Behavioral design with Alloy

Nuno Macedo
(original slides by Alcino Cunha)

Software *concepts*

THE ESSENCE OF SOFTWARE

WHY CONCEPTS MATTER FOR GREAT DESIGN

DANIEL JACKSON

Concepts

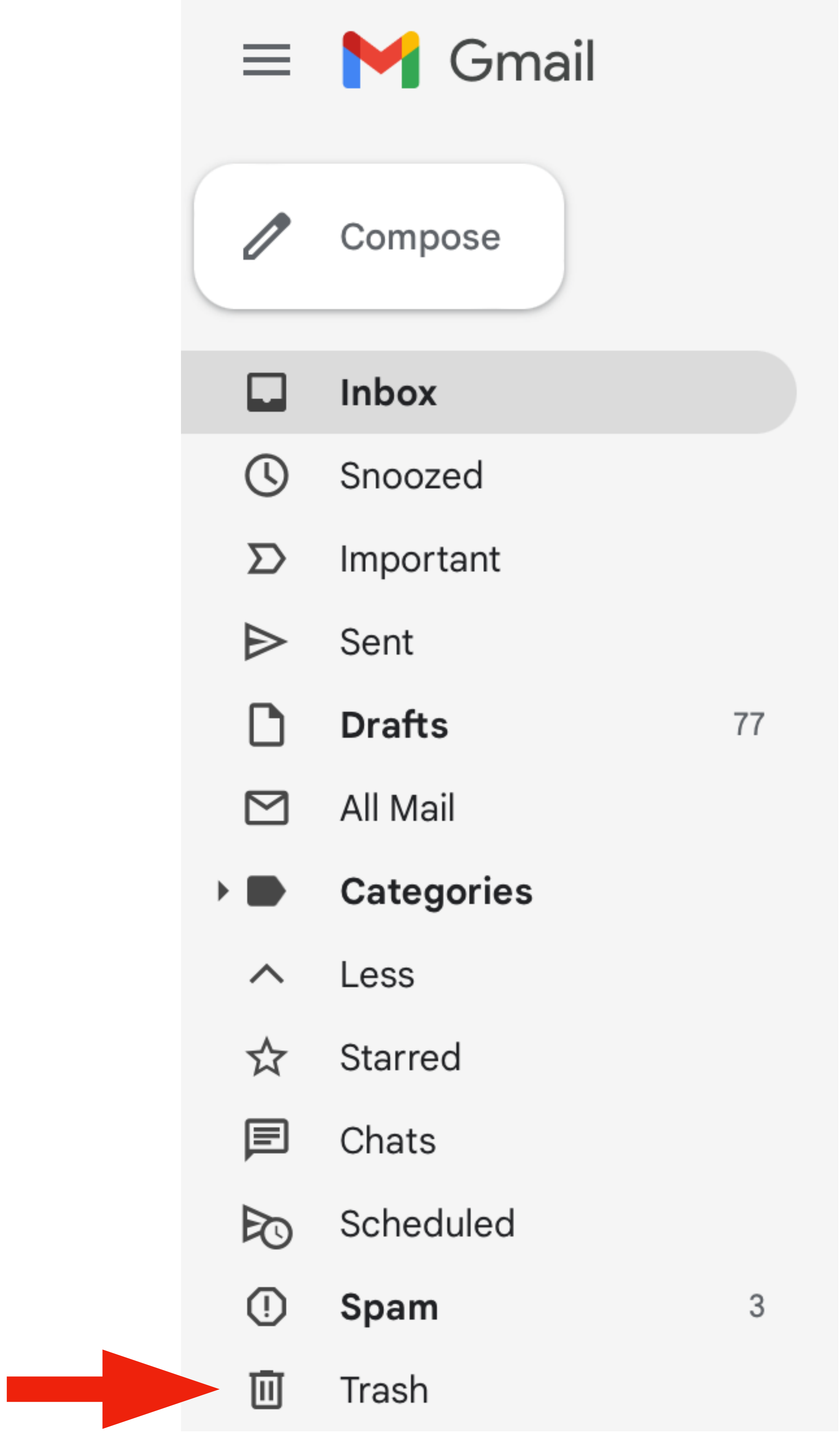
- Apps are made of recurring *concepts*
- Each *concept* is a self-contained unit of functionality with a clear purpose
- *Concepts* work together to provide the app overall functionality
- But can be understood independently of one another

Trash

Trash

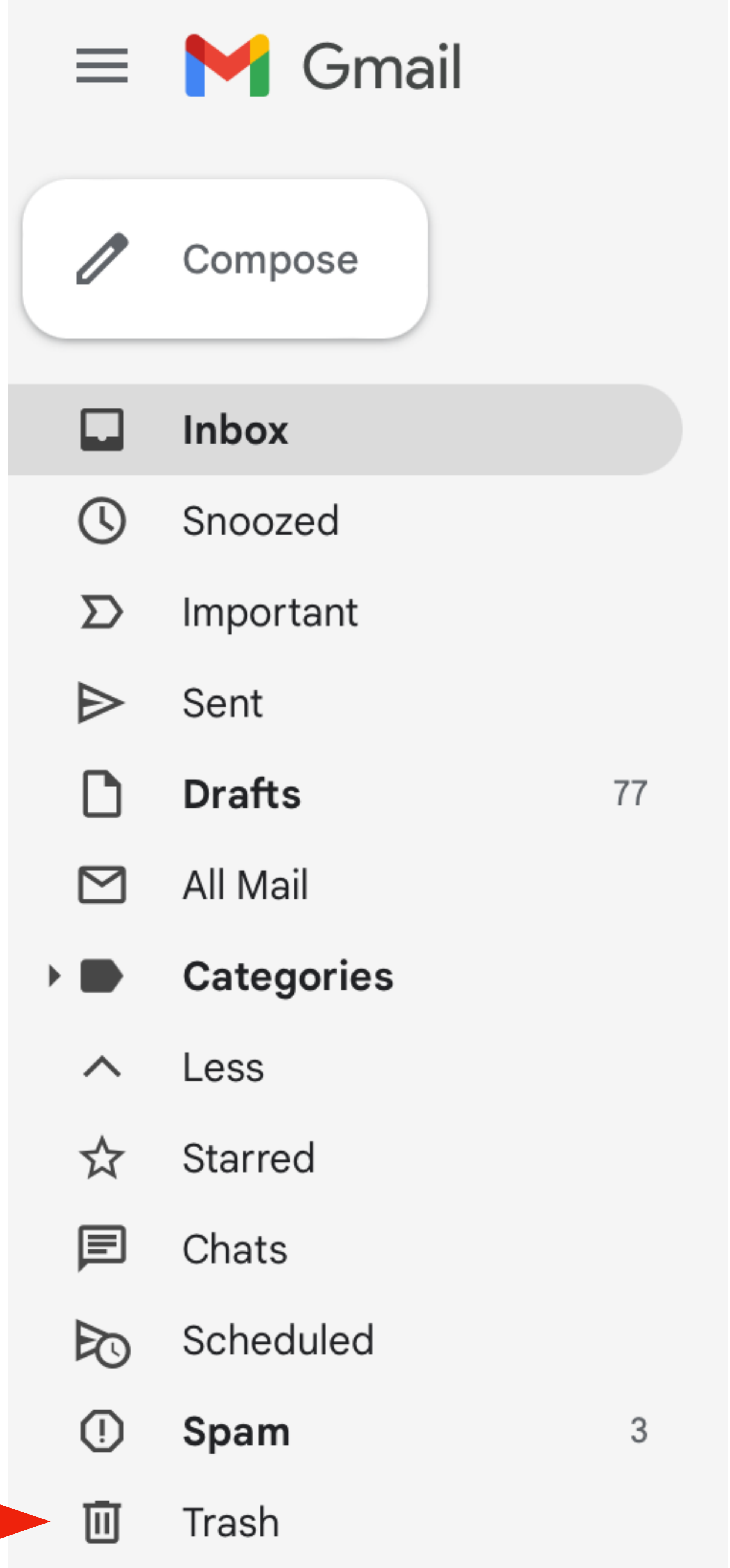
Trash

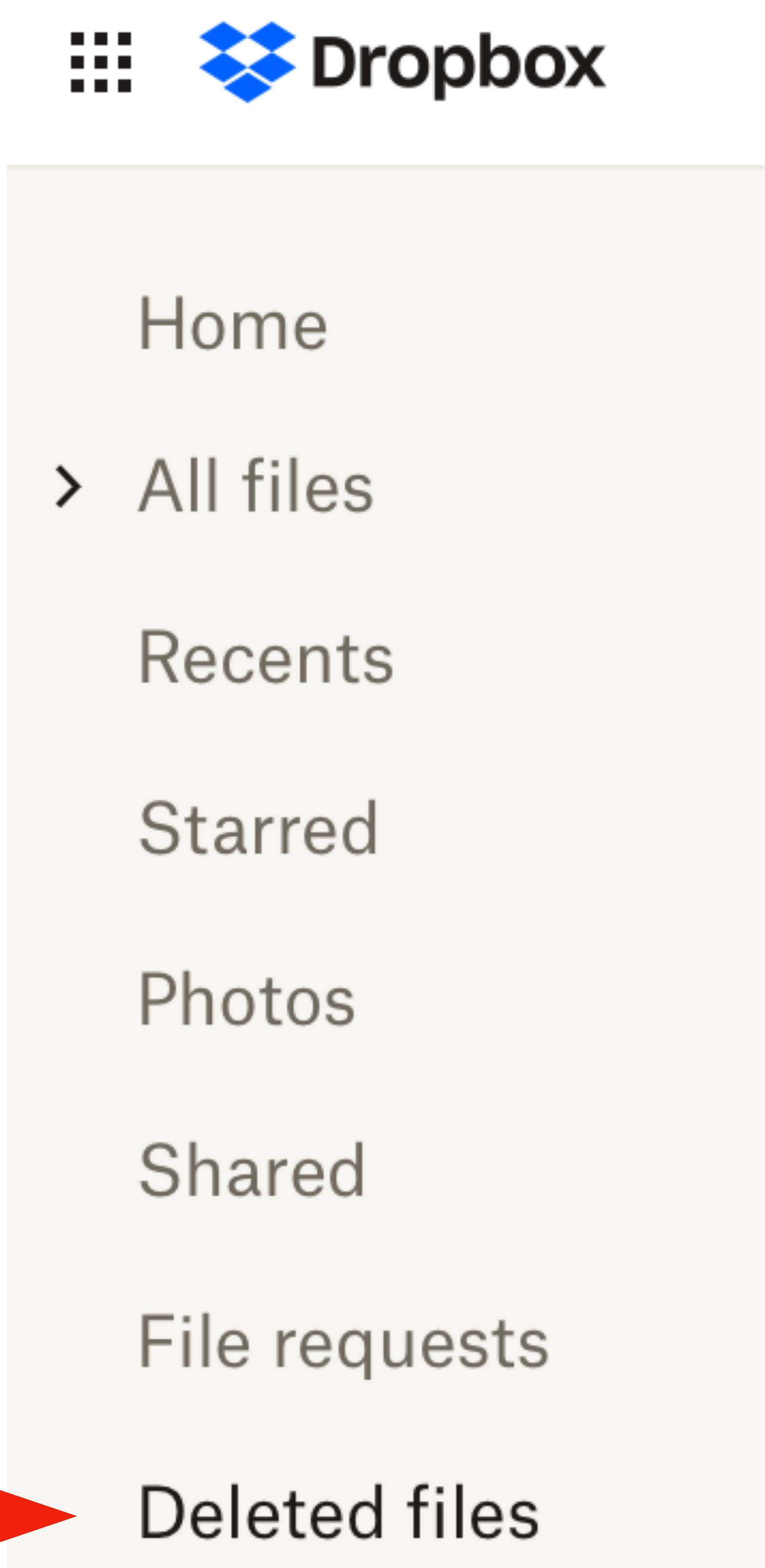

Open
Empty Trash


Gmail
Compose
Inbox
Snoozed
Important
Sent
Drafts 77
All Mail
Categories
Less
Starred
Chats
Scheduled
Spam 3
Trash

Trash

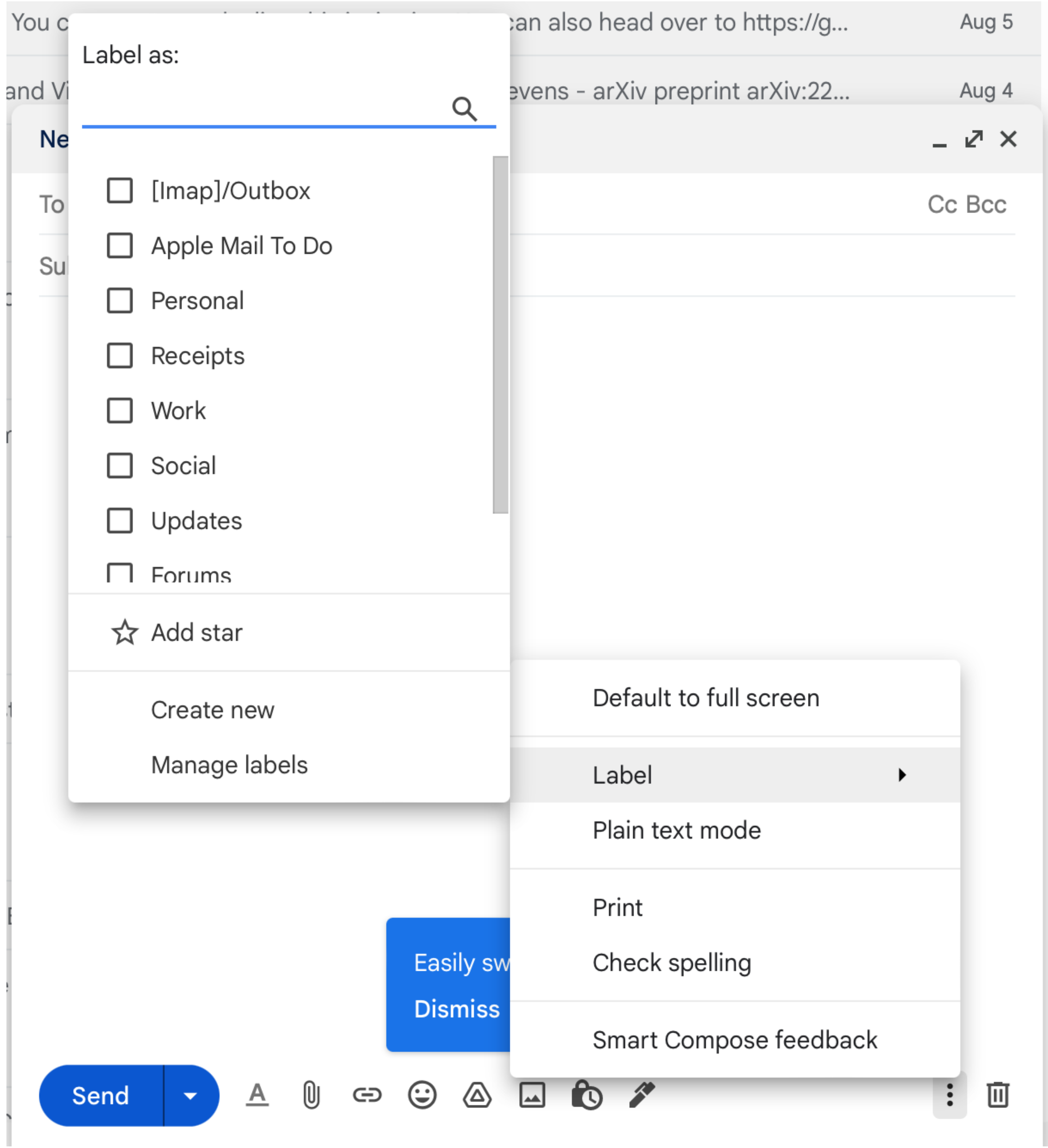

Open
Empty Trash


Gmail
Compose
Inbox
Snoozed
Important
Sent
Drafts 77
All Mail
Categories
Less
Starred
Chats
Scheduled
Spam 3
Trash

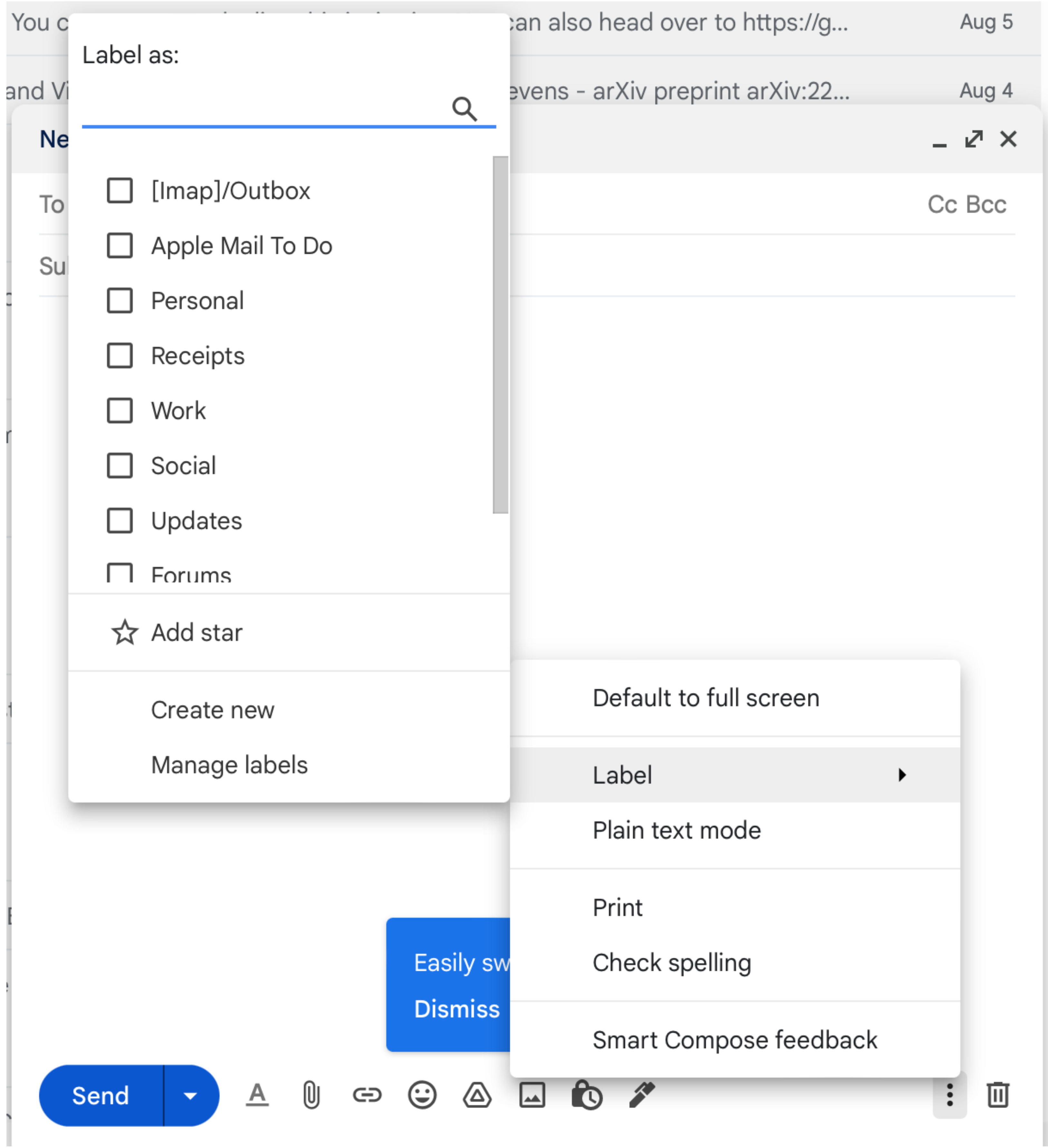

Dropbox
Home
All files
Recents
Starred
Photos
Shared
File requests
Deleted files

Label

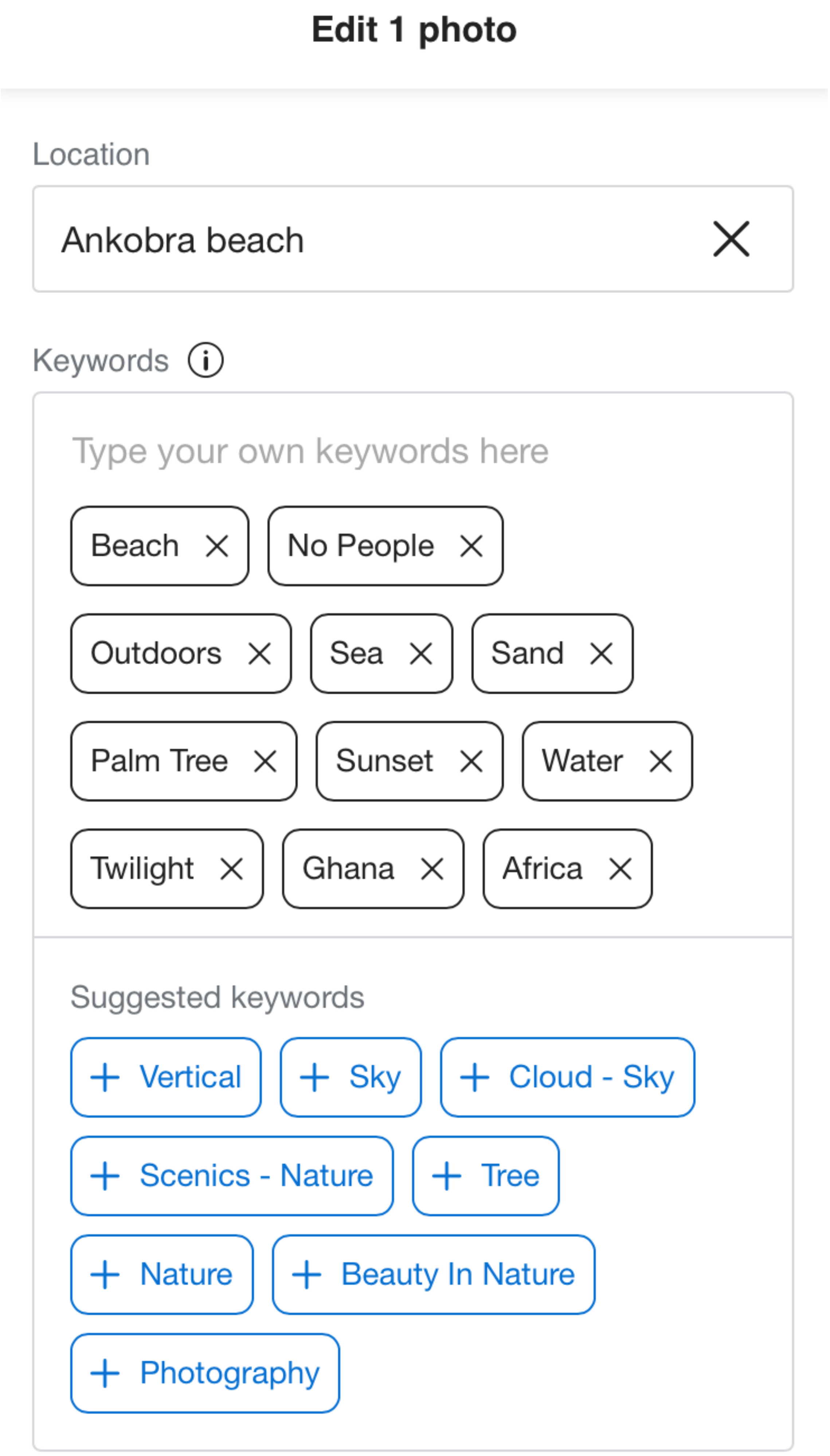
Label


You c
can also head over to https://g...
Aug 5
and Vi
evens - arXiv preprint arXiv:22...
Aug 4
Label as:
Ne
To
Cc Bcc
Su
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Easily sw
Dismiss
Send

