Dynamic Component Substitutability Analysis

Edmund Clarke
Natasha Sharygina*
Nishant Sinha

Carnegie Mellon University
The University of Lugano

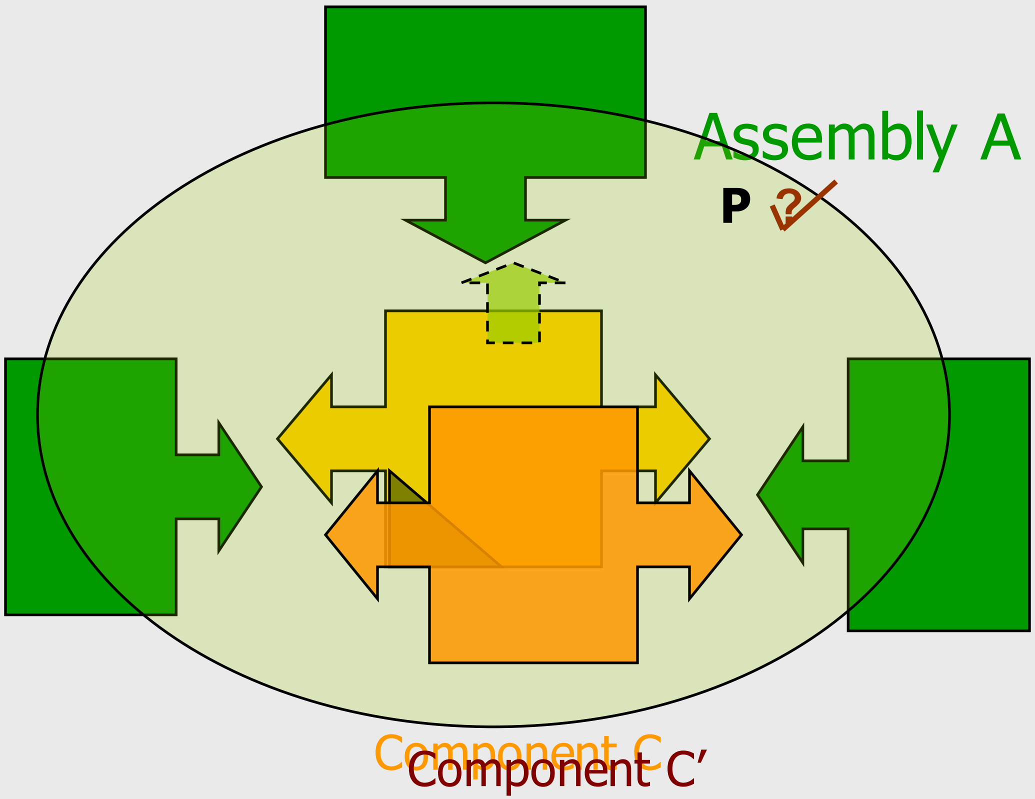
Motivation

- 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
- Discourages its widespread use when system *evolves*

