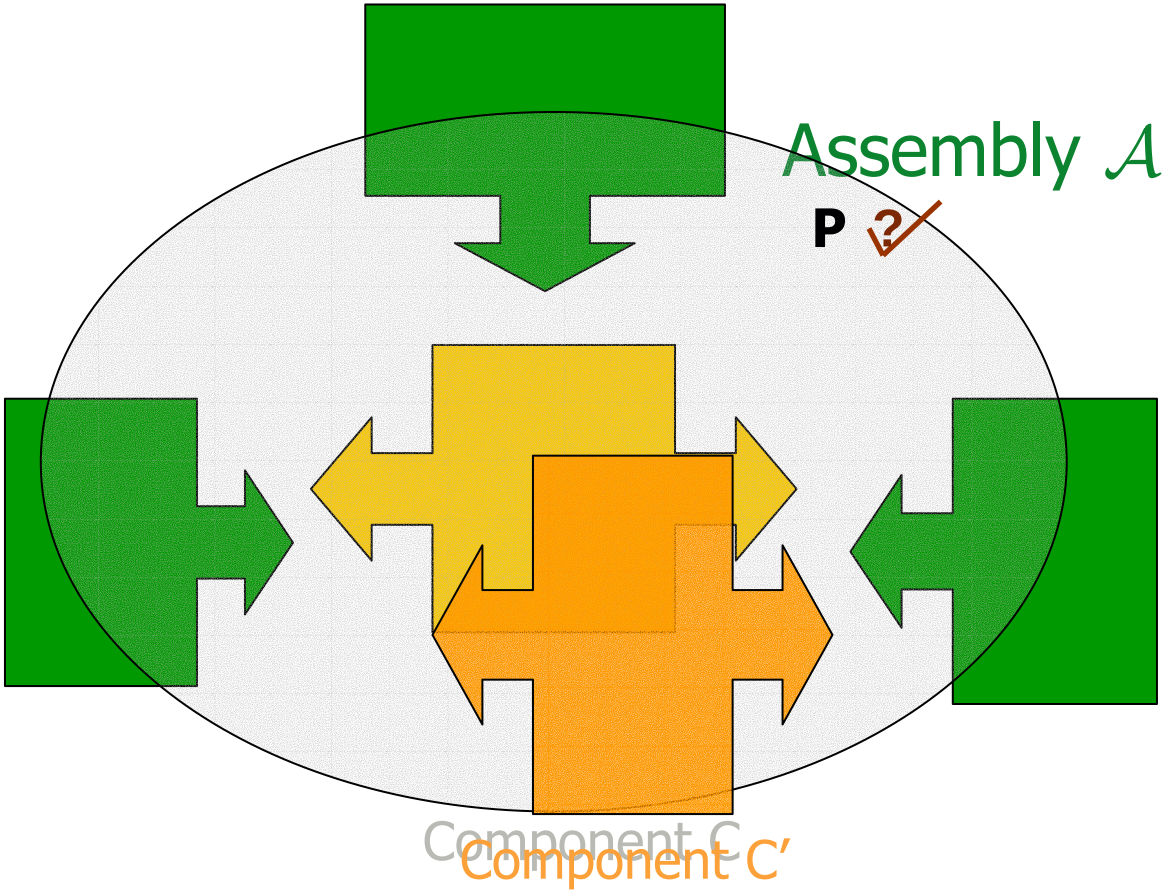
Component Substitutability Check

- Component-based Software
  - Software modules shipped by separate developers
  - Undergo several updates/bug-fixes during their lifecycle
- Component assembly verification
  - Necessary on upgrade of any component
  - High costs of complete global verification
- Idea:
  - Instead check locally for substitutability of new components

