

App design with Alloy

Alcino Cunha

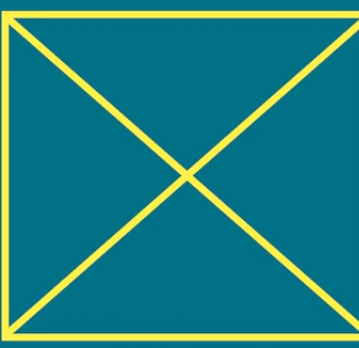
“I mean ‘design’ here in the same sense that the word is used in other design disciplines: the shaping of some artifact to meet a human need. [...] For software, that means determining what the behavior of the software should be: what controls it will offer, and what responses it will provide in return.”



-Daniel Jackson



THE
ESSENCE
OF
SOFTWARE



WHY CONCEPTS
MATTER FOR
GREAT DESIGN

DANIEL JACKSON

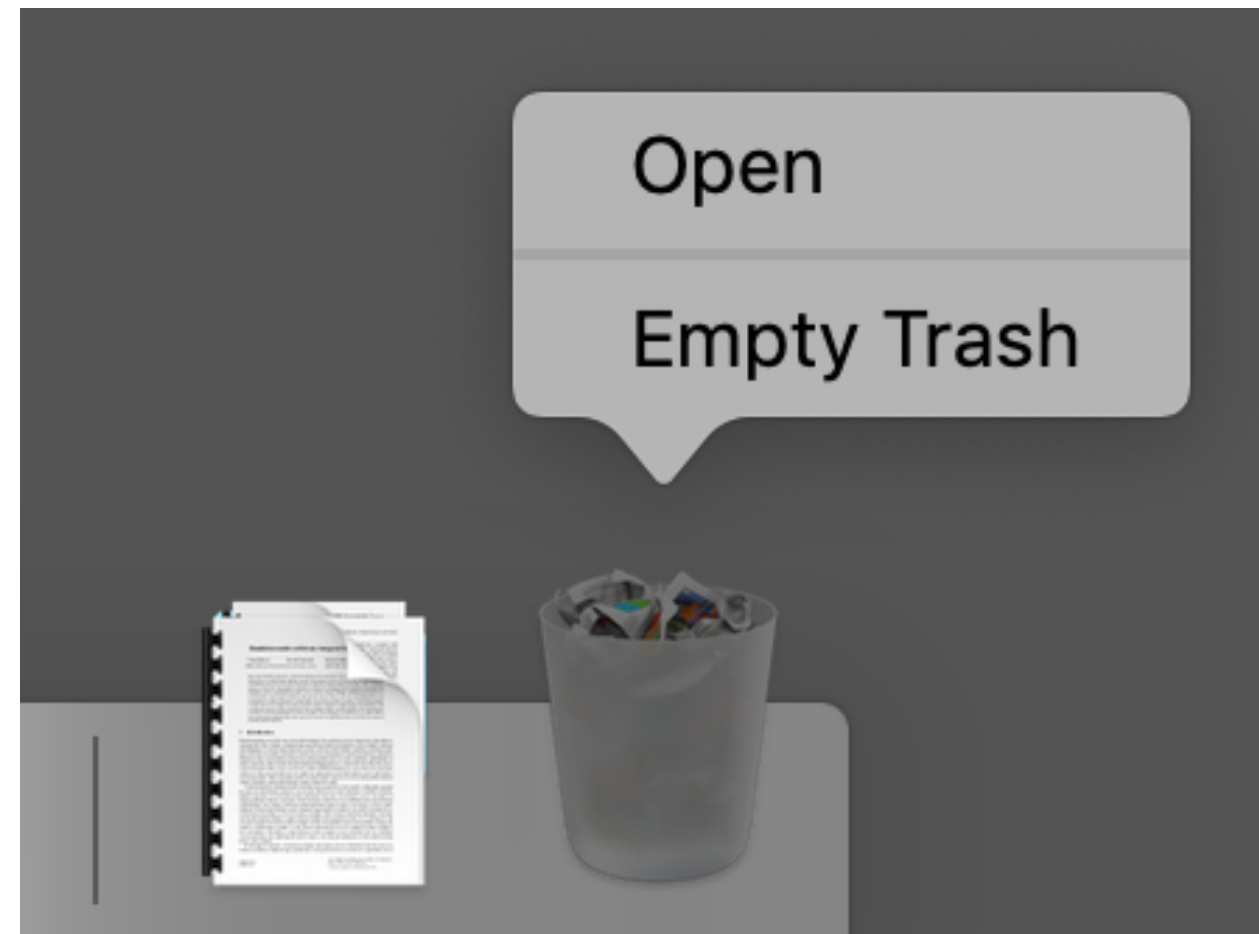


Concepts

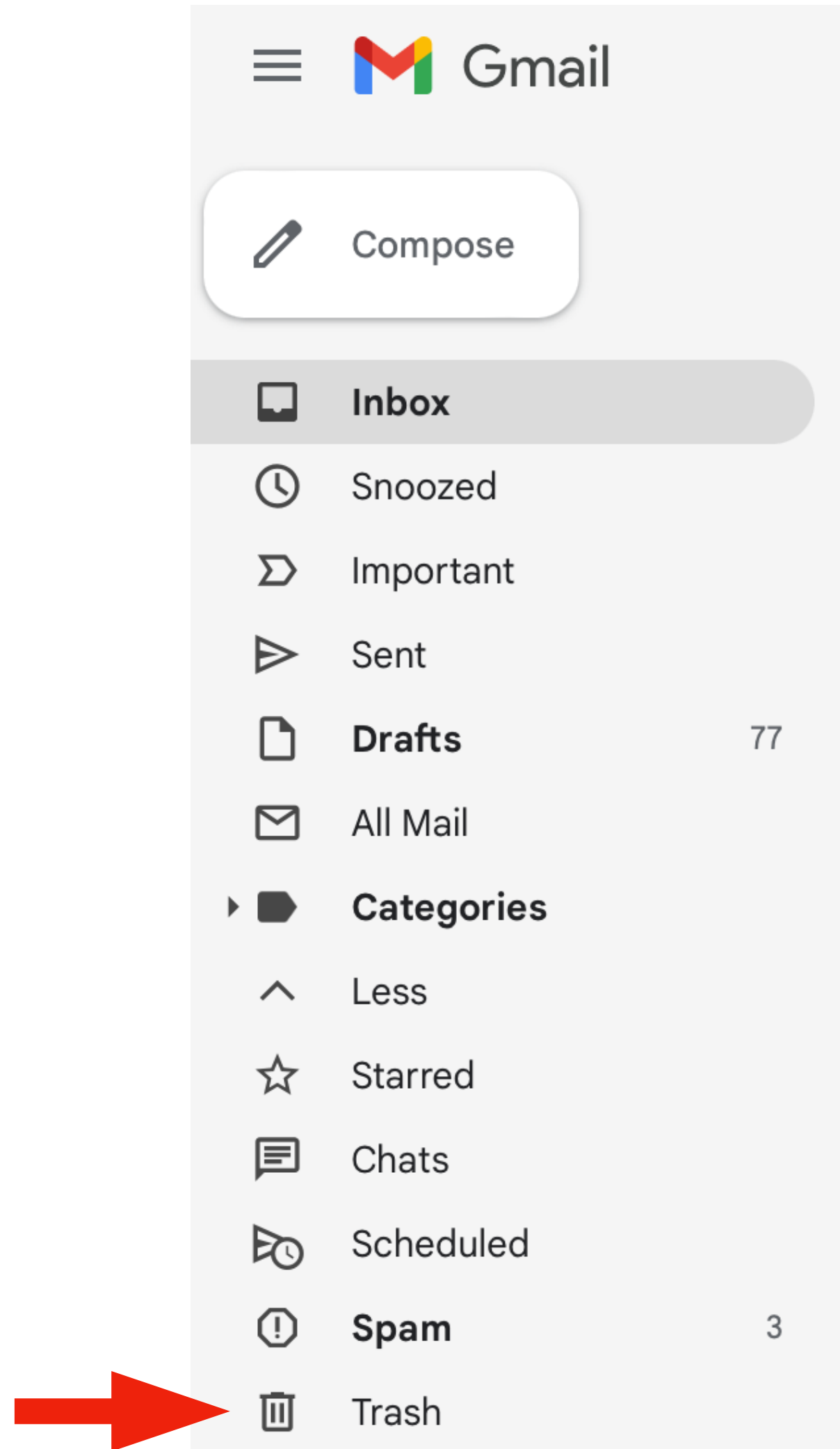
- Apps are made of *concepts*
- Each *concept* is a self-contained unit of functionality with a clear purpose
- *Concepts* work together to provide the app overall functionality
- But can be understood independently of one another

Trash

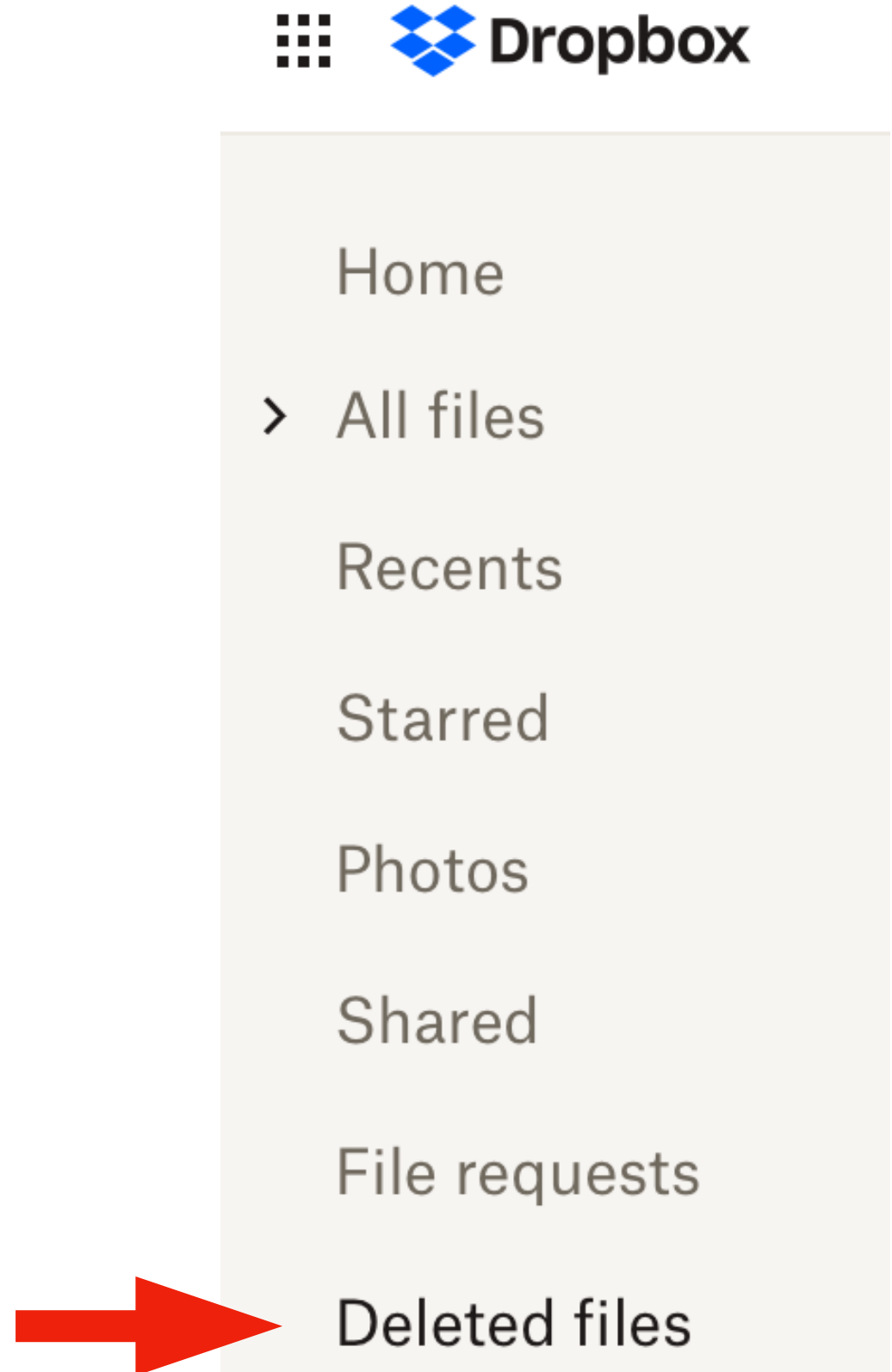
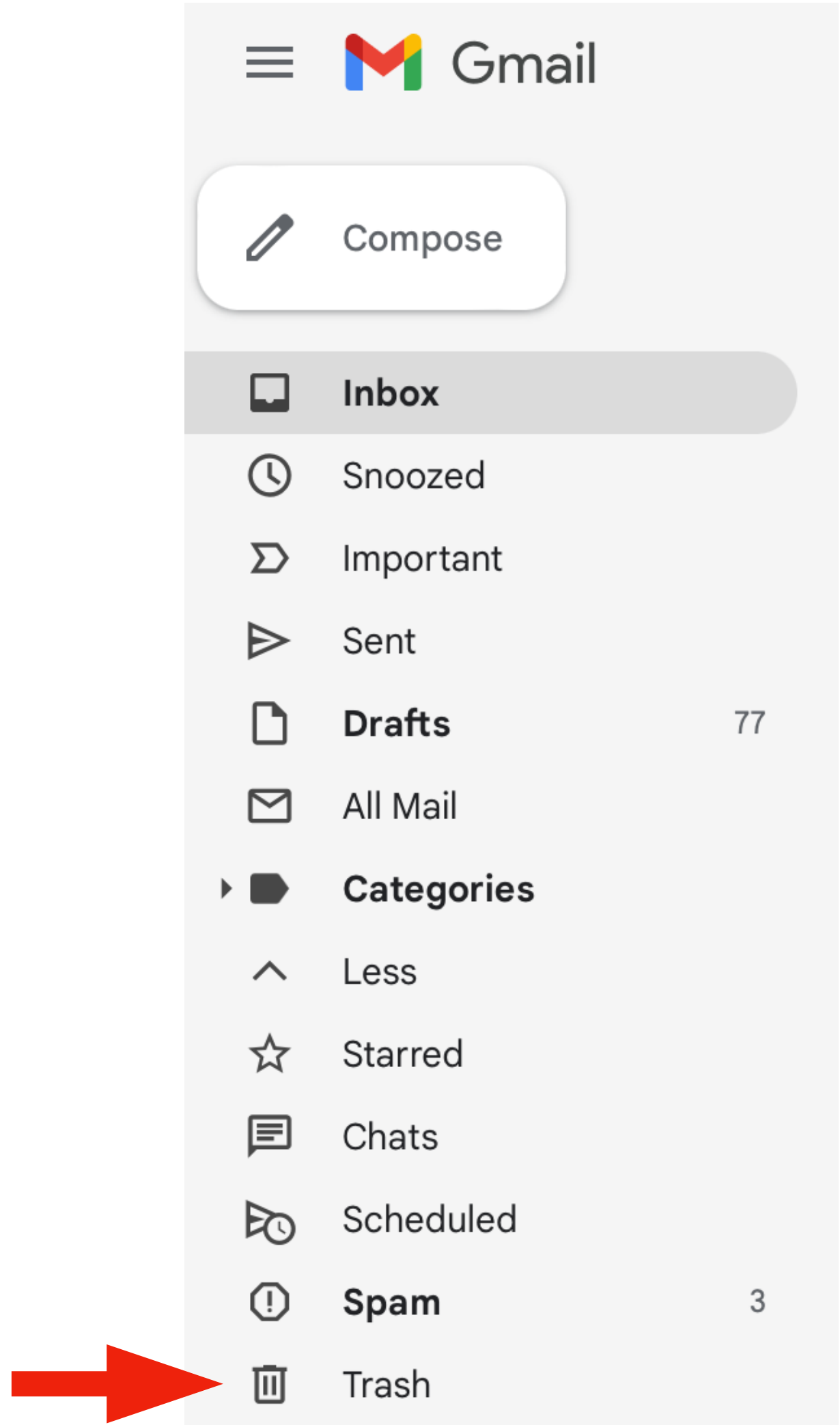
Trash



