Artificial Intelligence
for Games

Game AI

Game intelligence ≠ intelligence

Game AI is the illusion of intelligence.

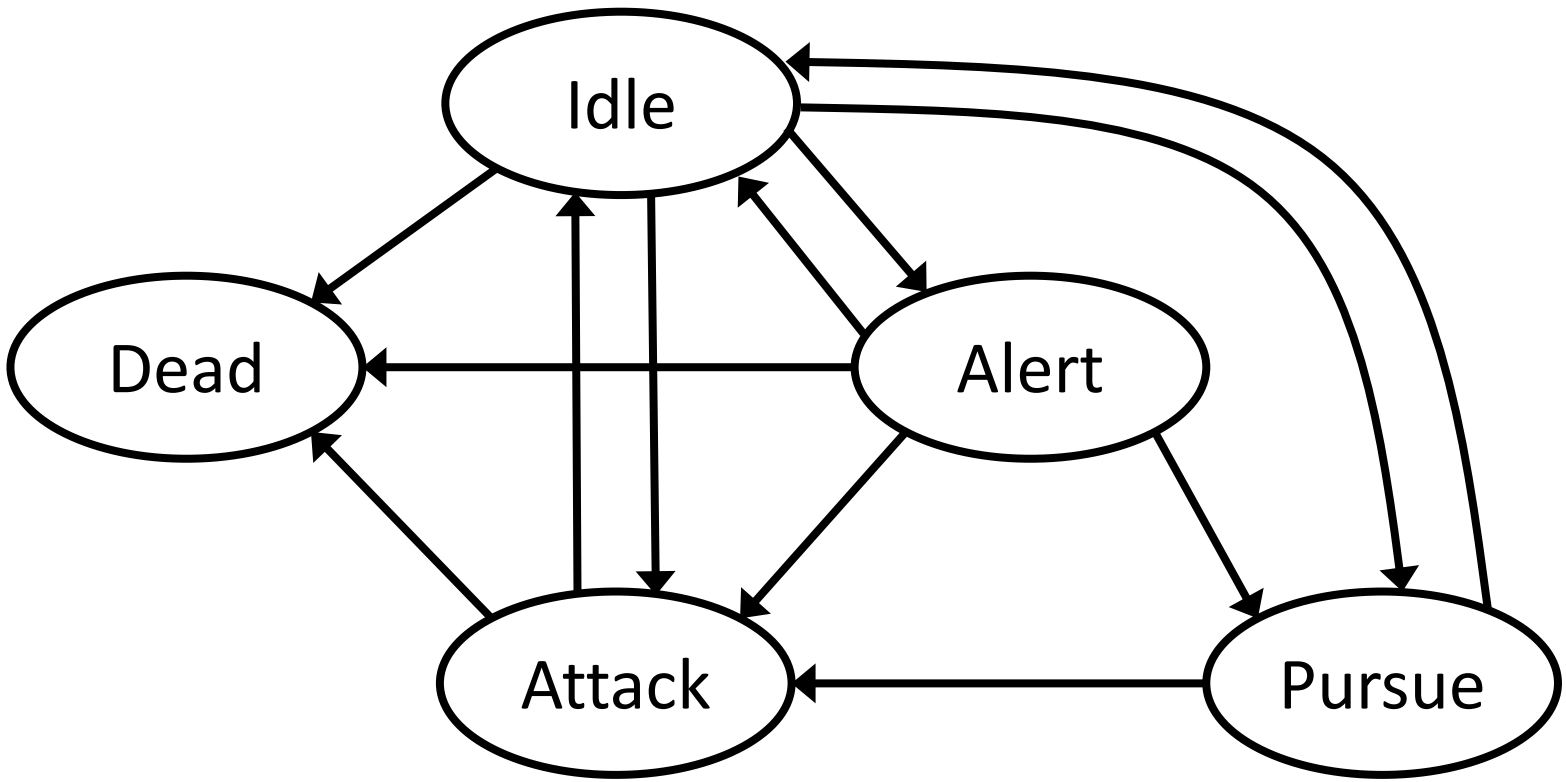
Scientific AI

- Robust
- Domain independent
- Computationally expensive

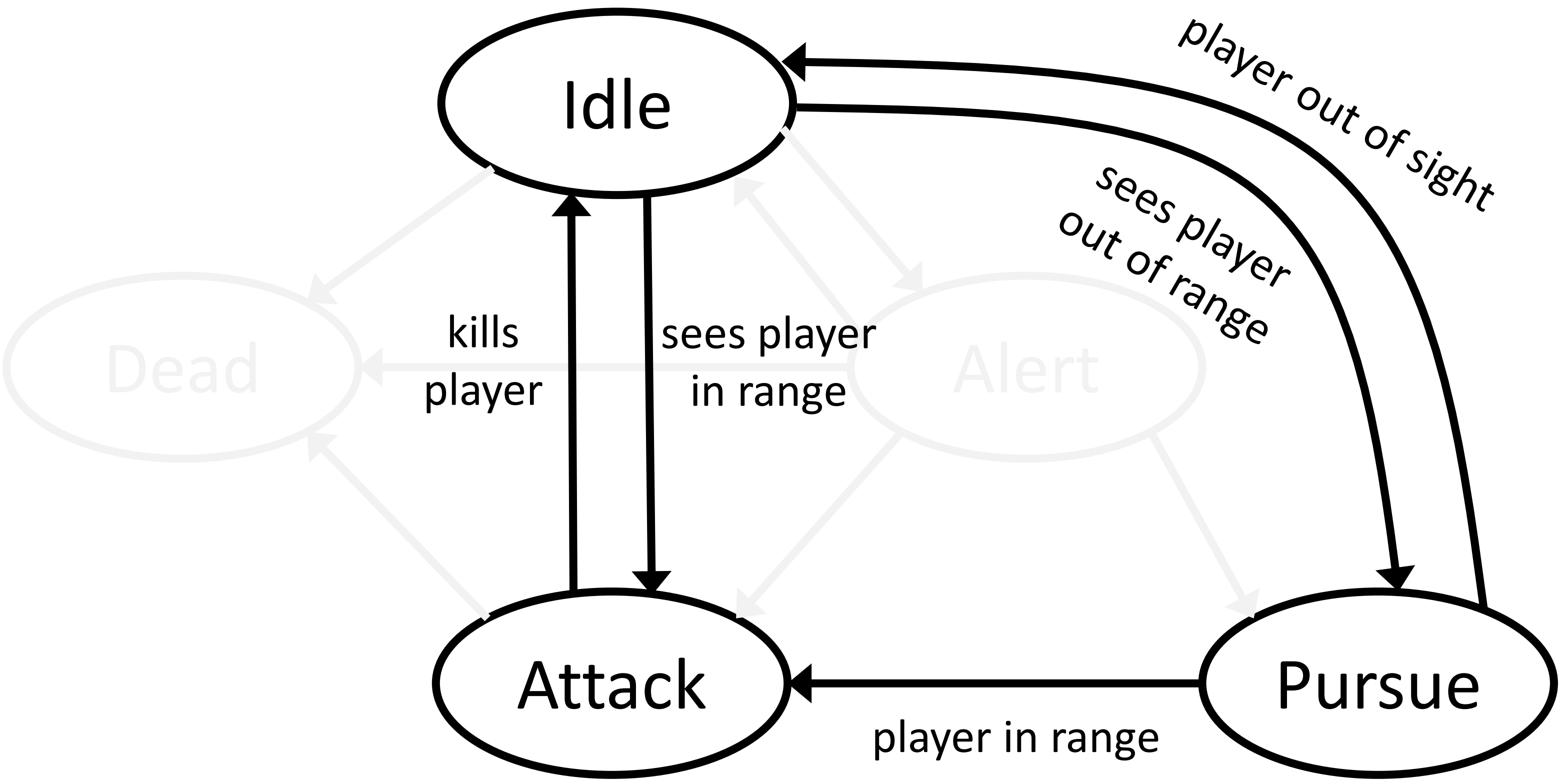
Game AI

- Simple
- Game specific
- Low-order polynomial time only

