Entity Behavior Model

Consensus

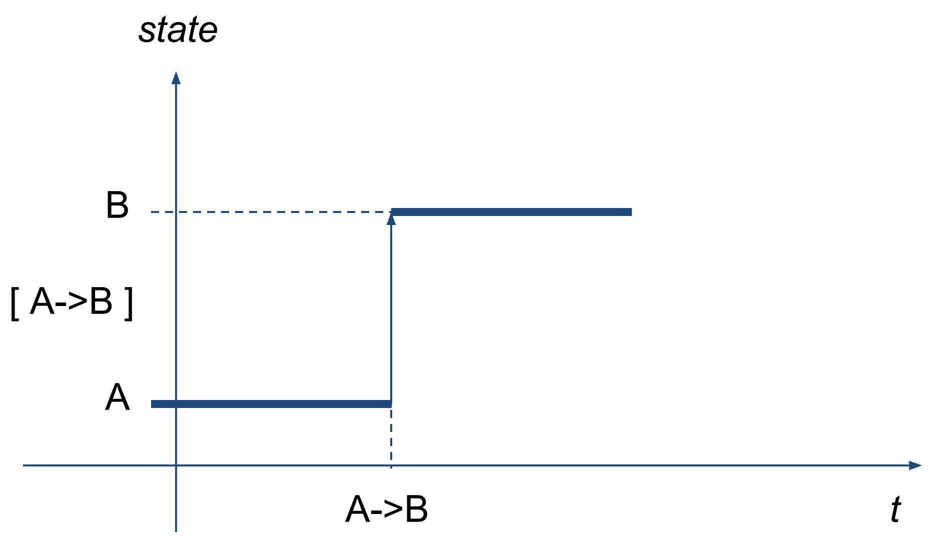
Patrick Bouchaud
2017-07-14

Fundamentals

Terms & Definitions

- Every journey starts with a **story** — and ends up uncovering many
- We call **system** a given frame of observation
- We call **occurrence** an observable within the system
- We call **event** an observable **change** within a system

something that changes is something which goes from one **state** into another



Story

- The behavior of a system is typically specified using such statements as

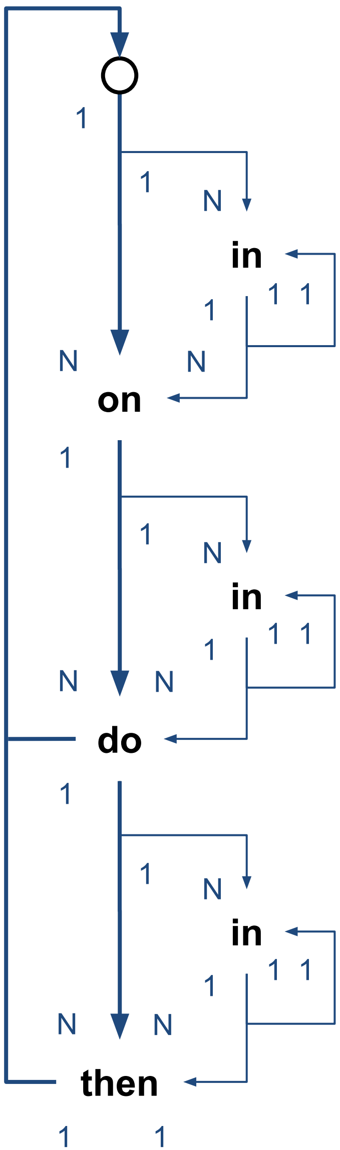
  In (*these* system conditions)
  when (the user does *this*) then (the system does *that*)
  when (the user does *this*) then (the system does *that*)
  . . .
  In (these *other* system conditions)
  ...

- We call **story** the formalization of these statements into the following structure

```
story       = on event behavior [ story ]
              | in condition << story >> [ story ]
behavior    = do action [ sequence ] [ behavior ]
              | in condition << behavior >> [ behavior ]
sequence    = then story [ sequence ]
              | in condition << sequence >> [ sequence ]
```

Note: The word *then* in the statements provided as example indicates *consequentiality*, that is: *that* should start when *this* finishes ; whereas in Consensus *then* indicates *sequentiality*, that is: that [*story*] should *not* start before this one finishes ; as in Consensus all *actions* execute **concurrently** unless specified otherwise - with *then*.

Narrative



The diagram on the left illustrates the logical *structure* of the Consensus *story* associated with a system, or system's **narrative**.

story = **on** *event* behavior [story]
| **in** *condition* **<<** story **>>** [story]
behavior = **do** *action* [sequence] [behavior]
| **in** *condition* **<<** behavior **>>** [behavior]
sequence = **then** story [sequence]
| **in** *condition* **<<** sequence **>>** [sequence]

Occurrence

- *condition*, *event* and *action* are each made of **occurrences**, where

  each occurrence is a single sentence written in the present tense and **affecting** a single object

- We have
  - *condition* is a **set** of *occurrences*
  - *event* is a **set** of *occurrence*
  - *action* is a **set** of *occurrence*
- We can now formulate our narrative all the way down to *occurrences,* using the following declarations in the appropriate places
  - *condition* = *occurrence* [*occurrence*]
  - *event* = *occurrence* [*occurrence*]
  - *action* = *occurrence* [*occurrence*]

co-systems & states

- We call **co-system** the object to which an *occurrence* applies
- We can then further translate each occurrence depending on its context as
  - in case of a *condition* occurrence: co-system is *in* **state**
  - in case of an *event* occurrence: co-system is *entering* or *leaving* **state**
  - in case of an *action* occurrence: co-system is *entering* or *leaving* **state**

occurrence	**mapping**		
context	co-system[1]: state	co-system[1]: ->state	co-system[1]: state->
event		X	X
condition	X		
action		X	(X)

- This mapping allows to enter the next phase of the system specification process (or **narration**) by applying it recursively to each identified co-system.

1. or nothing, in which case it means the System itself

Interface with Programming Languages

- Consensus provides built-in support for any programming language as follows:

  Each *occurrence* can also be mapped to a *programming expression* consisting

  - in case of a *condition*: of a *boolean expression*
  - in case of an *event*: of a *boolean transition*
  - In case of an *action*: of a *programming block*

context	**mapping**	
event	->(*boolean expression*)	(*boolean expression*)->
condition	(*boolean expression*)	
action	{ *programming block* }	

in case of a *programming block* it is the responsibility of the program to publish appropriate *states* and issue appropriate *state change* notifications — via the provided Consensus API — e.g. for synchronization purpose

Methodology

Methodology

1. Relate the story describing the system behaviour
2. Identify all occurrences. Apply the narrative structure e.g.

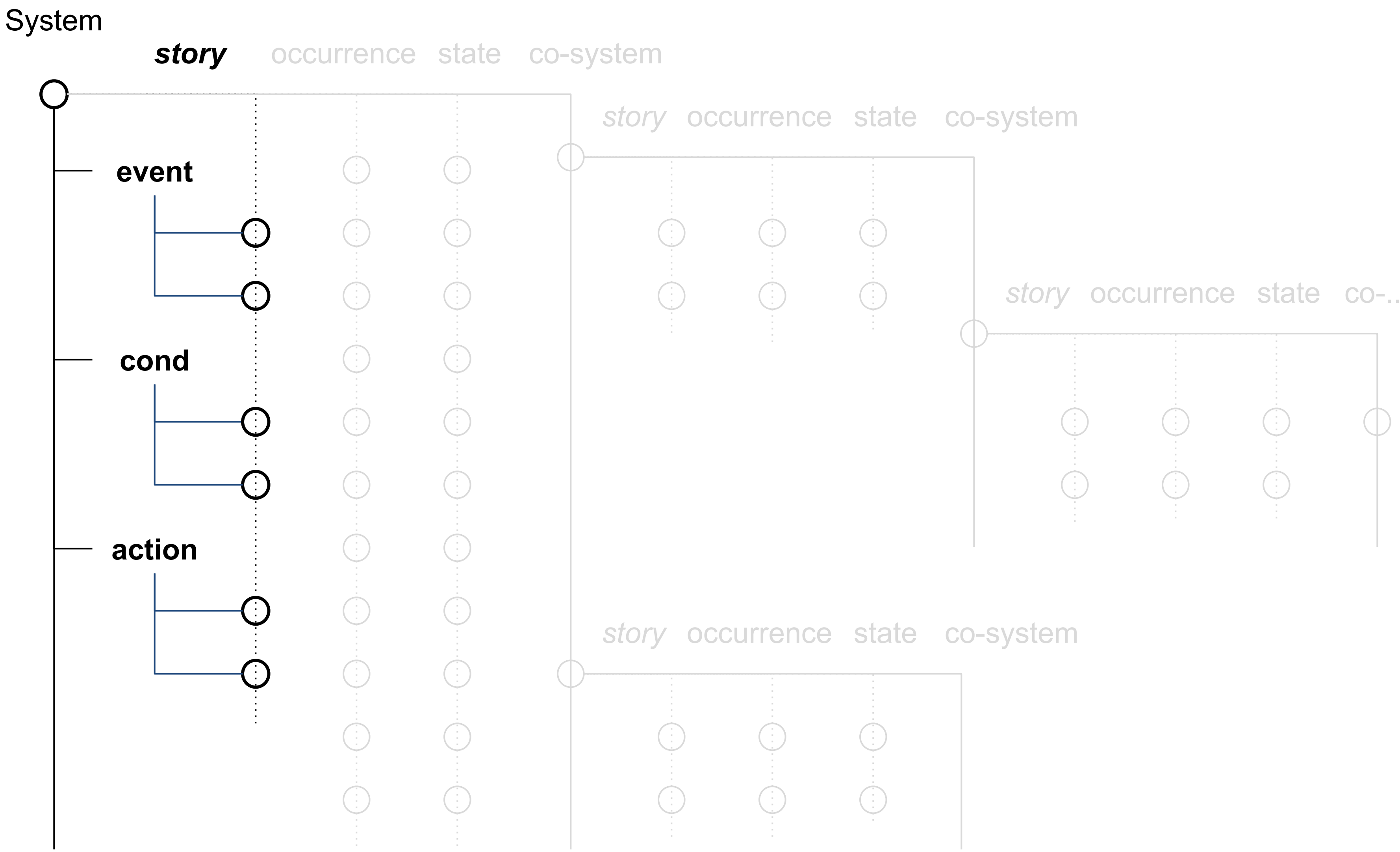
    in *condition*
        on *event* **do** *action* **then on** *event* **do** etc.
        on *event* **do** *etc.*
        etc.

3. Identify co-systems and map each occurrence to corresponding states or state changes

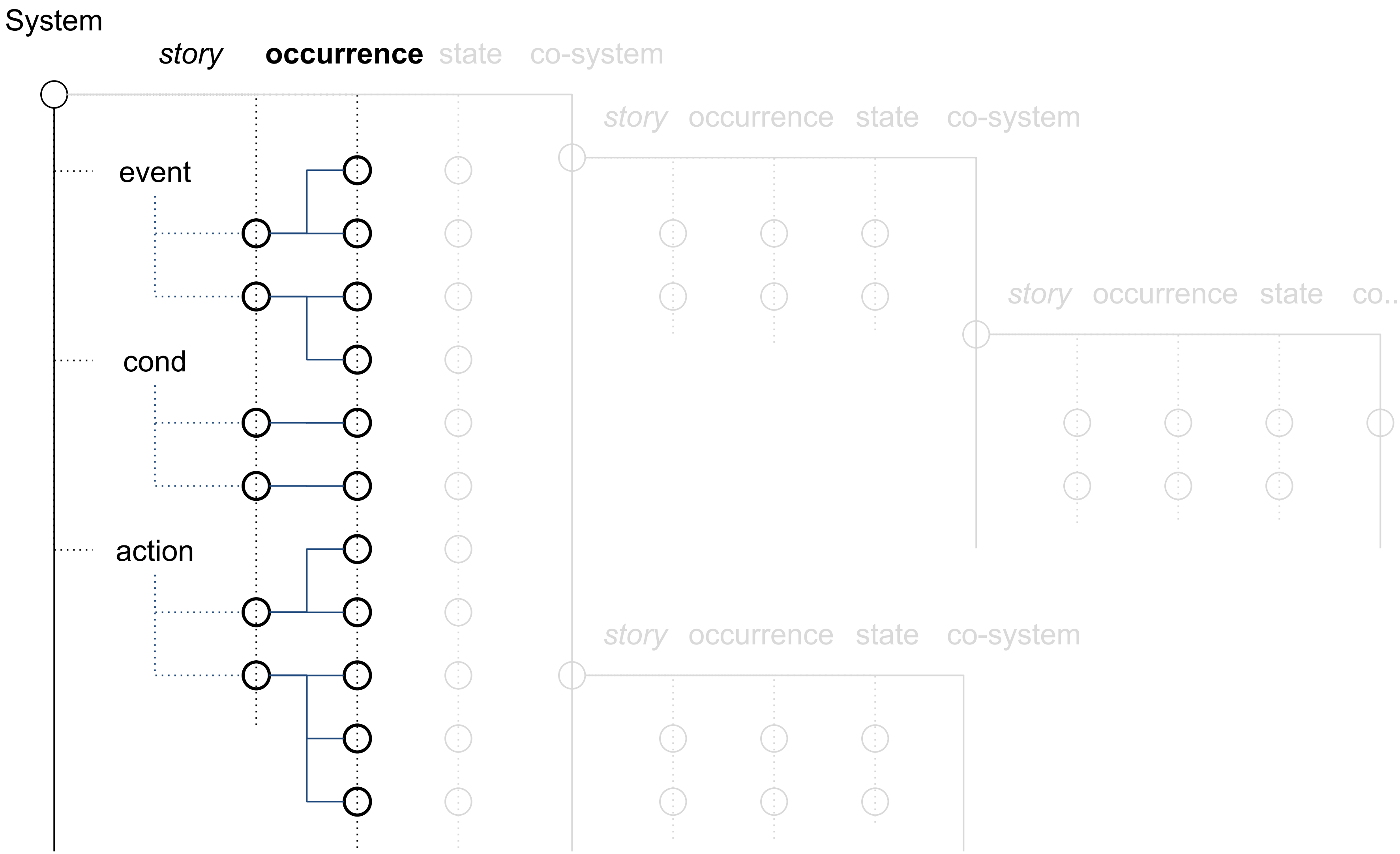
mapping	from	to
condition	(occurrence, …)	(co-system:state | boolean expression, ...)
event	(occurrence, …)	(co-system:->state | co-system:state-> | ->(*boolean expression*) | (*boolean expression*)->, …)
action	(occurrence, …)	(co-system:->state | co-system:state-> | { programming block }, …)

4. Apply recursively to all identified co-systems

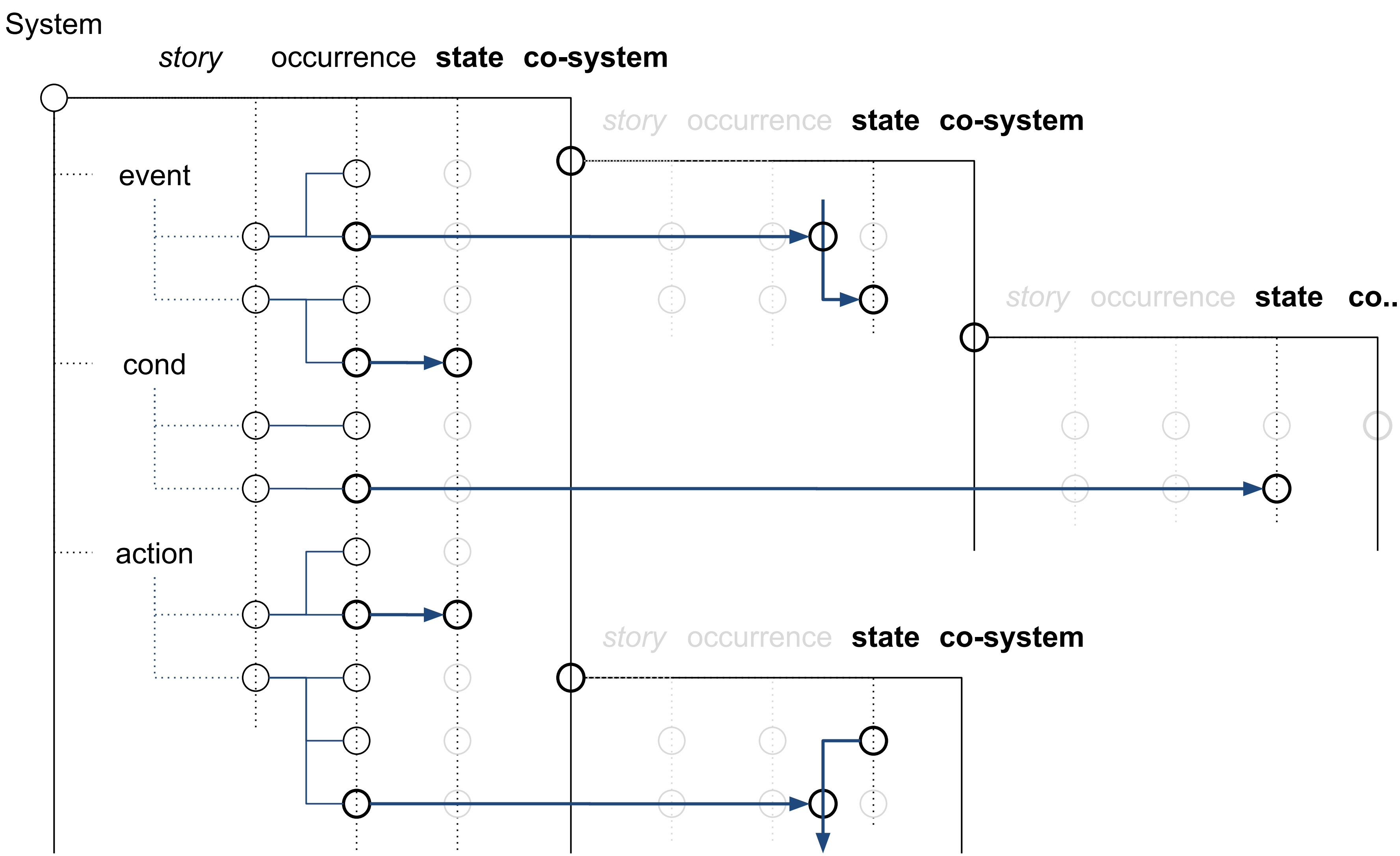
1. Relate the story and break down into conditions, events and actions