Trash

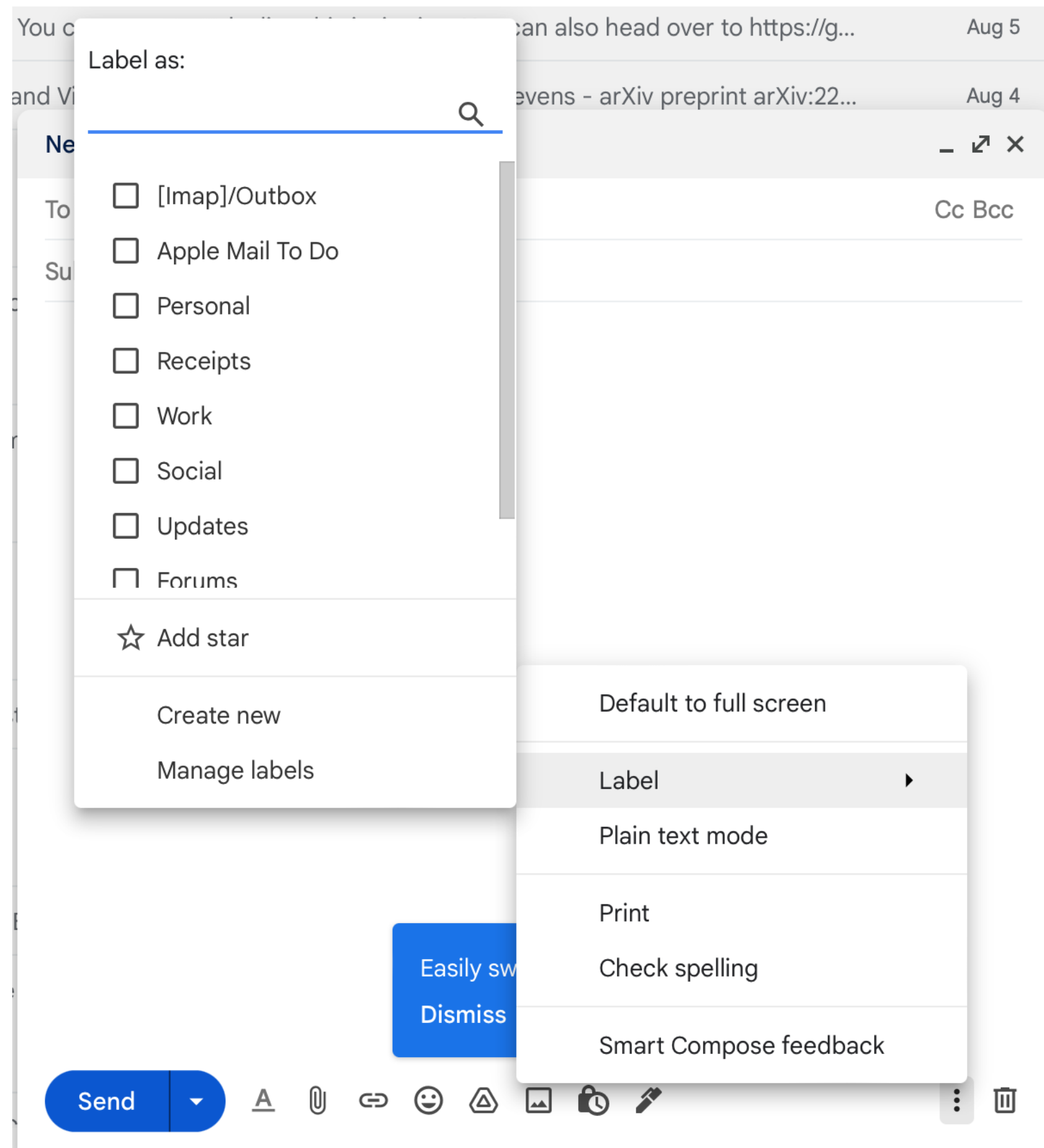


Trash

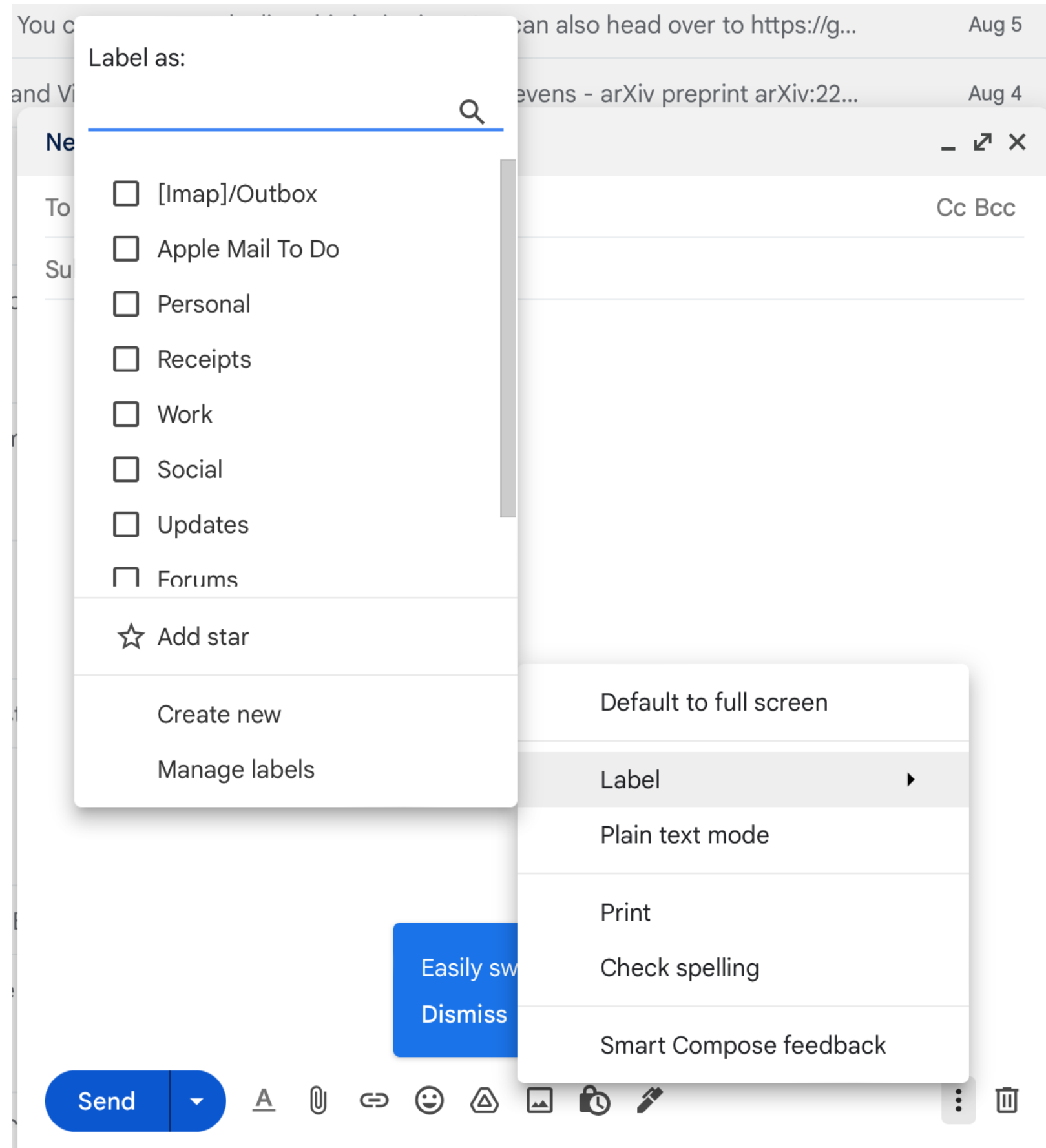


Label

Label



Label



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

This screenshot shows an email client interface. A 'Label as:' dropdown menu is open, displaying a search bar and a list of labels: [imap]/Outbox, Apple Mail To Do, Personal, Receipts, Work, Social, Updates, and Forums. Below the list are options for 'Add star', 'Create new', and 'Manage labels'. A secondary menu is also visible, containing options like 'Default to full screen', 'Label', 'Plain text mode', 'Print', 'Check spelling', and 'Smart Compose feedback'. At the bottom, a blue 'Send' button is visible along with various icons for attachments, links, emojis, and other email functions.

Edit 1 photo

This screenshot shows a photo editing interface. At the top, it says 'Edit 1 photo'. Below this, there is a 'Location' section with a text input field containing 'Ankobra beach' and a close button (X). Underneath is a 'Keywords' section with an information icon (i) and a text input field with the placeholder 'Type your own keywords here'. Below the input field are several keyword tags, each with a close button (X): Beach, No People, Outdoors, Sea, Sand, Palm Tree, Sunset, Water, Twilight, Ghana, and Africa. At the bottom, there is a 'Suggested keywords' section with several tags, each with a plus sign (+): Vertical, Sky, Cloud - Sky, Scenics - Nature, Tree, Nature, Beauty In Nature, and Photography.

This photo is in 1 album



Japan

377 items

Tags ?

Add tags

Japan

Tokina AT-X 124


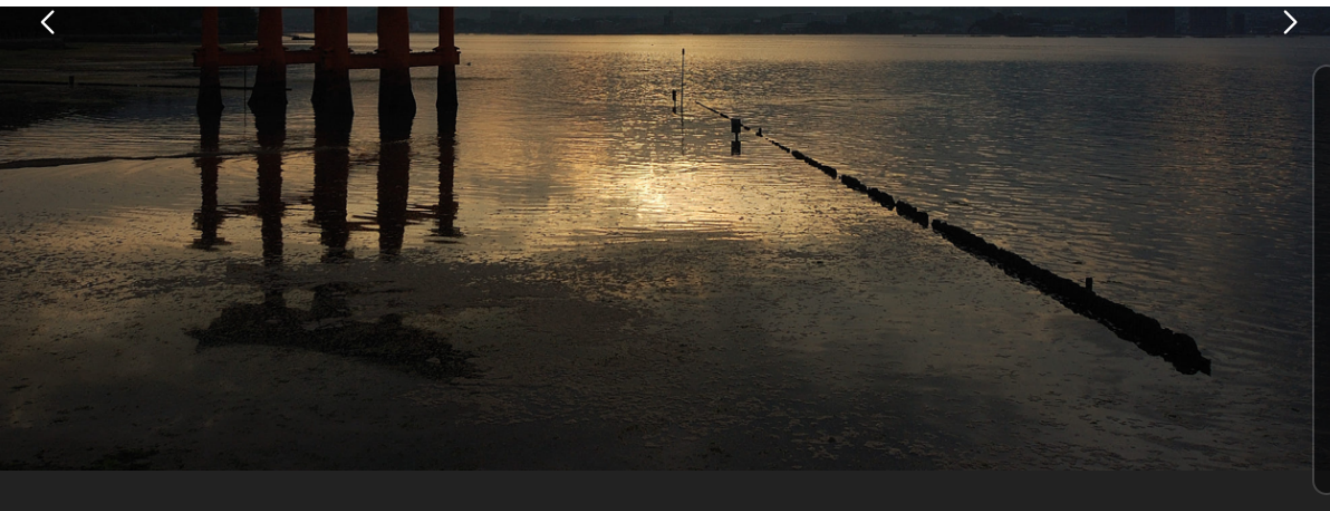

Miyajima

torii

sunset

Like

Like





one last postcard from miyajima
by Alcino Cunha >

Taken: Aug 17, 2008 **Uploaded:** almost 11 yrs ago

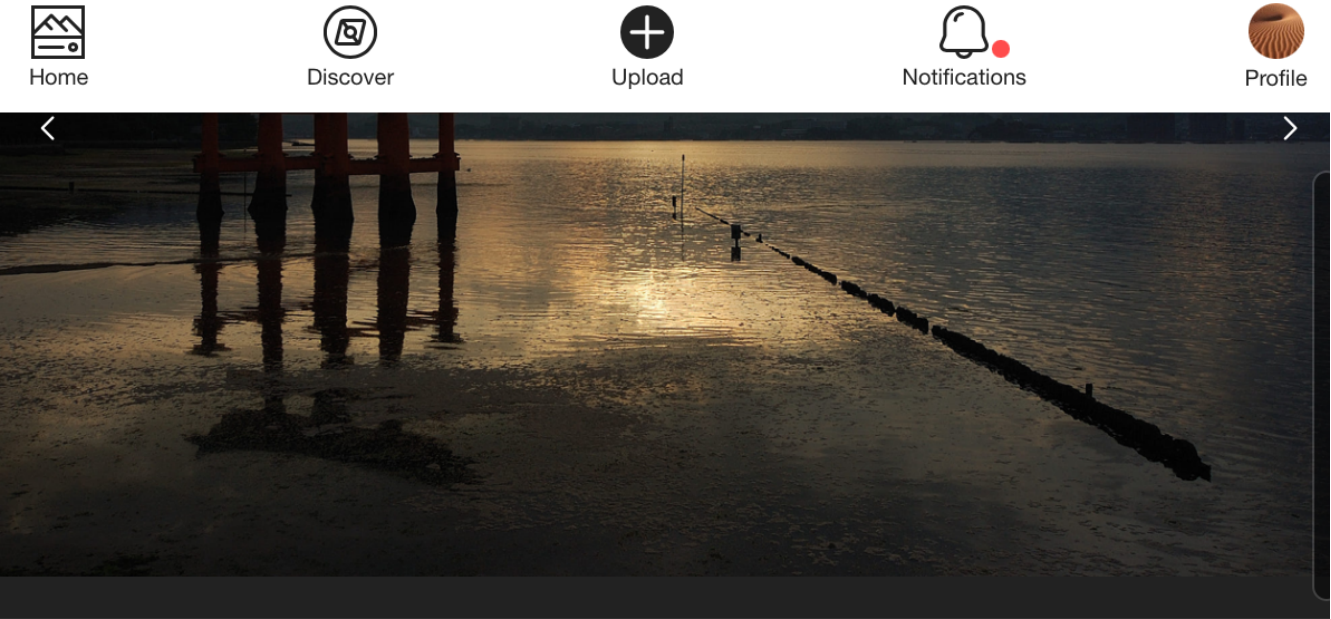
Miyajima, Japan [2008]

