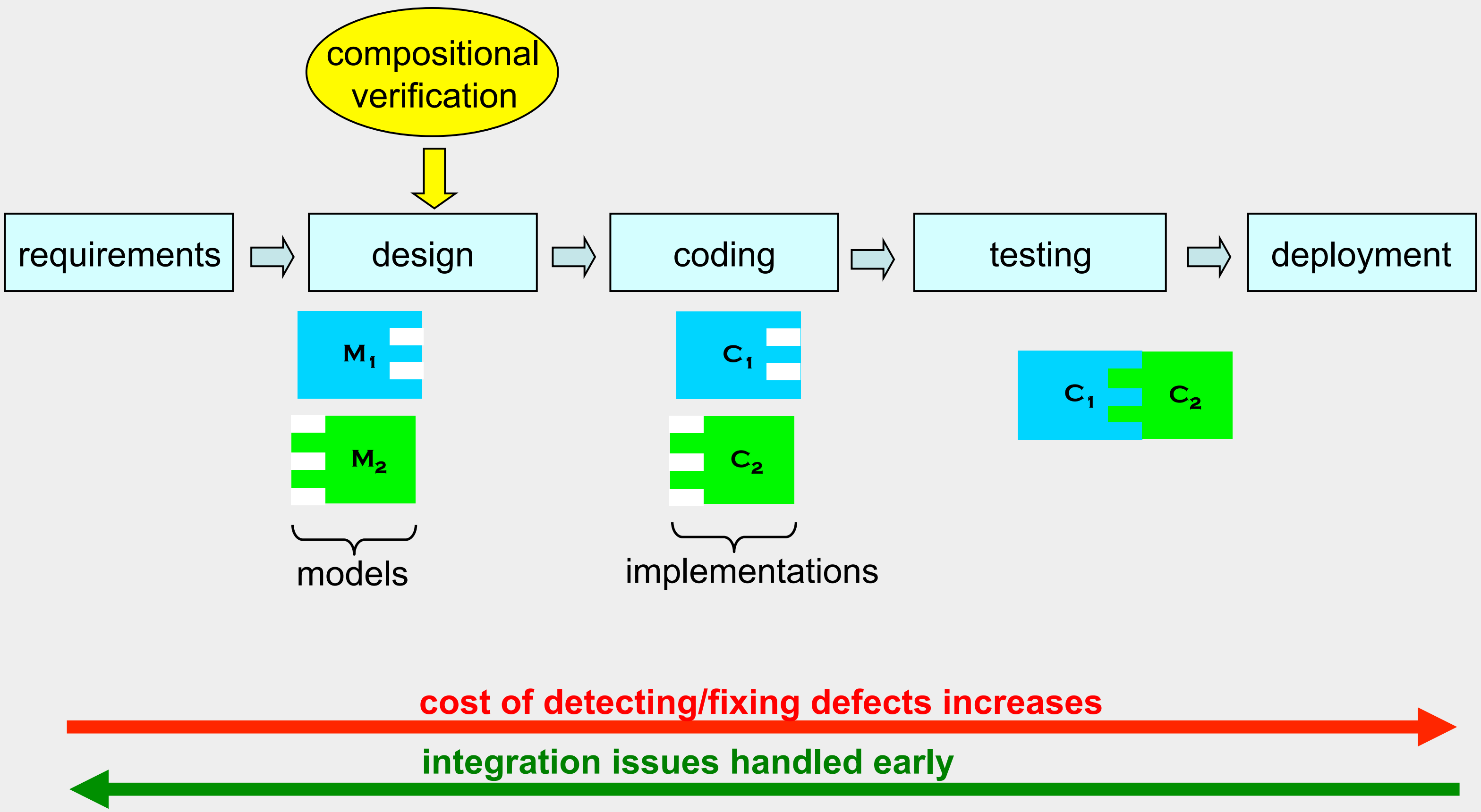
Automated Component-Based Verification

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center

component-based development

- component-based verification, for increased scalability, at design level
- early detection of integration problems
- use design level artifacts to improve/aid coding and testing



structure

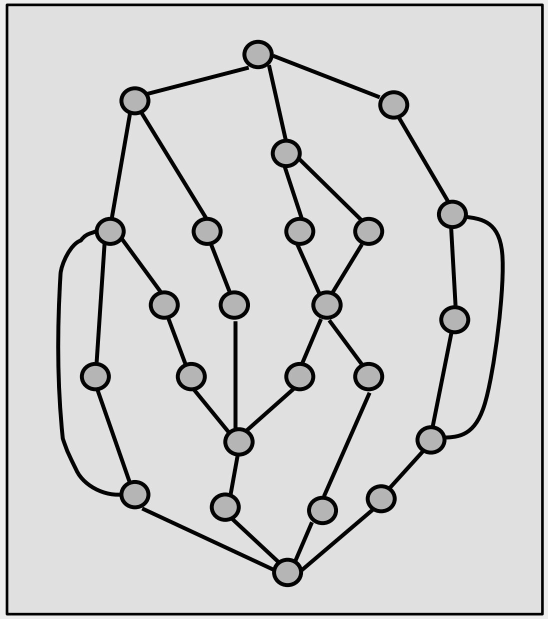
part 1 (Dimitra) assume-guarantee reasoning computing assumptions learning assumptions discussion	part 2 (Corina) multiple components alphabet refinement case studies discussion
lunch	
part 3 (Dimitra) component interfaces compositional JavaPathfinfer examples discussion	part 4 (Corina) reasoning about code abstraction related work conclusion

model checking

program / model

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

model checker



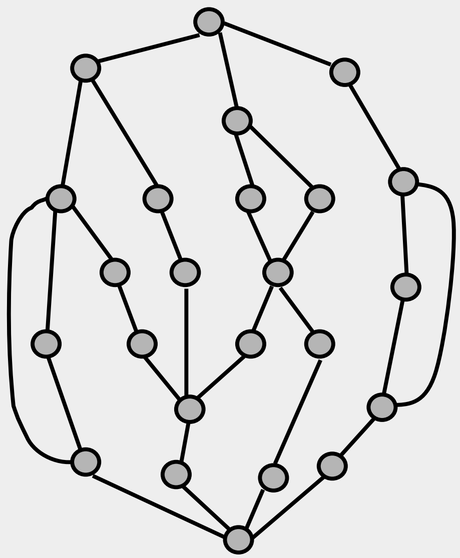
property

always(ϕ **or** ψ)

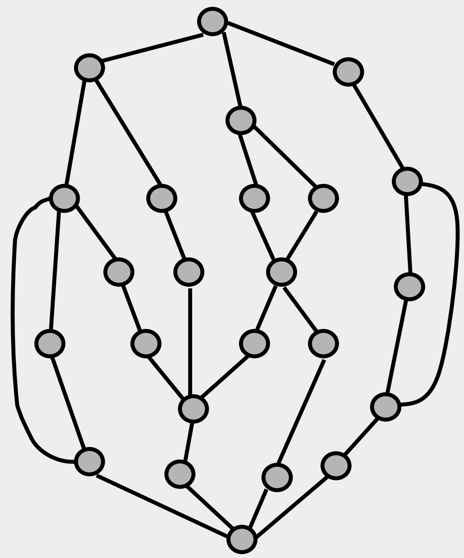
YES (property holds)

NO + counterexample:
(provides a violating execution)

model checking vs. testing



testing



model checking

compositional
verification

collaborators

Prof. Howard Barringer (Univ. of Manchester)

Colin Blundell (Upenn, IBM Research)

Jamieson Cobleigh (UMass, MathWorks)

Michael Emmi (UCLA)

Mihaela Gheorgiu (Univ. of Toronto, JPL)

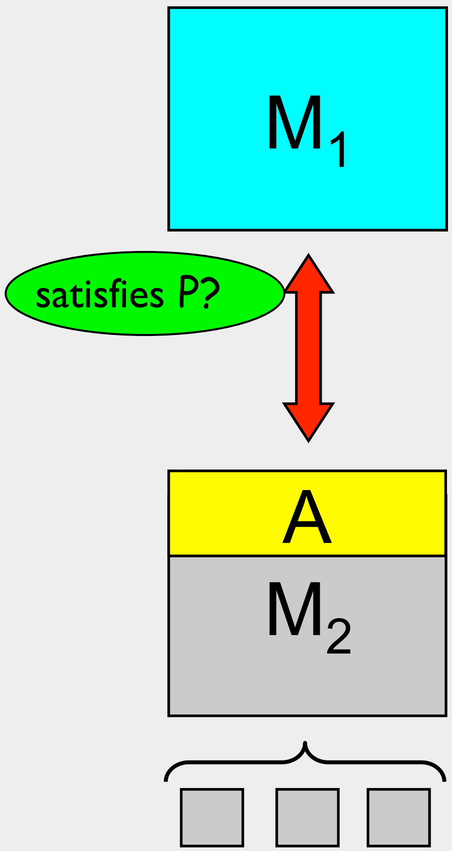
Chang-Seo Park (UC Berkeley)

Suzette Person (Univ. of Nebraska, NASA Langley)

Rishabh Singh (MIT)

compositional verification

does system made up of M_1 and M_2 satisfy property P?



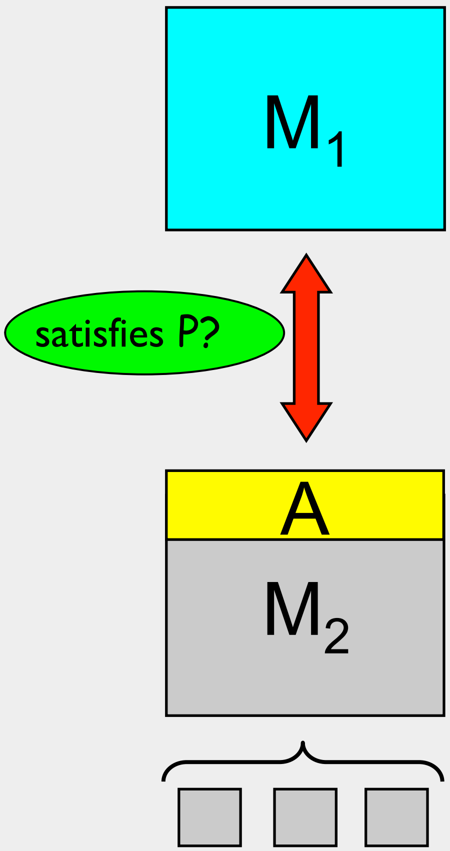
- check P on entire system: too many states!
- use system's natural decomposition into components to break-up the verification task
- check components in isolation:

does M_1 satisfy P?


"when we try to pick out anything by itself, we find
it hitched to everything else in the universe"
John Muir

assume-guarantee reasoning

introduces assumptions / reasons about triples:



$\langle A \rangle\ M\ \langle P \rangle$ is *true* if whenever M is part of a system that satisfies A, then the system must also guarantee P

simplest assume-guarantee rule (ASYM):

$$\frac{\begin{array}{ll} 1. & \langle A \rangle\ M_1\ \langle P \rangle \\ 2. & \langle \mathit{true} \rangle\ M_2\ \langle A \rangle \end{array}}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

"discharge" the assumption

examples of assumptions

- will not invoke "close" on a file if "open" has not previously been invoked
- accesses to shared variable "X" must be protected by lock "L"
- (rover executive) whenever thread "A" reads variable "V", no other thread can read "V" before thread "A" clears it first
- (spacecraft flight phases) a docking maneuver can only be invoked if the launch abort system has previously been jettisoned from the spacecraft

assume-guarantee reasoning

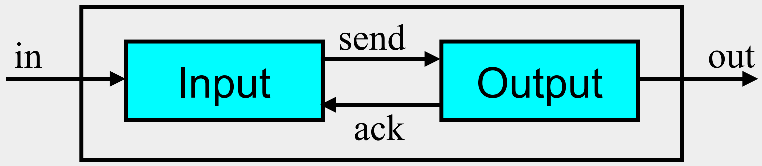
how do we come up
with the assumption?

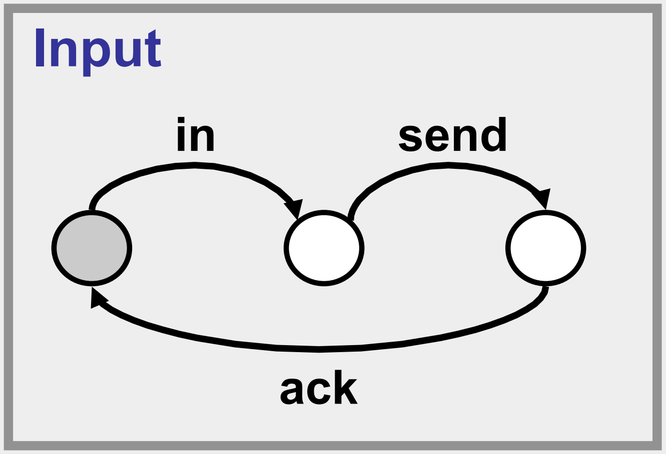
formalisms

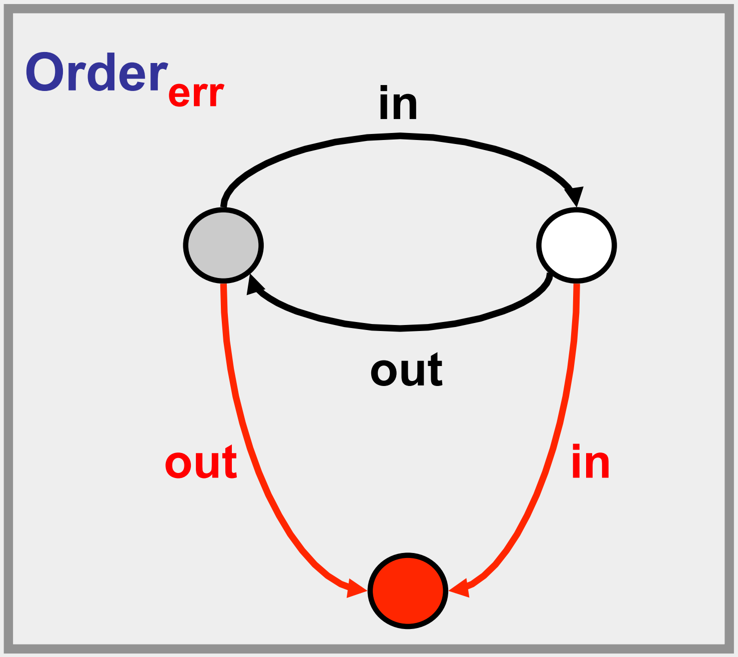
- components modeled as finite state machines (FSM)
  - FSMs assembled with parallel composition operator "||"
    - synchronizes shared actions, interleaves remaining actions
- a safety property P is a FSM
  - P describes all legal behaviors in terms of its alphabet
  - P_{err} – complement of P
    - determinize & complete P with an "error" state;
    - bad behaviors lead to error
  - component M satisfies P iff error state unreachable in $(M \mid\mid P_{err})$
- assume-guarantee reasoning
  - assumptions and guarantees are FSMs
  - $\langle A \rangle$ M $\langle P \rangle$ holds iff error state unreachable in $(A \mid\mid M \mid\mid P_{err})$

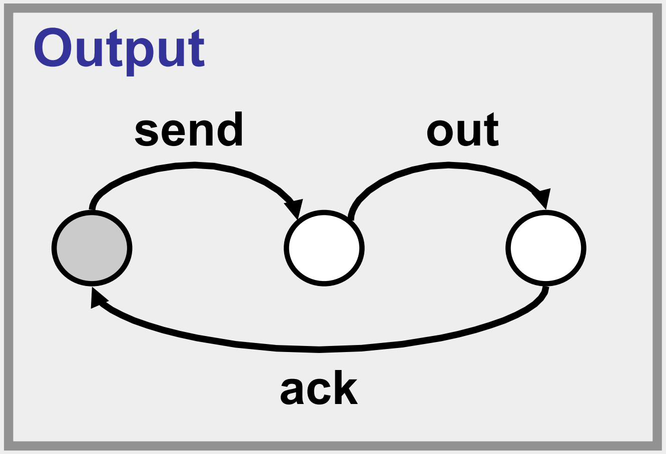
example

require in and out to alternate (property Order)

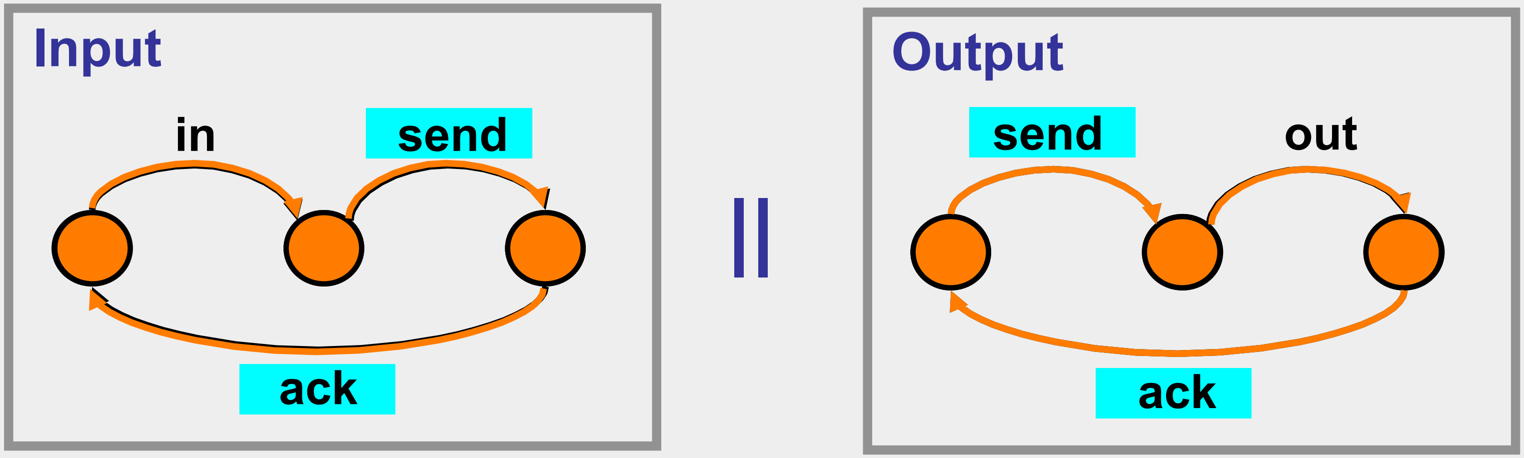




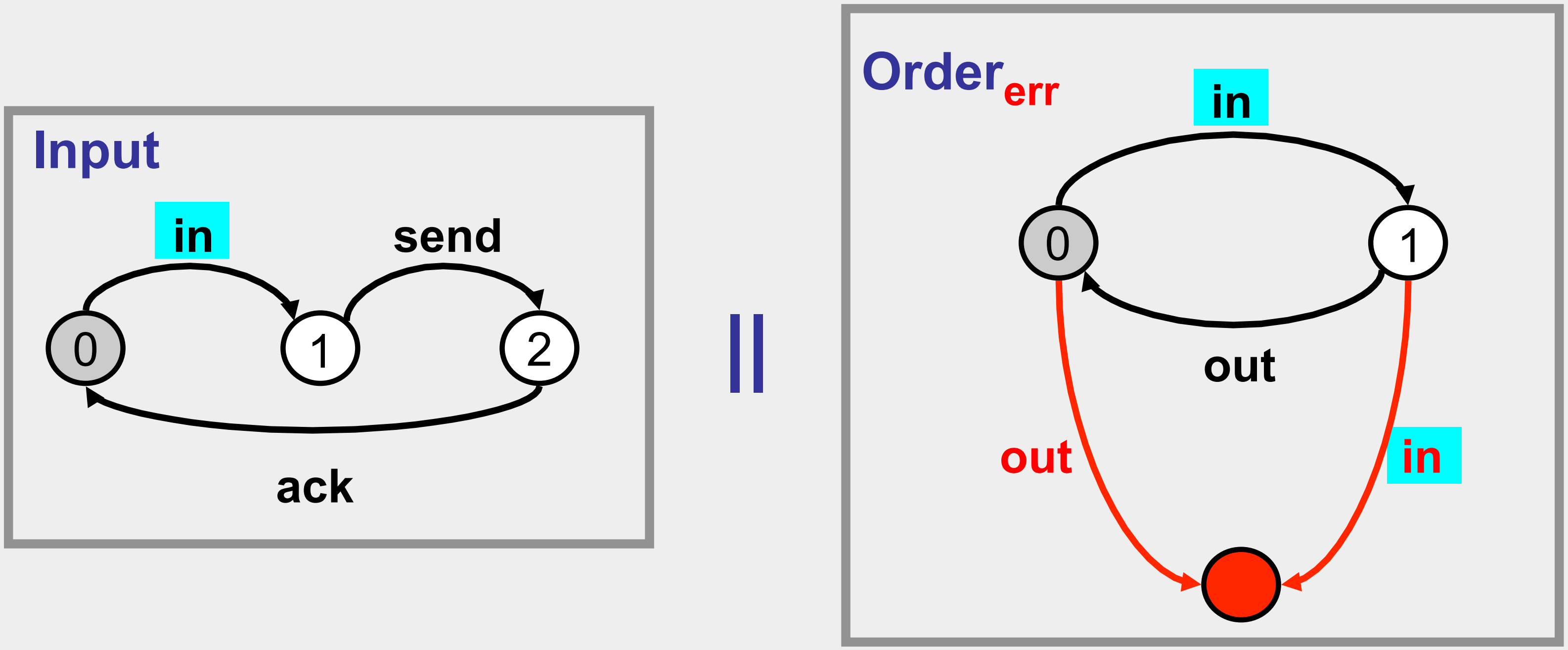




parallel composition



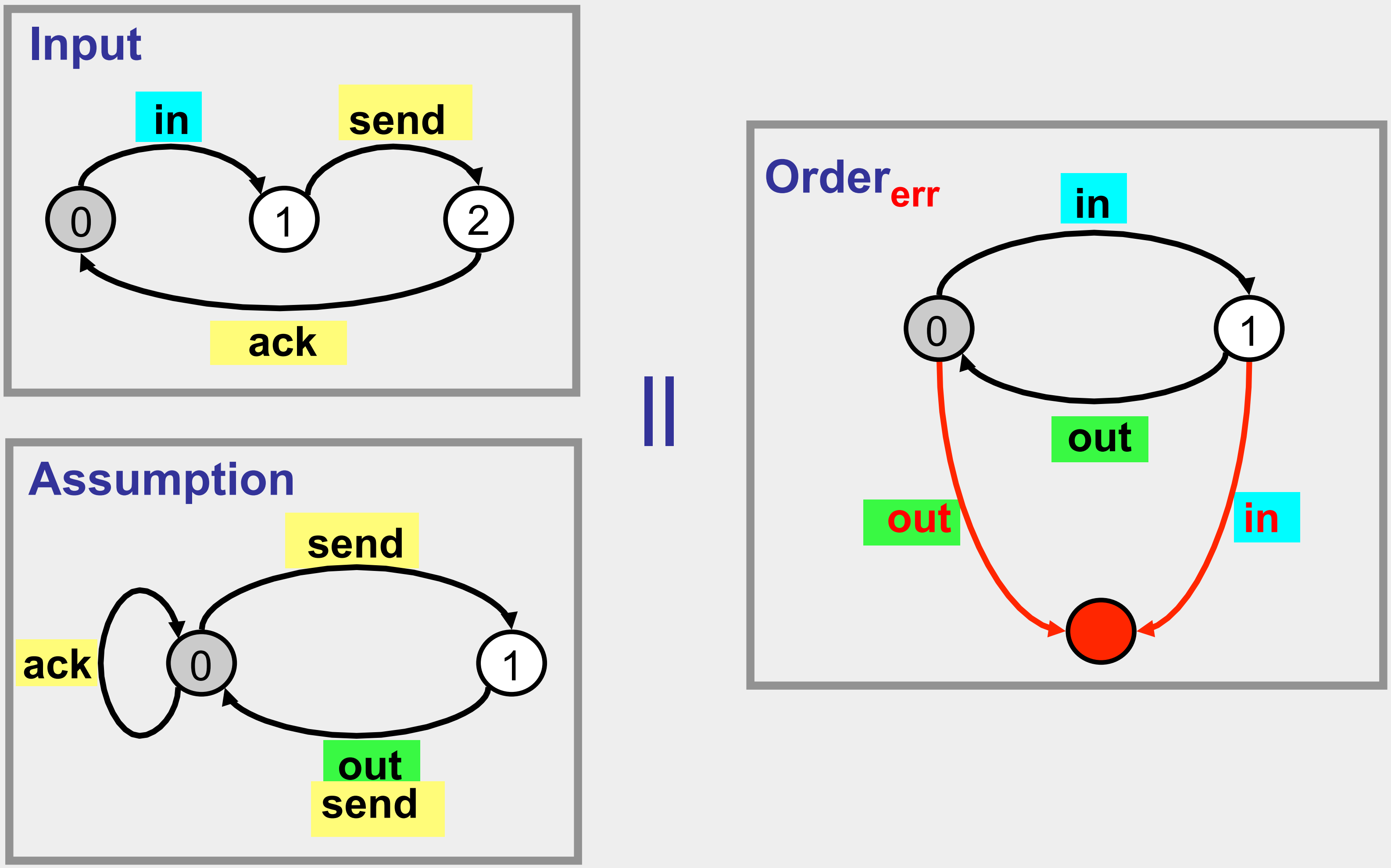
property satisfaction



crex. 1: (I_0, O_0) out (I_0, O_{error})

crex. 2: (I_0, O_0) in (I_1, O_1) send (I_2, O_1) out (I_2, O_0) out (I_2, O_{error})

assume-guarantee reasoning



crex 1: (I_0, A_0, O_0) out ✗

crex 2: (I_0, A_0, O_0) in (I_1, A_0, O_1) send (I_2, A_1, O_1) out (I_2, A_0, O_0) out ✗

the weakest assumption

- given component M, property P, and the interface of M with its environment, generate the weakest environment assumption WA such that: $\langle WA \rangle$ M $\langle P \rangle$ holds