Label

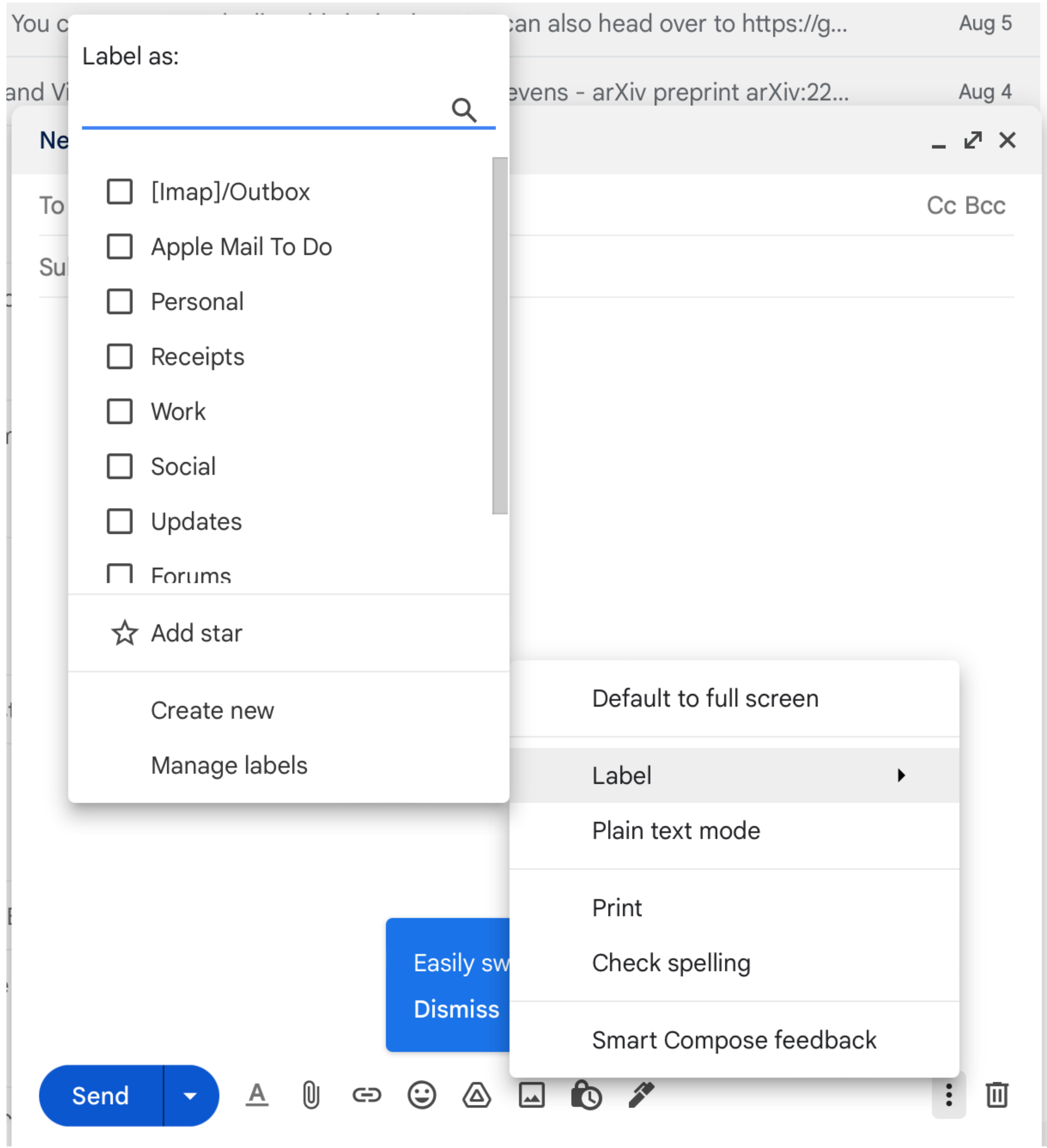


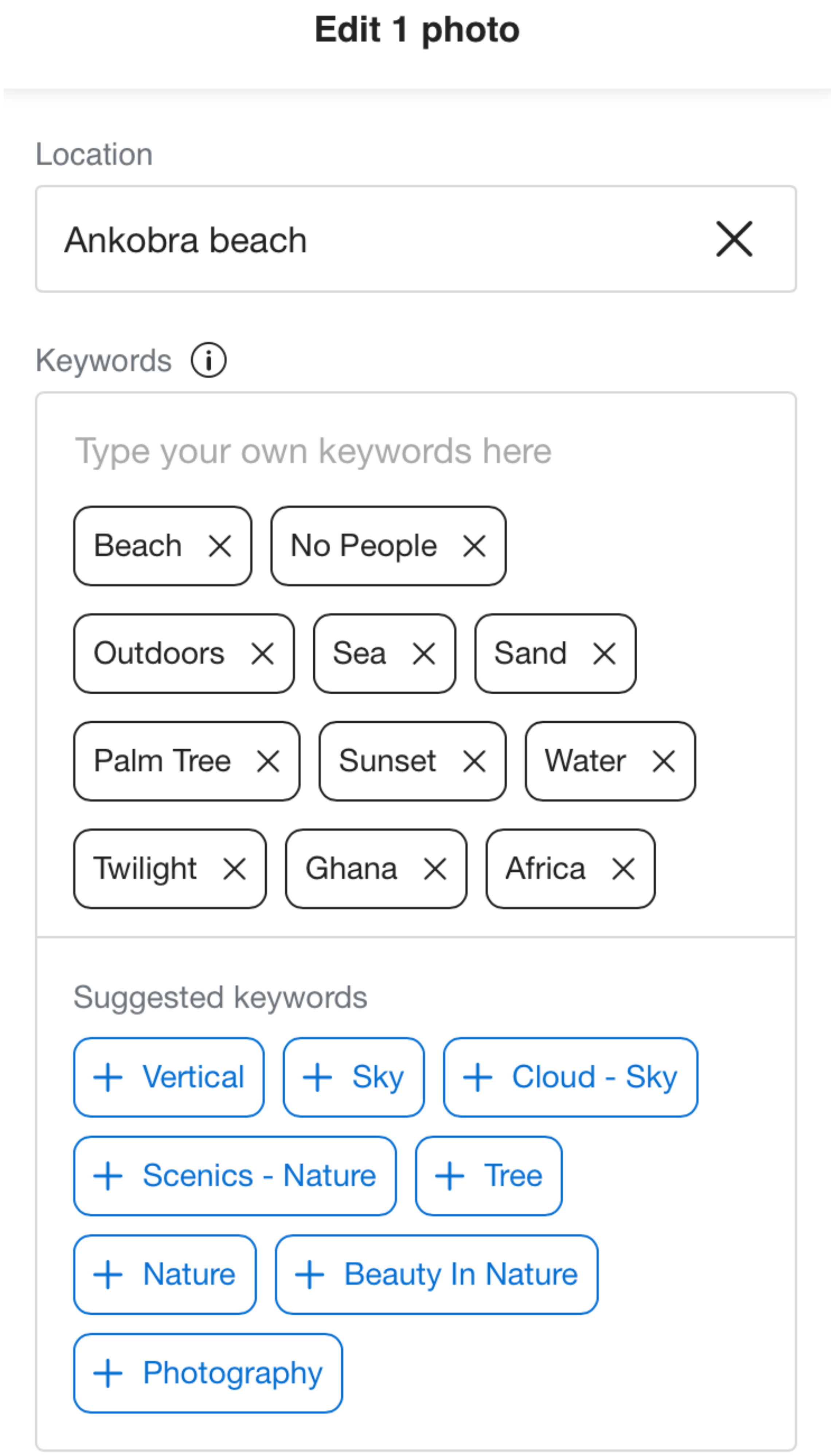
Label as:
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Easily sw
Dismiss
Send
Cc Bcc
Aug 5
Aug 4

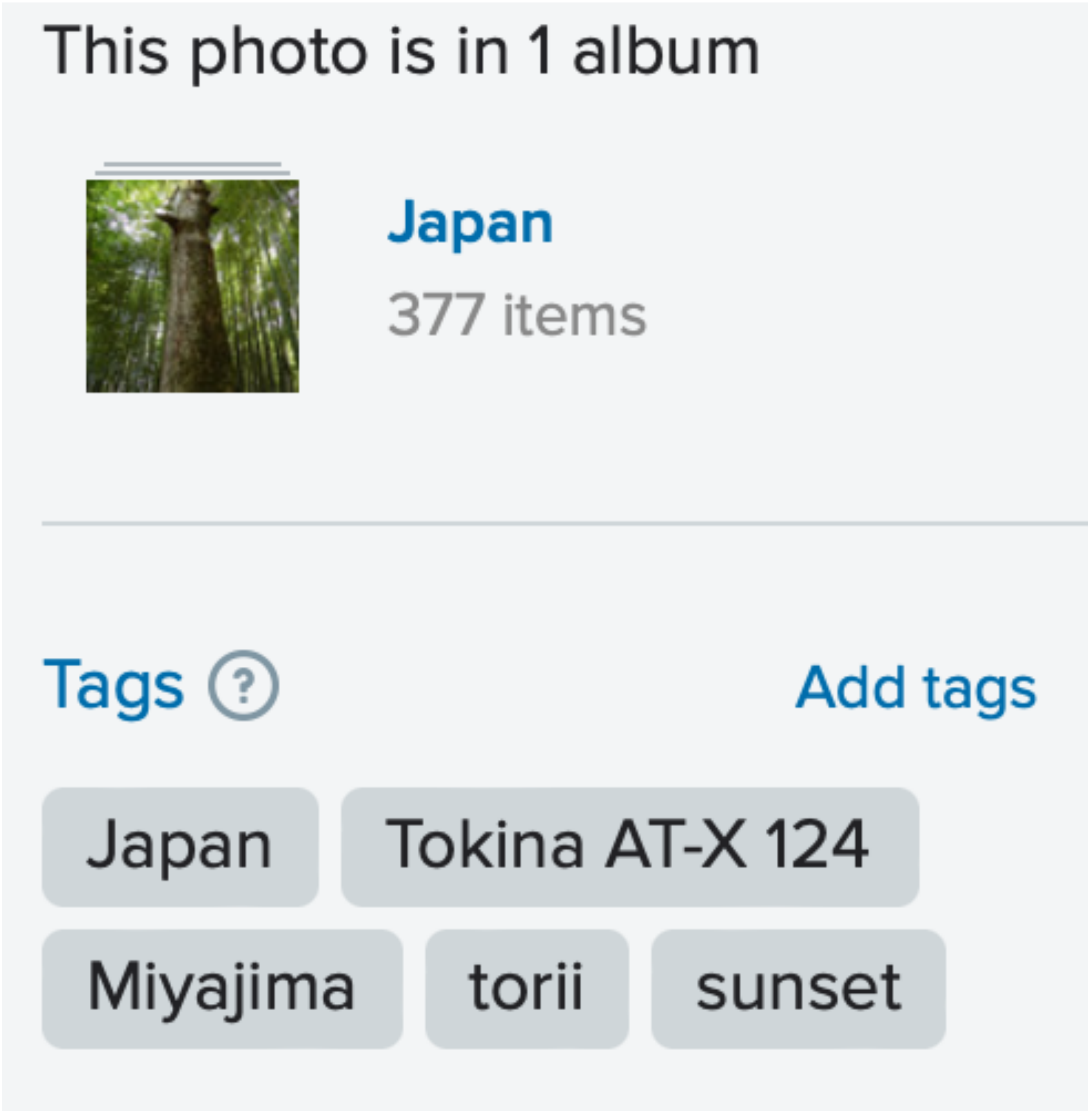


Edit 1 photo
Location
Ankobra beach
Keywords
Type your own keywords here
Beach
No People
Outdoors
Sea
Sand
Palm Tree
Sunset
Water
Twilight
Ghana
Africa
Suggested keywords
Vertical
Sky
Cloud - Sky
Scenics - Nature
Tree
Nature
Beauty In Nature
Photography

Label

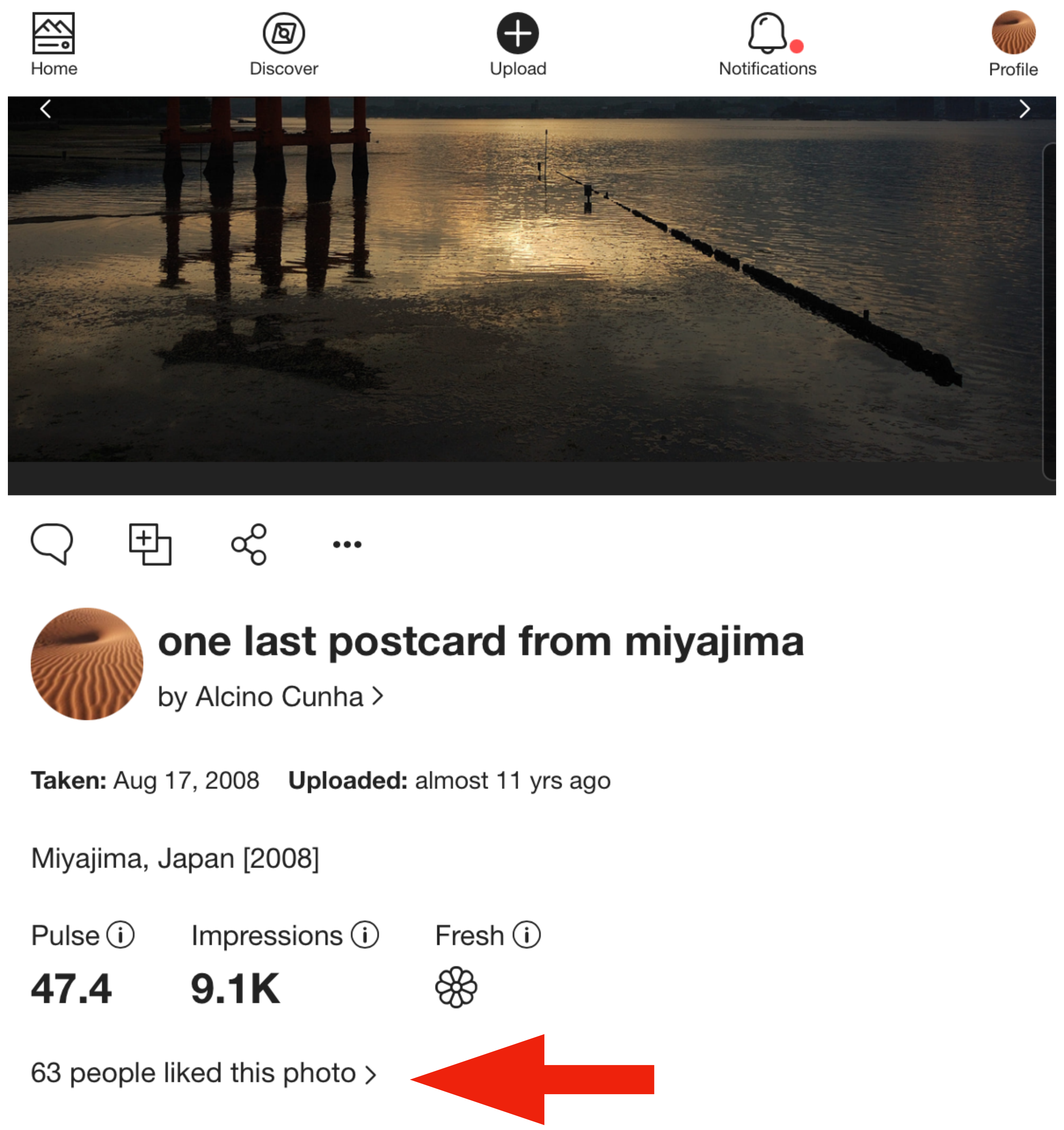

Label as:
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Send
Dismiss
Cc Bcc
Aug 5
Aug 4


Edit 1 photo
Location
Ankobra beach
Keywords
Type your own keywords here
Beach
No People
Outdoors
Sea
Sand
Palm Tree
Sunset
Water
Twilight
Ghana
Africa
Suggested keywords
Vertical
Sky
Cloud - Sky
Scenics - Nature
Tree
Nature
Beauty In Nature
Photography

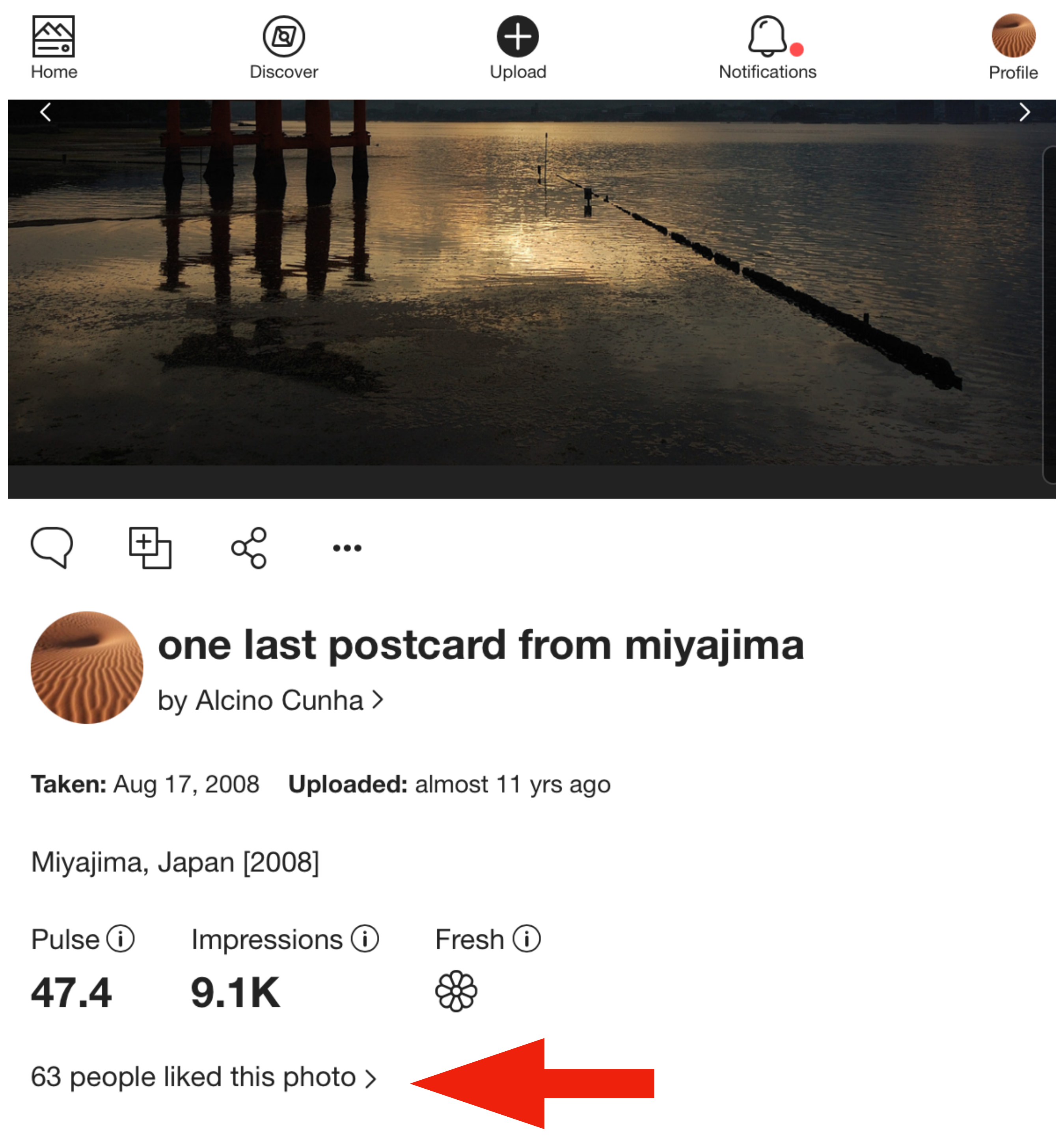

This photo is in 1 album
Japan
377 items
Tags
Add tags
Japan
Tokina AT-X 124
Miyajima
torii
sunset

Like

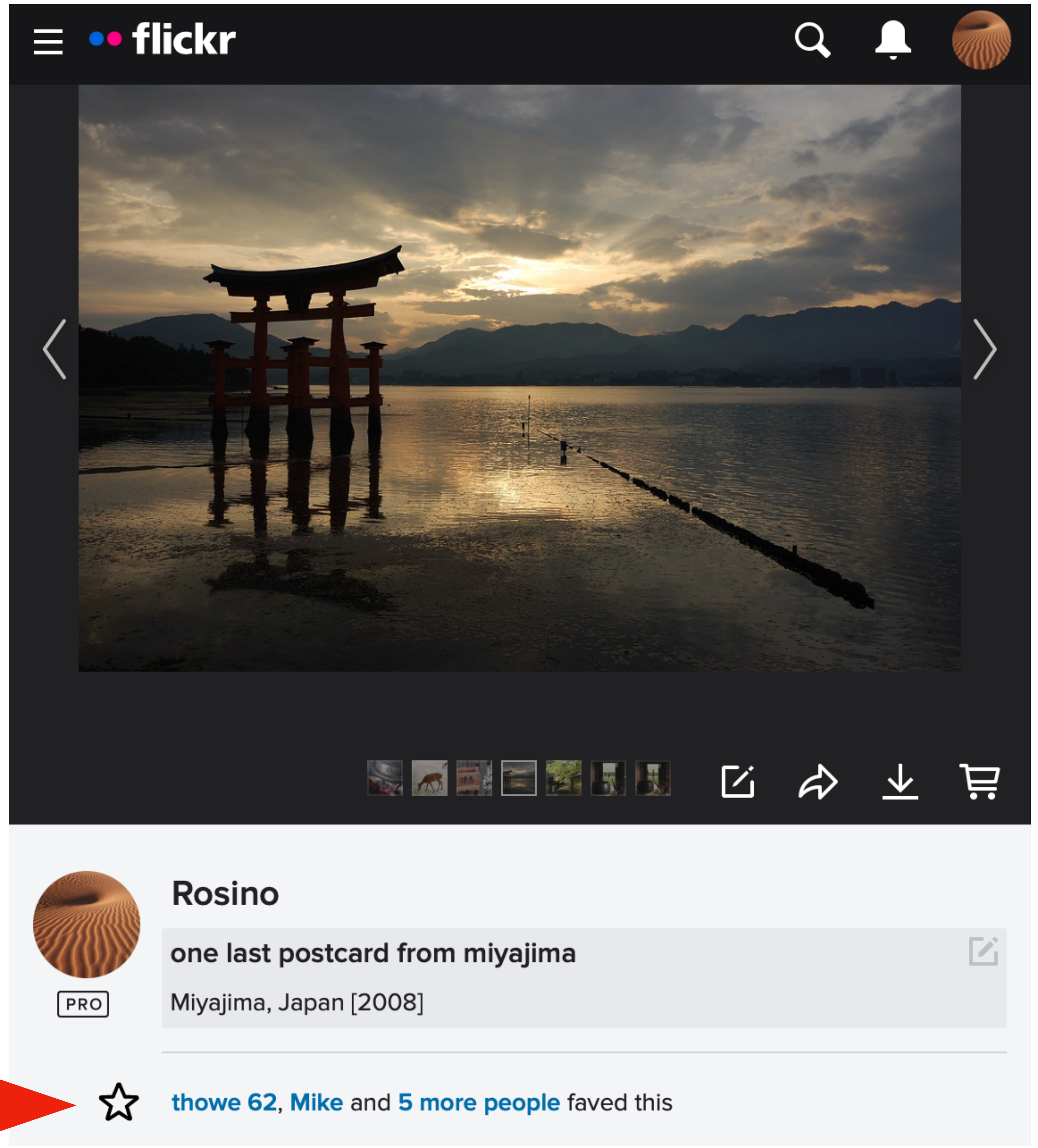
Like


Home
Discover
Upload
Notifications
Profile
one last postcard from miyajima
by Alcino Cunha
Taken: Aug 17, 2008 **Uploaded:** almost 11 yrs ago
Miyajima, Japan [2008]
Pulse
47.4
Impressions
9.1K
Fresh
63 people liked this photo

