Components based systems with Probabilistic choice

Doron Peled

Bar Ilan University

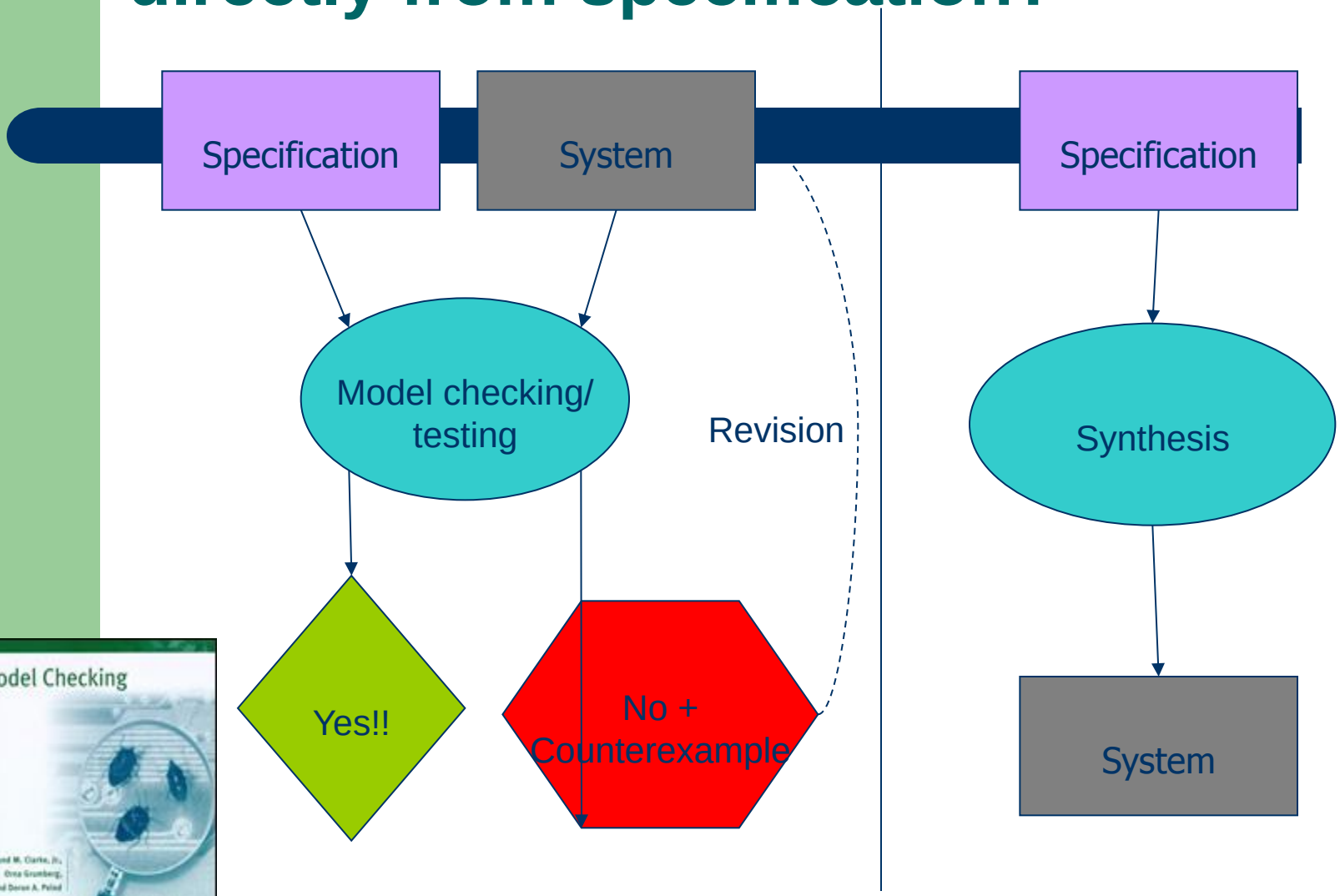
Structure

- Development of reliable systems.
- Implementing components based systems.
- Components based systems with probabilistic choices.

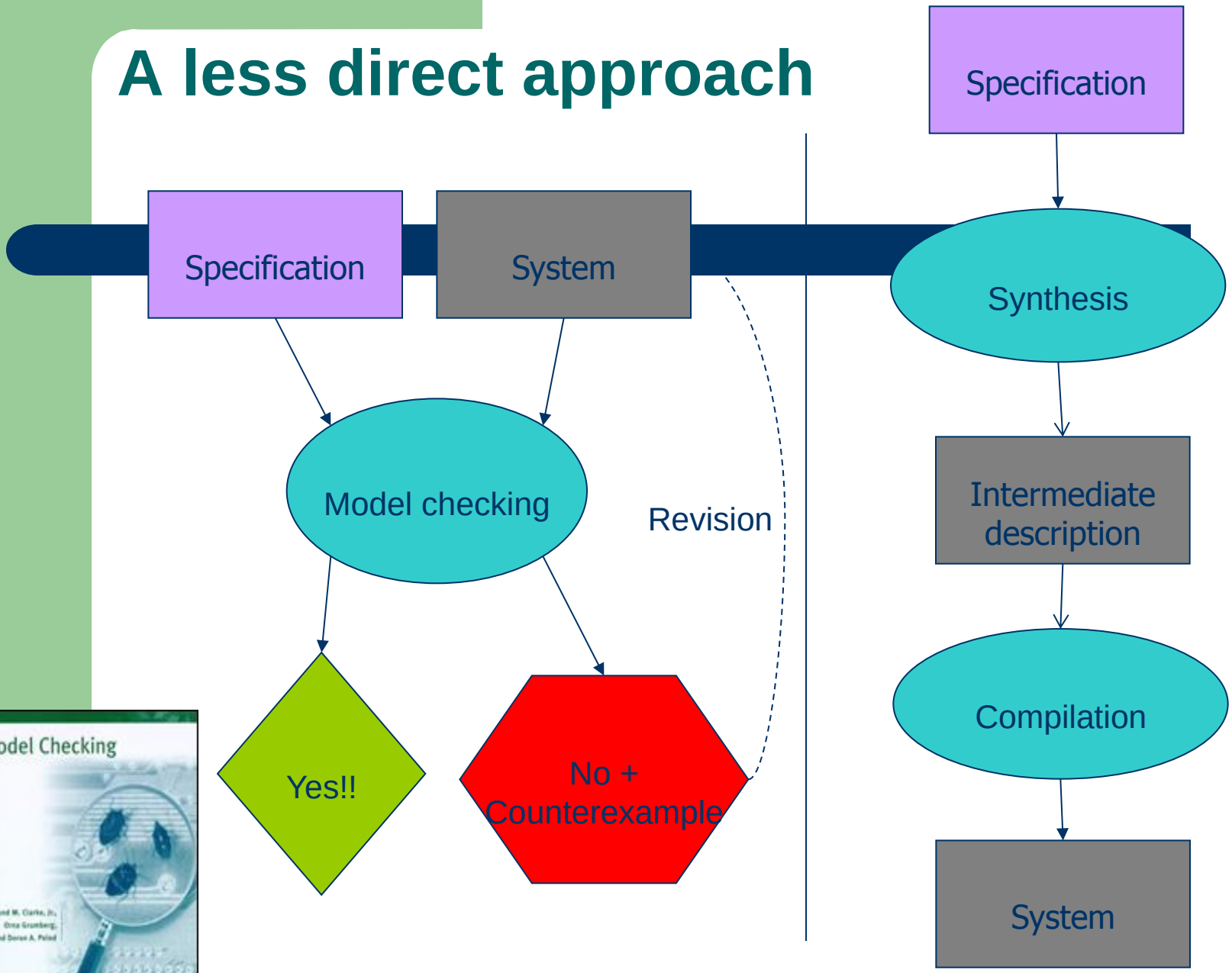
History of (concurrent) software development

- Early software development: programming [Ada Lovelance].
- Testing introduced into the software development cycle (inspection, walkthrough) [Meyer].
- Program verification is invented [Floyd, Hoare].
- Automatic verification (model checking) is invented [Clarke+Emerson, Quelle+Sifakis].
- Automatic synthesis?

Why not synthesize the software directly from specification?

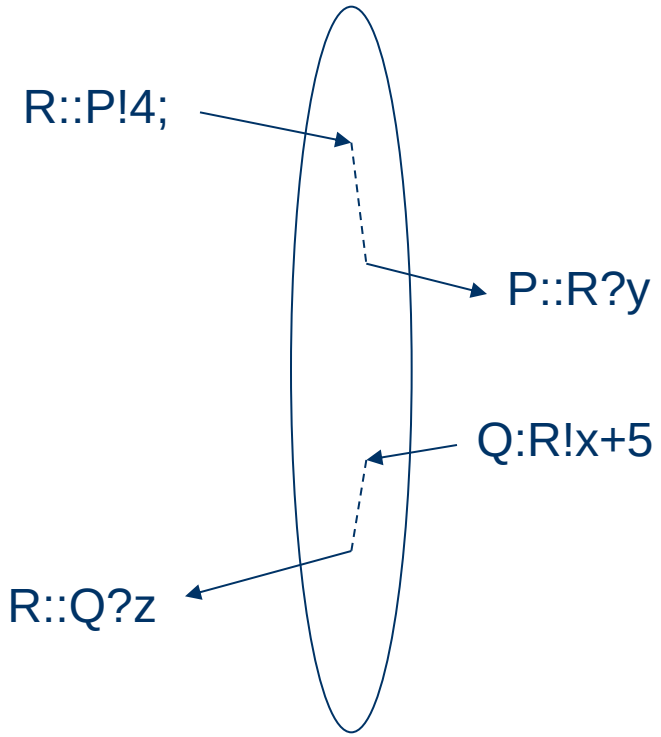


A less direct approach



In fact, synthesis was attempted quite early

- First model checking paper by [Clark+Emerson] was also about synthesis.
- Similar idea (different formalism) by [Manna+Wolper].
- Translate the specification into an automaton (tableau) that accepts the same executions. Then translate this automaton into code.
- The code is **sequential**, or controlled by a **centralized** scheduler.



Reactive sequential systems synthesis

- Problem [Buchi, Rabin]: How to automatically construct an automaton that interacts with an uncontrolled environment and guarantees that together they will satisfy some regular property.
- [Pnueli+Rosner]: Similar for the allowed executions to satisfy some temporal (LTL) property*.
- Solution: Translation (LTL→Automata) + Determinization + Game theory construction.

*This is a subproblem of the above, as LTL properties are star-free regular.

Complexity of sequential synthesis is high

- 2EXPTIME Complete for LTL specification.
- … But, there are provable systems where the number of states is doubly exponential.
- But must the size of a circuit that implements such a system be also doubly exponential?
- [Fearnly+Peled+Schewe]:
  If we knew, we could have decided whether EXPSPACE=2EXPTIME or not.

Concurrent synthesis

- Several processes, with some communication architecture. We want the system to satisfy some LTL property.
- [Pnueli+Rosner]: It is undecidable even to check whether there is a system with the given architecture that satisfies the LTL property.
- But under some strong assumptions (e.g., hierarchical systems) we can solve this [Pnueli+Rosner], [Finkbeiner+Schewe], [Thiagarajan+Madhusudan], [Kupferman+Vardi].

Mostly, negative results about synthesis of concurrent systems.