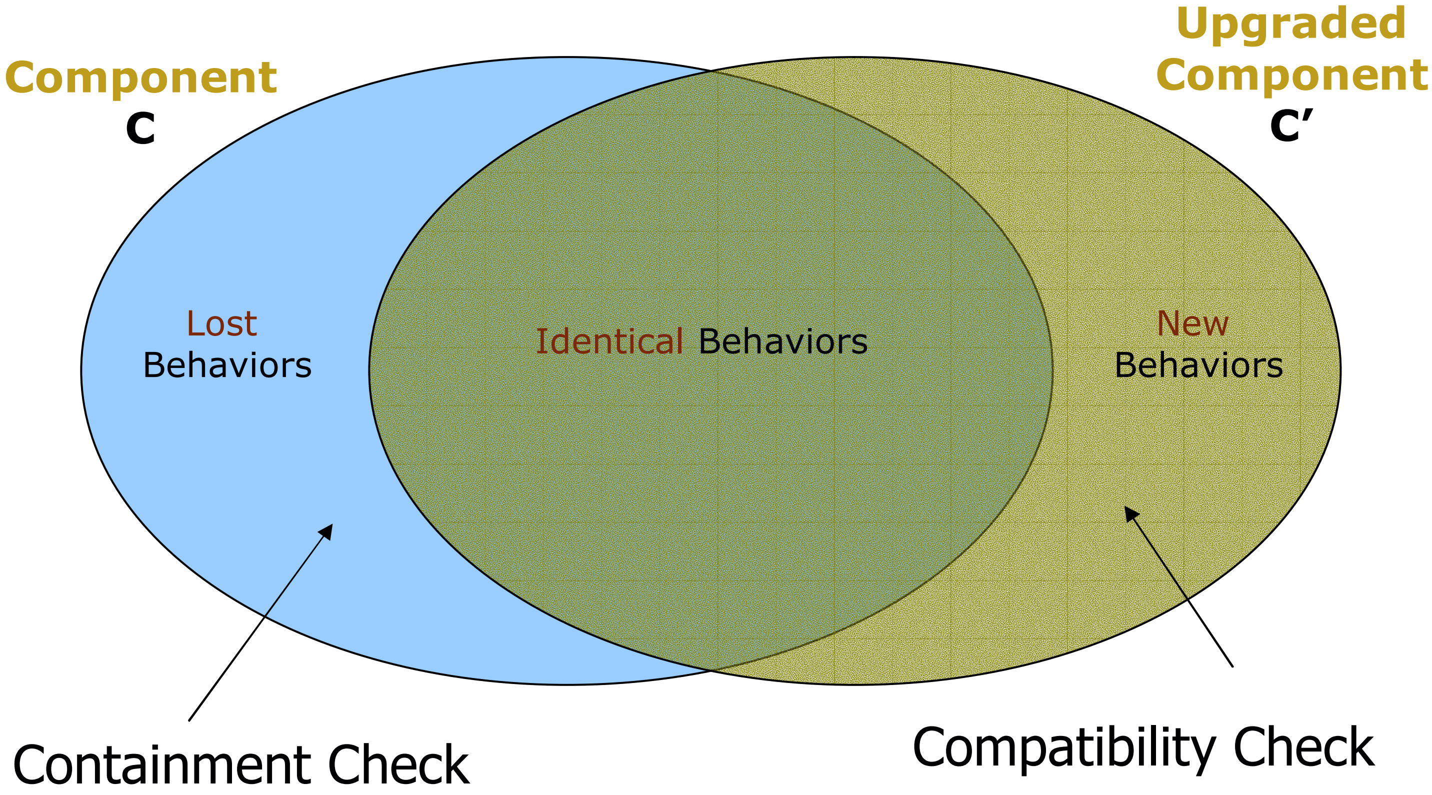
Potential Contribution

- Verify upgraded components locally
- Reuse previous verification results
- For example, for a system of 25 components (~20K LOC) and 100+ properties verification might take up to a month of verification effort
- If 3 components change, instead of repeating a month effort of re-verifying 100+ properties, our technique will ensure the substitutability of all properties in one iteration of the substitutability check (~ 1 day effort).