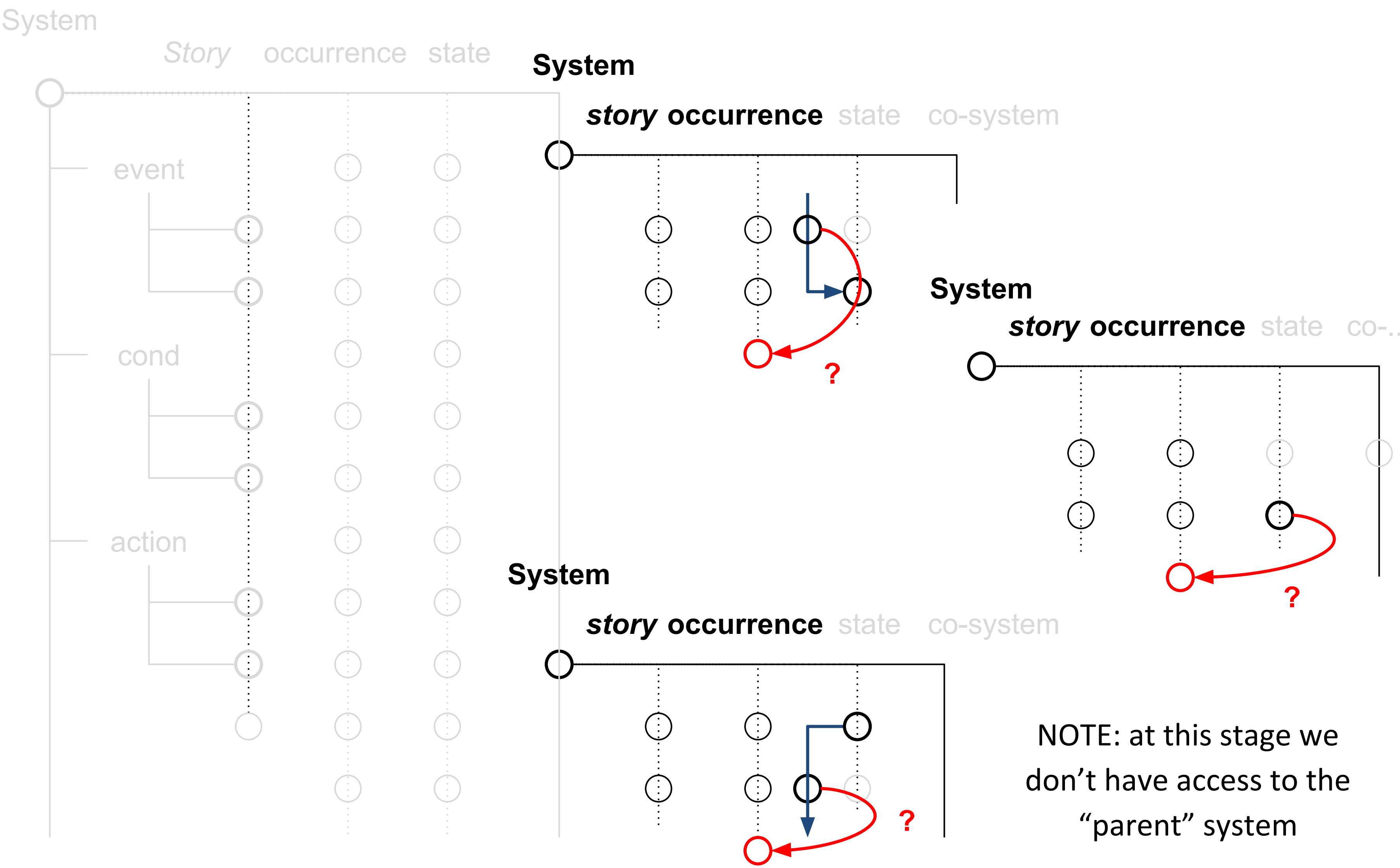
2. Map conditions, events and actions to occurrences



3. Identify co- and co systems and map each occurrence to corresponding states or state changes



4. Relate the stories of the identified co-systems, and map already identified states into occurrences

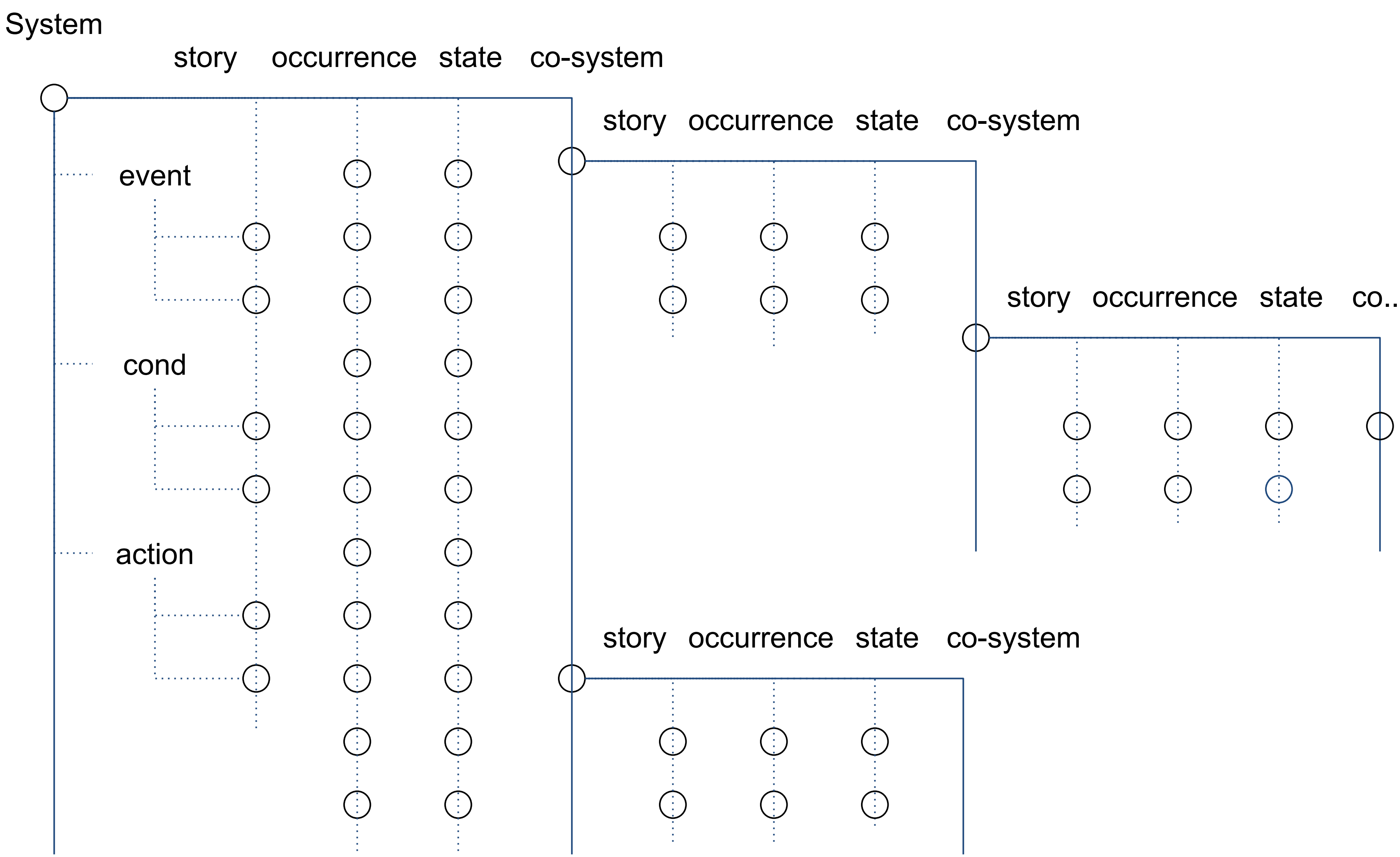


NOTE: at this stage we don’t have access to the “parent” system

System completion

The process ends when all narratives are complete, and all occurrences are mapped to corresponding co-systems states or state changes.

System Structure[1]



1. in System's view

Important Notes

1. The methodology must not be strictly enforced, e.g. some users may start directly and work mainly with states, others with stories, etc. The important thing is that the mapping cover all the System's relevant *events*, *actions* and *conditions*, as this is what will enable its *execution*[1]
2. The co-systems of a System are also Systems, in whose structure *this* System may (or may not) appear as a co-system, be it *parent*. The hierarchical organization of these entities only exists from each System's point of view, and it is not even required that it be consistent among them, so long as they can co-exist and function appropriately.
3. Each System's view in addition is conditioned by what the others publish[1], e.g. some systems will only access the states of their co-systems and have no need for their stories, whereas others will not have access to their states, and will use only public *functions*[1].
4. The methodology can be applied indifferently top-down or bottom-up — where the *end user* is traditionally considered at the top — as from a System's perspective there is no such notion: it is all about who or what is intended to subscribe to *my* story, and how *in my view* these co-systems relate to each other.

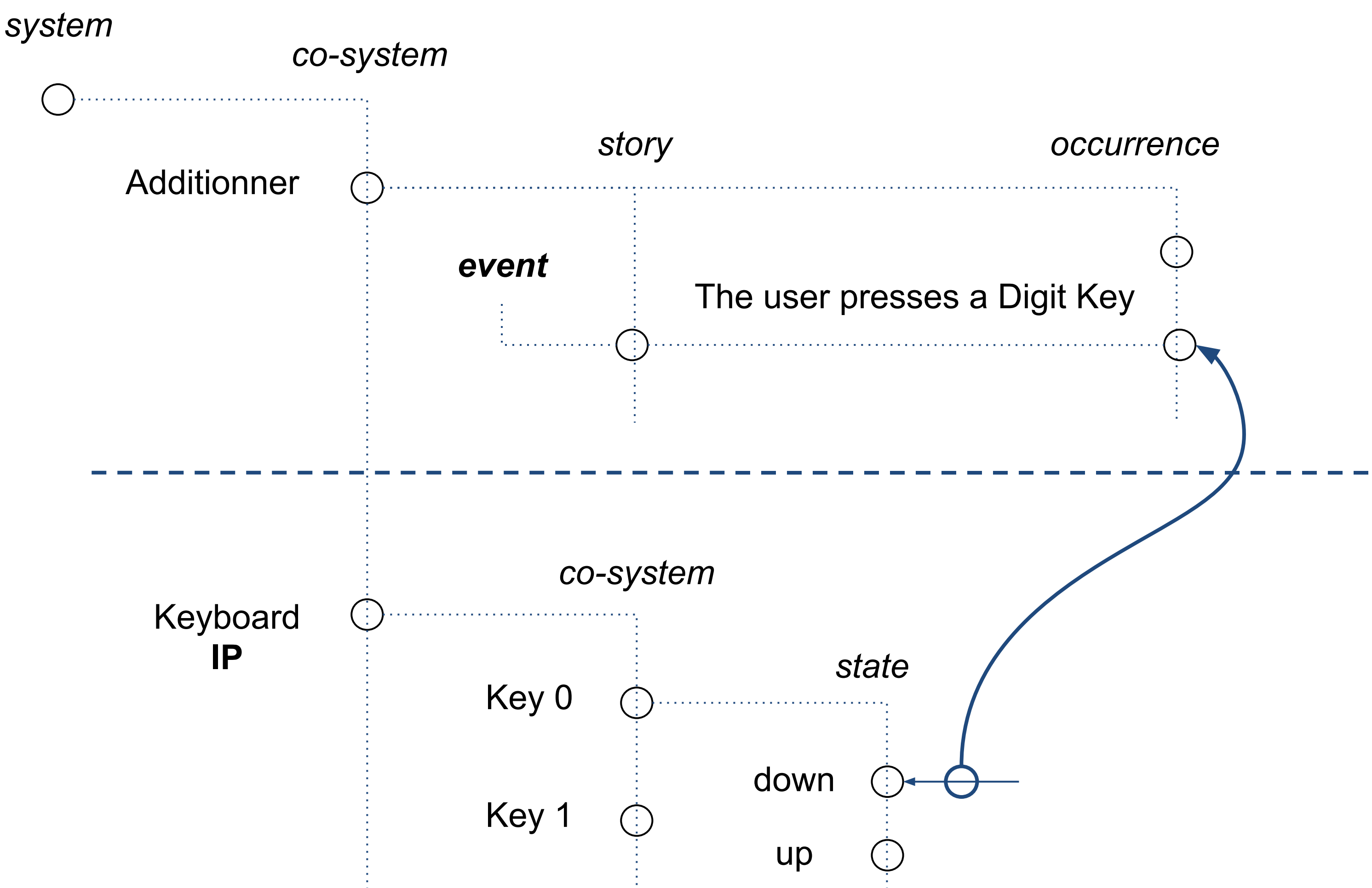
1. See later in this presentation

Interfaces

Interface with other systems

- Each system entity (story, co-system, occurrence, state, ...) can be set to either public or private
- Systems can connect to each other via their IP address and interact by
  - listening to each other's public *events* or *state changes*
  - checking each other's public *conditions* or *states*
  - triggering each other's public *actions* or *state changes*
- Usually the public interface of a system, or service front-end, will be implemented as a public co-system calling into the other, private systems' entities

Example

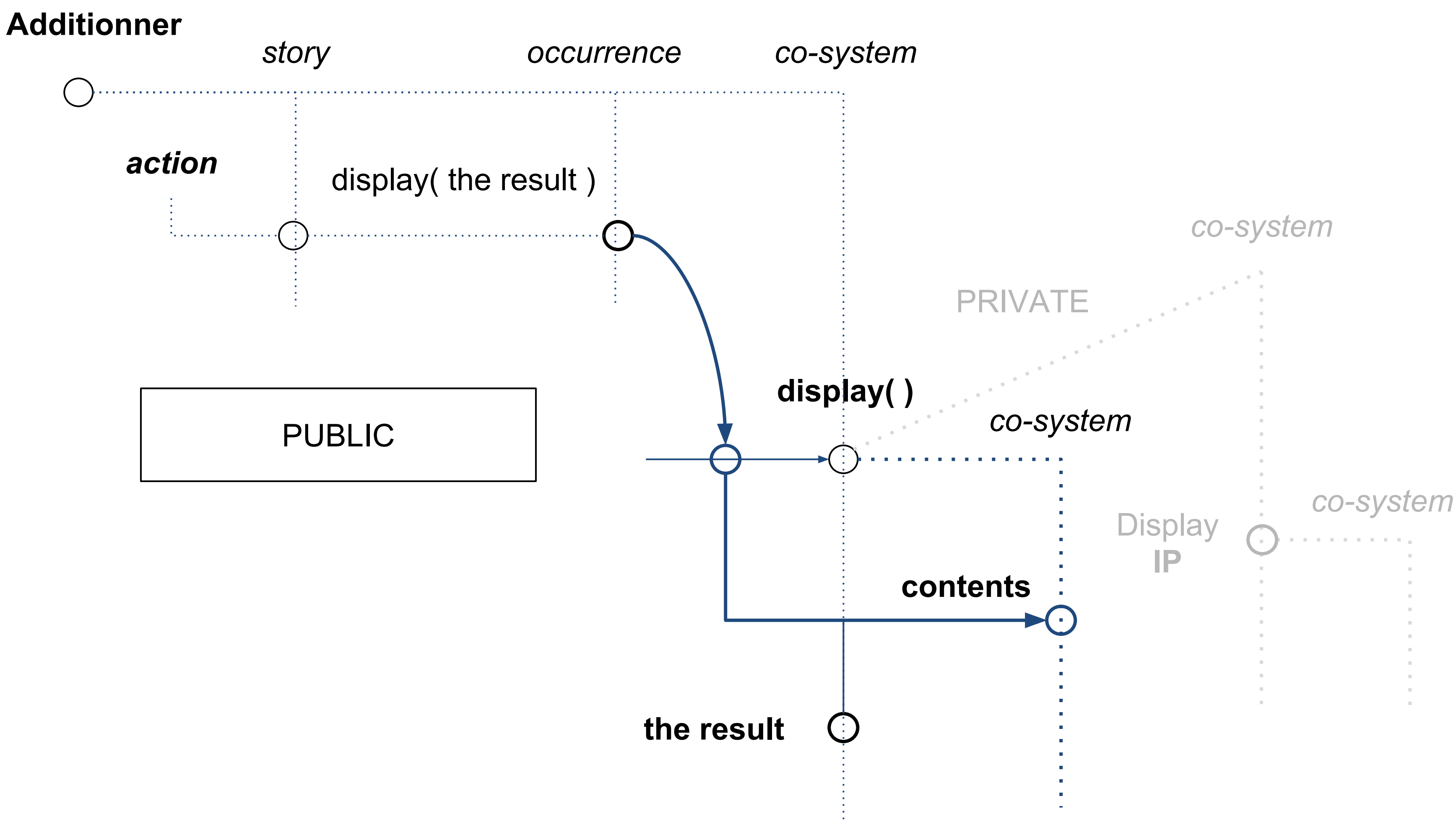

system
co-system
Additionner
story
occurrence
event
The user presses a Digit Key
co-system
Keyboard
IP
state
Key 0
down
Key 1
up

Functions

- A Consensus *action* can be made generic by
  - Moving the action's *narrative* to a dedicated **function** *co-system*
  - Replacing all references to the generalized *co-systems* with references to so-called **parameter** *co-systems* of that function
  - mapping the original *action* to a single *occurrence* invoking the *function*, together with **argument** *co-systems* — as in

    [: display (the result)]
- If published, the function can then be invoked by other systems using the appropriate occurrence — e. g. Additionner.display (the result)
- The function can in turn call back the caller's *set* and *get* value functions and trigger value changes — as in [: compute (the result)]
- It is the responsibility of the *function* to publish appropriate *states* and *state changes* e. g. for synchronization purpose

Example



Additionner
story
occurrence
co-system
action
display(the result)
co-system
PRIVATE
PUBLIC
display( )
co-system
co-system
Display
IP
contents
the result

Value and Account

- The **value** of an entity exists only in relation to a **value system**.
- A **value system** is a system that guarantees that value-system-specific events will occur when the entity is placed via a value-system-specific function into value-system-specific operating conditions.
- The value-system-specific value data of an entity are located in a value-system-specific **account** allocated in a **value bank**.
- each entity is responsible for interfacing with its *value bank* and for triggering value-system-specific banking operations (storage, retrieval, modification) by the bank on its own value-system-specific *account*.
- These operations are private and can only be triggered by the entity via its corresponding Consensus *set* and *get* Value functions, on demand from other entities.

  Note that the Consensus *get value* operation only makes sense in the context of the caller's account *assignment*, or *set account* operation

Account Operations

- The **opening** of an *account*, and likewise the **closing**, **setting** and **consulting** of the account must be requested from a *value bank* by an entity providing
  - a *unique identifier*
  - the *value system* to be associated with the account
  - a *value type* which is supported by the *value system*
  - the *value* to which the account should to be *set*
- If an account already exists with the requested characteristics, then the opening operation will fail.
- The Consensus built-in Value System allows Consensus entities to be created, assembled and navigated as per the Consensus Entity & Relationship Model. It also allows to create and integrate other Value Systems and Value Banks in the system.