Like

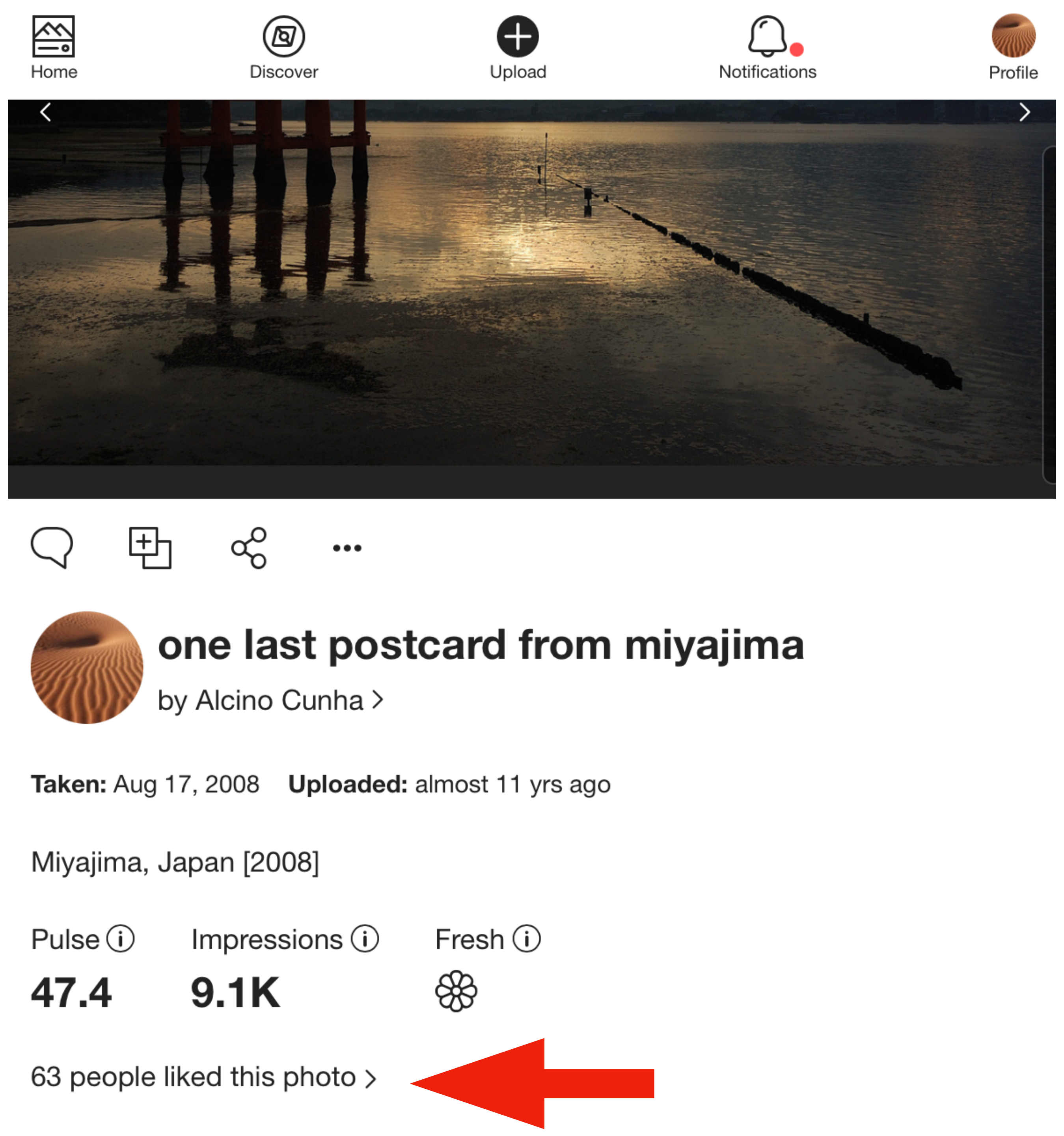


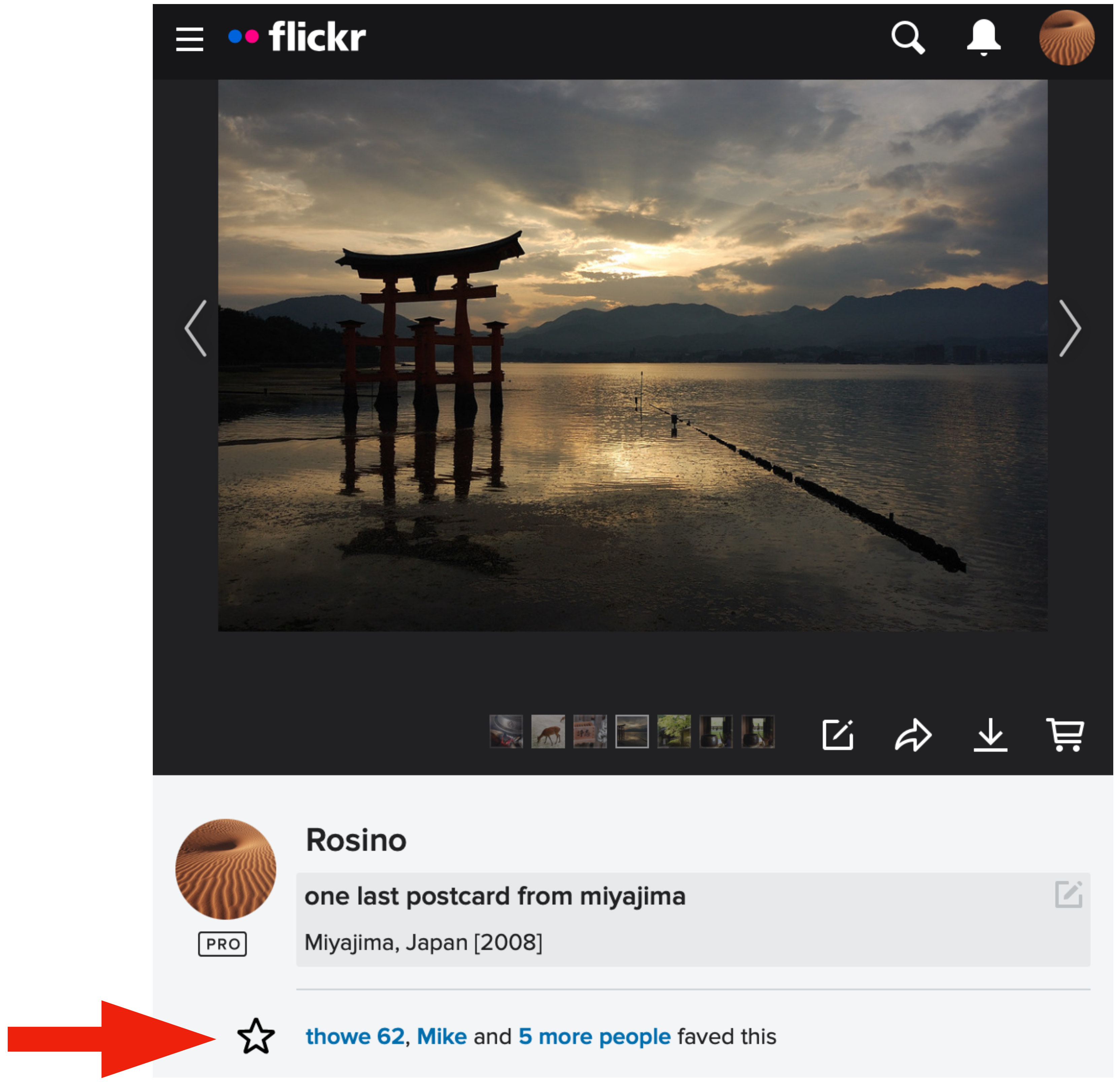
Home
Discover
Upload
Notifications
Profile
one last postcard from miyajima
by Alcino Cunha
Taken: Aug 17, 2008 Uploaded: almost 11 yrs ago
Miyajima, Japan [2008]
Pulse
47.4
Impressions
9.1K
Fresh
63 people liked this photo

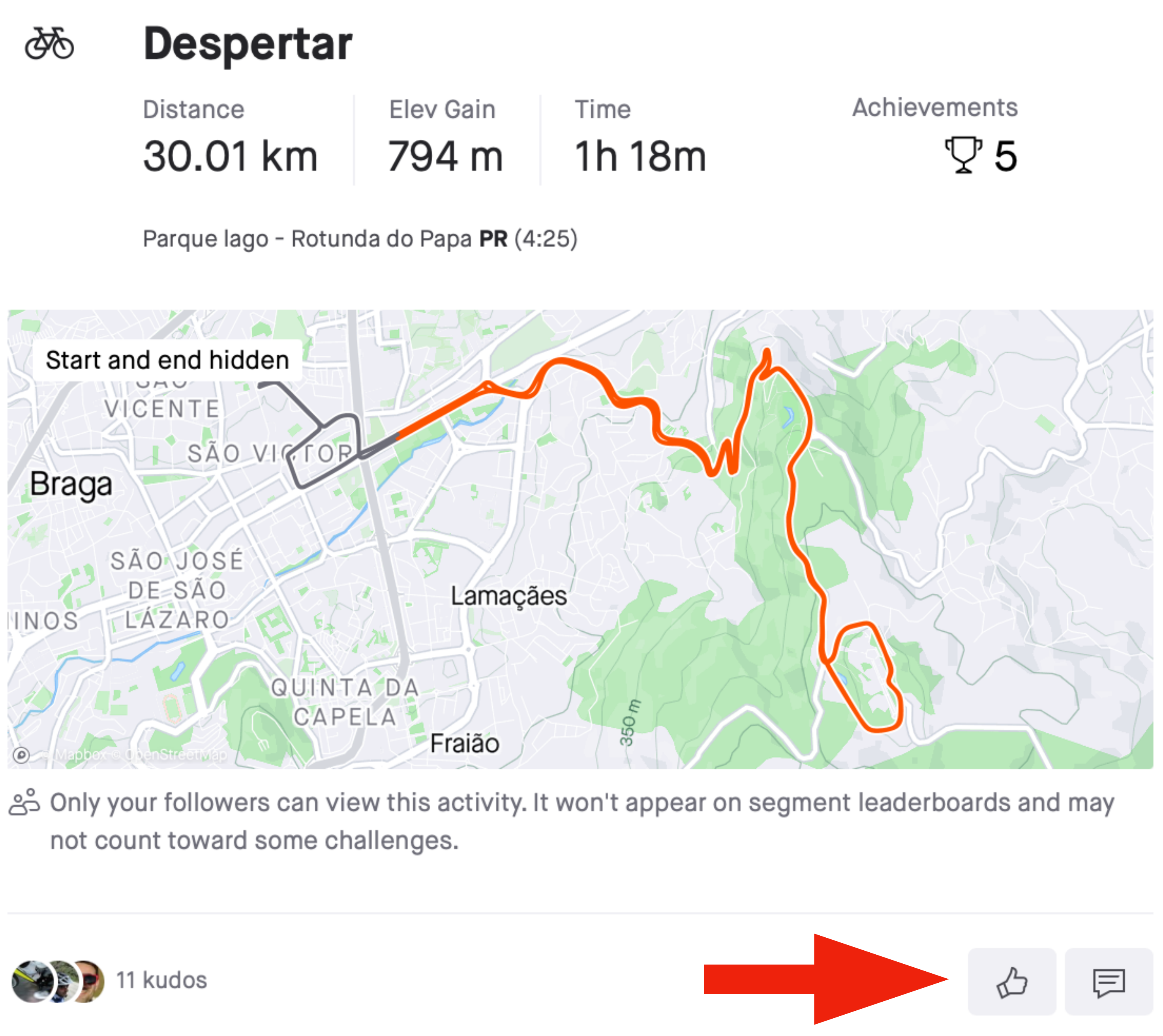


flickr
Rosino
PRO
one last postcard from miyajima
Miyajima, Japan [2008]
thowe 62, Mike and 5 more people faved this

Like


Home
Discover
Upload
Notifications
Profile
one last postcard from miyajima
by Alcino Cunha >
Taken: Aug 17, 2008 Uploaded: almost 11 yrs ago
Miyajima, Japan [2008]
Pulse
Impressions
Fresh
47.4
9.1K
63 people liked this photo >


flickr
Rosino
PRO
one last postcard from miyajima
Miyajima, Japan [2008]
thowe 62, Mike and 5 more people faved this


Despertar
Distance
30.01 km
Elev Gain
794 m
Time
1h 18m
Achievements
5
Parque lago - Rotunda do Papa PR (4:25)
Start and end hidden
Braga
Lamaçães
Fraião
Only your followers can view this activity. It won't appear on segment leaderboards and may not count toward some challenges.
11 kudos

Concept design

- Identify a clear purpose
- Choose the appropriate *state* and *actions* to fulfill that purpose
- The focus is on ensuring correctness and reusability

App design

- Identify the core *concepts*
- Compose them, maybe providing new functionality
- The focus is on exploration

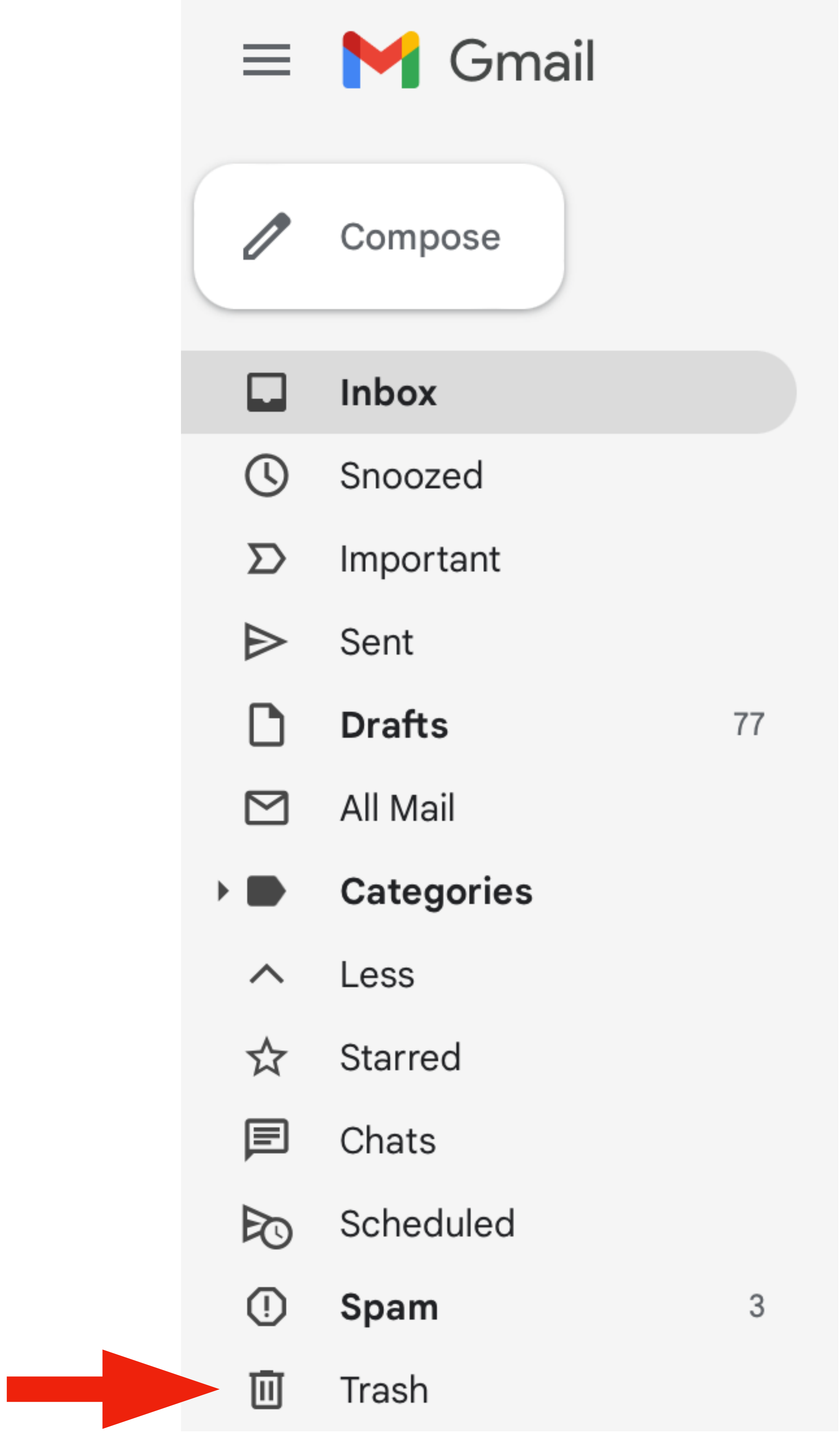
Modeling *concepts*

Trash

Trash

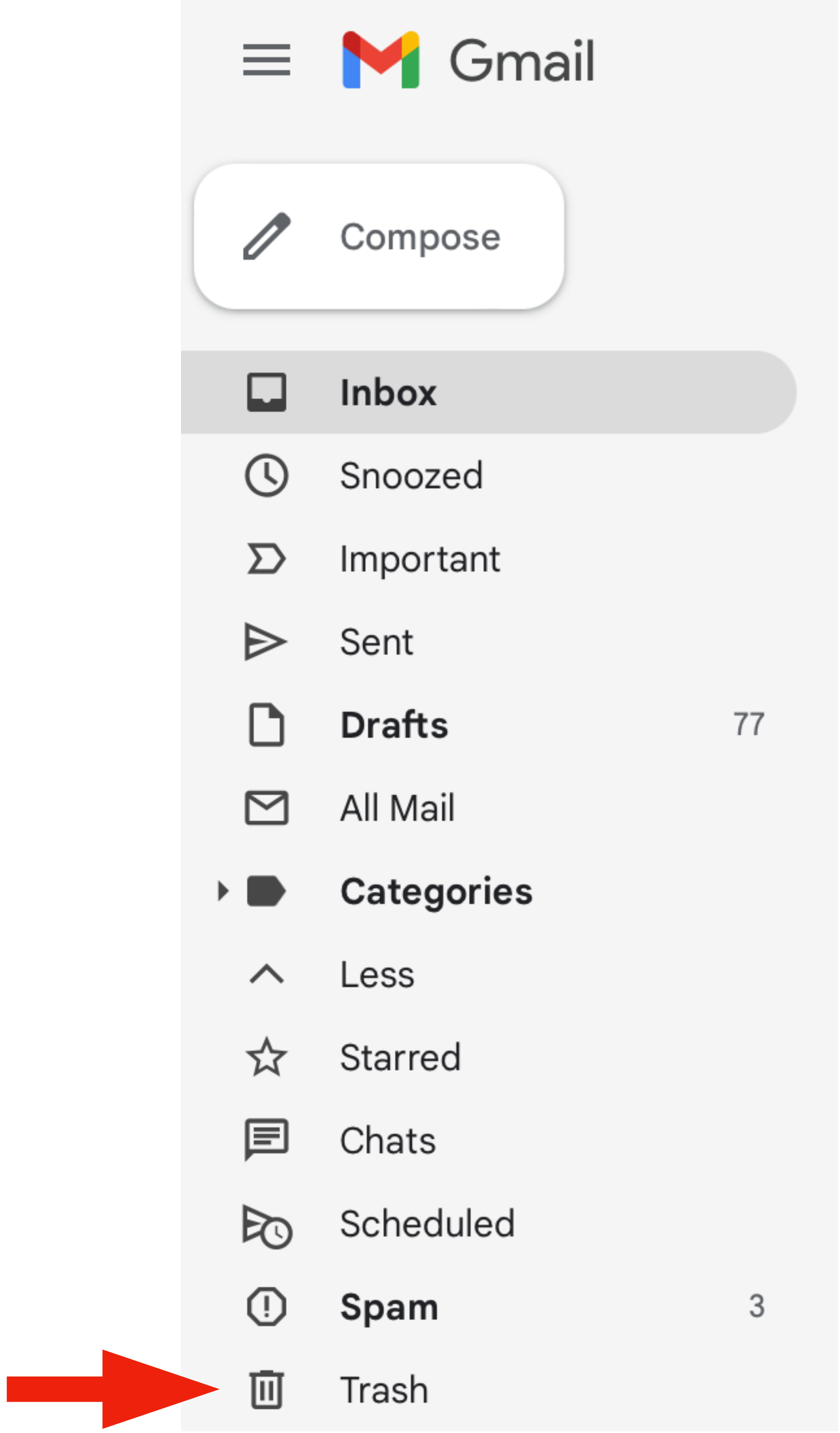
Trash

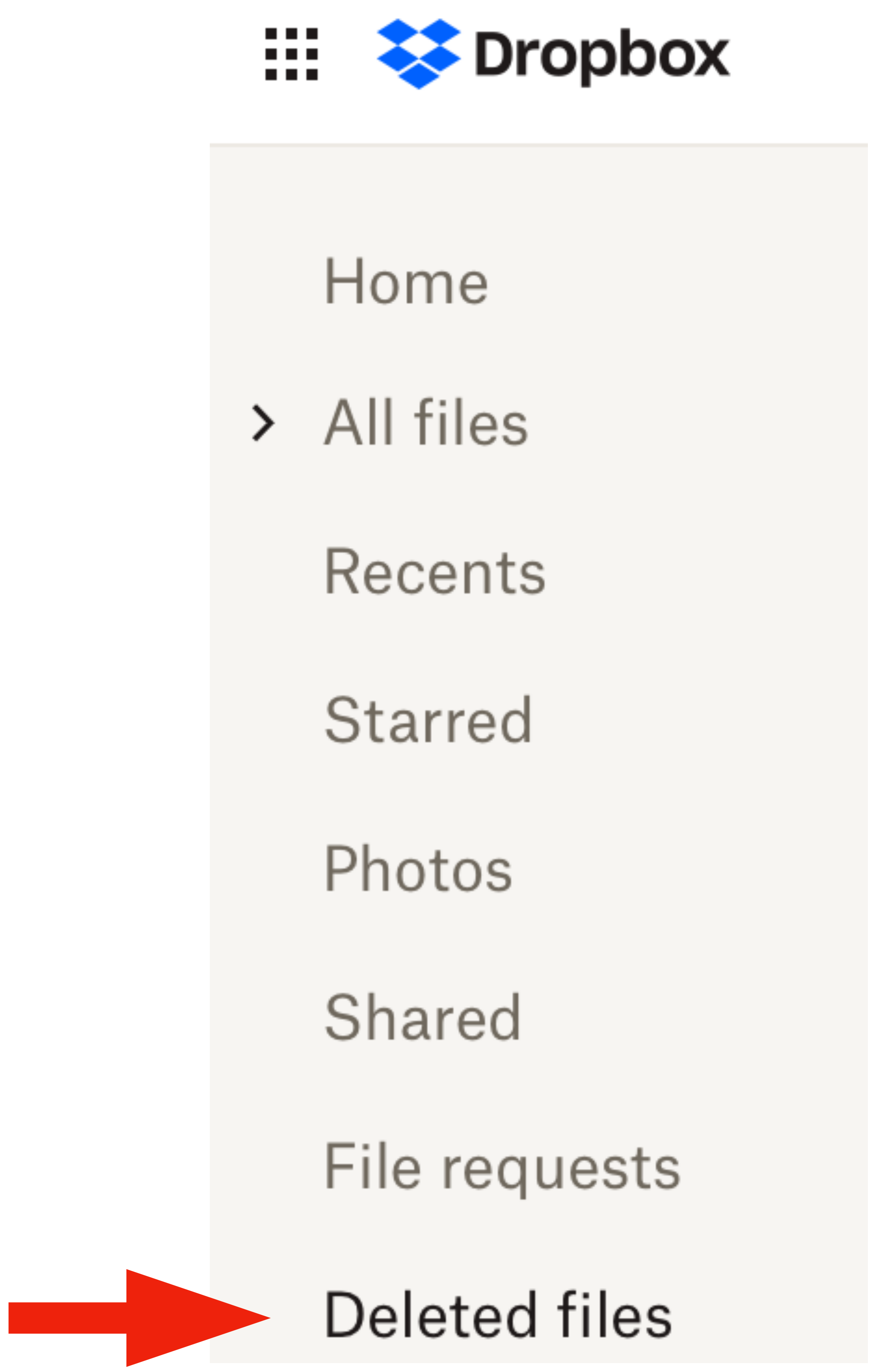

Open
Empty Trash


Gmail
Compose
Inbox
Snoozed
Important
Sent
Drafts 77
All Mail
Categories
Less
Starred
Chats
Scheduled
Spam 3
Trash