Software Evolution

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

Substitutability Check



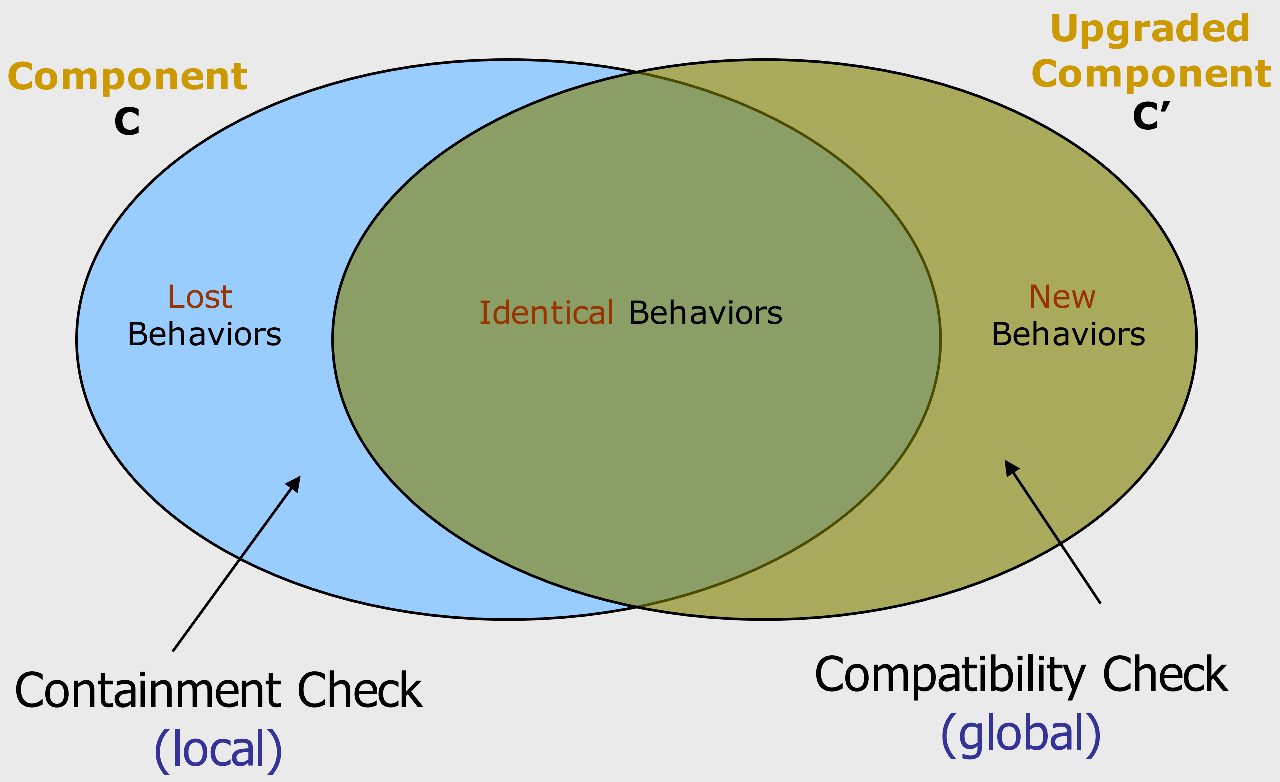
Motivation

- 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
  - Instead check for substitutability of new component

Substitutability Check

- Incremental in nature
- Two phases:
  - Containment check
    - All local behaviors (services) of the previous component contained in new one
  - Compatibility check
    - Safety with respect to other components in assembly: all global specifications still hold

Containment, Compatibility Duality

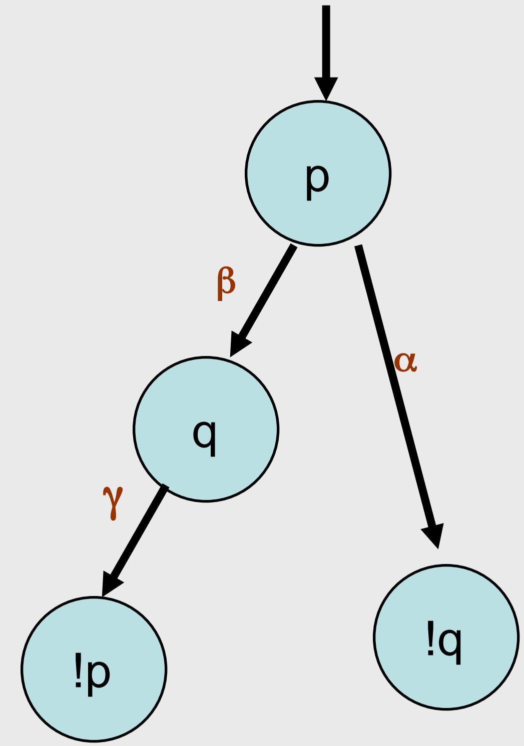


Substitutability Check

- Approaches
  - Obtain a finite behavioral model of all components by abstraction: Labeled Kripke structures
  - Containment:
    - Use under- and over- approximations
  - Compatibility:
    - Use dynamic assume-guarantee reasoning

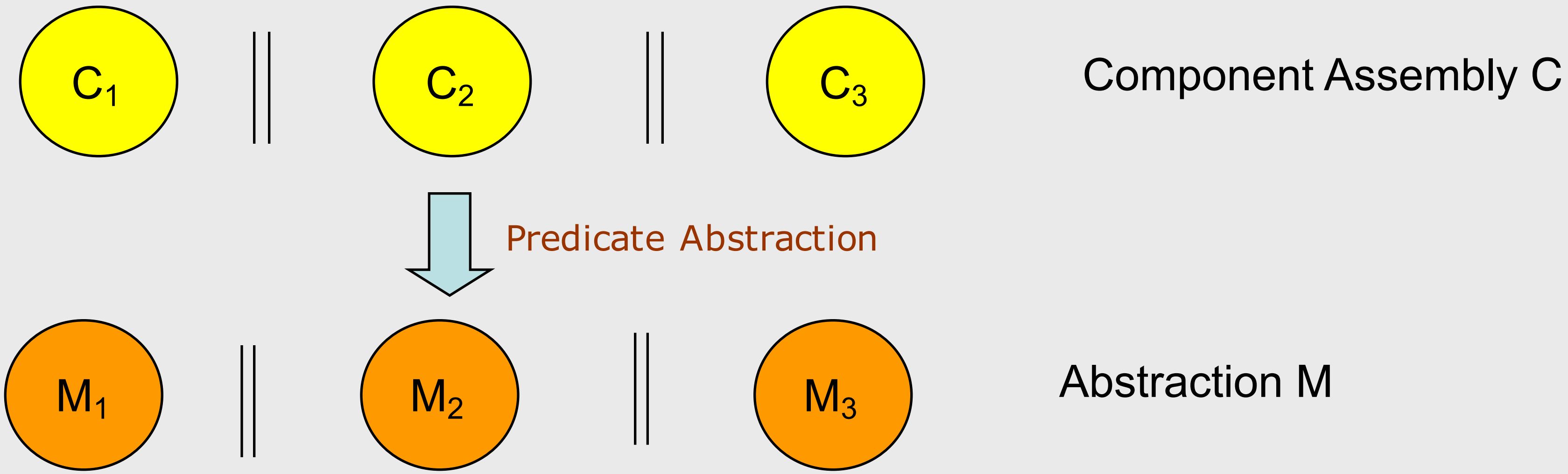
Predicate Abstraction into LKS

- Labeled Kripke Structures
  - <Q,Σ,T,P,L>
- Composition semantics
  - Synchronize on shared actions
- Represents abstractions