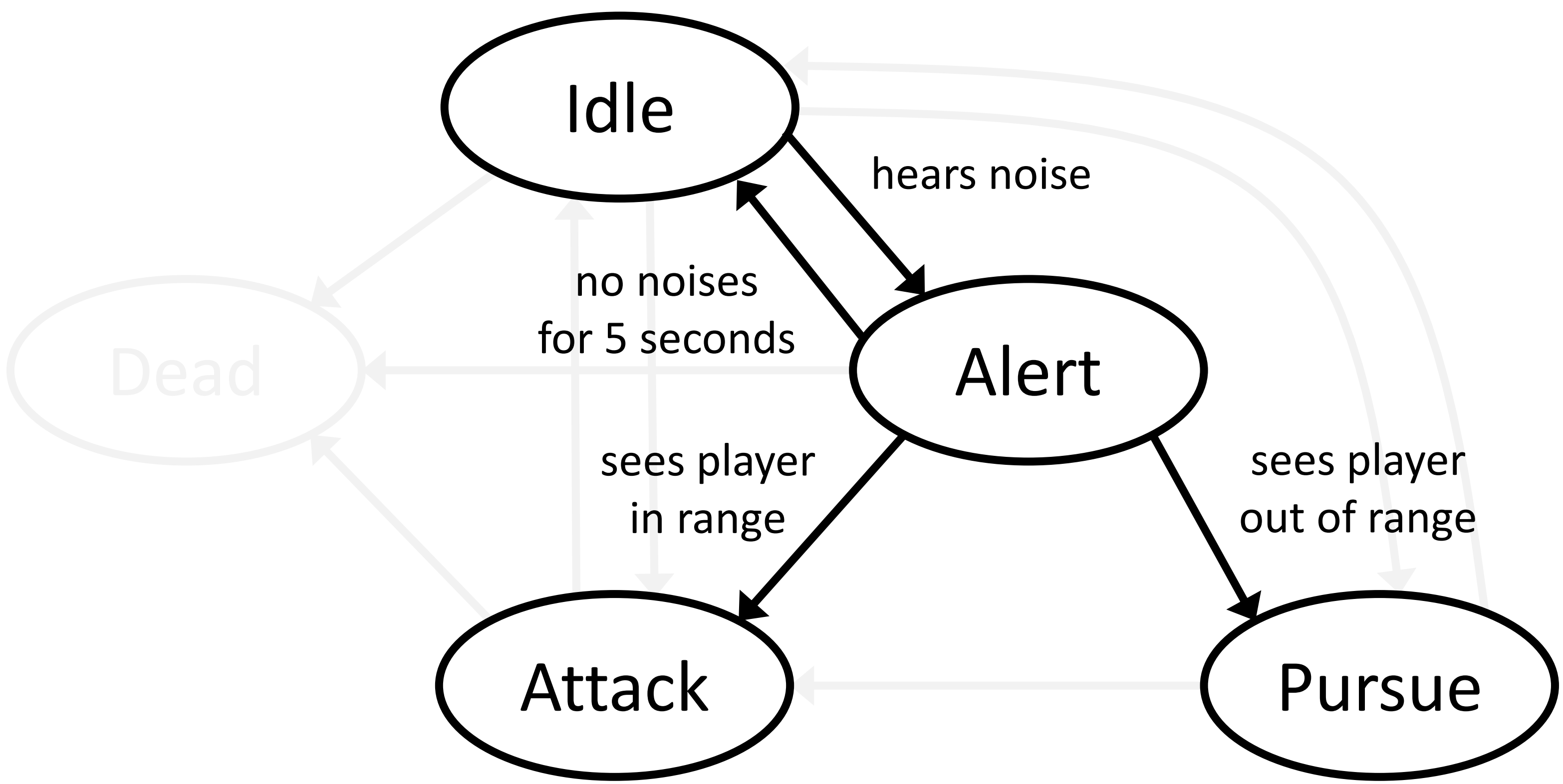
Example: Finite State Machines



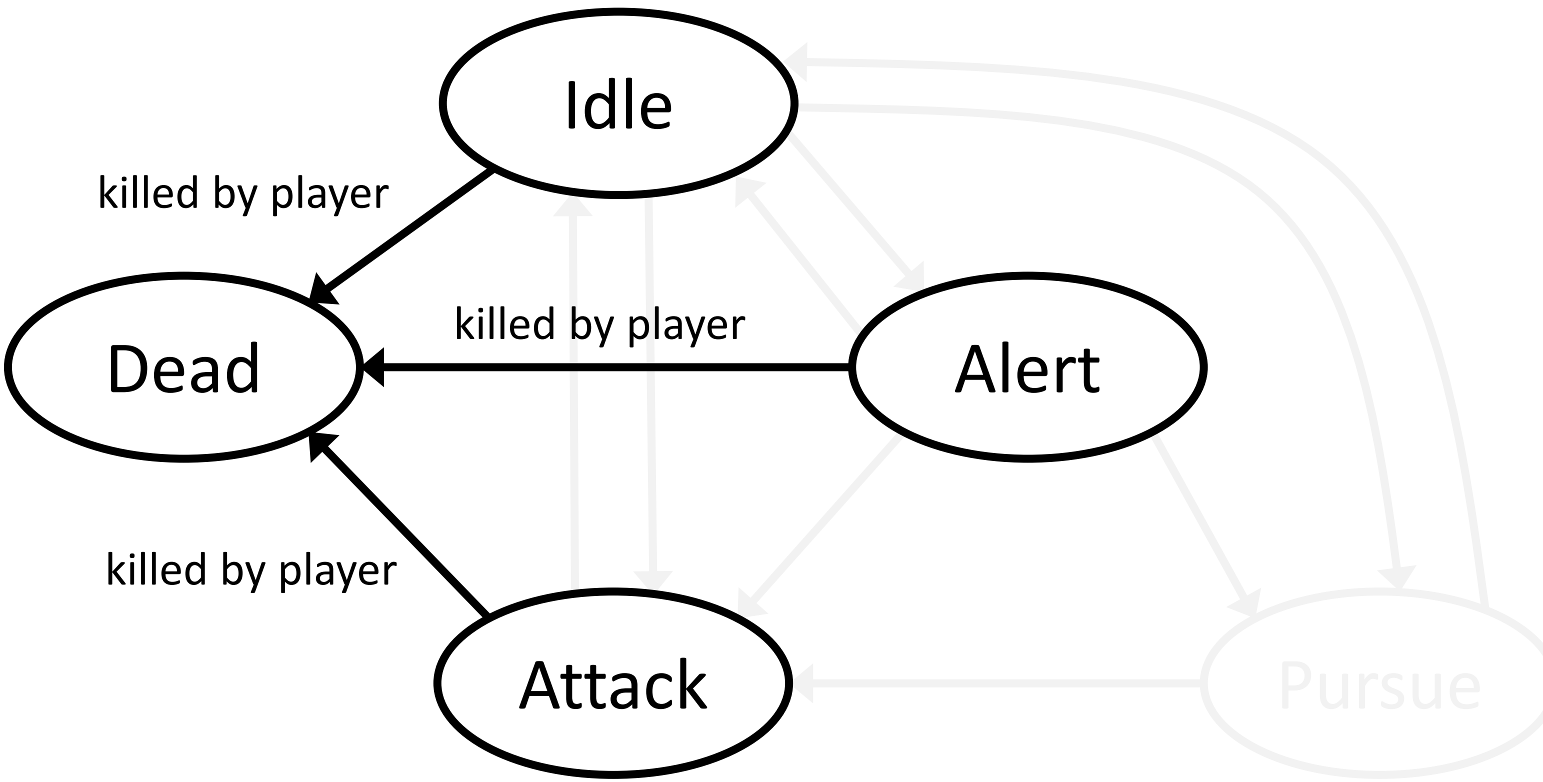
Example: Finite State Machines



Example: Finite State Machines



Example: Finite State Machines



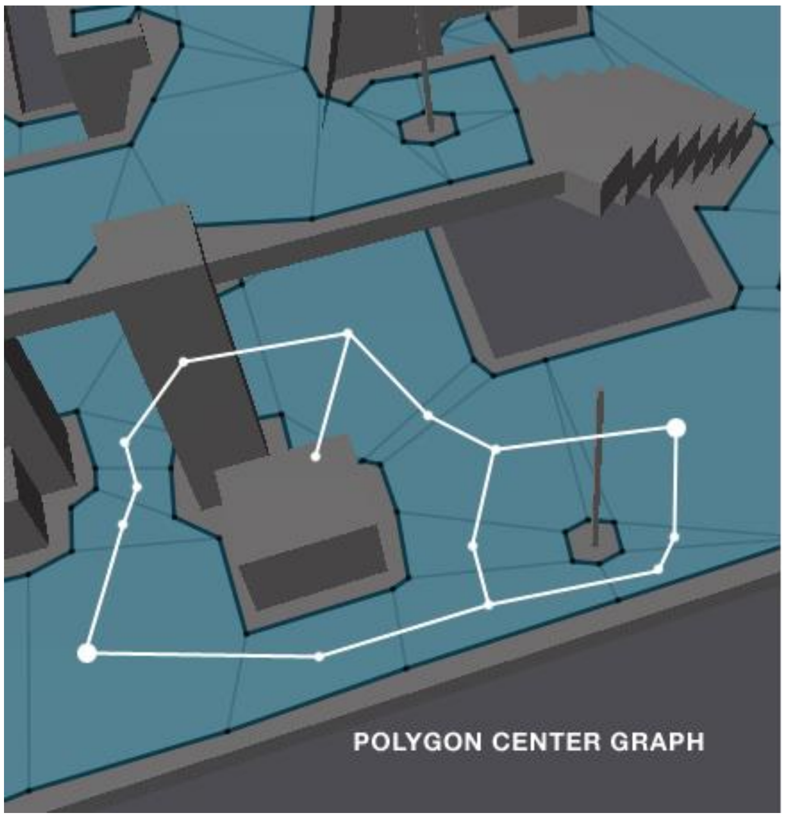
Topics in Game AI

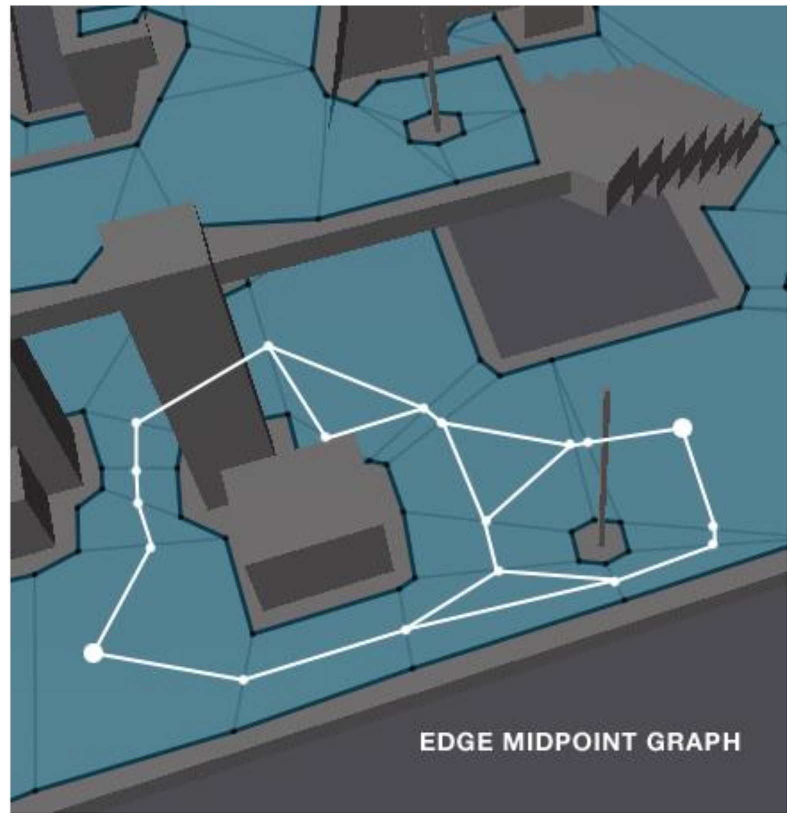
- A* Pathfinding
- Behavior Trees
- Alpha-Beta Pruning

Pathfinding

Given an origin, a destination, and a discretized physical space, find a route from the origin to the destination.

Navigation Meshes





Review of Graph Search

Until the solution is found…

- **Depth First:** expand the longest path
- **Breadth First:** expand the shortest path
- **Greedy:** expand the path whose endpoint is closest to the destination
- **Best First:** expand the path whose total cost is lowest

Heuristic

A function which efficiently estimates the distance between two locations.

An **admissible heuristic** never overestimates.

A* Algorithm

Let value(*loc*) = distance(*loc*) + h(*loc*, *destination*).

Let distance(*loc*) = ∞ for all locations.

Let distance(*origin*) = 0.

Mark *origin* as visited.

Until all paths are checked:

    Let *loc* be the unvisited location with lowest value.

    If *loc* = *destination*, return success.

    For every unvisited neighbor *n* of *loc*:

        Mark *n* as visited.