Component Substitutability Check



Substitutability Check Approach



Substitutability Check Approach

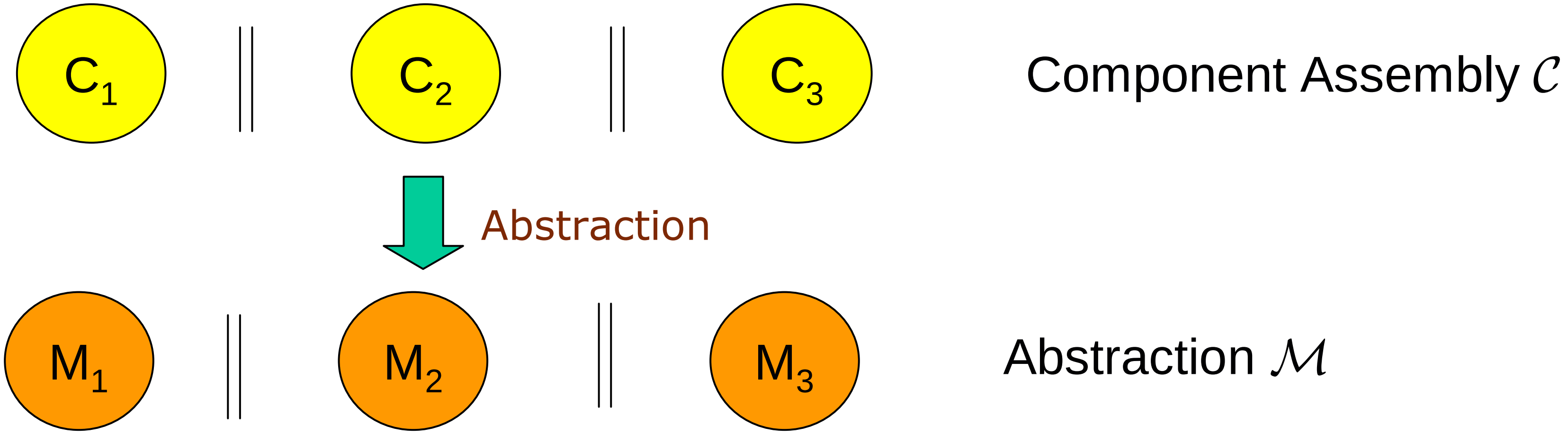
- Two phases:
  - – Containment check (Local correctness)
    - Are all local old services (properties) of the verified component contained in the upgraded component?
  - – Compatibility check (Global safety check)
    - Are new services of the upgraded component safe with respect to other components in assembly: all global specifications still hold?

Substitutability Check

- Approach:
  - Obtain finite state models of all components by abstraction
  - Containment Check:
    - Use under- and over- approximations (new)
  - Compatibility Check:
    - Use dynamic assume-guarantee reasoning (new)

Component Assembly

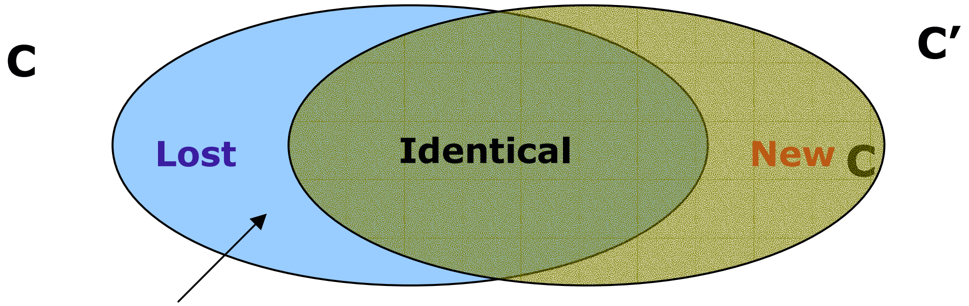
- A set of communicating concurrent C programs (components)
- Each component abstracted into a Component FSM