- Few positive results: …, it is decidable for some very limited architectures, mostly when there is a hierarchy between the processes.
- … in these cases, the complexity is very high …

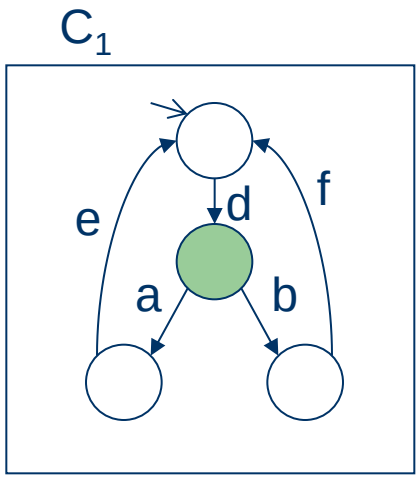
A SHORT HISTORY OF MODERNIST PAINTING" (DETAIL) MARK TANSEY

Alternative

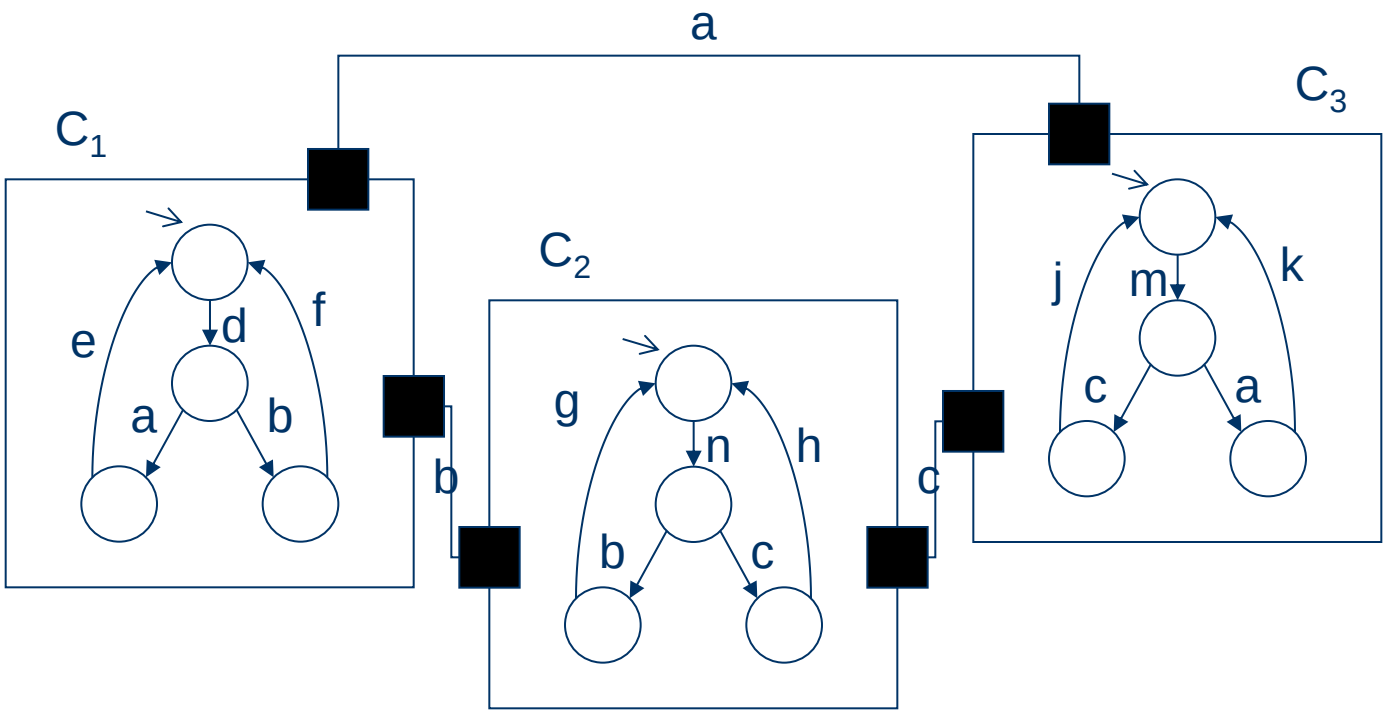
- Allow a formalism that is high level but already includes a distribution of the tastks among autonomous components.
- Provides a simple automatic way to transform the design into implementation.

Component based systems

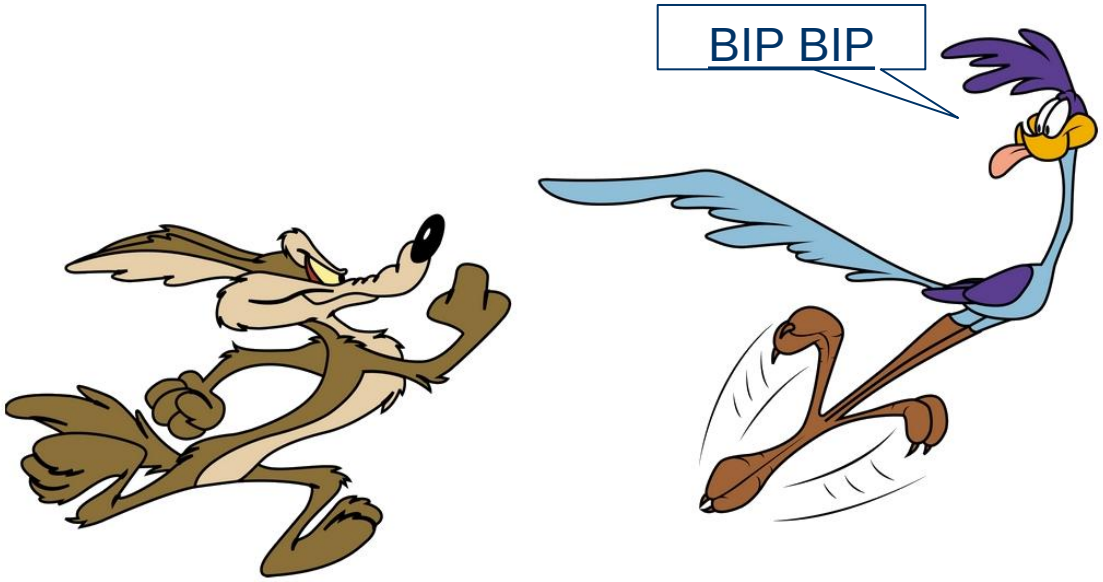
A component (process) is an automaton. From the marked state, *a* and *b* are enabled locally.



Components may share transitions, on which they coordinate.

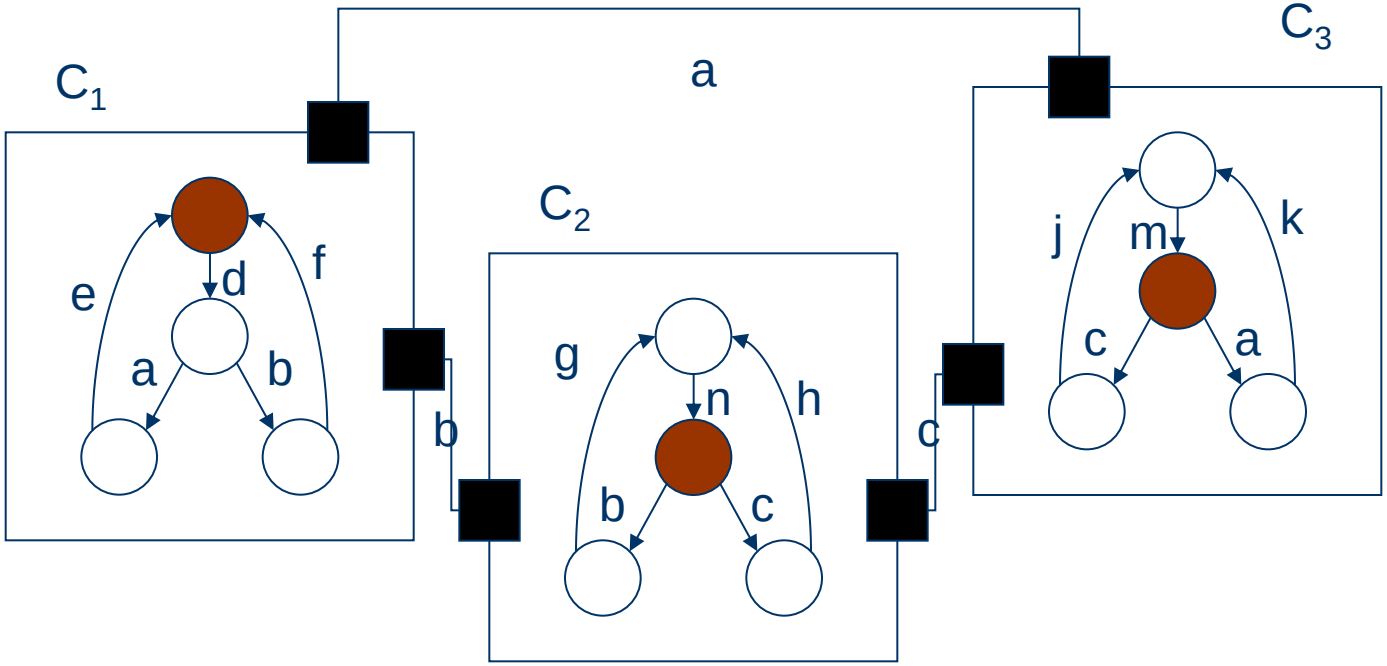


Verimag's BIP (Behavior Interactions Priorities) are component based systems!



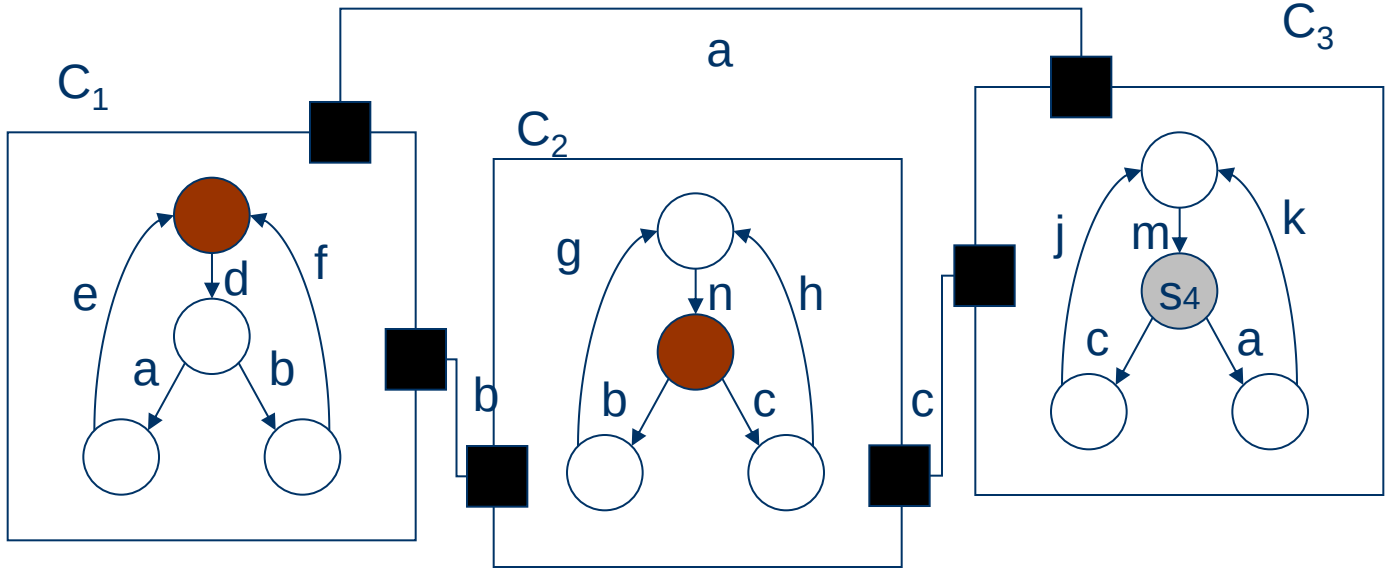
A global state is a combination of local states

From the given global state, *d*, *n* and *c* are enabled.



The local view of a component in a global state is the *local state + enabled actions*.

The local view of C3 here is <S_4,{c}>.