Pulse ⓘ Impressions ⓘ Fresh ⓘ

47.4 **9.1K** 

63 people liked this photo > 


Like



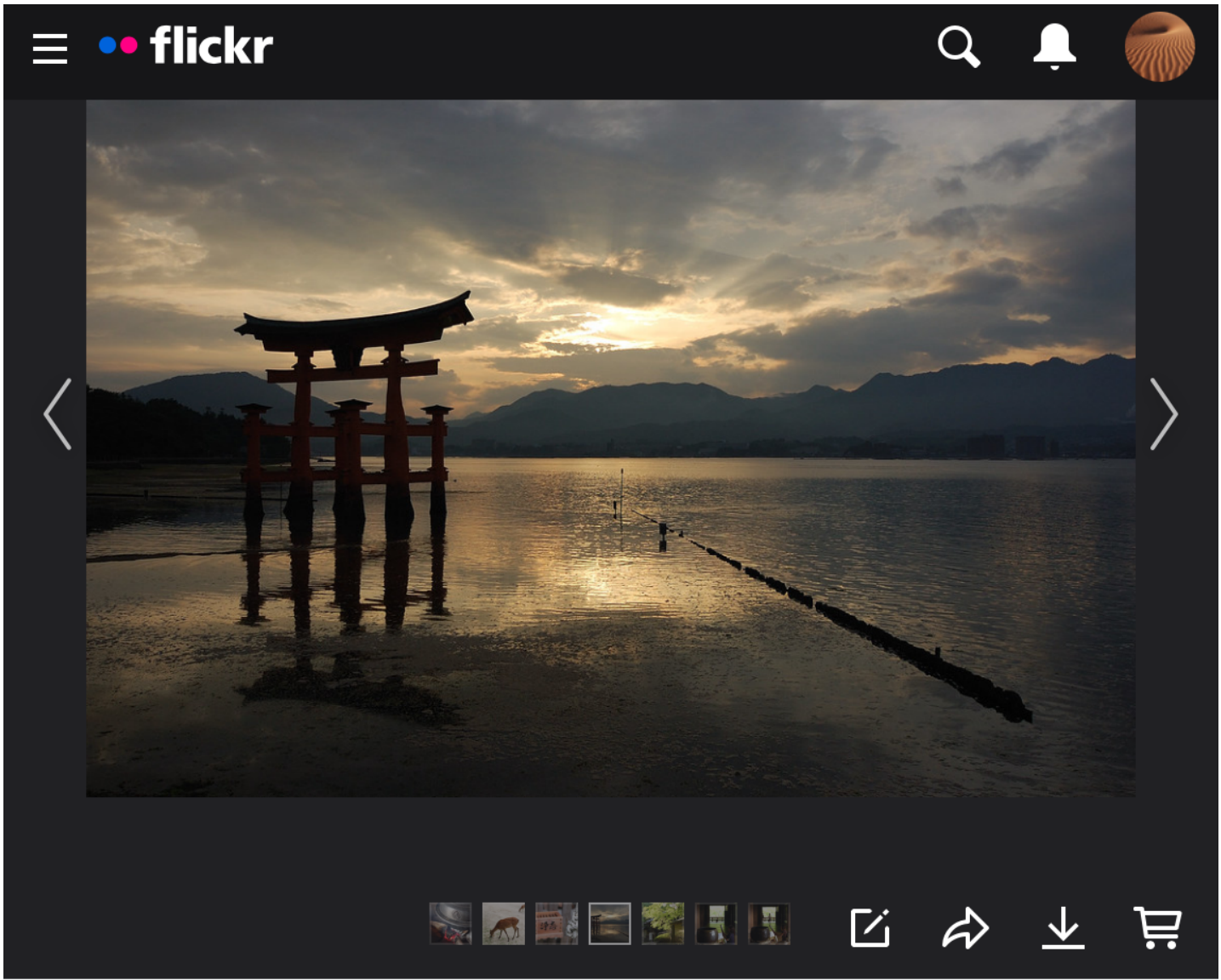
 **one last postcard from miyajima**
by Alcino Cunha >



Taken: Aug 17, 2008 Uploaded: almost 11 yrs ago


Miyajima, Japan [2008]

Pulse ⓘ Impressions ⓘ Fresh ⓘ
47.4 **9.1K** 

63 people liked this photo >

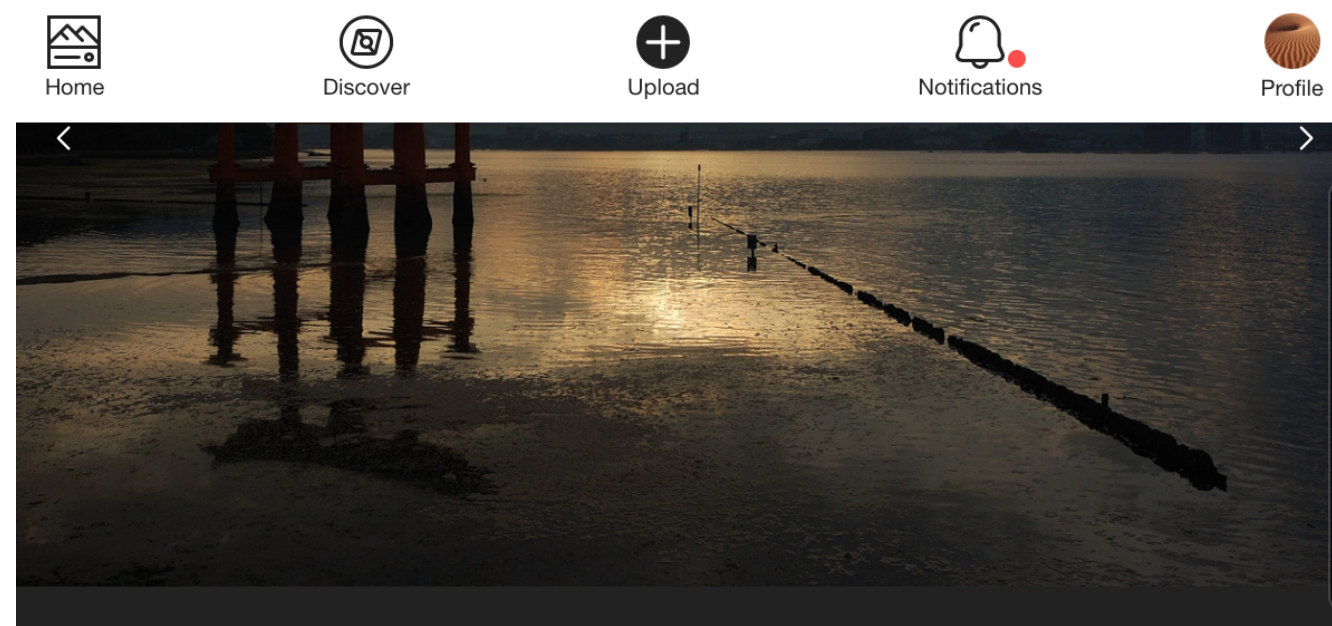


 **Rosino**
one last postcard from miyajima 
Miyajima, Japan [2008]

 **thowe 62, Mike and 5 more people** faved this



Like



 **one last postcard from miyajima**
by Alcino Cunha >

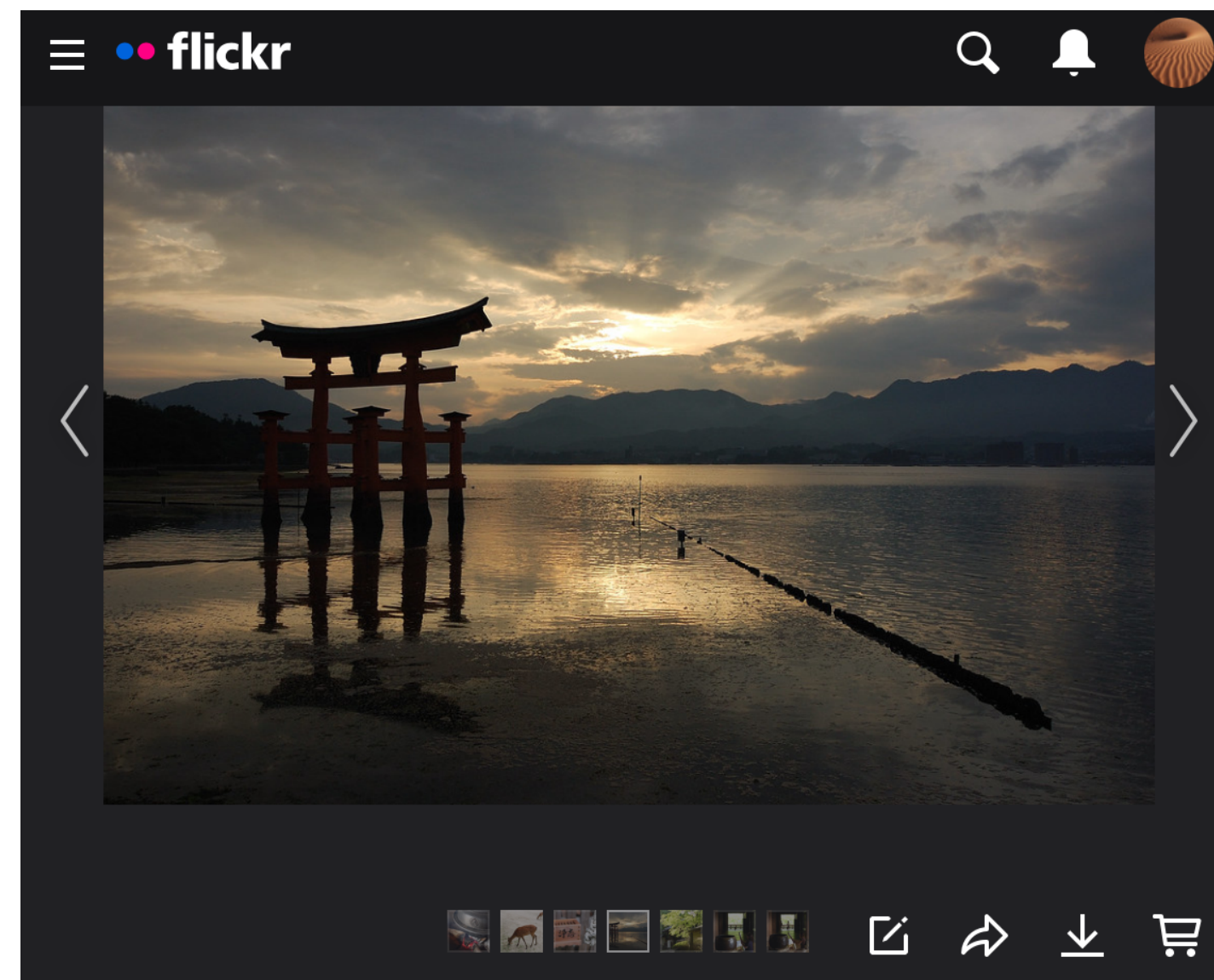
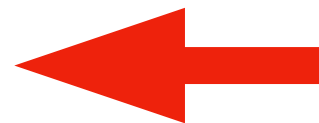
Taken: Aug 17, 2008 **Uploaded:** almost 11 yrs ago

Miyajima, Japan [2008]

Pulse ⓘ Impressions ⓘ Fresh ⓘ

47.4 **9.1K** 

63 people liked this photo >



Rosino

one last postcard from miyajima

Miyajima, Japan [2008]



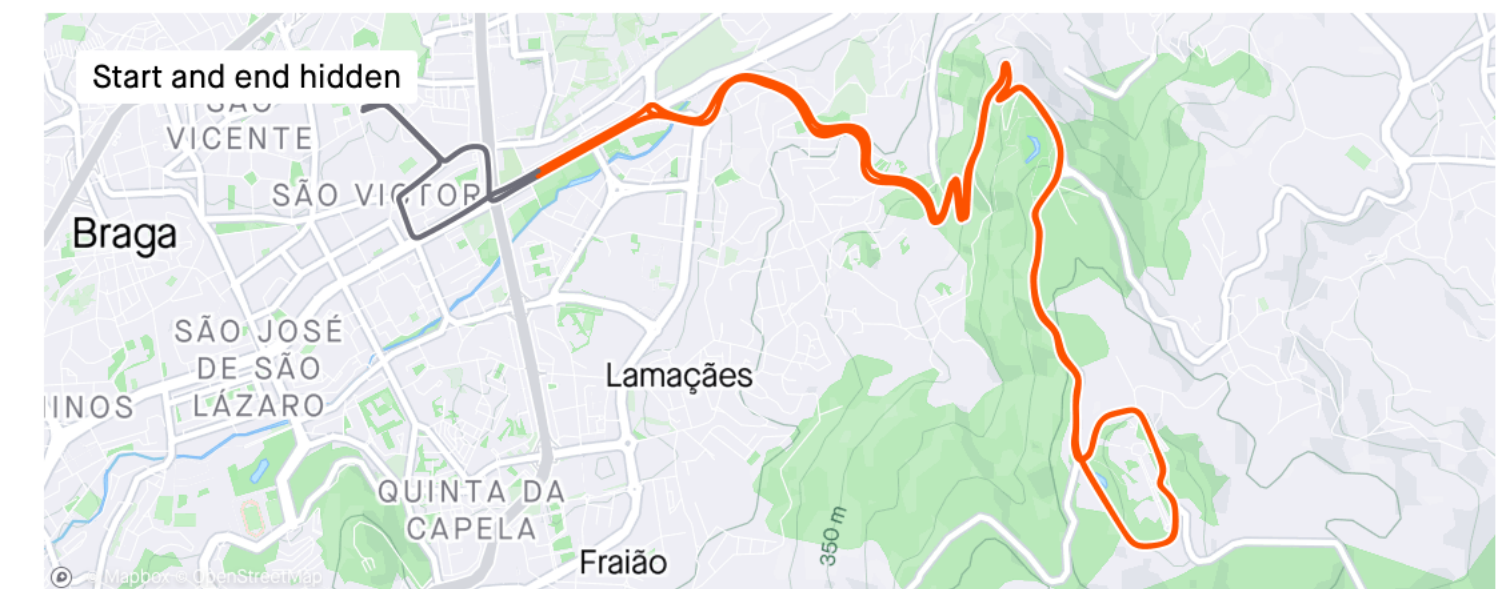
 **thowe 62, Mike and 5 more people** faved this




Despertar

Distance	Elev Gain	Time	Achievements
30.01 km	794 m	1h 18m	 5

Parque lago - Rotunda do Papa **PR** (4:25)



 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

“[...] For software, that means determining what the behavior of the software should be: what controls it will offer, and what responses it will provide in return. These questions have no right or wrong answers, only better or worse ones.”