Containment Check

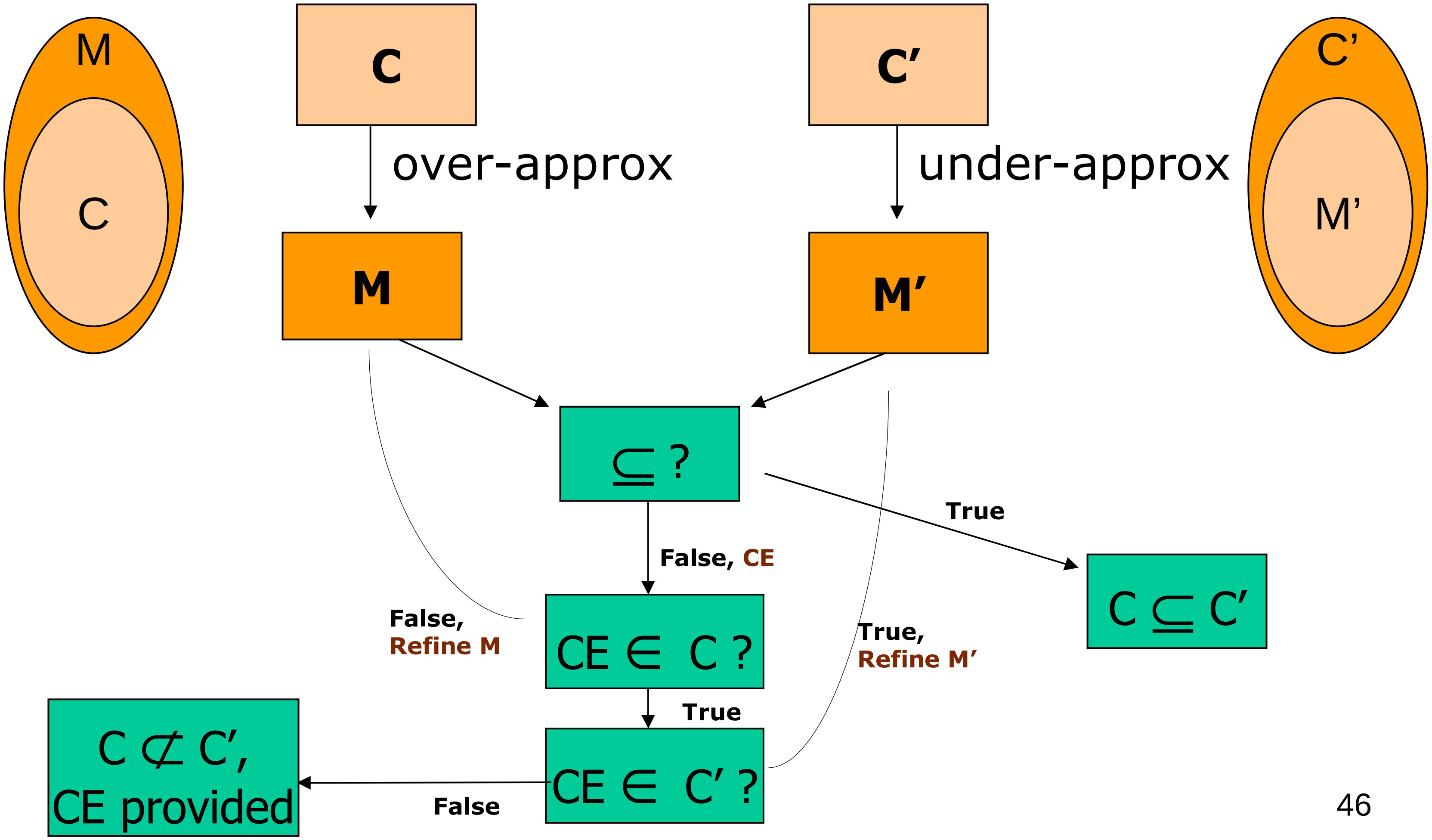
- Goal: Check C ⊆ C’ (Every behavior of C is an allowable behavior of C’)
  - All behaviors retained after upgrade

- Solution:
  - Create abstraction (over-approximation) M: C ⊆ M
  - Create abstraction (under-approximation) M’: M’ ⊆ C’
  - Check for M ⊆ M’



Containment Check

Containment Check (cont.)



M

C

C

over-approx

M

C′

under-approx

M′

C’

M’

$\subseteq$?

True

$C \subseteq C'$

False, CE

$CE \in C$?

False, Refine M

True, Refine M′

True

$CE \in C'$?