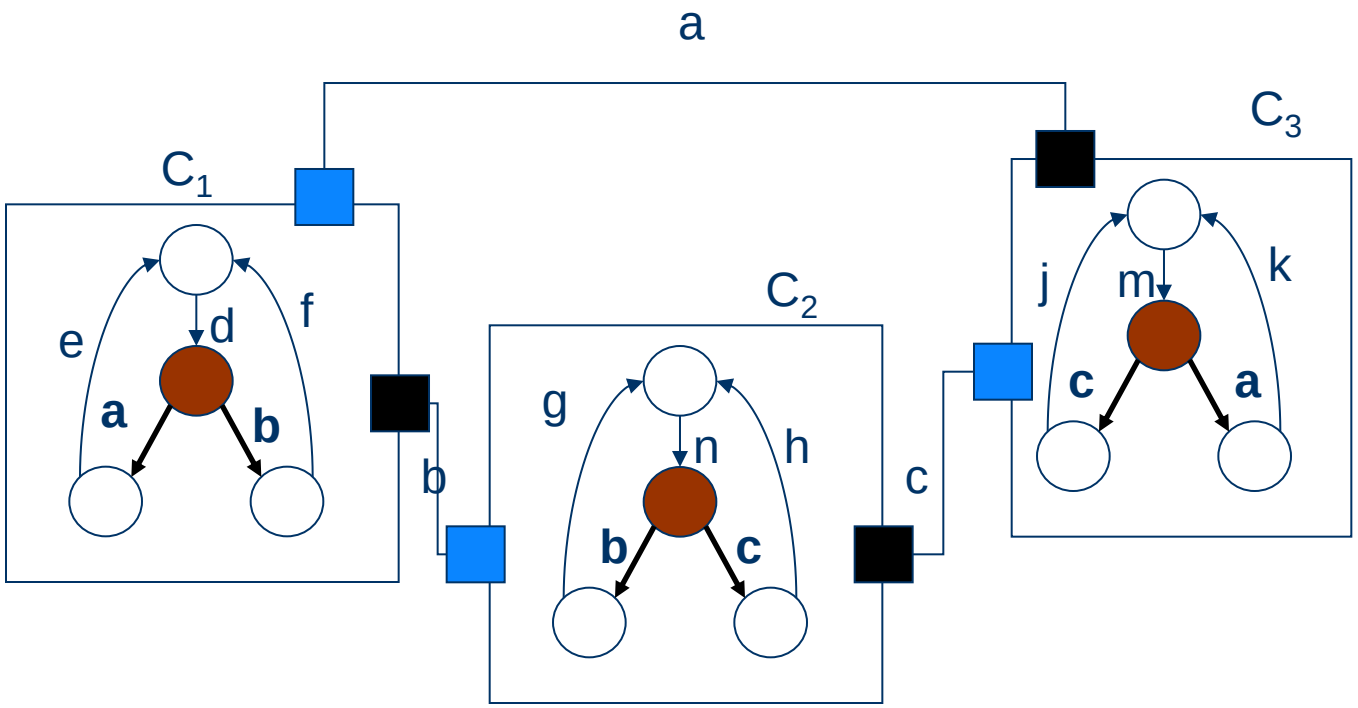


Assume a mechanism where a component can sense its local view

How to implement scheduling?

C_1 may pick up *a*, while C_2 may pick up *b*, but C_3 will pick up c.

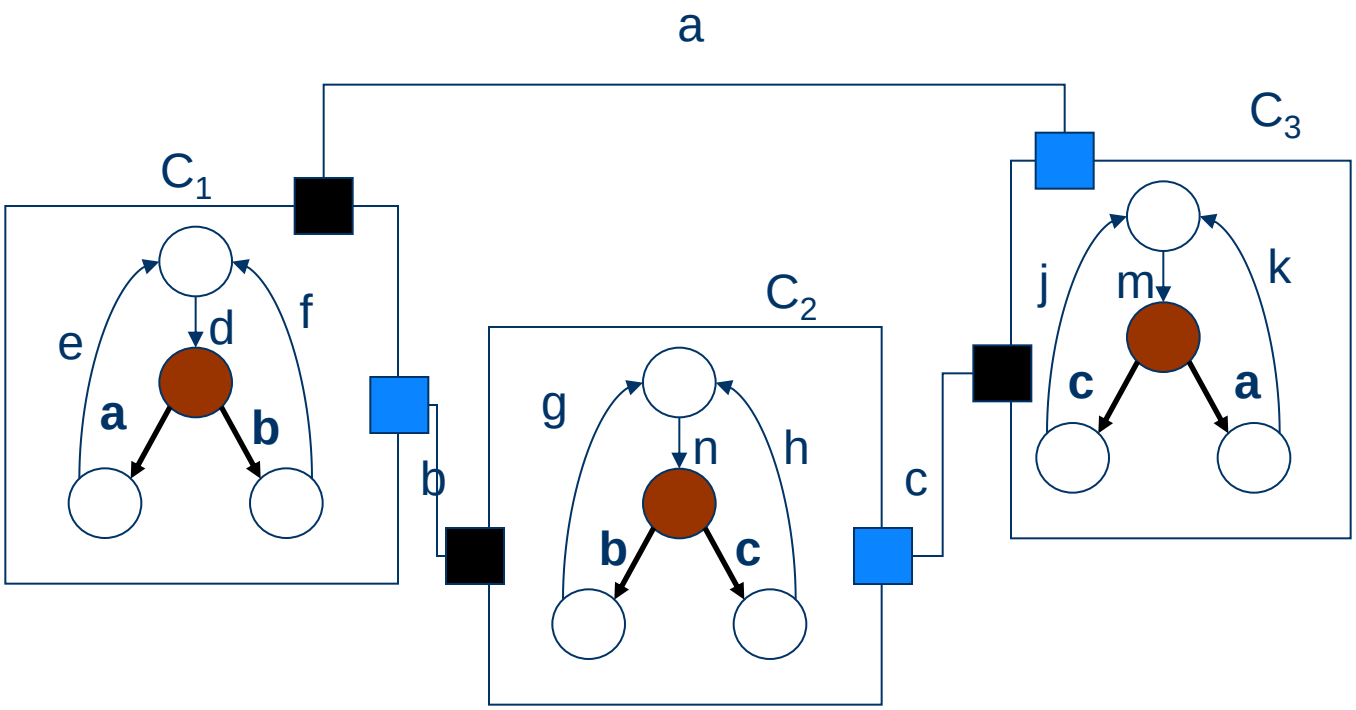
Not working..



How to implement scheduling?

So C_1 picks up *b*, C_2 picks up *c* and C_3 picks up *a*.

Not working again…



We need some mechanism for decisions

- Michael Rabin showed that a symmetric non-probabilistic algorithm cannot be given.
- Synchronizers like α–core, can be used.
- This is *interaction-centric* view.
- Consider the case that we want to guarantee selections by components (e.g., when we need to make some probabilistic choices).
- This is a *component-centric* scheduling.

Some retrospect

- Implementing concurrent systems from a simple model of component automata is not as simple as thought.
- Similar problems exist when implementing multiple choice synchronous communication as in CSP or ADA
  R::[P?x [] Q?y] || P::R!3 || Q::R!z

Interaction based scheduling using α-core

C_1 sends OFFER to a-interaction

C_2 sends OFFER to a-interaction

a-interaction sends LOCK to C_1

a-interaction sends LOCK to C_2

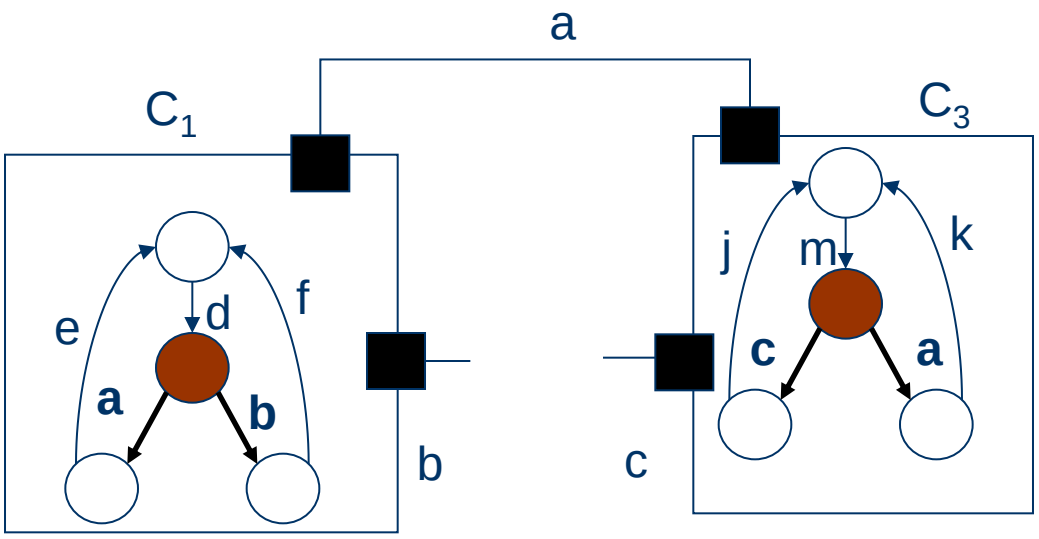
C_1 sends OK to a-interaction

C_2 sends OK to a-interaction

a-interaction sends START to C_1

a-interaction sends START to C_2

More messages for cancelling communication or finishing it.

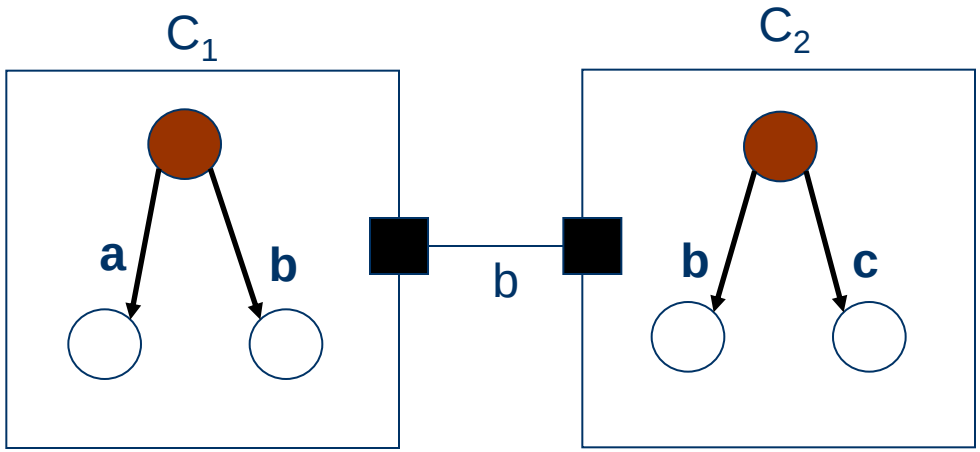


Problems with the interaction-based scheduling

- Expensive: requires quite a lot of message passing, and a process for each coordination.
- Does not support probabilistic decisions.
- An error is identified in the α-core algoritm and was fixed automatically using a genetic programming tool based on model checking [Katz+Peled].

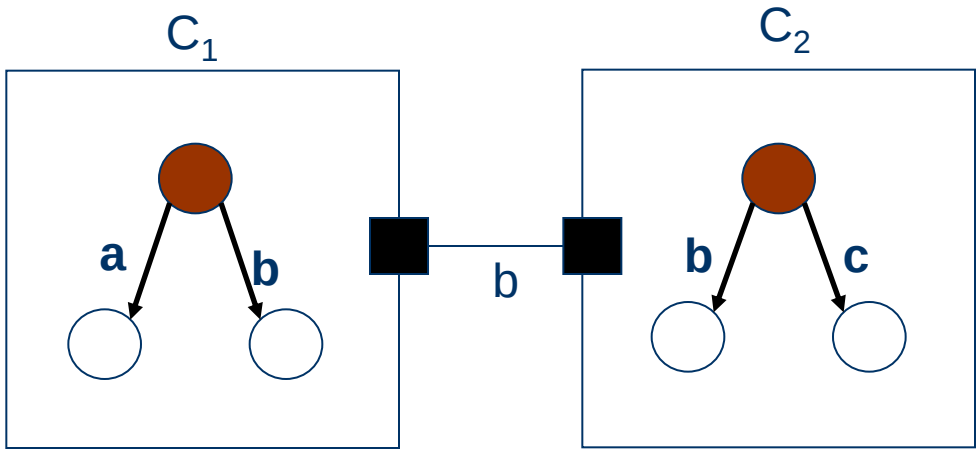
Consider the following situation

- *a* – I will write alone a paper to a conference.
- *b* – We will write together a paper to a conference.
- *c* – You will write alone a paper to a conference.
- Neither you or me have time to write two papers (but we can write one each, separately).



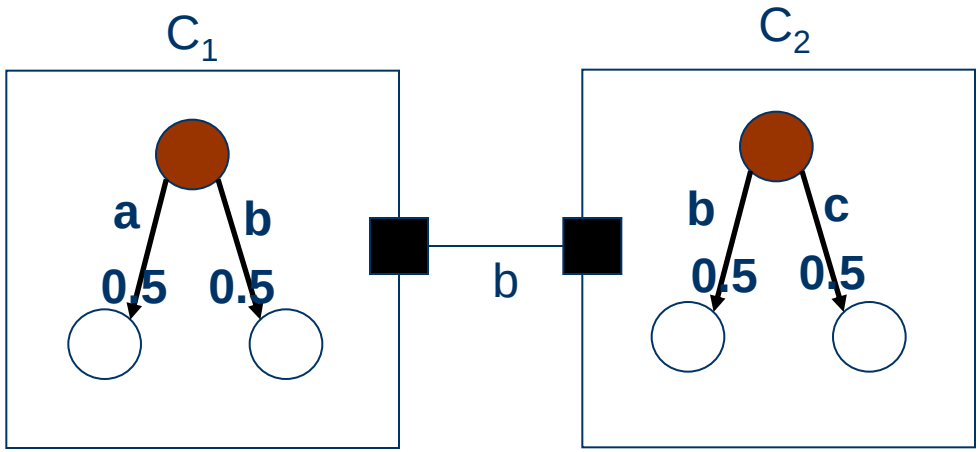
Difficult situation: what to do?