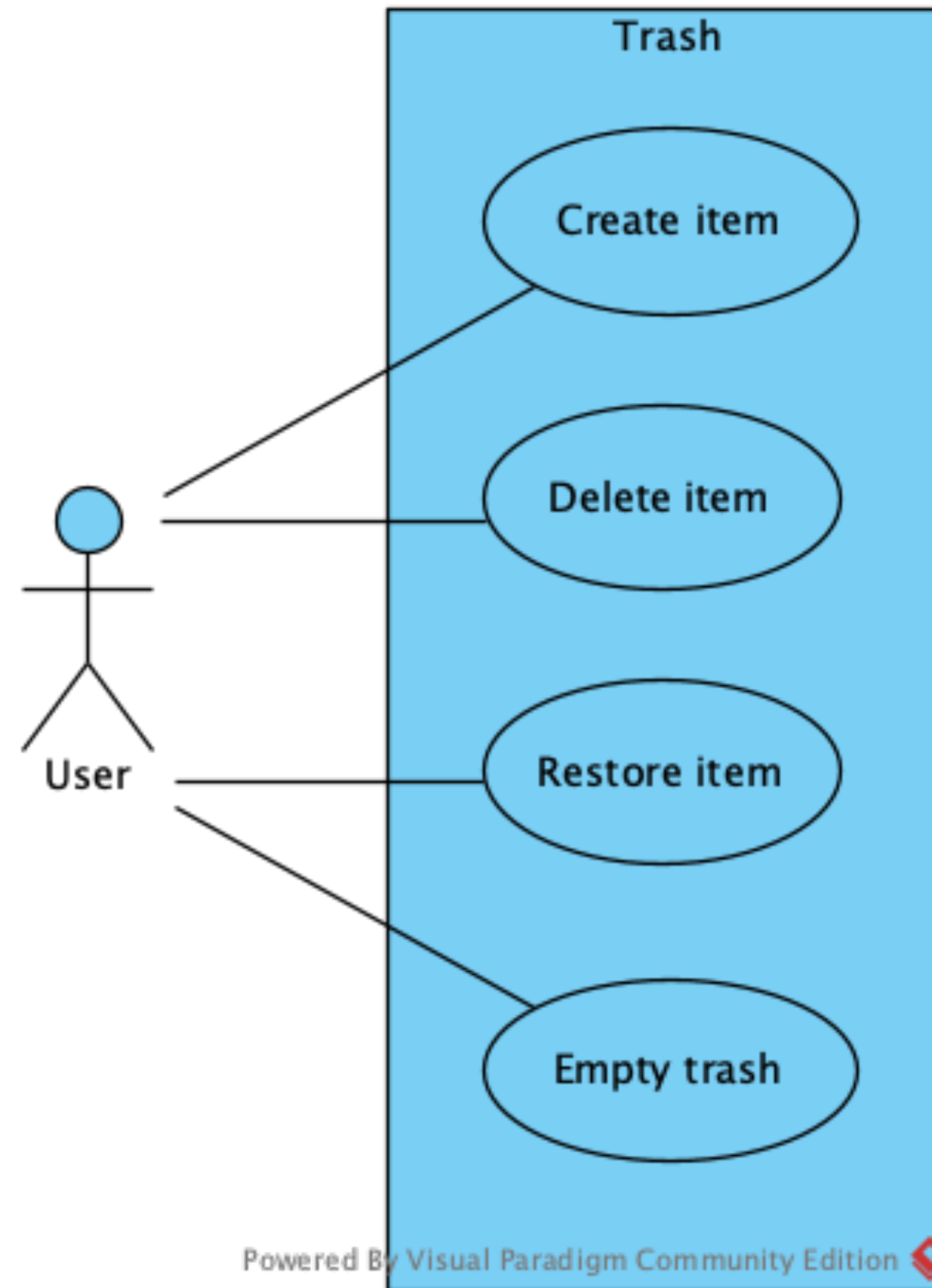


-Daniel Jackson

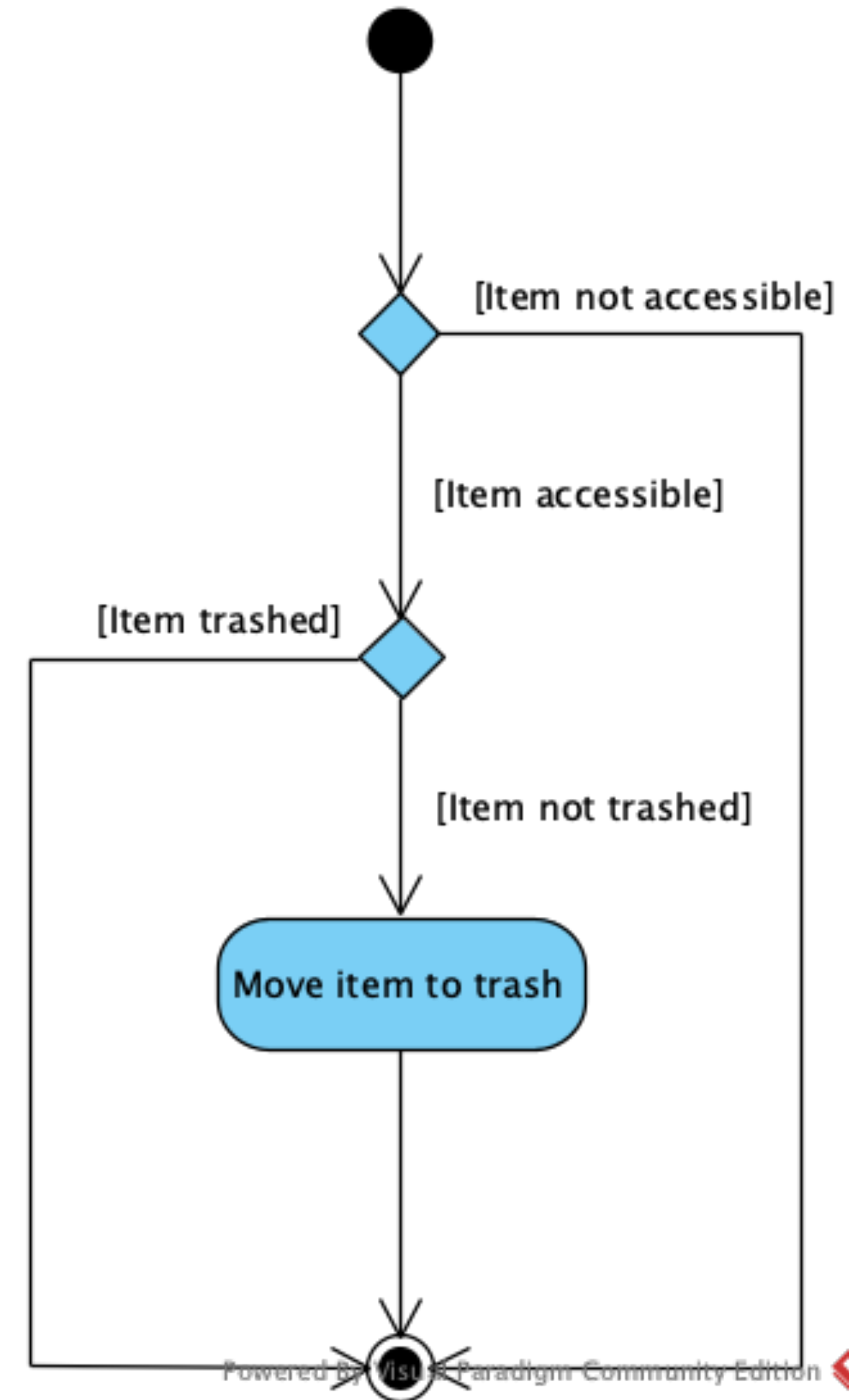
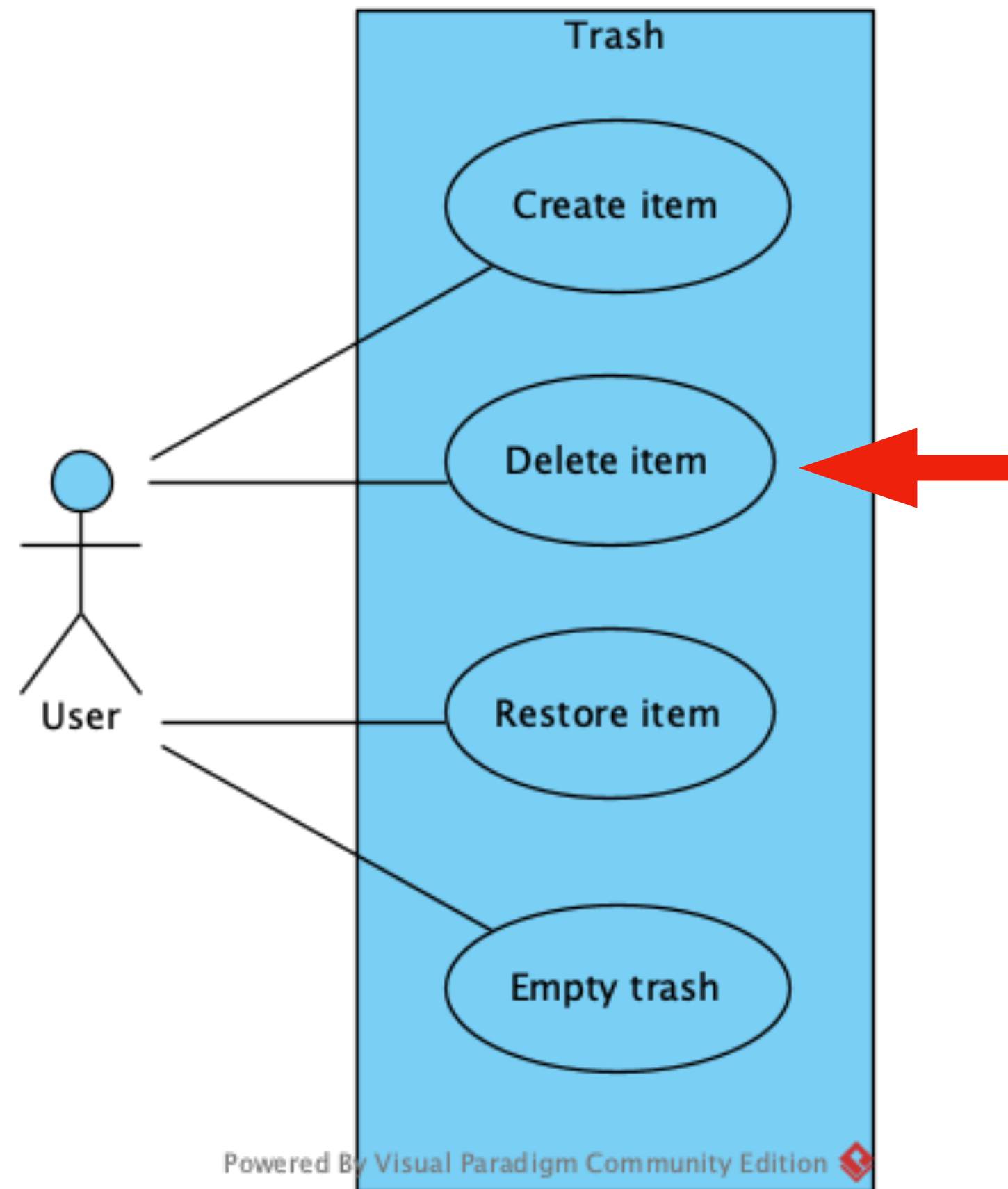
Modeling *Concepts*

Trash modeling *a la* UML

Trash modeling *a la* UML



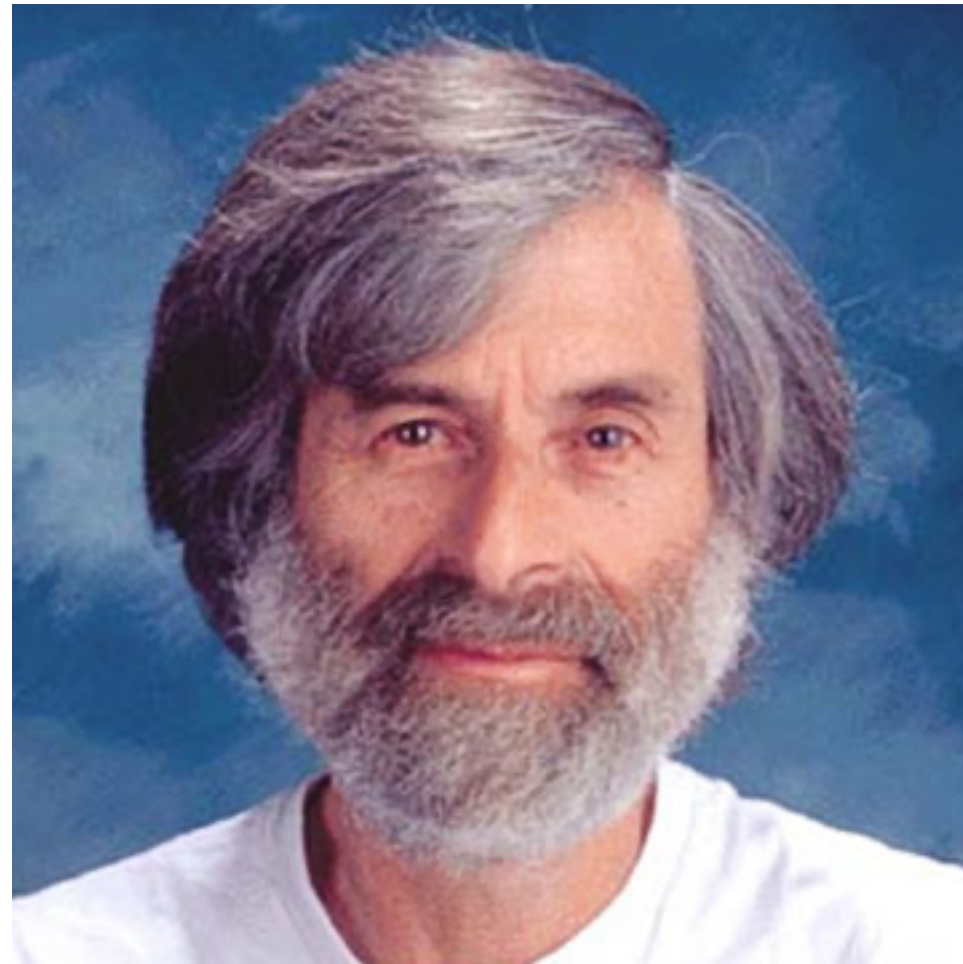
Trash modeling *a la* UML



Trash “modeling” with code

```
class Trash<Item> {  
  
    private HashSet<Item> accessible;  
    private HashSet<Item> trashed;  
  
    void delete(Item i) throws Exception {  
        if (!accessible.contains(i)) {  
            throw new Exception("Item not accessible");  
        }  
        if (trashed.contains(i)) {  
            throw new Exception("Item already trashed");  
        }  
        trashed.add(i);  
        accessible.remove(i);  
    }  
  
    ...  
}
```


“If you’re not writing a program, don’t use a programming language.”



-Leslie Lamport

Trash modeling *a 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 *a 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
 - Archetypical scenarios that show how the concept is fulfilled by the actions

Transition systems

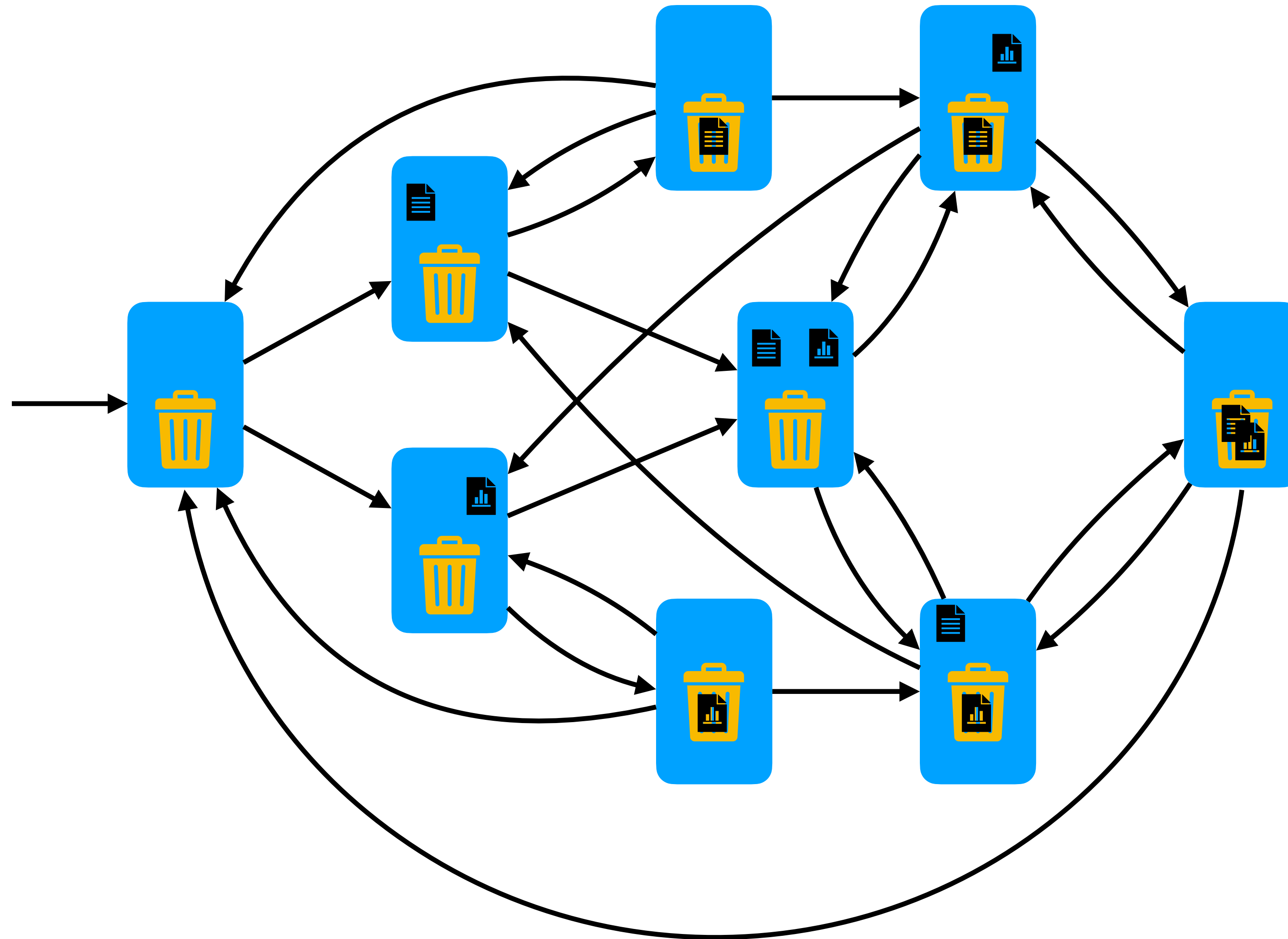
Transition systems

- A popular model to describe the behavior of a system
- In some areas a *model* is 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
 - ...