- weakest means that for all environments E:

$$\langle \mathit{true} \rangle\ M \parallel E\ \langle P \rangle \text{ IFF } \langle \mathit{true} \rangle\ E\ \langle WA \rangle$$

weakest assumption in AG reasoning

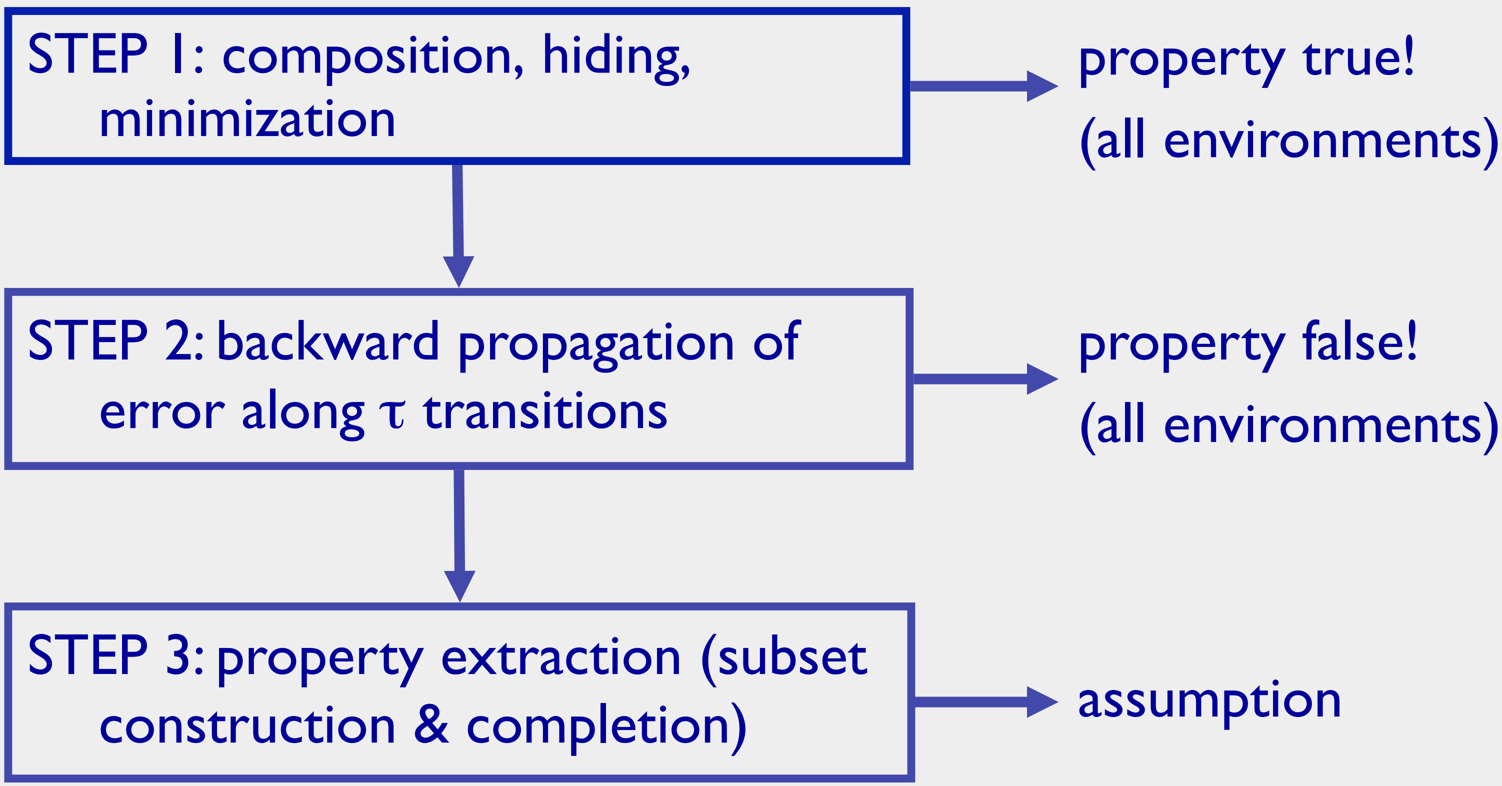
$$\frac{\begin{array}{l} 1.\quad \langle A\rangle\ M_1\ \langle P\rangle \\ 2.\ \langle \mathit{true}\rangle\ M_2\ \langle A\rangle \end{array}}{\langle \mathit{true}\rangle\ M_1 \,||\, M_2\ \langle P\rangle}$$

weakest assumption makes rule complete

for all E, $\langle \mathit{true}\rangle$ M || E $\langle P\rangle$ IFF $\langle \mathit{true}\rangle$ E $\langle WA\rangle$

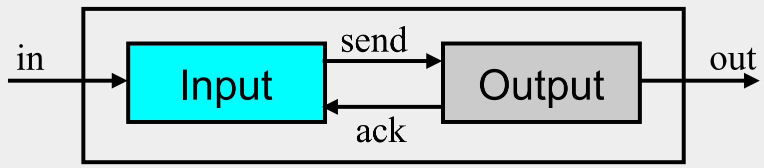
$\langle WA\rangle$ M_1 $\langle P\rangle$ holds (WA could be *false*)
$\langle \mathit{true}\rangle$ M_2 $\langle WA\rangle$ holds implies $\langle \mathit{true}\rangle$ $M_1 \,||\, M_2$ $\langle P\rangle$ holds
$\langle \mathit{true}\rangle$ M_2 $\langle WA\rangle$ not holds implies $\langle \mathit{true}\rangle$ $M_1 \,||\, M_2$ $\langle P\rangle$ not holds

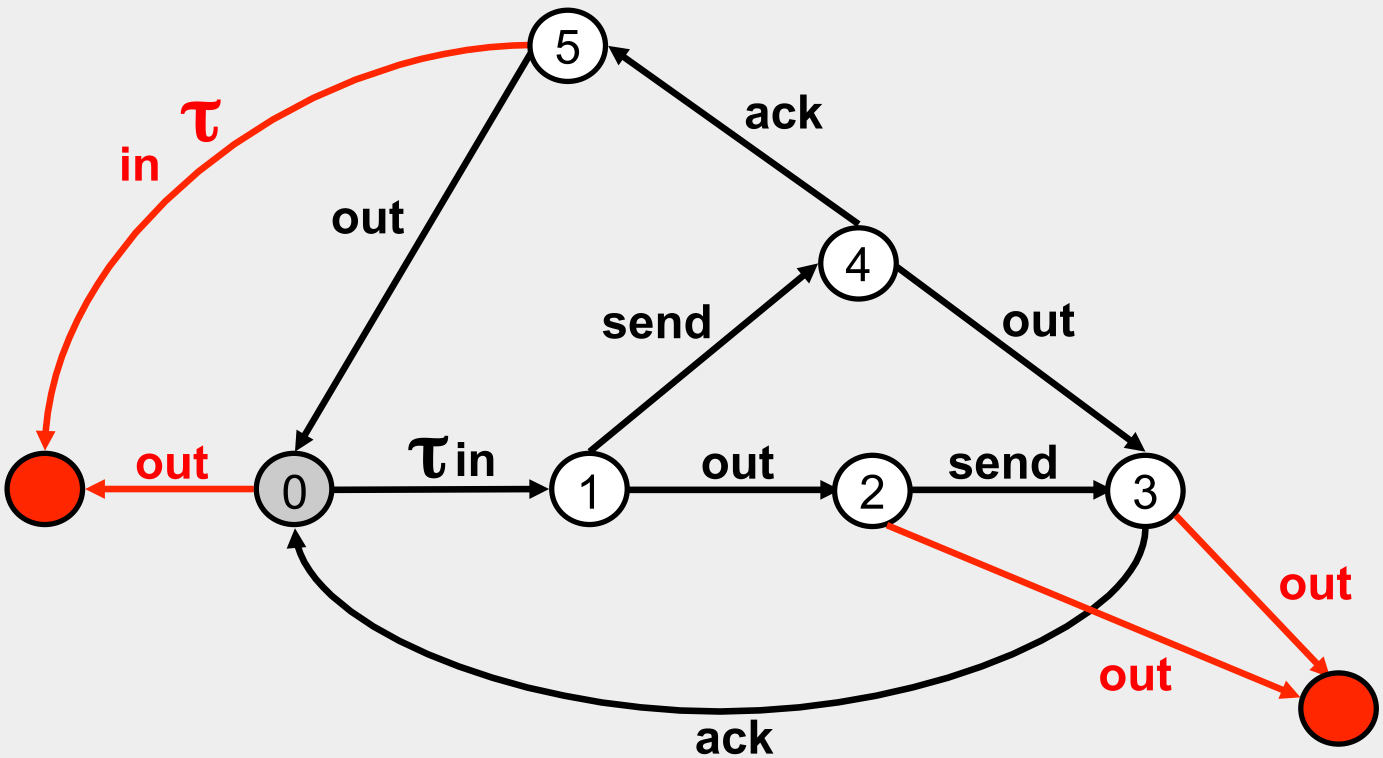
assumption generation [ASE'02]



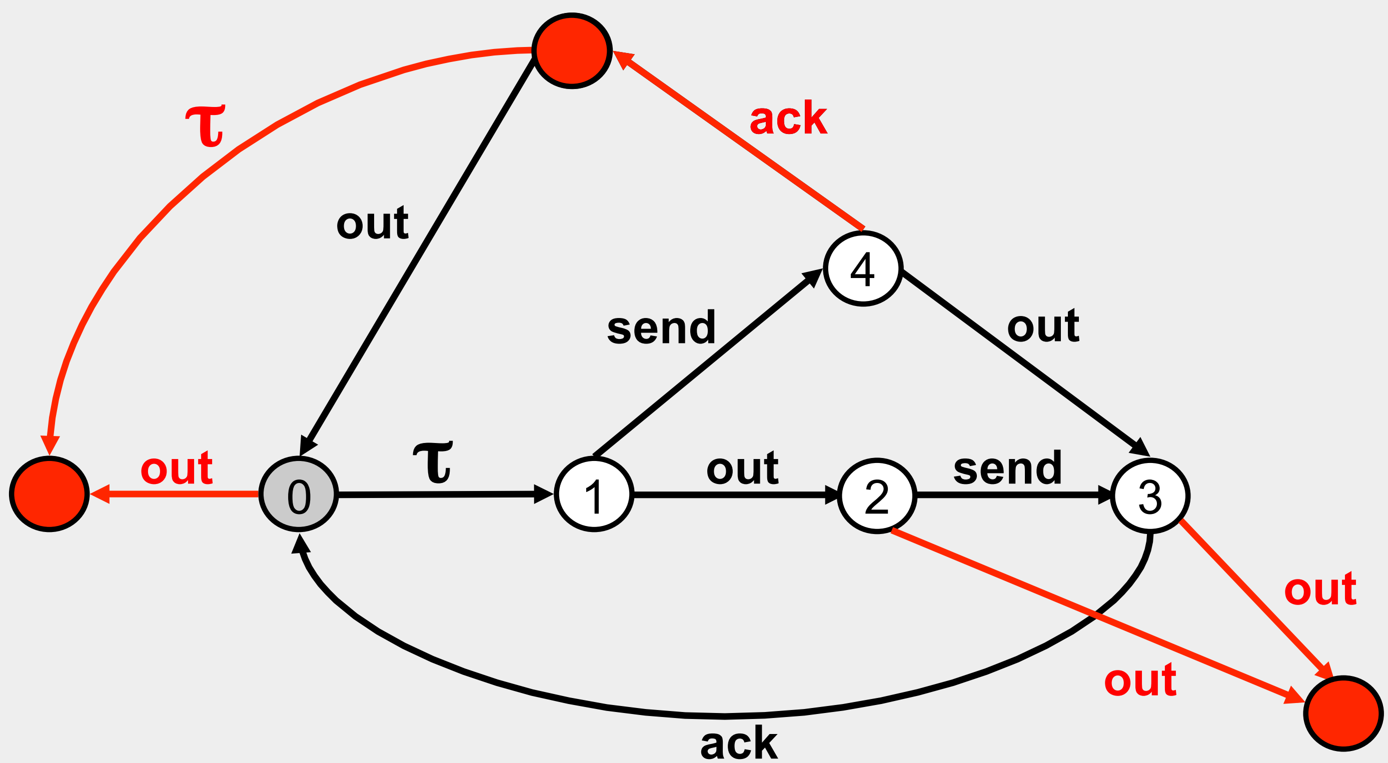
step 1: composition & hiding

Input || Order$_{err}$ \ {in}

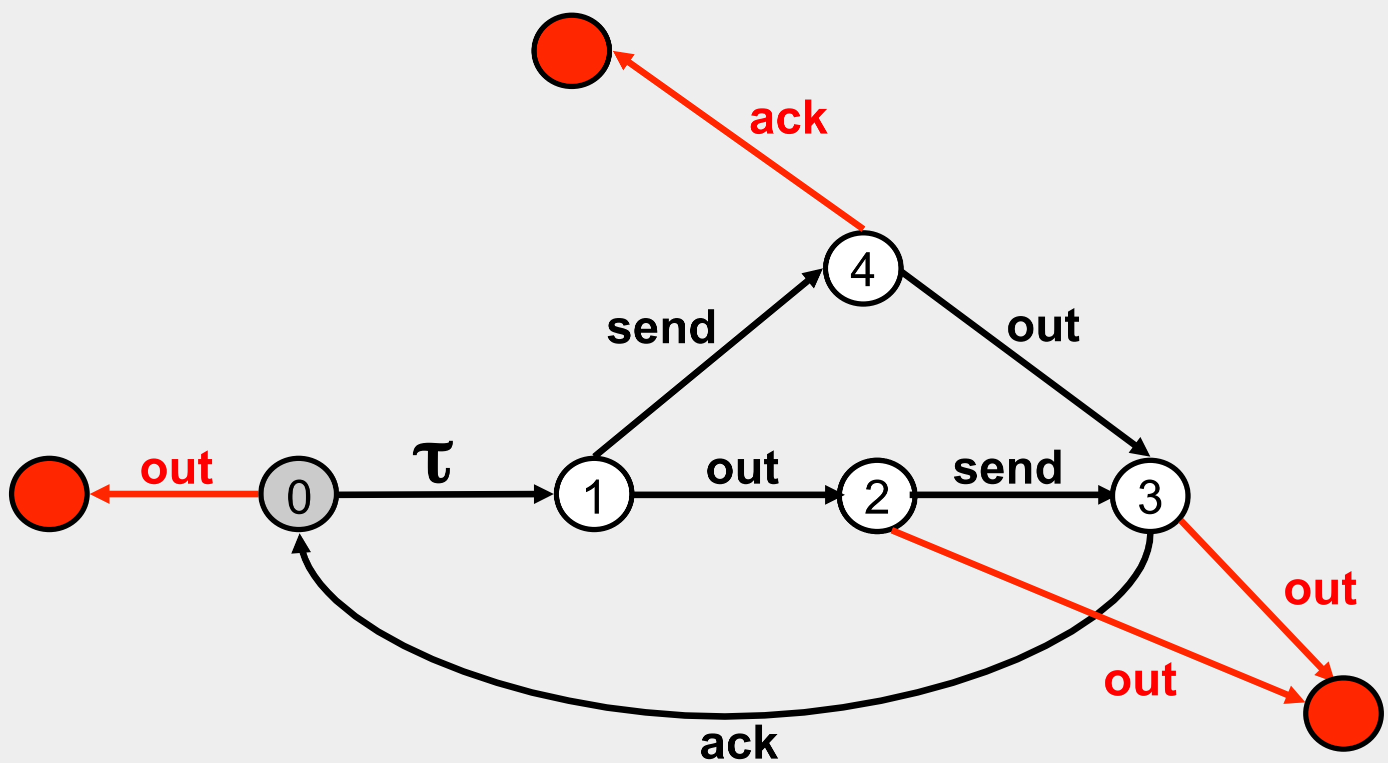




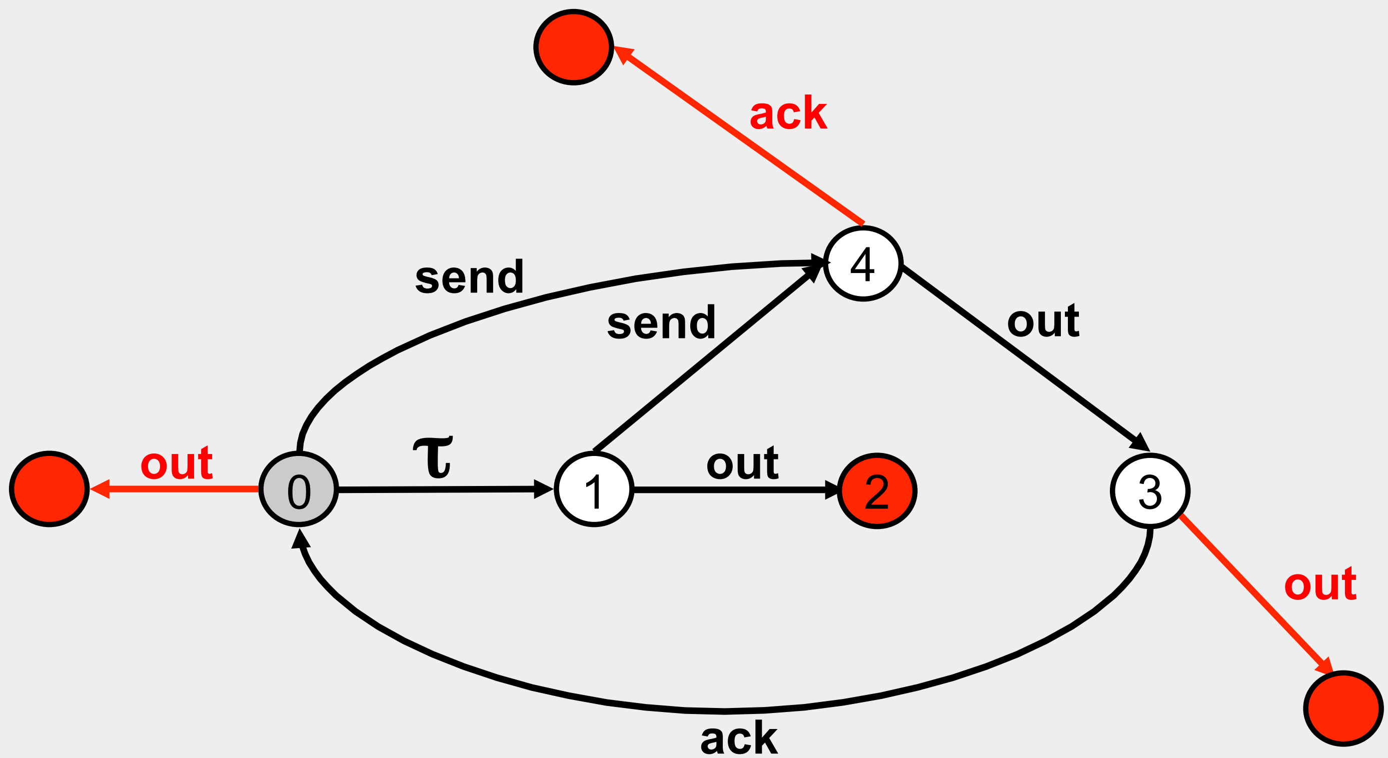
step 2: error propagation



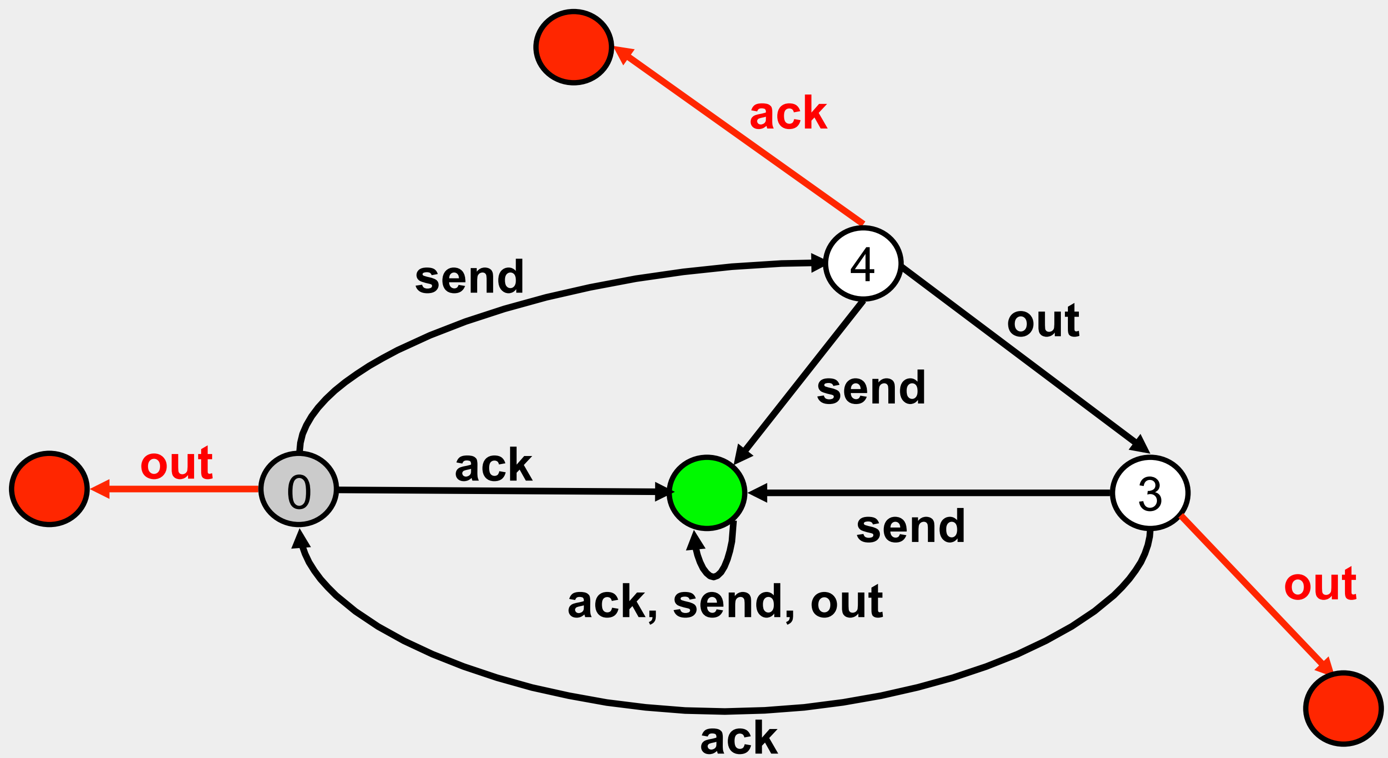
step 3: subset construction



step 3: subset construction



step 3: property construction



weakest assumption in AG reasoning

$$\frac{\begin{array}{l}1.\ \langle A\rangle\ M_1\ \langle P\rangle\\ 2.\ \langle \mathit{true}\rangle\ M_2\ \langle A\rangle\end{array}}{\langle \mathit{true}\rangle\ M_1 \mid\mid M_2\ \langle P\rangle}$$

weakest assumption makes rule complete

$\langle WA\rangle\ M_1\ \langle P\rangle$ holds (WA could be *false*)
$\langle \mathit{true}\rangle\ M_2\ \langle WA\rangle$ holds implies $\langle \mathit{true}\rangle\ M_1 \mid\mid M_2\ \langle P\rangle$ holds
$\langle \mathit{true}\rangle\ M_2\ \langle WA\rangle$ not holds implies $\langle \mathit{true}\rangle\ M_1 \mid\mid M_2\ \langle P\rangle$ not holds

learning assumptions

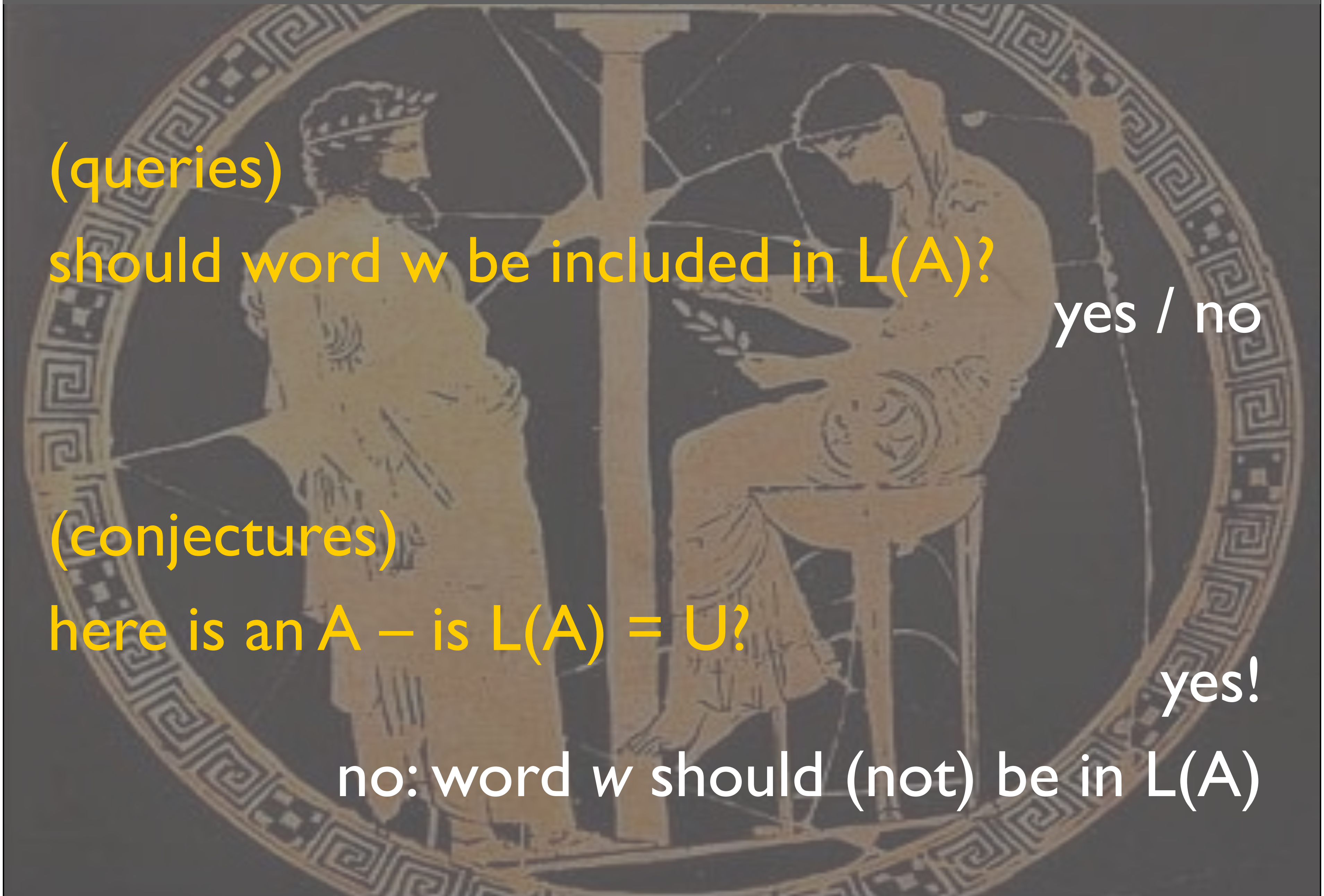
iterative solution +
intermediate results

L* learns unknown regular language U (over alphabet Σ) and produces minimal DFA A such that L(A) = U

(L originally proposed by Angluin)*

L* learner

the oracle



(queries)

should word w be included in L(A)?

yes / no

(conjectures)