False

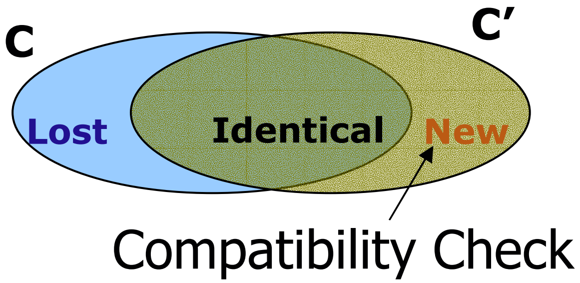
$C \not\subseteq C'$, CE provided

Containment Check (cont.)

- Computing over-approximation
  - Conventional predicate abstraction
- Computing under-approximation
  - Modified predicate abstraction
  - Compute Must transitions instead of May

Compatibility Check

- Assume-guarantee to verify assembly properties
  - Related: Cobleigh et. al. at NASA Ames



- Reuse previous verification results

$$\frac{M_1 \parallel A \models P \qquad M_2 \models A}{M_1 \parallel M_2 \models P}$$

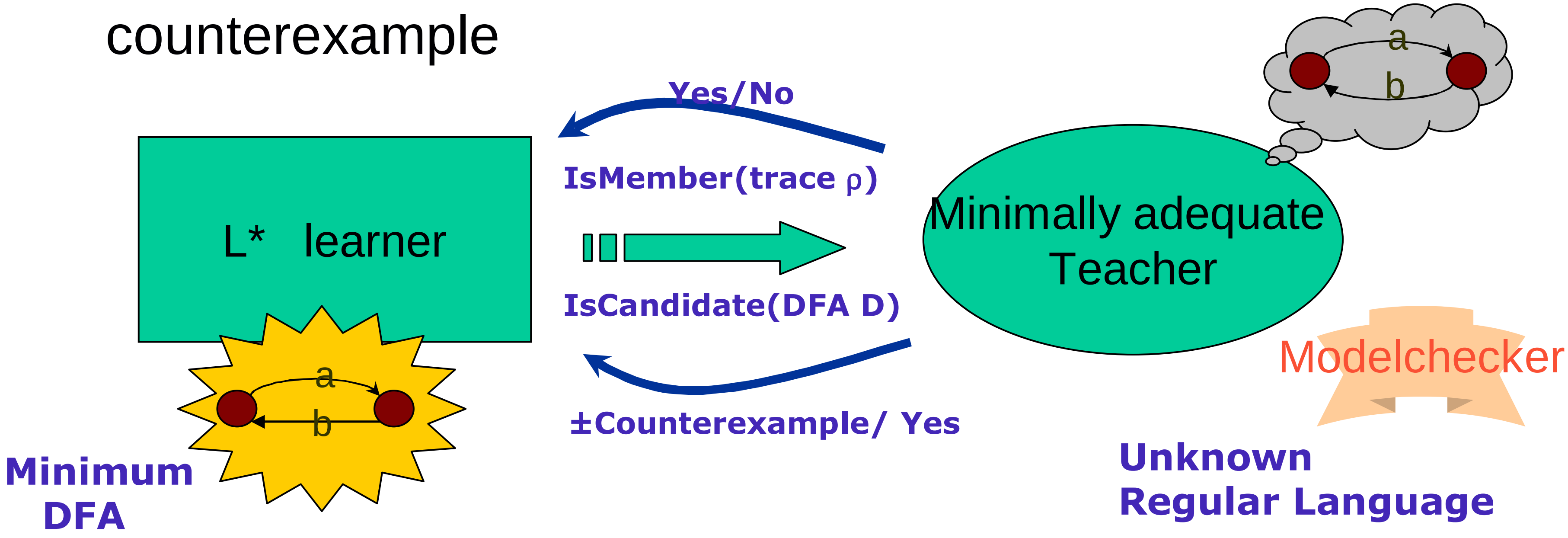
AG-NonCircular

- Use learning algorithm for regular languages, L*
- Automatically generate assumption A

Learning Regular languages: L*

- Proposed by D. Angluin, improved by Rivest et al.
  - *Learning regular sets from queries and counterexamples,* Information and Computation, 75(2), 1987.