Consensus Default Value Operations

- C++ equivalent: this = new(type);

  on [: ->new(*caller*, value_system, value_type)]
  do [: MyValueBank.openAccount (*this*, value_system, value_type)]

- C++ equivalent: delete this;

  on [:->delete(*caller*, value_system, value_type)]
  do [: MyValueBank.closeAccount (*this*, value_system, value_type)]

- C++ equivalent: this.setValue(value.get());

  on [: ->setValue(*caller*, value_system, value_type, value)
  do [: MyValueBank.setAccount (*this*, value_system, value_type,
  value.get(*this*, value_system, value_type))]

- C++ equivalent: this.getValue();

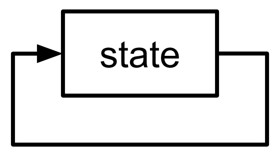
  on [: ->getValue(*caller*, value_system, value_type)
  do [: MyValueBank.getAccount (*this*, value_system, value_type)]

Interface with CMS

- Each system entity has its own web page which links appropriately to the other entities. This allows the user and others with appropriate rights to navigate the system.

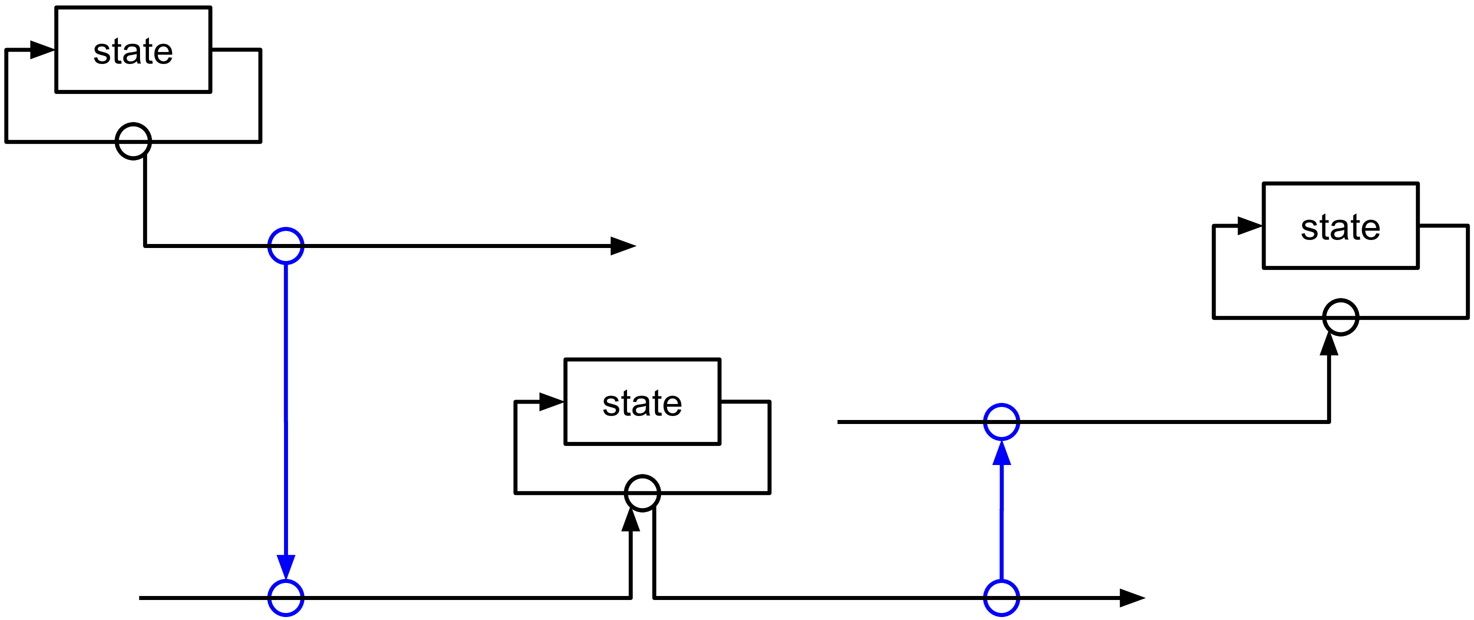
Execution Model

Change



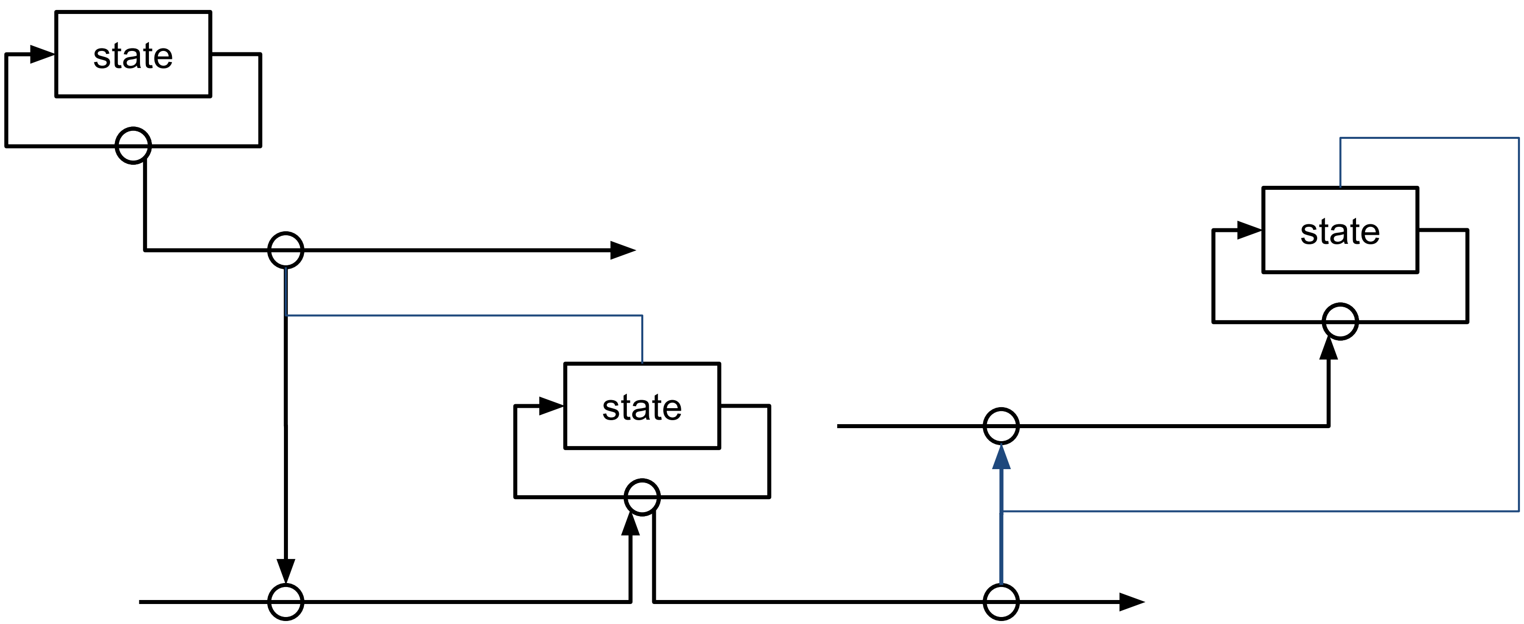
Something that changes is something that goes from one state into another

Propagation



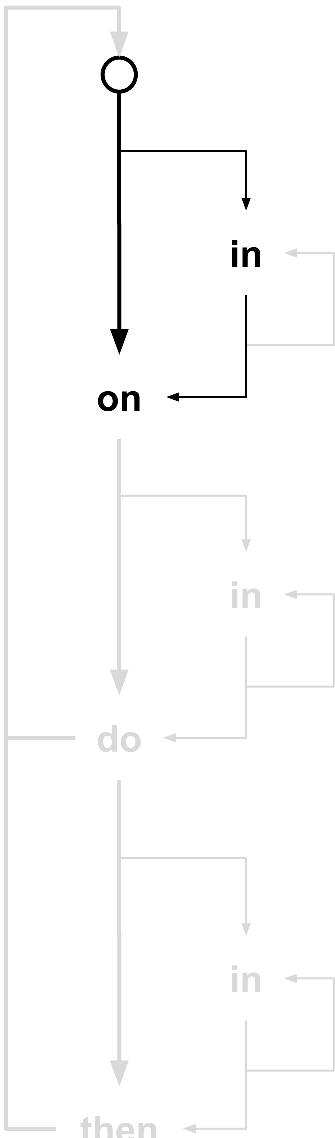
Change propagates...