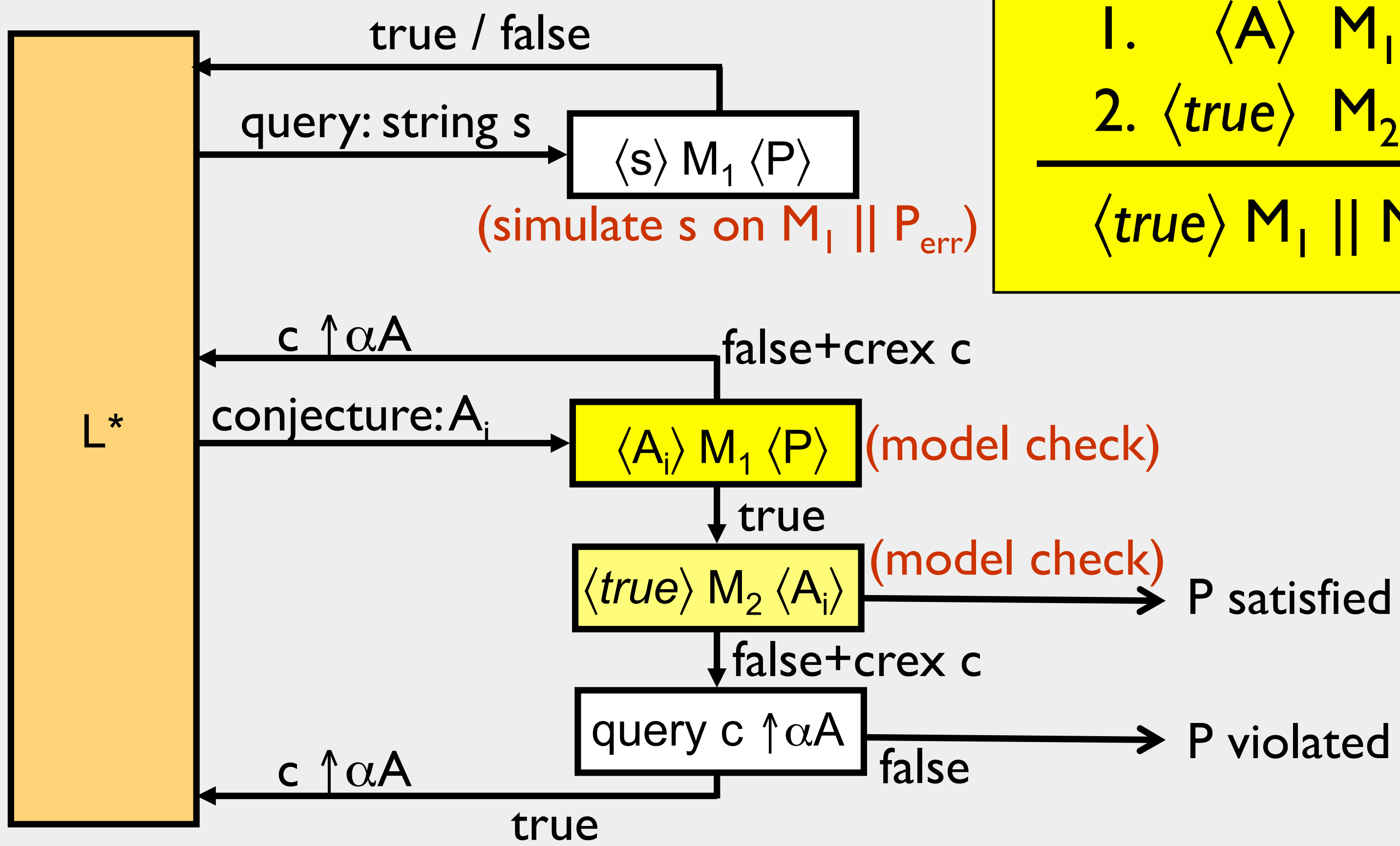
here is an A – is L(A) = U?

yes!

no: word *w* should (not) be in L(A)

oracle for WA in assume-guarantee reasoning

$$\frac{\begin{array}{l}1.\ \langle A\rangle\ M_1\ \langle P\rangle \\ 2.\ \langle \mathit{true}\rangle\ M_2\ \langle A\rangle\end{array}}{\langle \mathit{true}\rangle\ M_1 \,||\, M_2\ \langle P\rangle}$$



$\langle WA\rangle\ M_1\ \langle P\rangle$ holds

$\langle \mathit{true}\rangle\ M_2\ \langle WA\rangle$ holds implies $\langle \mathit{true}\rangle\ M_1 \,||\, M_2\ \langle P\rangle$ holds

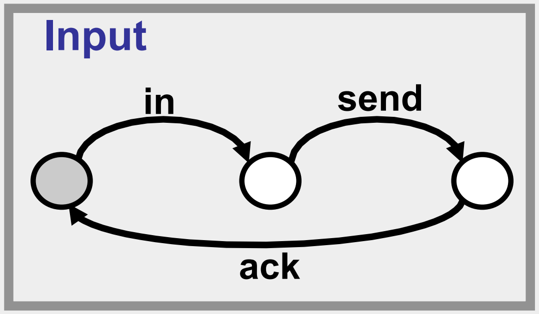
$\langle \mathit{true}\rangle\ M_2\ \langle WA\rangle$ does not hold implies $\langle \mathit{true}\rangle\ M_1 \,||\, M_2\ \langle P\rangle$ does not hold

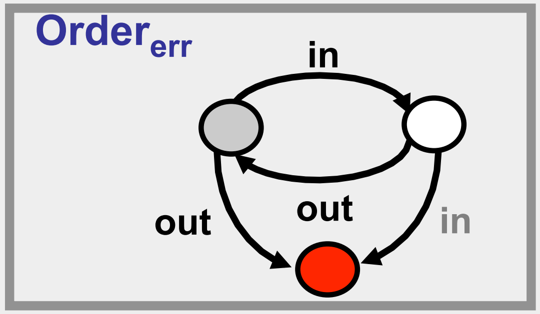
characteristics

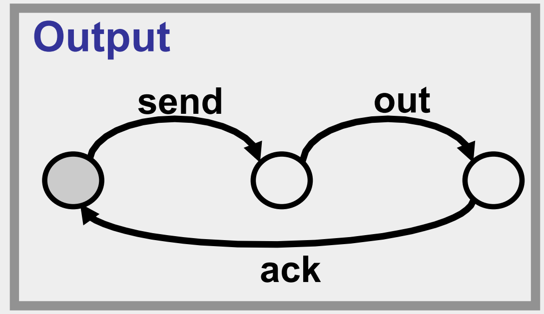
assumptions conjectured by L* are not comparable semantically

- terminates with *minimal* automaton A for U
- generates DFA candidates A_i: $|A_1| < |A_2| < \ldots < |A|$
- produces at most n candidates, where $n = |A|$
- # queries: $O(kn^2 + n \log m)$,
  - m is size of largest counterexample, k is size of alphabet
- for assume-guarantee reasoning, may terminate early with a smaller assumption than the weakest

example





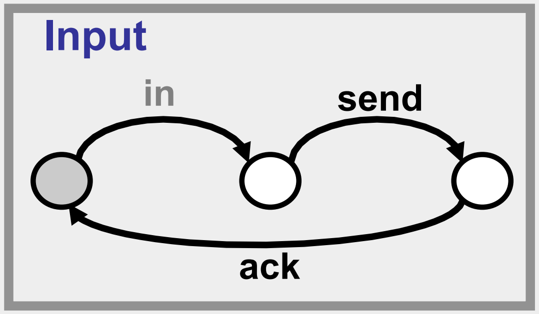


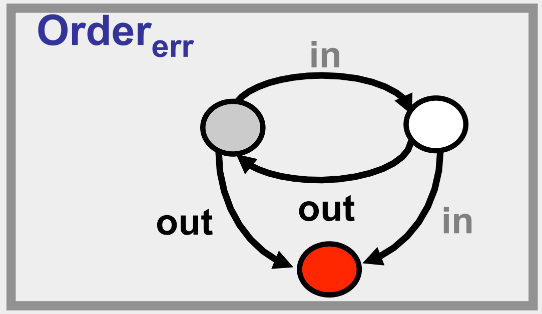
we check: ⟨true⟩ Input || Output ⟨Order⟩

M_1 = Input, M_2 = Output, P = Order

assumption alphabet: {send, out, ack}

queries

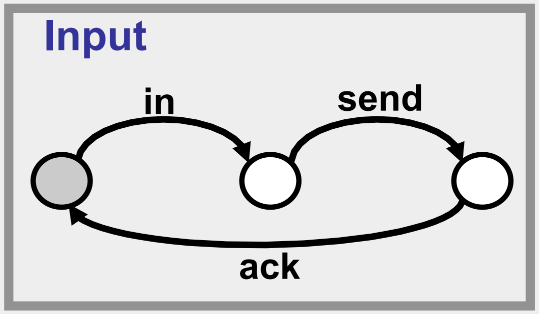


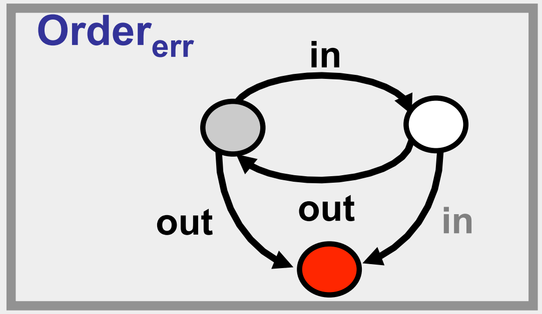


		E
	Table T	λ
S	λ	true
	out	false
S · Σ	ack	true
	out	false
	send	true
	out, ack	false
	out, out	false
	out, send	false

S = set of prefixes
E = set of suffixes

candidate construction



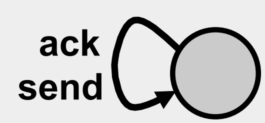


		E
	Table T	λ
S	λ	true
	out	false
$S \cdot \Sigma$	ack	true
	out	false
	send	true
	out, ack	false
	out, out	false
	out, send	false

2 states – error state omitted

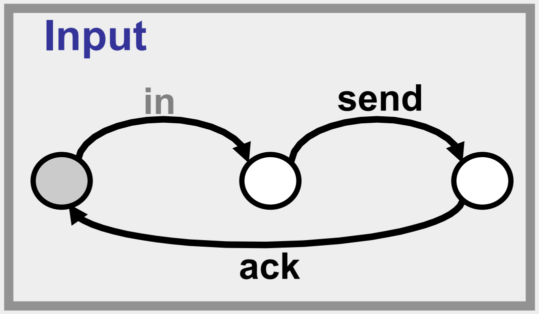
S = set of prefixes
E = set of suffixes

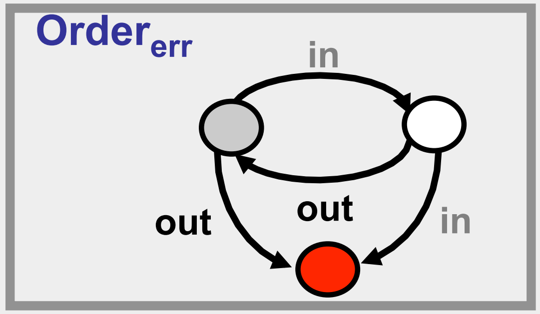
Assumption A_1

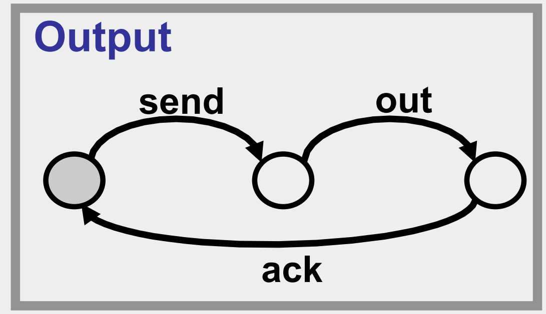


counterexamples add to *S*

conjectures







A_1:

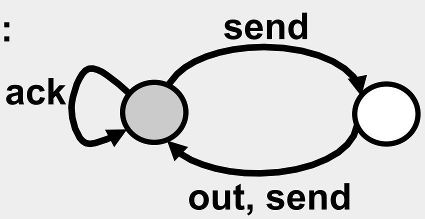
Oracle 1:
$\langle A_1 \rangle$ Input $\langle$Order$\rangle$

Counterexample:
c = $\langle$in,send,ack,in$\rangle$

Return to L*:
$c\!\uparrow\!\Sigma$ = $\langle$send,ack$\rangle$

Queries

A_2:



Oracle 1:
$\langle A_2 \rangle$ Input $\langle$Order$\rangle$
True

Oracle 2:
$\langle$true$\rangle$ Output $\langle A_2 \rangle$
True

property Order holds
on Input || Output

end of part 1

please ask LOTS of questions!

Automated Component-Based Verification

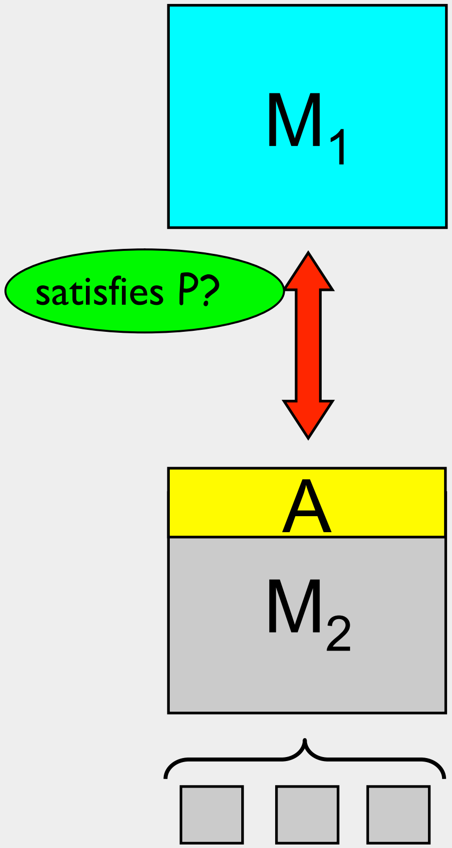
part II

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center

recap from part I

- Compositional Verification
- Assume-guarantee reasoning
- Weakest assumption
- Learning framework for reasoning about 2 components

compositional verification



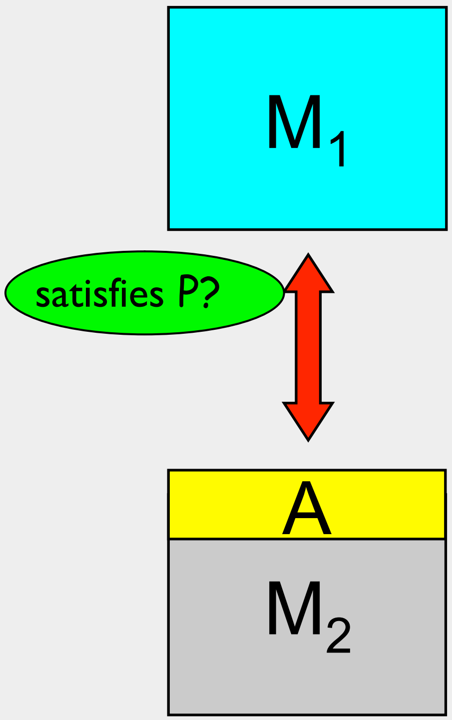
Does system made up of M_1 and M_2 satisfy property P?

- Check P on entire system: too many states!
- Use the natural decomposition of the system into its components to break-up the verification task
- Check components in isolation:

  Does M_1 satisfy P?

  – Typically a component is designed to satisfy its requirements in specific contexts / environments
- Assume-guarantee reasoning:
  – Introduces assumption A representing M_1's "context"

assume-guarantee reasoning



- Reason about triples:

  $\langle A \rangle\ M\ \langle P \rangle$

  The formula is *true* if whenever M is part of a system that satisfies A, then the system must also guarantee P

- Simplest assume-guarantee rule – ASYM

$$\frac{\begin{array}{llll} 1. & \langle A \rangle & M_1 & \langle P \rangle \\ 2. & \langle \mathit{true} \rangle & M_2 & \langle A \rangle \end{array}}{\langle \mathit{true} \rangle\ M_1 \,||\, M_2\ \langle P \rangle}$$

"discharge" the assumption

How do we come up with the assumption A?
(usually a difficult manual process)

Solution: synthesize A automatically

the weakest assumption

- Given component M, property P, and the interface of M with its environment, generate the weakest environment assumption WA such that: $\langle WA \rangle$ M $\langle P \rangle$ holds

- Weakest means that for all environments E:

$$\langle \mathit{true} \rangle \; M \parallel E \; \langle P \rangle \text{ IFF } \langle \mathit{true} \rangle \; E \; \langle WA \rangle$$

assumption generation [ASE'02]

STEP 1: composition, hiding, minimization → property true! (all environments)

STEP 2: backward propagation of error along τ transitions → property false! (all environments)

STEP 3: property extraction (subset construction & completion) → assumption

learning for assume-guarantee reasoning

- Use an off-the-shelf learning algorithm to build appropriate assumption for rule ASYM

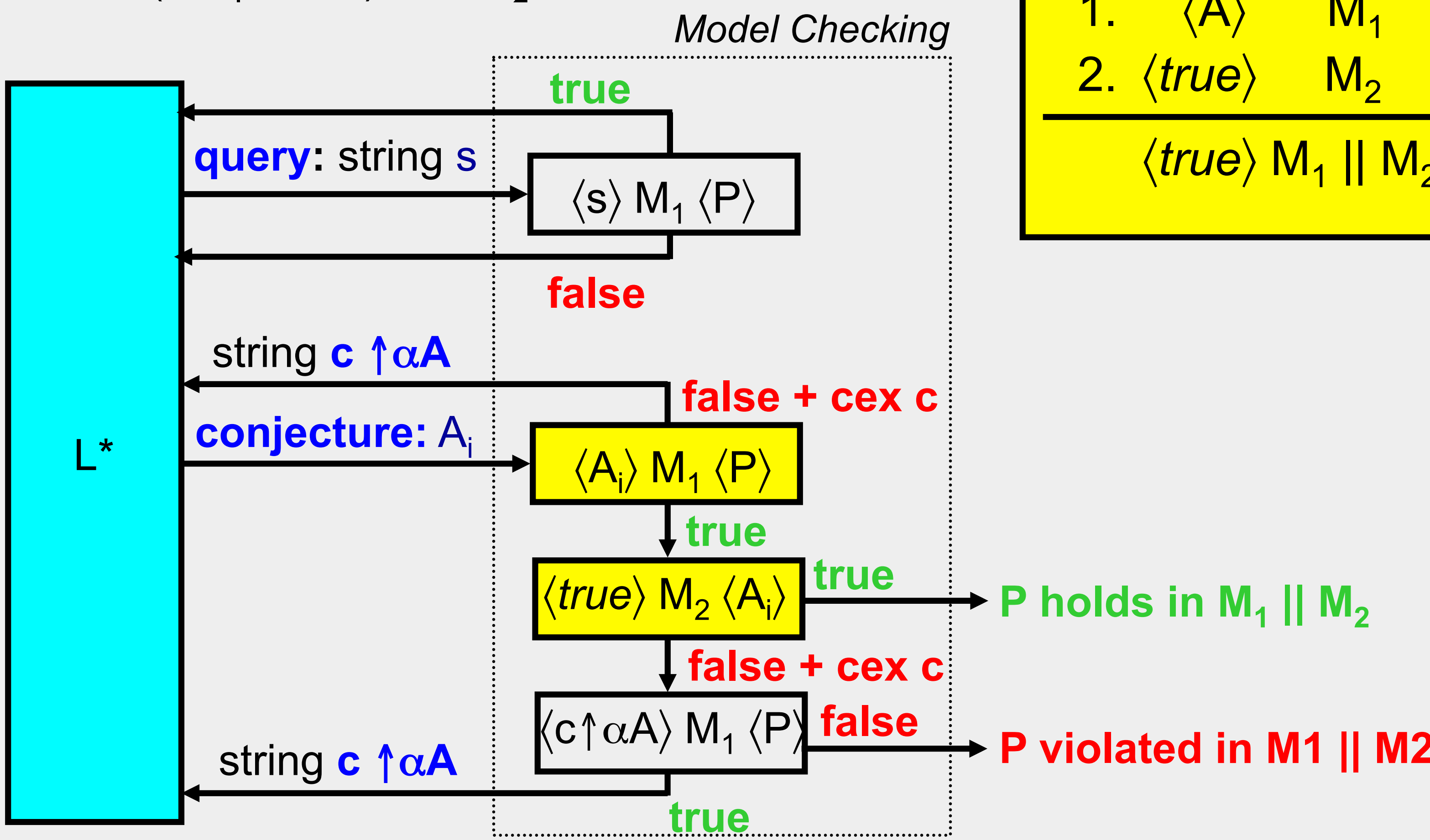
$$\frac{\begin{array}{llll} 1. & \langle A \rangle & M_1 & \langle P \rangle \\ 2. & \langle \mathit{true} \rangle & M_2 & \langle A \rangle \end{array}}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

- Process is *iterative*
- Assumptions are generated by querying the system, and are gradually refined
- Queries are answered by model checking
- Refinement is based on counterexamples obtained by model checking
- Termination is guaranteed

learning assumptions

- Use L* to generate candidate assumptions
- $\alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$

$$\begin{array}{l} 1.\ \ \langle A\rangle\ \ M_1\ \ \langle P\rangle \\ 2.\ \ \langle \mathit{true}\rangle\ \ M_2\ \ \langle A\rangle \\ \hline \langle \mathit{true}\rangle\ M_1 \,\|\, M_2\ \langle P\rangle \end{array}$$



- Guaranteed to terminate
- Reaches weakest assumption or terminates earlier

part II

- compositional verification
- assume-guarantee reasoning
- weakest assumption
- learning framework for reasoning about 2 components

extensions:

- reasoning about n > 2 components
- symmetric and circular assume-guarantee rules
- alphabet refinement

extension to *n* components

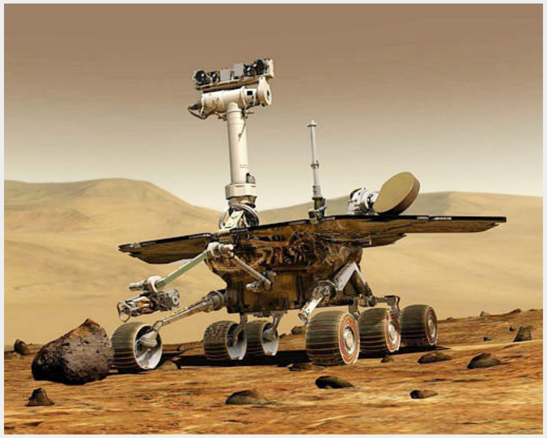
- To check if $M_1 \parallel M_2 \parallel \ldots \parallel M_n$ satisfies P
  - decompose it into M_1 and $M'_2 = M_2 \parallel \ldots \parallel M_n$
  - apply learning framework recursively for 2nd premise of rule
  - A plays the role of the property

$$\frac{\begin{array}{ll} 1. & \langle A \rangle\ M_1\ \langle P \rangle \\ 2. & \langle \mathit{true} \rangle\ M_2 \parallel \ldots \parallel M_n\ \langle A \rangle \end{array}}{\langle \mathit{true} \rangle\ M_1 \parallel M_2 \ldots \parallel M_n\ \langle P \rangle}$$

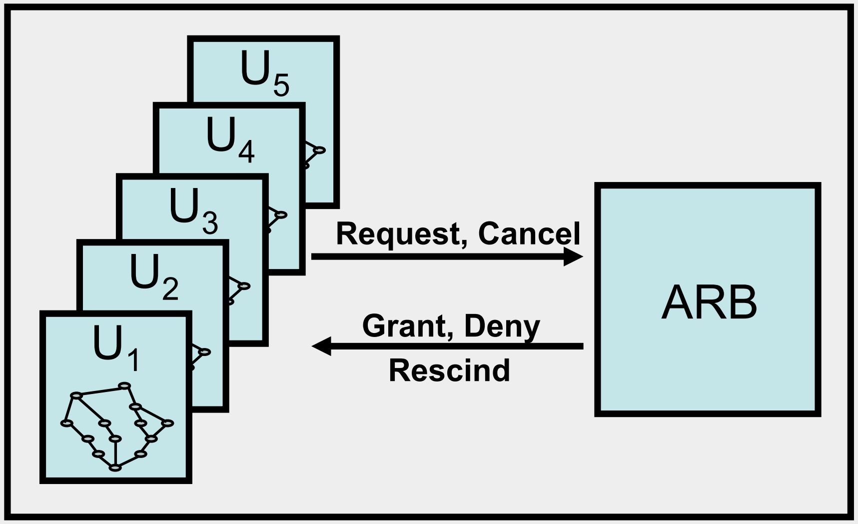
- At each recursive invocation for M_j and $M'_j = M_{j+1} \parallel \ldots \parallel M_n$
  - use learning to compute A_j such that
    - $\langle A_i \rangle\ M_j\ \langle A_{j-1} \rangle$ is true
    - $\langle \mathit{true} \rangle\ M_{j+1} \parallel \ldots \parallel M_n \langle A_j \rangle$ is true

example

- Model derived from Mars Exploration Rover (MER) Resource Arbiter
  - Local management of resource contention between resource consumers (e.g. science instruments, communication systems)
  - Consists of k user threads and one server thread (arbiter)
- Checked mutual exclusion between resources
  - E.g. driving while capturing a camera image are mutually incompatible
- Compositional verification scaled to >5 users vs. monolithic verification ran out of memory [SPIN'06]



Resource Arbiter



recursive invocation

- Compute $A_1 \ldots A_5$ s.t.