Component Assembly

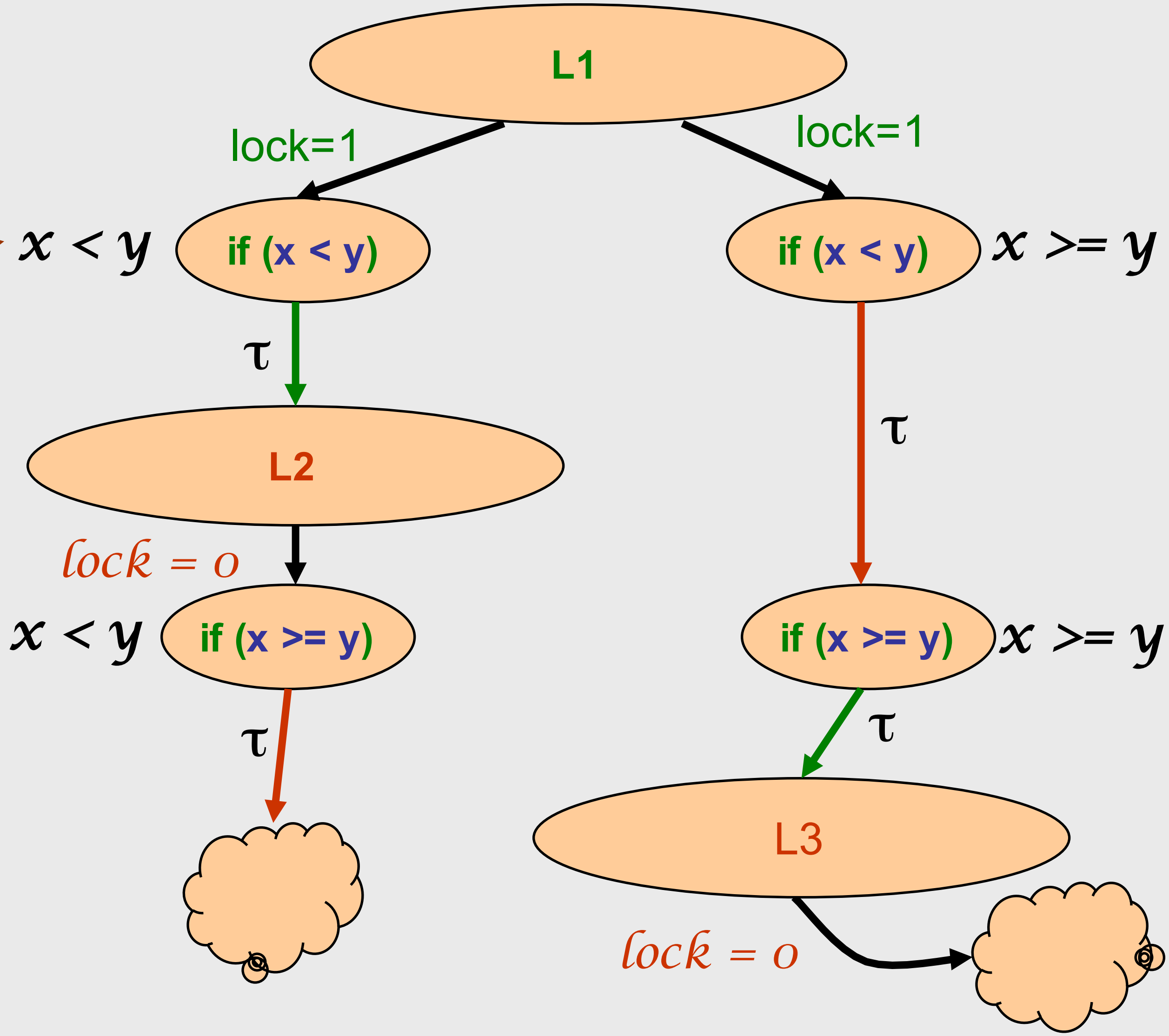
- A set of communicating concurrent C programs
  - No recursion, procedures inlined
- Each component abstracted into a Component LKS
  - Communication between components is abstracted into interface actions



Predicate Abstraction into LKS

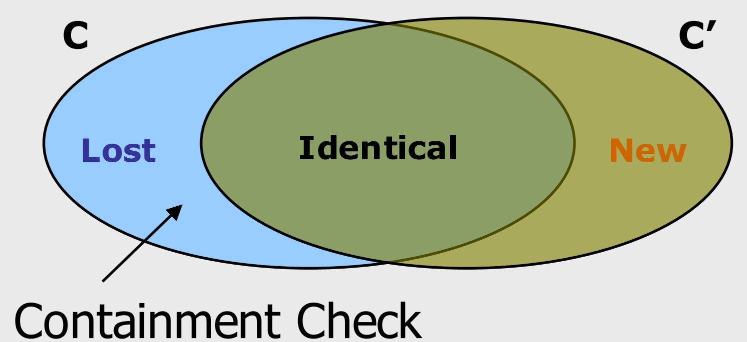
```
void OSSemPend(…) {

  L1: lock = 1;
  if (x < y) {
      L2: lock = 0;
    …
  }
  if (x >= y) {
    …
     L3: lock = 0;
    …
  } else {
    …
  }
}
```