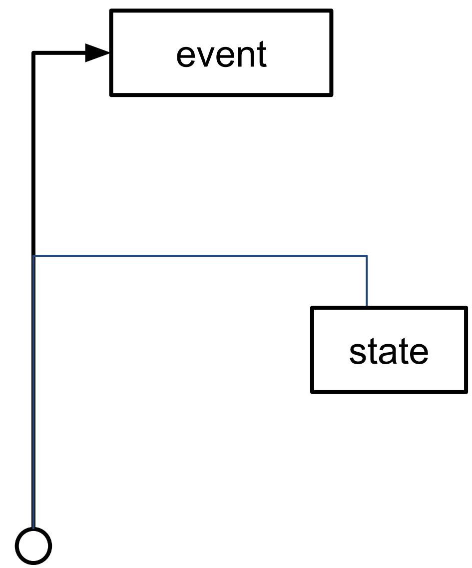
State



… or not — depending on the *receiver* state

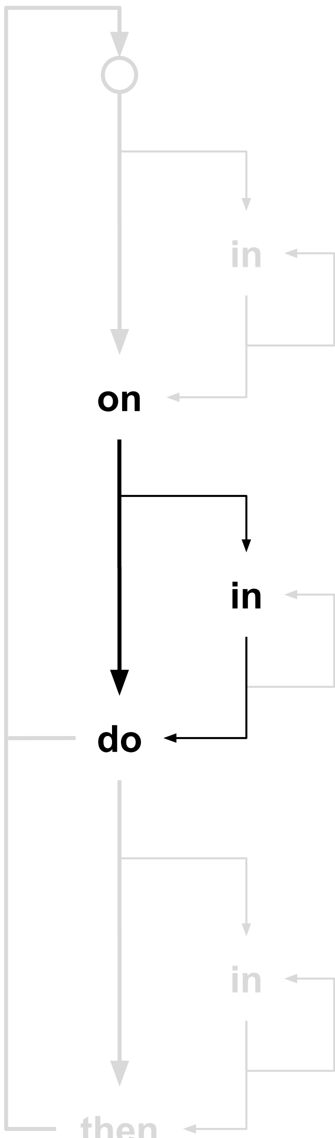
Perception

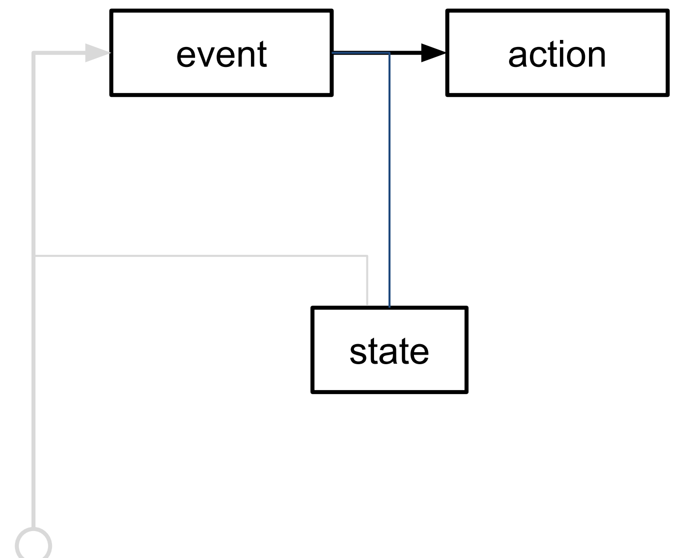




Change may trigger event

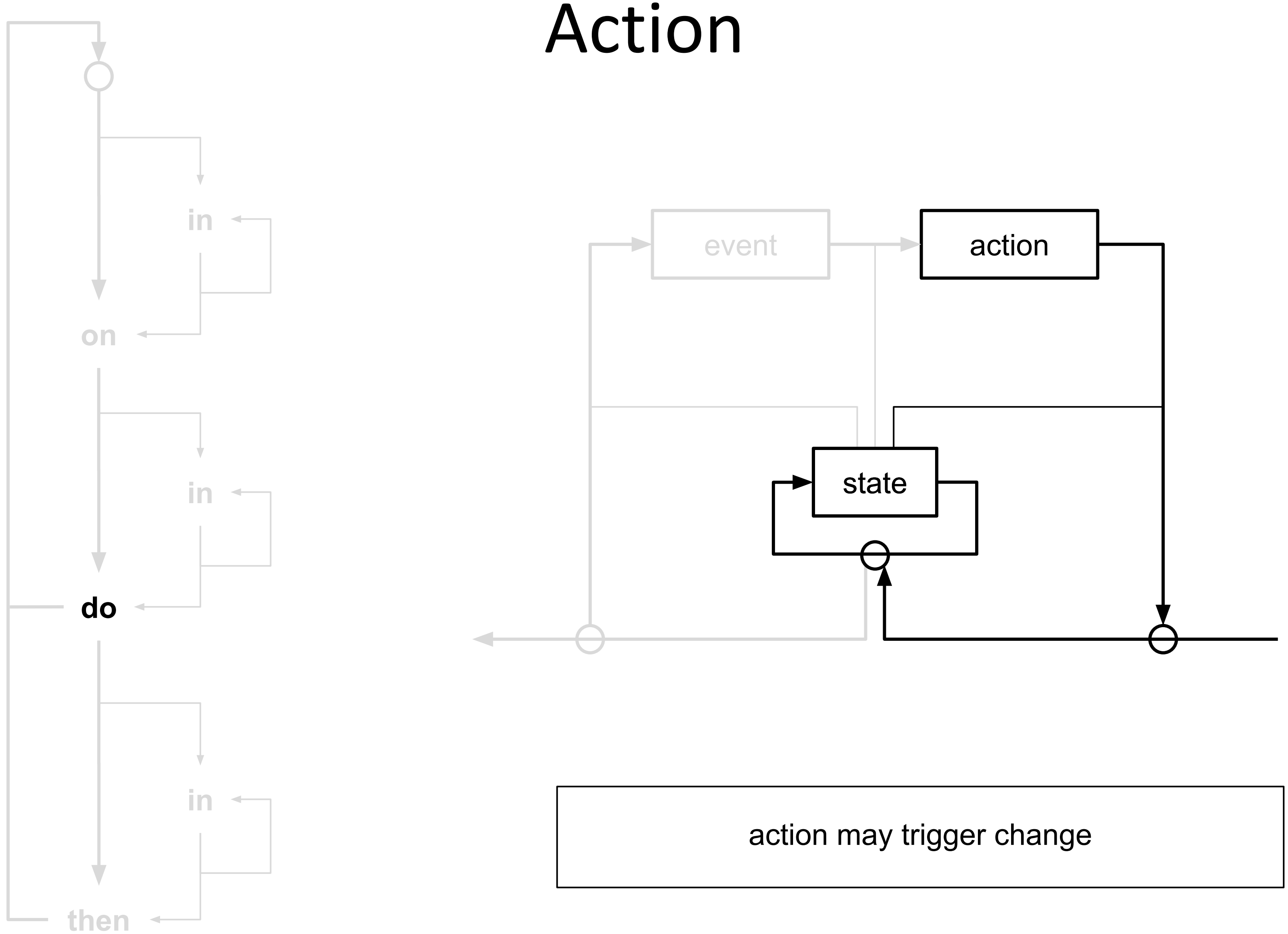
Reflexion





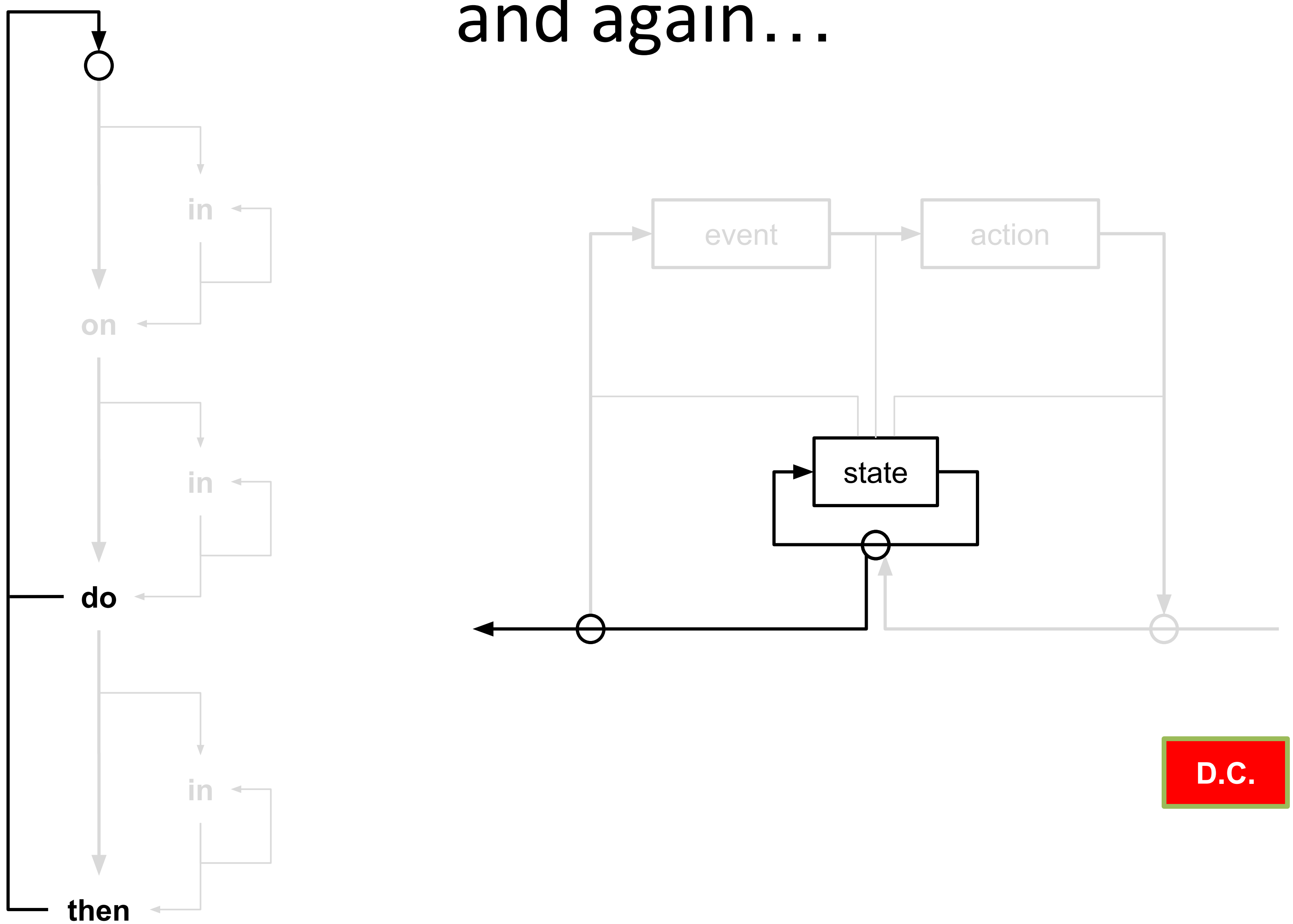
event may trigger action

Action

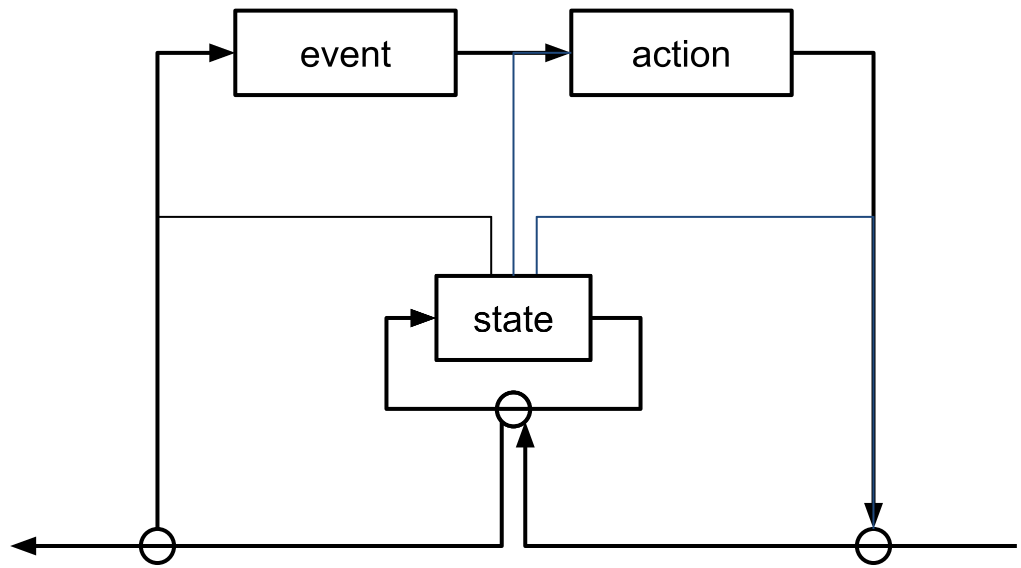


action may trigger change

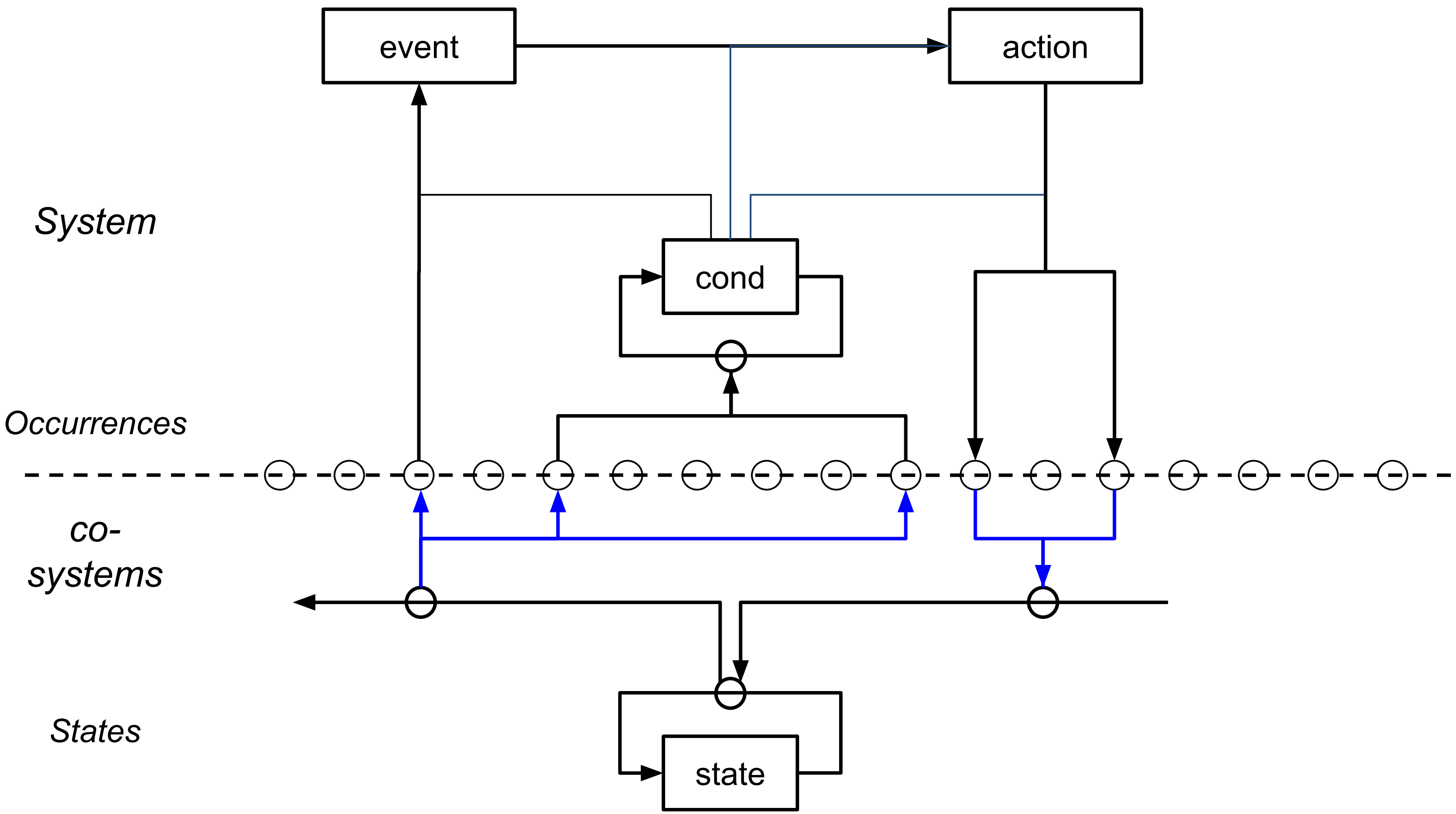
and again…



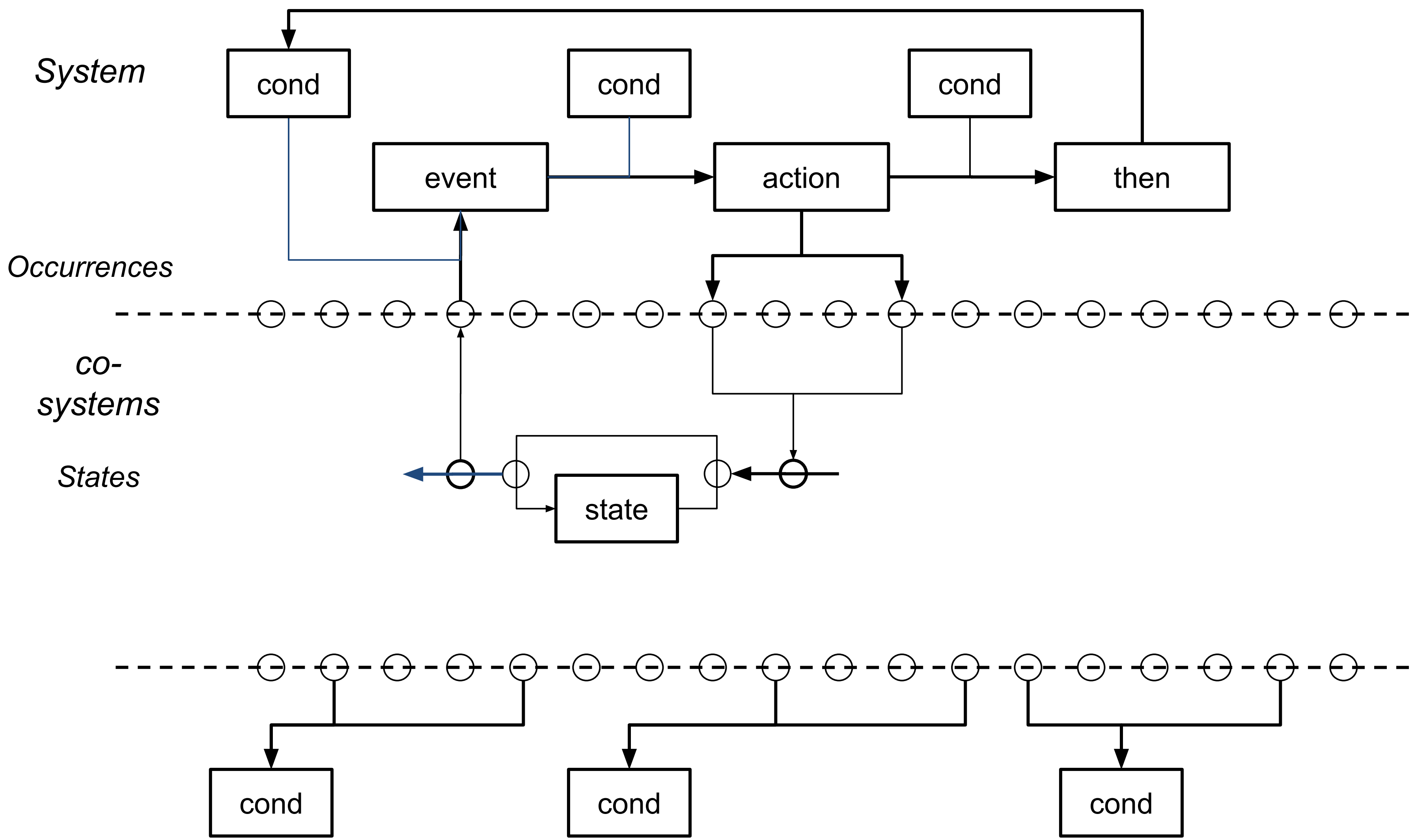
Model



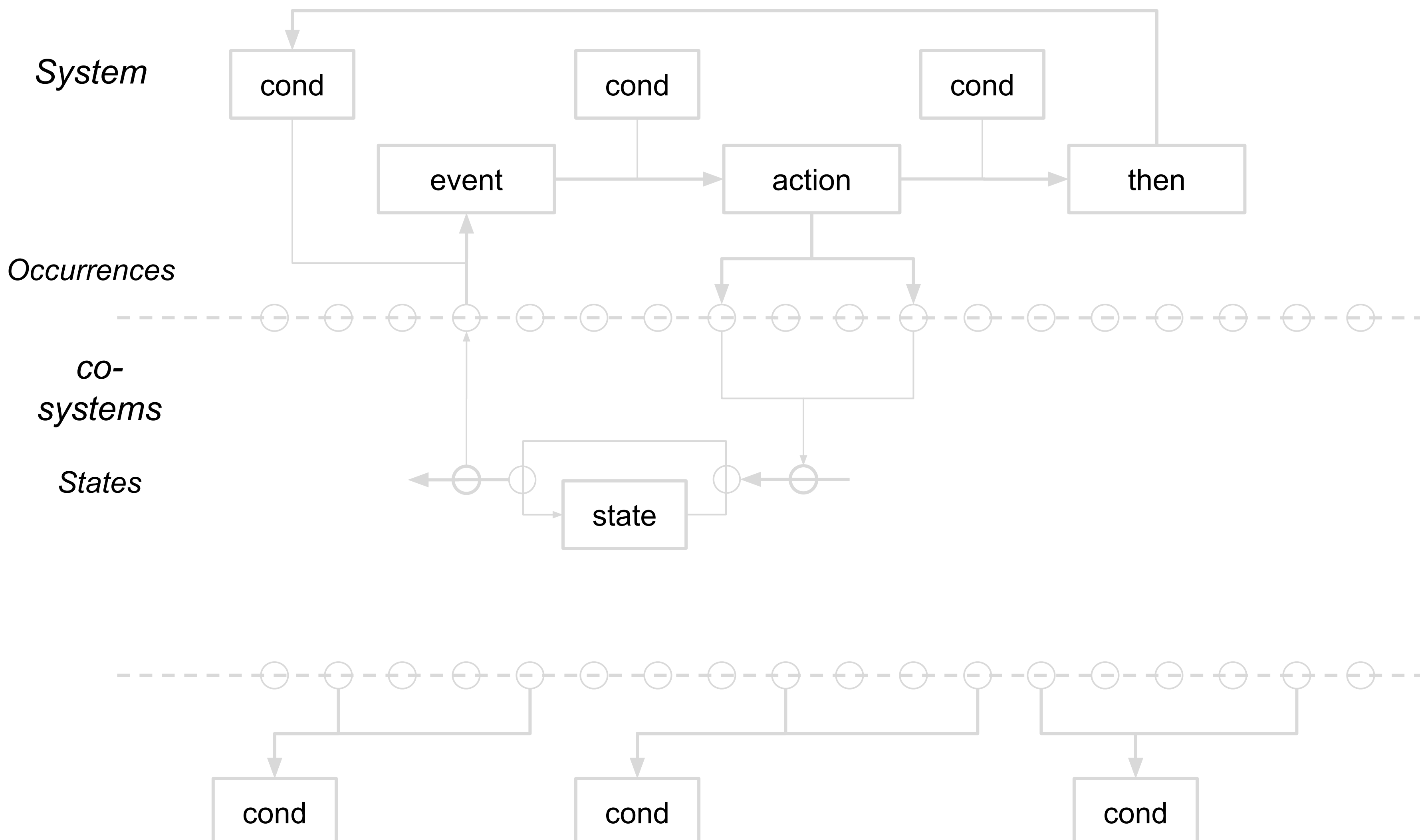
Implementation



Simulator

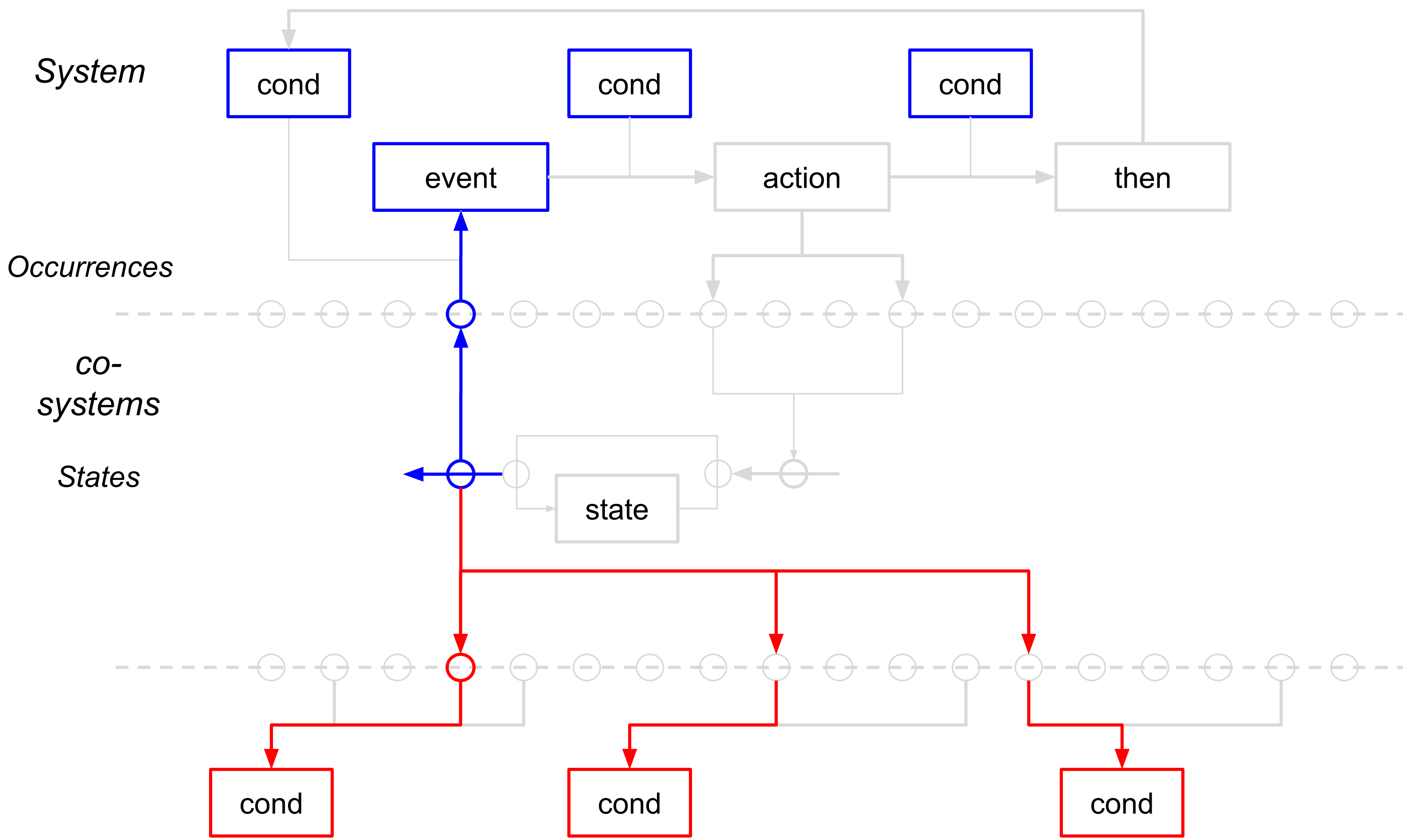

System
cond
cond
cond
event
action
then
Occurrences
co-
systems
States
state
cond
cond
cond

in…

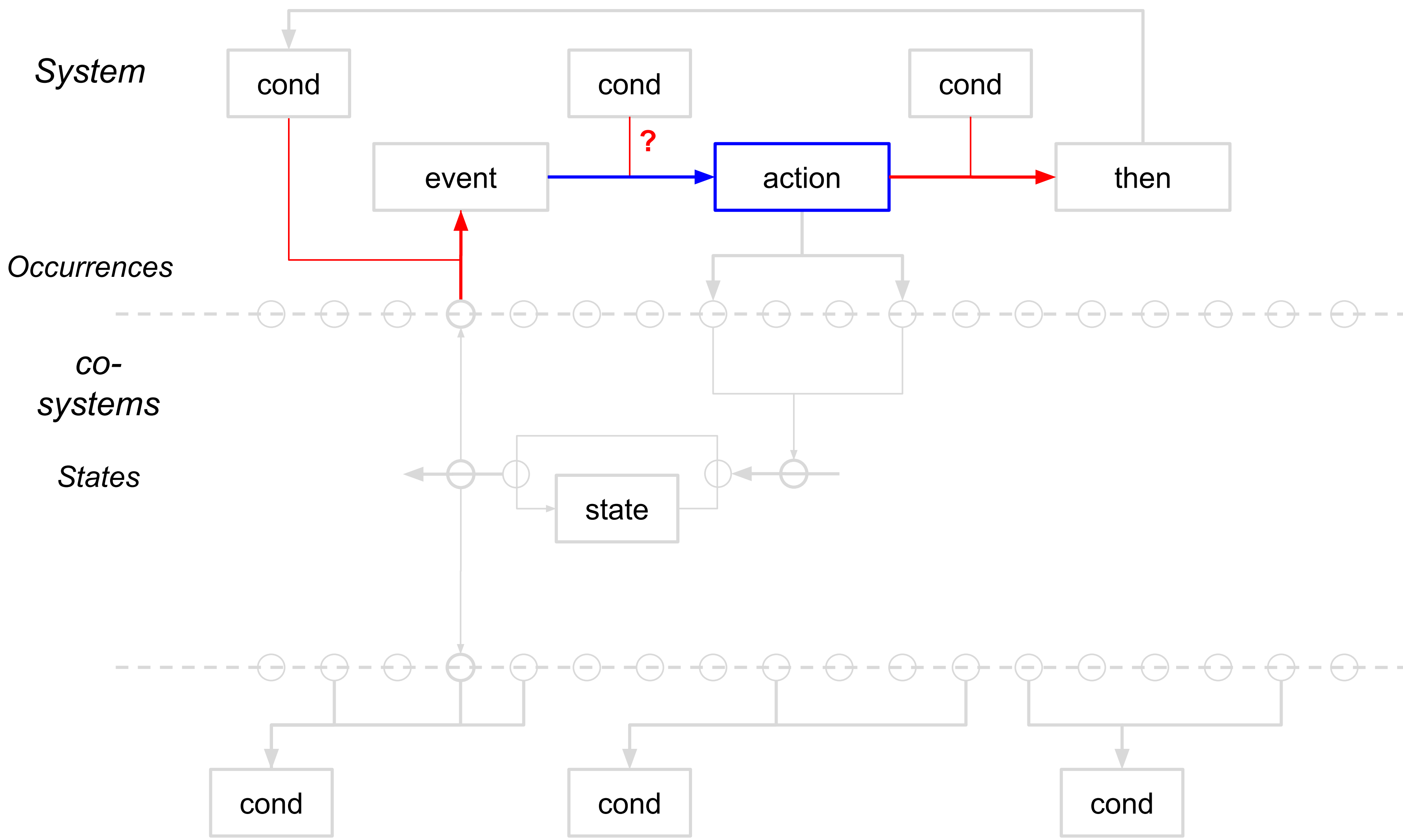


System
cond
cond
cond
event
action
then
Occurrences
co-
systems
States
state
cond
cond
cond