- We can decide separately, but need to decide consistently: cannot be that C_1 decides on working separately and C_2 decides to collaborate.
- To decide on collaboration, we needs to coordinate.

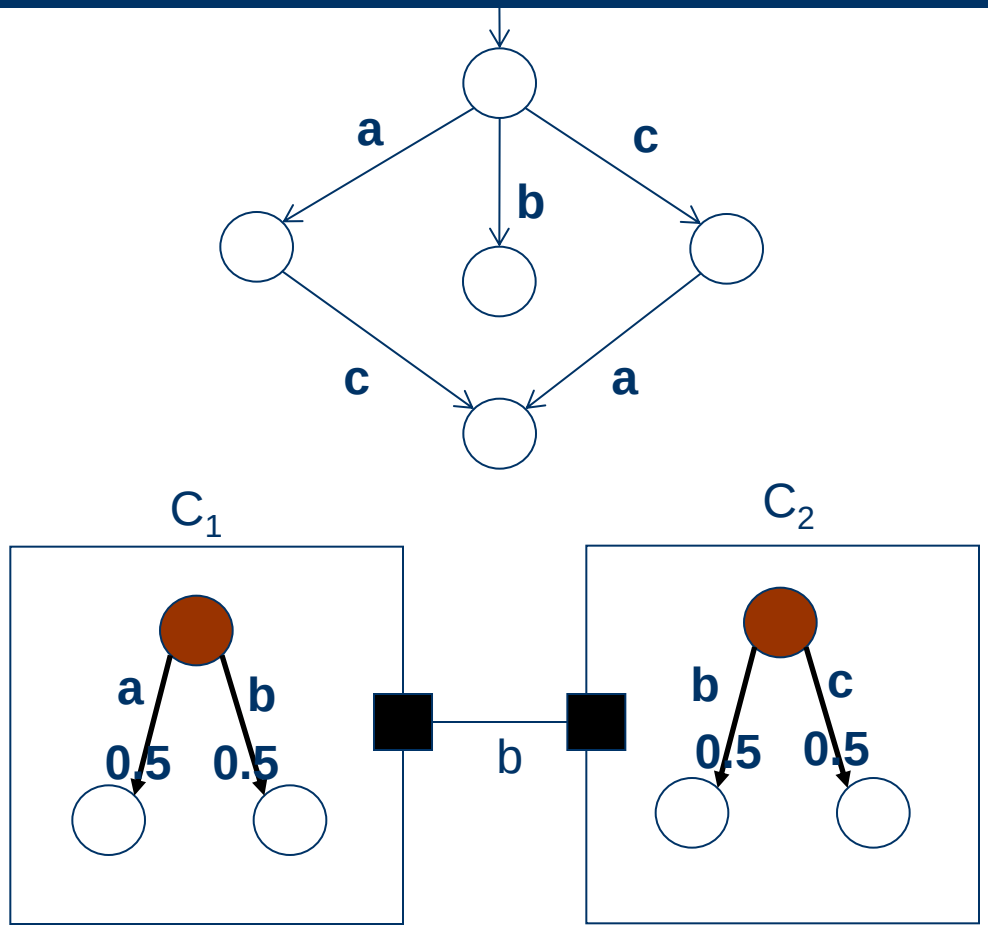


Difficult situation: what to do?

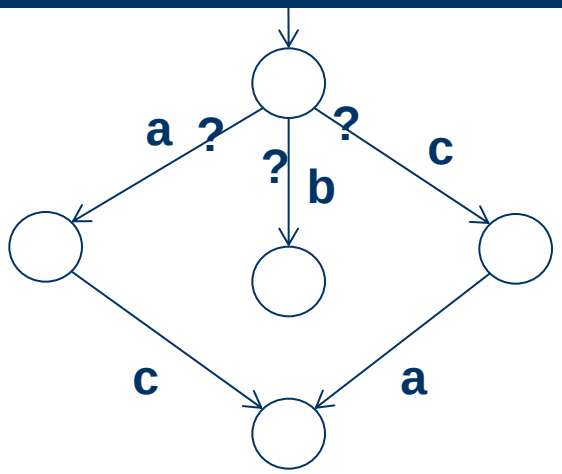
- Perhaps we need to throw a coin. Say we both have a fair coin.
- So, the chance of collaboration is 50-50?
- One of us throws the coin faster…

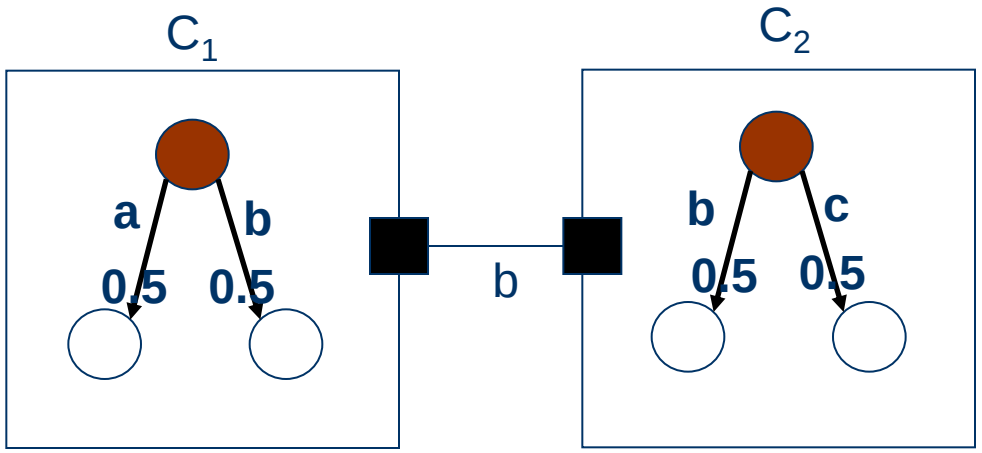


How to model this w.r.t. the state space?

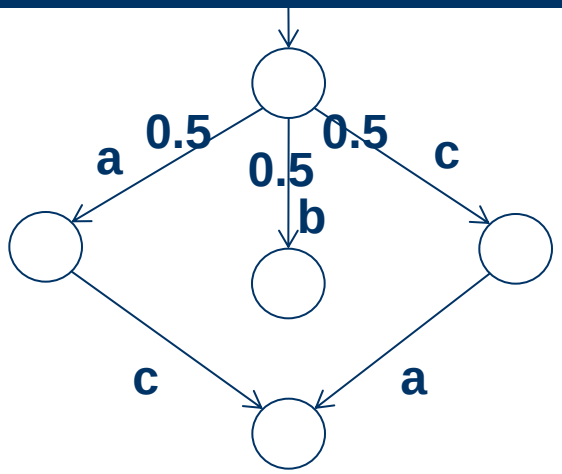


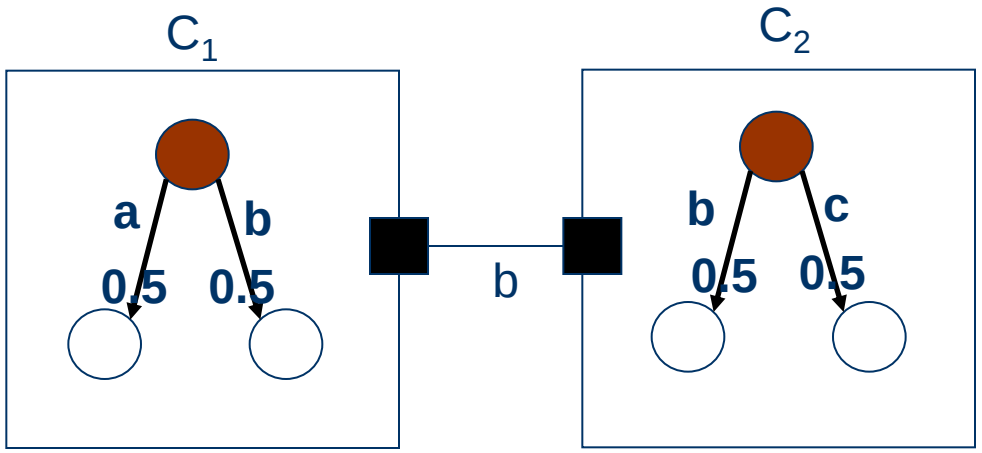
Markov Process?
A graph + a distribution function from each state to the set of states



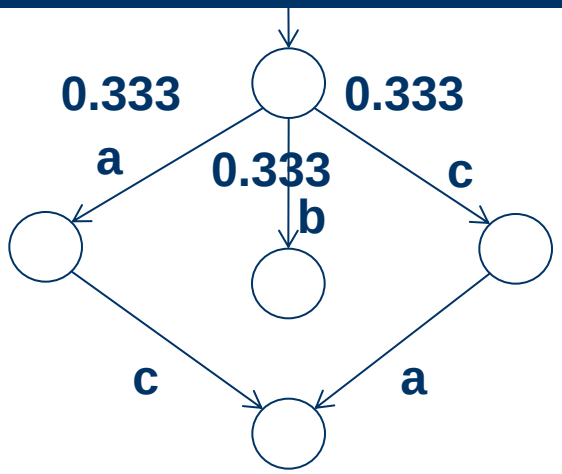


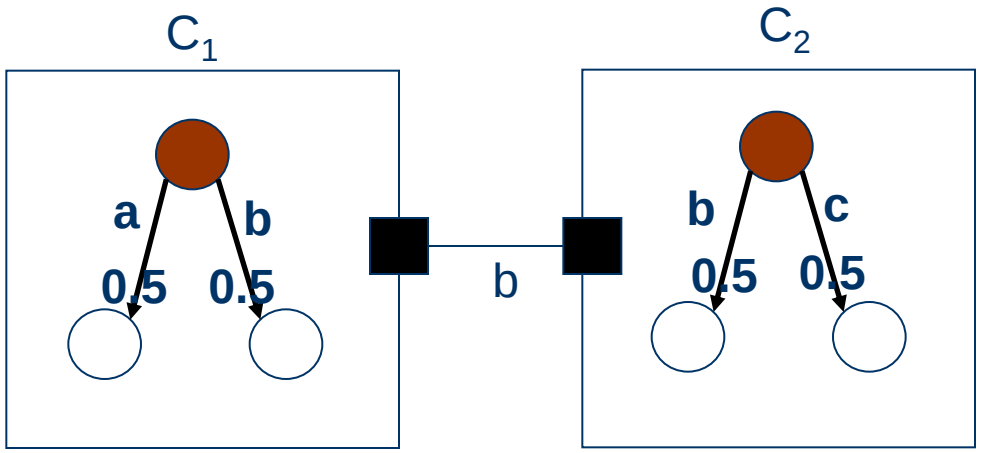
Certainly not… distribution must sums up to 1 !!