$\langle A_1 \rangle\ U_1\ \langle P \rangle$

$\langle true \rangle\ U_2 \parallel U_3 \parallel U_4 \parallel U_5 \parallel ARB\ \langle A_1 \rangle$

$\langle A_2 \rangle\ U_2\ \langle A_1 \rangle$

$\langle true \rangle\ U_3 \parallel U_4 \parallel U_5 \parallel ARB\ \langle A_2 \rangle$

$\langle A_3 \rangle\ U_3\ \langle A_2 \rangle$

$\langle true \rangle\ U_4 \parallel U_5 \parallel ARB\ \langle A_2 \rangle$

$\langle A_4 \rangle\ U_4\ \langle A_3 \rangle$

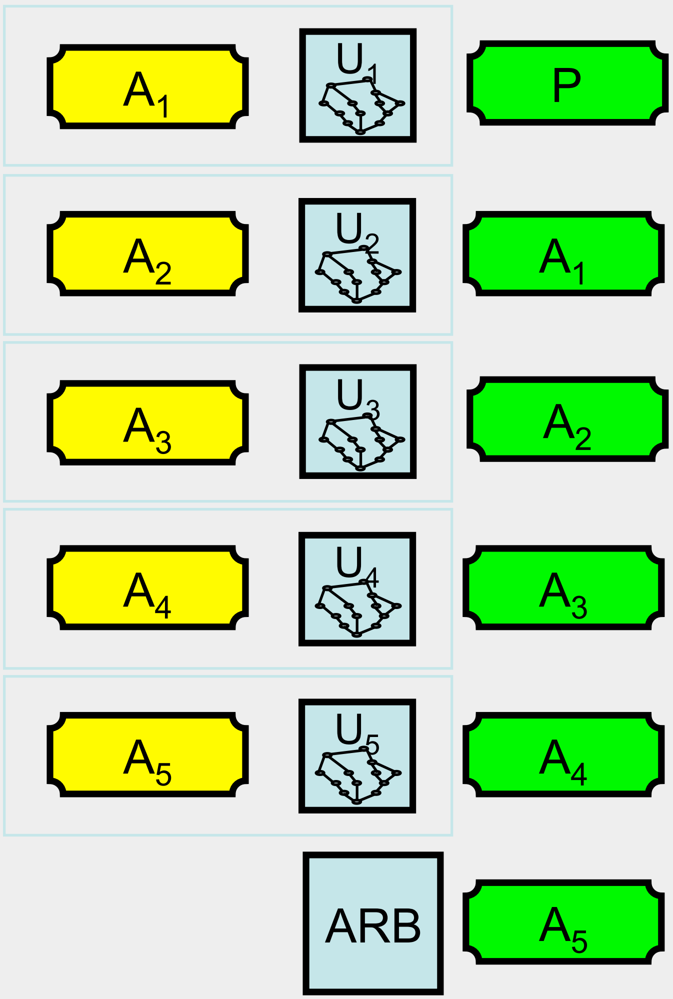
$\langle true \rangle\ U_5 \parallel ARB\ \langle A_4 \rangle$

$\langle A_5 \rangle\ U_5\ \langle A_4 \rangle$

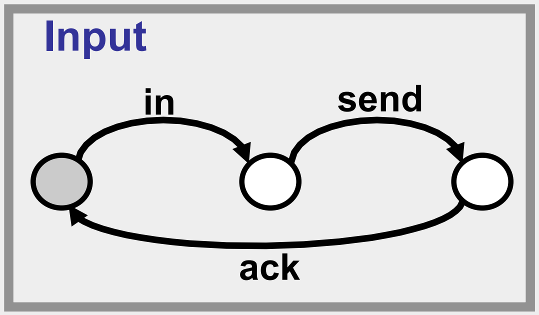
$\langle true \rangle\ ARB\ \langle A_5 \rangle$

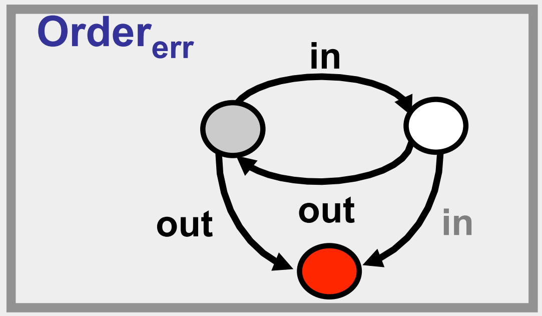
- Result:

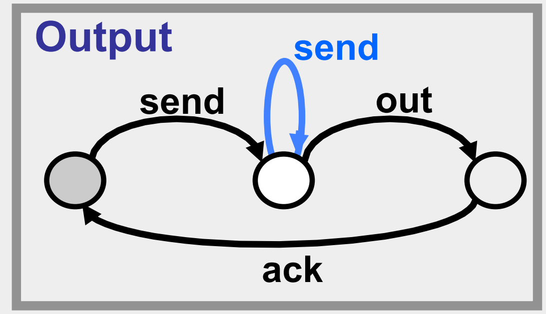
$\langle true \rangle\ U_1 \parallel .. \parallel U_5 \parallel ARB\ \langle P \rangle$



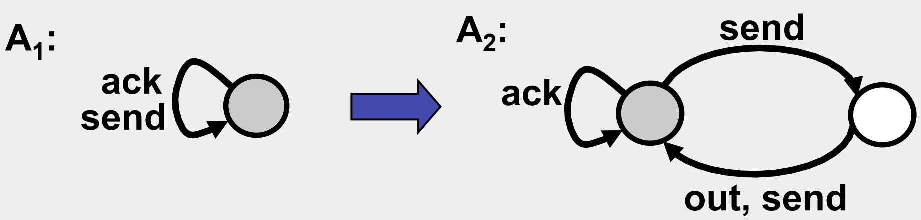
symmetric rules: motivation

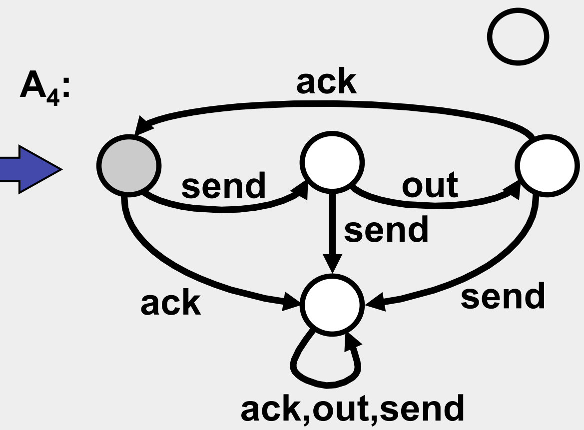




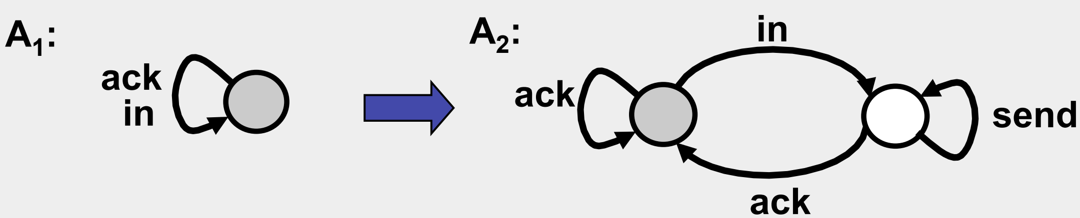


M_1 = Input, M_2 = Output, P = Order





M_1 = Output, M_2 = Input, P = Order



symmetric rules

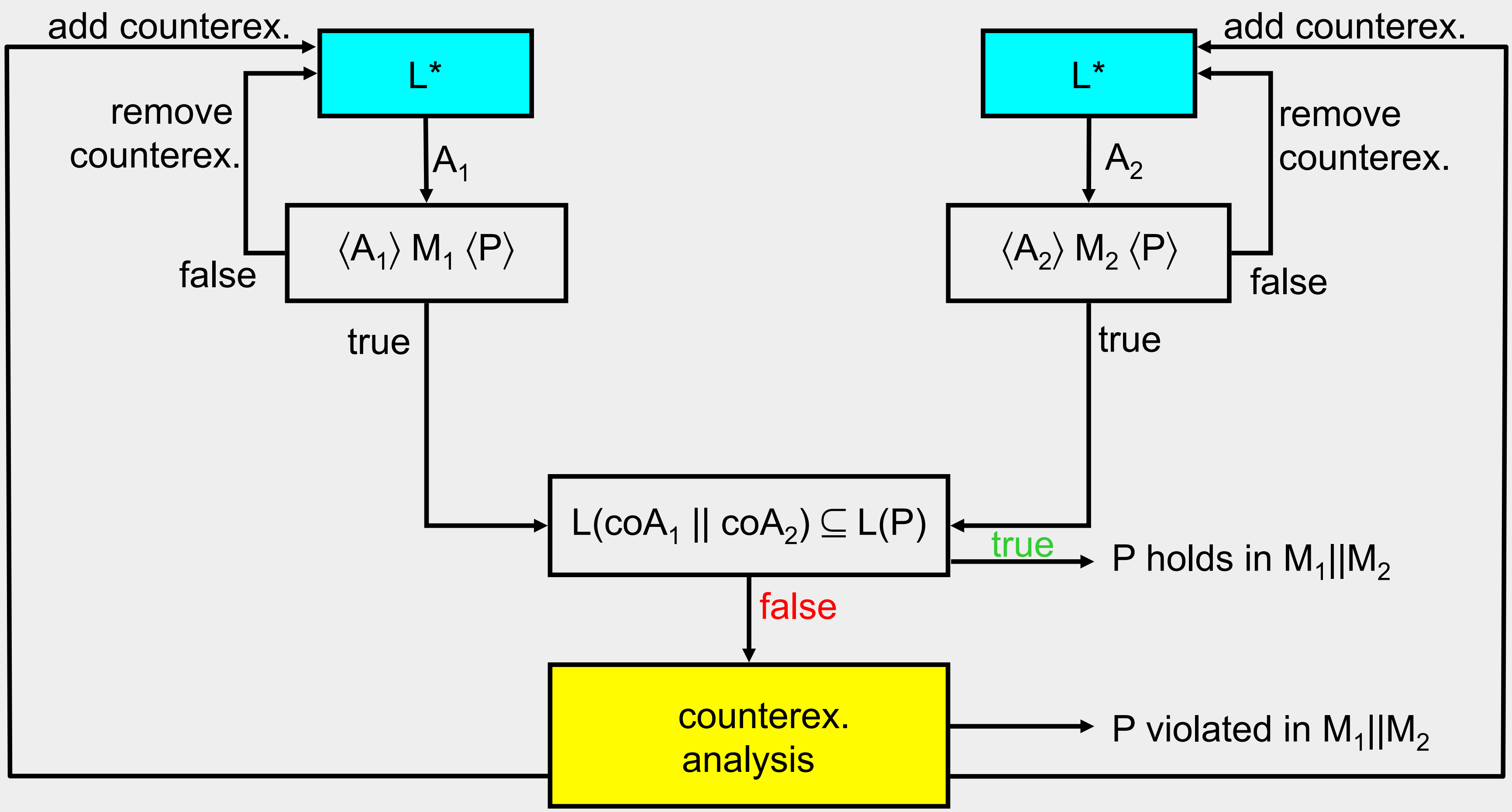
- Assumptions for both components at the same time
  - Early termination; smaller assumptions
- Example symmetric rule – SYM

$$\begin{array}{ll} 1. & \langle A_1\rangle\ M_1\ \langle P\rangle \\ 2. & \langle A_2\rangle\ M_2\ \langle P\rangle \\ 3. & L(coA_1 \,||\, coA_2) \subseteq L(P) \\ \hline & \langle \mathit{true}\rangle\ M_1 \,||\, M_2\ \langle P\rangle \end{array}$$

Ensure that any common trace ruled out by both assumptions satisfies P.

- coA_i = complement of A_i, for i=1,2
- Requirements for alphabets:
  - $\alpha P \subseteq \alpha M_1 \cup \alpha M_2$; $\alpha A_i \subseteq (\alpha M_1 \cap \alpha M_2) \cup \alpha P$, for i =1,2
- The rule is sound and complete
- Completeness needed to guarantee termination
- Straightforward extension to *n* components

learning framework for rule SYM



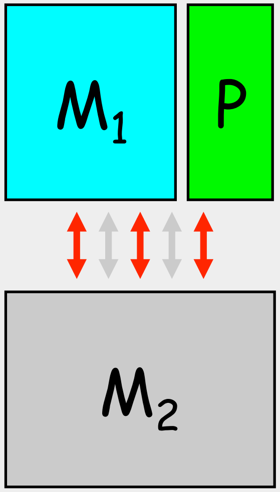
circular rule

- Rule CIRC – from [Grumberg&Long – Concur'91]

$$\begin{array}{ll} 1. & \langle A_1\rangle\ M_1\ \langle P\rangle \\ 2. & \langle A_2\rangle\ M_2\ \langle A_1\rangle \\ 3. & \langle \mathit{true}\rangle\ M_1\ \langle A_2\rangle \\ \hline & \langle \mathit{true}\rangle\ M_1 \parallel M_2\ \langle P\rangle \end{array}$$

- Similar to rule ASYM applied recursively to 3 components
  - First and last component coincide
  - Hence learning framework similar
- Straightforward extension to *n* components

assumption alphabet refinement



- Rule ASYM
  - Assumption alphabet was fixed during learning
  - $\alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$
- [SPIN'06]: A subset alphabet
  - May be sufficient to prove the desired property
  - May lead to smaller assumption
- How do we compute a good subset of the assumption alphabet?
- Solution – iterative alphabet refinement
  - Start with small alphabet
  - Add actions as necessary
  - Discovered by analysis of counterexamples obtained from model checking

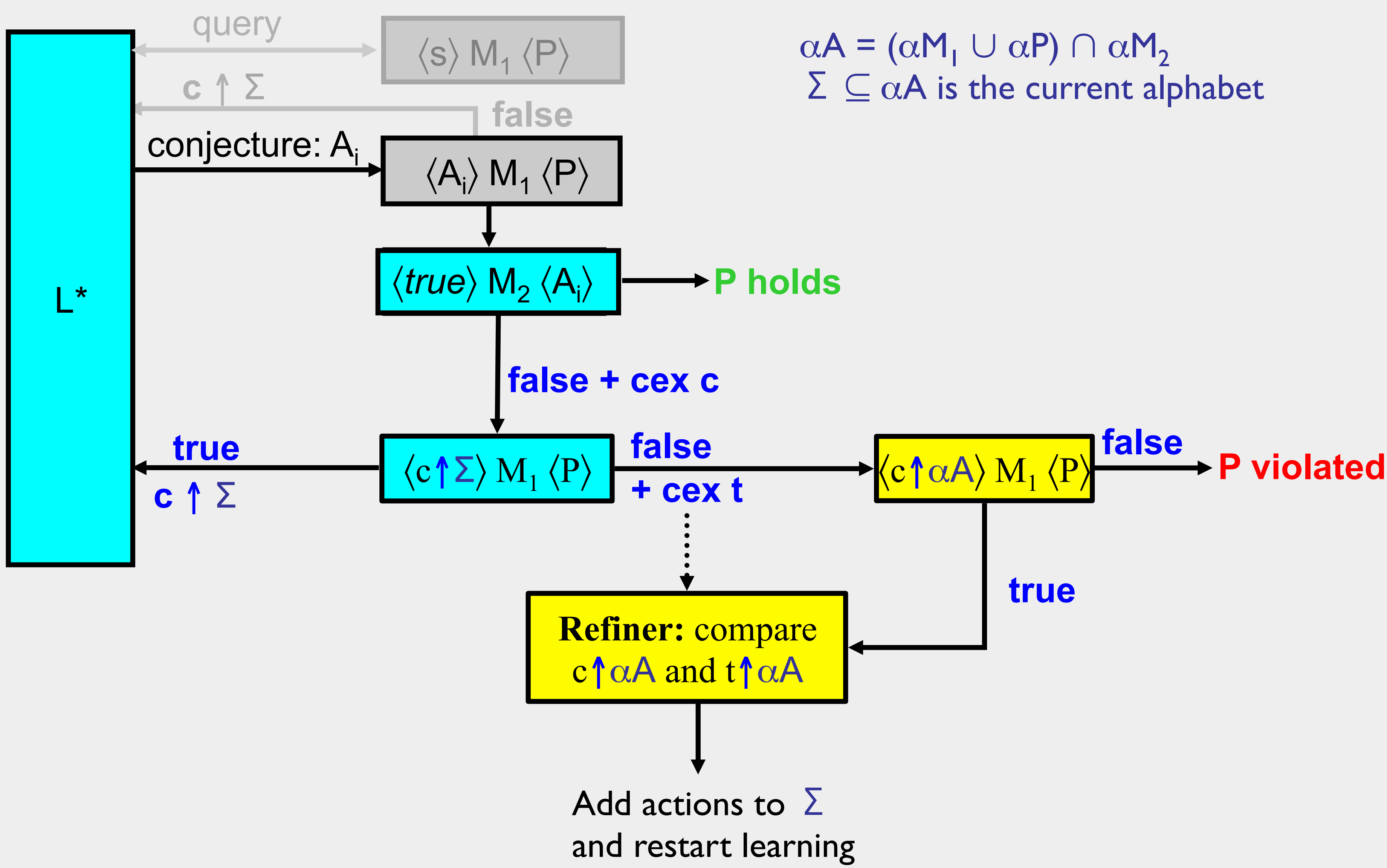
learning with alphabet refinement

1. Initialize Σ to subset of alphabet $\alpha A = (\alpha M_1 \cup \alpha P) \cap \alpha M_2$
2. If learning with Σ returns true, return true and go to 4. (END)
3. If learning returns false (with counterexample c), perform extended counterexample analysis on c.

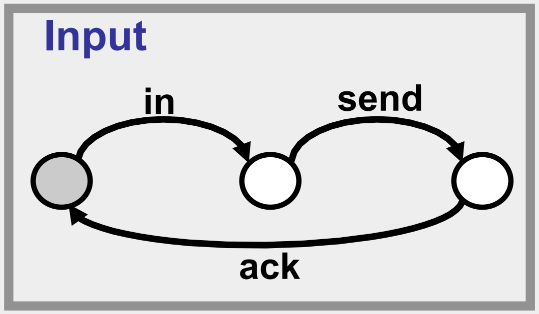
   If c is real, return false and go to 4. (END)

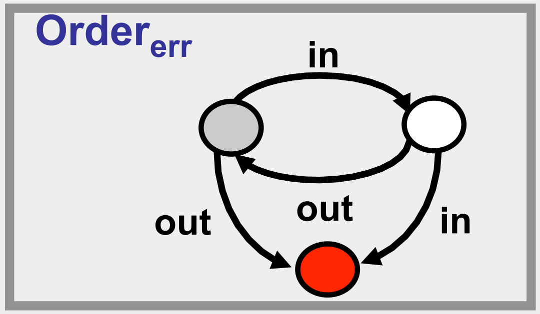
   If c is spurious, add more actions from αA to Σ and go to 2.
4. END

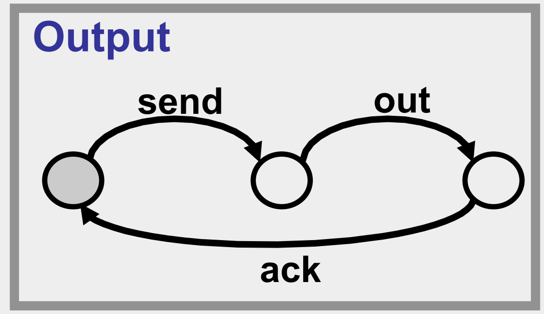
extended counterexample analysis



alphabet refinement







$\Sigma = \{ \text{out} \}$ $\alpha A = \{ \text{send, out, ack} \}$

$\langle \text{true} \rangle$ Output $\langle A_i \rangle$ ⇒ false with $c = \langle \text{send, out} \rangle$ → $c{\uparrow}\Sigma = \langle \text{out} \rangle$; → $c{\uparrow}\alpha A = \langle \text{send, out} \rangle$

$\langle c{\uparrow}\Sigma \rangle$ Input $\langle P \rangle$ ⇒ false with counterex. $t = \langle \text{out} \rangle$

$\langle c{\uparrow}\alpha A \rangle$ Input $\langle P \rangle$ ⇒ true

compare $\langle \text{out} \rangle$ with $\langle \text{send, out} \rangle$ ⇒ add "send" to Σ

characteristics

- Initialization of Σ
  - Empty set or property alphabet $\alpha P \cap \alpha A$
- Refiner
  - Compares $t{\uparrow}\alpha A$ and $c{\uparrow}\alpha A$
  - Heuristics:

  AllDiff adds all actions in the symmetric difference of the trace alphabets

  Forward scans traces in parallel forward adding first action that differs

  Backward symmetric to previous
- Termination
  - Refinement produces at least one new action and the interface is finite
- Generalization to *n* components
  - Through recursive invocation

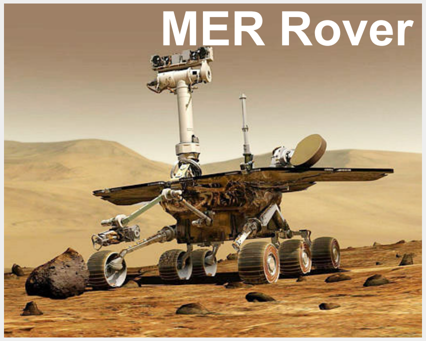
- See also learning with optimal alphabet refinement
  - Developed independently by Chaki & Strichman 07

implementation & experiments

- Implementation in the LTSA tool
  - Learning using rules ASYM, SYM and CIRC
  - Supports reasoning about two and *n* components
  - Alphabet refinement for all the rules
- Experiments
  - Compare effectiveness of different rules
  - Measure effect of alphabet refinement
  - Measure scalability as compared to non-compositional verification
- Extensions for
  - SPIN
  - JavaPathFinder

case studies

- Model of Ames K9 Rover Executive
  - Executes flexible plans for autonomy
  - Consists of main Executive thread and ExecCondChecker thread for monitoring state conditions
  - Checked for specific shared variable: if the Executive reads its value, the ExecCondChecker should not read it before the Executive clears it
- Model of JPL MER Resource Arbiter
  - Local management of resource contention between resource consumers (e.g. science instruments, communication systems)
  - Consists of k user threads and one server thread (arbiter)
  - Checked mutual exclusion between resources



results

- Rule ASYM more effective than rules SYM and CIRC
- Recursive version of ASYM the most effective
  - When reasoning about more than two components
- Alphabet refinement improves learning based assume guarantee verification significantly
- Backward refinement slightly better than other refinement heuristics
- Learning based assume guarantee reasoning
  - Can incur significant time penalties
  - Not always better than non-compositional (monolithic) verification
  - Sometimes, significantly better in terms of memory

analysis data

	ASYM			ASYM + refinement			Monolithic	
Case	|A|	Mem	Time	|A|	Mem	Time	Mem	Time
MER 2	40	8.65	21.90	6	1.23	1.60	1.04	0.04
MER 3	501	240.06	--	8	3.54	4.76	4.05	0.111
MER 4	273	101.59	--	10	9.61	13.68	14.29	1.46
MER 5	200	78.10	--	12	19.03	35.23	14.24	27.73
MER 6	162	84.95	--	14	47.09	91.82	--	600
K9 Rover	11	2.65	1.82	4	2.37	2.53	6.27	0.015

|A| = assumption size
Mem = memory (MB)
Time (seconds)
-- = reached time (30min) or memory limit (1GB)

end of part II

please ask LOTS of questions!

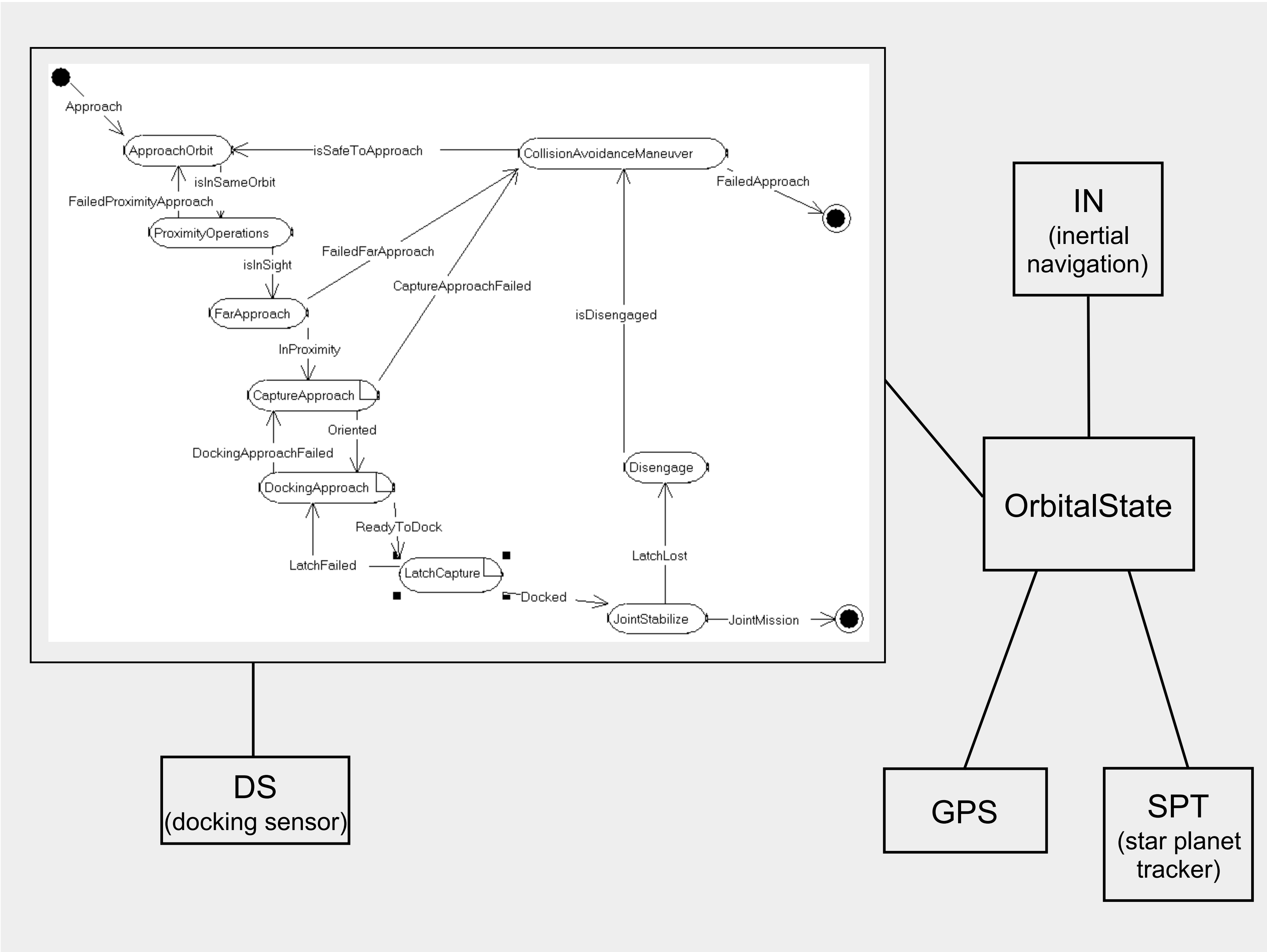
Automated Component-Based Verification

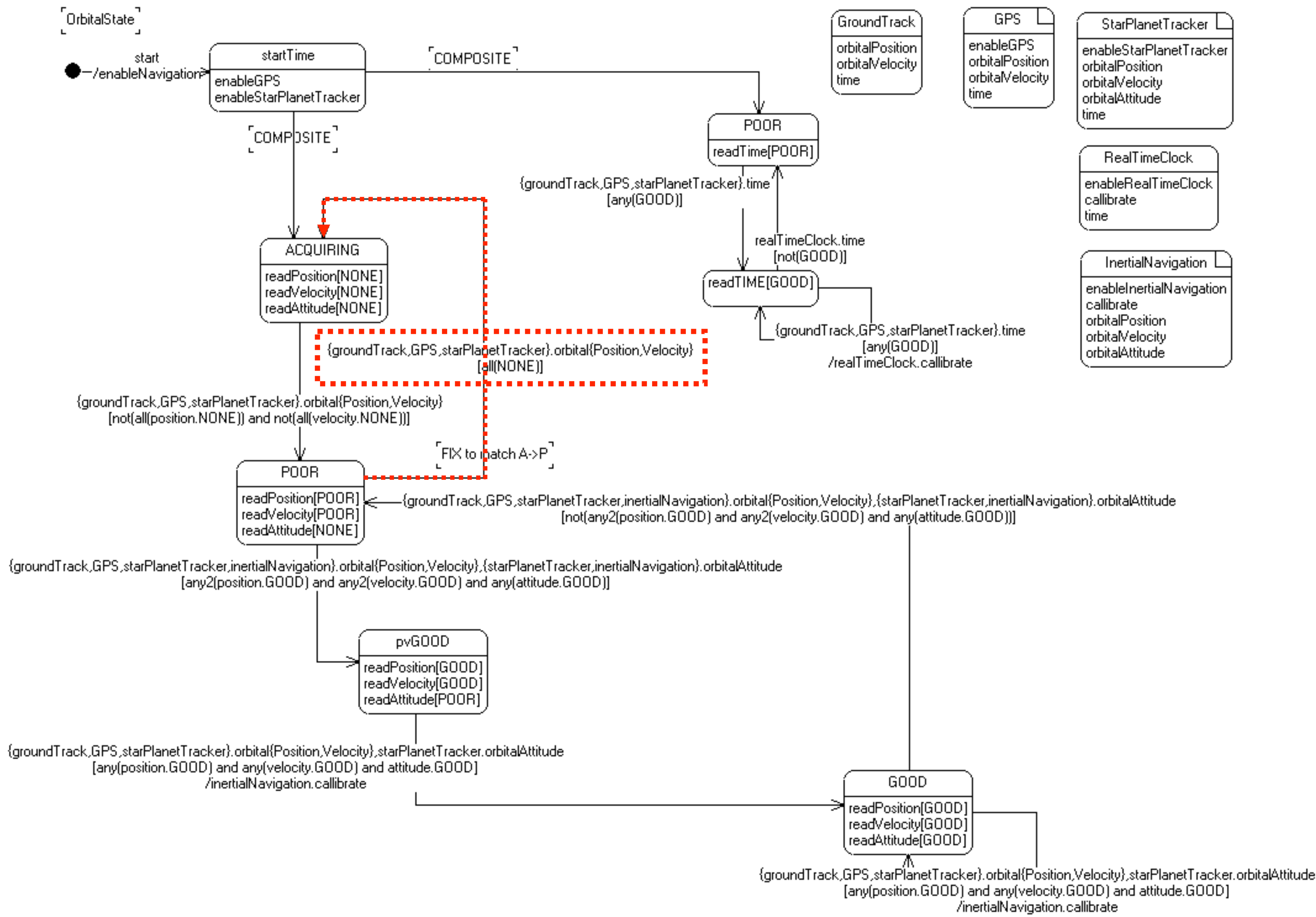
part III

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center

example: autonomous rendezvous and docking

- input provided as UML state-charts, properties of type:
  - "you need at least two operational sensors to proceed to next mode"
  - "your state estimator will only return good values if it has good readings from at least 2 sensors"
- 3 bugs detected
- scaling achieved with compositional verification:
  - non-compositional verification runs out of memory after exploring > 13M states
  - compositional verification terminates successfully in secs. Analyzes one component at a time. The largest assumption has 10 states and the largest component has 5947 states (so largest state space explored is less that 60K states, as opposed to 13M)