Trash

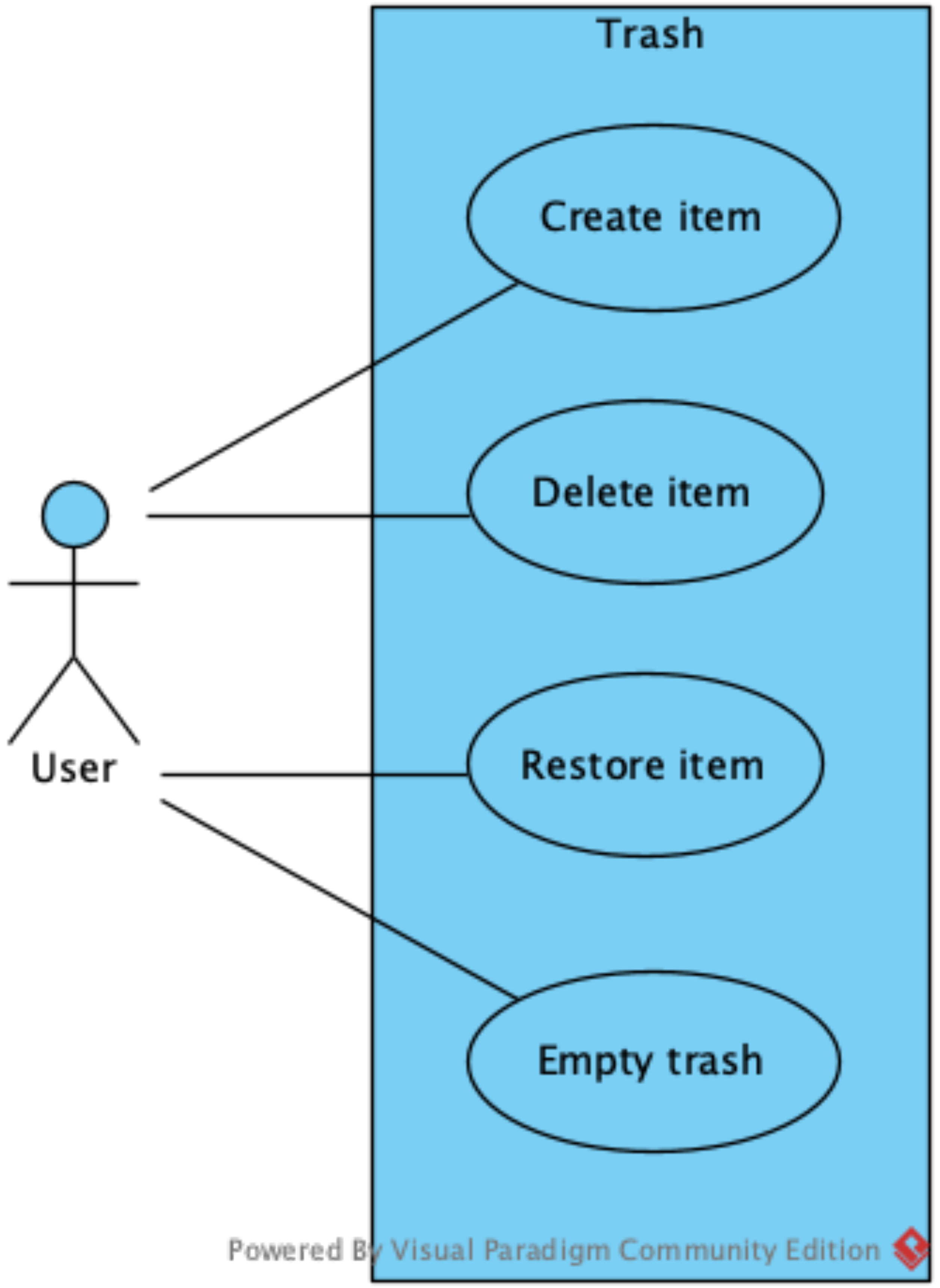

Open
Empty Trash


Gmail
Compose
Inbox
Snoozed
Important
Sent
Drafts
77
All Mail
Categories
Less
Starred
Chats
Scheduled
Spam
3
Trash

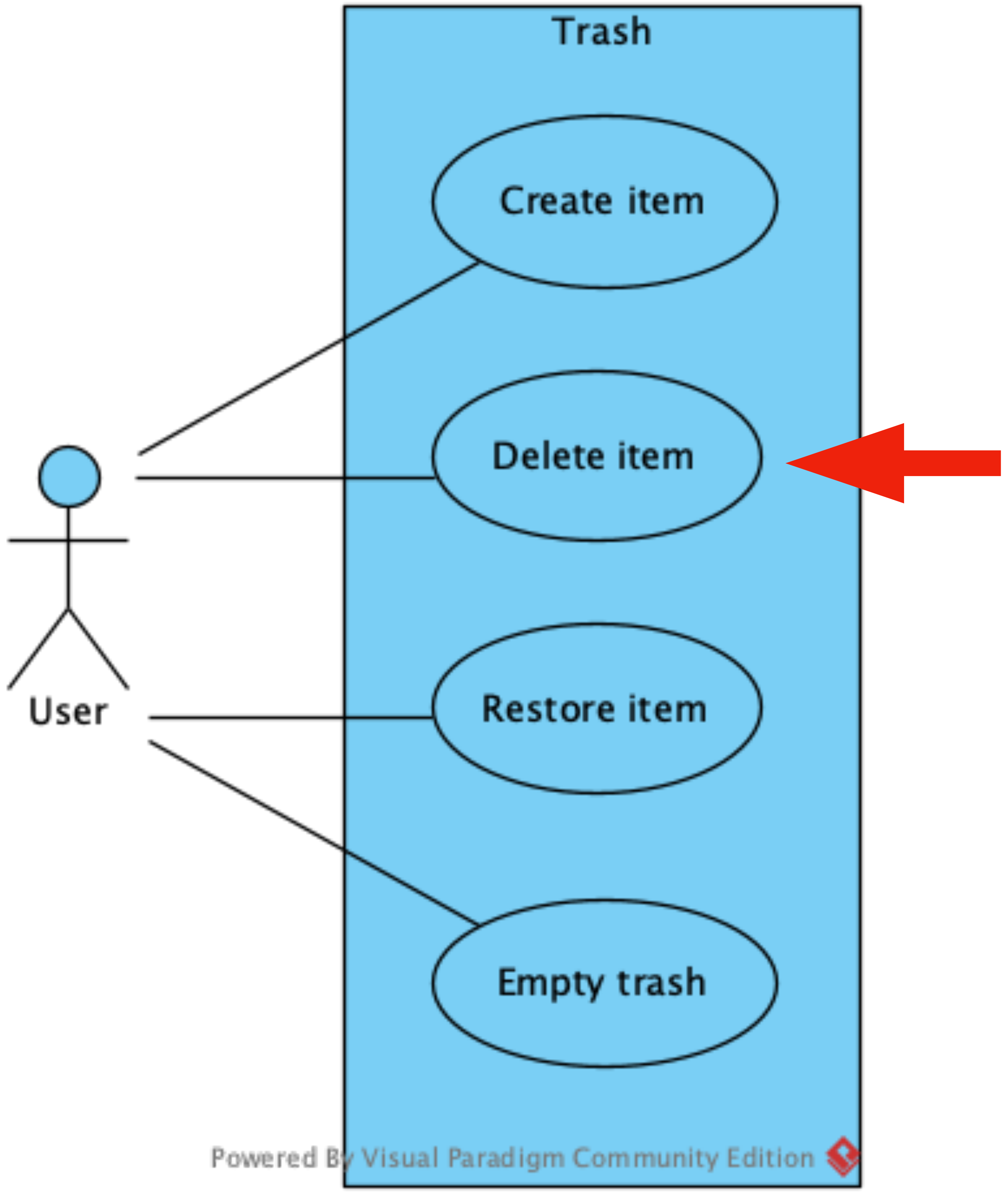

Dropbox
Home
All files
Recents
Starred
Photos
Shared
File requests
Deleted files

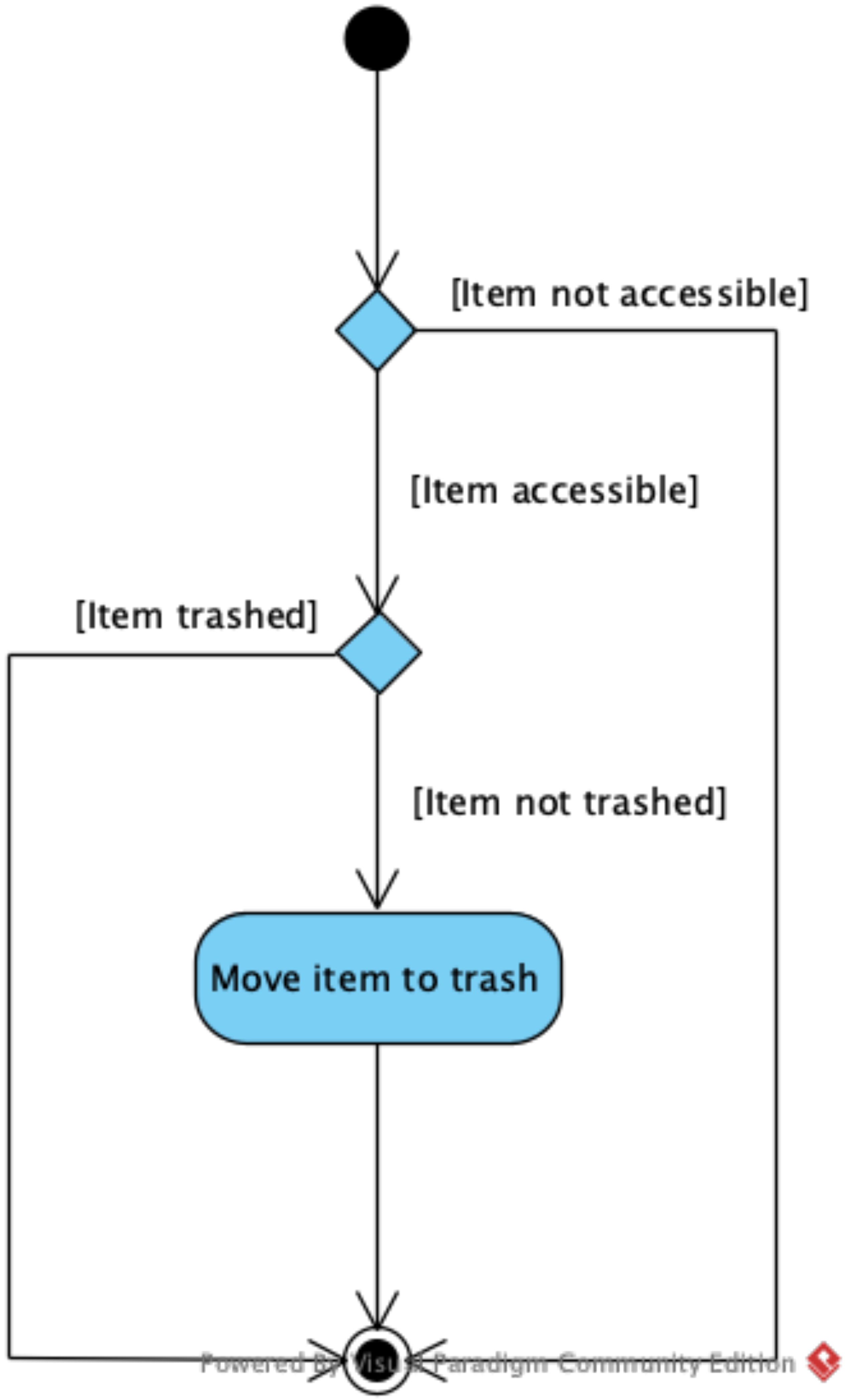
Trash modeling *à la* UML

Trash modeling *à la* UML



Trash modeling *à la* UML





Trash modeling *à la* Jackson

```
concept trash [Item]
purpose
  to allow undoing of deletions
state
  accessible, trashed : set Item
actions
  create (x : Item)
    when x not in accessible or trashed
    add x to accessible
  delete (x : Item)
    when x in accessible but not trashed
    move x from accessible to trashed
  restore (x : Item)
    when x in trashed
    move x from trashed to accessible
  empty ()
    when some item in trashed
    remove every item from trashed
operational principle
  after delete(x), can restore(x) and then x in accessible
  after delete(x), can empty() and then x not in accessible or trashed
```

Concept modeling *à la* Jackson

- Name
  - Optionally parametrized by types that can be specialized when composing
- Purpose
  - A clear reason why you might want it
- State + Actions
  - A description of the concept behavior using a *transition system*
- Operational principle
  - Properties that show how the purpose is fulfilled by the actions

Transition systems

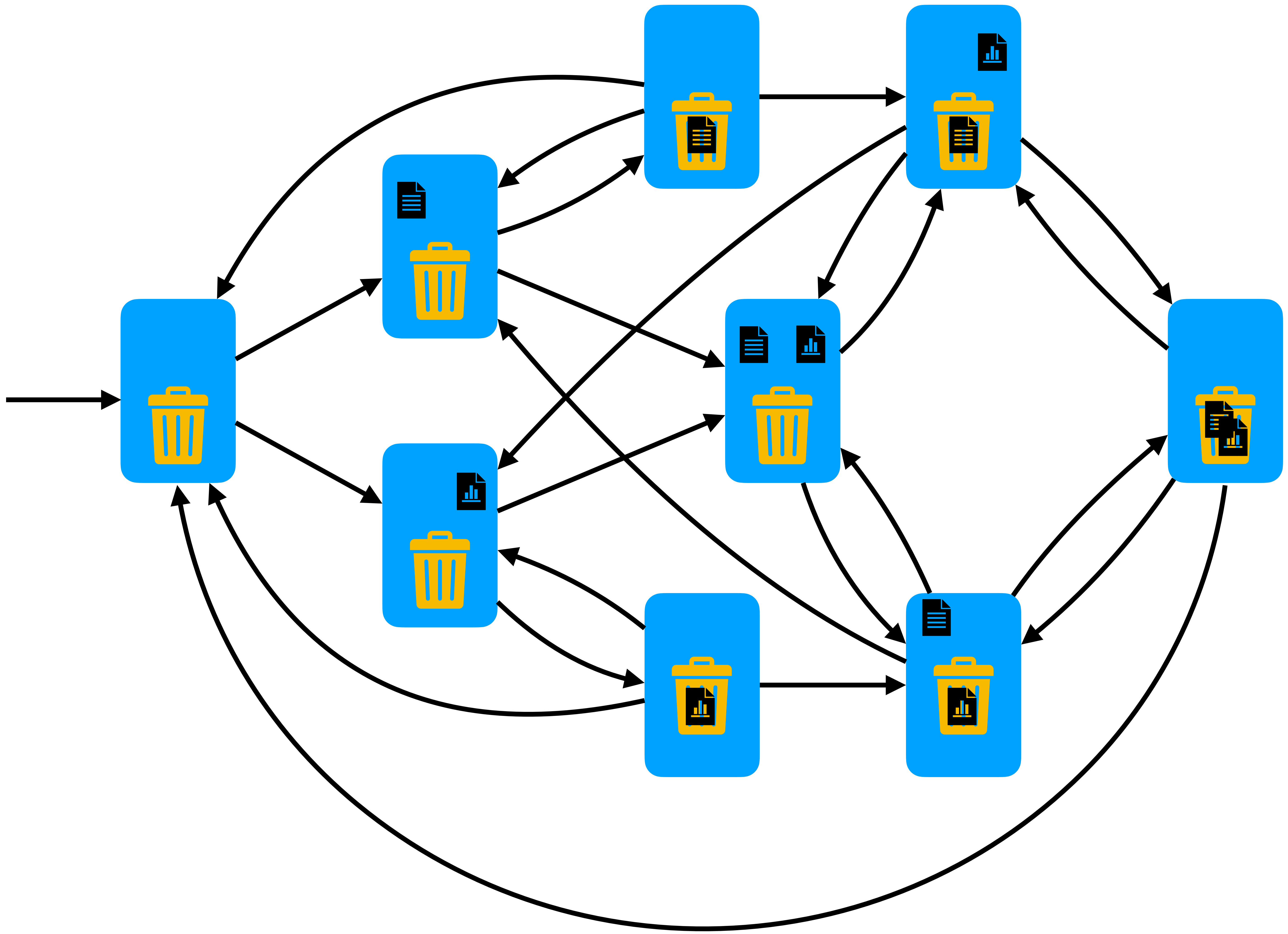
Transition systems

- A popular model to describe the behavior of a system
- A *model* is often a synonym for a transition system
- There are many variants and related formalisms
  - Labeled transition systems
  - Kripke structures
  - Finite state machines
  - Hybrid and timed automata
  - ...

States, transitions, and traces

- States
  - A *state* is a possible valuation to the structures of the system
  - *Initial* states describe how the system starts
- Transitions
  - A *transition* is a possible evolution between states
  - Transitions originate from actions of the system or the environment
- Traces
  - A *trace* is a sequence of states, describing a possible execution
  - A valid trace in a transition system is a path starting in an initial state

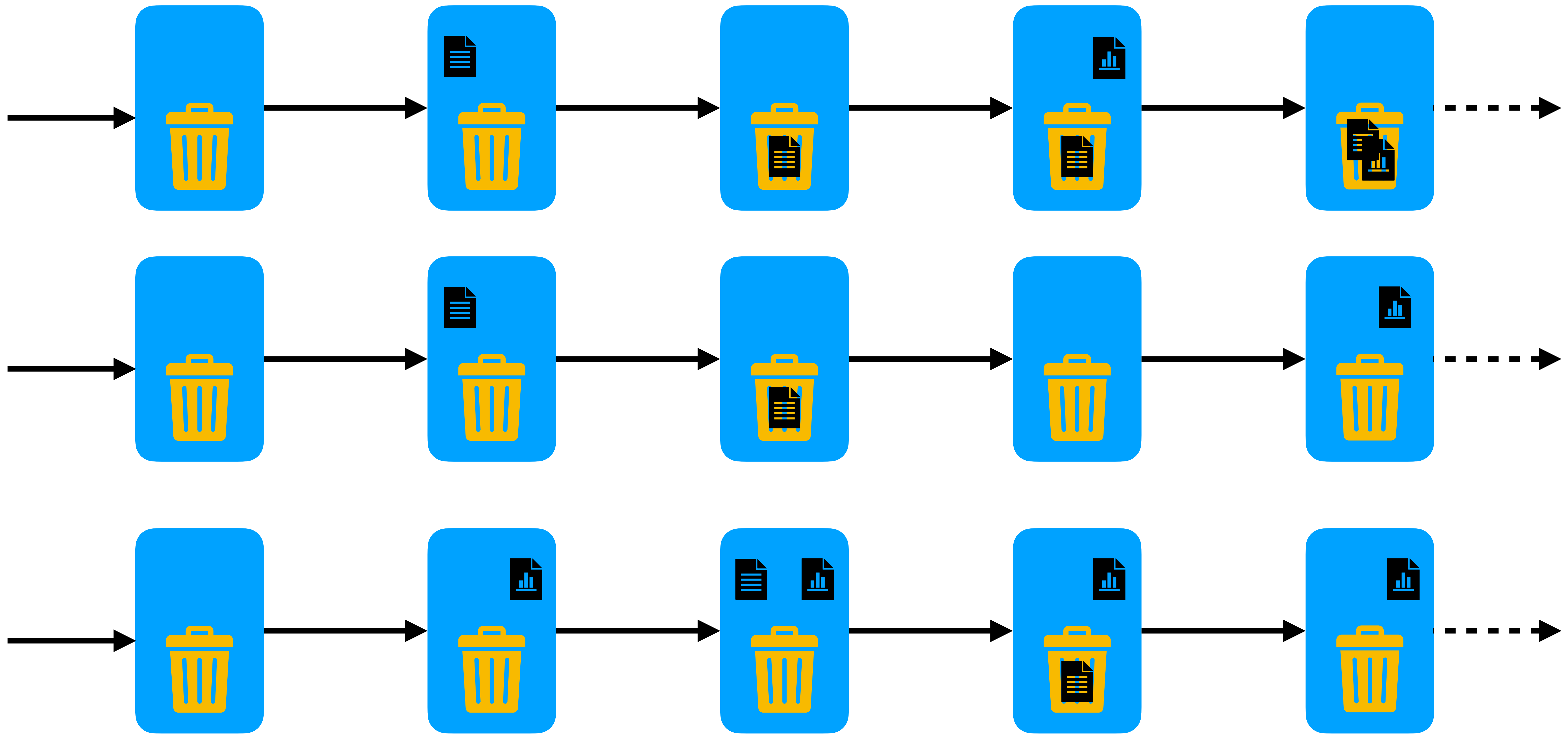
Trash transition system



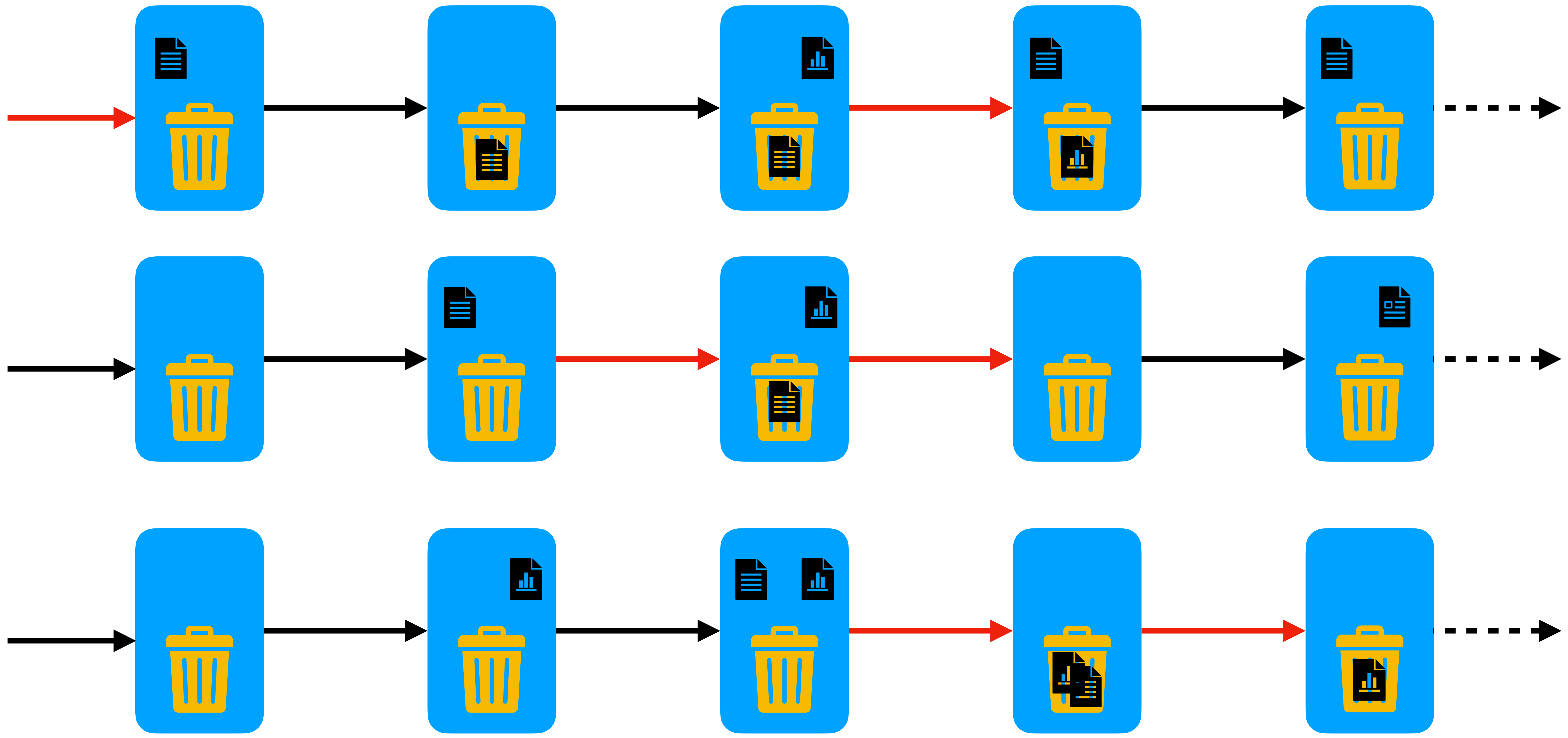
Declarative modeling

- It is possible to describe a transition system by specifying instead which are its valid traces
- This requires specifying a property whose validity is established in a trace and not just in a single state
- The specification of properties about traces requires some sort of *temporal logic*

Valid trash traces



Invalid trash traces



Specifying transition systems

Mutability

- In Alloy 6 mutable signatures and fields can be declared with keyword **`var`**
- Warnings are thrown when:
  - Static field inside mutable signature
  - Static signature extending or subset of a mutable signature