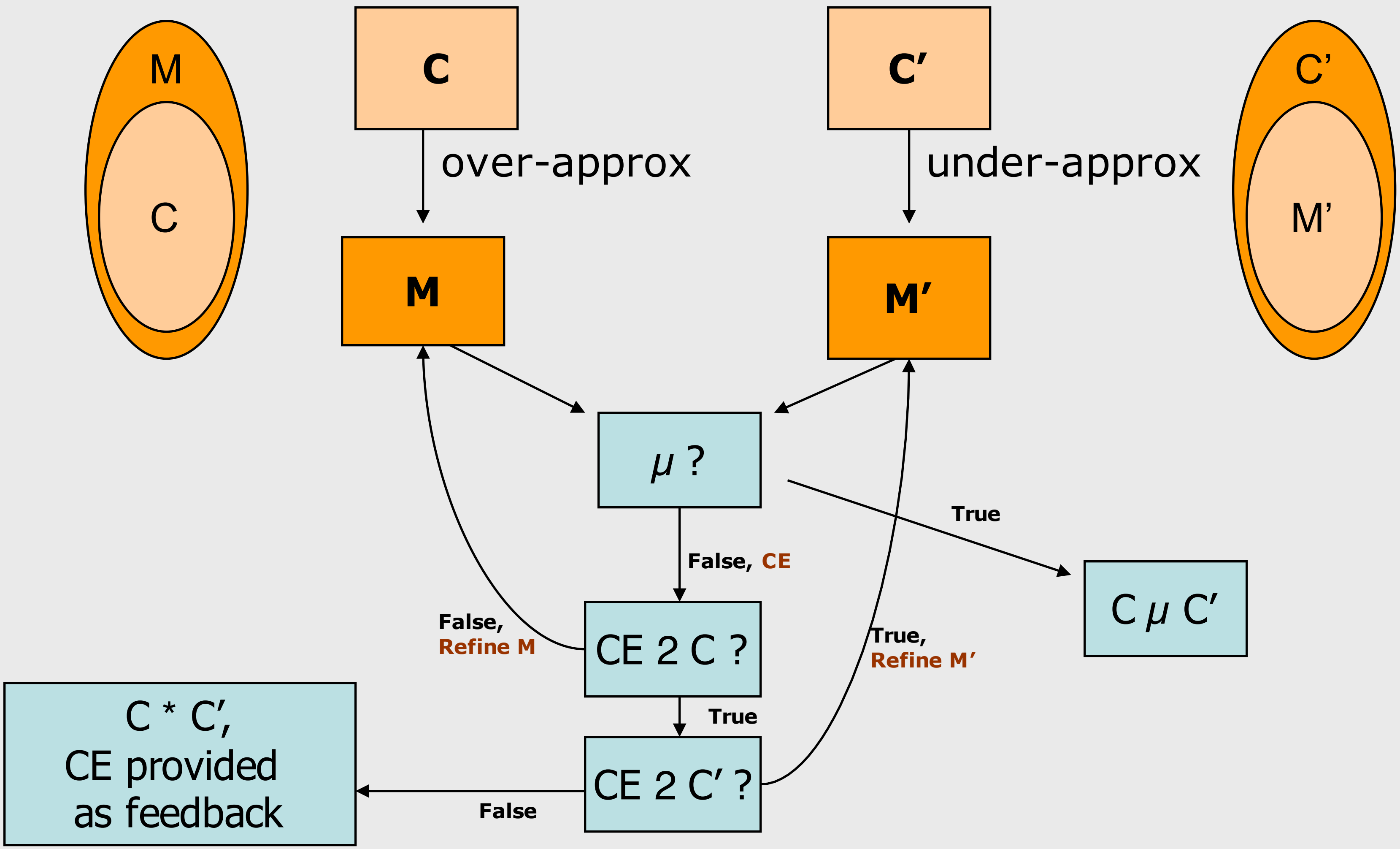


Containment Check

- Goal: Check C μ C′
  - All behaviors retained after upgrade
  - Cannot check directly: need approximations
- Idea: Use both under- and over-approximations
- Solution:
  - Compute M: C μ M
  - Compute M′: M′ μ C′
  - Check for M μ M′



Containment (contd.)

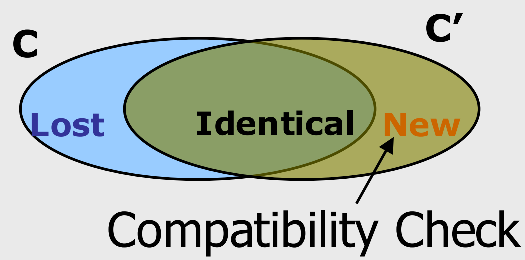


Containment (contd.)