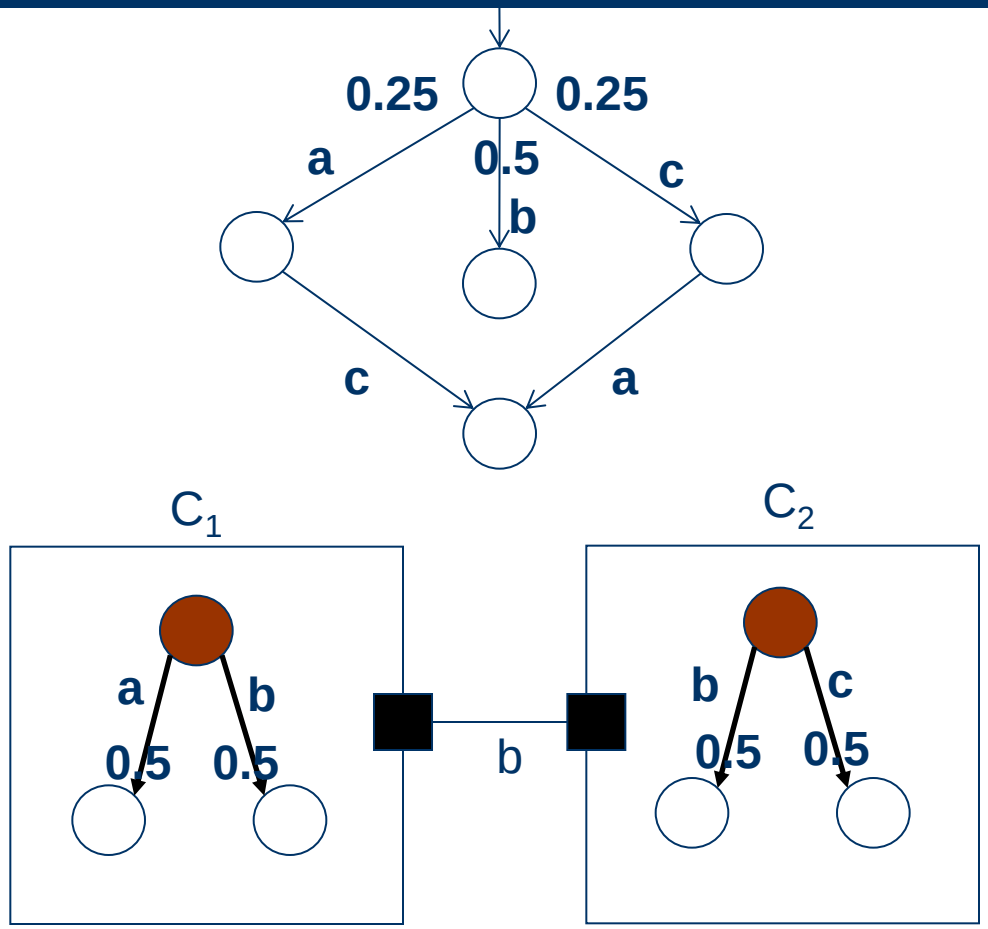


No, we use fair *coins*…

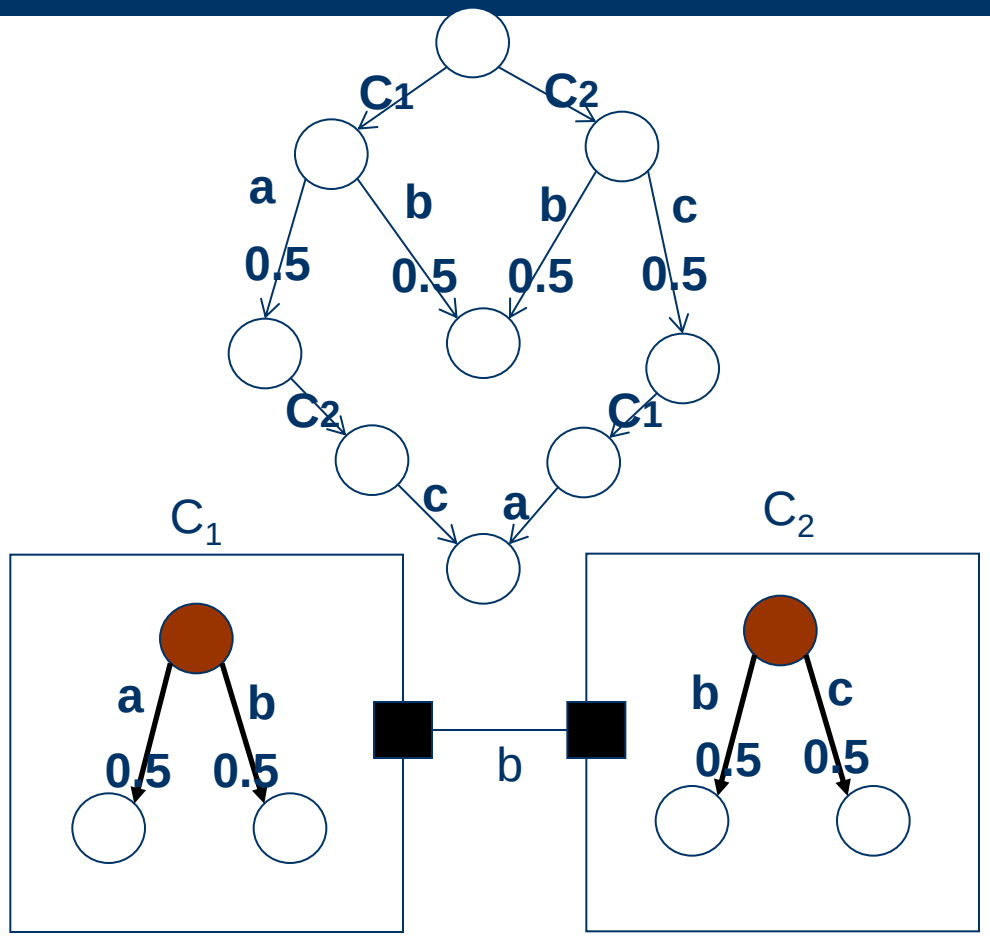




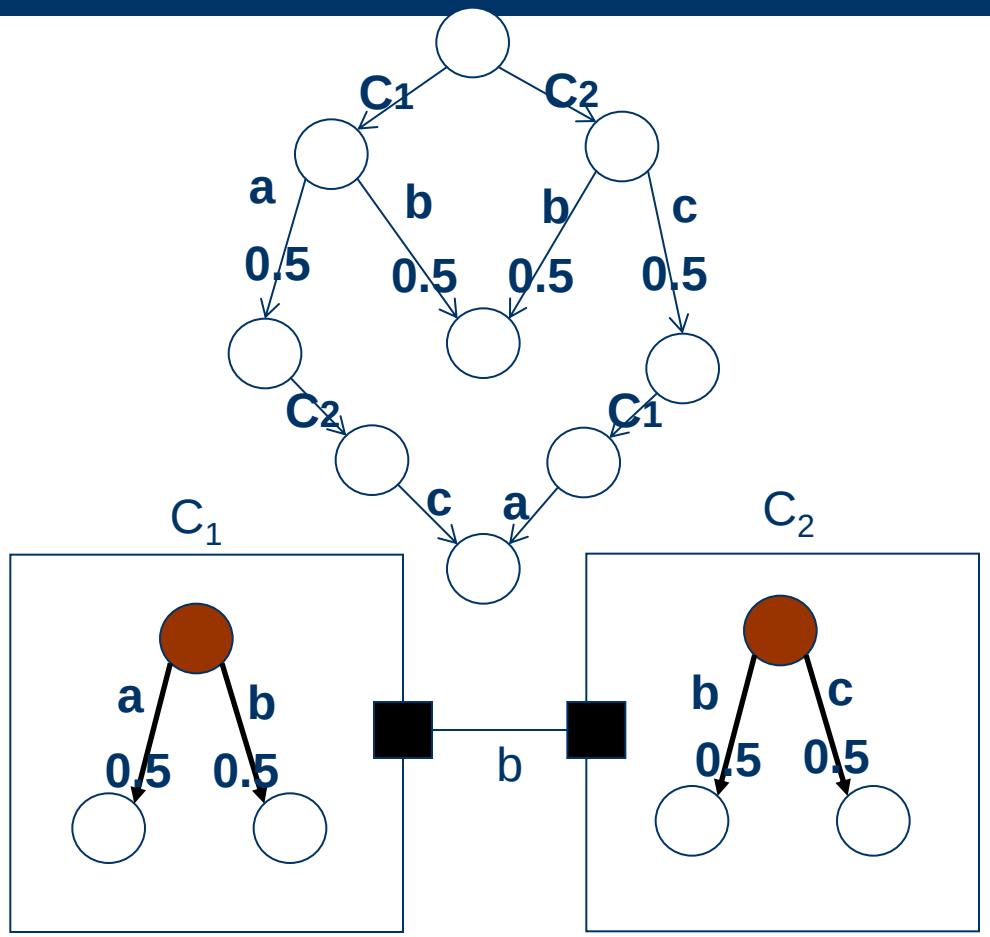
Only if the chance of C1 and C2 to throw the coin first is 0.5 each!



A more realistic model: depends on which component throws the coin first.

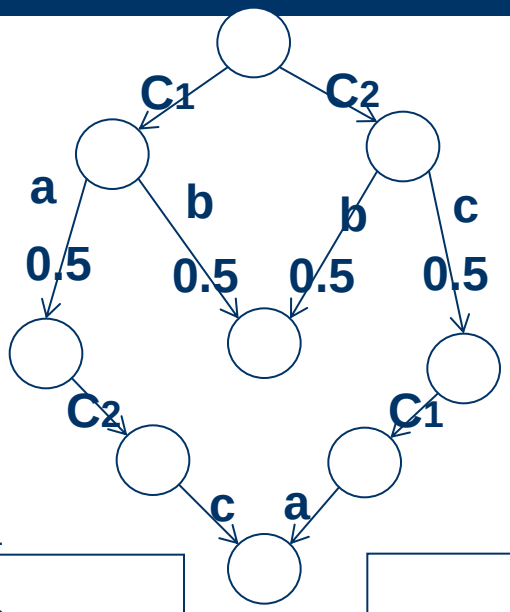


What is the chance of *a* to occur? *b* to occur? *c* to occur?

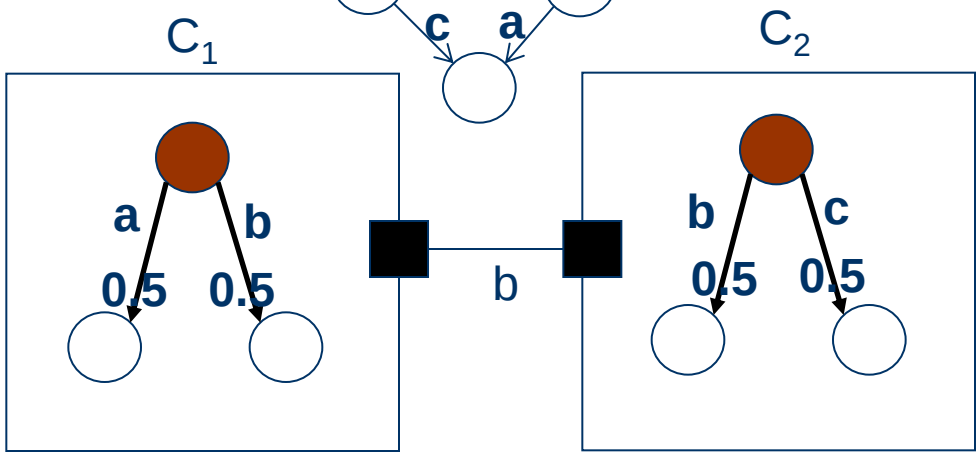


What is the chance of *a* to occur? *b* to occur? *c* to occur?

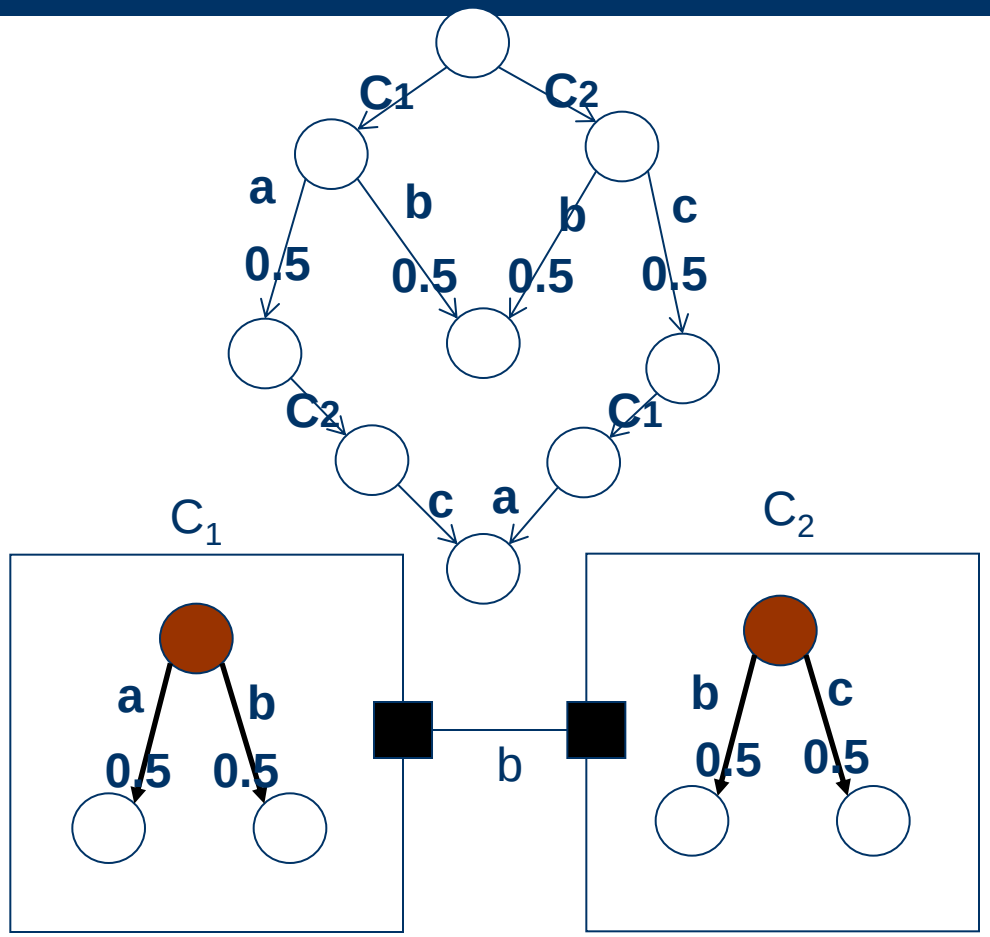
It really depends on the scheduling of throwing the coin.