Trash states

```
sig Item {}
var sig Accessible in Item {}
var sig Trashed in Item {}
```

Instances

- When mutable structures are declared, instances become **infinite** *traces*
- Analysis commands only return traces that can be represented finitely
  - Traces that loop back at some point
- Static signatures and fields are known as the *configuration* and have the same value in all states
- If there are mutable top-level signatures, **`univ`** (and **`iden`**) are also mutable

Temporal logic

- Alloy 6 also supports linear *temporal logic*
- Temporal logic adds *temporal operators* to relational logic
- Allow “quantifying” the validity of a formula over different states of a trace
- A formula without temporal operators refers to the *first* state
- Alloy 6 has both *future* and *past* temporal operators
- Also has the *prime* operator to retrieve the value of a term in the next state

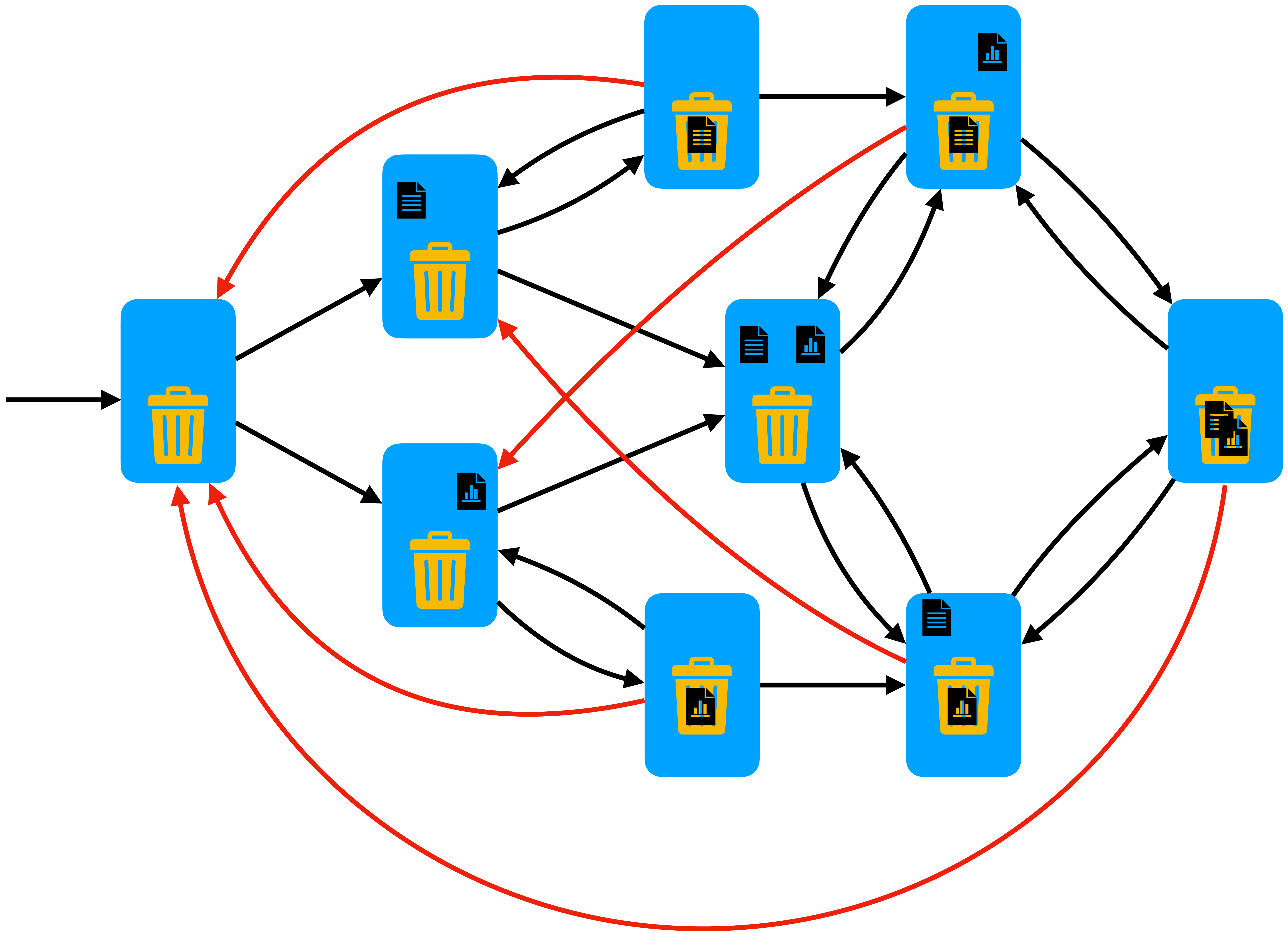
Always, eventually, and prime

`always` ϕ	ϕ is true in all future states
`eventually` ϕ	ϕ is true in some future state
R`'`	The value of R in the next state

Actions

- A set of transitions can be specified declaratively with an *action*
- An action is a formula without temporal operators, referring to primed and unprimed variables
  - A condition without primed variables is a *guard* that specifies when is the action enabled
  - A condition with a primed variable is an *effect* that specifies what are the possible values for that variable after the action occurs
  - If a variable does not change, a *frame condition* should be included stating that the next value of the variable remains the same
- By combining actions with the **always** temporal operator we can specify a system behavior
- Actions were first introduced by Leslie Lamport in the *Temporal Logic of Actions*

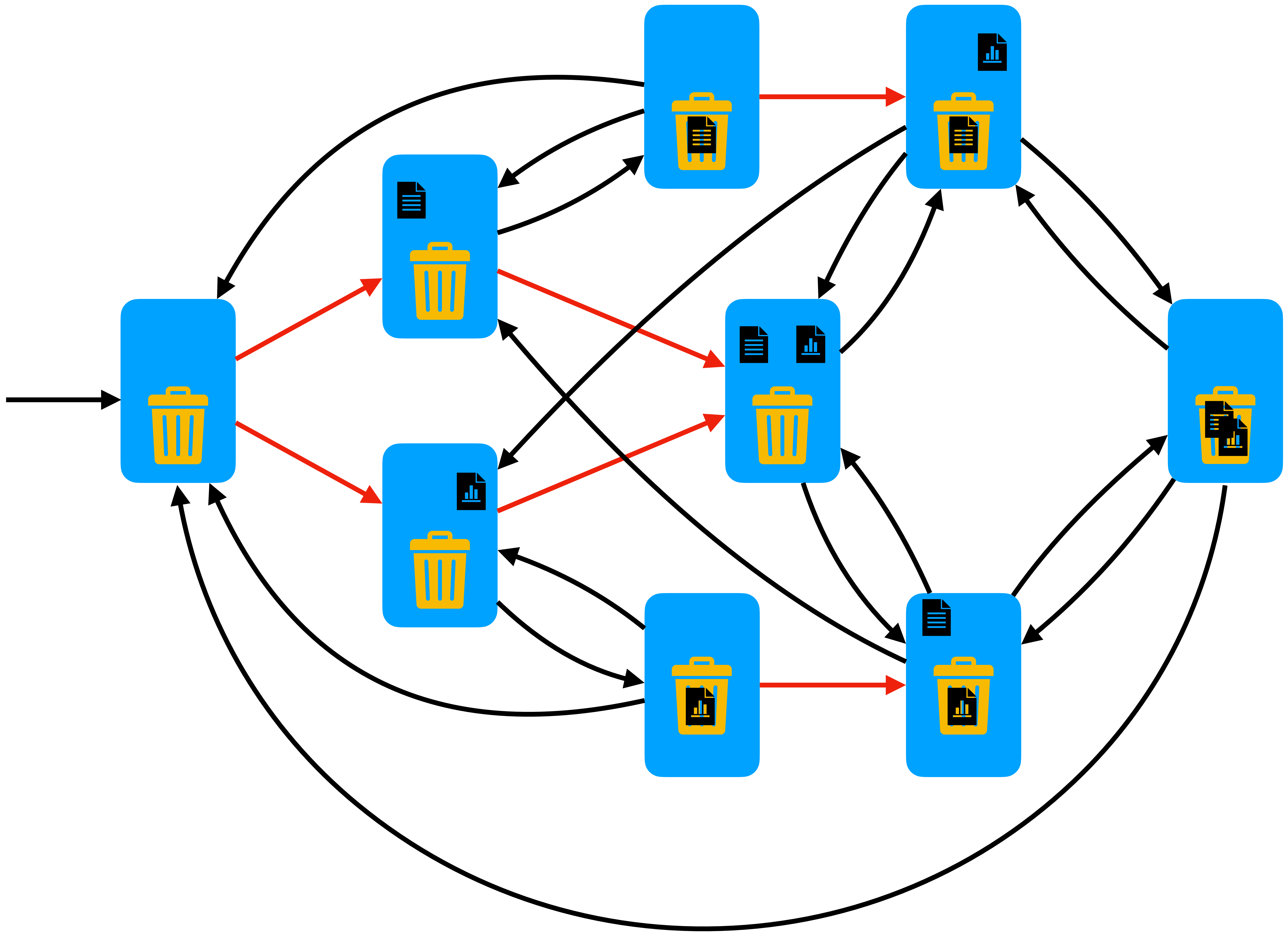
Empty trash



Empty trash

```
pred empty {
  // guard
  some Trashed
  // effect
  no Trashed'
  // frame condition
  Accessible' = Accessible
}
```

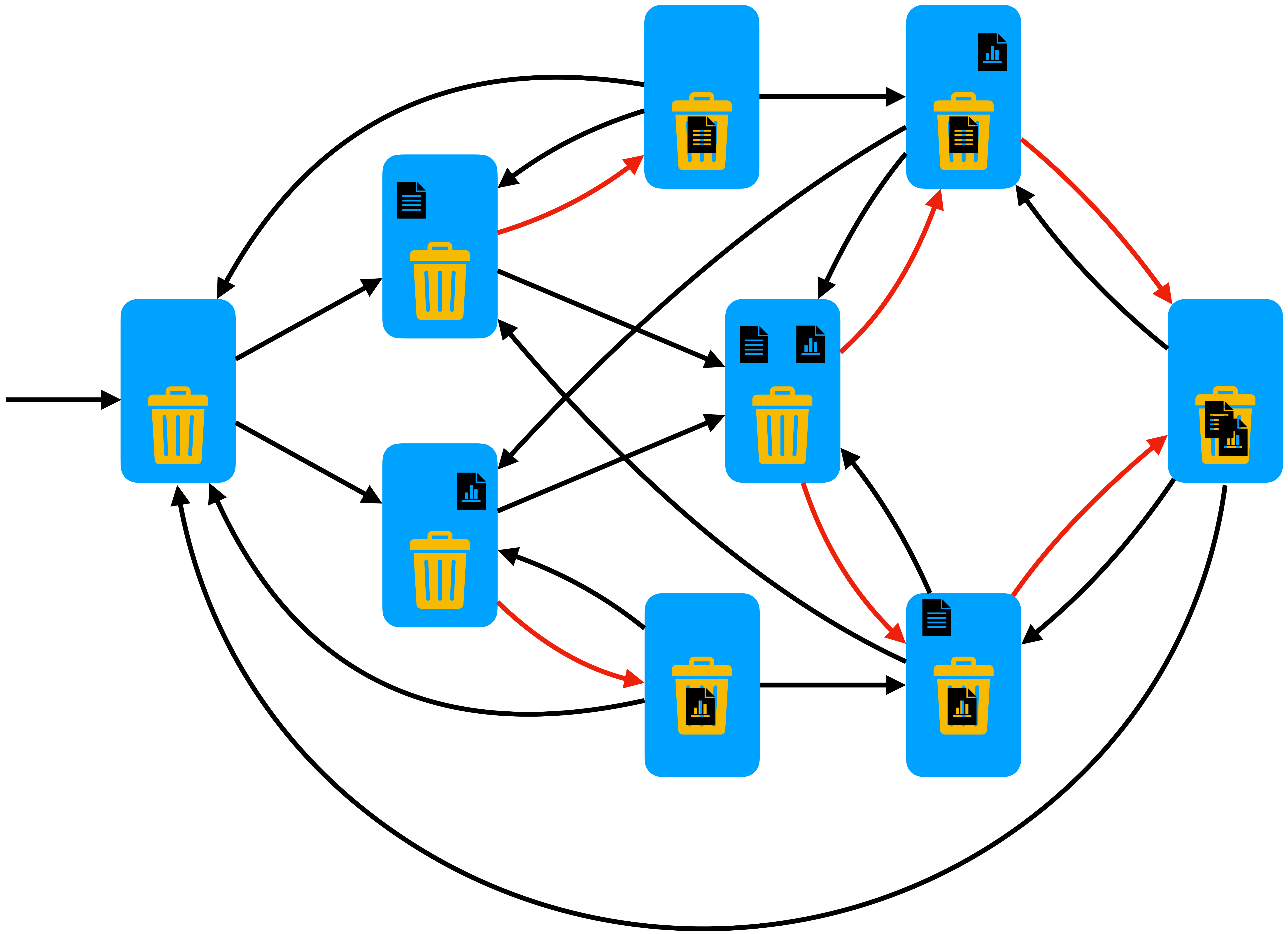
Create item



Create item

```
pred create[i : Item] {
  // guard
  i not in Accessible + Trashed
  // effect
  Accessible' = Accessible + i
  // frame condition
  Trashed' = Trashed
}
```

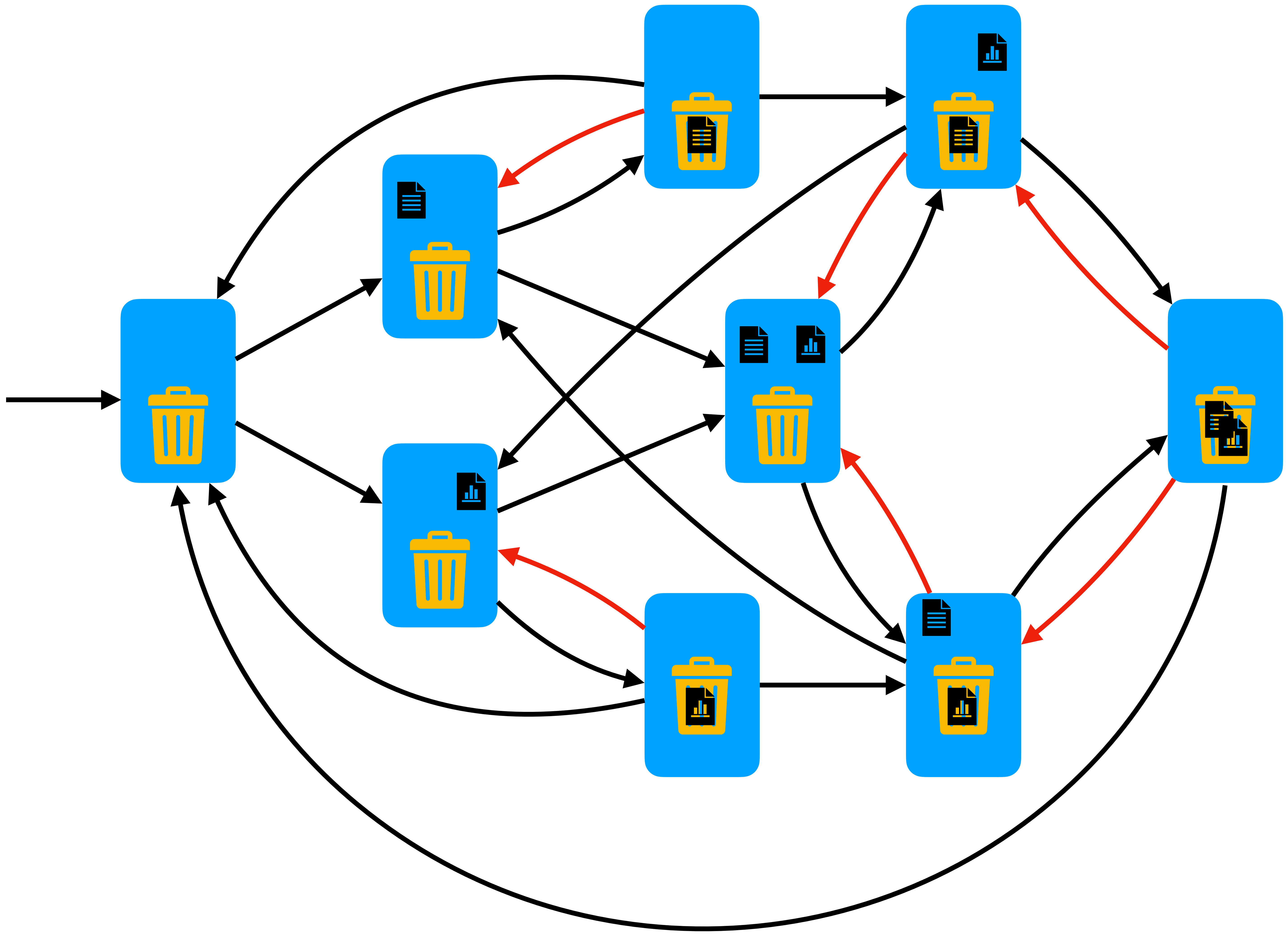
Delete item



Delete item

```
pred delete[i : Item] {
  // guard
  i in Accessible
  // effects
  Accessible' = Accessible - i
  Trashed' = Trashed + i
}
```

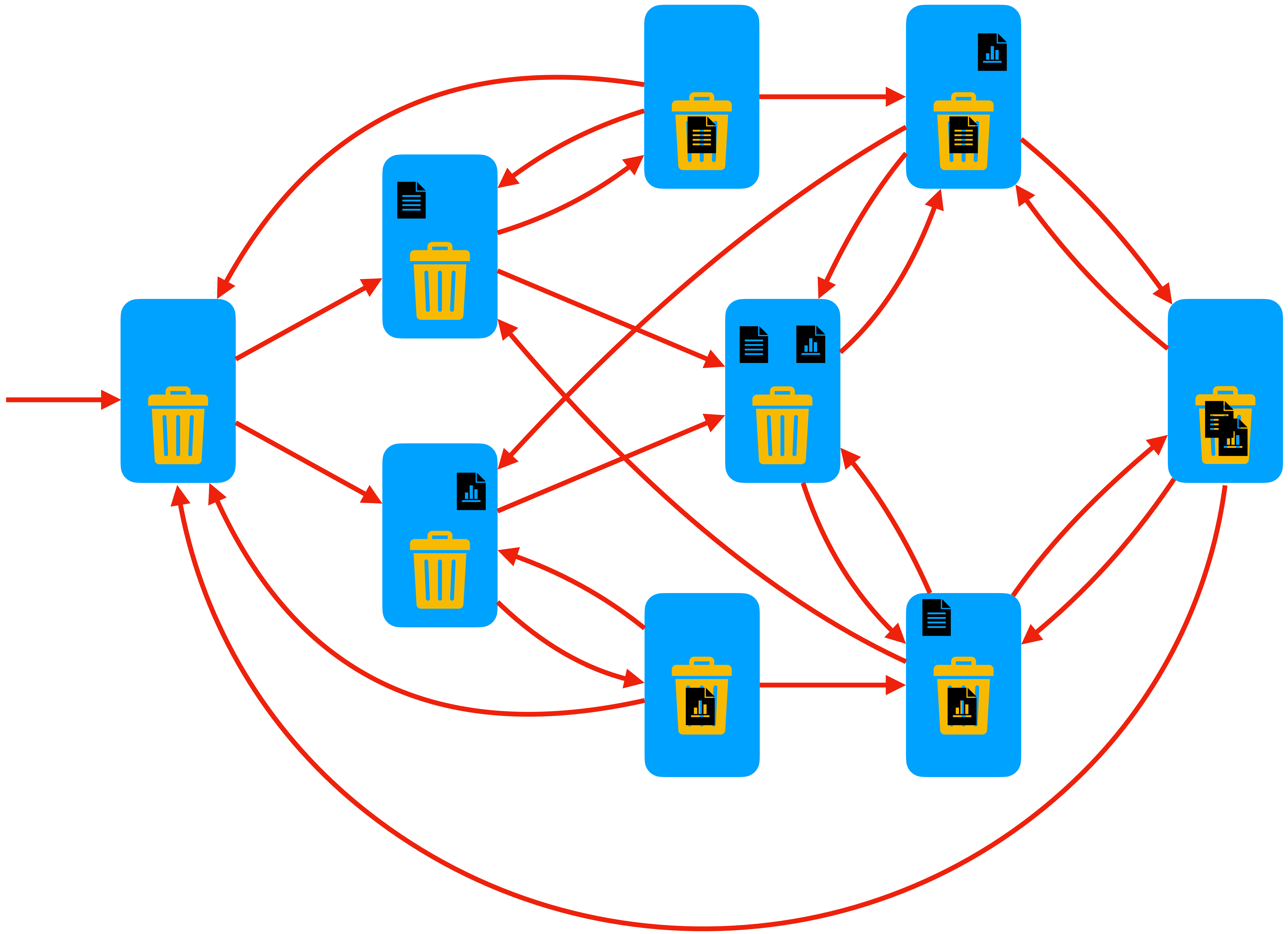
Restore item



Restore item

```
pred restore[i : Item] {
  // guard
  i in Trashed
  // effects
  Accessible' = Accessible + i
  Trashed' = Trashed - i
}
```

Trash behavior



Trash behavior

```
fact Behavior {
  // initial state
  no Accessible
  no Trashed
  // possible transitions
  always {
    (some i : Item | create[i] or delete[i] or restore[i])
    or
    empty
  }
}
```

Validation

Run commands

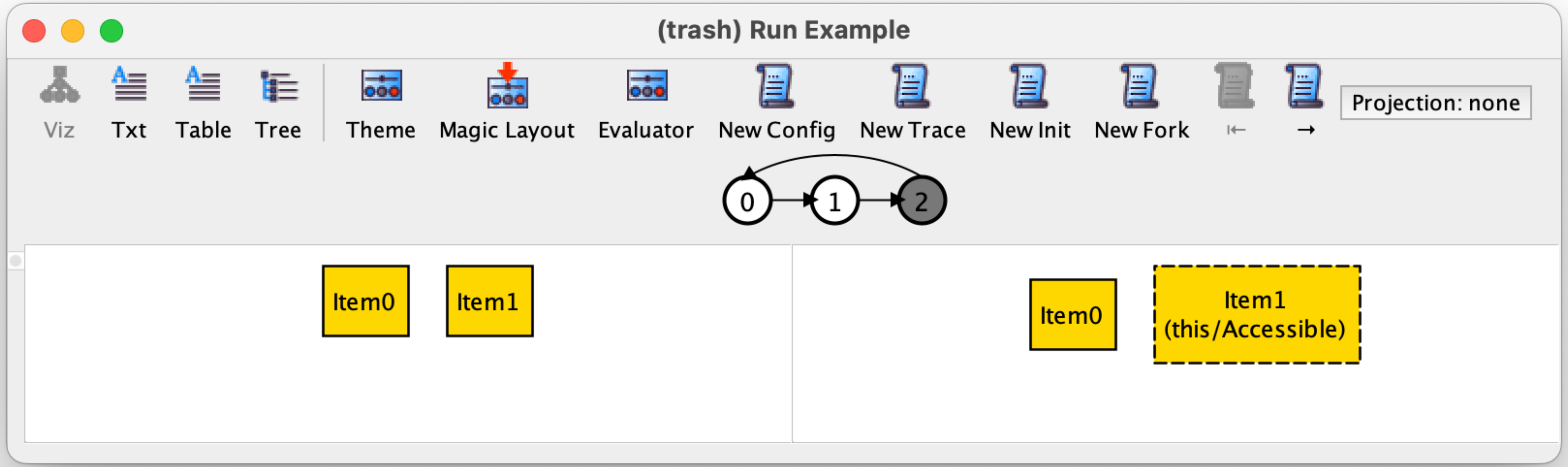
- As usual, **`run`** commands can be used to validate the model
- The scope of a mutable signature defines the maximum number of different atoms in the full trace, not a maximum per state