- 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

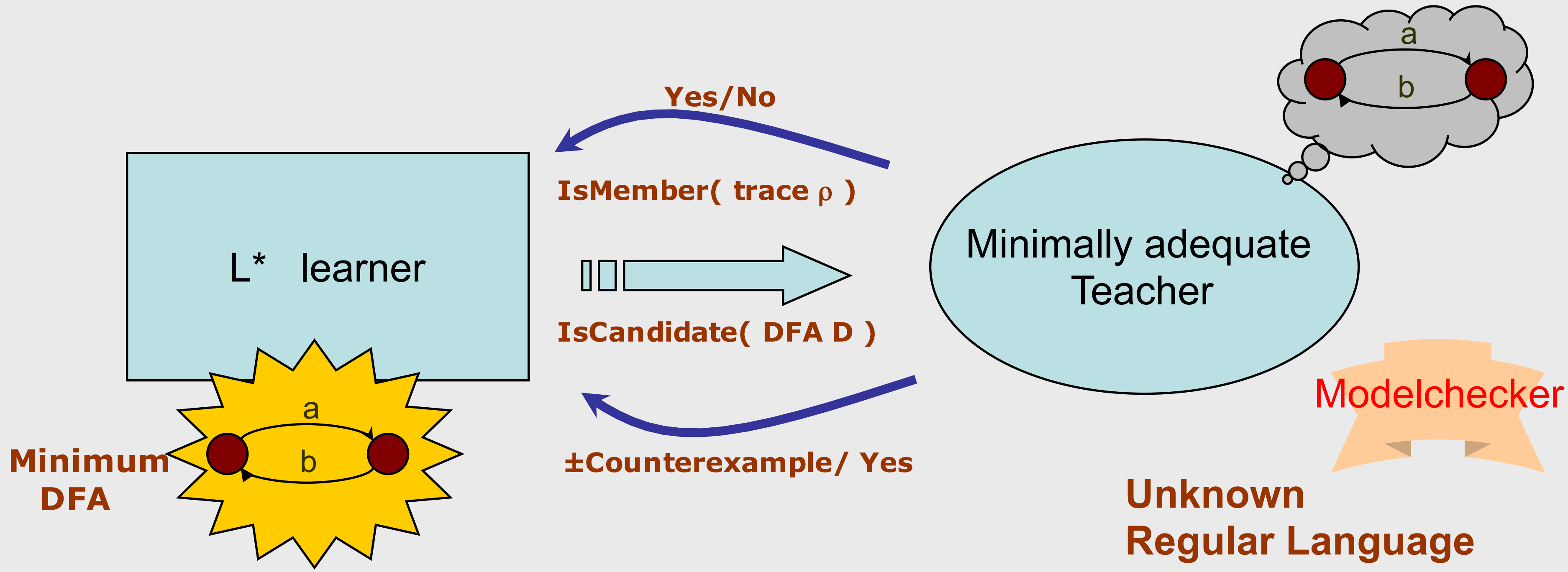
$$\frac{\begin{array}{c} M_1 \,||\, A \;{}^{2}\; P \\ M_2 \;{}^{2}\; A \end{array}}{M_1 \,||\, M_2 \;{}^{2}\; P}$$

AG - Non Circular

- Automatically generate assumption A
  - Cobleigh et. al. at NASA Ames
- Use learning algorithm for regular languages, L*
- Goal: Reuse previous verification results