If we do not have the probabilities for that, we can only provide lower and upper bound.

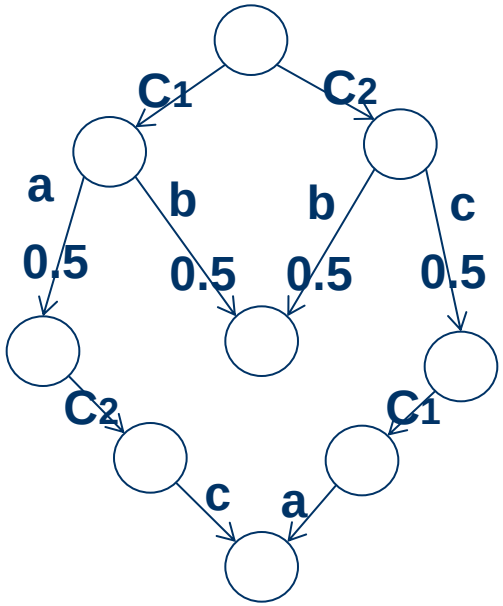


Chance of *a* (of *c*) to occur immediately between 0 and 0.5.
Chance *b* to occur 0.5.



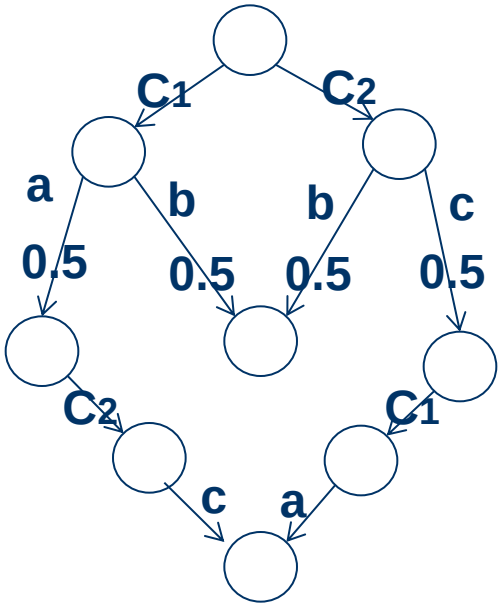
Markov Decision Process

- Set of states *S*.
- [Initial state s_0.]
- Actions *A*.
- For each state *s* in *S*, *A(s)* is a distribution function on *S*.
  I.e., $0 \leq A(s)[s'] \leq 1$ and
  $\Sigma_{s \text{ in } S} A(s)[s'] = 1$.

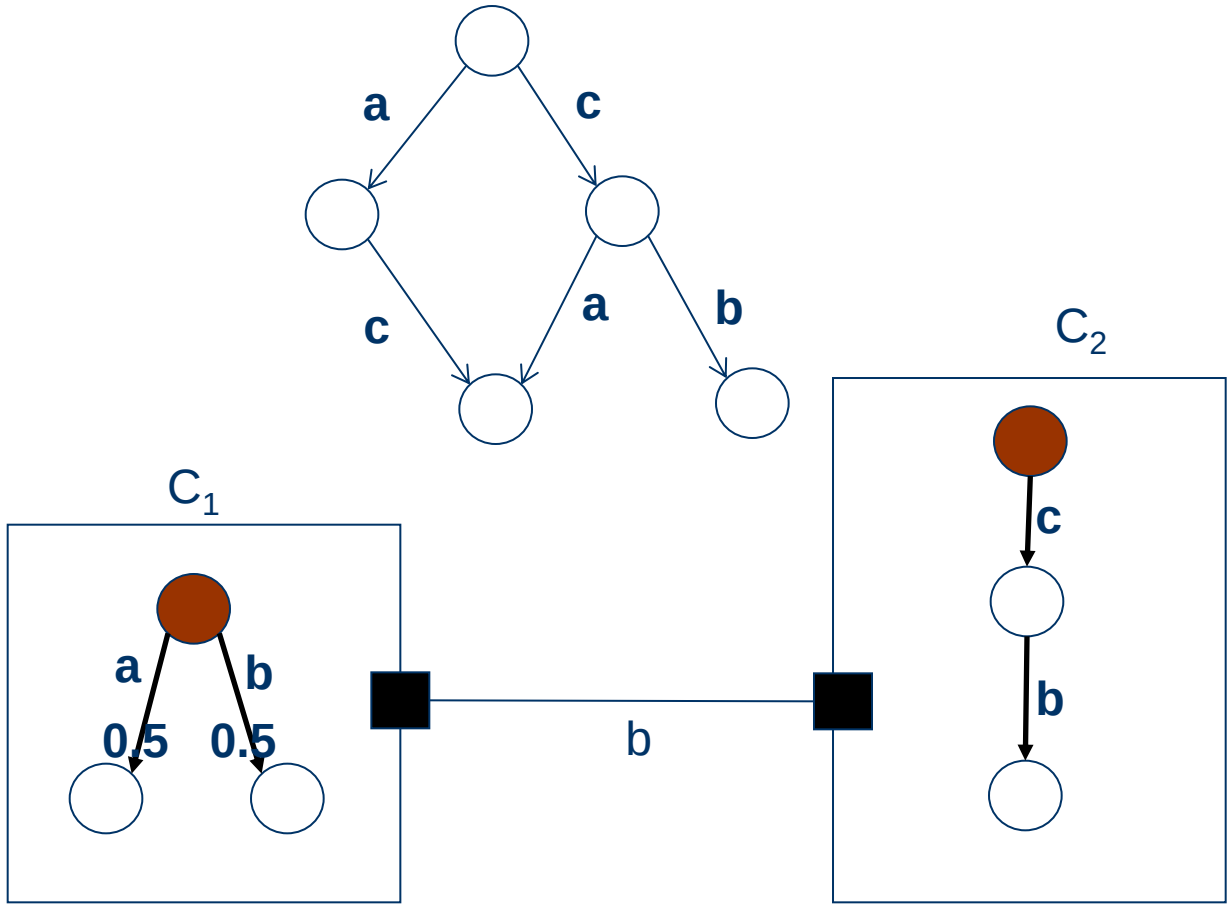


Markov Decision Process Modeling component based systems

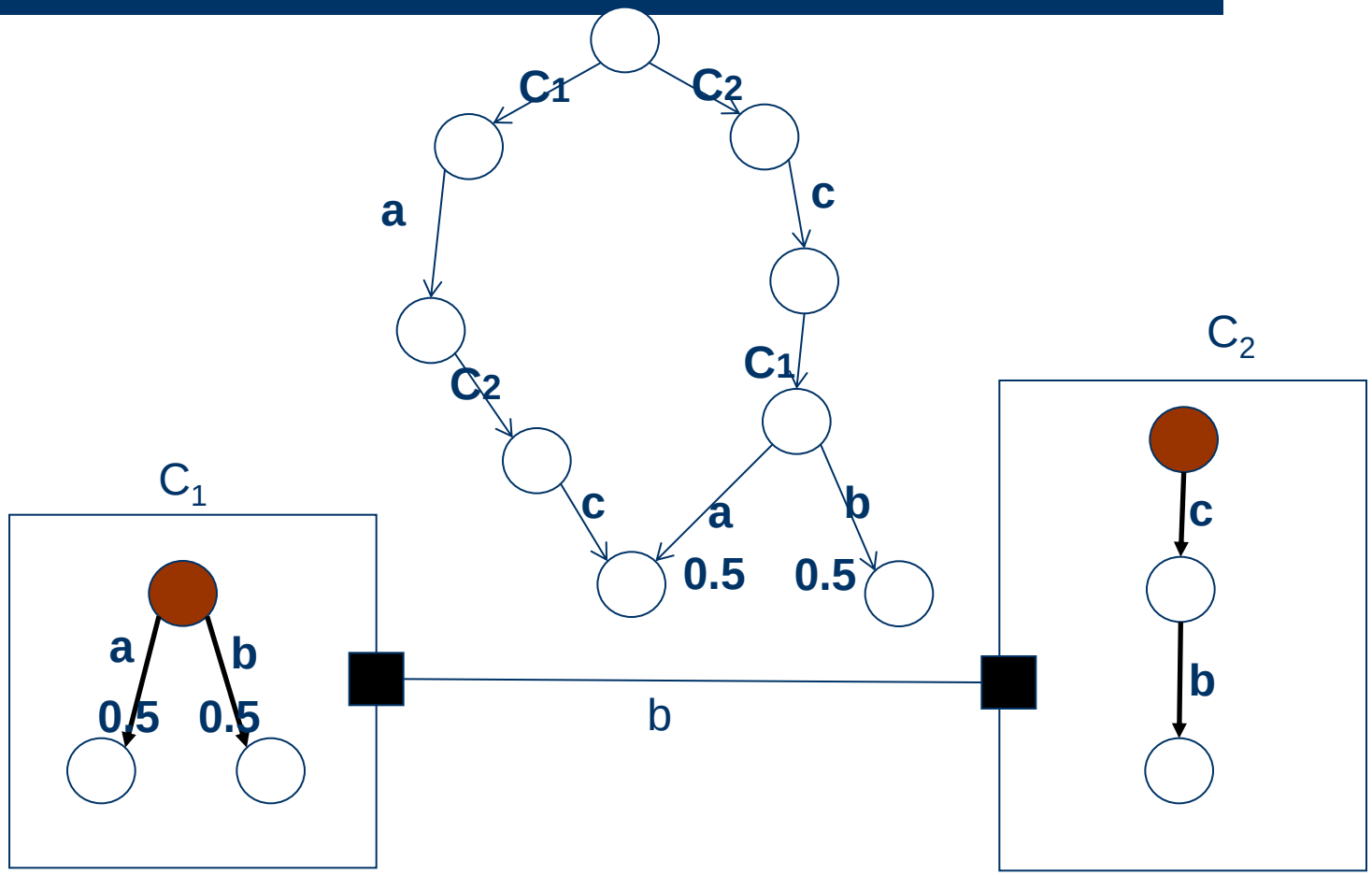
- Set of states *S*.
- Actions *A*: in our case, actions are components making a probabilitstic decision.
- The probability of an action made by a component only depends on its local view: For each state *s, s'* in *S* with the same local view for component *A*, *A(s)=A(s').*



A different scenario: The execution of *c* will allow an alternative choice for a.