Transition system = States + Transitions

- 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 valid path (a sequence of states) in the transition system, starting in an initial state

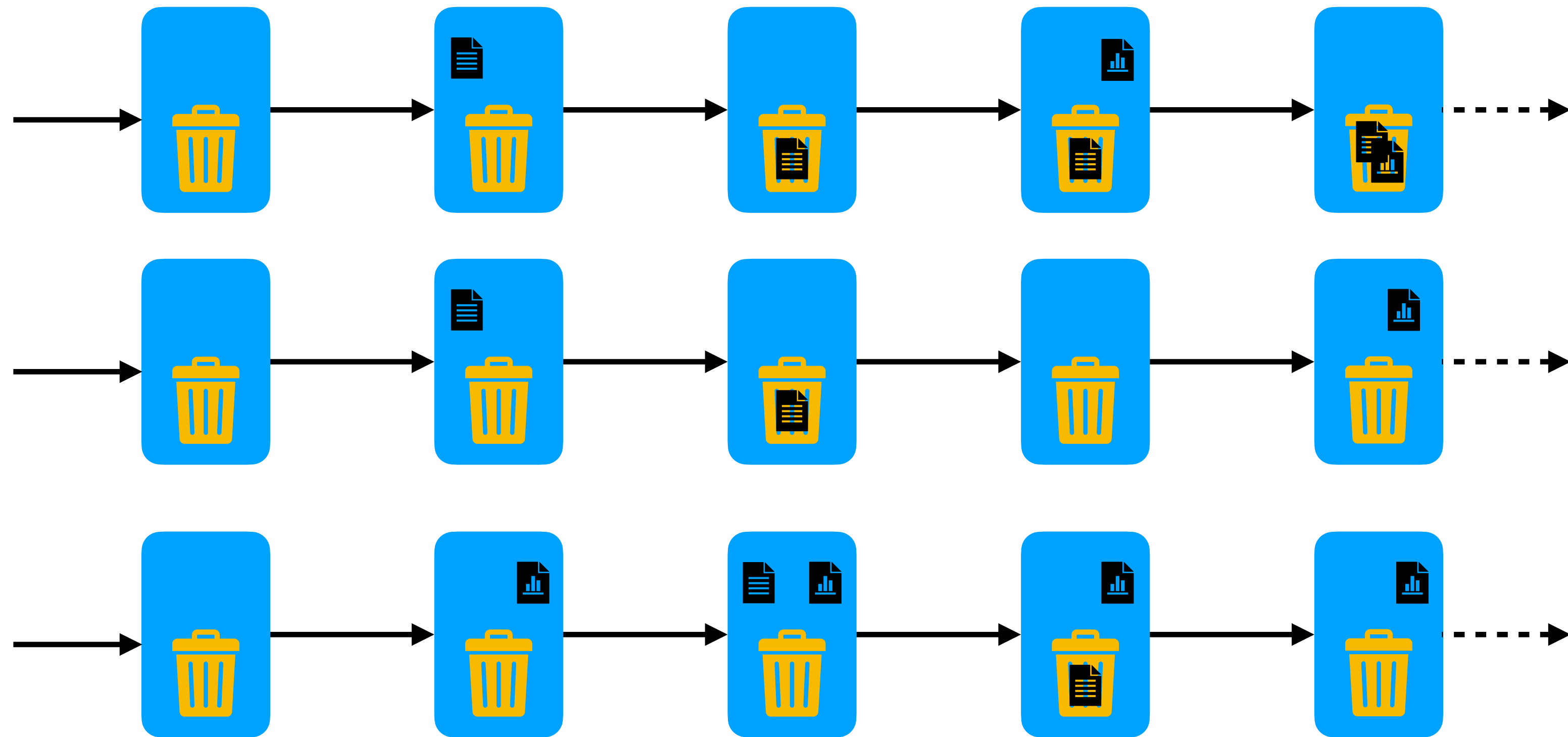
Trash transition system



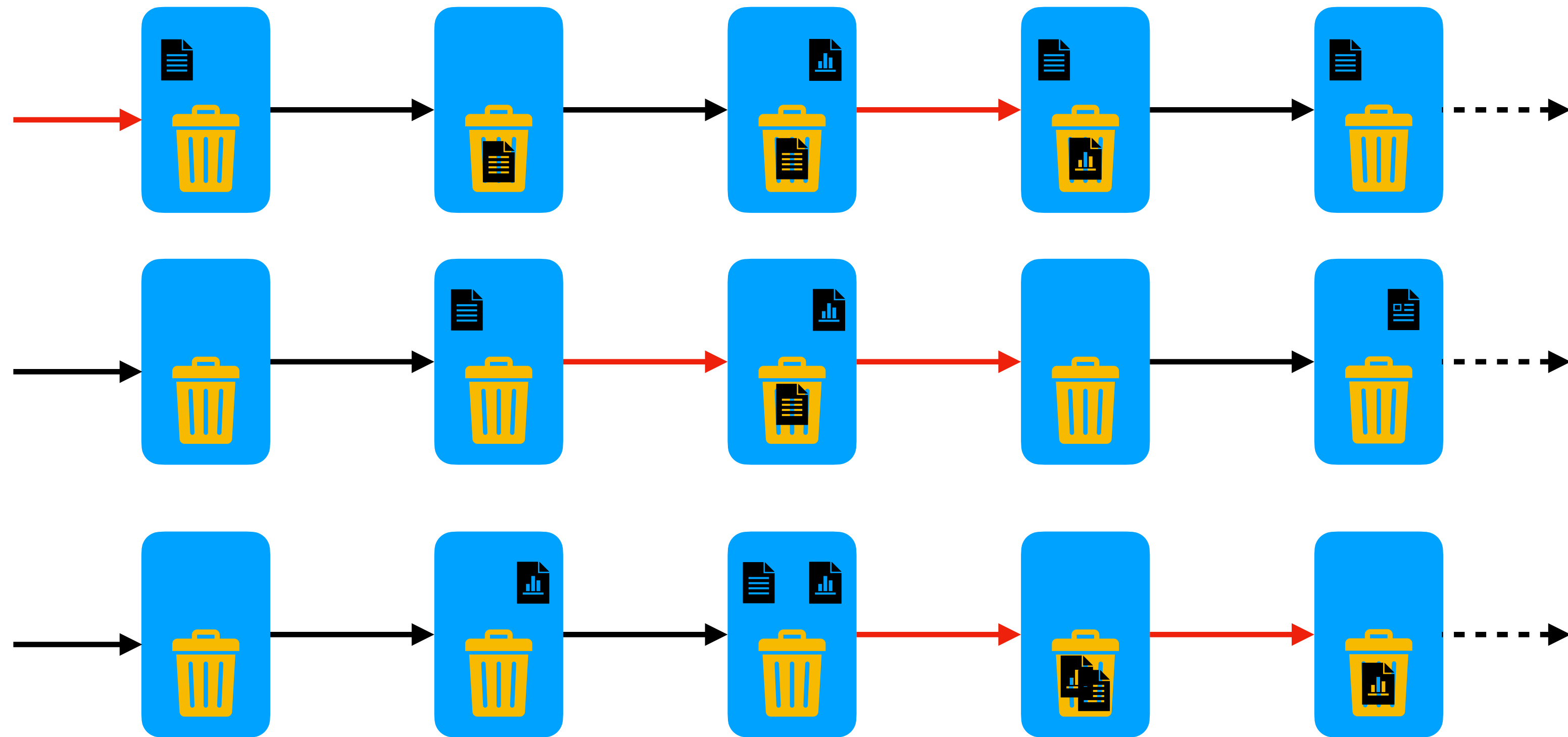
Declarative modeling

- It is possible to describe a transition system by specifying which traces are valid
- The specification talks about a sequence of states and not just about a single state
- And thus requires some sort of *temporal logic*

Valid trash traces



Invalid trash traces



Transition systems in Alloy

Mutability

- In Alloy 6 mutable signatures and fields can be declared with keyword **var**
 - Previously only possible with the Electrum extension
 - It was possible to describe behavior in Alloy 5 by explicitly modeling the concept of state
 - But it was confusing and error prone
- Mixing mutable and static structures
 - Static field inside mutable signature yields a warning
 - Same for static signature extending or inside mutable one

Trash states

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

Instances

- When mutable structures are declared, instances are **infinite traces**
- Analysis commands only return traces that can be represented finitely
 - Instances are traces that loop back at some point
- Static signatures and fields 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
- They allow us to “quantify” the validity of a formula over the different states of a trace
- A formula without temporal operators is only required to hold in the initial states
- Alloy 6 has both future and past temporal operators

Always, historically, and prime

always ϕ

ϕ will always be true

historically ϕ

ϕ was always true

R'

The value of R in the next state

Trash behavior

```
fact Behavior {  
  // Initial state  
  no Accessible  
  no Trashed  
  // Transitions  
  always {  
    // At most one item is created or deleted  
    lone (Accessible - Accessible') + (Accessible' - Accessible)  
    // All deleted items go to the trash  
    Accessible - Accessible' = Trashed' - Trashed  
    // If no item was deleted or created an empty must have occurred  
    Accessible' = Accessible implies no Trashed'  
    // ...  
  }  
}
```



Trash behavior

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

Anatomy of an action

- The specification of an action is a conjunction of three kinds of formulas
 - *Guards*, that specify when it can occur
 - *Effects*, that specify what changes when it occurs
 - *Frame conditions*, special effects that specify what does not change when it occurs
- Guards usually have no temporal operators
 - But can use past temporal operators to recall something about the past
- Effects and frame conditions use the prime operator to specify the relation between the present and the next value of mutable signatures and fields
 - If nothing is specified about a mutable signature or field it can change freely

Create item

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

Delete item

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

Restore item

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

Empty trash

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


Validation

Validation

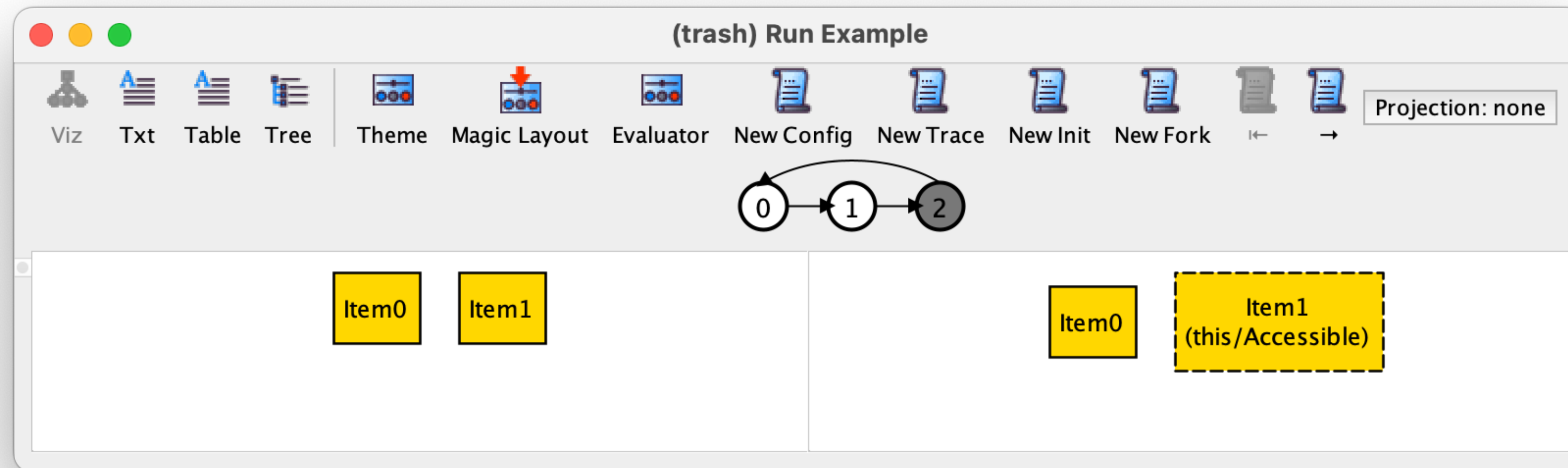
- As usual, **run** commands can be used to validate the model
- The scope of a mutable signature defines the maximum number of different atoms in the full trace, not a maximum per state

Trace visualization

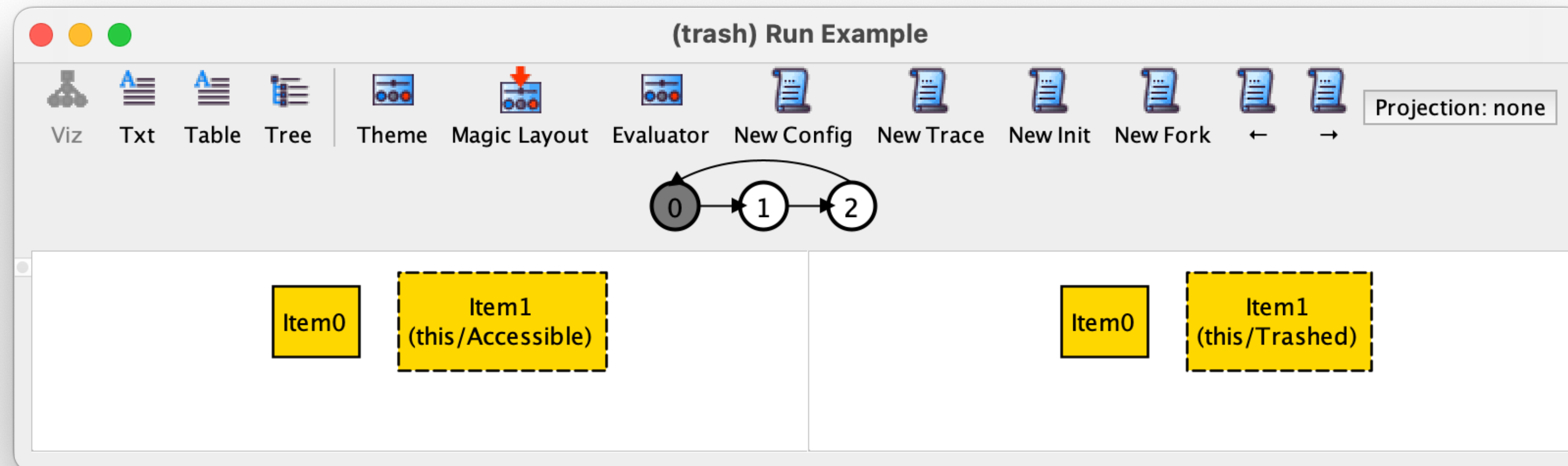
- The visualizer depicts two consecutive states of the trace side-by-side
 - By default mutable structures are depicted with dashed lines
- A representation of the infinite trace is shown above
 - Different states have different numbers
 - The loop back is explicitly depicted
 - Clicking on a state focus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