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 max counterexample

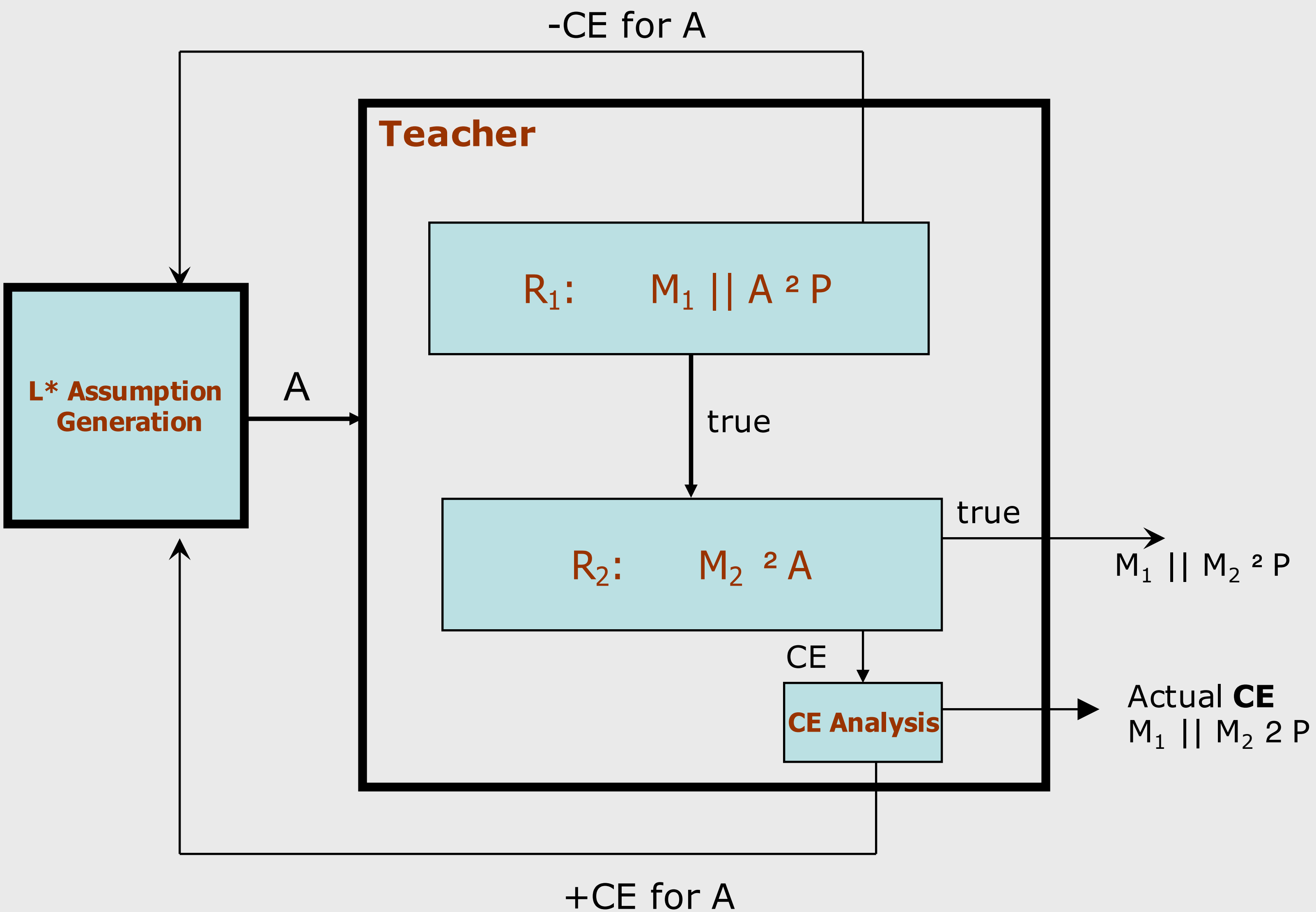


Learning for Verification

- Model checker as a Teacher
  - Possesses information about concrete components
  - Model checks and returns true/counterexample
- Learner builds a model sufficient to verify properties
- Relies on both learner and teacher being efficient

- Finding wide applications
  - Adaptive Model Checking: Groce et al.
  - Automated Assume-Guarantee Reasoning: Cobleigh et al.
  - Synthesize Interface Specifications for Java Programs: Alur et al.
  - Regular Model Checking: Vardhan et al., Habermehl et al.

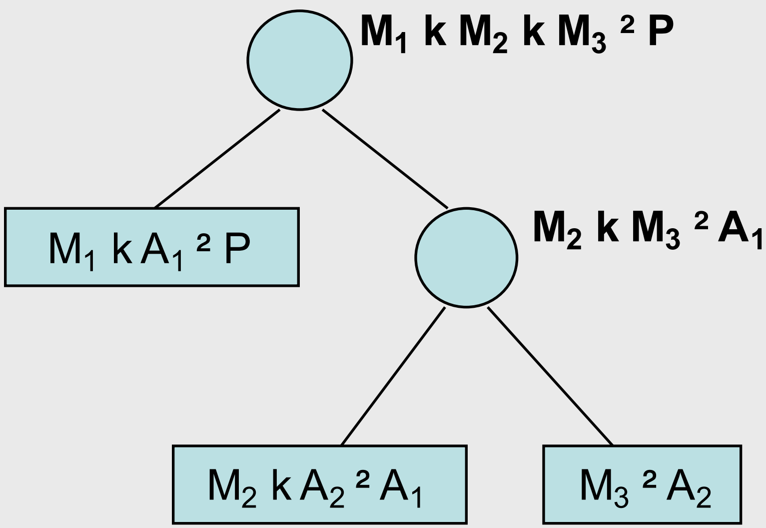
Compatibility Check



Handling Multiple Components

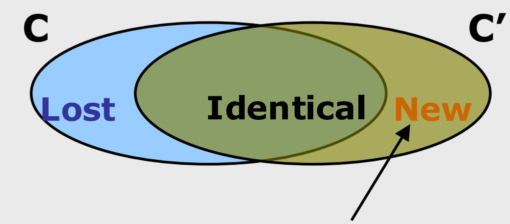
- AG-NC is recursive
  - (Cobleigh et al.)

$$\begin{array}{ll} R_1: & M_1 \,||\, A \text{ ² } P \\ R_2: & M_2 \text{ ² } A \\ \hline & M_1 \,||\, M_2 \text{ ² } P \end{array}$$