Reflecting the choices

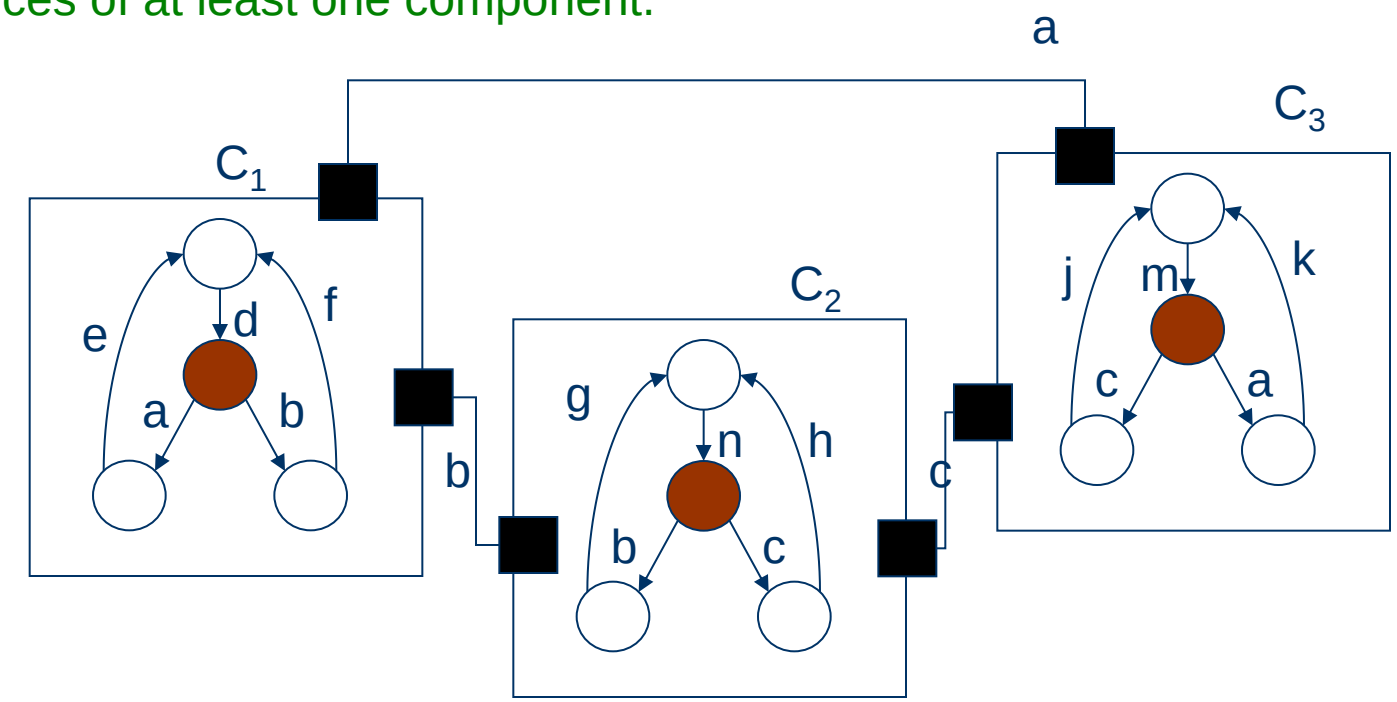


How to implement a component based system?

How to allow at least one component the selection?

Avoid a centralized (sequential) solution.

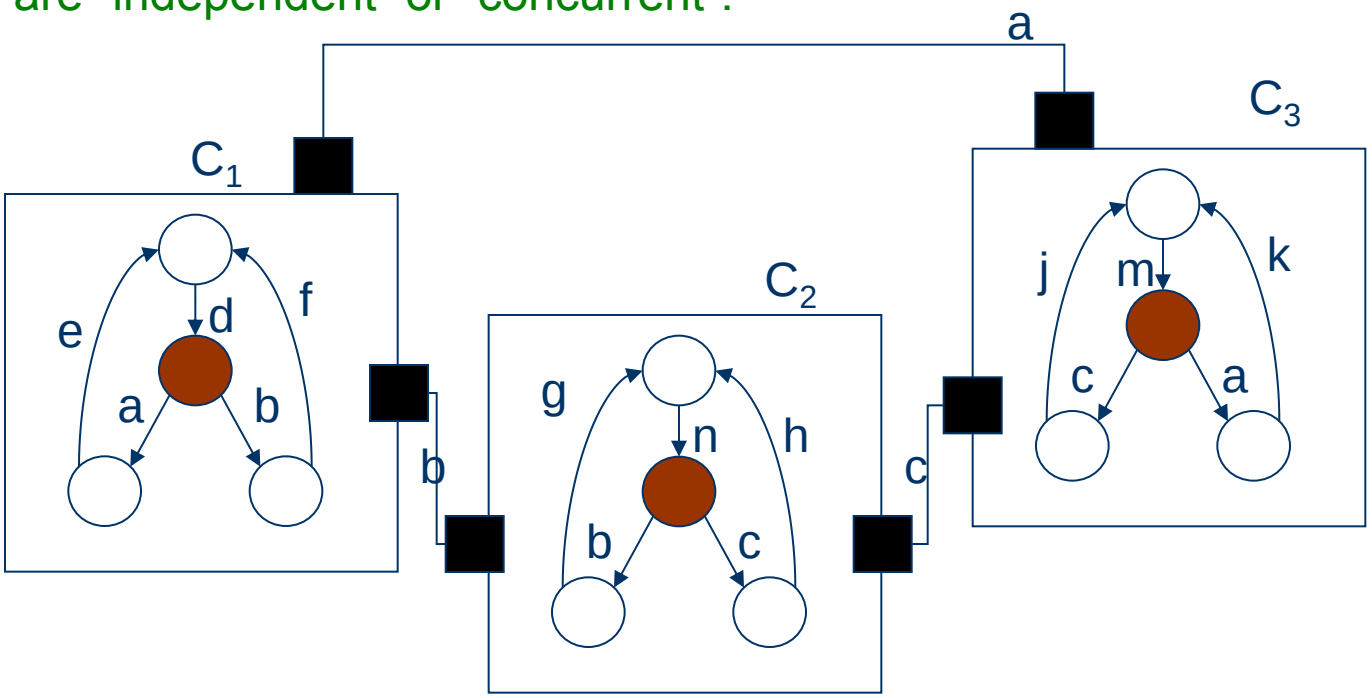
Lock the choices of at least one component.



The notion of a conflict

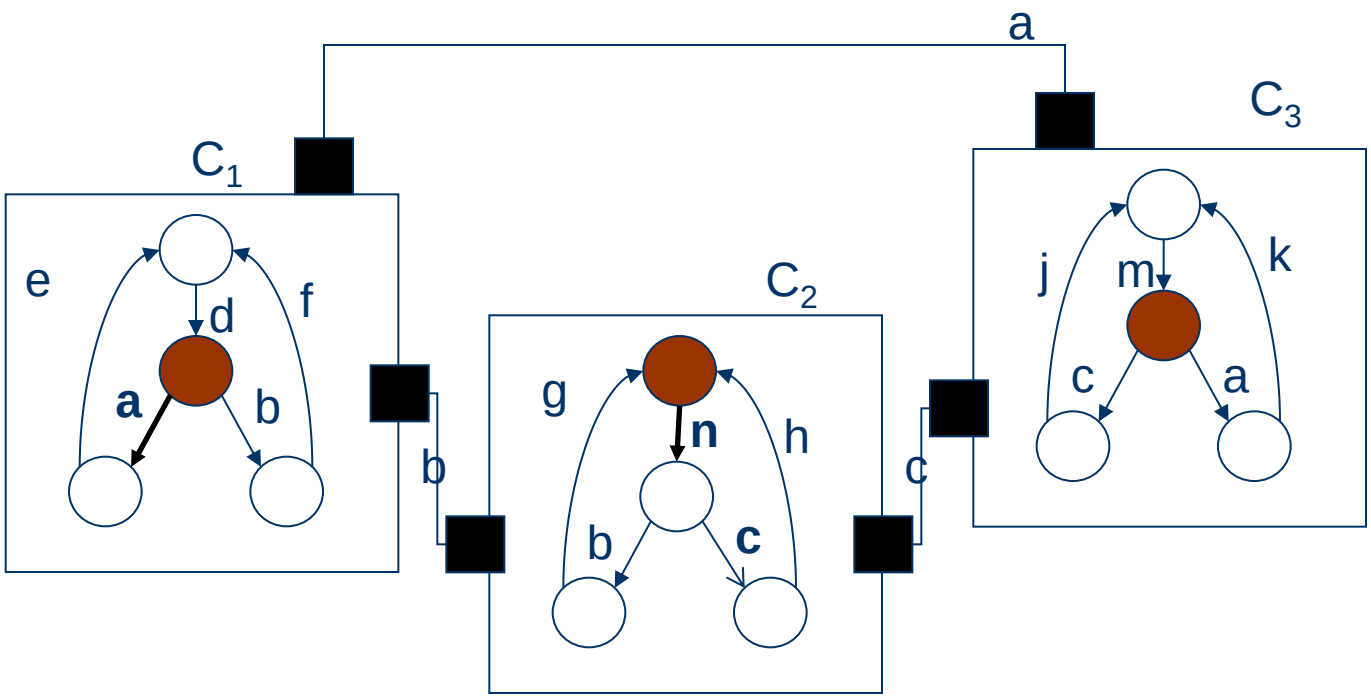
Transitions of a component are “dependent”, when they share the same component, e.g., *a* and *b*, or *d* and *b* in C_1. We say that *a* and *b* are *in conflict* and that *d* and *b* are *sequential*.

Otherwise they are “independent” or “concurrent”.



The notion of a “confusion”

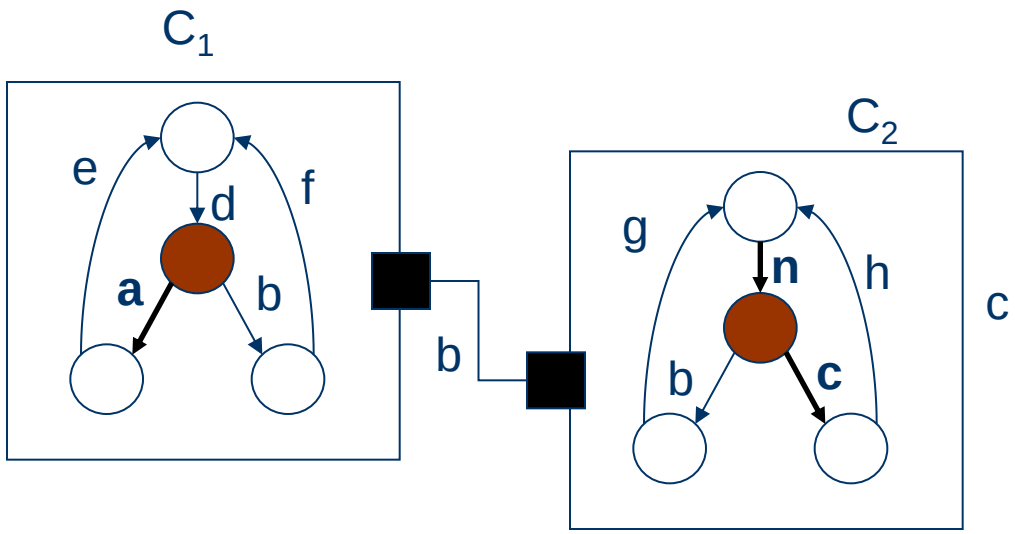
Is a pair of transitions of different components (*a*, *c*) such that executing *c* will either add or remove a conflicting choice to *a*.



The notion of a “confusion”

Transitions are in “confusion” if they are independent but the execution of one can change the alternative choices for the other.