Trace visualization

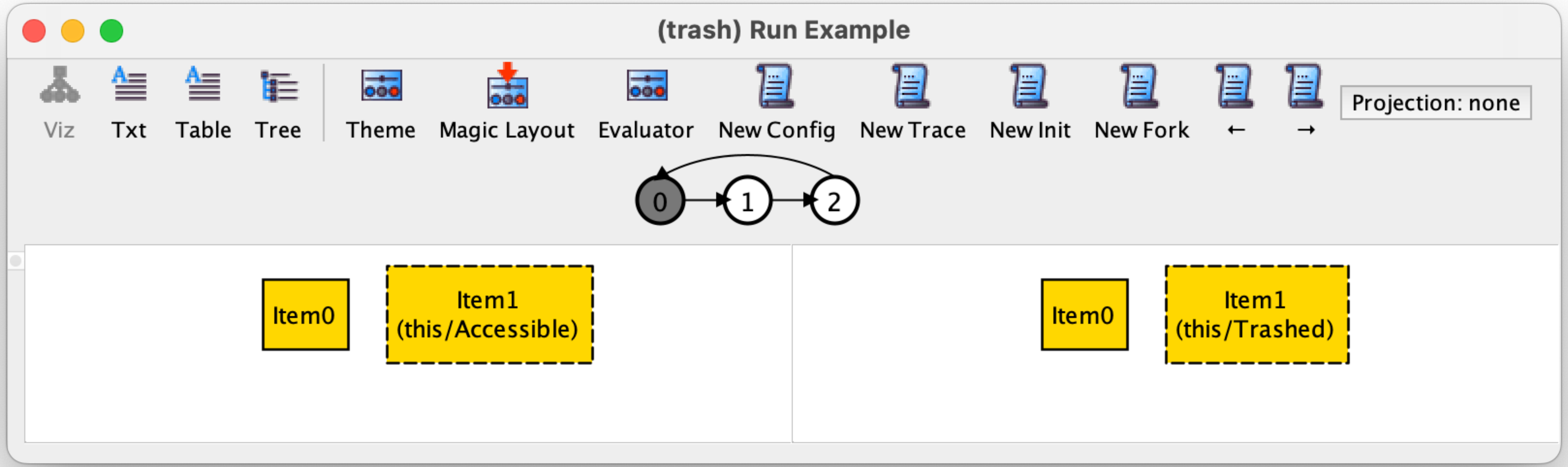
- The visualizer depicts two consecutive states of the trace side-by-side
    - By default mutable structures are depicted with dashed lines
- A representation of the infinite trace is shown on top
    - Different states have different numbers
    - The loop back is explicitly depicted
    - Clicking on a state focus on that (and the succeeding) state
    - It is possible to move forwards and backwards in the trace with → and ←

Trace visualization

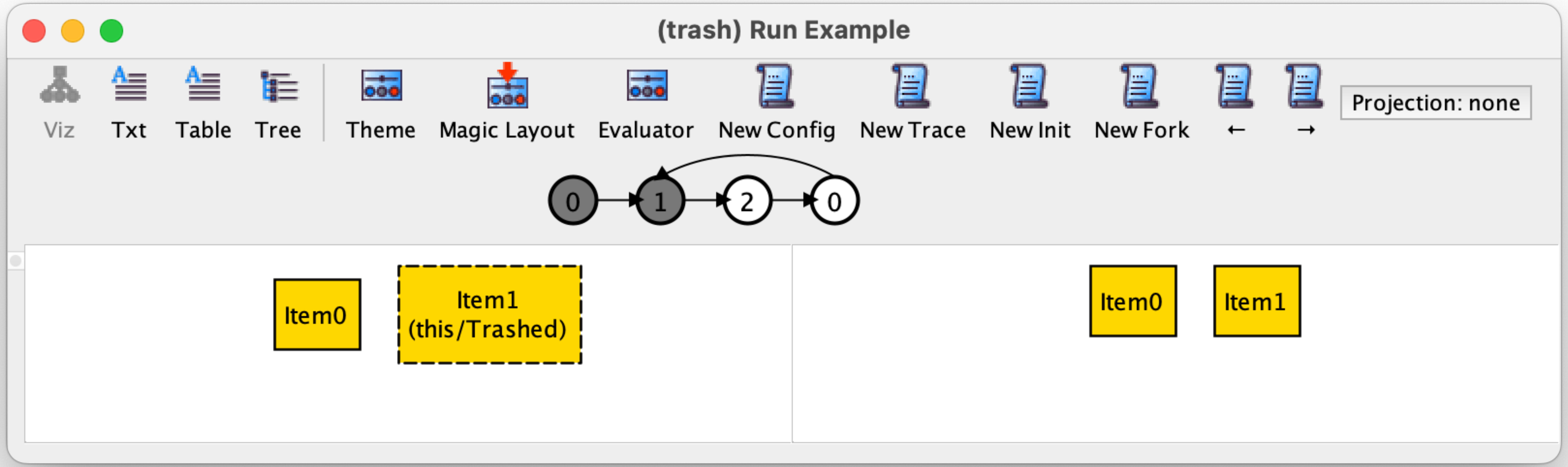
Trace visualization



Trace visualization



Trace visualization



Simulation

- It possible to perform “simulation” with the *New* instance buttons
  - *New config*, returns a trace with a different configuration (a different value for the immutable structures)
  - *New trace*, returns any different trace with the same configuration
  - *New init*, returns a trace with the same configuration, but a different initial state
  - *New fork*, returns a trace with the same prefix, but a different next state (at the focused transition)

Simulation

DEMO

Specifying scenarios

- A formula can be given in a **`run`** command to look for specific scenarios
- Keyword **`expect`** can be used to distinguish positive and negative scenarios

Semi-colon

$\phi \;\; \mathtt{;} \;\; \psi$

ψ is valid after ϕ

Some trash scenarios

```
run Scenario1 {
  some i : Item {
    create[i]; delete[i]; restore[i]; delete[i]; empty
  }
} expect 1

run Scenario2 {
  some disj i,j : Item {
    create[i]; delete[j]
  }
} expect 0

run Scenario3 {
  some i : Item {
    create[i]; delete[i]; empty
  }
} for 1 Item expect 1
```

Stuttering

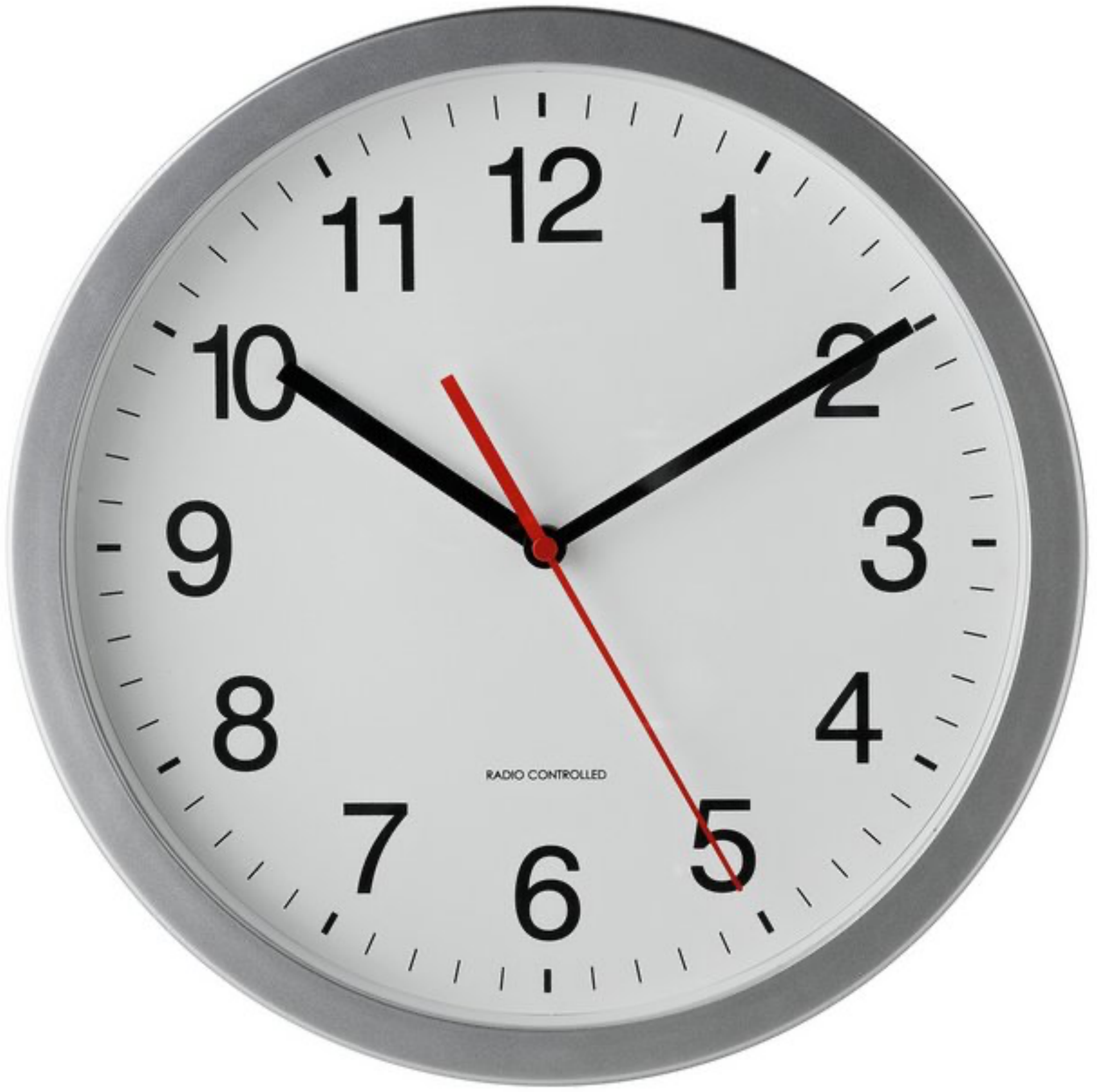
A clock specification

```
pred clock_spec {
  h = 12 and m = 0
  always {
    m'=(m+1)%60 and
    m=59 implies h'=(h%12)+1 else h'=h
 }
}
```

Ceci n'est pas une montre?!

```
check clock_spec
```

```
Executing "Check clock_spec"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..2 steps. 55 vars. 12 primary vars. 59 clauses. 3ms.
   Counterexample found. Assertion is invalid. 3ms.
```



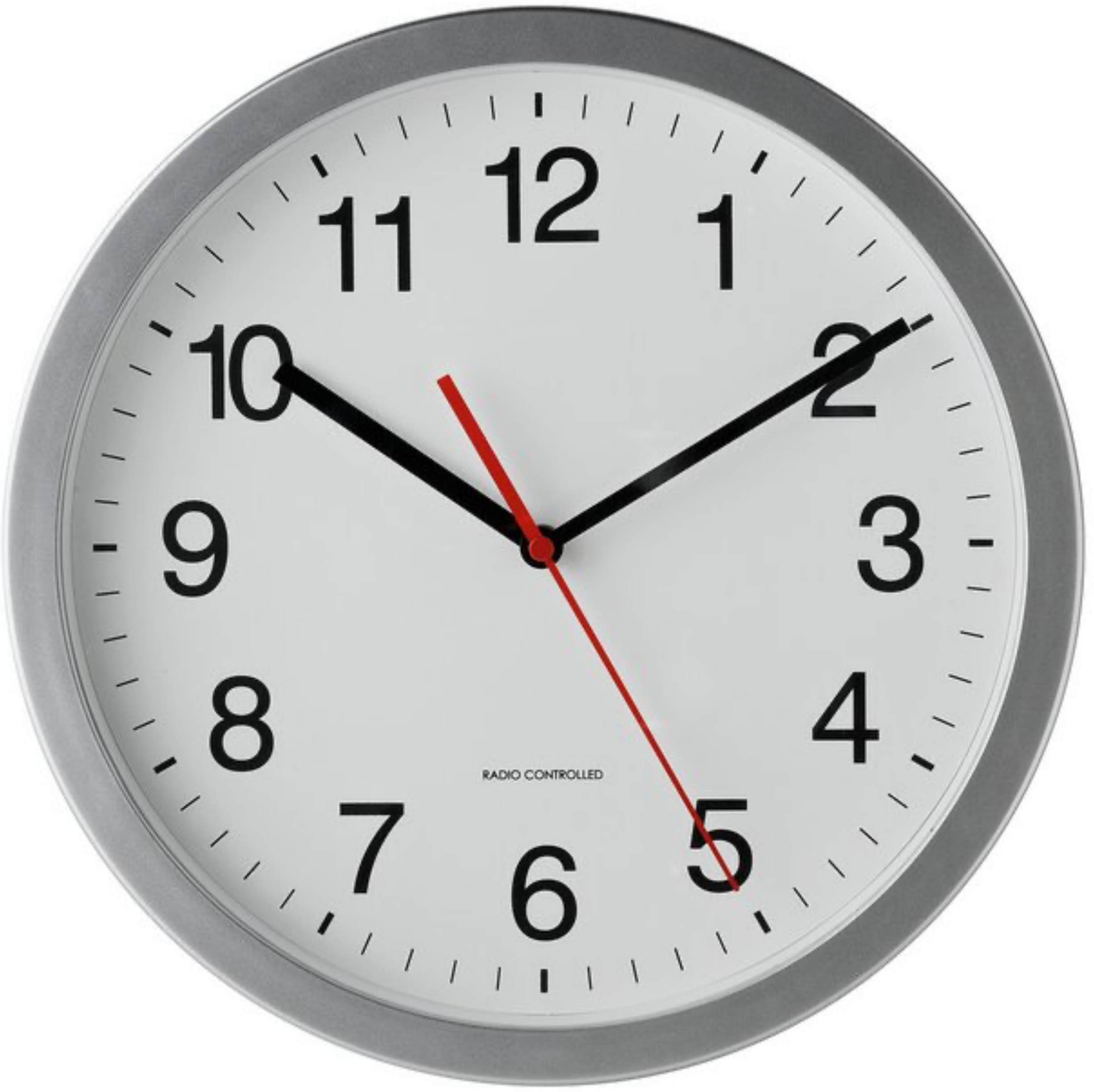
A clock specification

```
pred clock_spec {
  h = 12 and m = 0
  always {
    m’=(m+1)%60 and
    m=59 implies h’=(h%12)+1 else h’=h
    or
    m’=m and h’=h
  }
}
```

Another clock

```
check clock_spec
```

```
Executing "Check clock_spec"
   Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
   1..10 steps. 151901 vars. 1875 primary vars. 413006 clauses. 1042ms.
   No counterexample found. Assertion may be valid. 298ms.
```



Stuttering

- *Stuttering* can represent events by the environment or by other components of the system (not yet modeled)
- Stuttering enables *refinement*
    - adding detail or new components to a system
    - namely, it enables concepts to be composed to build apps
- In terminating systems, stuttering enables traces to be infinite

Trash stuttering

```
pred stutter {
  Accessible' = Accessible
  Trashed' = Trashed
}
```

Trash behavior

```
fact Behavior {
  // initial state
  no Accessible
  no Trashed
  // possible transitions
  always {
    (some i : Item | create[i] or delete[i] or restore[i])
    or
    empty
    or
    stutter
  }
}
```

Verification

Model checking

- *Model checking* is the process of automatically verifying if a temporal logic specification holds in a finite transition system model of a system
  - If the specification is false a counter-example is returned
  - A finite transition system may have infinite non-looping traces
  - But every invalid specification can be falsified with a looping trace
- *Complete* or *unbounded* model checking explores all traces of the transition system
- *Bounded* model checking explores all traces up to a given maximum number of transitions before looping back

Verification

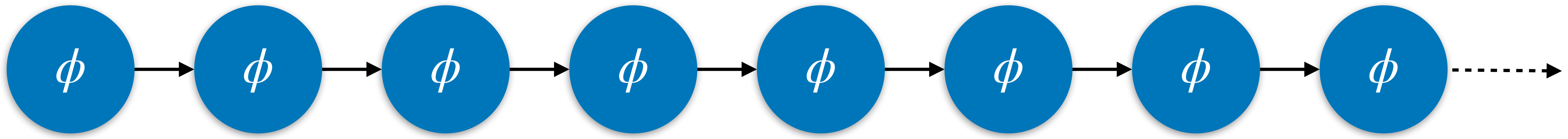
- **`check`** commands can be used to verify temporal assertions
- The default verification mechanism is bounded model checking
  - The default maximum number of transitions is 10
  - This can be changed by setting a scope for **`steps`**
- Alloy 6 also supports unbounded model checking
  - Activated by the special scope `1..` **`steps`**
  - Requires model checkers nuXmv or NuSMV to be installed

Future temporal operators

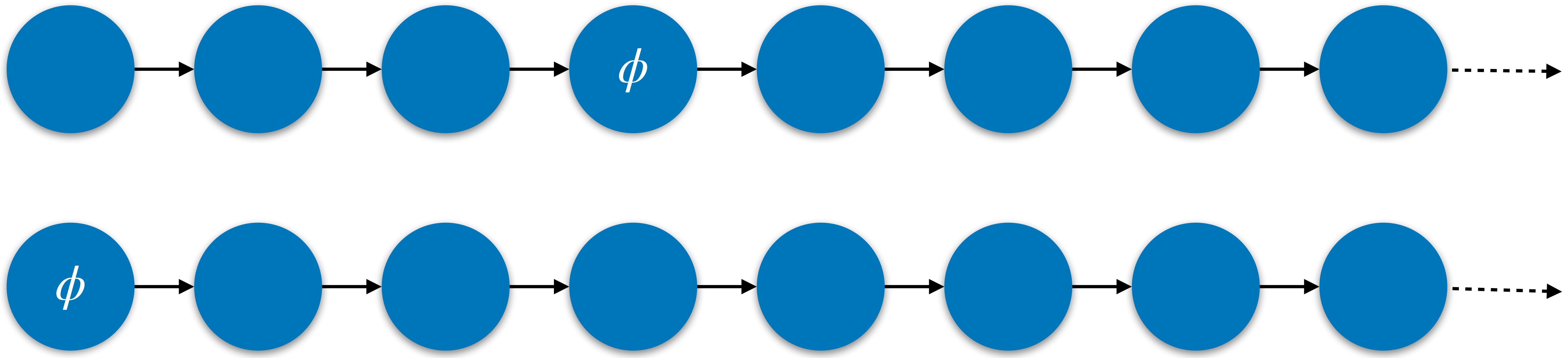
`always` ϕ	ϕ will always be true
`eventually` ϕ	ϕ will eventually be true
`after` ϕ	ϕ will be true in the next state
ψ **`until`** ϕ	ϕ will eventually be true and ψ is true until then
ϕ **`releases`** ψ	ψ can only stop being true after ϕ

Future operators

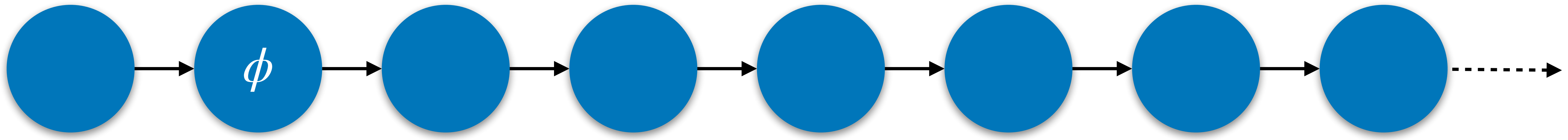
always ϕ



eventually ϕ

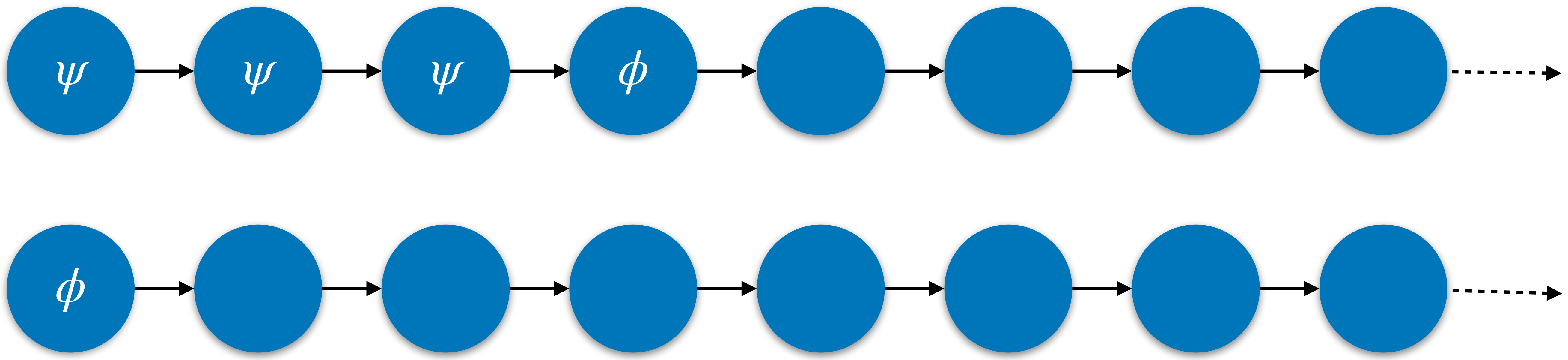


after ϕ

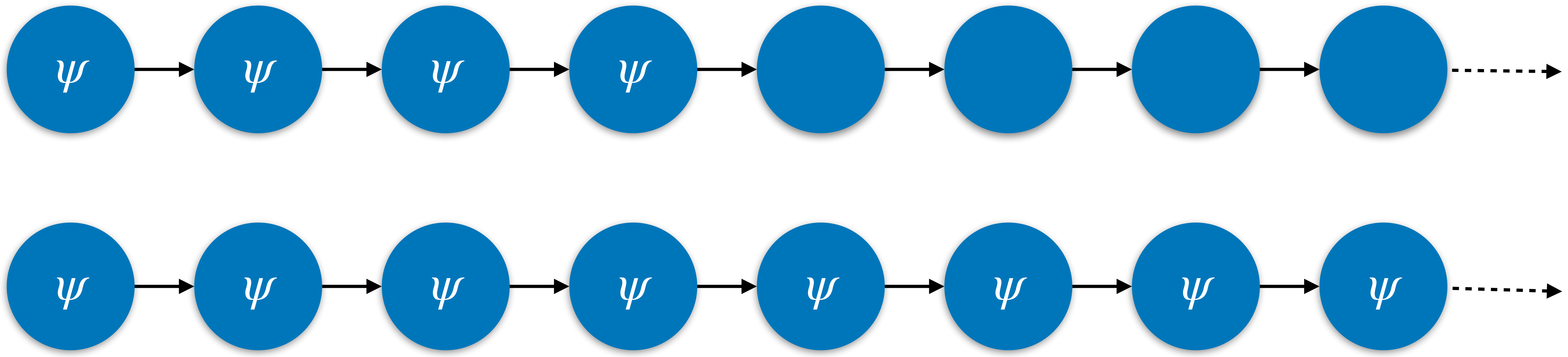


Future operators

ψ **until** ϕ

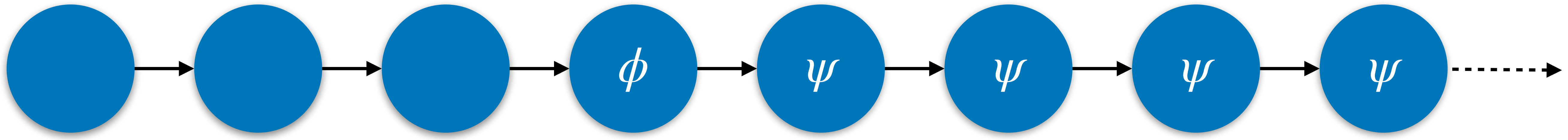


ϕ **releases** ψ

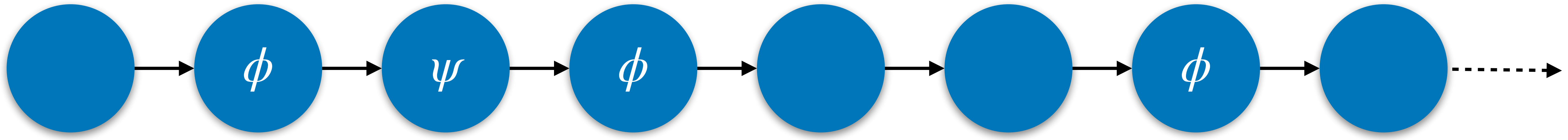


Mixing operators

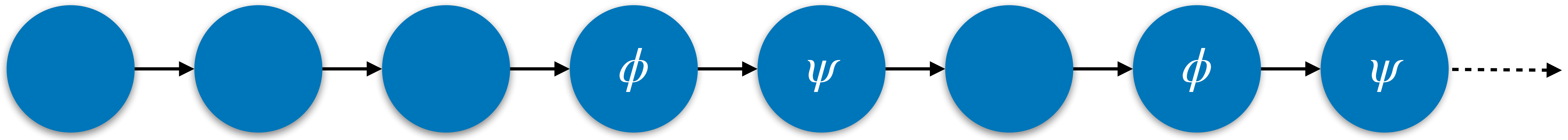
always (ϕ implies always ψ)



always (ϕ implies eventually ψ)