Approach
ApproachOrbit
isSafeToApproach
CollisionAvoidanceManeuver
isInSameOrbit
FailedProximityApproach
FailedApproach
ProximityOperations
isInSight
FailedFarApproach
CaptureApproachFailed
FarApproach
isDisengaged
InProximity
CaptureApproach
Oriented
DockingApproachFailed
DockingApproach
Disengage
ReadyToDock
LatchFailed
LatchCapture
LatchLost
Docked
JointStabilize
JointMission
IN
(inertial
navigation)
OrbitalState
DS
(docking sensor)
GPS
SPT
(star planet
tracker)


OrbitalState
start
/enableNavigation
startTime
enableGPS
enableStarPlanetTracker
COMPOSITE
COMPOSITE
POOR
readTime[POOR]
{groundTrack,GPS,starPlanetTracker}.time
[any(GOOD)]
realTimeClock.time
[not(GOOD)]
readTIME[GOOD]
{groundTrack,GPS,starPlanetTracker}.time
[any(GOOD)]
/realTimeClock.callibrate
GroundTrack
orbitalPosition
orbitalVelocity
time
GPS
enableGPS
orbitalPosition
orbitalVelocity
time
StarPlanetTracker
enableStarPlanetTracker
orbitalPosition
orbitalVelocity
orbitalAttitude
time
RealTimeClock
enableRealTimeClock
callibrate
time
InertialNavigation
enableInertialNavigation
callibrate
orbitalPosition
orbitalVelocity
orbitalAttitude
ACQUIRING
readPosition[NONE]
readVelocity[NONE]
readAttitude[NONE]
{groundTrack,GPS,starPlanetTracker}.orbital{Position,Velocity}
[all(NONE)]
{groundTrack,GPS,starPlanetTracker}.orbital{Position,Velocity}
[not(all(position.NONE)) and not(all(velocity.NONE))]
FIX to match A->P
POOR
readPosition[POOR]
readVelocity[POOR]
readAttitude[NONE]
{groundTrack,GPS,starPlanetTracker,inertialNavigation}.orbital{Position,Velocity},{starPlanetTracker,inertialNavigation}.orbitalAttitude
[not(any2(position.GOOD) and any2(velocity.GOOD) and any(attitude.GOOD))]
{groundTrack,GPS,starPlanetTracker,inertialNavigation}.orbital{Position,Velocity},{starPlanetTracker,inertialNavigation}.orbitalAttitude
[any2(position.GOOD) and any2(velocity.GOOD) and any(attitude.GOOD)]
pvGOOD
readPosition[GOOD]
readVelocity[GOOD]
readAttitude[POOR]
{groundTrack,GPS,starPlanetTracker}.orbital{Position,Velocity},starPlanetTracker.orbitalAttitude
[any(position.GOOD) and any(velocity.GOOD) and attitude.GOOD]
/inertialNavigation.callibrate
GOOD
readPosition[GOOD]
readVelocity[GOOD]
readAttitude[GOOD]
{groundTrack,GPS,starPlanetTracker}.orbital{Position,Velocity},starPlanetTracker.orbitalAttitude
[any(position.GOOD) and any(velocity.GOOD) and attitude.GOOD]
/inertialNavigation.callibrate

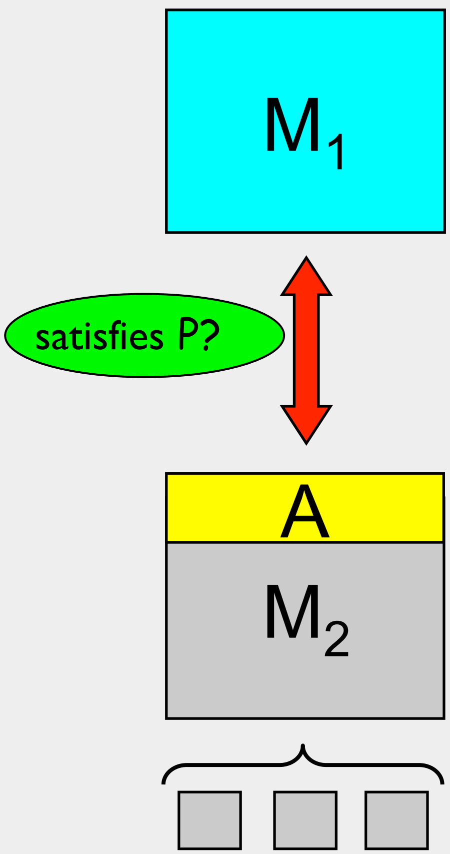
structure

part 1 (Dimitra) assume-guarantee reasoning computing assumptions learning assumptions discussion	part 2 (Corina) multiple components alphabet refinement case studies discussion
lunch	
part 3 (Dimitra) component interfaces compositional JavaPathfinfer examples discussion	part 4 (Corina) reasoning about code abstraction related work conclusion

recap in reverse order…

- assume-guarantee reasoning
- learning framework for 2 components
- weakest assumption

assume-guarantee reasoning



reasons about triples:

$\langle A \rangle\ M\ \langle P \rangle$

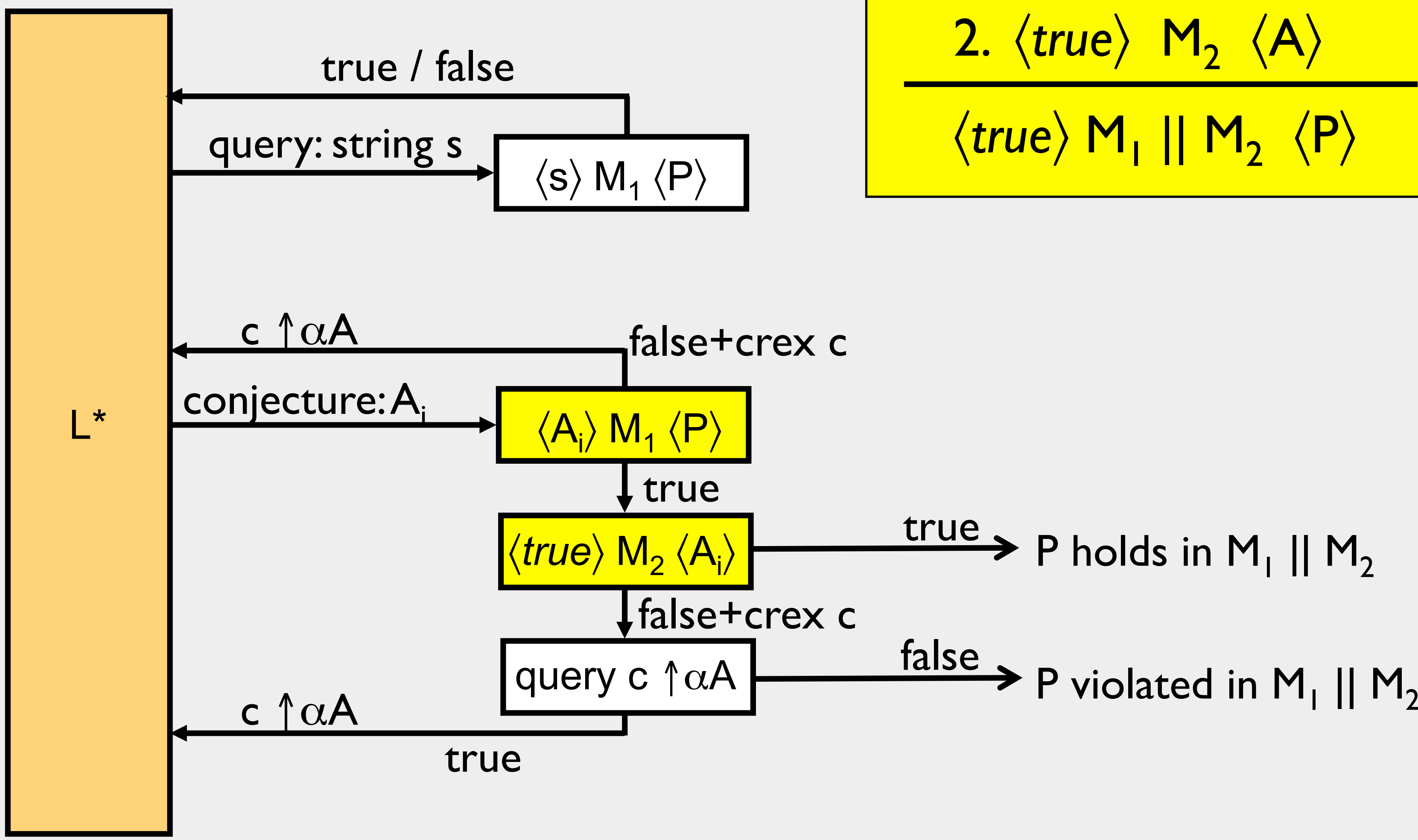
is *true* if whenever M is part of a system that satisfies A, then the system must also guarantee P

simplest assume-guarantee rule (ASYM):

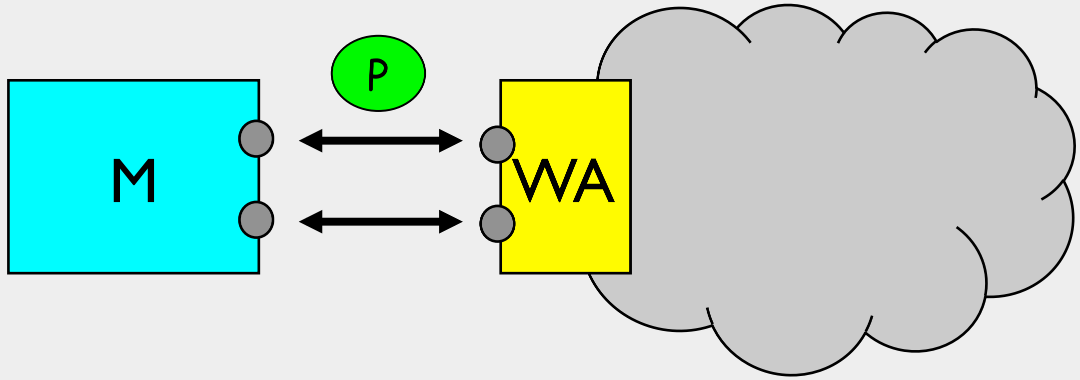
$$\frac{\begin{array}{ll} 1. & \langle A \rangle\ M_1\ \langle P \rangle \\ 2. & \langle \mathit{true} \rangle\ M_2\ \langle A \rangle \end{array}}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

learning assumptions for AG reasoning

$$\frac{\begin{array}{l}1.\quad \langle A\rangle\ M_1\ \langle P\rangle \\ 2.\ \langle \mathit{true}\rangle\ M_2\ \langle A\rangle\end{array}}{\langle \mathit{true}\rangle\ M_1 \parallel M_2\ \langle P\rangle}$$



the weakest assumption



- given component M, property P, and the interface ∑ of M with its environment, generate the weakest environment assumption WA such that: ⟨WA⟩ M ⟨P⟩ holds

- weakest means that for all environments E:

⟨*true*⟩ M || E ⟨P⟩ **IFF** ⟨*true*⟩ E ⟨WA⟩

assumption generation [ASE'02]

STEP 1: composition, hiding, minimization → property true! (all environments)

STEP 2: backward propagation of error along τ transitions → property false! (all environments)

STEP 3: property extraction (subset construction & completion) → assumption

part III

- assume-guarantee reasoning
- learning framework for 2 components
- weakest assumption WA

- component interfaces / learning WA
- compositional JavaPathFinder
- examples and discussion

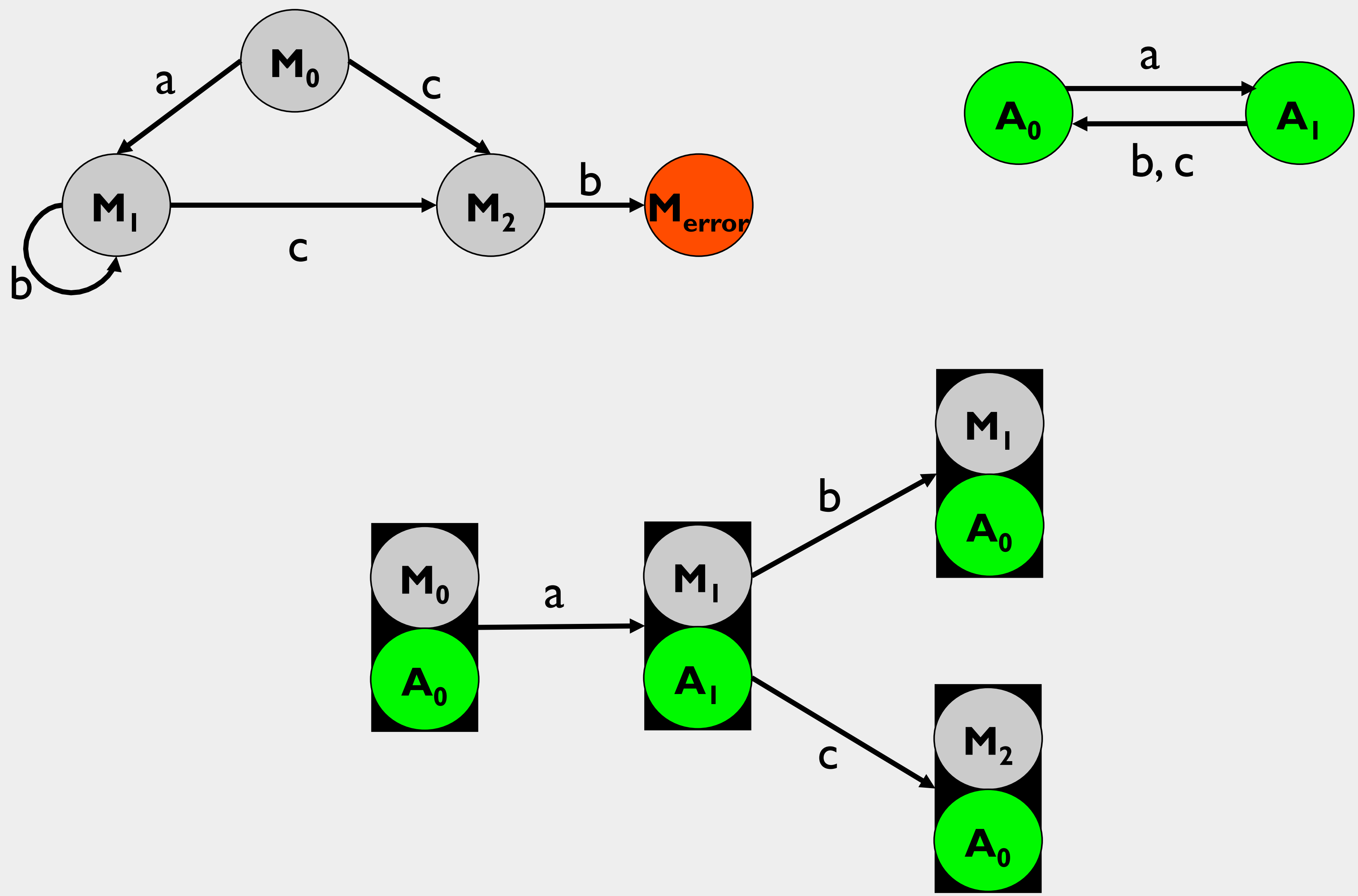
component interfaces

- beyond syntactic interfaces (“open” file before “close”)
- document implicit assumptions

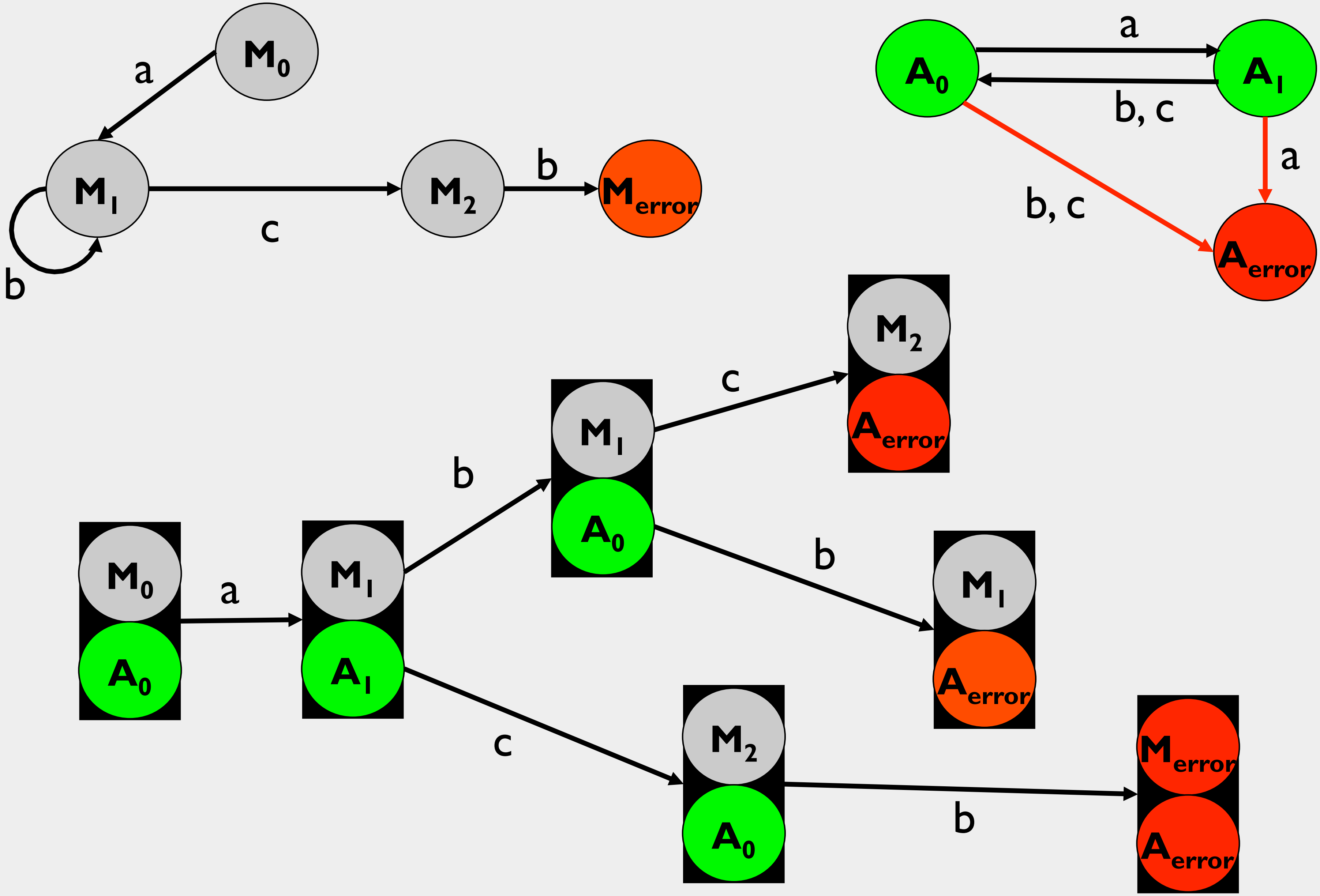
weakest assumption (WA):

- safe: accept NO illegal sequence of calls
- permissive: accept ALL legal sequences of calls

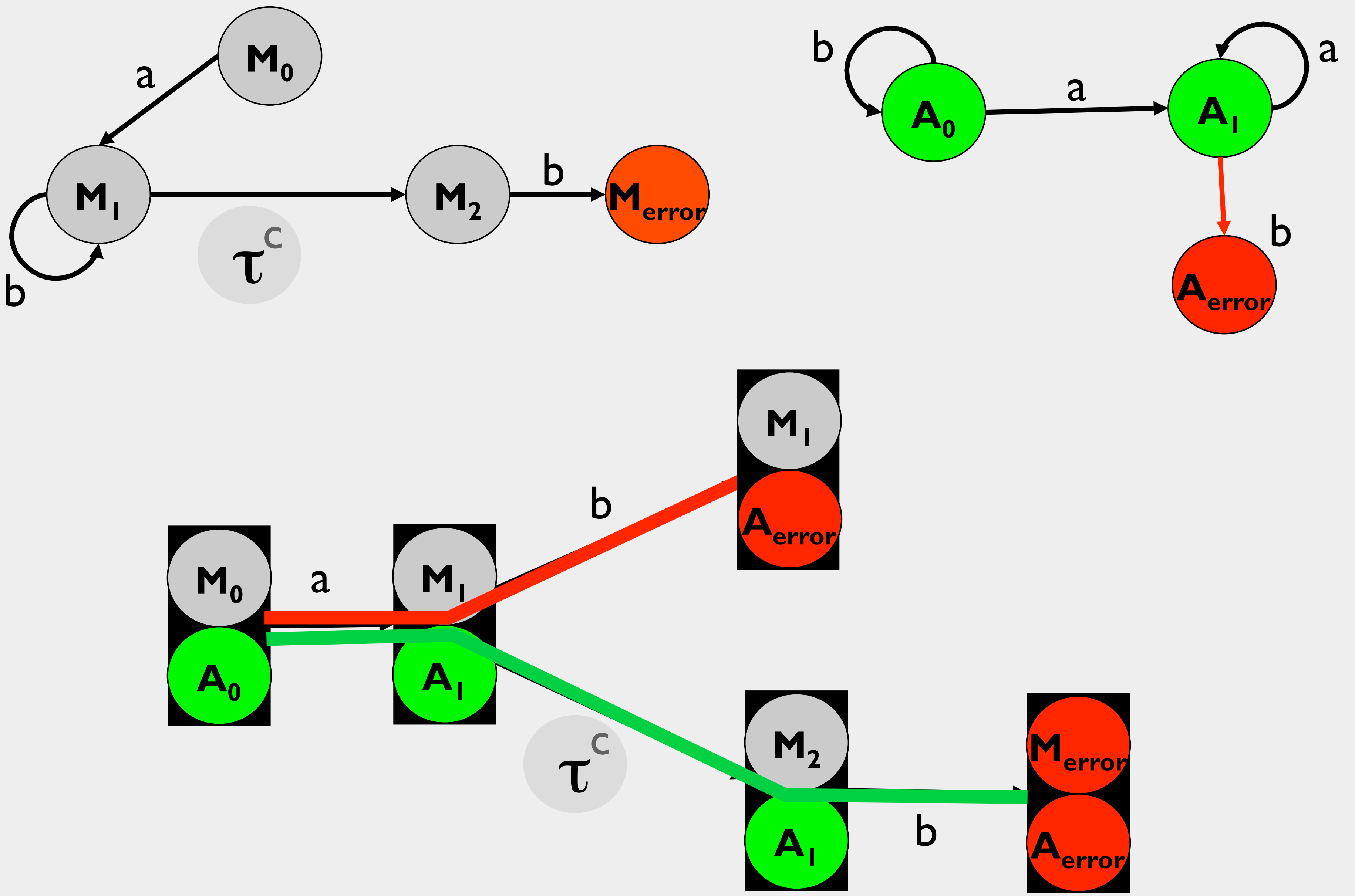
safety check

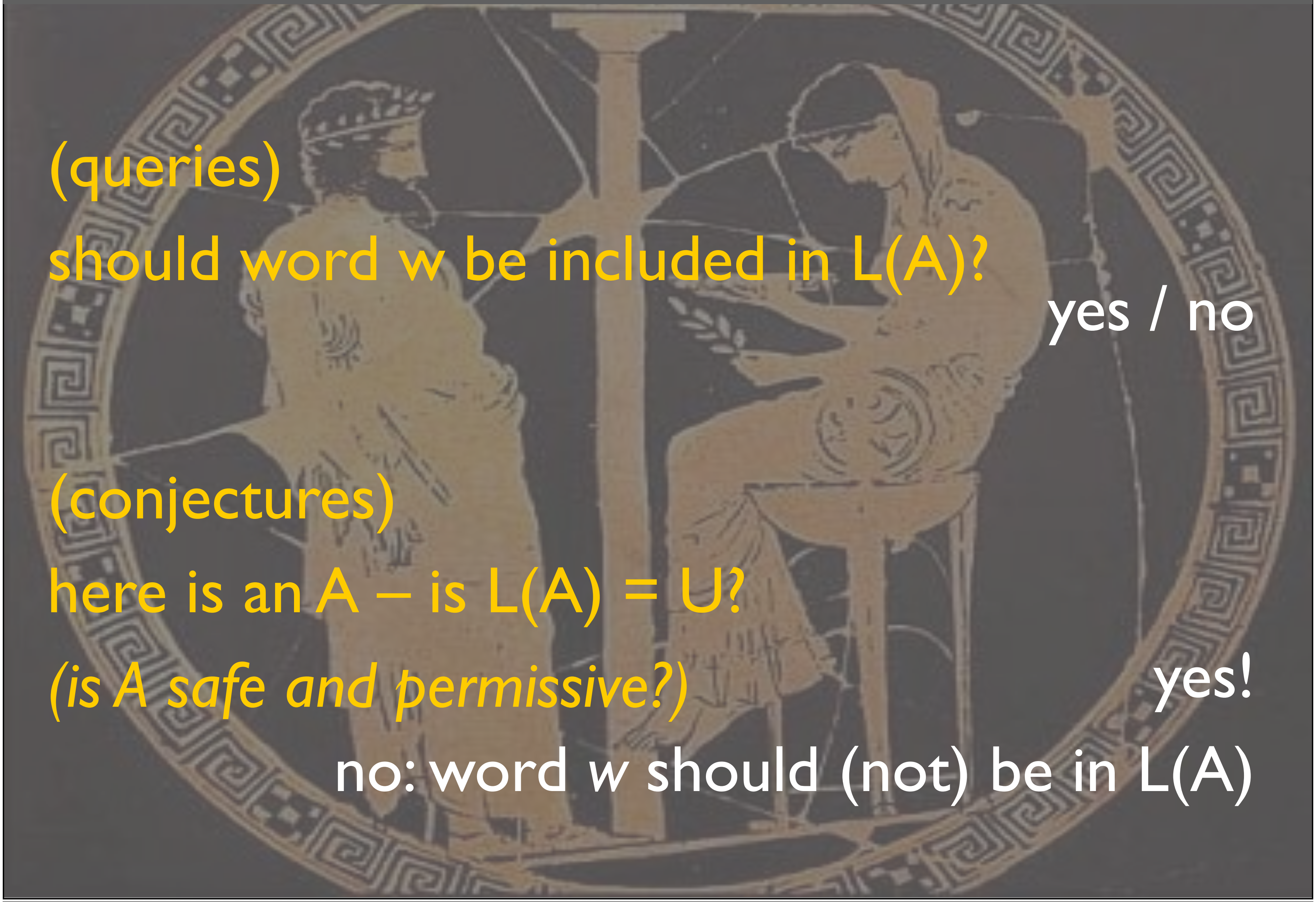


permissiveness check



permissiveness: the problem





L* learner

the oracle

(queries)

should word w be included in L(A)?

yes / no

(conjectures)

here is an A – is L(A) = U?