… on…

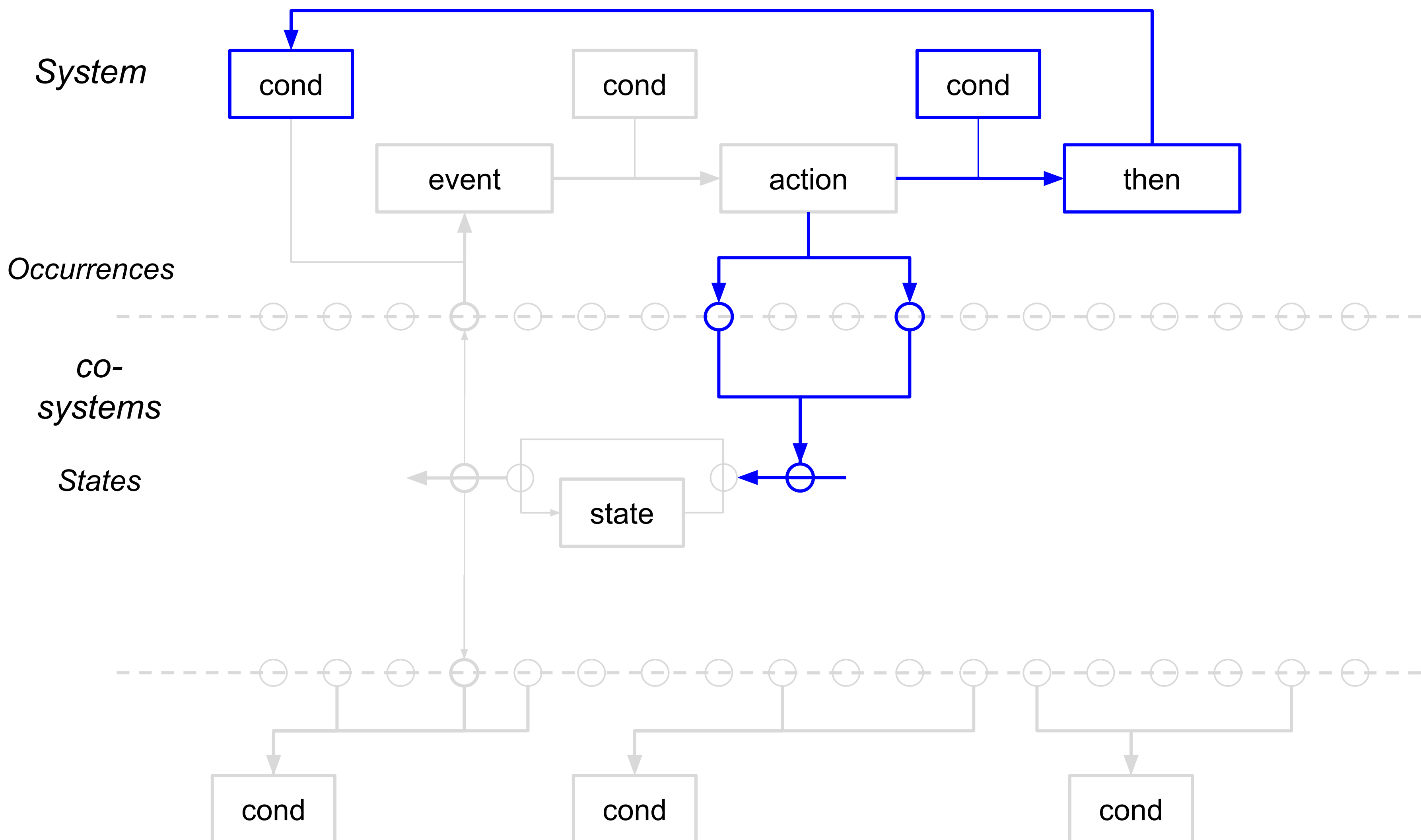

System
cond
cond
cond
event
action
then
Occurrences
co-
systems
States
state
cond
cond
cond

… do…



System
cond
cond
cond
?
event
action
then
Occurrences
co-
systems
States
state
cond
cond
cond

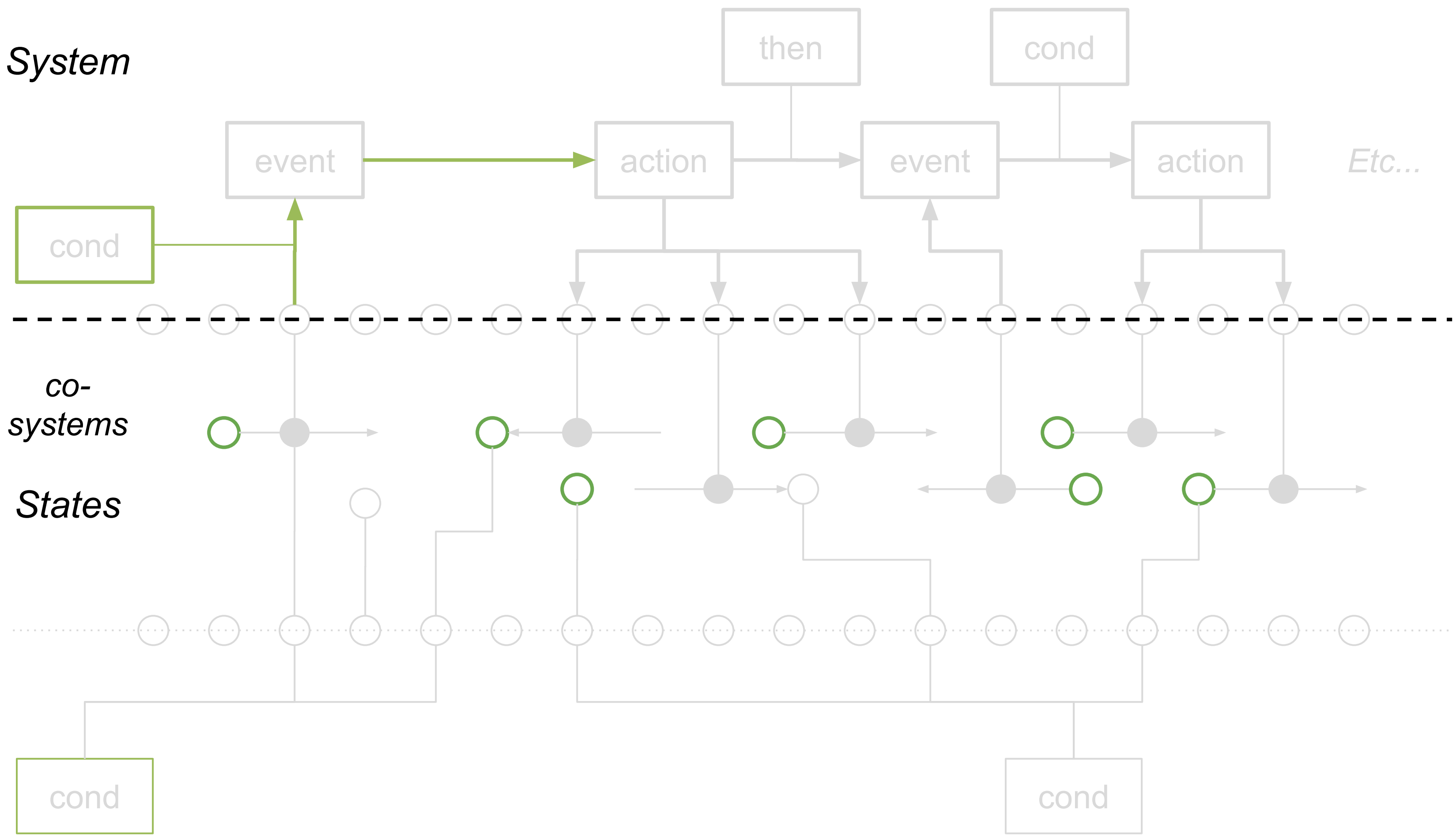
… then…



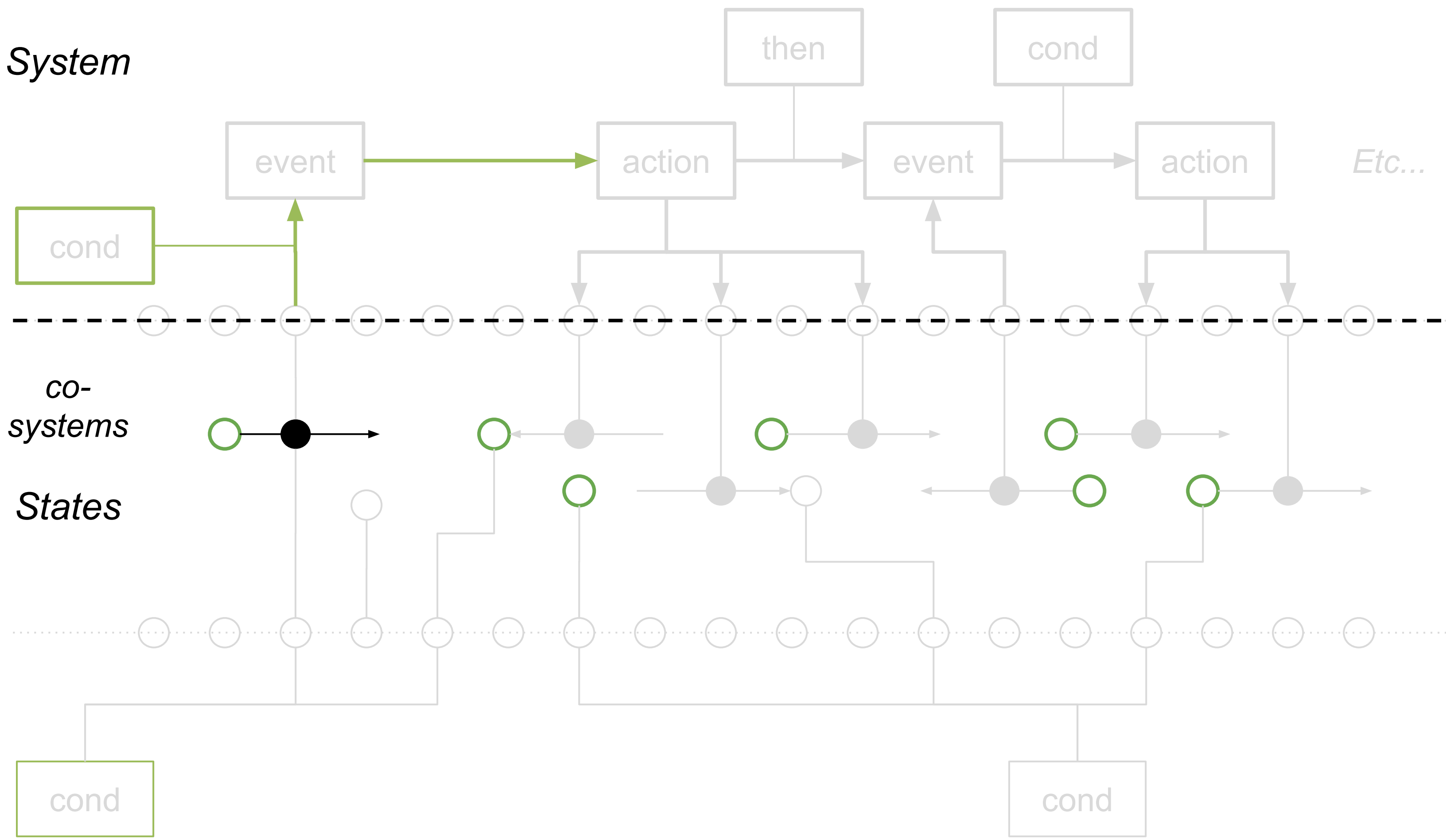
System
cond
cond
cond
event
action
then
Occurrences
co-
systems
States
state
cond
cond
cond