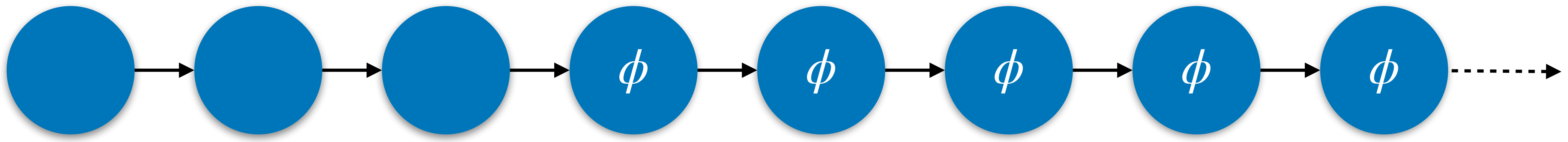


always (ϕ implies after ψ)

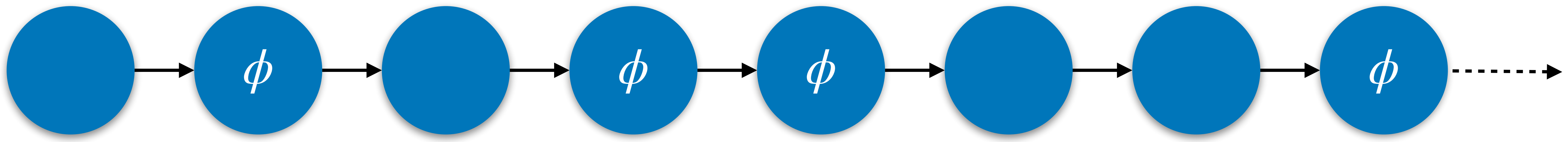


Mixing operators

`eventually (always` ϕ`)`



`always (eventually` ϕ`)`

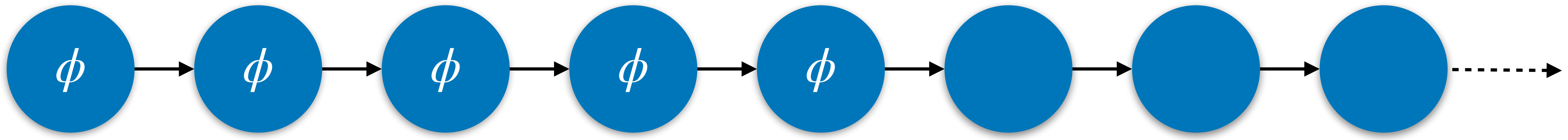


Past temporal operators

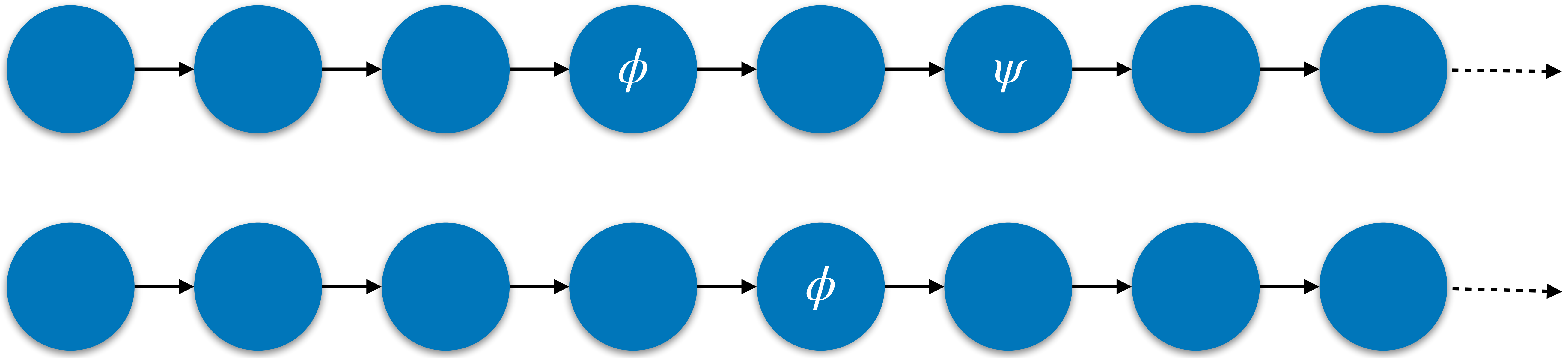
`historically` ϕ	ϕ was always true
`once` ϕ	ϕ was once true
`before` ϕ	ϕ was true in previous state
ψ **`since`** ϕ	ϕ was once true and ψ was true since then
ϕ **`triggered`** ψ	ψ was always true back to the point where ϕ was true

Past operators

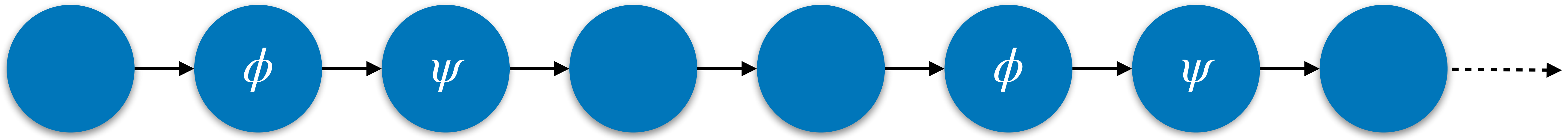
always (ψ implies historically ϕ)



always (ψ implies once ϕ)

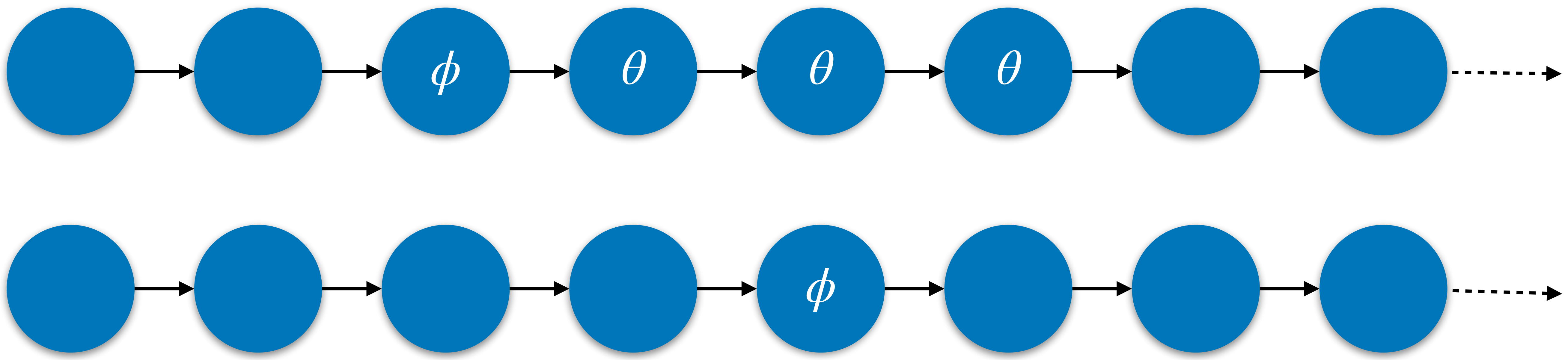


always (ψ implies before ϕ)

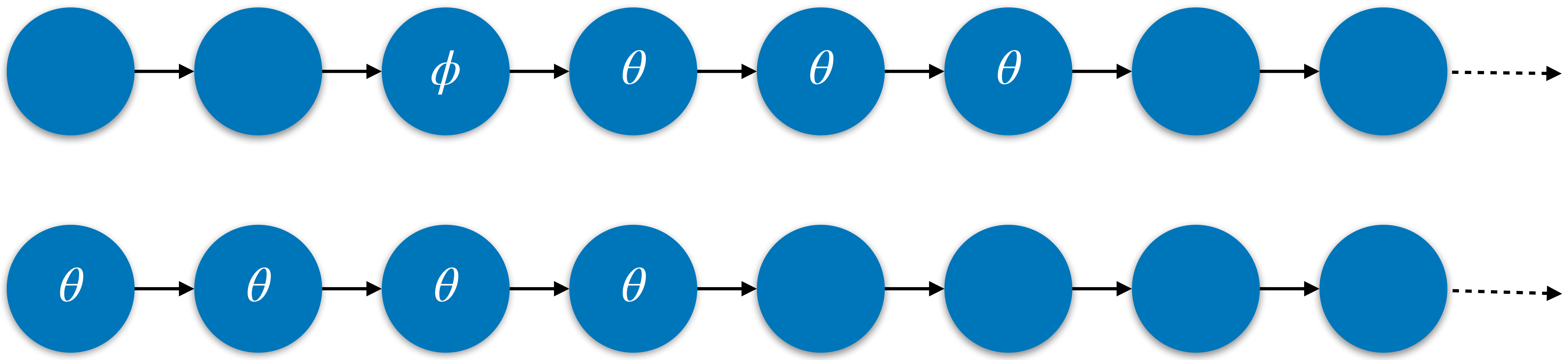


Past operators

always (ψ **implies** θ **since** ϕ)



always (ψ **implies** ϕ **triggered** θ)



Safety properties

- *Safety* properties prevent some undesired behaviors from happening
    - Easier to model check, since it suffices to search for a finite sequence of steps that leads to a bad state
    - It is irrelevant what happens afterwards, any continuation leads to a counter-example
    - The archetypal safety property is an *invariant* specified as **`always`** ϕ

Liveness properties

- *Liveness* properties force some desired behaviors to happen
  - Harder to model check, since it is necessary to search for a complete infinite trace where the desired behavior never happens
  - Harder to specify, since they require fairness assumptions that prevent the system from stuttering forever
  - The archetypal liveness property is **`eventually`** ϕ

Some operational principles

```
check invariant {
    // No item can simultaneously be accessible and trashed
    always no Accessible & Trashed
}
check restore_after_delete {
    // A restore is only possible after a delete
    all x : Item | always (restore[x] implies once delete[x])
}
check accessible_after_delete {
    // A deleted item only becomes accessible again after being restored or created
    all x : Item | always {
        delete[x] implies after {
            (restore[x] or create[x]) releases x not in Accessible
        }
    }
}
```

The key operational principles

```
pred can_restore[x : Item] { x in Trashed }
check delete_restore {
    // After delete(x), can restore(x) and then x in accessible
    all x : Item | always {
        delete[x] implies after can_restore[x]
        (delete[x]; restore[x]) implies x in Accessible''
    }
} for 4 Item, 20 steps

pred can_empty { some Trashed }
check delete_empty {
    // After delete(x), can empty() and then x not in accessible or trashed
    all x : Item | always {
        delete[x] implies after can_empty
        delete[x] and after empty implies x not in (Trashed+Accessible)''
    }
} for 4 Item, 20 steps
```

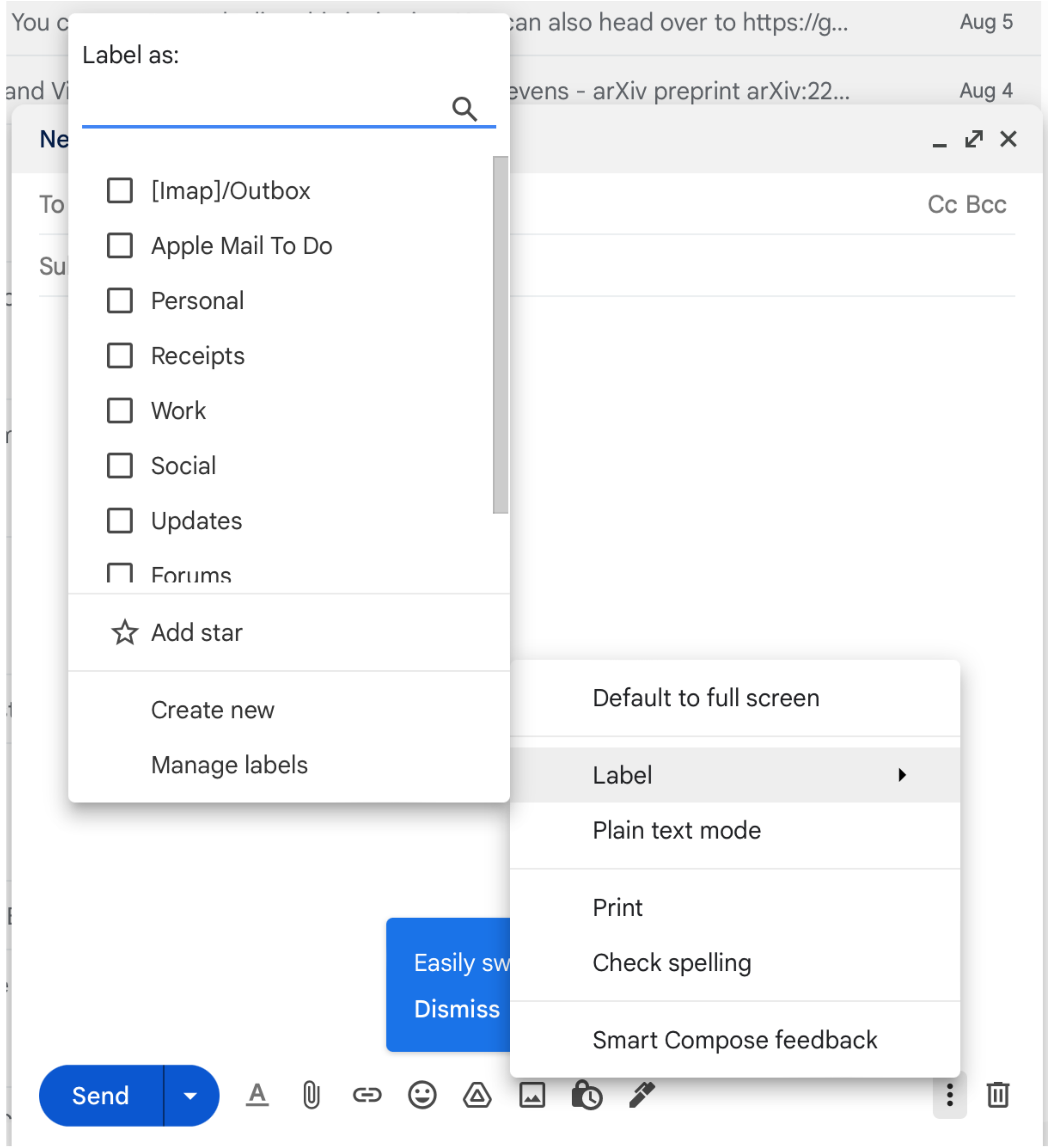
Verification

```
8 commands were executed. The results are:
   #1: Instance found. Scenario1 is consistent, as expected.
   #2: No instance found. Scenario2 may be inconsistent, as expected.
   #3: Instance found. Scenario3 is consistent, as expected.
   #4: No counterexample found. invariant may be valid.
   #5: No counterexample found. restore_after_delete may be valid.
   #6: No counterexample found. accessible_after_delete may be valid.
   #7: No counterexample found. delete_restore may be valid.
   #8: No counterexample found. delete_empty may be valid.
```

Another *concept*

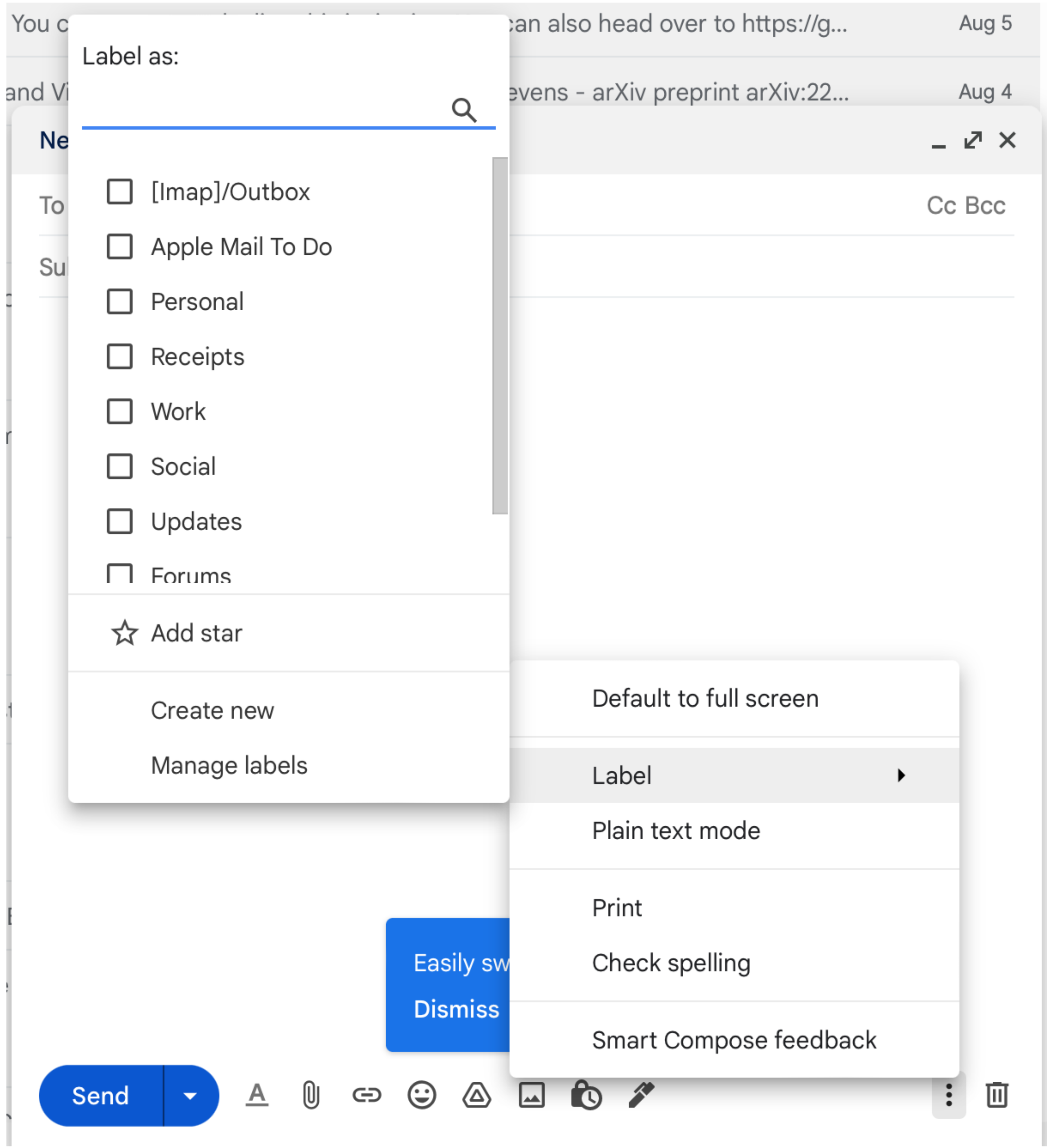
Label

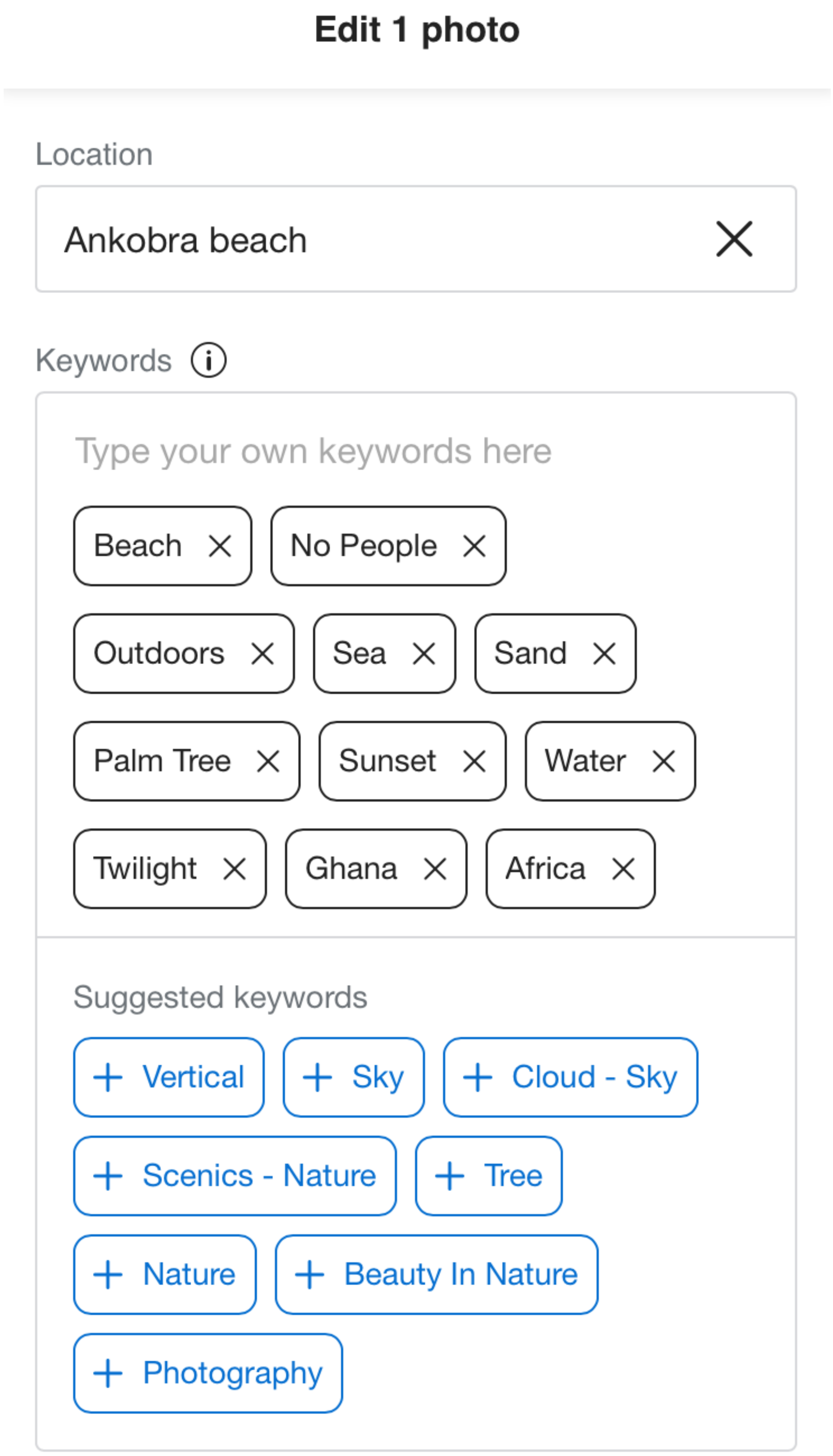
Label



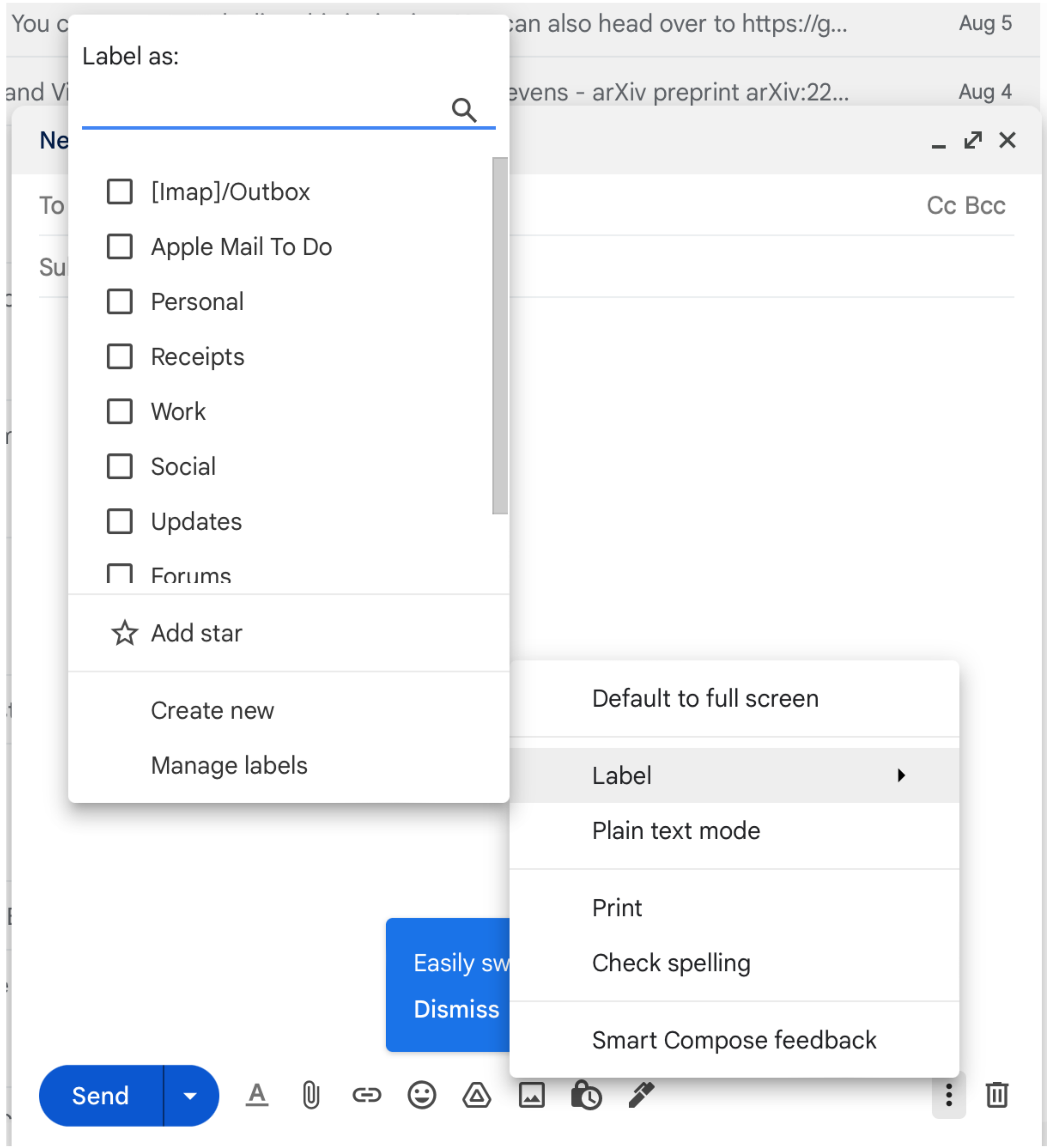
You c
can also head over to https://g...
Aug 5
and Vi
evens - arXiv preprint arXiv:22...
Aug 4
Label as:
Ne
To
Su
Cc Bcc
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Easily sw
Dismiss
Send