        Let distance(*n*) = distance(*loc*) + cost(*loc* → *n*).

        Calculate h(*n*, *destination*).

Return failure.

A* Algorithm Implementation

Let *MPQ* be a min priority queue.

Let distance(*loc*) = ∞ for all locations. Let distance(*origin*) = 0.

Mark *origin* as visited.

Add *origin* to *MPQ* with key h(*origin*, *destination*).

Until *MPQ* is empty:

    Pop *loc* from *MPQ*.

    If *loc* = *destination*, return success.

    For every unvisited neighbor *n* of *loc*:

        Mark *n* as visited.

        Let distance(*n*) = distance(*loc*) + cost(*loc* → *n*).

        Add *n* to *MPQ* with key distance(*n*) + h(*n*, *destination*).

Return failure.

Reconstructing the Path

Once we know that a path exists, how do we reconstruct it?

Keep a history array which, for some location, tells us the previous location visited.

Use this array to build the path backwards from the destination.

A* Pathfinding

Let *MPQ* be a min priority queue.

Let distance(*loc*) = ∞ for all locations. Let distance(*origin*) = 0.

Mark *origin* as visited and add to *MPQ* with key h(*origin*, *destination*).

Until *MPQ* is empty:

    Pop *loc* from *MPQ*.

    If *loc* = *destination*, return success.

    For every unvisited neighbor *n* of *loc*:

        Mark *n* as visited.

        Let history(*n*) = *loc*.

        Let distance(*n*) = distance(*loc*) + cost(*loc* → *n*).

        Add *n* to *MPQ* with key distance(*n*) + h(*n*, *destination*).

Return failure.

Reconstructing the Path

Given: the history array and knowledge that a path exists.

Let *path* be an empty list.

Call walk(*destination*, *path*).

Function walk(*loc*, *path*):

> If *loc* = *origin*, return.
>
> Let *previous* = history(*loc*).
>
> Call walk(*previous*, *path*).
>
> Add the edge *previous* → *loc* to the end of *path*.

A* Paths

If an admissible heuristic is used, A* will return a shortest path from the origin to the destination.

Behavior Trees

A simple, reusable, graphical way to build complex behaviors from a set of simple ones.

Node Types

- Composite
  - Multiple children
- Decorator
  - Exactly 1 child
- Leaf
  - No children

Arguments

Nodes can pass arguments to their children.

Unless explicitly stated otherwise, nodes pass the same argument they were passed on to their children.

In this exercise, we assume 1 or 0 arguments, which is always a Sprite (location on the grid).

Node Types

Composite

- Sequence / Random Sequence
- Selector / Random Selector

Decorator

- Succeeder / Failer
- Inverter
- Iterator

Leaf

- All pre-written behaviors

Sequence (Composite)

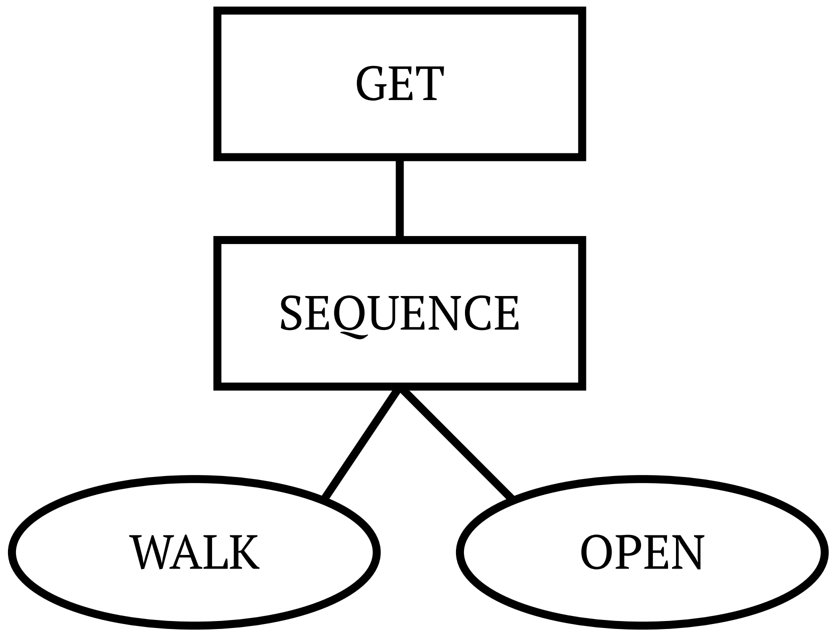
Call each child one at a time.