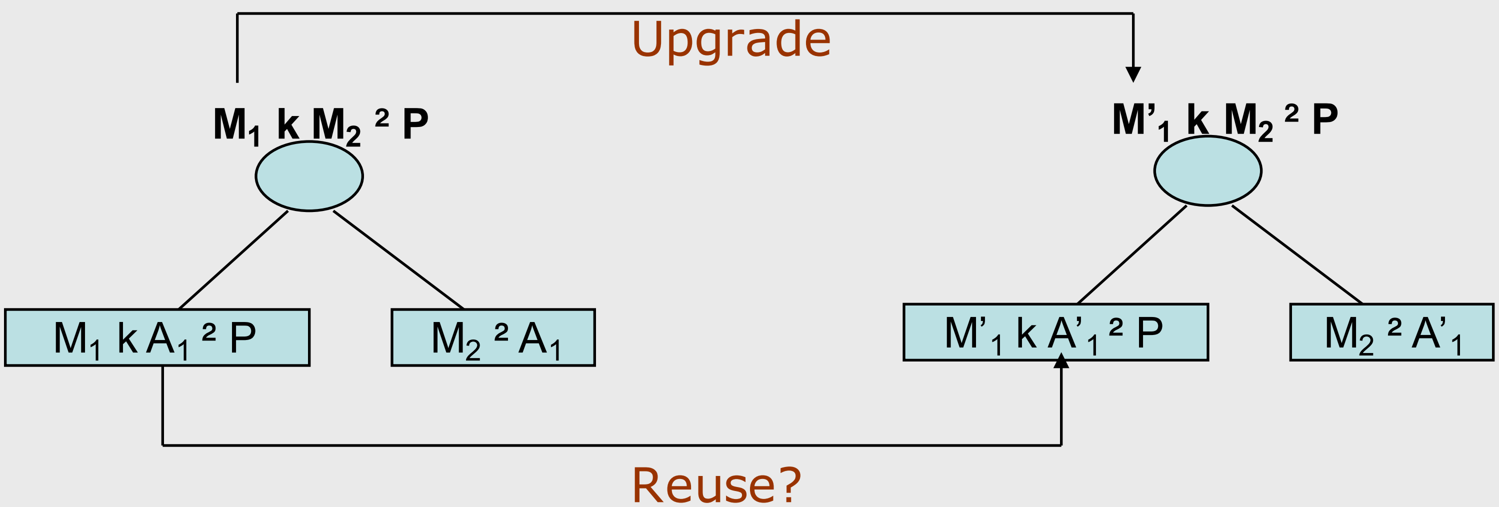


- Each A_i computed by a separate L* instantiation

Compatibility of Upgrades



- Suppose assumptions are available from the old assembly
- Dynamic AG: Reuse previous verification results



- Can we reuse previous assumptions directly?
  - NO: upgrades may change the unknown U to be learned
- Requires Dynamic L*

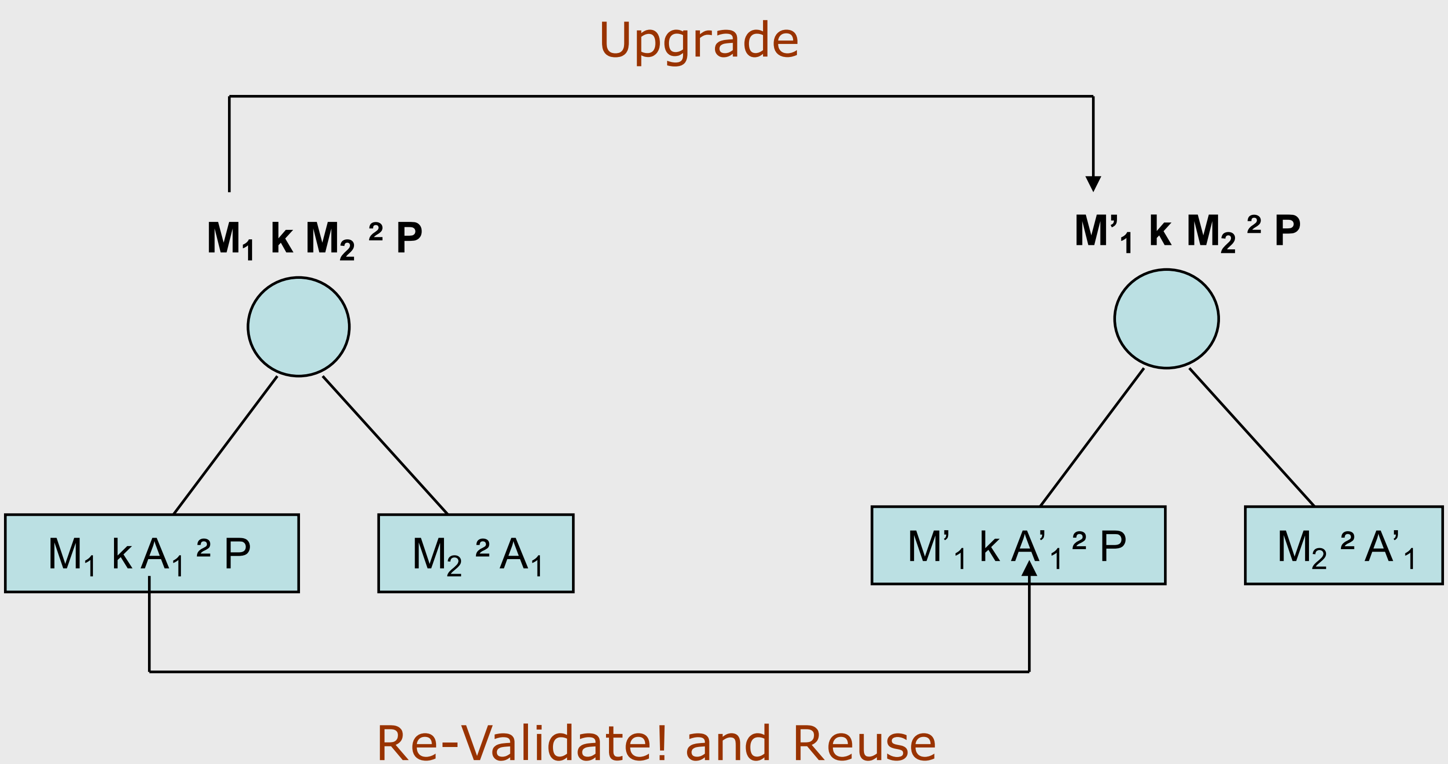
Dynamic L*

- Learn DFA A corresponding to U
- Unknown language U changes to U’
- Goal: Continue learning from previous model A
- Central Idea: Re-validate A to A’ which agrees with U’

Dynamic L*

- L* maintains a table data-structure to store samples
- Definition: Valid Tables
  - All table entries agree with U
- Theorem
  - L* terminates with any valid observation table, OT
- When U changes to U',
  - Suppose the last candidate w.r.t. U is A
  - Re-validate OT of A w.r.t. U'
  - Obtain A' from OT'
  - Continue learning from A'