Example

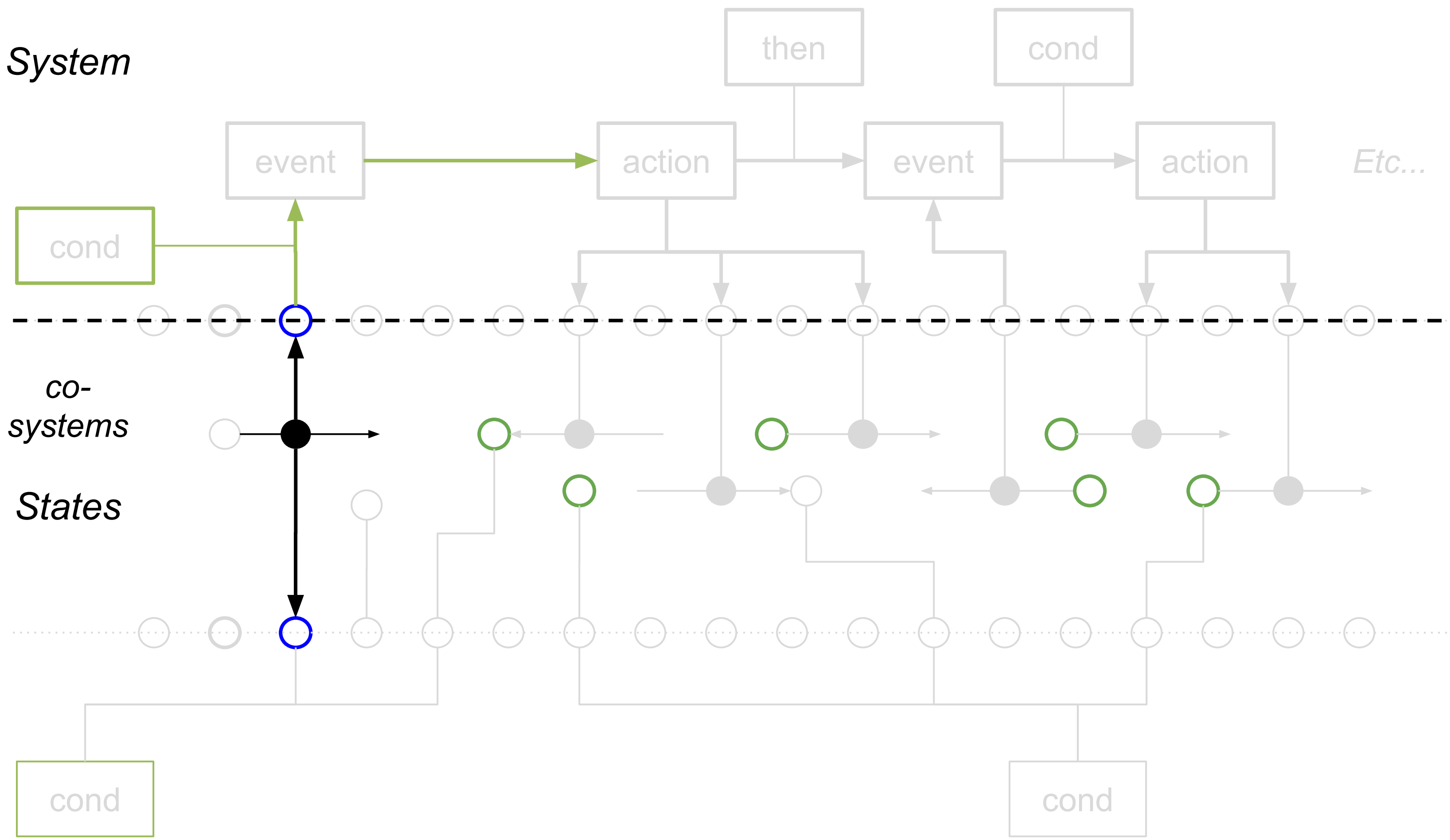
Initial State



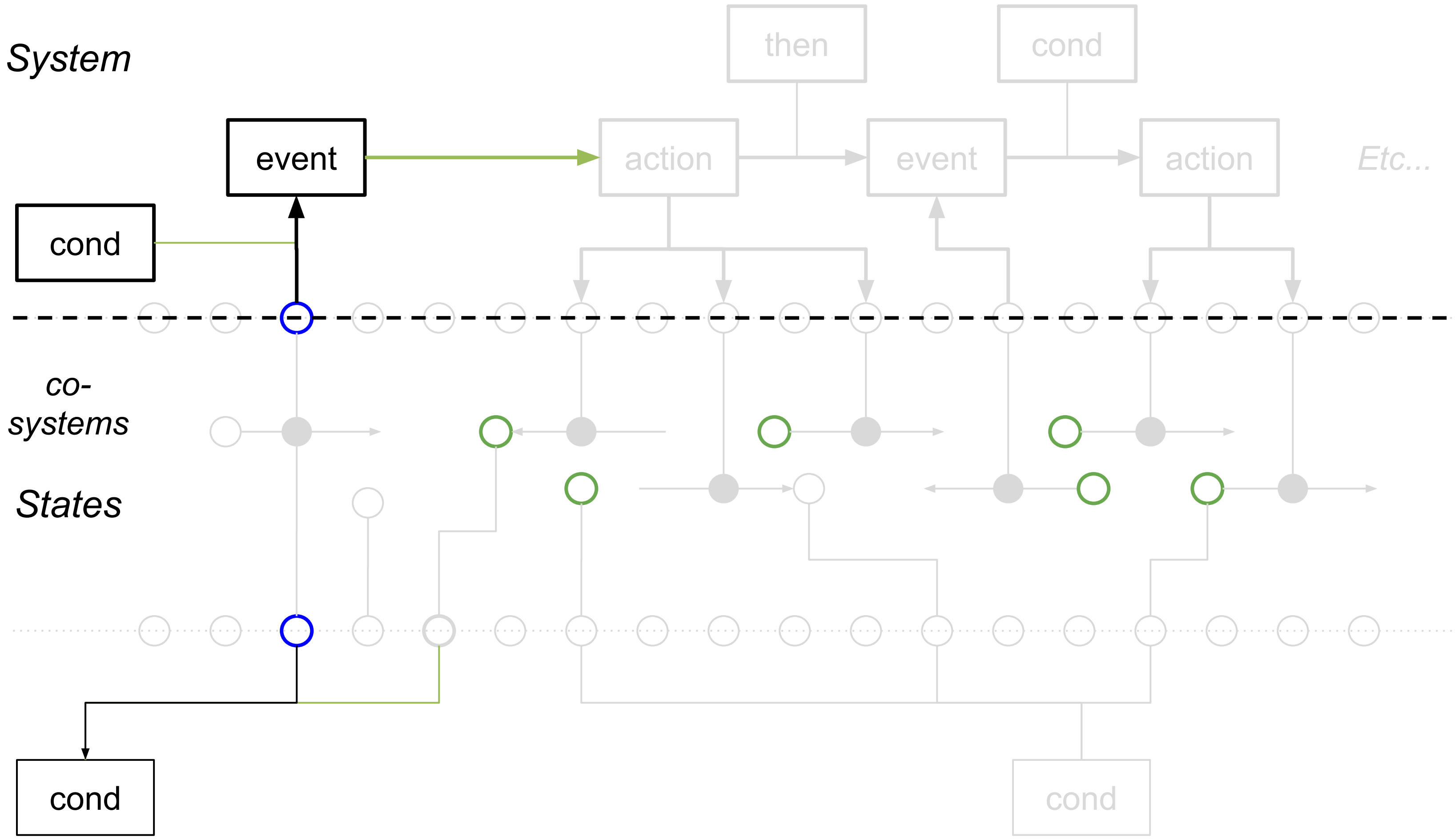
Change happens



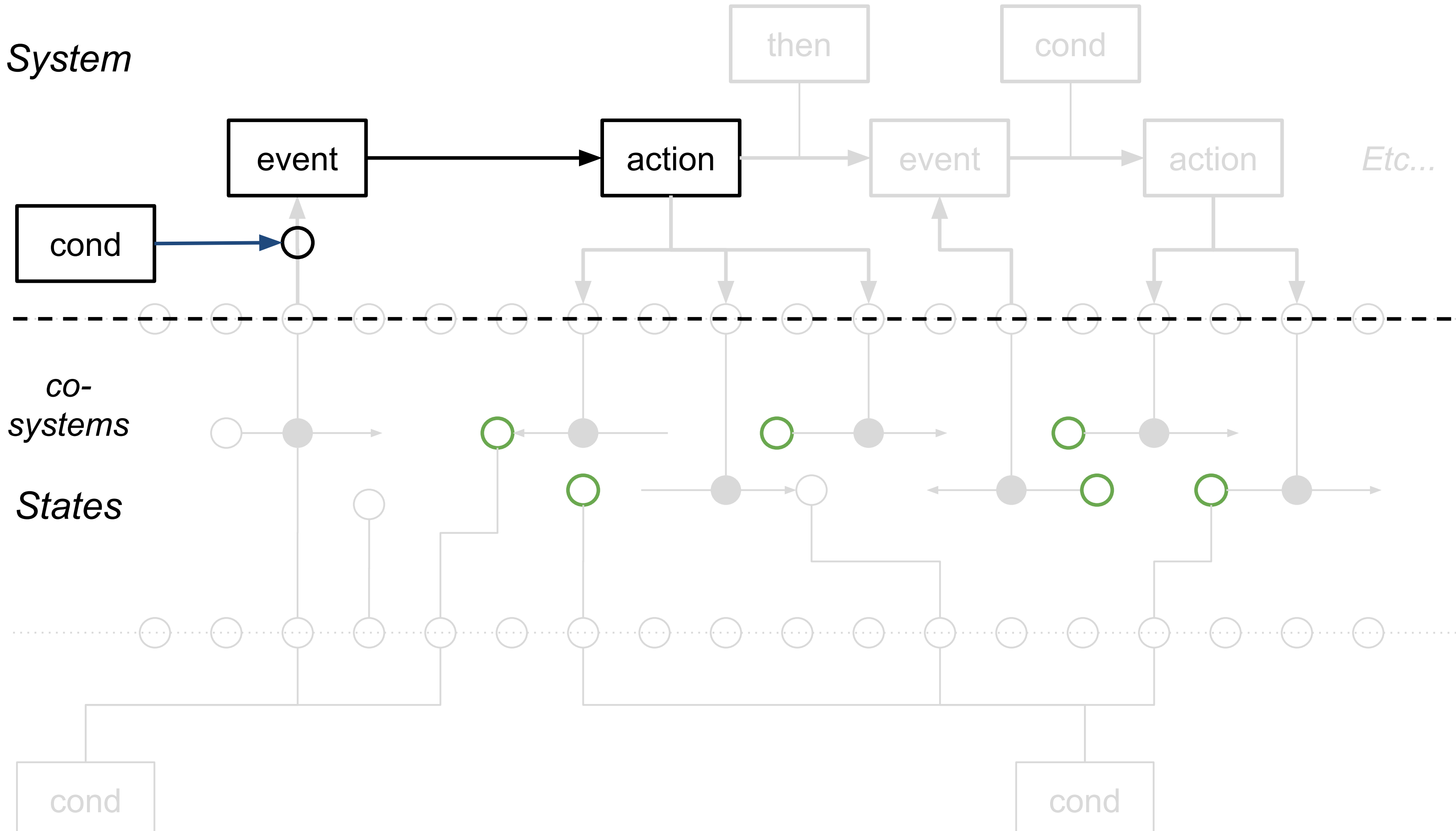
Change Propagates



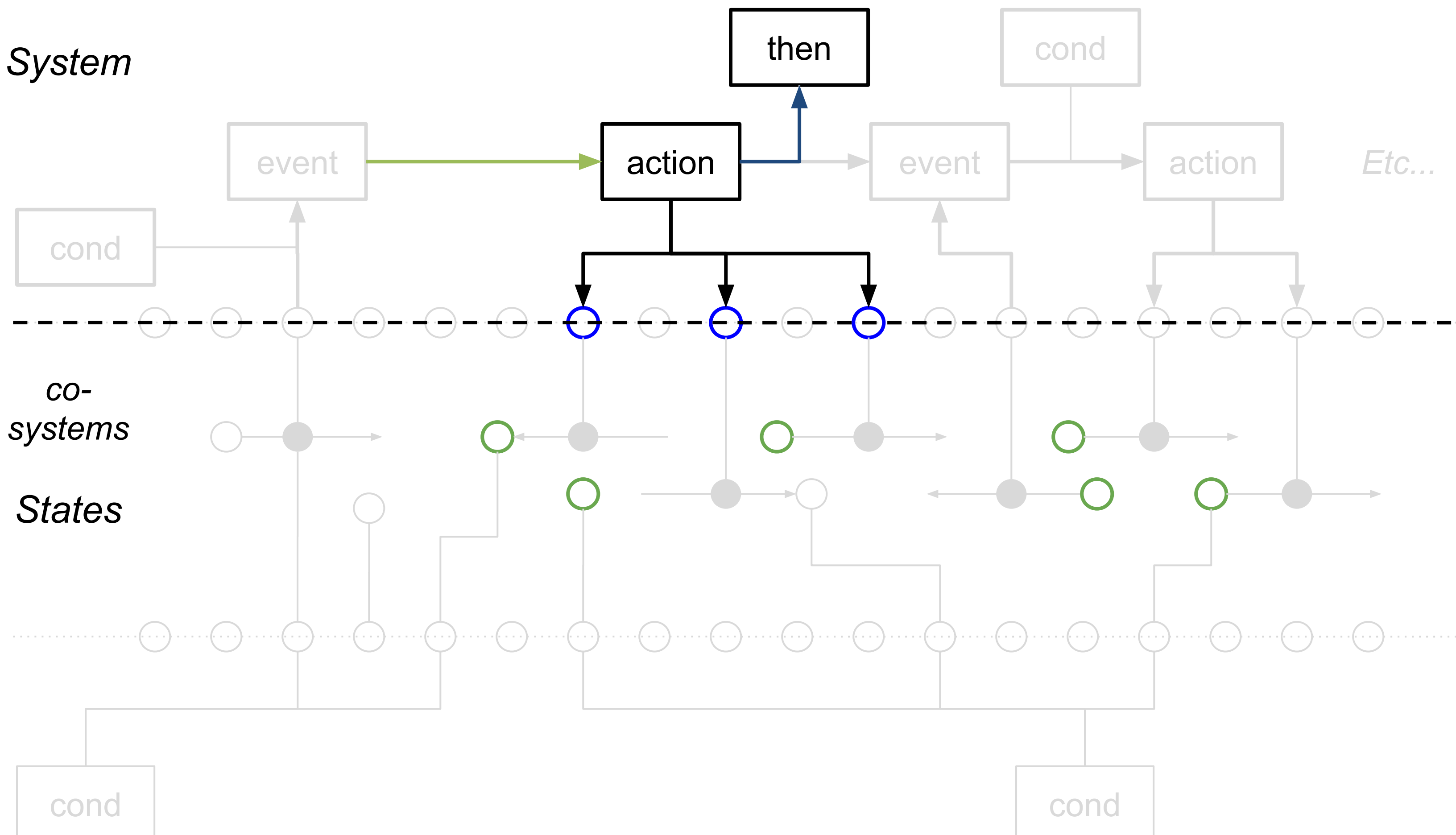
Condition and Event are notified



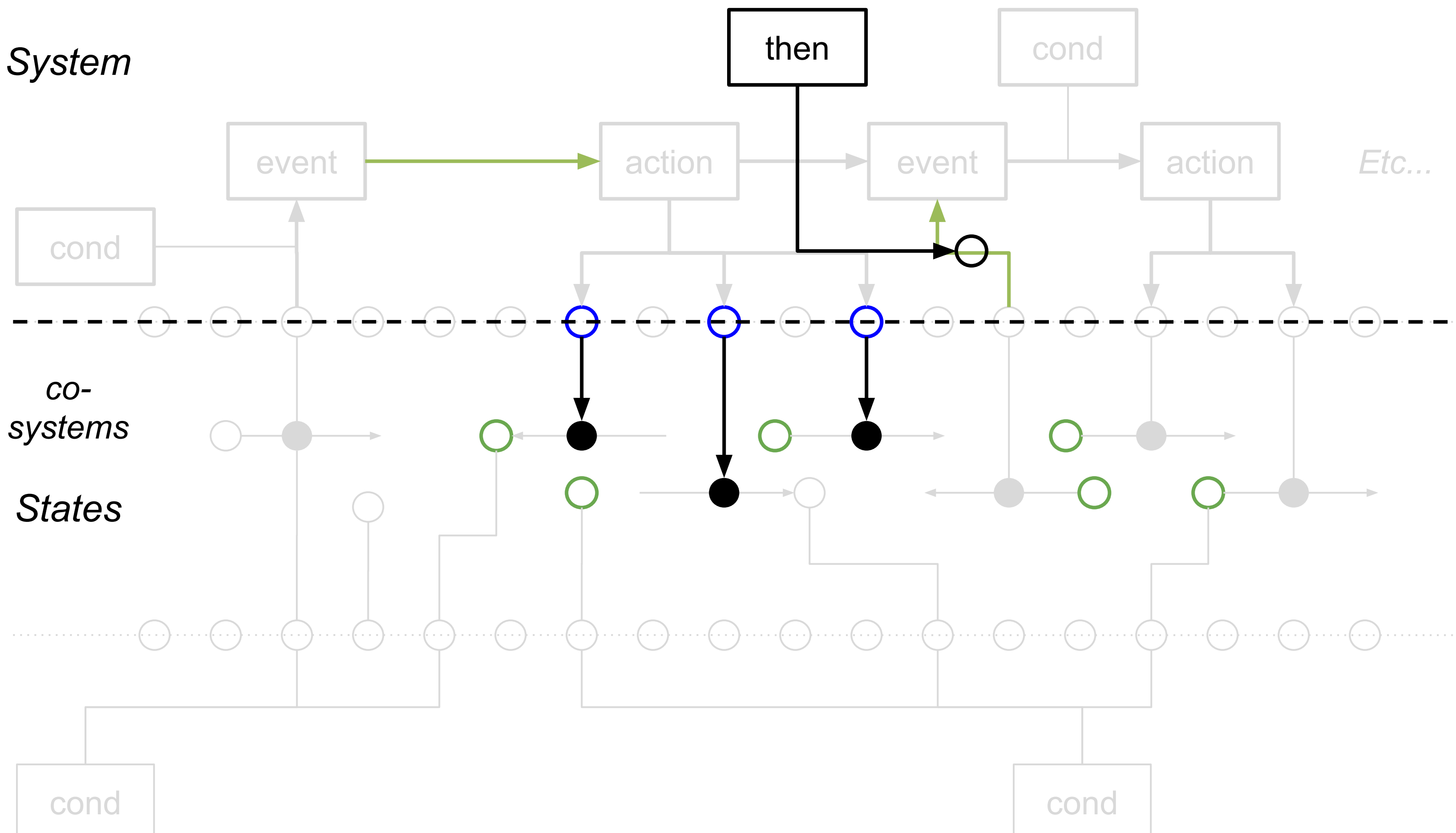
Event triggers Action, Condition applies



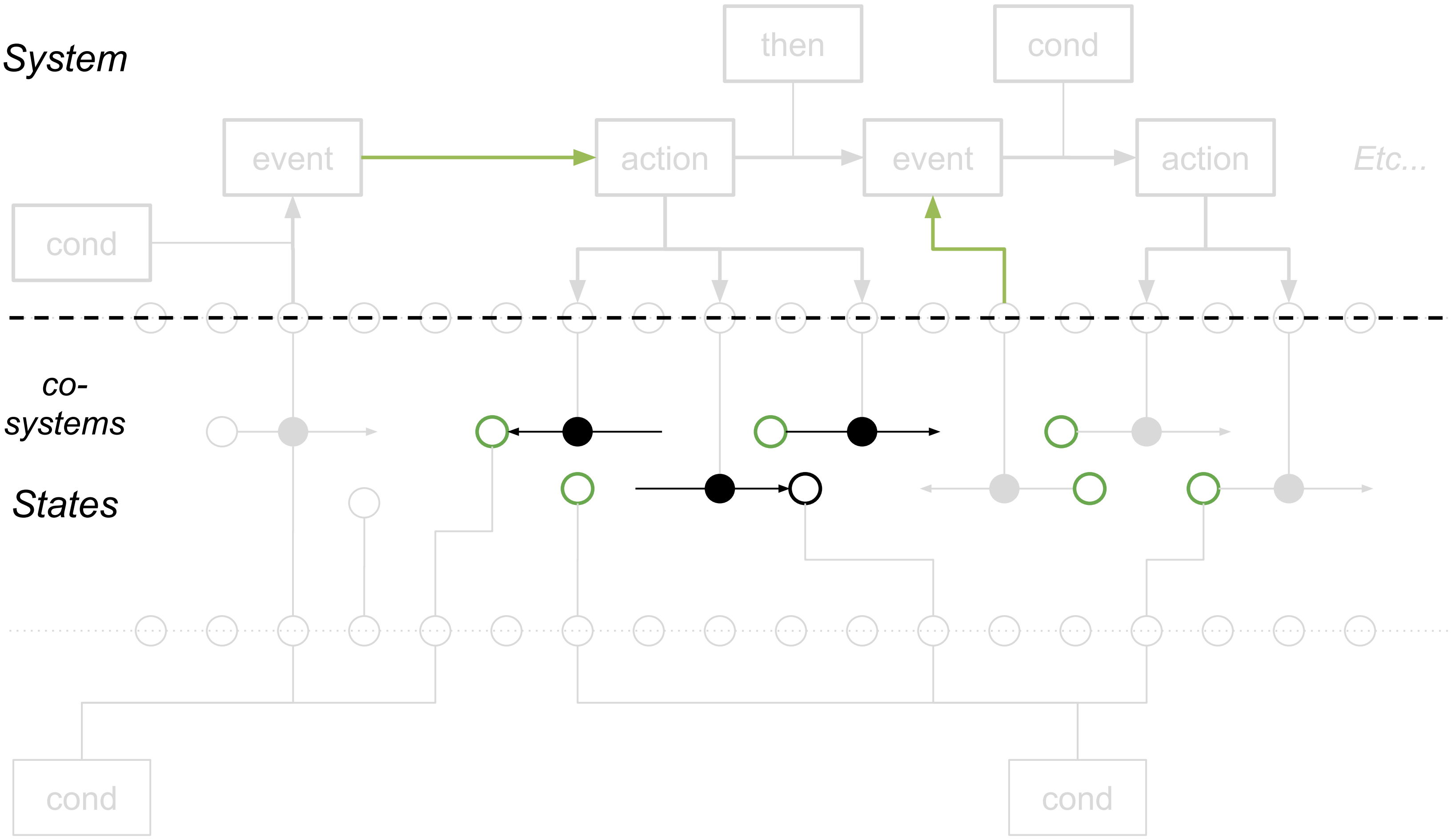
Action executes



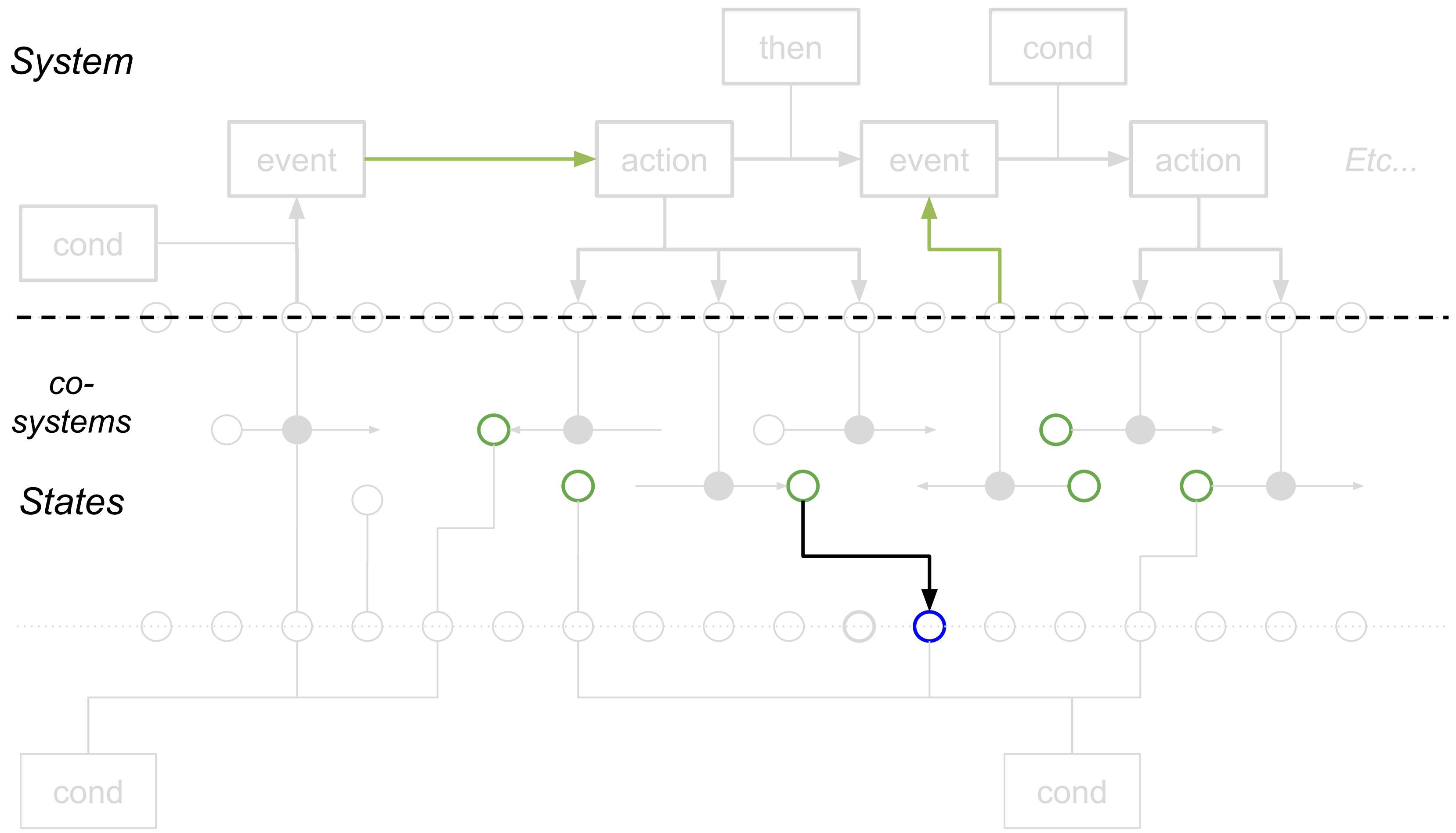
Changes are triggered, Condition applies