Trace visualization

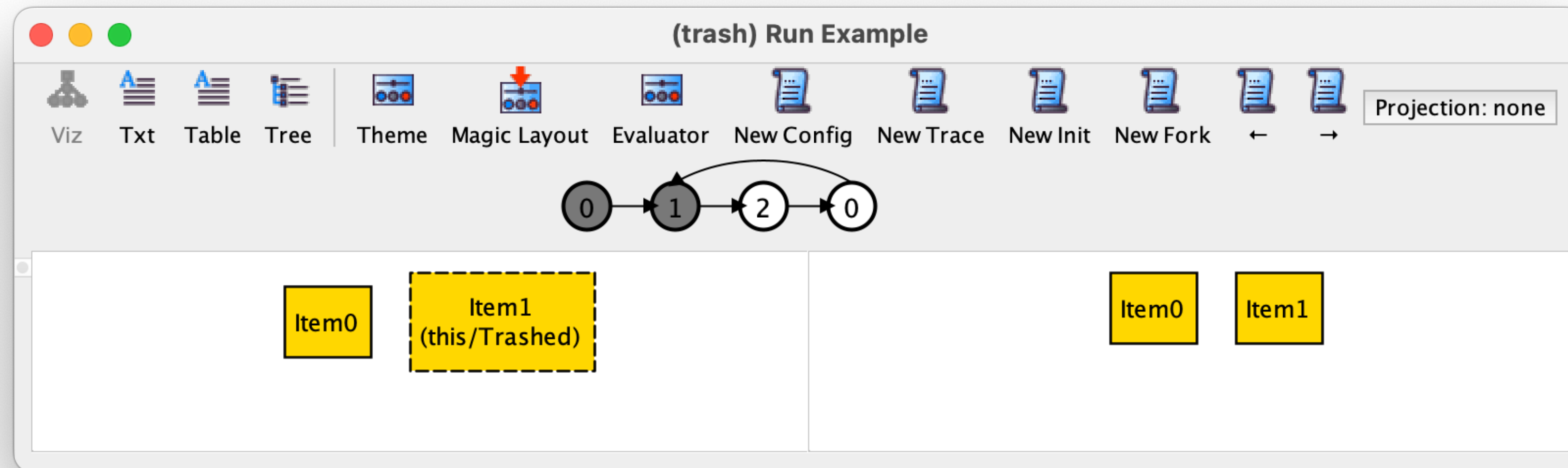
Trace visualization



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 to the immutable structures)
 - *New trace*, returns any different trace with the same configuration
 - *New init*, returns a trace with the same config, but a different initial state
 - *New fork*, returns a trace with the same prefix, but a different next state

Simulation



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

$\phi ; \psi$

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
```

Some trash scenarios

Executing "Run Scenario1 expect 1"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF Mode=batch
2..8 steps. 17099 vars. 378 primary vars. 28707 clauses. 355ms.

Instance found. **Predicate** is consistent, as expected. 76ms.

Executing "Run Scenario2 expect 0"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
2..10 steps. 51735 vars. 998 primary vars. 86980 clauses. 496ms.

No instance found. **Predicate** may be inconsistent, as expected. 20ms.

Executing "Run Scenario3 for 1 Item expect 1"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF Mode=batch
2..10 steps. 74773 vars. 1322 primary vars. 121547 clauses. 199ms.

No instance found. **Predicate** may be inconsistent, contrary to expectation. 7ms.

3 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: No instance found. Scenario3 may be inconsistent, contrary to expectation.



Inconsistency

- This scenario is not possible because it cannot be extended to an infinite trace
- Once the trash is empty no other action is possible and we stated that at every state some action must occur
- At least a *stuttering* action should be possible at that point

Stuttering

A clock specification

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

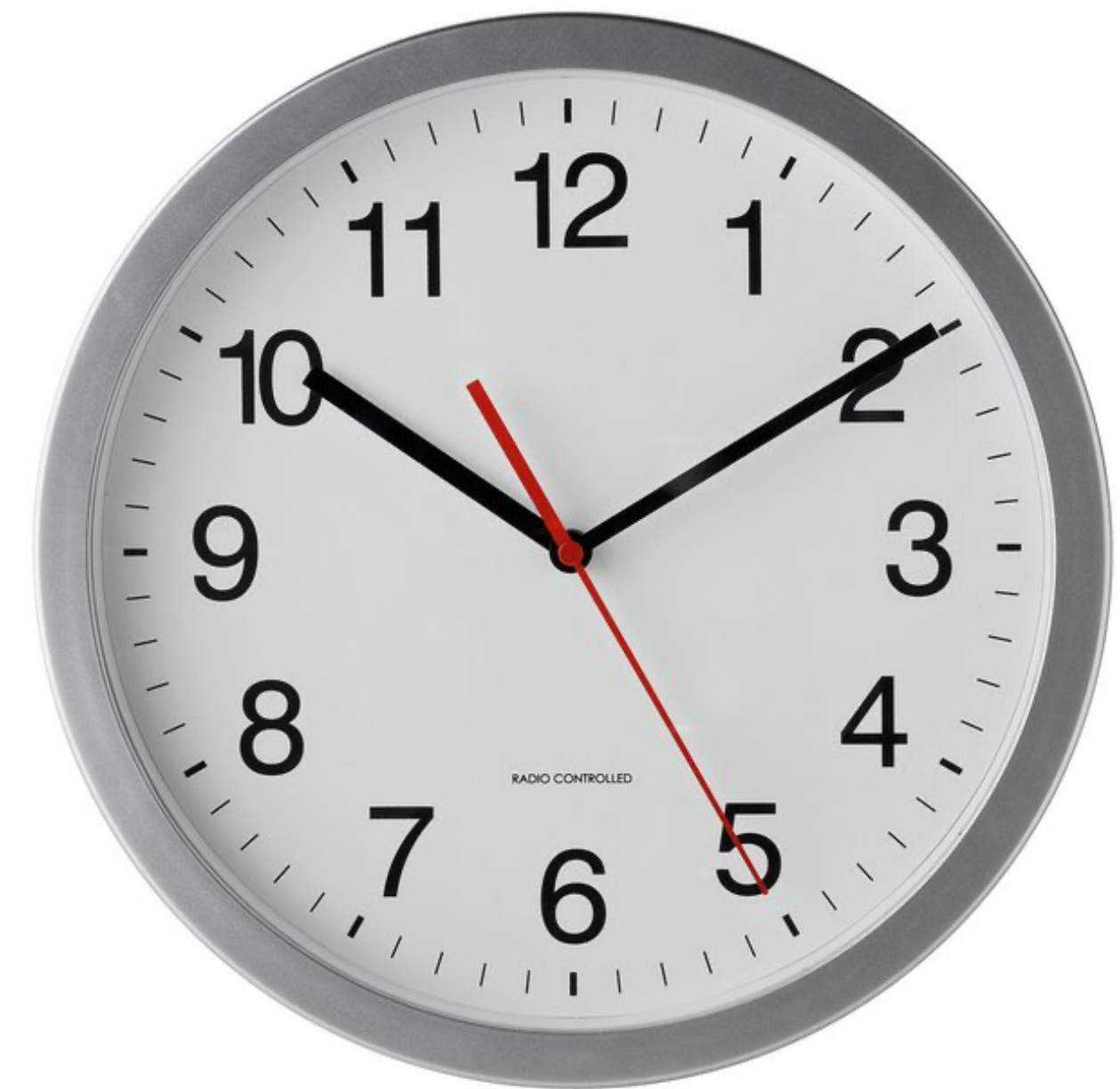


Ceci n'est pas une montre?!

```
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 = 0 and m = 0  
  always {  
    m' = (m+1) % 60 and  
    m=59 implies h' = (h+1) % 12 and  
    m != 59 implies h' = h  
    or  
    m' = m and h' = h  
  }  
}
```

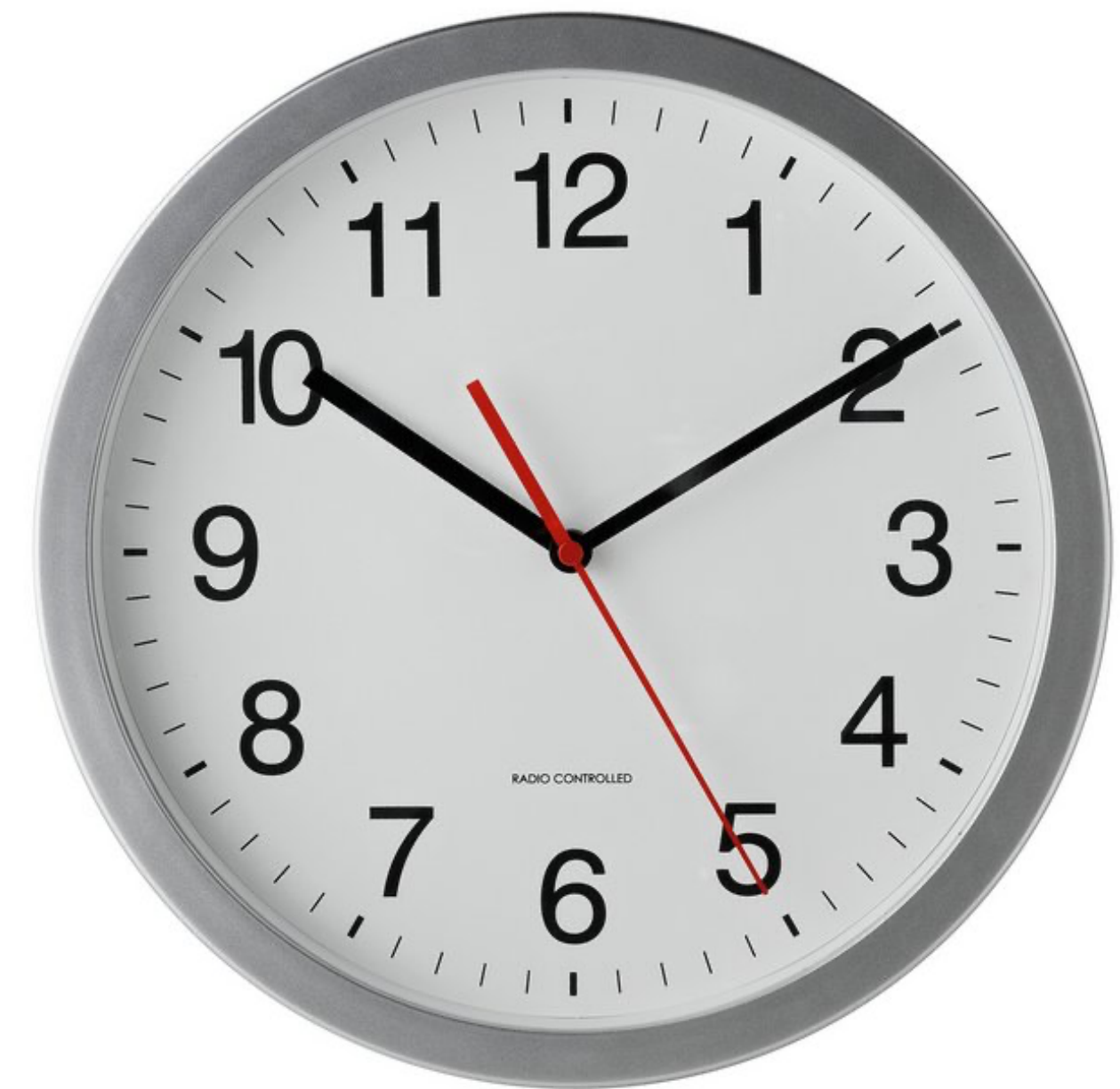


A clock

```
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

- It is good practice to allow the system to stutter in every state
- Stuttering can represent events by the environment or by other components of the system (not yet modeled)
- Stuttering will enable the composition of concepts when specifying apps