Dynamic AG


Upgrade
M_1 k M_2 ² P
M'_1 k M_2 ² P
M_1 k A_1 ² P
M_2 ² A_1
M'_1 k A'_1 ² P
M_2 ² A'_1
Re-Validate! and Reuse

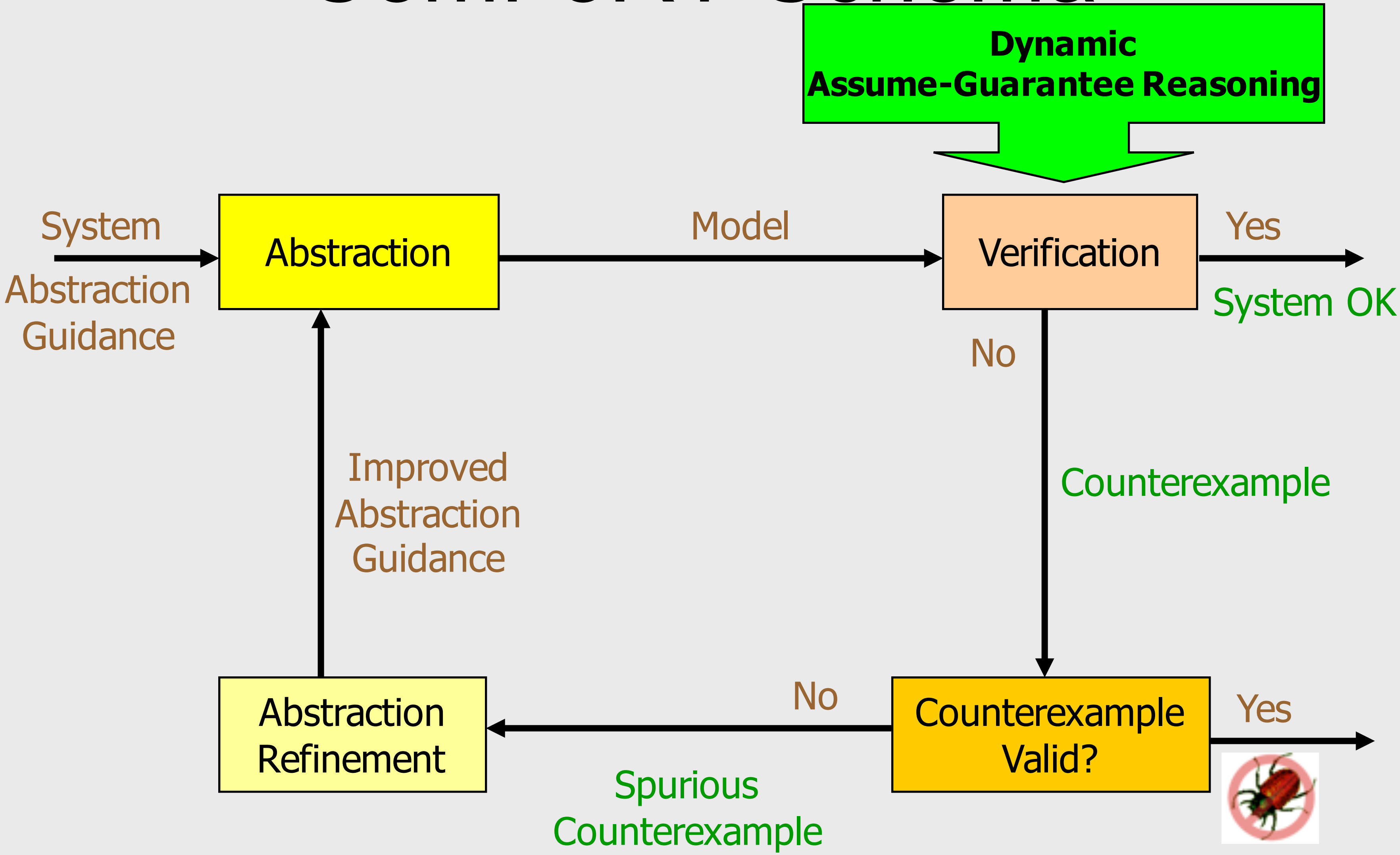
Implementation

- Comfort framework – explicit model checker
- Industrial benchmark
  - ABB Inter-process Communication (IPC) 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

Experimental Results

Upgrade# (Property)	#Mem Queries	T_{orig} (msec)	T_{ug} (msec)
Ipc_1 (P_1)	279	2260	13
Ipc_1 (P_2)	308	1694	14
Ipc_2 (P_1)	358	3286	17
Ipc_2 (P_2)	232	805	10
Ipc_3 (P_1)	363	3624	17
Ipc_3 (P_2)	258	1649	14
Ipc_4 (P_1)	355	1102	24

ComFoRT Schema



Conclusion

- Automated Substitutability Checking
  - Containment and Compatibility
  - Reuses previous verification results
  - Handles multiple upgrades
  - Built upon CEGAR framework
- Implementation
  - ComFoRT framework
  - Promising results on an industrial example

Future Directions

- Symbolic analysis, i.e., using SATABS
- Assume-Guarantee for Liveness
- Other AG Rules, e.g., Circular
- Combining static analysis with dynamic testing for facilitate abstraction and learning

Ph.D. position is open

- New EU project on verification of evolving networked software
  - Collaboration with IBM, ABB, VTT, Uni Milano and Oxford