(is A safe and permissive?)

yes!

no: word *w* should (not) be in L(A)

learning interfaces

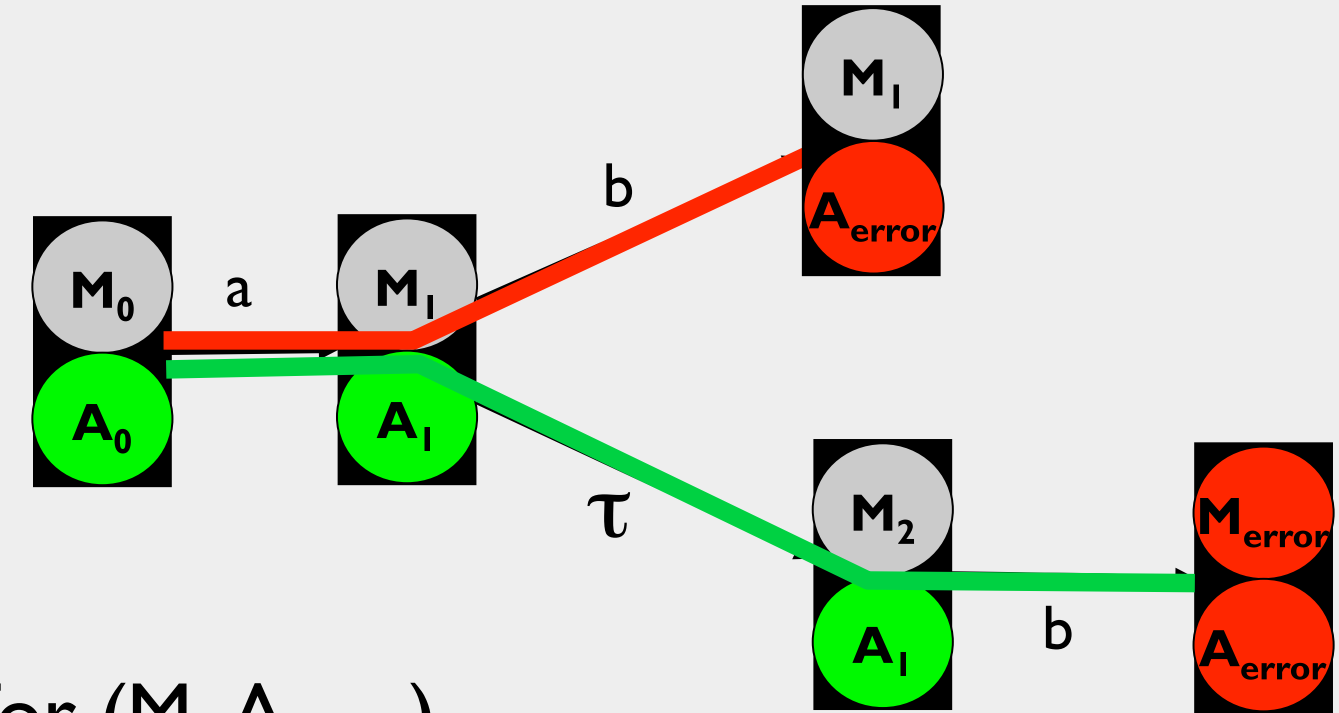
queries (simulate / model check)

conjecture – safe (model check)

conjecture – permissive

Alur et al, 2005, Henzinger et al, 2005

our approach (Giannakopoulou & Pasareanu, FASE 2009)



model check for (M_i, A_{error})

- reached (M_1, A_{error}) by "a b"
- query "a b"
- no ("a b" should not be in A)
- backtrack and continue search…

invoke a model checker

within a model checker?

permissiveness check

MC: model check for (M_i, A_{error})

reached (M_i, A_{error}) by trace t

if (memoized(t) == no) *// t is spurious*

backtrack and continue search

else *// memoized(t) == yes or t not in memoized*

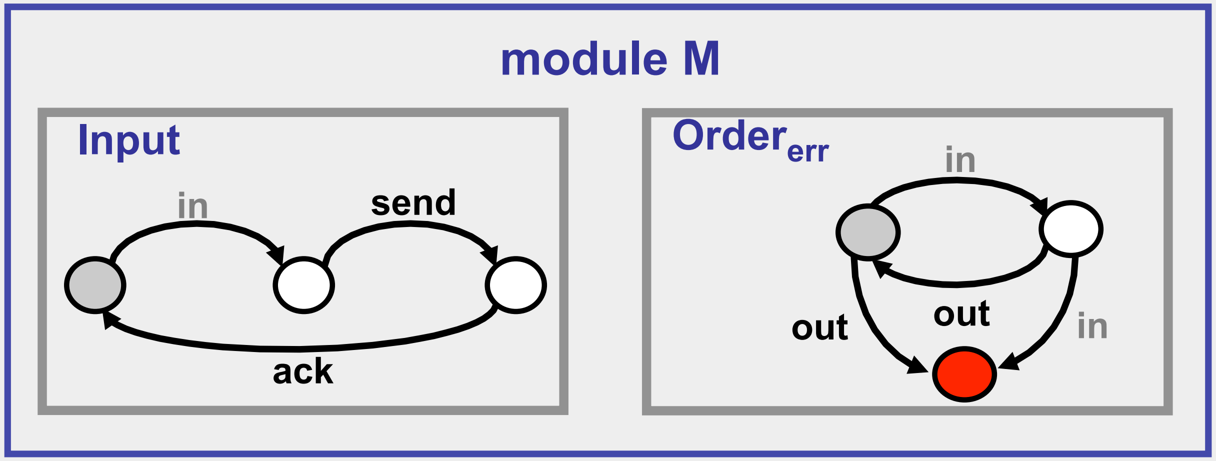
model checker produces t

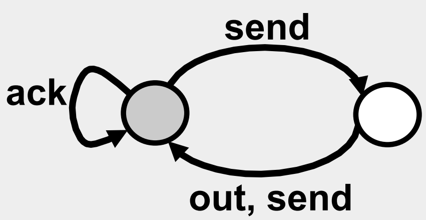
if (query(t) == yes)

return t to L* *// not permissive*

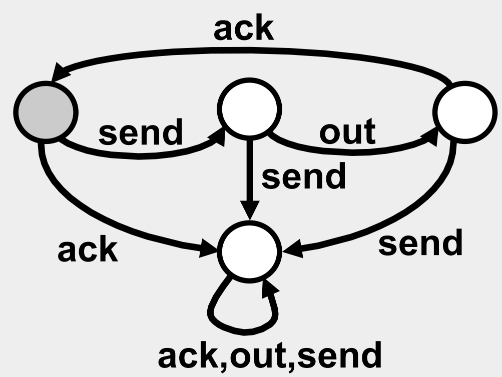
else restart at MC

example





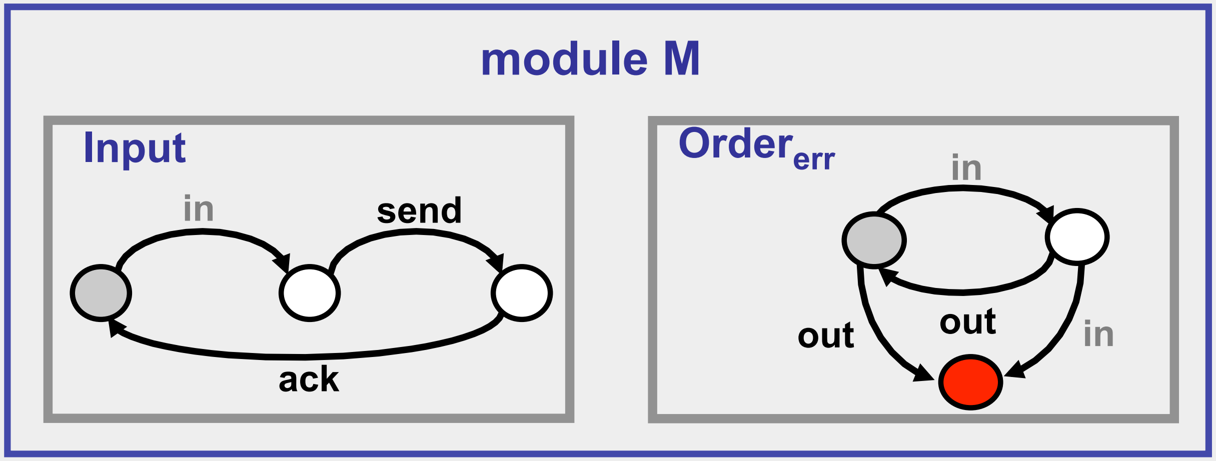
⟨ ack, out ⟩ ?

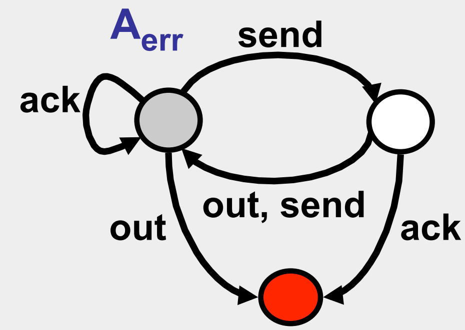


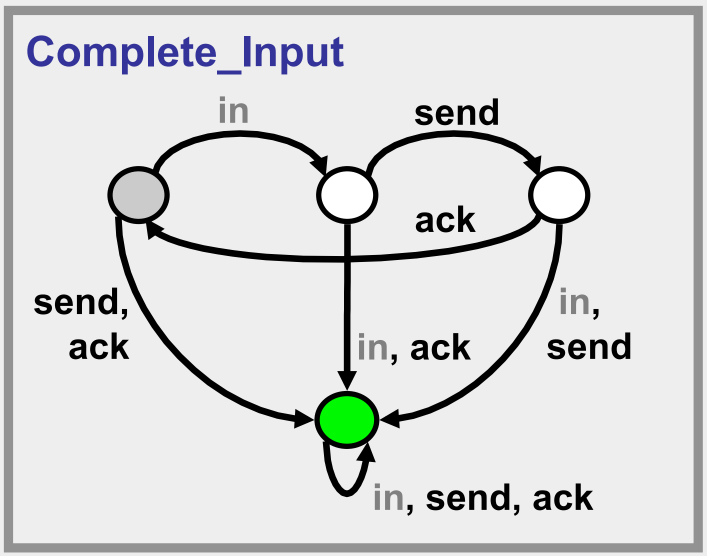
assumption learned for AG reasoning

weakest assumption

complete module for permissiveness check







- queries performed on Input || Order$_{err}$
- safety checked on Input || Order$_{err}$|| A$_{err}$
- permissiveness performed on Complete_Input || Order$_{err}$|| A$_{err}$

check reachability of states:
(sink, *, error) or (*, non error, error)

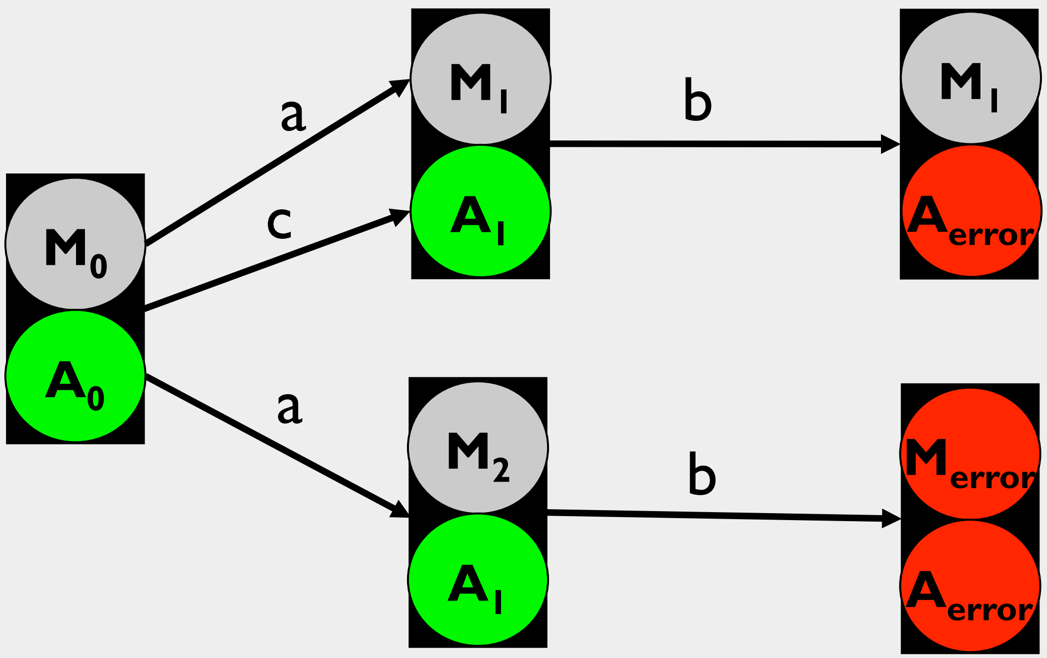
⟨ ack, out ⟩: (sink, error, error)

in summary…

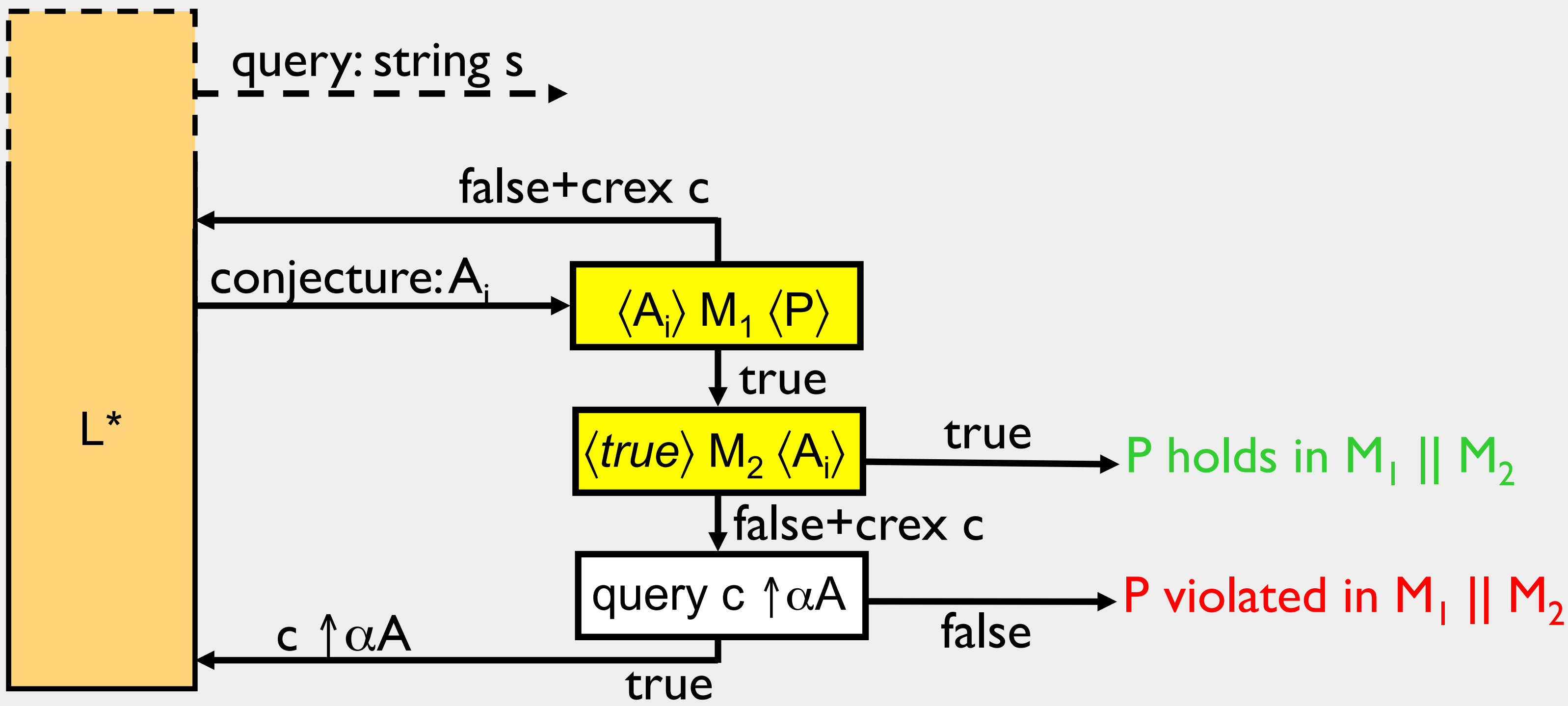
resolve non-determinism

dynamically & selectively

remember, it's a heuristic



assume-guarantee reasoning



only need permissiveness with respect to M_2 !

JavaPathfinder

UML statecharts

assume-guarantee reasoning

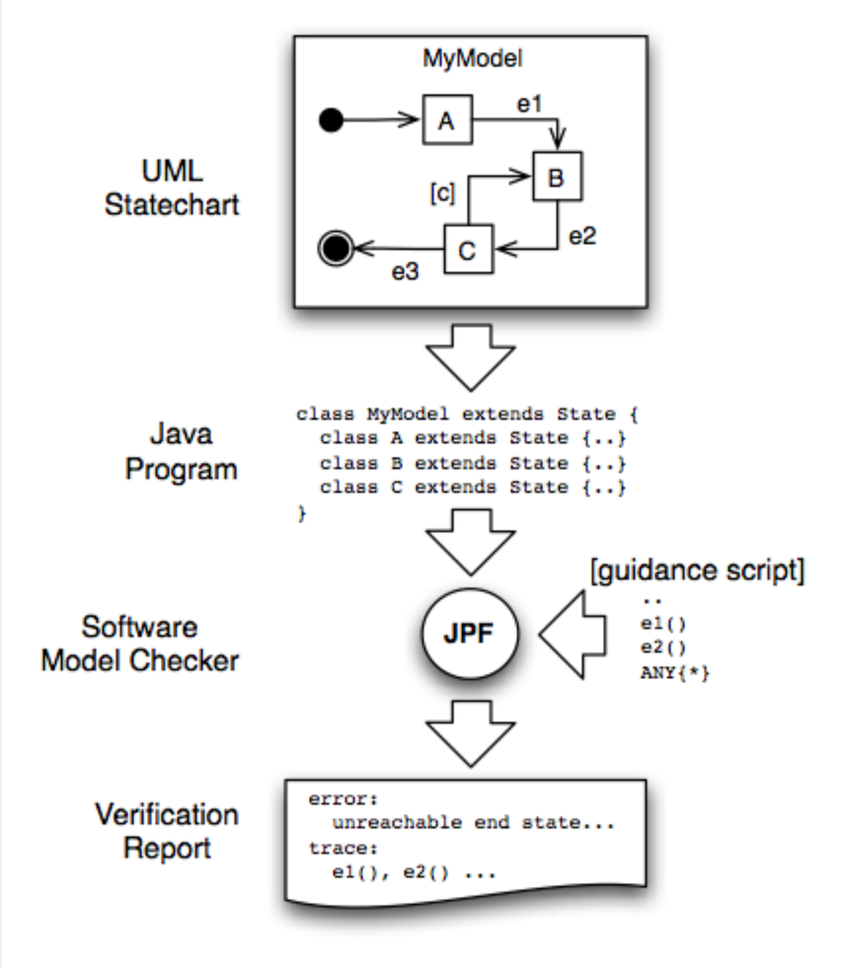
interface generation / discharge

jpf-cv

http://babelfish.arc.nasa.gov/trac/jpf

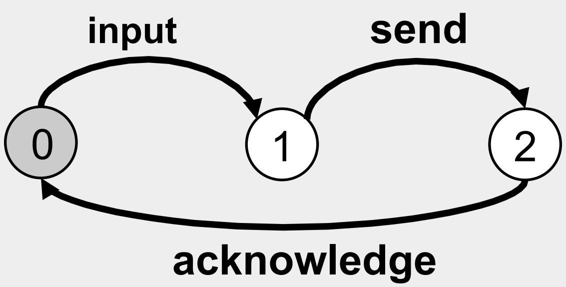
UML framework in JPF

- JPF supports model checking of UML state-machines with an approach that consists of three steps:
  - translate the UML model into a corresponding Java program, using JPF's state chart (sc) extension and application model
  - choose model properties to verify, and configure verification tools accordingly
  - optionally provide a guidance script that represents the environment of the model (external event sequence)

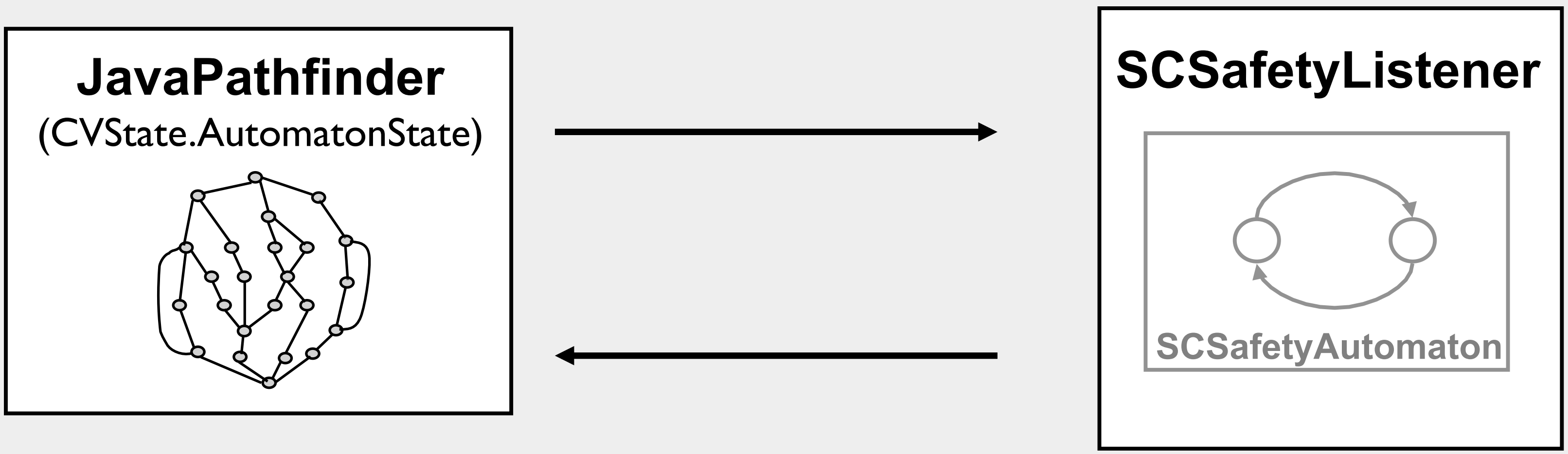


example

```
package ICSETutorial;

import gov.nasa.jpf.sc.State;

public class Input extends State {

  class S0 extends State {

      public void input() {
          setNextState(s1);
      }
  } S0 s0 = makeInitial(new S0());

  class S1 extends State {

      public void send() {
          setNextState(s2);
      }
  } S1 s1 = new S1();

  class S2 extends State {

      public void acknowledge() {
          setNextState(s0);
      }
  } S2 s2 = new S2();
}
```

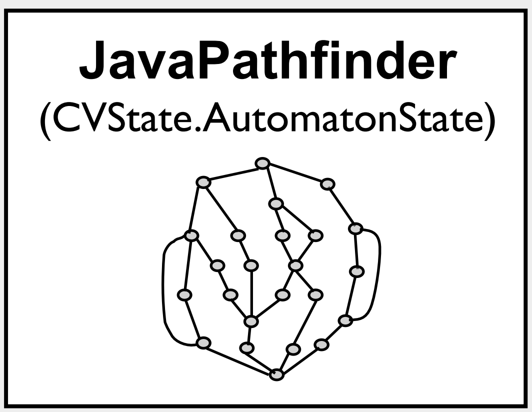


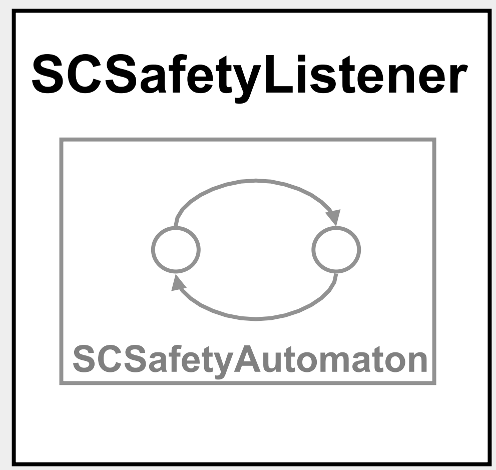
AG reasoning in JPF



assumptions

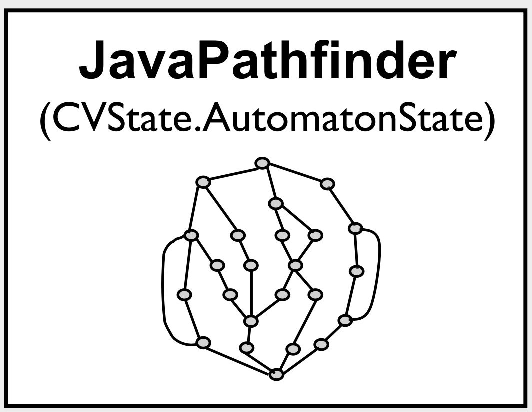
- choiceGeneratorAdvanced
  - if selected action leads assumption to error state then do "vm.getSystemState().setIgnored(true)" (backtrack)
- instructionExecuted
  - advance automaton & set CVState.AutomatonState
- stateBacktracked
  - get CVState.AutomatonState

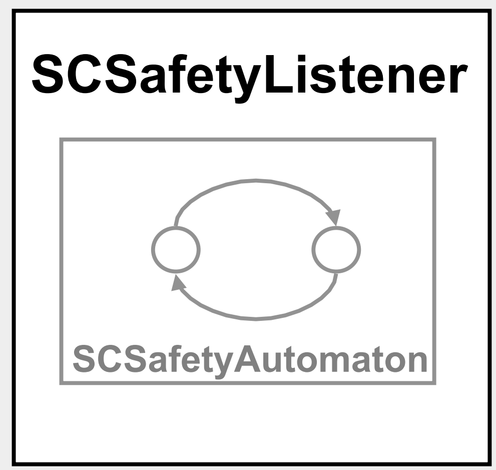




properties

- instructionExecuted
  - advance automaton & set CVState.AutomatonState
  - if automaton reaches error state, then check() returns false
- stateBacktracked
  - get CVState.AutomatonState





interface generation in JPF

- queries and assumption safety checks
  - same as assume-guarantee reasoning
- assumption permissiveness check
  - requires special listener

conformance listener

- executeInstruction
  - if instruction to be executed is assertion violation, then perform "ti.skipInstruction()" (do not process exception) and "vm.getSystemState().setIgnored(true)" (backtrack)
- instructionExecuted
  - advance automaton & set CVState.AutomatonState
  - if automaton reaches error state, check memoized table (why?)