- Polynomial in the number of states and length of counterexample



Yes/No

L* learner

IsMember(trace ρ)

IsCandidate(DFA D)

±Counterexample/ Yes

Minimally adequate Teacher

Modelchecker

Minimum DFA

Unknown Regular Language

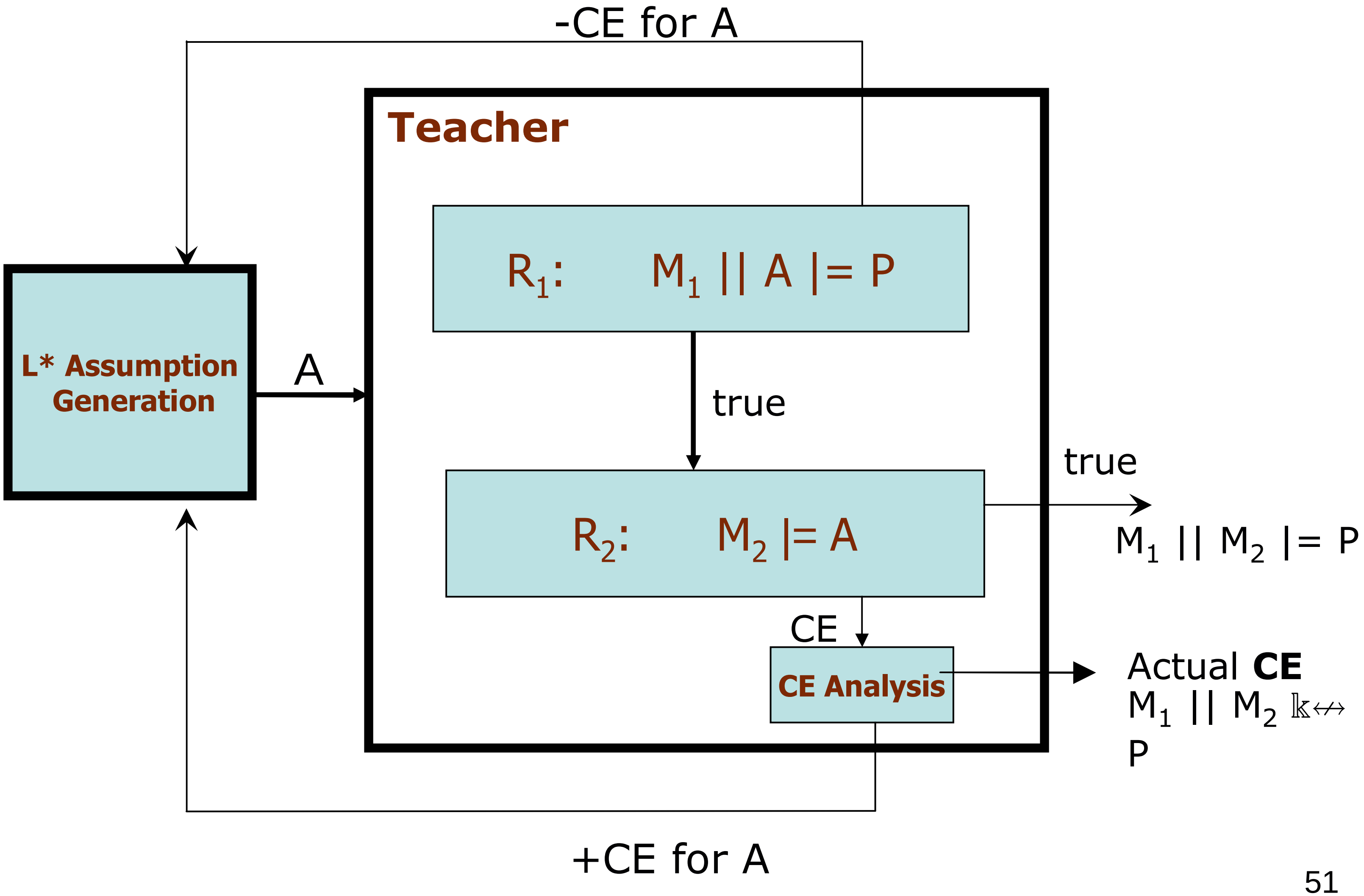
Learning for Verification

- Model checker as a Teacher
  - Possesses information about concrete components
  - Model checks and returns true/counterexample

- Learner builds a model sufficient to verify properties

- Wide applications:
  - *Adaptive Model Checking*: Groce et al.
  - *Automated Assume-Guarantee Reasoning*: Cobleigh et al.
  - *Synthesize Interface Specifications for Java Programs*: Alur et al.