Stuttering

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

Fixing the trash behavior

```
fact Behavior {  
  // Initial state  
  no Accessible  
  no Trashed  
  // 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

- As usual, **check** commands can be used to verify 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

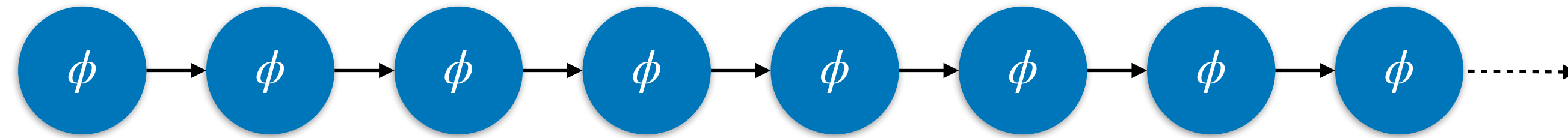
ϕ will eventually be true and ψ is true until then

ϕ **releases** ψ

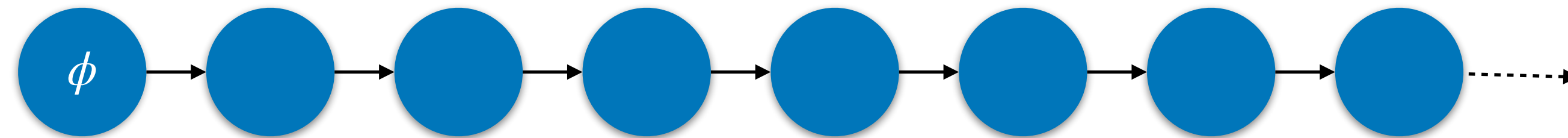
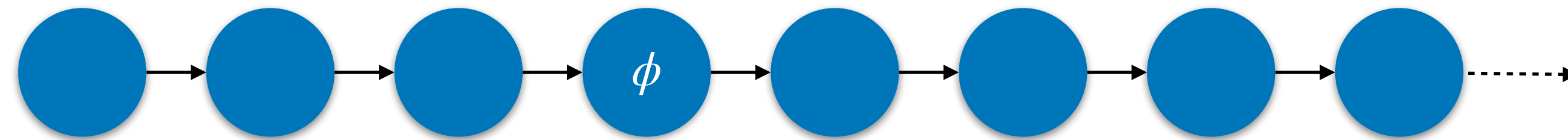
ψ can only stop being true after ϕ

Future operators

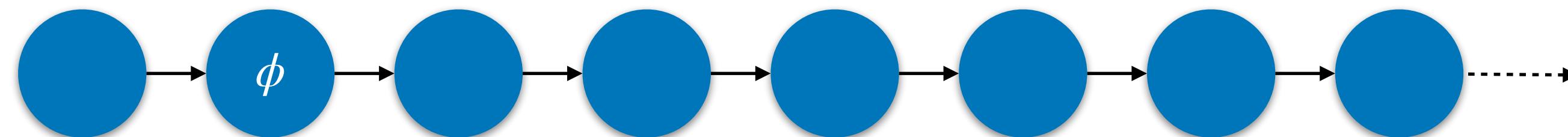
always ϕ



eventually ϕ

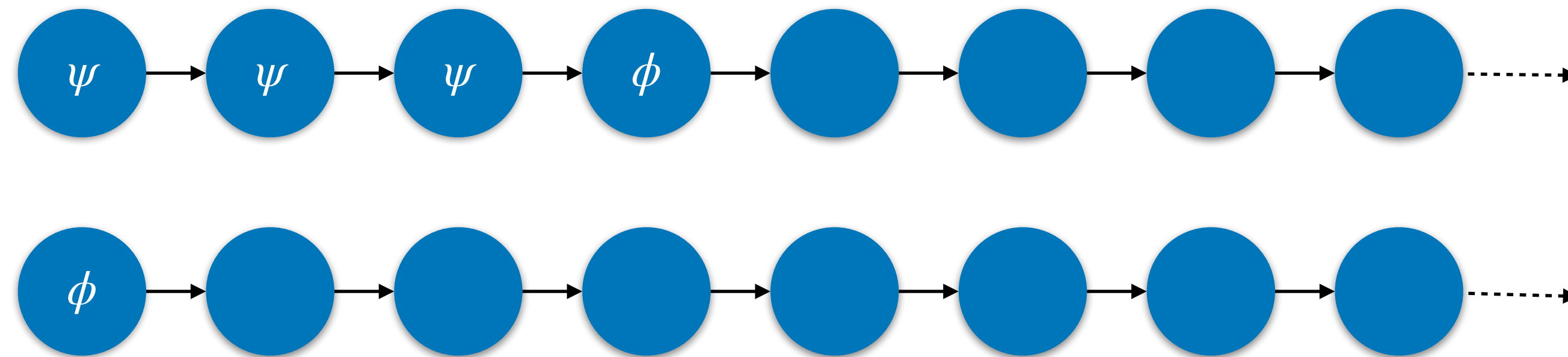


after ϕ

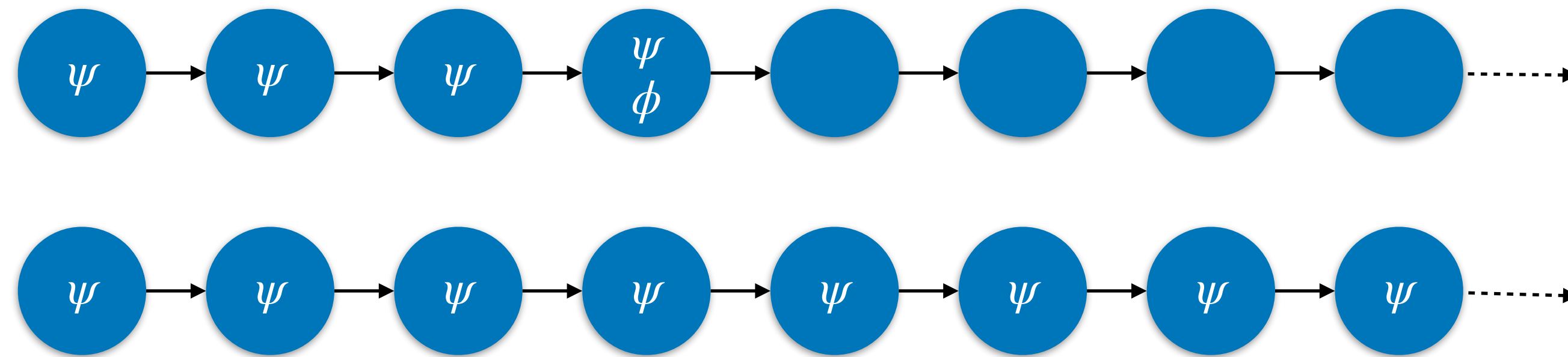


Future operators

ψ until ϕ

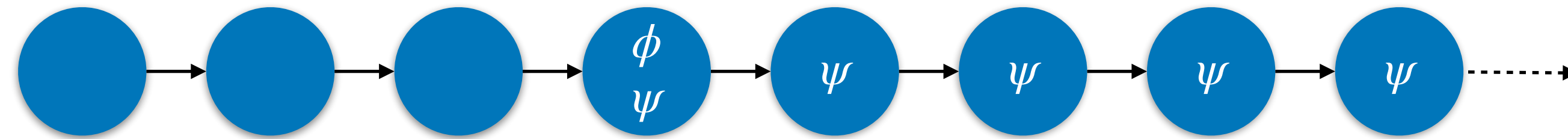


ϕ releases ψ

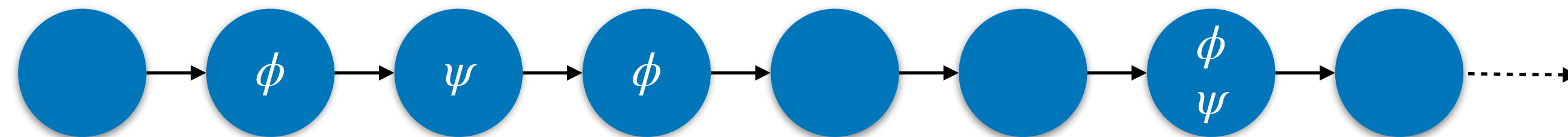


Mixing operators

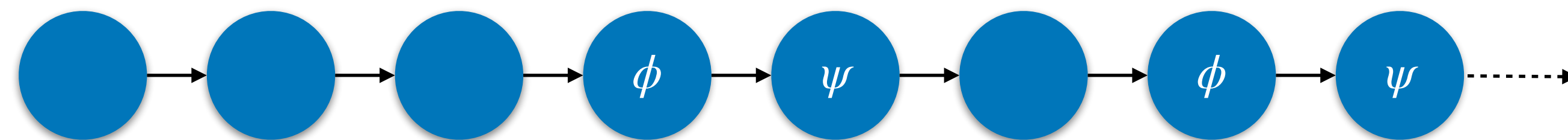
always (ϕ implies always ψ)



always (ϕ implies eventually ψ)

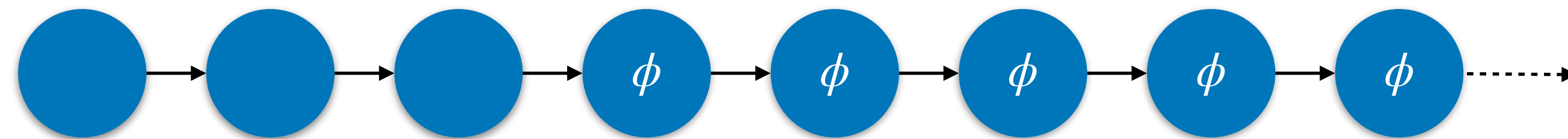


always (ϕ implies after ψ)

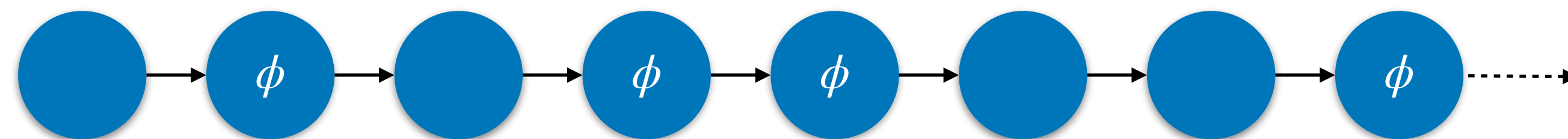


Mixing operators

eventually (always ϕ)



always (eventually ϕ)

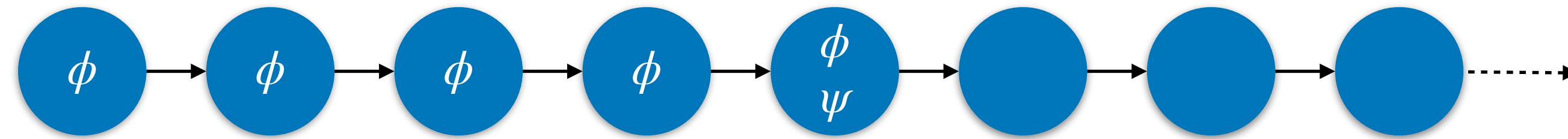


Past temporal operators

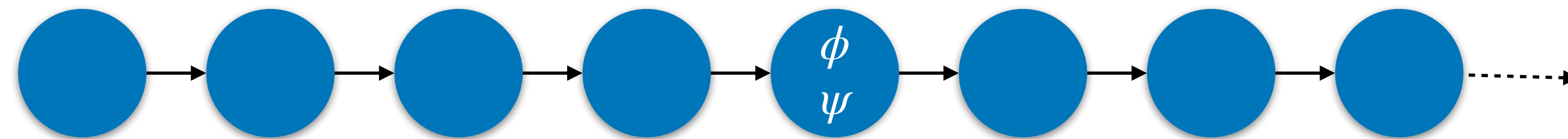
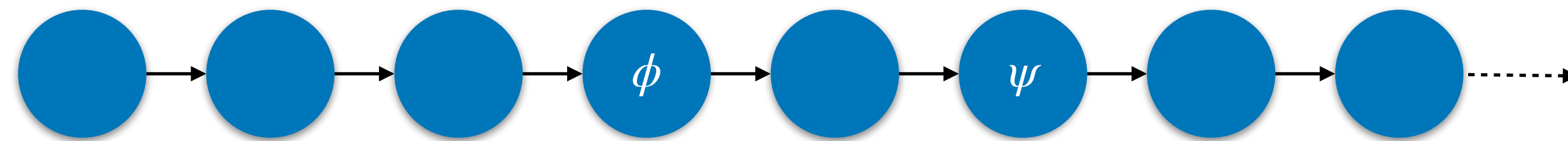
historically ϕ	ϕ was always true
once ϕ	ϕ was once true
before ϕ	ϕ was true in previous state
ψ since ϕ	ϕ was once true and ψ was true since then
ϕ triggered ψ	ψ was always true back to the point where ϕ was true

Past operators

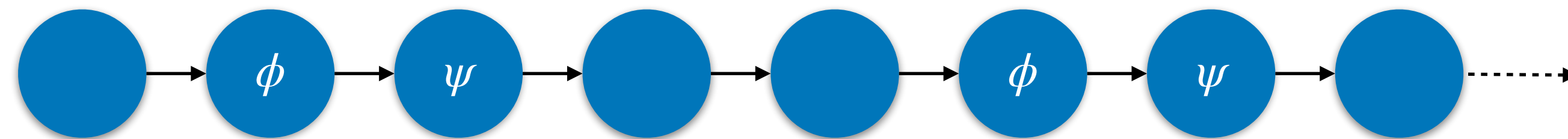
always (ψ implies historically ϕ)



always (ψ implies once ϕ)

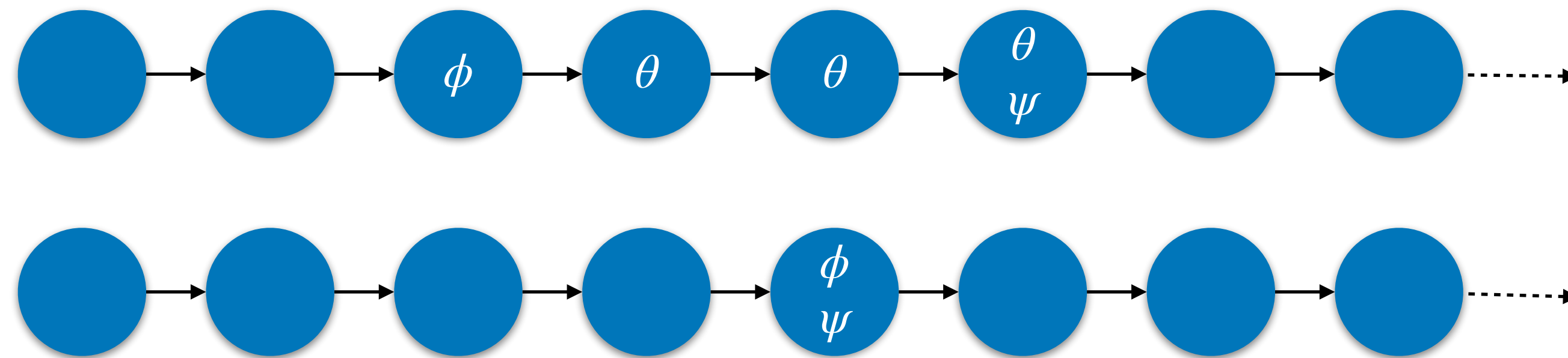


always (ψ implies before ϕ)

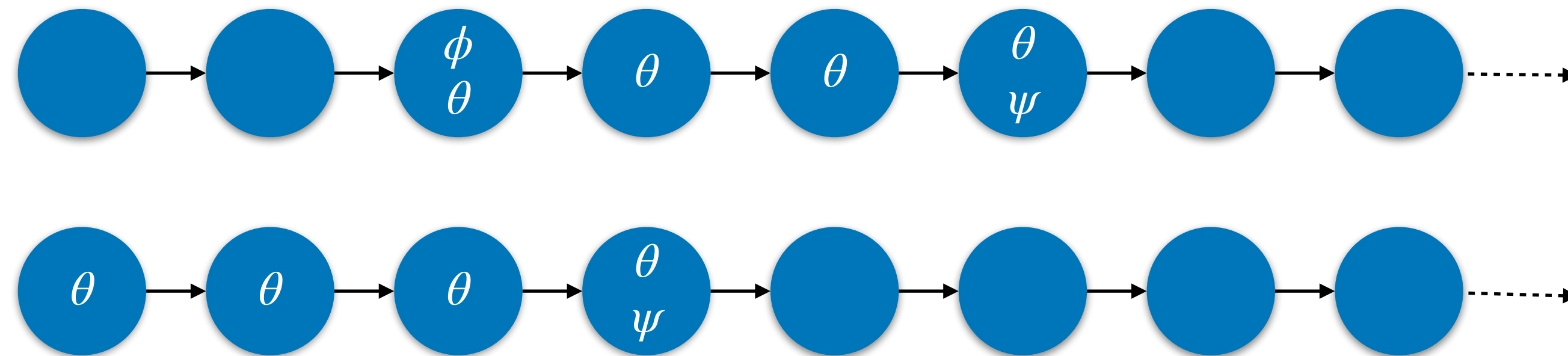


Past operators

always (ψ implies θ since ϕ)



always (ψ implies ϕ triggered θ)



Safety vs Liveness

- *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, and any continuation leads to a counter-example
 - The archetypal safety property is **always** ϕ
- *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 happened
 - 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  
    all x : Item | always (delete[x] implies after (restore[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*

The label

concept label [Item]

purpose

organize items into overlapping categories

state

labels : Item -> set Label

actions

affix (i : Item, l : Label)

add l to the labels of i

detach (i : Item, l : Label)

remove l from the labels of i

find (l : Label) : set Item

return the items labelled with l

clear (i : Item)

remove item i and all its labels

operational principle

after affix(i,l) and no detach(i,l), i in find(l)

The label in Alloy

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

fun find [l : Label] : set Item { labels.l }

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

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

Affix label

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

Detach label

```
pred detach [i : Item, l : Label] {  
  // guard  
  i in find[l]  
  // 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

```
assert affix_find {  
  all i : Item, l : Label {  
    // after affix(i,l) and no detach(i,l), i in find(l)  
    always (affix[i,l] implies after (detach[i,l] releases i in find[l]))  
  }  
}  
check affix_find
```


Label operational principle

Executing "Check affix_find"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
1..3 steps. 2546 vars. 108 primary vars. 6238 clauses. 446ms.

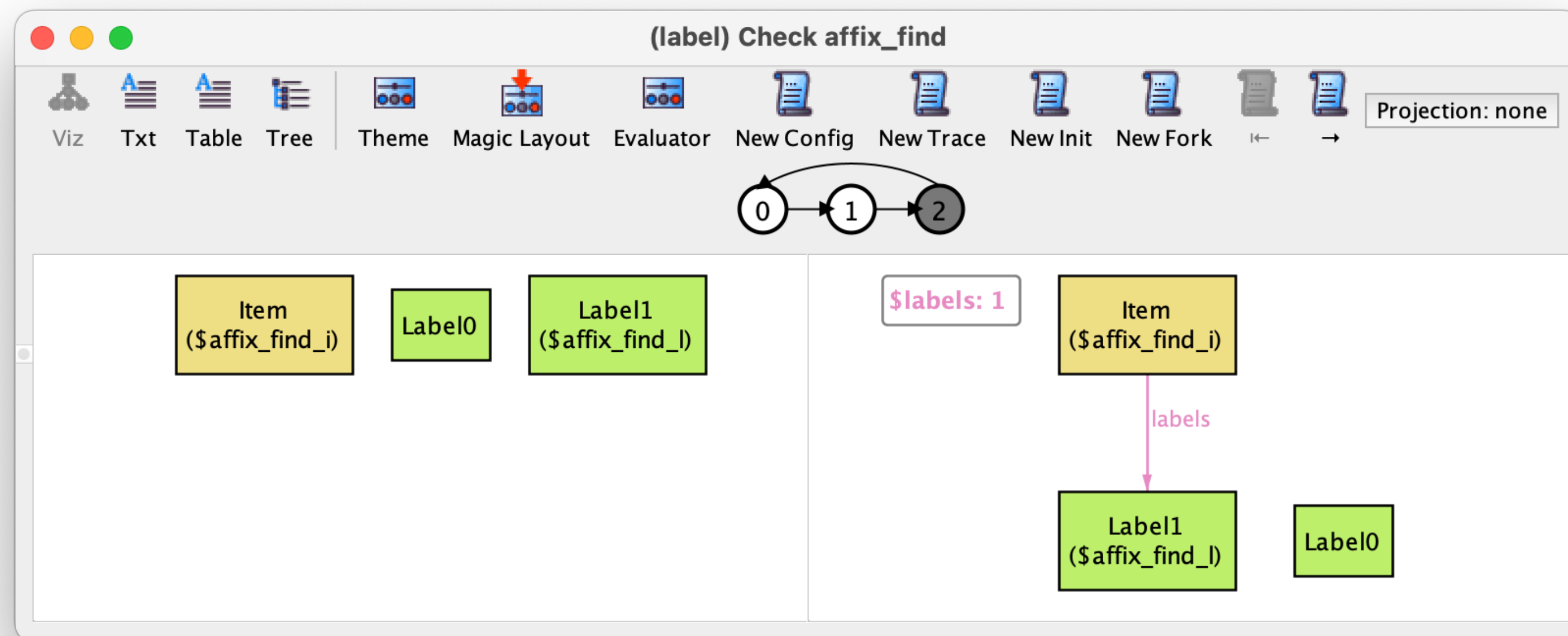
Counterexample found. Assertion is invalid. 127ms.