    if counterexample stored and spurious, backtrack

    else check() returns false
- stateBacktracked
  - get CVState.AutomatonState

permissiveness check

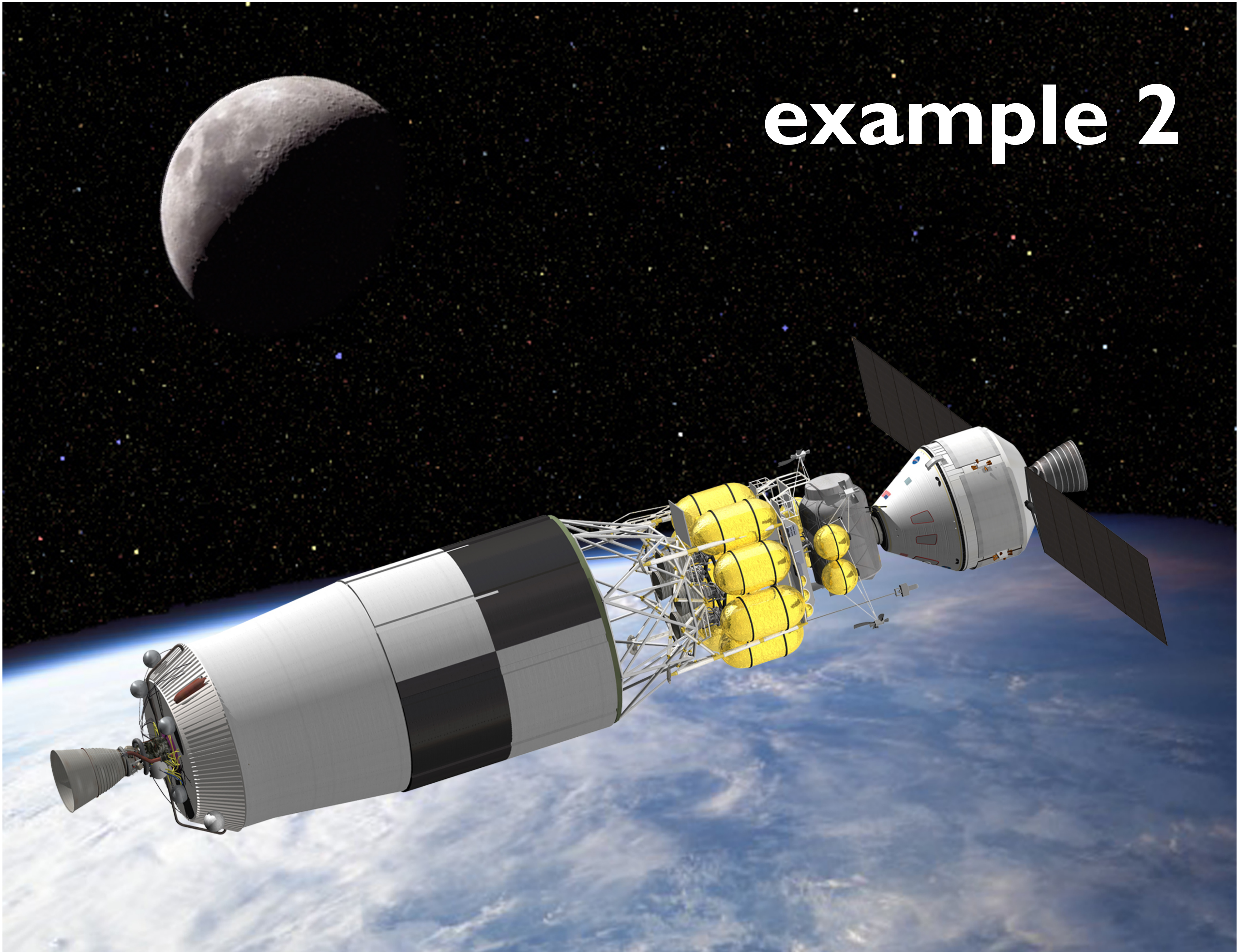
```
boolean done = false;
while (!done){
    counterexample = null;

    …

    SCConformanceListener assumption = new SCConformanceListener(
        new SCSafetyAutomaton(false, assume, alphabet_, "Assumption",
        CompleteModule , memoized_));
    JPF jpf = createJPFInstance(assumption, property, CompleteModule);
    jpf.run();

    Path jpfPath = assumption.getCounterexample();
    if (jpfPath != null){
    //nonerror in M & error in Aerr - this is what we are looking for

        counterexample = assumption.convert(jpfPath);
        if( query(counterexample)){ // cex is in L(A)
            done = true; // a real counterexample for L*
        } // otherwise you need to continue with your loop
    }else
        done = true; // interface is permissive
}
```

input output example

Input component with Order Property:

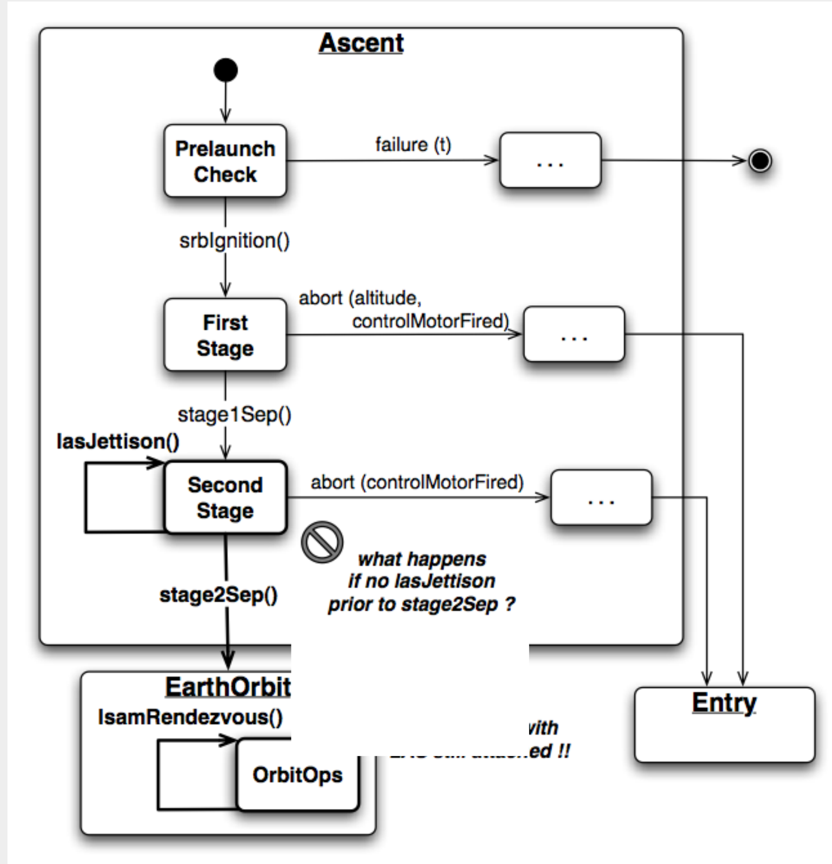
```
package ICSETutorial;

import gov.nasa.jpf.sc.State;
import gov.nasa.jpf.cv.CVState;


public class InputWithProperty
extends CVState {

  class S0 extends State {

      public void input() {
          setNextState(s1);
      }

       public void output() {
          assert(false);
      }

  } S0 s0 = makeInitial(new S0());


  . . .
}
```

```
S0 = ( send -> S2
     | acknowledge -> S1),
S1 = ( output -> S1
     | send -> S1
     | acknowledge -> S1),
S2 = ( output -> S3
     | send -> S1),
S3 = ( send -> S1
     | acknowledge -> S0).
```


example 2

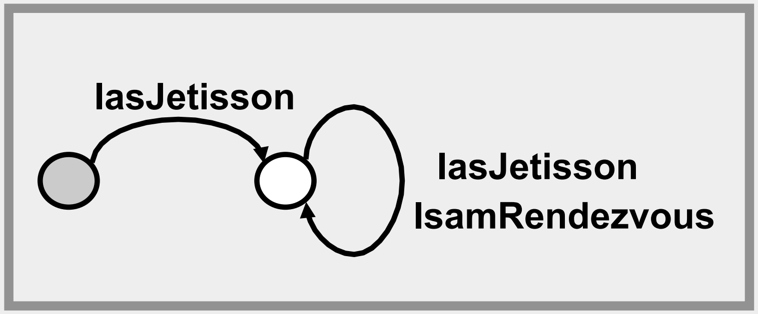
example: crew exploration vehicle

- tool: JavaPathfinder
- UML statechart model of the *Ascent* and *EarthOrbit* flight phases of a spacecraft
- properties:
  - *"An event lsamRendezvous, which represents a docking maneuver with another spacecraft, fails if the LAS (launch abort system) is still attached to the spacecraft"*
  - *"Event tliBurn (trans-lunar interface burn takes spacecraft out of the earth orbit and gets it into transition to the moon) can only be invoked if EDS (Earth Departure Stage) rocket is available"*

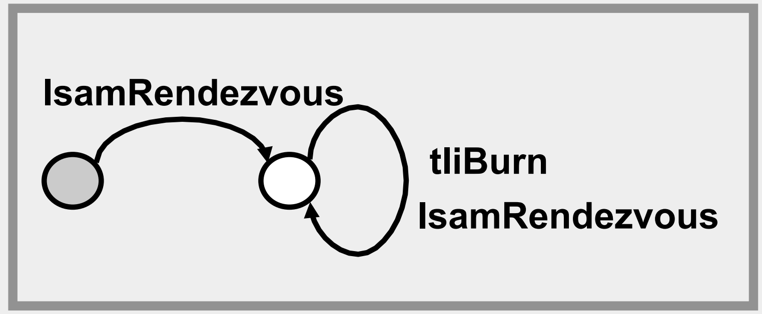


results

Assumption 1:



Assumption 2:



generated interface assumptions encode Flight Rules in terms of events

conclusions

- learning assumptions is not a panacea
  - may perform worse than monolithic verification
  - performs well when alphabets & assumptions are small
- computed interfaces may not be permissive
  - in our studies interfaces were satisfactory
  - there is more to say about this in part IV
- limited to statecharts
  - but wish to extend; it's open source, help us!!!
- got funding
  - so expect a lot of activity on jpf-cv over the next year

Automated Component-Based Verification

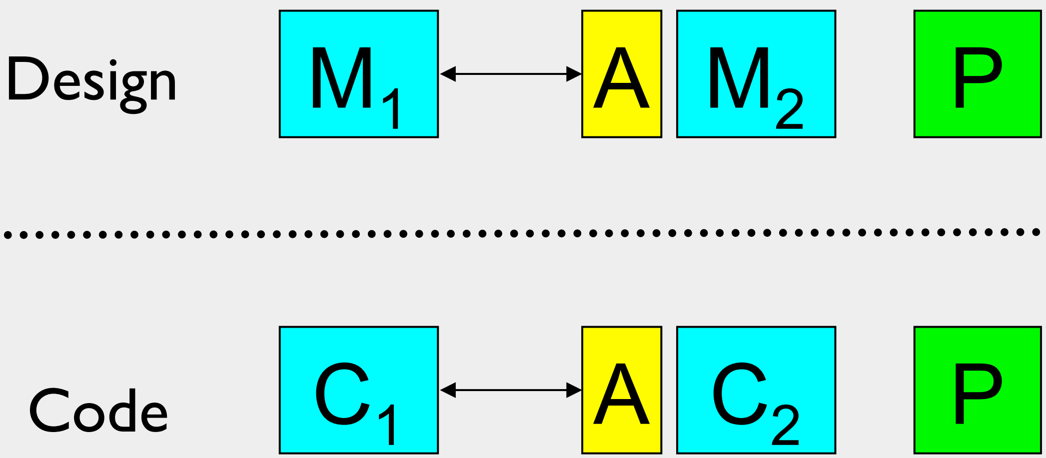
part IV

Dimitra Giannakopoulou and Corina Păsăreanu
CMU / NASA Ames Research Center

part IV

- reasoning about code
- introducing abstraction
  - to reason about very large or infinite state spaces
- related approaches

reasoning about code



- Does $M_1 \parallel M_2$ satisfy P? Model check; **build** assumption A
- Does $C_1 \parallel C_2$ satisfy P? Model check; **use** assumption A

  [ICSE'2004] – good results but may not scale

  Solution: replace model checking with testing!

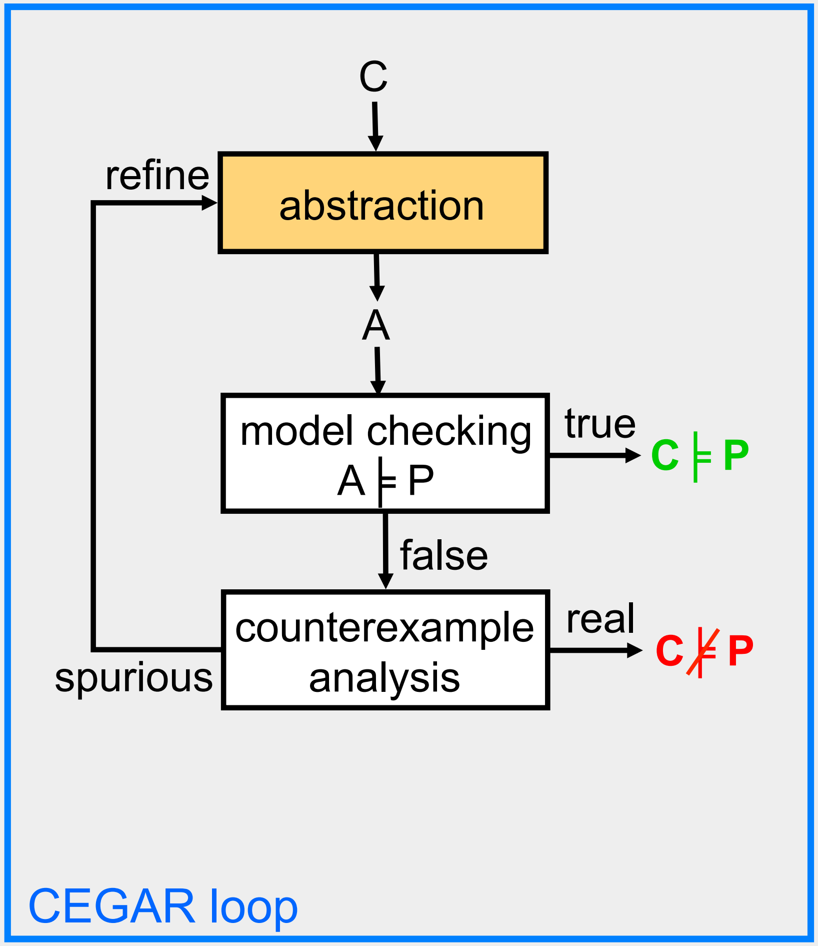
introducing abstraction…

- apply predicate abstraction [Graf & Saidi, CAV 1997]
- apply learning to abstracted components
- use counterexamples to automatically refine abstractions as needed, using CEGAR (Counter-example Guided Abstraction Refinement) [Clarke et al., CAV 2000]

- interfaces: novel combination of under- and over- approximations with L* avoids exponentially expensive determinization step and generates minimal and precise interfaces [CAV 2010]

- implemented in ARMC model checker (and previously Magic)
- successfully applied to several benchmarks (Java2SDK library classes, OpenSSL)

CEGAR for compositional verification



- CEGAR: counterexample guided abstraction refinement – Clarke et al. 00
  - incremental construction of abstractions
  - abstractions are conservative
  - abstract counterexamples obtained may be spurious (due to over-approximation)
  - spurious counterexamples are used for abstraction refinement
- two level compositional abstraction refinement – Chaki et al. 03
  - analyze $C_1 \parallel C_2 \parallel \ldots \parallel C_n \models P$
  - build finite-state abstractions: $A_1, A_2, \ldots A_n$
  - minimize: $M_1, M_2, \ldots M_n$
  - analyze: $M_1 \parallel M_2 \parallel \ldots \parallel M_n \models P$?
  - refine based on counterexamples
- permissive interfaces – Henzinger et al. 05
  - uses CEGAR to compute interfaces

assume-guarantee abstraction refinement (AGAR)

- Challenge: instead of learning A, build A as an abstraction of M_2

$$\begin{array}{llll} 1. & \langle A \rangle & M_1 & \langle P \rangle \\ 2. & \langle \mathit{true} \rangle & M_2 & \langle A \rangle \\ \hline & \multicolumn{3}{l}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle} \end{array}$$

- build A as an abstraction of $M_{2;}$ $\langle \mathit{true} \rangle$ M_2 $\langle A \rangle$ holds by construction
- check Premise 1: $\langle A \rangle$ M_1 $\langle P \rangle$
- obtained counterexamples are analyzed and used to refine A
- variant of CEGAR with differences:
  - use counterexample from M_1 to refine abstraction of M_2
  - A keeps information only about the interface (abstracts away the internal info)
- implemented in LTSA; combined with alphabet refinement;
- compares favorably with learning approach
- [CAV'08]

AGAR vs learning

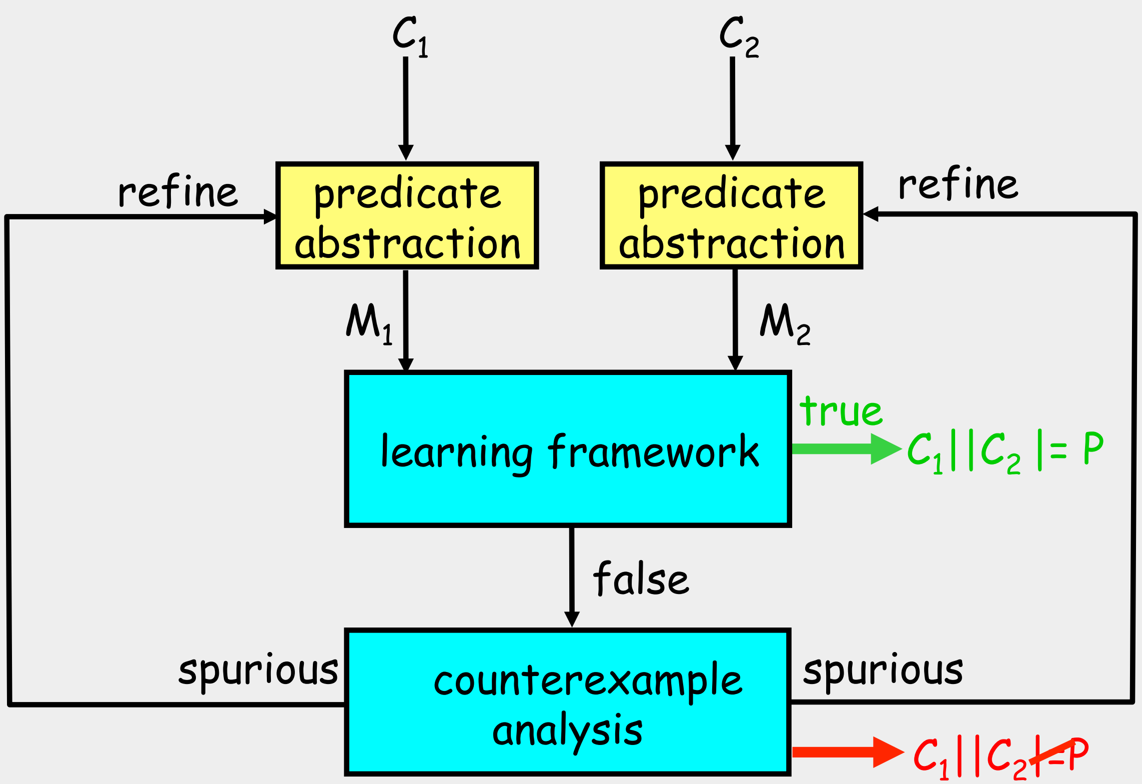
Table 1. Comparison of AGAR and learning for 2 components, with and without alphabet refinement.

Case	k	No alpha. ref. AGAR $\|A\|$	AGAR Mem.	AGAR Time	Learning $\|A\|$	Learning Mem.	Learning Time	With alpha. ref. AGAR $\|A\|$	AGAR Mem.	AGAR Time	Learning $\|A\|$	Learning Mem.	Learning Time	Sizes $\|M_1 \parallel P_{err}\|$	$\|M_2\|$
Gas Station	3	**16**	**4.11**	**3.33**	177	42.83	–	**5**	**2.99**	**2.09**	8	3.28	3.40	1960	643
	4	**19**	**37.43**	**23.12**	195	100.17	–	**5**	**22.79**	**12.80**	8	25.21	19.46	16464	1623
	5	**22**	**359.53**	**278.63**	45	206.61	–	**5**	216.07	**83.34**	8	**207.29**	188.98	134456	3447
Chiron, Property 2	2	10	**1.30**	**0.92**	9	**1.30**	1.69	10	1.30	**1.56**	**8**	**1.22**	5.17	237	102
	3	36	**2.59**	**5.94**	**21**	5.59	7.08	36	**2.44**	**10.23**	**20**	6.00	30.75	449	1122
	4	160	**8.71**	152.34	**39**	27.1	**32.05**	160	**8.22**	252.06	**38**	41.50	**180.82**	804	5559
	5	4	55.14	–	**111**	**569.23**	**676.02**	3	58.71	–	110	–	386.6	2030	129228
Chiron, Property 3	2	**4**	**1.07**	**0.50**	9	1.14	1.57	4	1.23	**0.62**	**3**	**1.06**	0.91	258	102
	3	**8**	**1.84**	**1.60**	25 n jmj	4.45	7.72	8	**2.00**	3.65	**3**	2.28	**1.12**	482	1122
	4	**16**	**4.01**	**18.75**	45	25.49	36.33	16	**5.08**	107.50	**3**	**7.30**	1.95	846	5559
	5	4	52.53	–	**122**	**134.21**	**271.30**	1	81.89	–	**3**	**163.45**	**19.43**	2084	129228
MER	2	**34**	**1.42**	11.38	40	6.75	**9.89**	**5**	**1.42**	5.02	6	1.89	**1.28**	143	1270
	3	**67**	**8.10**	**247.73**	335	133.34	–	9	11.09	180.13	**8**	**8.78**	**12.56**	6683	7138
	4	58	341.49	–	38	377.21	–	9	532.49	–	**10**	**489.51**	**1220.62**	307623	22886
Rover Exec.	2	**10**	**4.07**	**1.80**	11	2.70	2.35	**3**	2.62	**2.07**	4	2.46	3.30	544	41

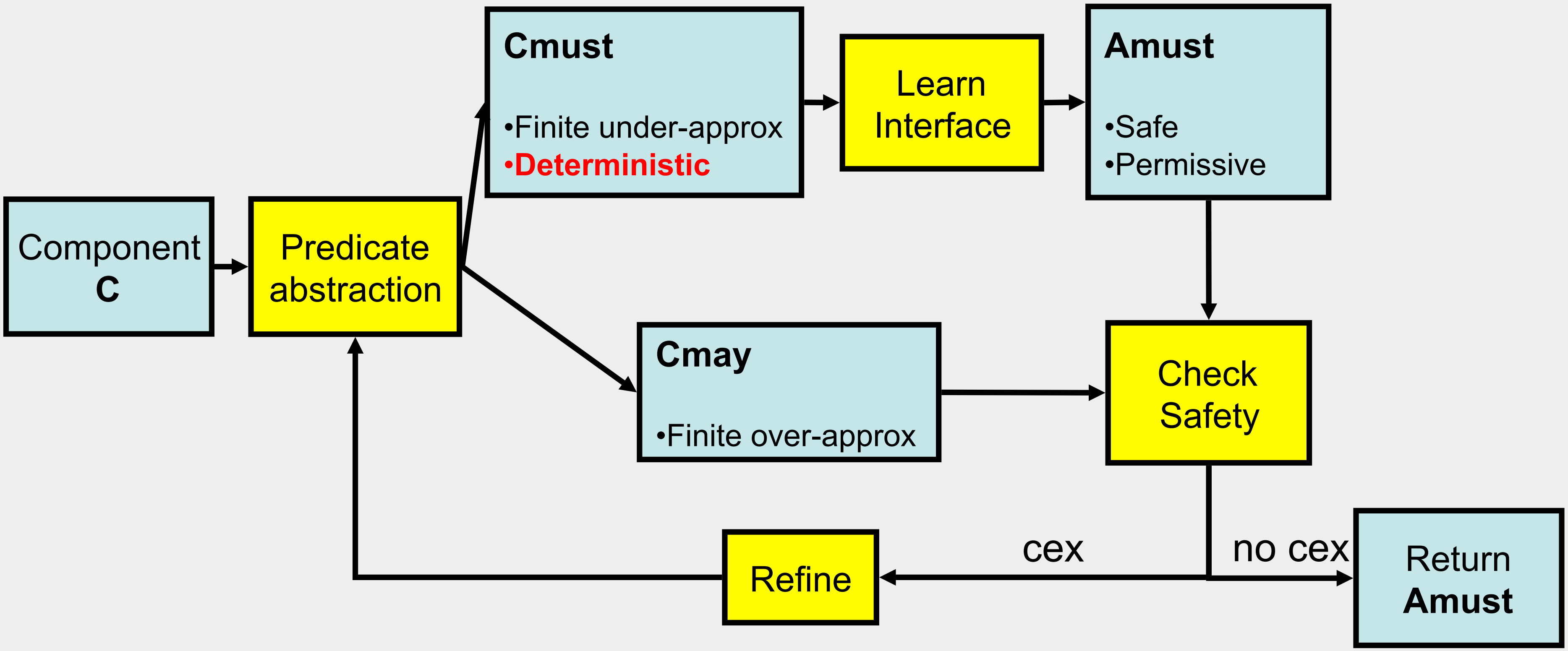
compositional verification for C

- Check composition of software C components

$$C_1||C_2 \models P$$



interface generation for infinite-state components



Theorem:

*An interface **A** permissive w.r.t **C**'s **must** abstraction safe w.r.t **C**'s **may** abstraction and is safe and permissive for C.*

interface generation for infinite-state components

- conceptually simple and elegant
- expensive learning restarts

- need of tighter integration of abstraction refinement with L*
- LearnReuse method

Query(σ, C)

1. if checkSafe(σ,Cmust) != null
2. return “no”
3. cex = checkSafe(σ,Cmay)
4. if cex == null
5. return “yes”
6. Preds = Preds U Refine(cex)
7. Query(σ, C)

Conjecture : Oracle 1

```
1.  cex = checkSafe(A, Cmay)
2.  if cex == null
3.          invoke Oracle2
4.  If Query(cex, C) == "no"
5.          return cex to L*
6.  else
7.  goto 1
```

Conjecture : Oracle 2

1. cex = checkPermissive(A, Cmust)
2. if cex == null
3.   return A
4. If Query(cex, C) == “yes”
5.   return cex to L*
6. else
7. goto 1

NASA case study

- NASA CEV 1.5 EOR-LOR mission
- 26 methods

- Only LearnReuse finished
- 74 predicates, 14 states
- 52 minutes

our previous work at a glance

- learning-based AG reasoning (TACAS 2003)
- recursive application of simple rule for reasoning about n > 2 components (FMSD 2009)
- symmetric and circular assume-guarantee rules (SAVCBS 2003, FMSD 2009)
- assume-guarantee reasoning for code (ICSE 2004, SAVCBS 2005, IET Software 2009)
- learning with alphabet refinement (TACAS 2007)
- learning assumptions for interface automata (FM 2008)
- assume-guarantee abstraction refinement (CAV 2008)
- interface generation in JPF (FASE 2009)
- interface generation for large/infinite-state components (CAV 2010)

other related work

(interfaces)

Alur et al, 2005 (1)
Henzinger et al, 2005

Ammons et al, 2002
Whaley et al, 2002
Tkachuk et al, 2003

(L for separating automata)*

Gupta et al, 2007
Chen et al, 2009

(L for AG reasoning)*

Alur et al, 2005 (2)
Chaki et al, 2005-2007
Cobleigh et al, 2006

(L for NFAs & liveness)*

Bollig et al, 2009
Farzan et al, 2008

(L for model extraction)*

Groce et al, 2002
Margaria et al, 2007

other related work

- minimal separating automaton for disjoint languages L_1 and L_2
  - accept all words in L_1
  - accept no words in L_2
  - have the least number of states
- assume-guarantee reasoning
  - minimal separating automaton for $L(M_2)$ and $L(M_1) \cap L(coP)$
- algorithms
  - Gupta at al. 07: query complexity exponential in the size of the minimal DFAs for the two input languages
  - Chen et al. 09: query complexity quadratic in the product of the sizes of the minimal DFAs for the two input languages. Use 3 valued DFAs

- compositional verification in symbolic setting (Alur et al. 05)
- learning omega-regular languages for liveness (Farzan et al. 08)
- learning non-deterministic automata (Bollig et al. 09)