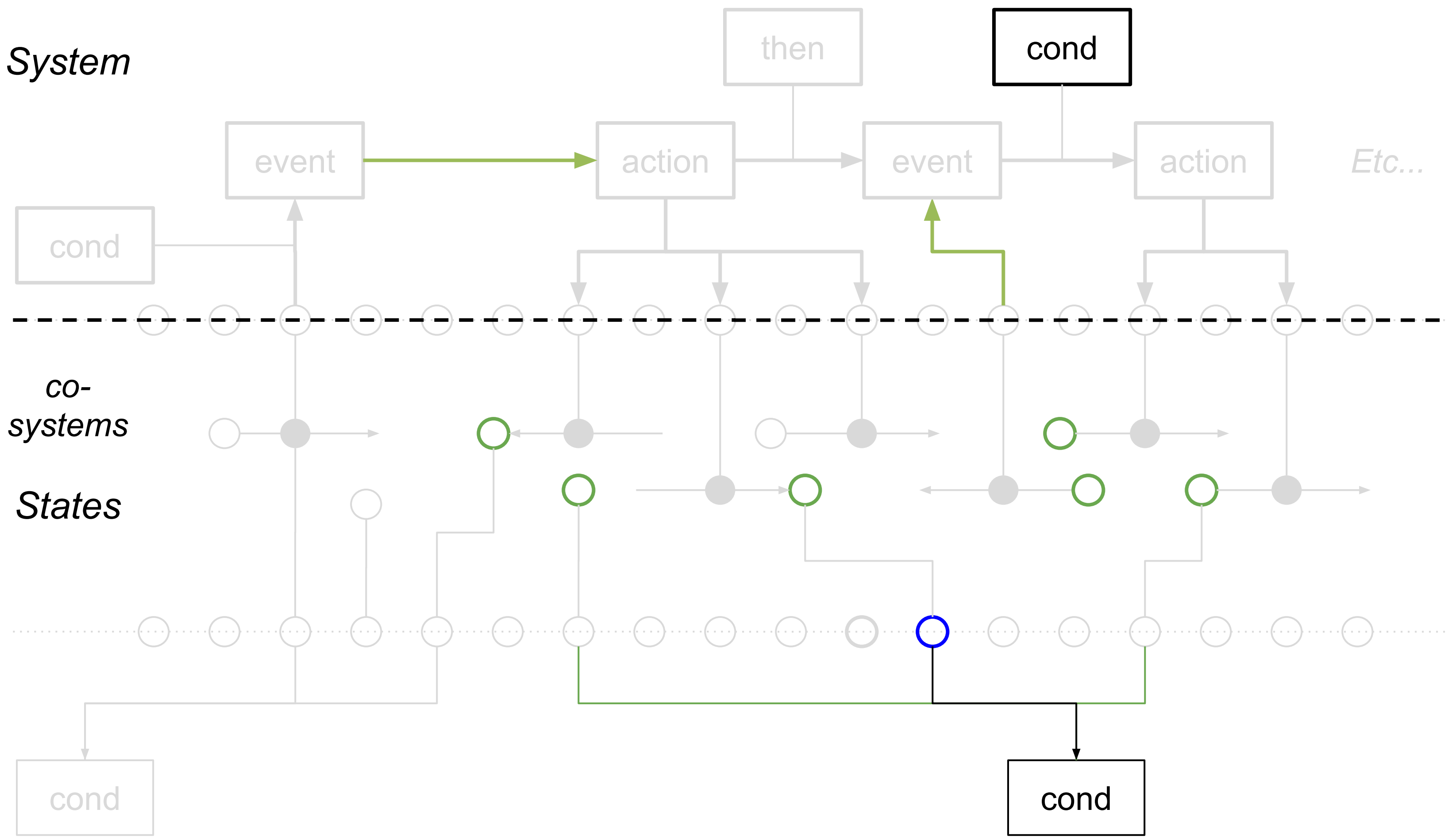
Changes happen



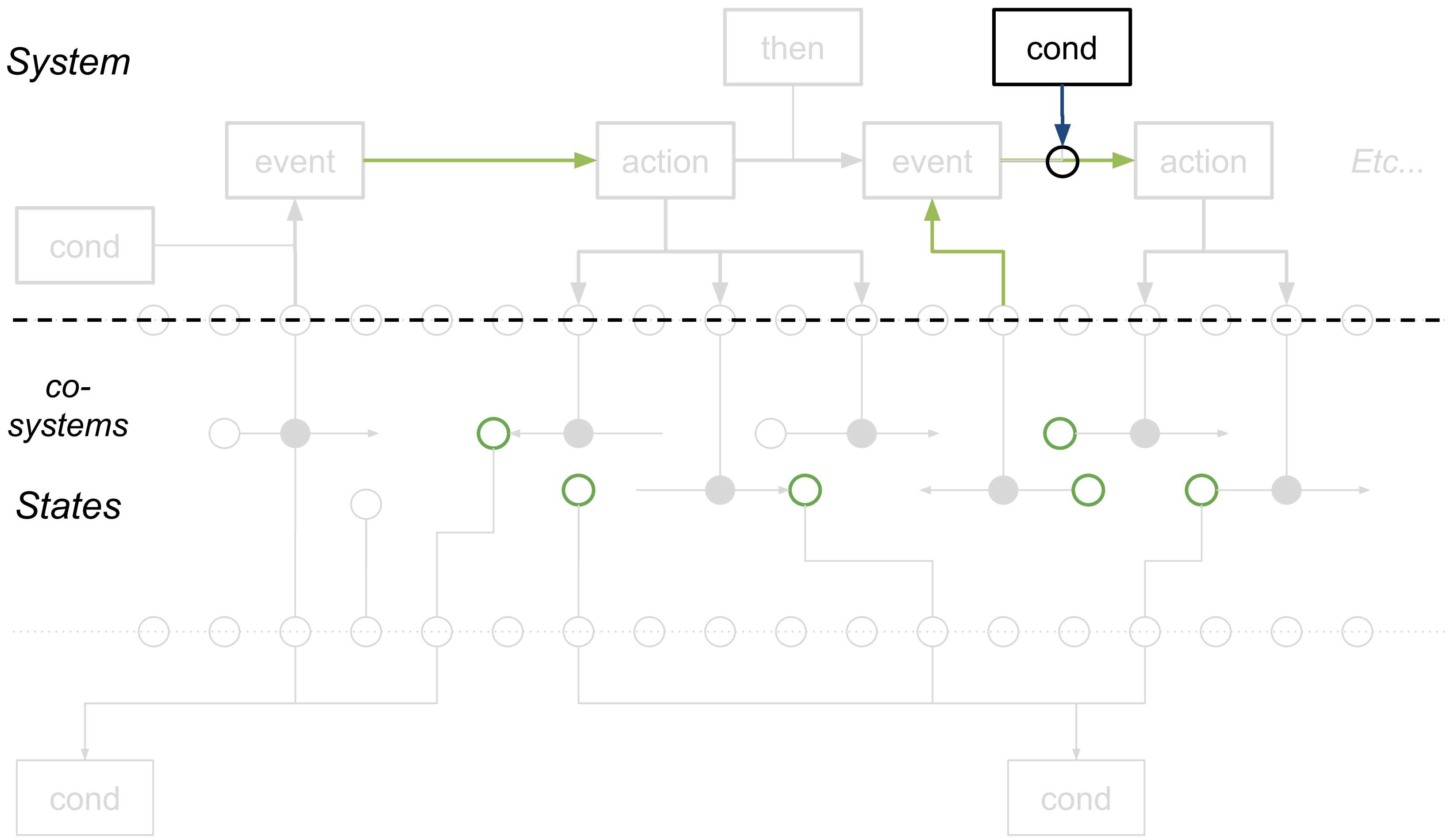
Change propagates



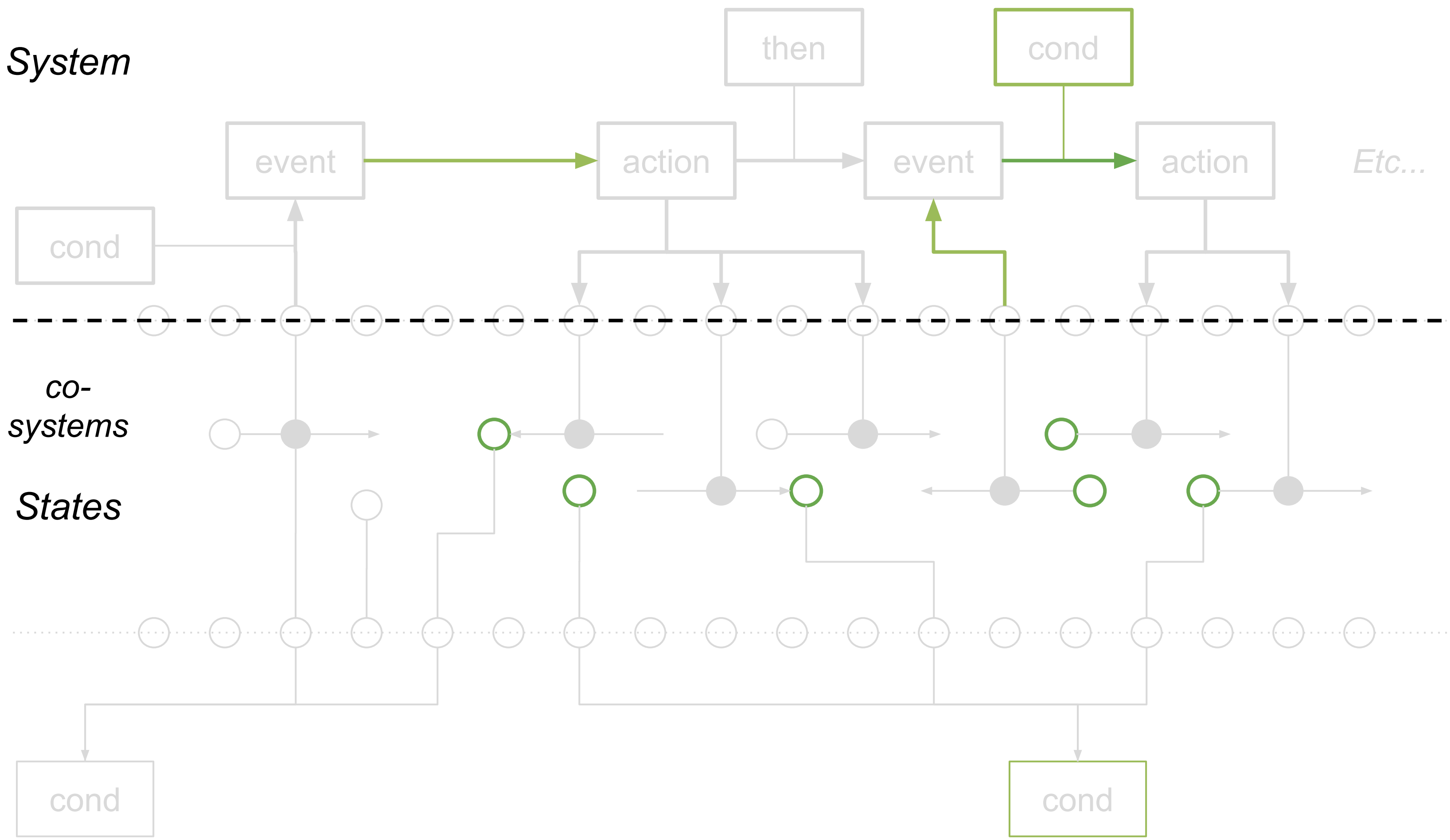
Condition is notified



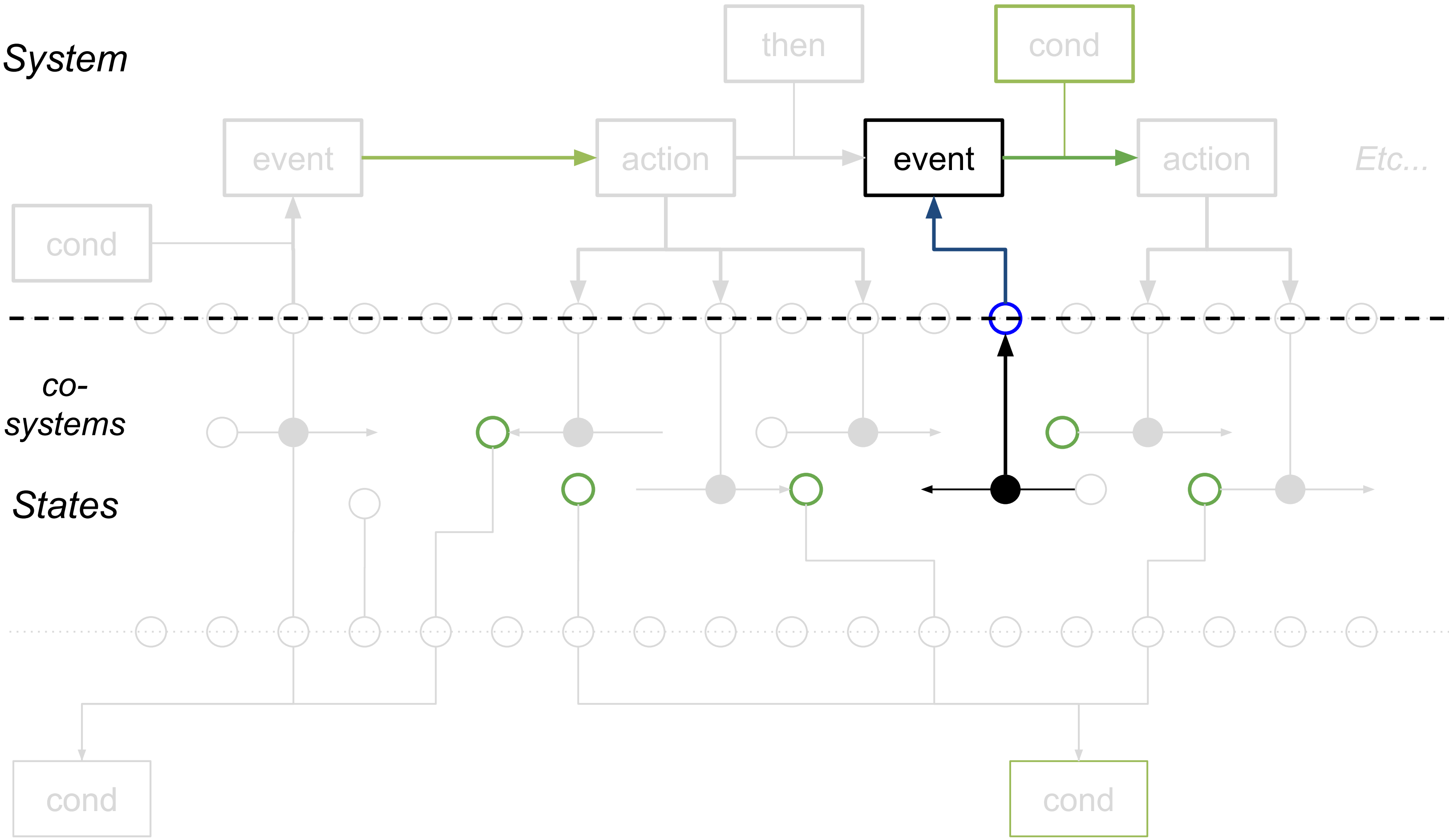
Condition applies



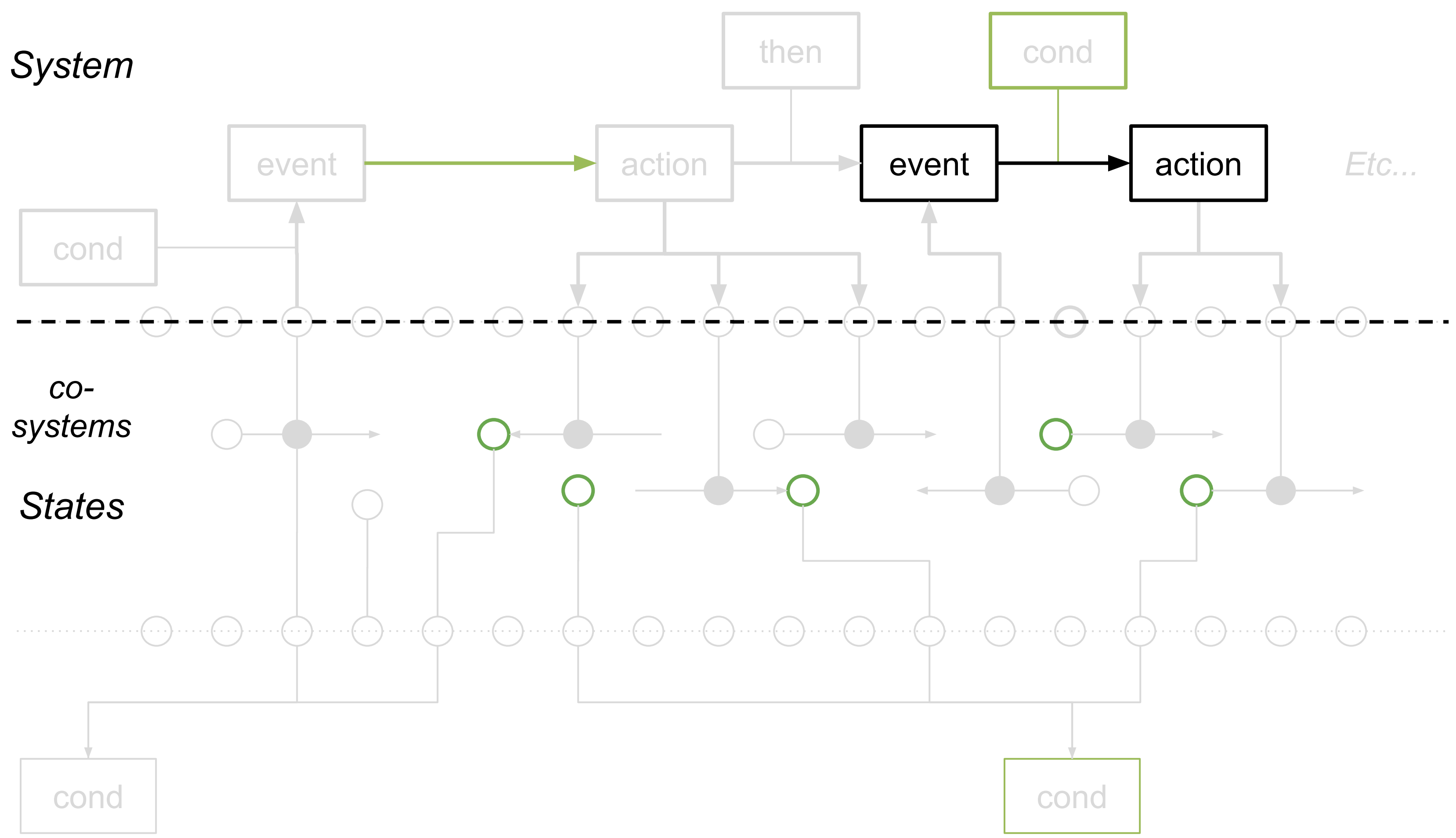
New System State



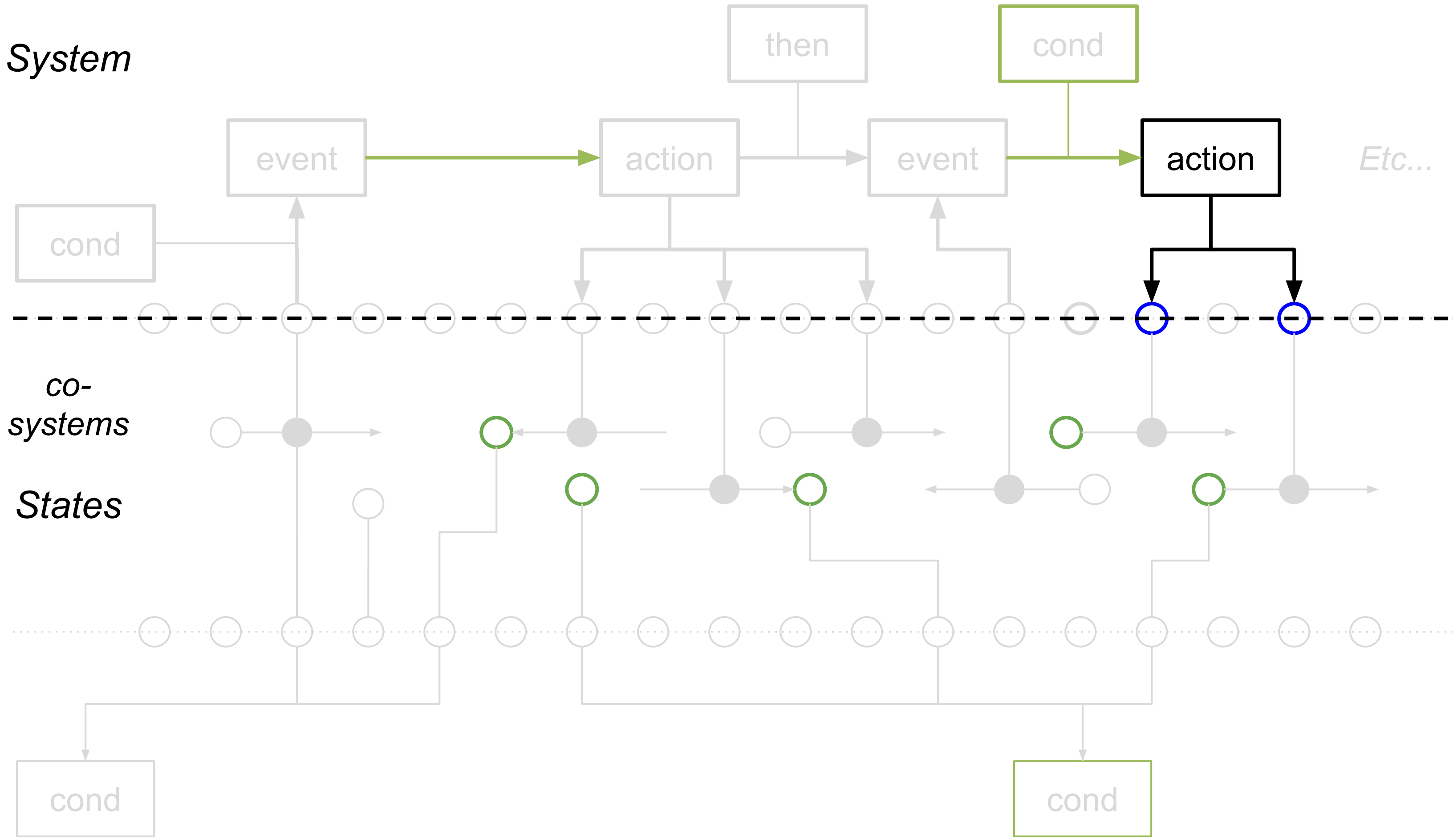
Event is notified



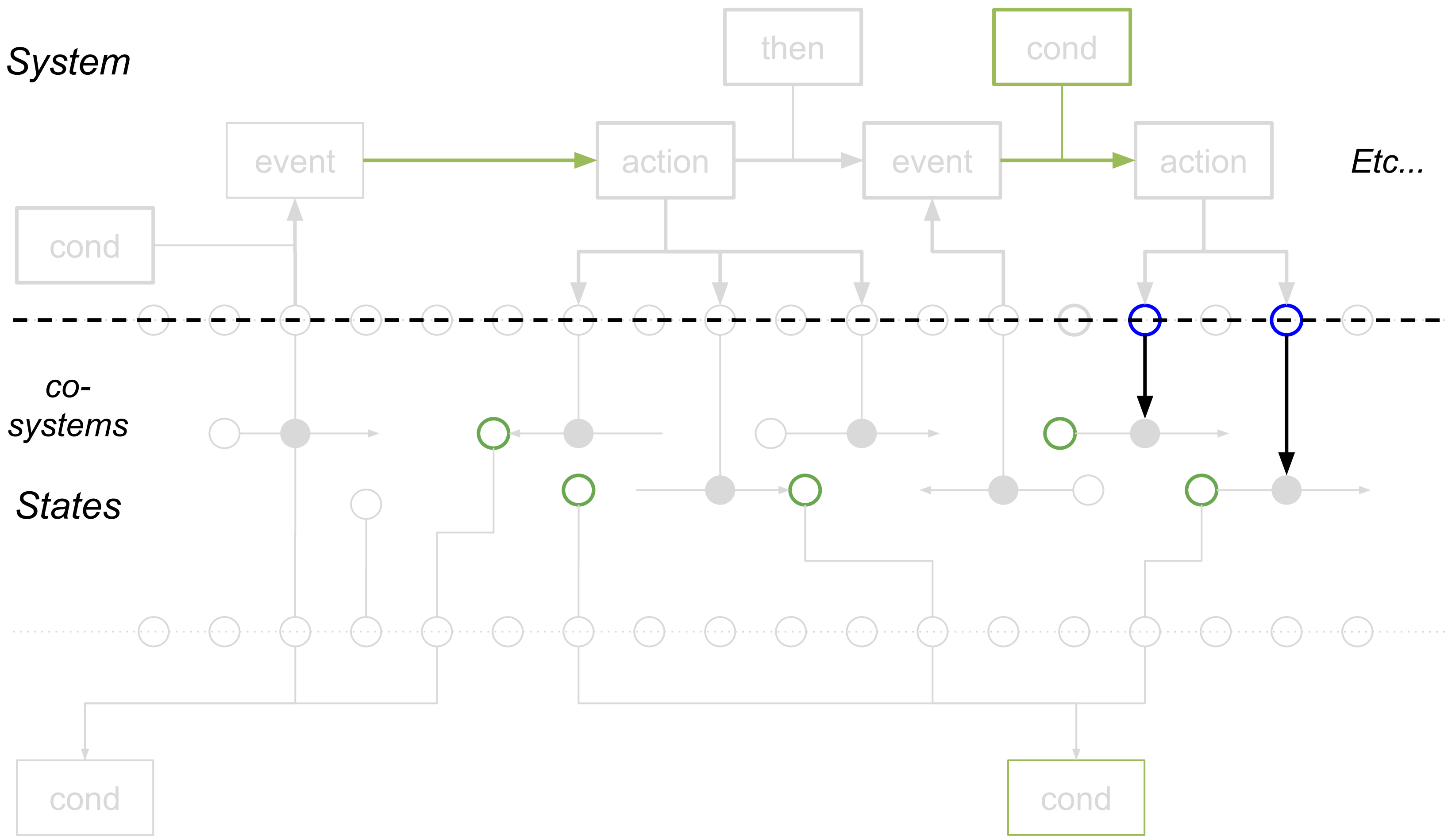
Event triggers Action



Action executes



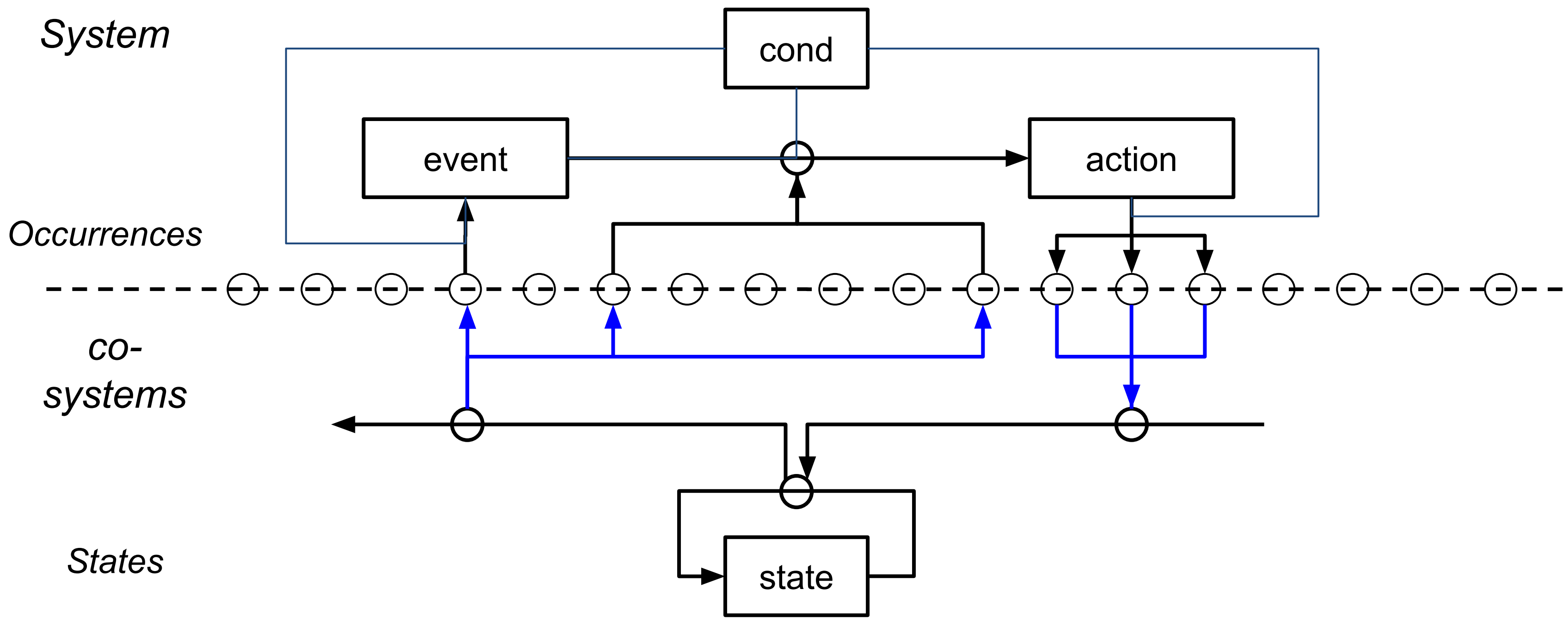
Etc.



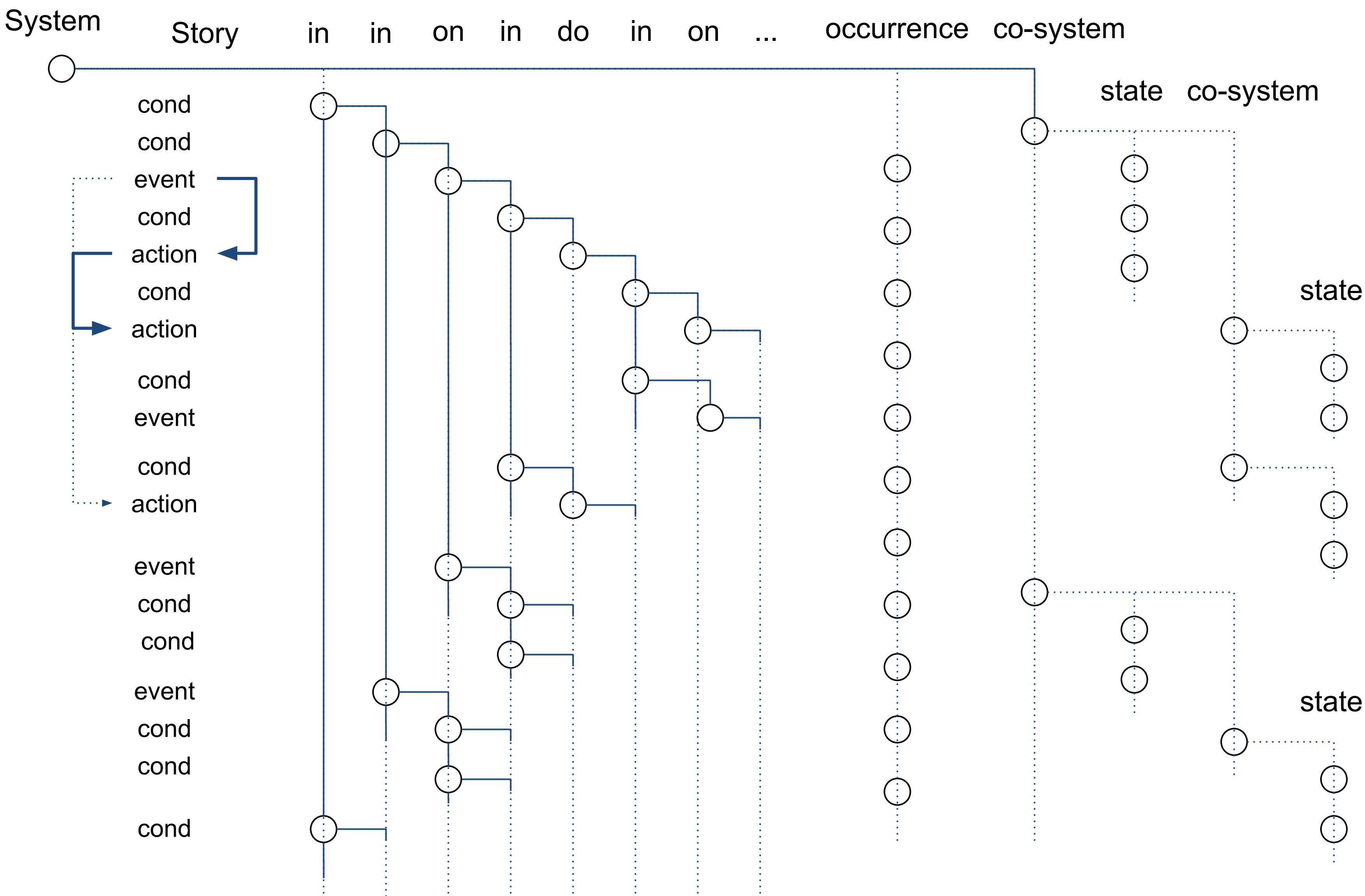
System
then
cond
event
action
event
action
Etc...
cond
co-
systems
States
cond
cond

Other Views

Top Down vs. Inside Out



In the text[1]



1. A Story is a *graph* of Occurrences

Thank You!

Annexes

Permissible Occurrence Expressions

expression		**event**	**cond**	**action**
[string of characters]		X	X	X
system[1] : state[2]			X	
system[1] : ->state[2]	system[1] : state[2]->	X		(X)[3]
system[1] : *function* (*arg*, ...)				X
: (*boolean expr.*)[1]			X	
Interface with programming language				
(*boolean expr.*)			X	
->(*boolean expr.*)	(*boolean expr.*)->	X		X
{ *programming block* }				X

1. system can be a co-system or nothing — meaning: *this* system
2. including permissible condition expressions
3. in *action* — co-system:->state denotes a remote procedure call. co-system:state-> in this context could denote a remote procedure kill.

Permissible Event Expressions

Notation	Description
system: state	a state of the system
system: state->	the event corresponding to the system leaving given state
system: ->state	the event corresponding to the system entering given state
Extension[1]	
system: A->B	the event corresponding to the system having completed its transition from state A to state B
system: [A-> B]	the state corresponding to the system transitioning from state A to state B

Note

- This notation is extremely powerful and requires careful consideration for proper usage. For instance, in the eventuality of the intermediate state *[A-> B]* being part of the *story*, then the event *->[A->B]* occurs *before* the system enters the intermediate state *[A->B]*, but after it has left state A. This allows supporting what is known to software developers as pre- and post- callbacks.