Label

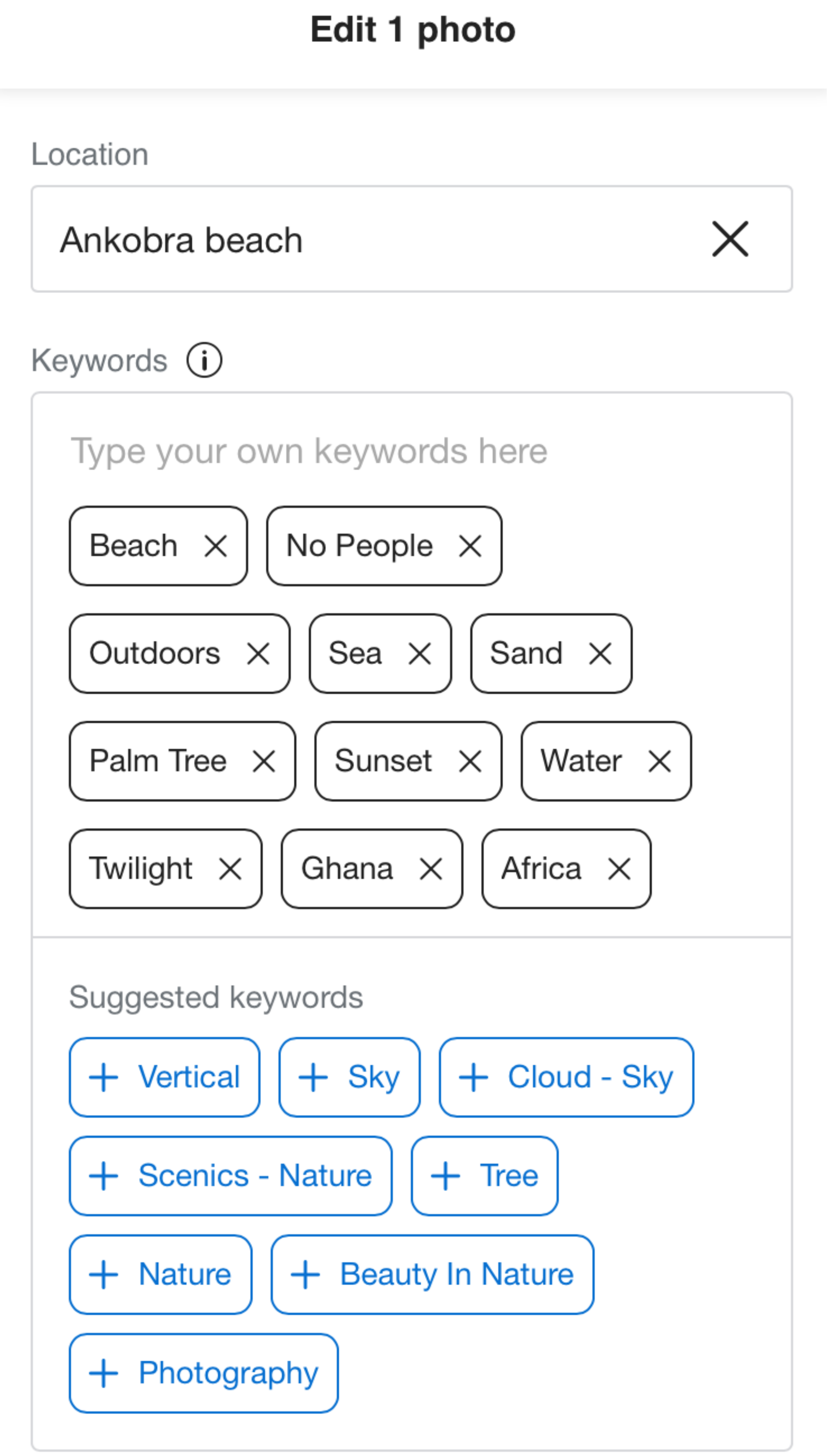

Label as:
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
can also head over to https://g...
Aug 5
evens - arXiv preprint arXiv:22...
Aug 4
Cc Bcc
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Easily sw
Dismiss
Send


Edit 1 photo
Location
Ankobra beach
Keywords
Type your own keywords here
Beach
No People
Outdoors
Sea
Sand
Palm Tree
Sunset
Water
Twilight
Ghana
Africa
Suggested keywords
Vertical
Sky
Cloud - Sky
Scenics - Nature
Tree
Nature
Beauty In Nature
Photography

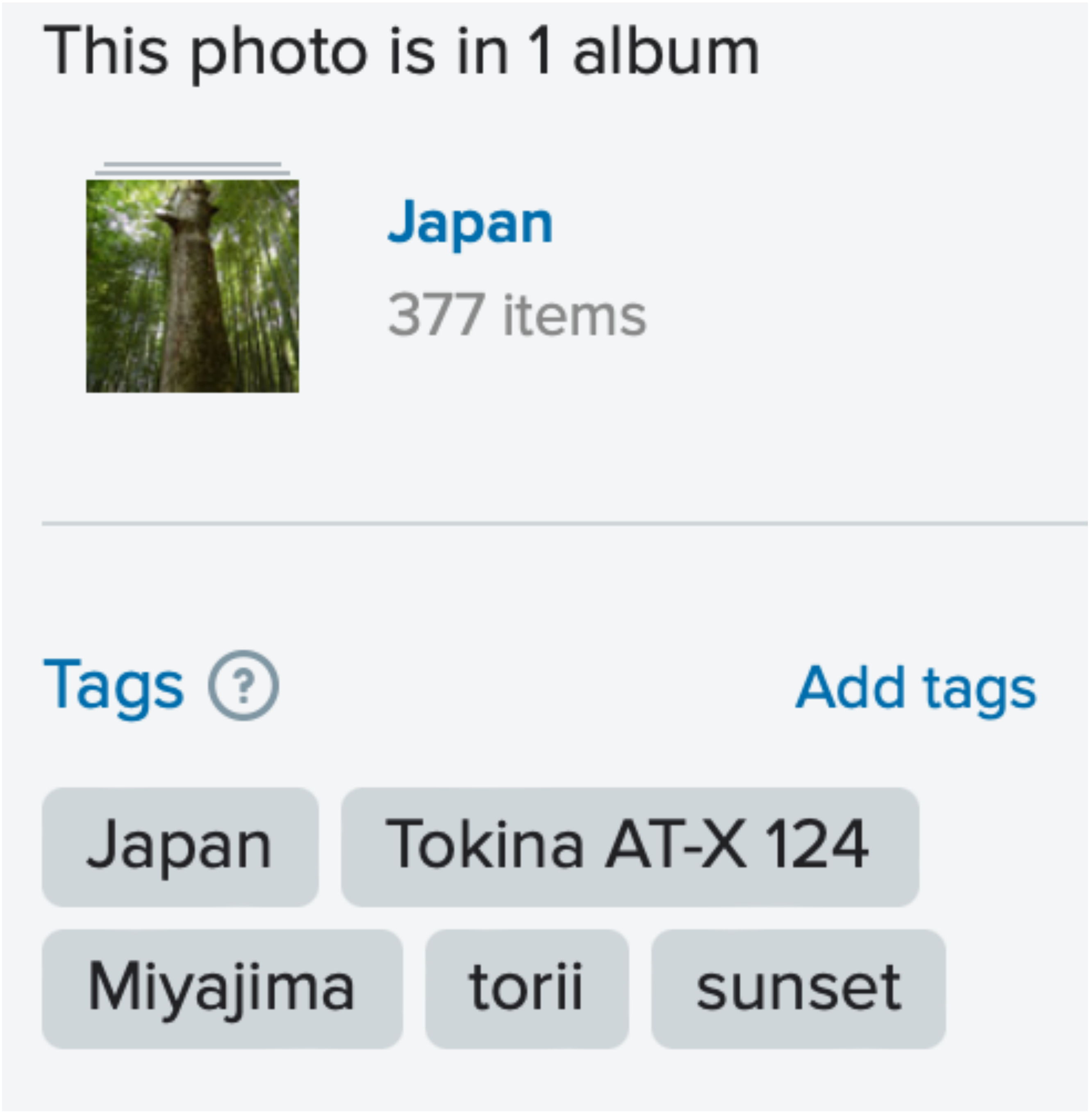
Label



Label as:
[Imap]/Outbox
Apple Mail To Do
Personal
Receipts
Work
Social
Updates
Forums
Add star
Create new
Manage labels
can also head over to https://g...
Aug 5
evens - arXiv preprint arXiv:22...
Aug 4
Cc Bcc
Default to full screen
Label
Plain text mode
Print
Check spelling
Smart Compose feedback
Dismiss
Send



Edit 1 photo
Location
Ankobra beach
Keywords
Type your own keywords here
Beach
No People
Outdoors
Sea
Sand
Palm Tree
Sunset
Water
Twilight
Ghana
Africa
Suggested keywords
Vertical
Sky
Cloud - Sky
Scenics - Nature
Tree
Nature
Beauty In Nature
Photography



This photo is in 1 album
Japan
377 items
Tags
Add tags
Japan
Tokina AT-X 124
Miyajima
torii
sunset

Label modeling *à la* Jackson

```
concept label [Item]
purpose
  organize items into overlapping categories
state
  labels : Item -> set Label
actions
  affix (i : Item, l : Label)
    when l not in the labels of i
    add l to the labels of i
  detach (i : Item, l : Label)
    when l in the labels of i
    remove l from the labels of l
  clear (i : Item)
    when i has some labels
    remove all labels of i
operational principle
  after affix(i,l), while no detach(i,l) and no clear(i), i is in the labels of l
```

The label in Alloy

```
sig Item {
  var labels : set Label
}
sig Label {}

fact Behavior {
  no labels
  always {
    (some i : Item, l : Label | affix[i,l] or detach[i,l])
    or
    (some i : Item | clear[i])
    or
    stutter
  }
}
```

Affix label with *point-wise* effect

```
pred affix[i : Item, l : Label] {
  // guard
  l not in i.labels
  // effect
  i.labels' = i.labels + l
  // frame condition
  all j : Item - i | j.labels' = j.labels
}
```

Affix label with *point-free* effect

```
pred affix[i : Item, l : Label] {
  // guard
  l not in i.labels
  // effect
  labels' = labels + i->l
}
```

Detach label

```
pred detach[i : Item, l : Label] {
  // guard
  l in i.labels
  // effect
  labels' = labels - i->l
}
```

Clear item

```
pred clear[i : Item] {
  // guard
  some i.labels
  // effect
  labels' = labels - i->Label
}
```

Label scenarios

```
run Scenario1 {
  some i : Item, disj l,m : Label {
    affix[i,l]; affix[i,m]; clear[i]
  }
} expect 1

run Scenario2 {
  some i : Item, l : Label {
    affix[i,l]; affix[i,l]
  }
} expect 0
```

Label operational principle

```
check affix_find {
    // after affix(i,l), while no detach(i,l) and no clear(i), i is in the labels of l
    all i : Item, l : Label | always {
        affix[i,l] implies after ((detach[i,l] or clear[i]) releases l in i.labels)
    }
}
```

App design

Modularizing concepts

- To enable reuse and instantiation each concept should be in a parametrized module
- The module can still be used on its own, as Alloy implicitly declares parameter signatures
- Since a parameter signature cannot be extended with new fields, some tricks might be necessary to declare them

Trash

```
module Trash[Item]

sig Item {}
var sig Accessible in Item {}
var sig Trashed in Item {}

...
```

Label

```
module Label[Item]

sig Item {
  var labels : set Label
}
sig Aux in Item {
  var labels : set Label
}
fact { Aux = Item }
sig Label {}

...
```

Specifying apps

- Import the required concepts, instantiating parameter signatures as needed
- Compose the concepts
  - Enforce interleaving, by requiring at most one concept not to stutter
  - Synchronize actions as needed
- Validate, validate, validate
- Check some expected properties

A filesystem app

- Composed of trash and label
- Many options to explore
  - When to allow affixing labels?
  - When to delete labels?
  - Whether to use special labels?

Free composition

```
open Trash[File] as trash
open Label[File] as label

sig File {}

fact Interleave {
  always {
    trash/stutter or
    label/stutter
  }
}

run Example {}
```

Filesystem v1

- Allow labelling only when accessible
- Clear labels when file is deleted

Filesystem v1

```
fact Synchronization {
  // allow affixing only if file is accessible
  all f : File, l : Label | always (affix[f,l] implies f in Accessible)

  // clear all labels after file is deleted
  all f : File | always (delete[f] and some f.labels implies after clear[f])
}
```

Filesystem v1

```
run Scenario1 {
  some f : File, l : Label {
    create[f]; affix[f,l]; delete[f]
  }
} expect 1

run Scenario2 {
  some f : File, l : Label {
    create[f]; delete[f]; affix[f,l]
  }
} expect 0
```

Filesystem v1

DEMO

Filesystem v2

- Allow labelling when accessible or trashed
- Clear labels when trash is emptied

Filesystem v2

```
fact Synchronization {
  // allow labelling when accessible or trashed
  all f : File, l : Label | always (affix[f,l] implies f in Accessible+Trashed)

  // clear labels when trash is emptied
  always {
    empty implies after {
     (some f : File-Accessible | clear[f]) until no (File-Accessible).labels
    }
  }
}
```

Filesystem v2

```
run Scenario1 {
  some f : File, l : Label {
    create[f]; affix[f,l]; delete[f]
  }
} expect 1

run Scenario2 {
  some f : File, l : Label {
    create[f]; delete[f]; affix[f,l]
  }
} expect 1
```

Filesystem v2

```
run Scenario3 {
    some f : File, l : Label {
        create[f]; delete[f]; empty; affix[f,l]
    }
} expect 0

run Scenario4 {
    some disj f1,f2 : File, l : Label {
        create[f1]; create[f2]; delete[f1]; affix[f2,l]; delete[f2]; affix[f1,l]; empty
    }
} expect 1

run Scenario5 {
    some disj f1,f2 : File, l : Label {
        create[f1]; delete[f1]; affix[f1,l]; empty; create[f2]
    }
} expect 0
```

Filesystem v2

DEMO

Filesystem v3

- Allow labelling when accessible or trashed
- Clear labels when trash is emptied
- Affix special label Trashed when file is deleted
- Detach special label Trashed when file is restored

Filesystem v3

```
one sig Dirty extends Label {}

fact Synchronization {
  // allow labelling when accessible or trashed
  all f : File, l : Label | always (affix[f,l] implies f in Accessible+Trashed)

  // clear labels when trash is emptied
  always {
    empty implies after ((some f : File-Accessible | clear[f]) until no (File-Accessible).labels)
  }

  // affix label Trashed after delete
  all f : File | always (delete[f] and Dirty not in f.labels implies after affix[f,Dirty])

  // detach label Trashed after restore
  all f : File | always (restore[f] and Dirty in f.labels implies after detach[f,Dirty])
}
```

Filesystem v3

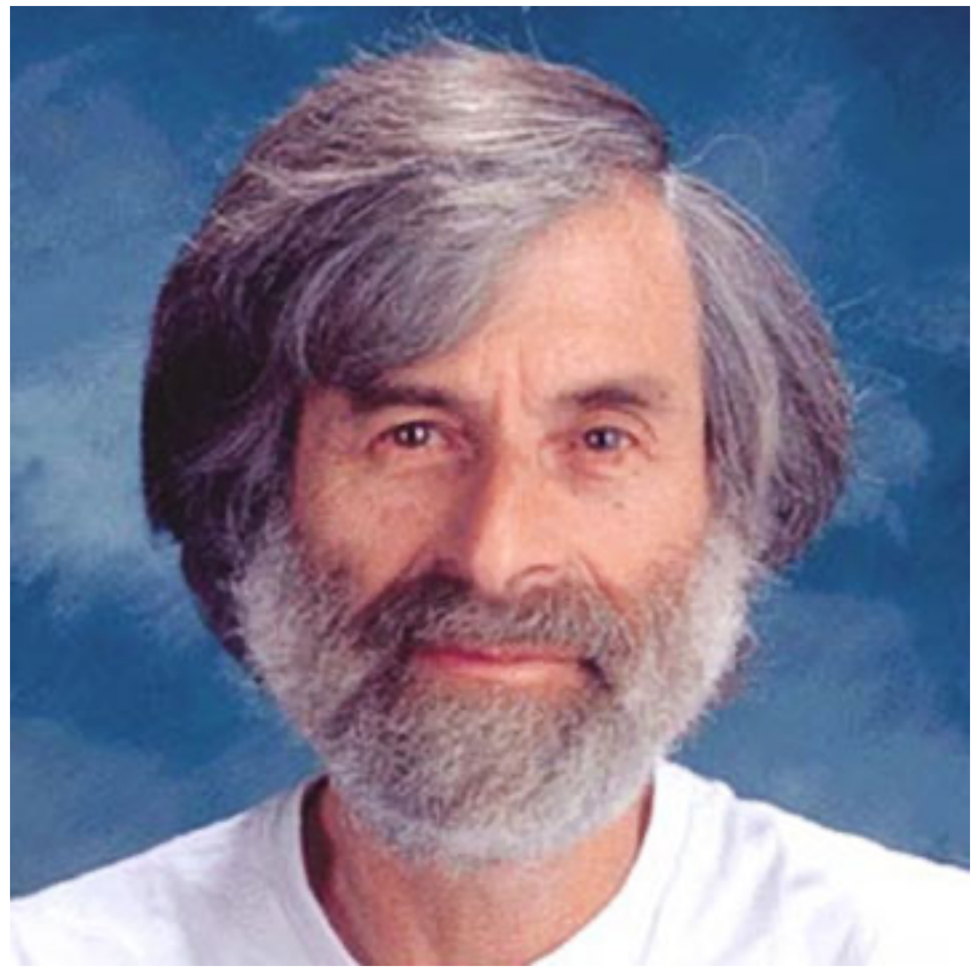
DEMO

Epilogue

"Software is built on abstractions. Pick the right ones, and programming will flow naturally from design [...] **Pick the wrong ones, and programming will be a series of nasty surprises.**"

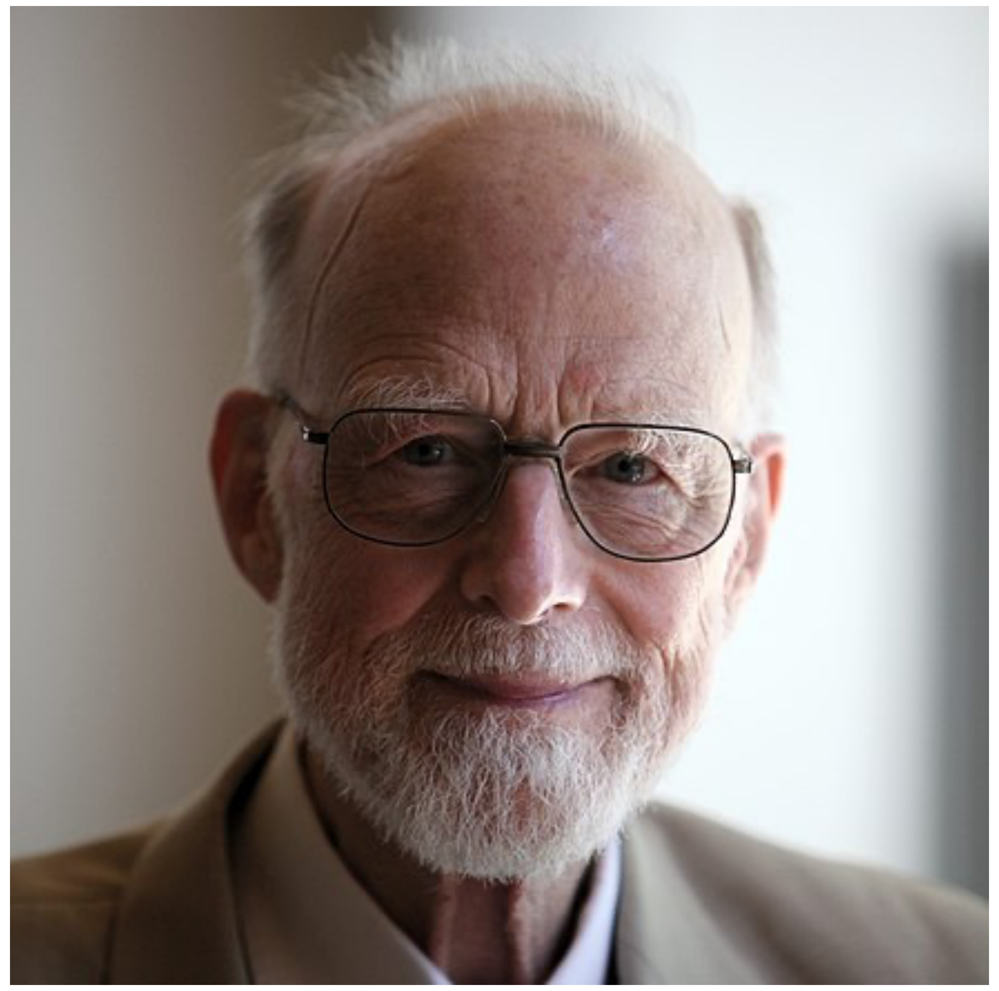
–Daniel Jackson

“A specification is an *abstraction*. [...] But I don’t know how to teach you about abstraction. A good engineer knows how to abstract the essence of a system and suppress the unimportant details when specifying and designing it. **The art of abstraction is learned only through experience.**”



–Leslie Lamport

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. **The first method is far more difficult.**”



–Tony Hoare

Epigram 31

“Simplicity does not precede complexity, but follows it.”

–Alan Perlis

always eventually some Alloy