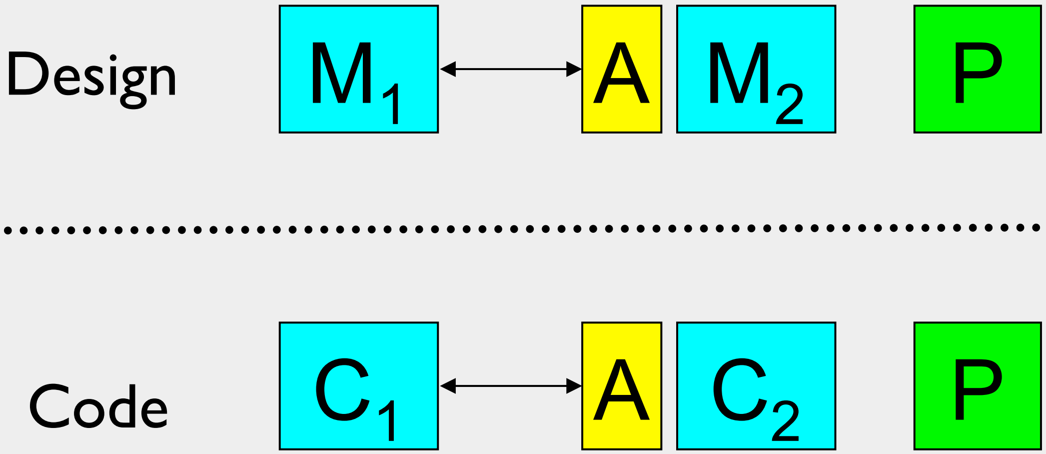
thank you!

part IV

- parts I – III: reasoning about finite-state models

- reasoning about code
- introducing abstraction
  to reason about very large or infinite state spaces
- related approaches

reasoning about code



- Does $M_1 \| M_2$ satisfy P? Model check; **build** assumption A
- Does $C_1 \| C_2$ satisfy P? Model check; **use** assumption A

  [ICSE'2004] – good results but may not scale

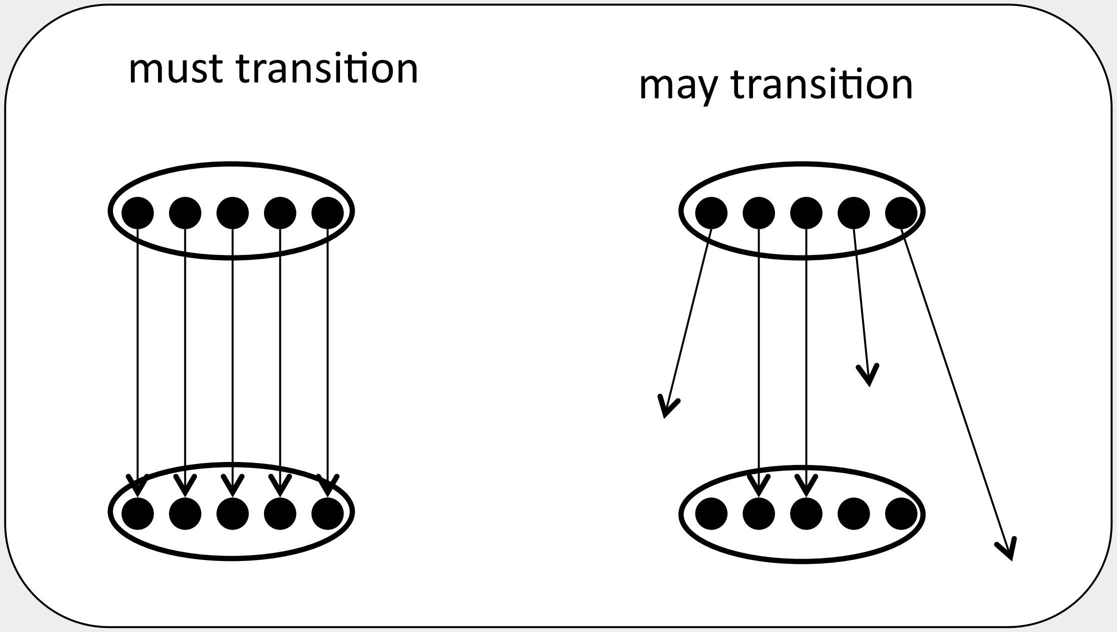
  Solution: replace model checking with testing! [IET Software 2009]

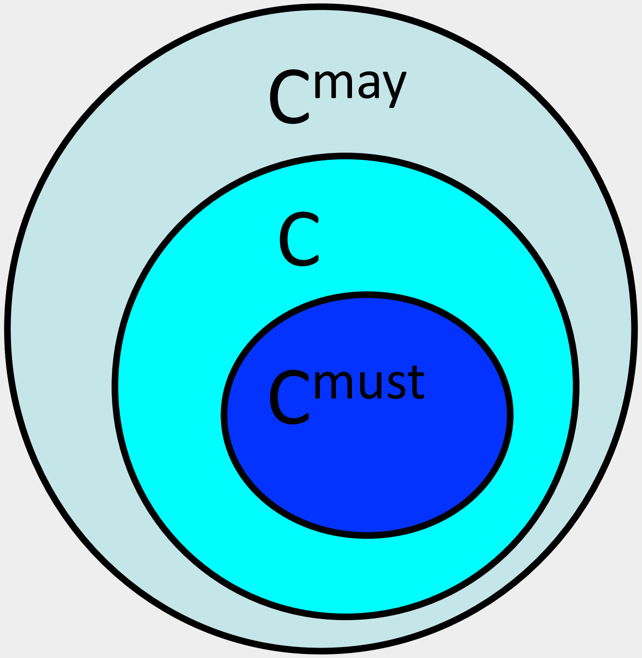
abstraction

- Reduces large/infinite data domains into small/finite abstract domains; e.g. replace *int* with {*ZERO, POS, NEG*}
- Produces a finite state abstract model that operates on the abstract domain
- Abstraction maps
  - Concrete states to abstract states
  - Concrete transitions to abstract transitions
- Framework of abstract interpretation

may and must abstraction

- May abstraction produces a finite over-approximation
- Must abstraction produces a finite under-approximation





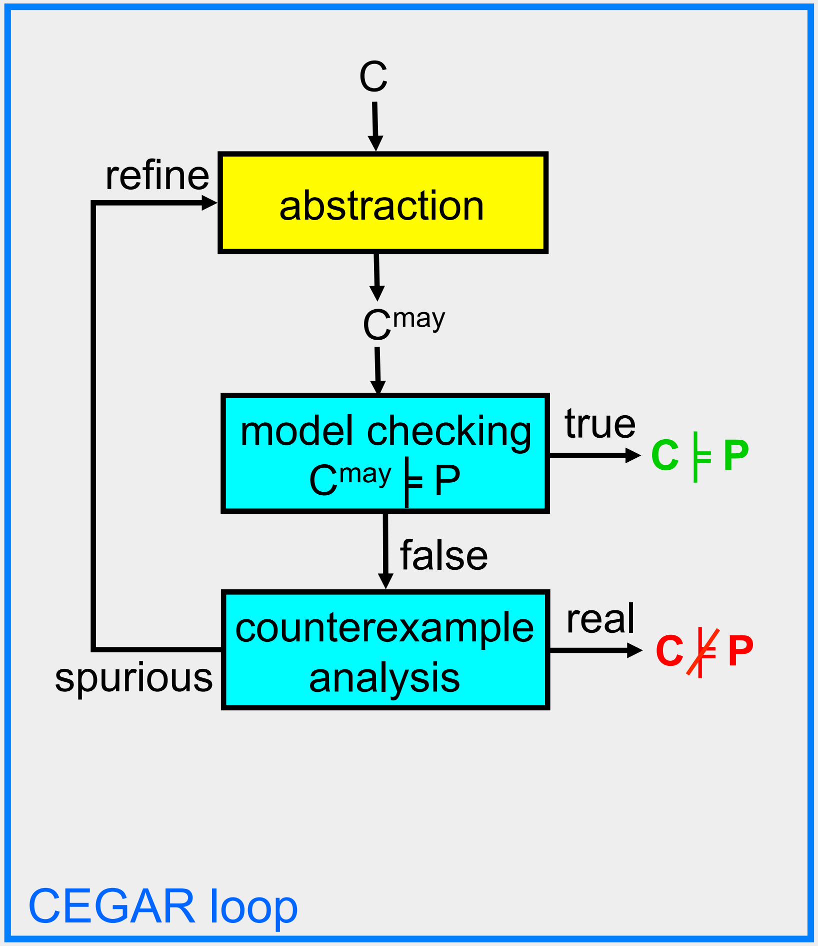
abstraction in compositional verification

- apply (predicate) abstraction [Graf & Saidi, CAV 1997]
- apply learning to abstracted components
- use counterexamples to automatically refine abstractions/assumptions as needed [Magic]

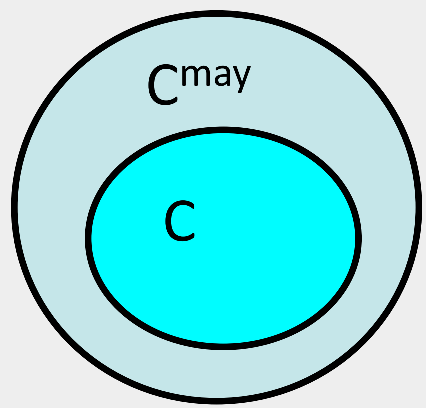
- use abstraction refinement as an alternative to learning for building assumptions [AGAR, CAV 2008]

- interfaces: novel combination of under- and over- approximations with L* avoids exponentially expensive determinization step and generates minimal and precise interfaces [CAV 2010]
- implemented in ARMC model checker
- successfully applied to several benchmarks (Java2SDK library classes, OpenSSL)

CEGAR



- CEGAR: counterexample guided abstraction refinement [Clarke et al. 00]
  - incremental construction of (may) abstractions
  - abstract counterexamples obtained may be spurious (due to over-approximation)
  - spurious counterexamples are used for abstraction refinement

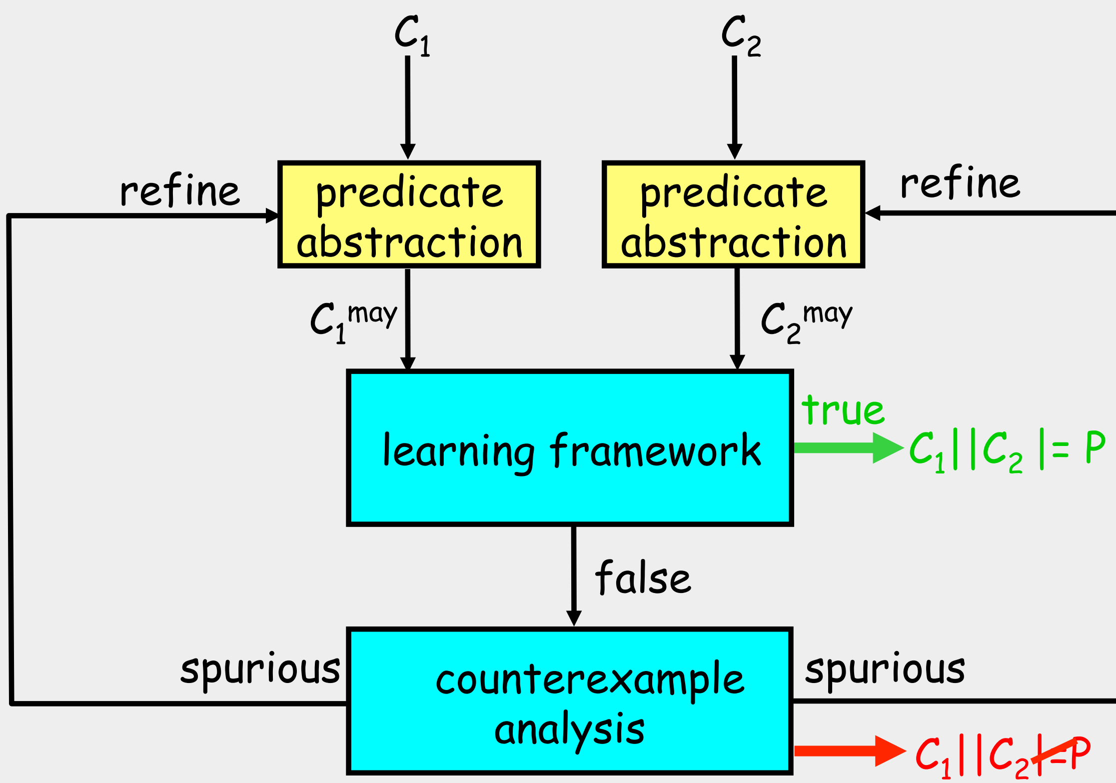


CEGAR for compositional verification

- two level compositional abstraction refinement – Chaki et al. 03
  - analyze $C_1 \parallel C_2 \parallel \ldots \parallel C_n \models P$
  - build finite-state abstractions: $A_1, A_2, \ldots A_n$
  - minimize: $M_1, M_2, \ldots M_n$
  - analyze: $M_1 \parallel M_2 \parallel \ldots \parallel M_n \models P$?
  - refine based on counterexamples
- permissive interfaces – Henzinger et al. 05
  - uses CEGAR to compute interfaces

learning-based compositional verification for C code

- Check composition of software C components $C_1||C_2 \models P$
- C_1, C_2 are large/infinite state



assume-guarantee abstraction refinement (AGAR)

- Challenge: instead of learning A, build A as an abstraction of M_2

$$
\begin{array}{llll}
1. & \langle A \rangle & M_1 & \langle P \rangle \\
2. & \langle \mathit{true} \rangle & M_2 & \langle A \rangle \\
\hline
 & \langle \mathit{true} \rangle & M_1 \parallel M_2 & \langle P \rangle
\end{array}
$$

assume-guarantee abstraction refinement (AGAR)

- Challenge: instead of learning A, build A as an abstraction of M_2

$$\begin{array}{llll} 1. & \langle A \rangle & M_1 & \langle P \rangle \\ 2. & \langle \mathit{true} \rangle & M_2 & \langle A \rangle \\ \hline & \multicolumn{3}{l}{\langle \mathit{true} \rangle\ M_1 \parallel M_2\ \langle P \rangle} \end{array}$$

- build A as an (may) abstraction of $M_{2;}$ $\langle \mathit{true} \rangle$ M_2 $\langle A \rangle$ holds by construction
- check Premise 1: $\langle A \rangle$ M_1 $\langle P \rangle$
- obtained counterexamples are analyzed and used to refine A
- variant of CEGAR with differences:
  - use counterexample from M_1 to refine abstraction of M_2
  - A keeps information only about the interface (abstracts away the internal info)
- implemented in LTSA; combined with alphabet refinement;
- compares favorably with learning approach
- [CAV'08]

AGAR vs learning

Table 1. Comparison of AGAR and learning for 2 components, with and without alphabet refinement.

Case	k	No alpha. ref.						With alpha. ref.						Sizes	
		AGAR			Learning			AGAR			Learning				
		$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|A\|$	Mem.	Time	$\|M_1 \\| P_{err}\|$	$\|M_2\|$
Gas Station	3	**16**	**4.11**	**3.33**	177	42.83	–	**5**	**2.99**	**2.09**	8	3.28	3.40	1960	643
	4	**19**	**37.43**	**23.12**	195	100.17	–	**5**	**22.79**	**12.80**	8	25.21	19.46	16464	1623
	5	**22**	**359.53**	**278.63**	45	206.61	–	**5**	216.07	**83.34**	8	**207.29**	188.98	134456	3447
Chiron, Property 2	2	10	**1.30**	**0.92**	9	**1.30**	1.69	10	1.30	**1.56**	**8**	**1.22**	5.17	237	102
	3	36	**2.59**	**5.94**	**21**	5.59	7.08	36	**2.44**	**10.23**	**20**	6.00	30.75	449	1122
	4	160	**8.71**	152.34	**39**	27.1	**32.05**	160	**8.22**	252.06	**38**	41.50	**180.82**	804	5559
	5	4	55.14	–	**111**	**569.23**	**676.02**	3	58.71	–	110	–	386.6	2030	129228
Chiron, Property 3	2	**4**	**1.07**	**0.50**	9	1.14	1.57	4	1.23	**0.62**	**3**	**1.06**	0.91	258	102
	3	**8**	**1.84**	**1.60**	25 n jmj	4.45	7.72	8	**2.00**	3.65	**3**	2.28	**1.12**	482	1122
	4	**16**	**4.01**	**18.75**	45	25.49	36.33	16	**5.08**	107.50	**3**	**7.30**	1.95	846	5559
	5	4	52.53	–	**122**	**134.21**	**271.30**	1	81.89	–	**3**	**163.45**	**19.43**	2084	129228
MER	2	**34**	**1.42**	11.38	40	6.75	**9.89**	**5**	**1.42**	5.02	6	1.89	**1.28**	143	1270
	3	**67**	**8.10**	**247.73**	335	133.34	–	9	11.09	180.13	**8**	**8.78**	**12.56**	6683	7138
	4	58	341.49	–	38	377.21	–	9	532.49	–	**10**	**489.51**	**1220.62**	307623	22886
Rover Exec.	2	**10**	**4.07**	**1.80**	11	2.70	2.35	**3**	2.62	**2.07**	4	2.46	3.30	544	41

interface generation for infinite-state components

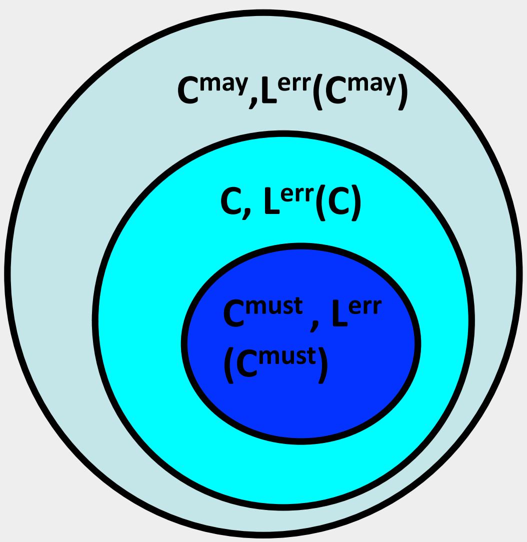
- Use predicate abstraction to build may and must abstractions of component C
- $L^{safe}(C)=\overline{L^{err}(C)}$

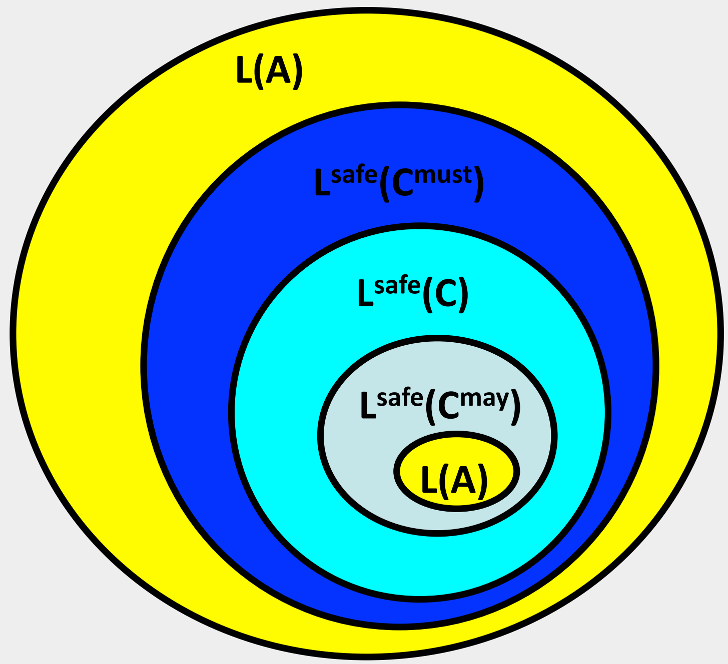
  Interface A is safe: $L(A) \subseteq L^{safe}(C)$

  Interface A is permissive: $L^{safe}(C) \subseteq L(A)$

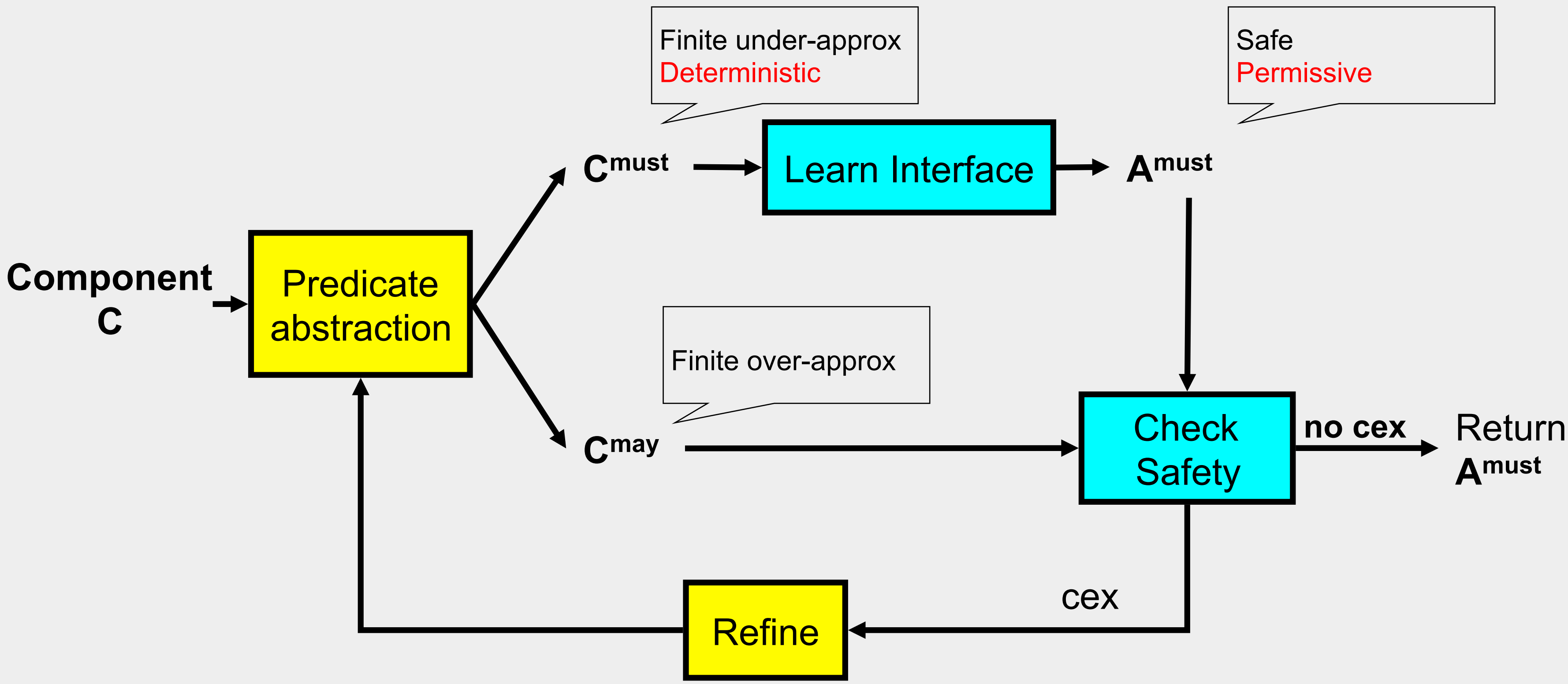
- <u>*Theorem:*</u>

  *An interface **A** permissive w.r.t. **C**'s **must** abstraction and safe w.r.t **C**'s **may** abstraction is safe and permissive for C.*





interface generation for infinite-state components



Correctness:

If algorithm terminates then the returned interface A is safe and permissive for C.

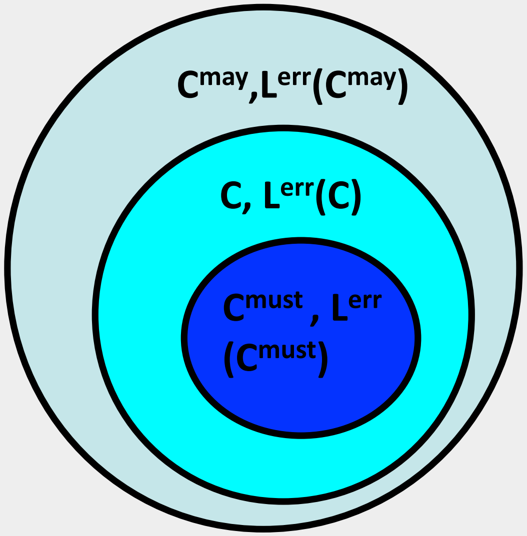
interface generation for infinite-state components

- conceptually simple and elegant
- expensive learning restarts

- need of tighter integration of abstraction refinement with L*
- LearnReuse method

Query(σ, C)

1. if checkSafe(σ,C^{must}) != null
2. return "no"
3. cex = checkSafe(σ,C^{may})
4. if cex == null
5. return "yes"
6. Preds = Preds U Refine(cex)
7. Query(σ, C)



Gives answers consistent with C

Conjecture : Oracle 1

```
1.  cex = checkSafe(A, C^may)
2.  if cex == null
3.          invoke Oracle2
4.  If Query(cex, C) == “no”
5.          return cex to L*
6.  else
7.  goto 1
```

Conjecture : Oracle 2

1. cex = checkPermissive(A, C^{must})
2. if cex == null
3. return A
4. If Query(cex, C) == “yes”
5. return cex to L*
6. else
7. goto 1

NASA case study

- NASA CEV 1.5 EOR-LOR mission
- 26 methods

- Only LearnReuse finished
- 74 predicates, 14 states
- 52 minutes

our previous work at a glance

- learning-based AG reasoning (TACAS 2003)
- recursive application of simple rule for reasoning about n > 2 components (FMSD 2009)
- symmetric and circular assume-guarantee rules (SAVCBS 2003, FMSD 2009)
- assume-guarantee reasoning for code (ICSE 2004, SAVCBS 2005, IET Software 2009)
- learning with alphabet refinement (TACAS 2007)
- learning assumptions for interface automata (FM 2008)
- assume-guarantee abstraction refinement (CAV 2008)
- interface generation in JPF (FASE 2009)
- interface generation for large/infinite-state components (CAV 2010)

other related work

(interfaces)

Alur et al, 2005 (1)
Henzinger et al, 2005

Ammons et al, 2002
Whaley et al, 2002
Tkachuk et al, 2003

(L for separating automata)*

Gupta et al, 2007
Chen et al, 2009

(L for AG reasoning)*

Alur et al, 2005 (2)
Chaki et al, 2005-2007
Cobleigh et al, 2006

(L for NFAs & liveness)*

Bollig et al, 2009
Farzan et al, 2008

(L for model extraction)*

Groce et al, 2002
Margaria et al, 2007

other related work

- minimal separating automaton for disjoint languages L_1 and L_2
  - accept all words in L_1
  - accept no words in L_2
  - have the least number of states
- assume-guarantee reasoning
  - minimal separating automaton for $L(M_2)$ and $L(M_1) \cap L(coP)$
- algorithms
  - Gupta at al. 07: query complexity exponential in the size of the minimal DFAs for the two input languages
  - Chen et al. 09: query complexity quadratic in the product of the sizes of the minimal DFAs for the two input languages. Use 3 valued DFAs

- compositional verification in symbolic setting (Alur et al. 05)
- learning omega-regular languages for liveness (Farzan et al. 08)
- learning non-deterministic automata (Bollig et al. 09)

conclusion

- Compositional verification and assume-guarantee reasoning
- Techniques for automatic assumption generation and compositional verification
  - Finite state systems and safety properties
- Data abstraction to deal with very large/infinite state spaces
- Techniques are promising in practice

Future:

- Techniques for discovering good system decompositions
- Parallelization for increased scalability
- Beyond safety: liveness, timed properties, probabilistic reasoning
- Run-time analysis
- More?

thank you!