Label operational principle

Executing "Check affix_find"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
1..3 steps. 2546 vars. 108 primary vars. 6238 clauses. 446ms.

Counterexample found. Assertion is invalid. 127ms.

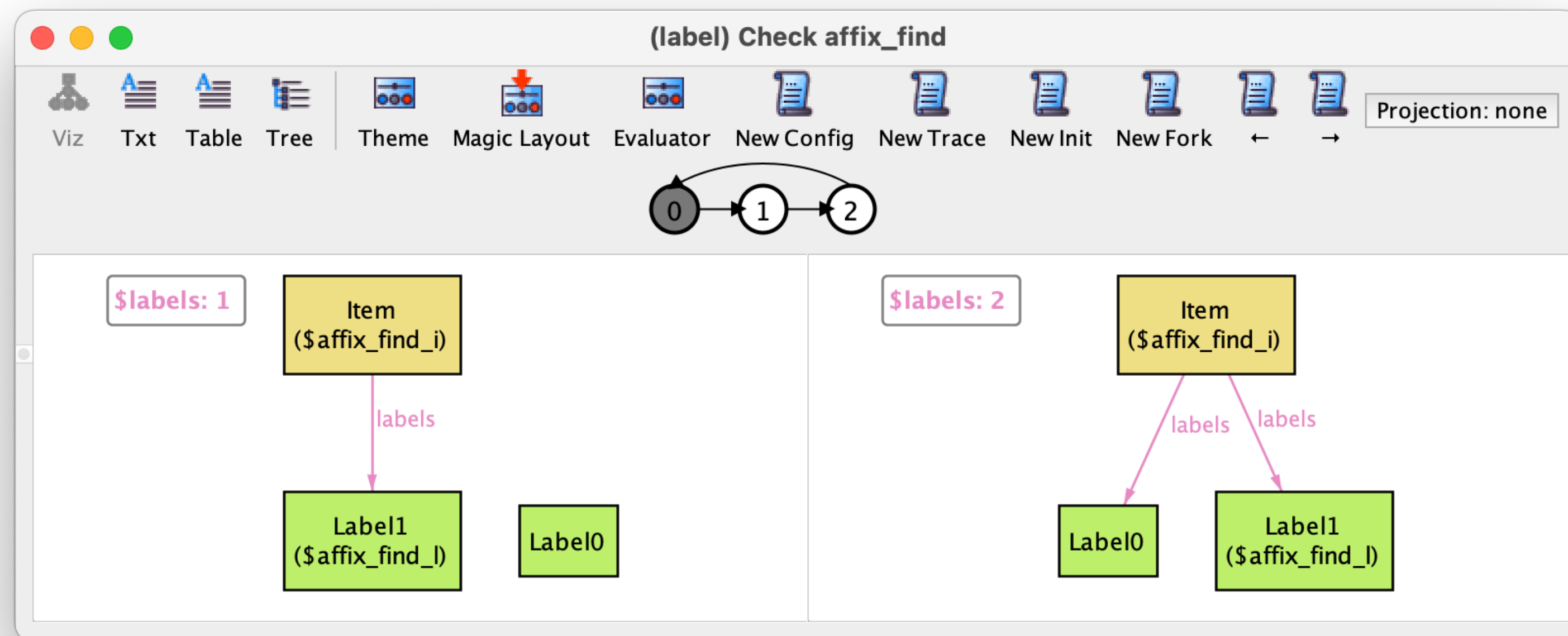


Label operational principle

Executing "Check affix_find"

Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch
1..3 steps. 2546 vars. 108 primary vars. 6238 clauses. 446ms.

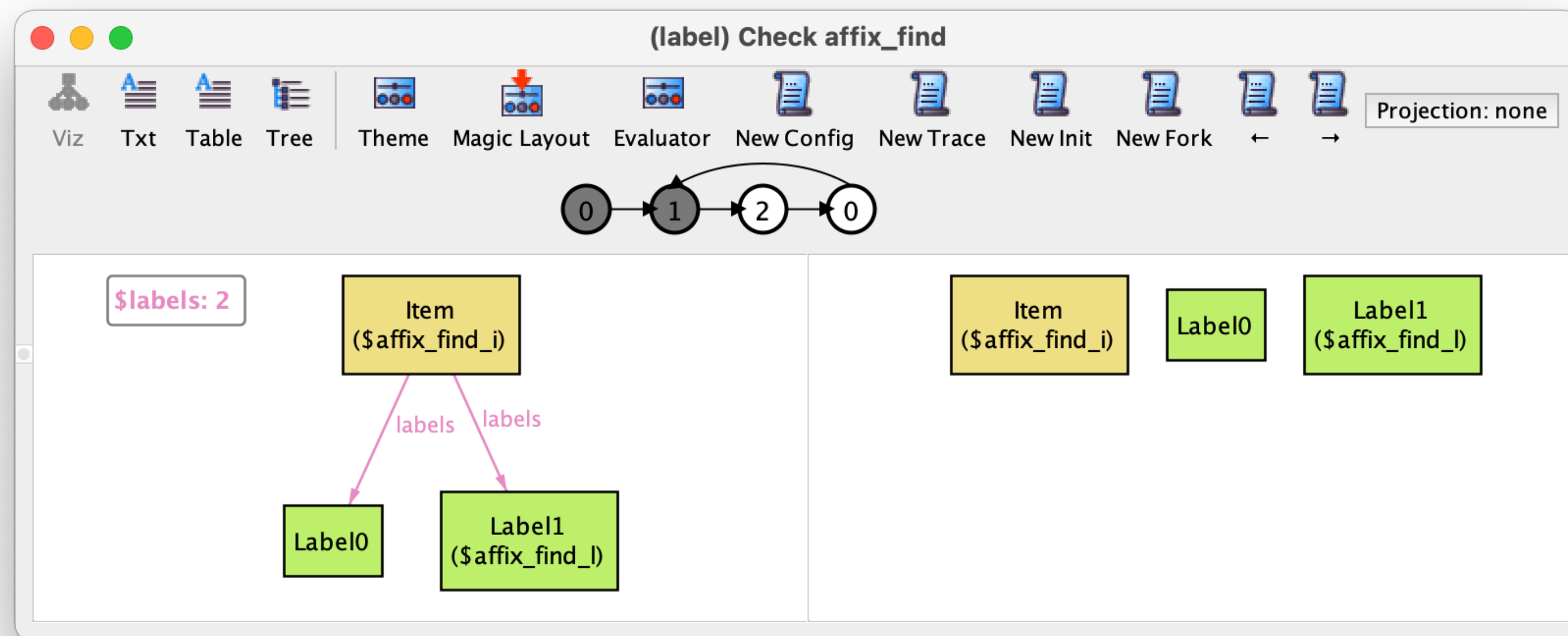
Counterexample found. Assertion is invalid. 127ms.



Label operational principle

Executing "Check affix_find"

```
Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20 Mode=batch  
1..3 steps. 2546 vars. 108 primary vars. 6238 clauses. 446ms.  
Counterexample found. Assertion is invalid. 127ms.
```



Label operational principle

```
assert affix_find {
  all i : Item, l : Label {
    // after affix(i,l) and no detach(i,l), i in find(l)
    always (affix[i,l] implies after ((detach[i,l] or clear[i]) releases i in find[l]))
  }
}
check affix_find
```

***Concept* composition**

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 trick 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 Label {  
  var items : set Item  
}
```

```
fun labels : Item -> set Label {  
  ~items  
}
```

```
...
```

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

- 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 {}
```

Simulation



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
```


Simulation



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  
  all f : File | always {  
    delete{f} f in Trashed and empty and some f.labels implies after clear[f]  
  }  
}
```

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

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

```
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
```

```
run Scenario4 {  
  some f : File, l : Label | create[f]; delete[f]; affix[f,l]  
} expect 1
```

Executing "Run Scenario4 expect 1"
Solver=minisat(jni) Steps=1..10 Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=OFF Mode=batch
2..10 steps. 72419 vars. 1791 primary vars. 158243 clauses. 453ms.
No instance found. **Predicate** may be inconsistent, contrary to expectation. 16ms.

```
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
```

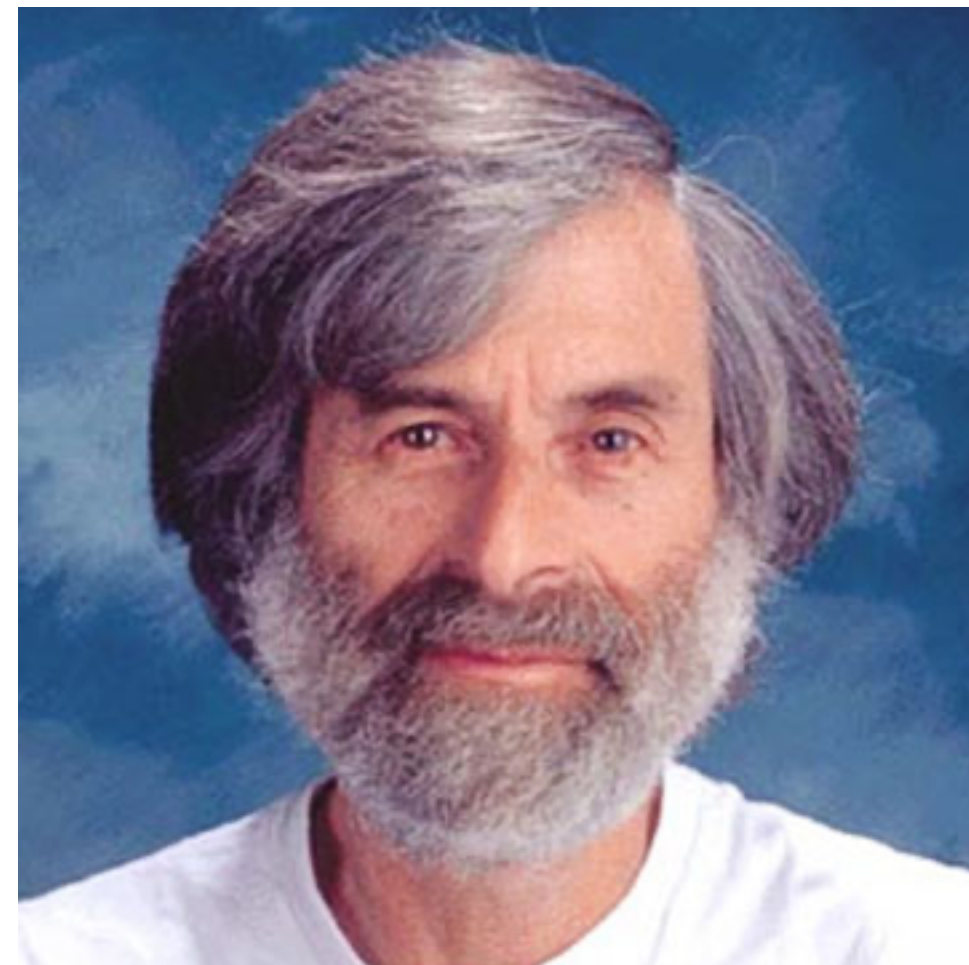
```
run Scenario5 {  
  some disj f1,f2 : File, l : Label | create[f1]; delete[f1]; affix[f1,l]; empty; create[f2]  
} expect 0
```

Filesystem v2

```
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)  
  }  
}
```

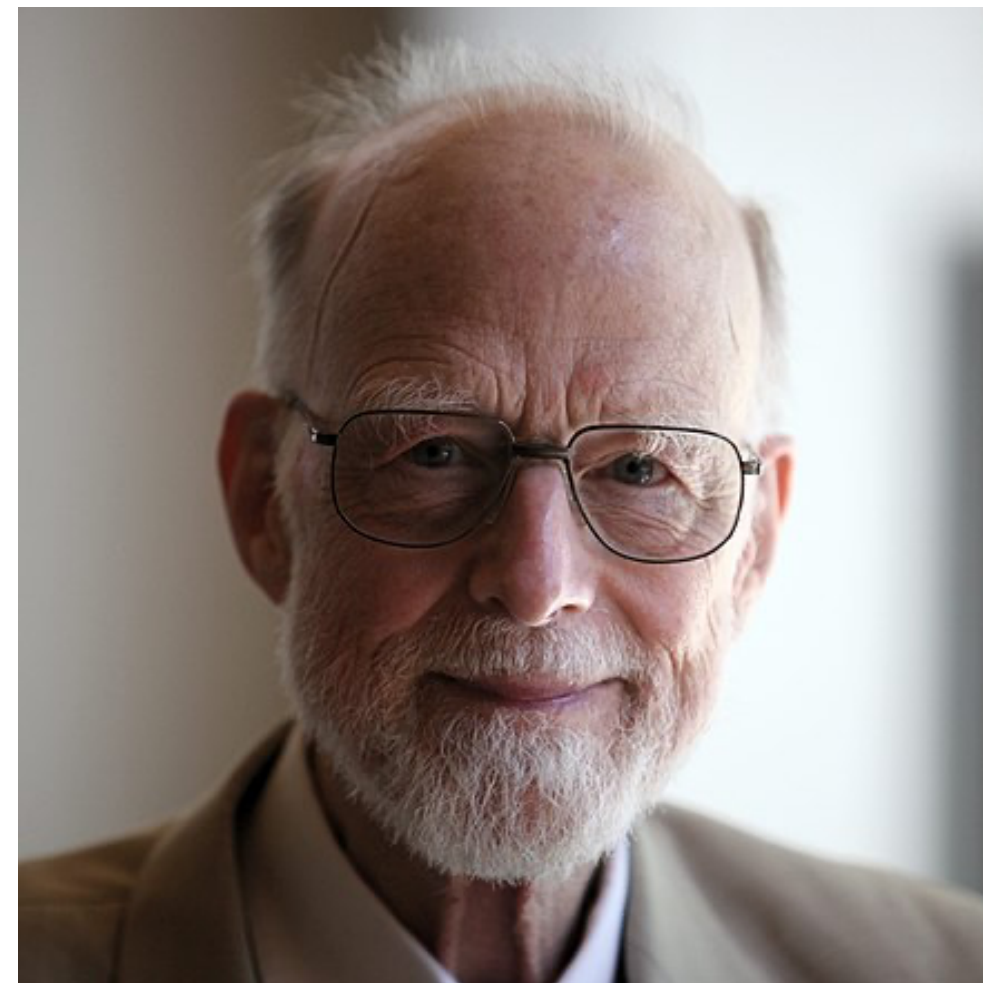
Epilogue

“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