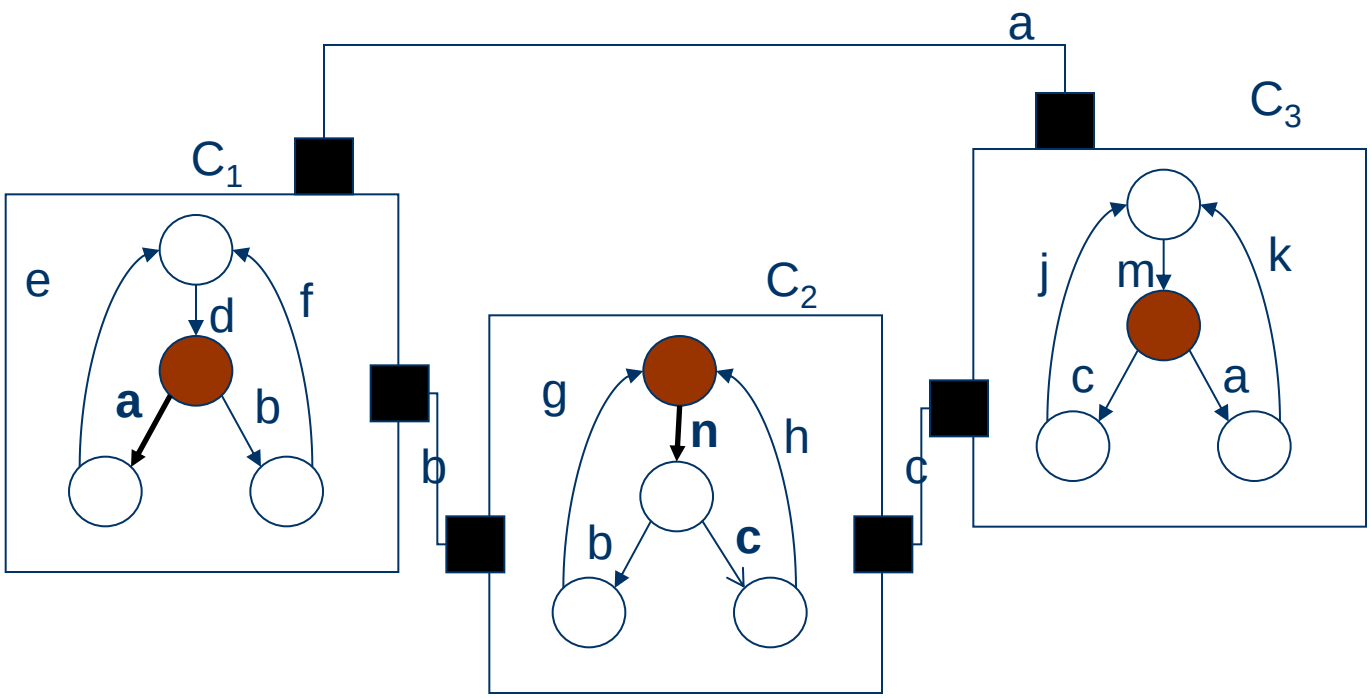
Symmetric confusion: *(a,c)* Transitions are independent but the execution of one can decrease the alternative choices for the other. Consider *n* and *c*.



The notion of a “confusion”

Transitions are in “confusion” if they are independent but the execution of one can change the alternative choices for the other.

Asymmetric confusion: *(n, a)*
Transitions are independent but the execution of one can increase the alternative choices for the other. Executing *n* enables *b* as additional alternative to *a*.



A “micro scheduler” using semaphores

- First, we need to freeze the local view of components. This includes the local state and enabled transitions.
- Then a component can make a decision based on the distribution that depends on its local view.
- What can make a change to the local view?
  - another component making a different choice about an enabled joined transition.
  - due to a symmetric confusion, some joint transitions disabled.
  - due to an asymmetric confusion, some joint transitions enabled.

A “micro scheduler” using semaphores

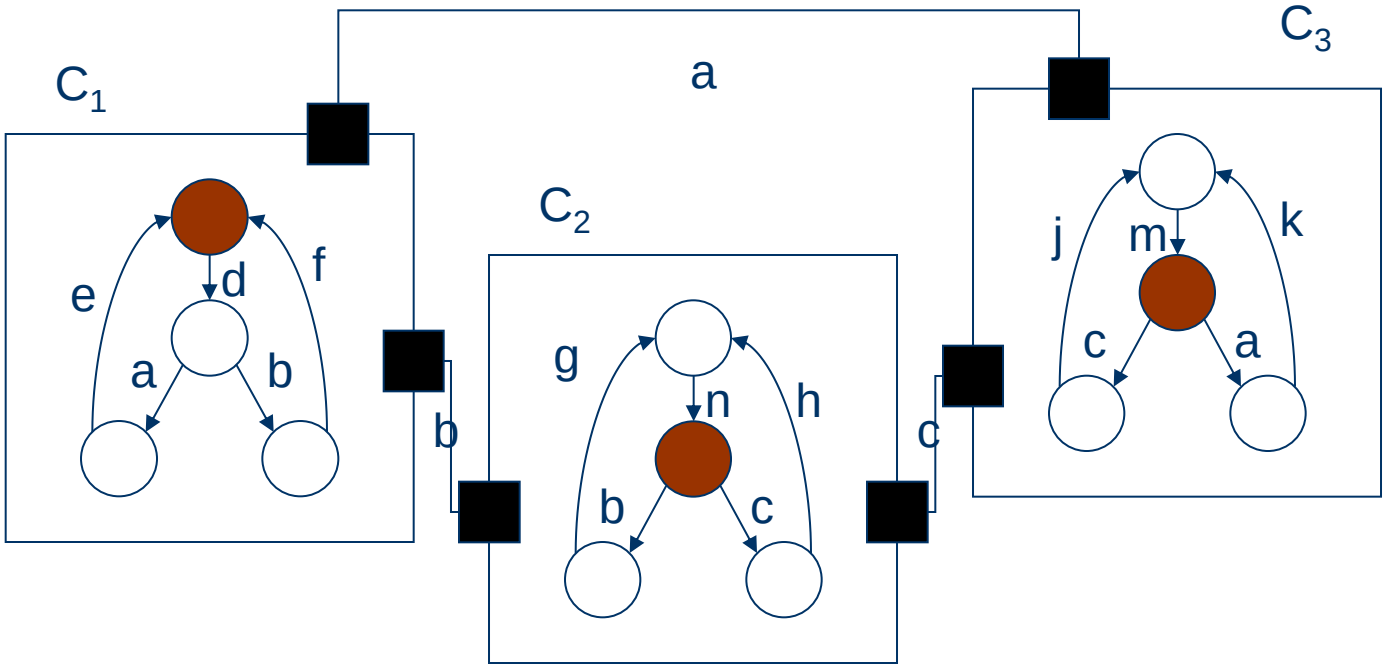
- Provide semaphores *per each component*.
- Provide semaphores per each confusion.
- In order to make a decision about some choice, a component needs to catch the relevant semaphores.
- Check local view.
- Catch the semaphores related to components and confusions invovled in view.
- Check that the view has not changed.
- To prevent deadlocks, number the semaphores and catch them in *ascending* order, release in descending order [Dijkstra].

A “micro scheduler” using semaphores

- Optimization: we do no need semaphores for symmetric confusions: case would be eliminated by capturing semaphore for components.
- Could alternatively catch semaphores for the components that share the locally but not globally enabled transitions .
  Then do not need semaphores for asymmetric confusion.
  But this will reduce concurrency.

For C2 to move…

c is enabeld. Catches semaphores on C2, on C3 and on the confusion (*d*, *c*). *Why?*



Implementation

- Components serve in a *master – slave* mechanism. A slave offers updates on locally enabled transitions through shared variables and execute its part in a selected transition.
- Also solves a long-lasting problem on modeling and implementing concurrent systems with probabilistic choices [Katoen+Peled].
- Allows parallelism.
- No deadlocks.
- Previous solutions involved allowing only one transition to fire in the entire system.
- Simple to implement (implemented by Ayoub Nouri) using standard semaphore operations.
- If probabilistic choice is not necessary, we can remove semaphores for asymmetric confusion.

Implementation

- Allows parallelism.
- Transitions are not necessarily small or simple. Behaves as if atomic; linerarizability.
- No deadlocks.
- Previous solutions involved allowing only one transition to fire in the entire system [Lynch et al].
- Simple to implement (implemented with Verimag) using standard semaphore operations.
- If probabilistic choice is not necessary, we can remove semaphores for asymmetric confusion.

Shared transitions

- Each component sets up a shared flag to indicate when a shared transition *a* is locally enabled (and resets when it becomes disabled).
- To check its view, a component needs to check which other components that share its locally enabled transitions have set up their shared flags.
- Can we eliminate these expensive checks?

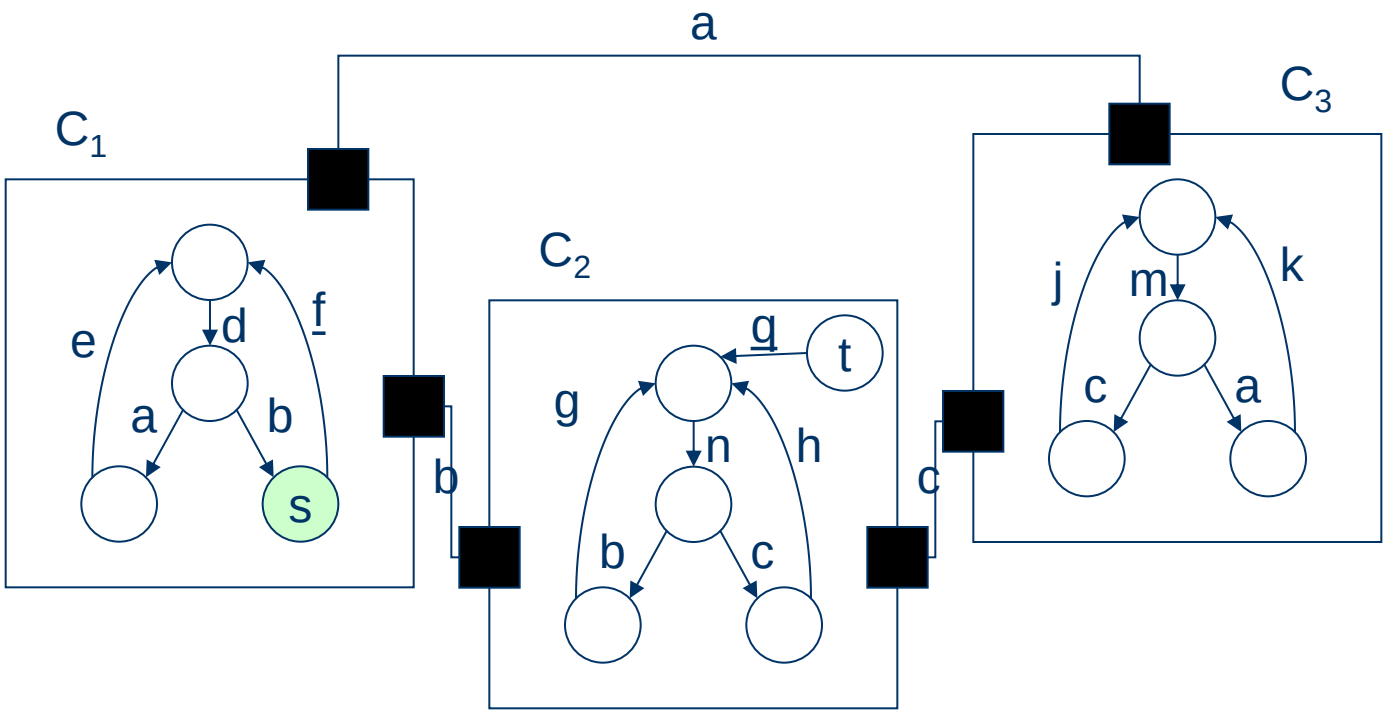
How to eliminate repeated checks for enabledness of shared transitions?

- Can use a synchronizer like α-core (not cheap…).
- Can use « knowledge » to eliminate some of the checks;

What is knowledge?

- A component *X* "knows" a property *p* about the system if every (global) state of the system that includes the current local state of *X*, *p* holds.
- Joint knowledge of components: when combining their local states. More components know together more.
- Calculate the reachable states that have the same current local state of component *X.* Then check whether these states satisfy *p*.
- Can be done using model checking techniques, e.g., based on BDDs.

When the left component is in local state *s*, it knows that the middle component cannot be in state *t*.



How to use precalculated knowledge?

- Make a table that tells each component in which of their local states the enabledness in other components is known.
- Knowledge is not complete (and often sparse).

History preserving knowledge

- Can use memory to remember information about the history of the system.
- Then we may « know » much more about what happens in other components, as different histories diffrentiate between cases.
- This is expensive. Essentially we need to remember the set of global states where the system can be, given its local history (a subset construction of the global state space) in each component.

Retrospect

- If probabilistic choice is not necessary, we can remove semaphores for asymmetric confusion.
- In each one of the scheduling mechanisms, there are some internal choice that happen. After these choices, the set of possibilities available for the system is limited;
  if the system is coupled with some observer, it would be able to sense the change from the theoretical branching structure.

Conclusions

- **Components based systems can be used as at an intermediate design level.**
- **Implementing them involves a scheduling mechanisms that depends on architecture.**
- **For making a probabilistic choices, we need to protect the choices by each component.** Solves a long-lasting problem on modeling and implementing concurrent systems with probabilistic choices [Katoen+Peled].
- **The goal is to provide a complete system that can be compiled directly on hardware. Scheduling is part of the compiled code.**