Compatibility Check



-CE for A

Teacher

R_1: $M_1 \,||\, A \models P$

true

L* Assumption Generation

A

R_2: $M_2 \models A$

true

$M_1 \,||\, M_2 \models P$

CE

CE Analysis

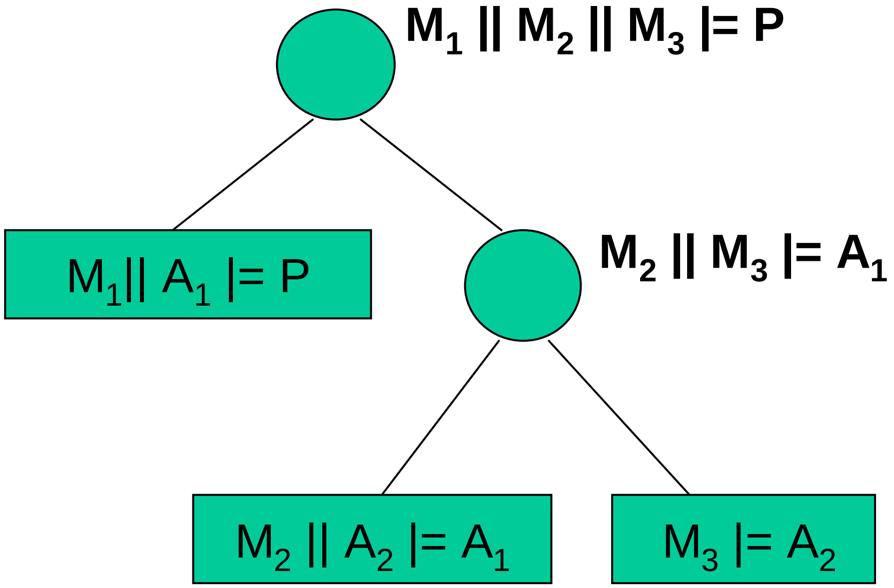
Actual **CE**
$M_1 \,||\, M_2$ 𝕜↮
P

+CE for A

Handling Multiple Components

- AG-NC is recursive
  - (Cobleigh et al.)

$$\begin{array}{ll} R_1: & M_1 \parallel A \models P \\ R_2: & M_2 \models A \\ \hline & M_1 \parallel M_2 \models P \end{array}$$



$M_1 \parallel M_2 \parallel M_3 \models P$

$M_1 \parallel A_1 \models P$

$M_2 \parallel M_3 \models A_1$

$M_2 \parallel A_2 \models A_1$

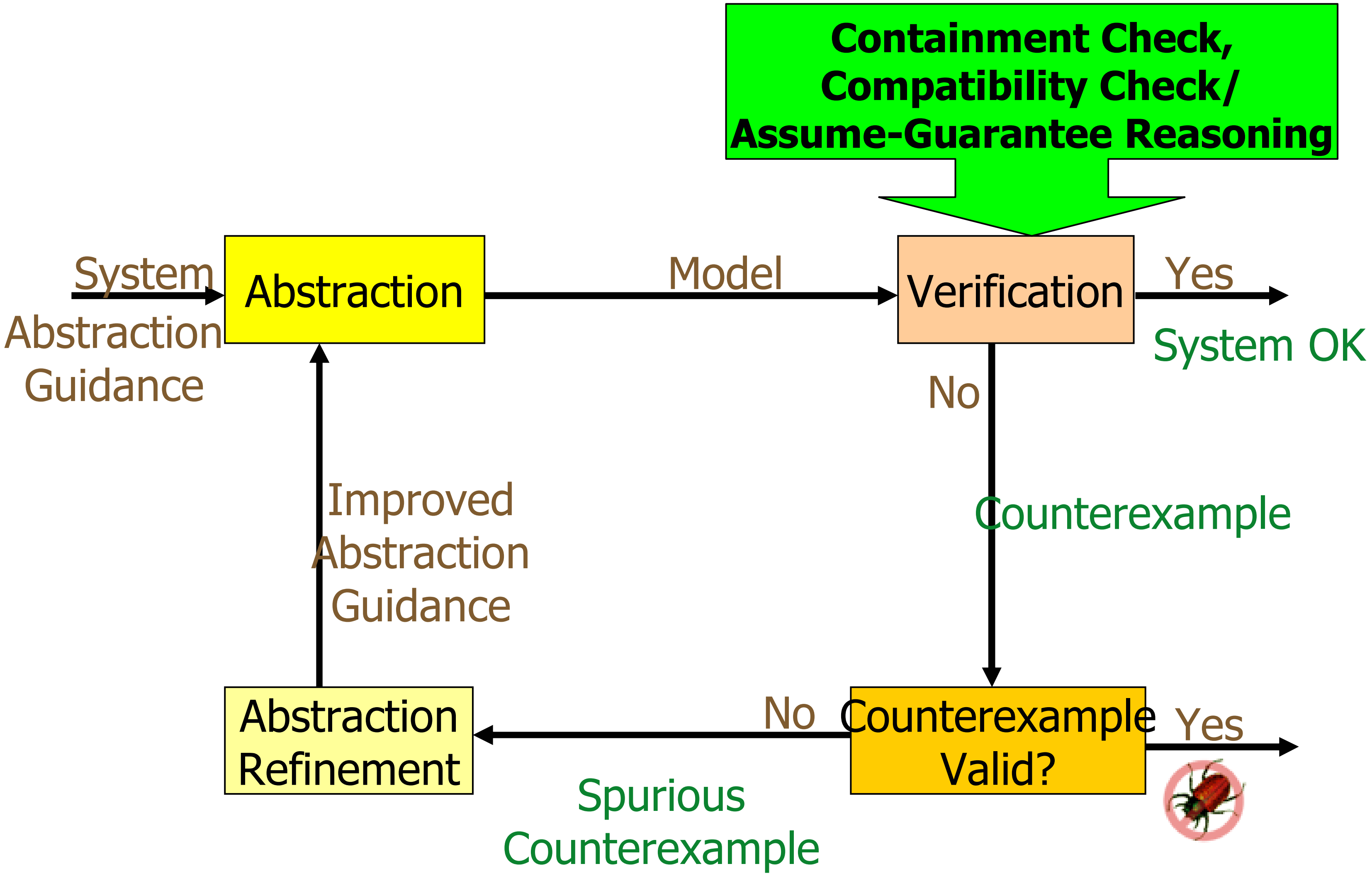
$M_3 \models A_2$

- Each A_i computed by a separate L* instantiation

Implementation

- ComFoRT Framework
- Validated on an Industrial benchmark
  - Inter-process Communication (IPC) ABB software
  - 4 main components – CriticalSection, IPCQueue, ReadMQ, WriteMQ
- Evaluated on single and simultaneous upgrades
  - WriteMQ and IPCQueue components
- Properties
  - P_1: Write after obtaining CS lock
  - P_2: Correct protocol to write to IPCQueue

ComFoRT Schema



Containment Check, Compatibility Check/ Assume-Guarantee Reasoning

System

Abstraction Guidance

Abstraction

Model

Verification

Yes

System OK

No

Counterexample

Improved Abstraction Guidance

Abstraction Refinement

No

Counterexample Valid?

Yes

Spurious Counterexample

Lab Assignment

- Spit into groups of 4-5 people
- Design, implementation and verification of the current surge protector
  - In PROMELA/SPIN
  - In ComFoRT
- Comparative validation
- Presentations on March 31, 2005

Lab Assignment (2)

- Questions about ComFoRT
  - Natasha Sharygina: nys@sei.cmu.edu - *theory*
  - Sagar Chaki: chaki@sei.cmu.edu – *tool support*

Collaboration Opportunities

- Research and development projects on verification of software (ComFoRT project)
- As part of the PACC (Predictable Assembly from Certifiable Components) project at the SEI
- Joint work with Prof. Ed Clarke

Collaboration Opportunities

- Independent studies
- M.S. and Ph.D. Research (jointly with your current advisors)
- Internships

If interested contact me and we can discuss options