If a child fails, return false.

If all children succeed, return true.

“Do all these things until something goes wrong.”

Sequence Example

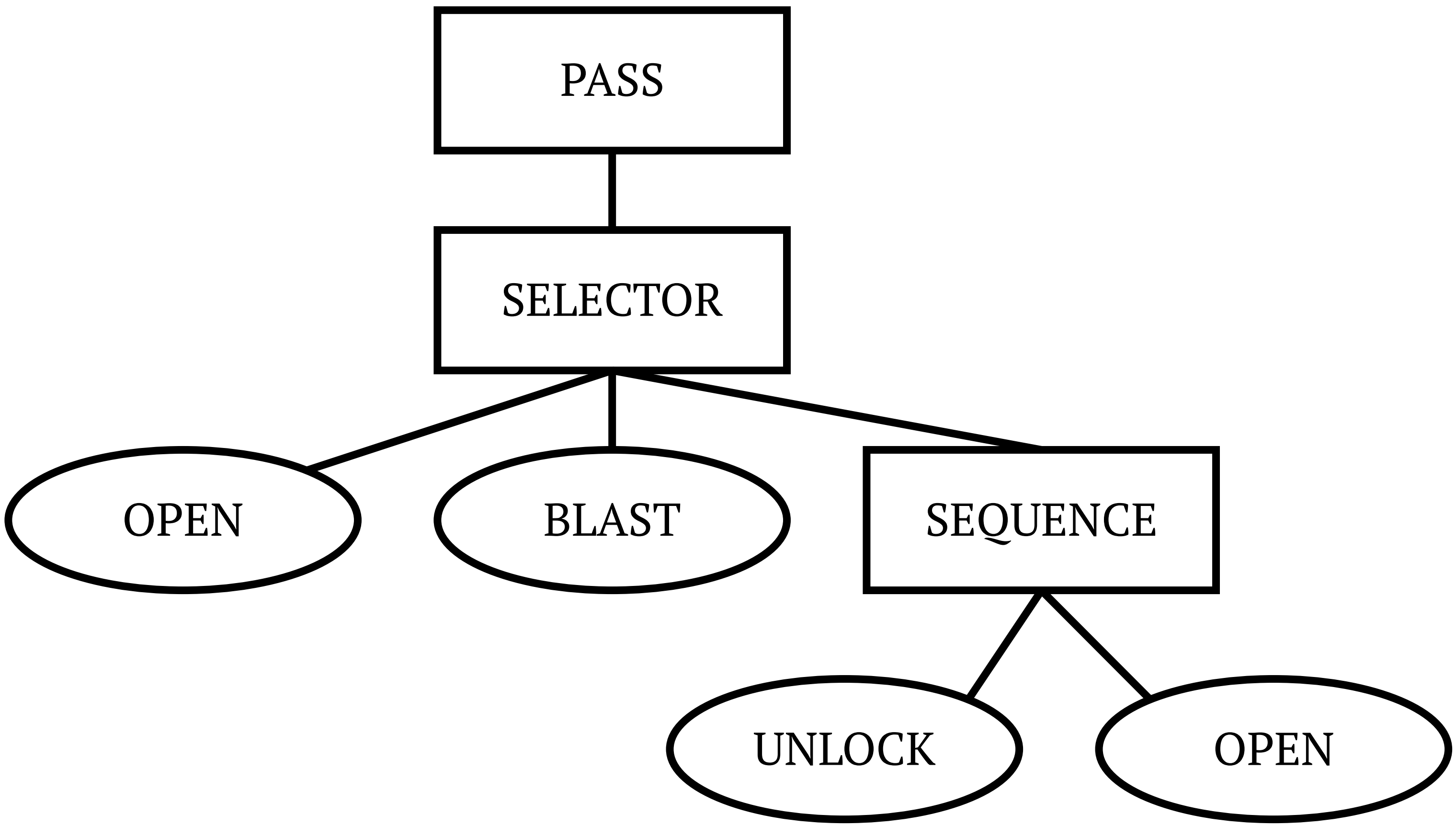


Selector (Composite)

Call each child one at a time.
If a child succeeds, return true.
If all children fail, return false.

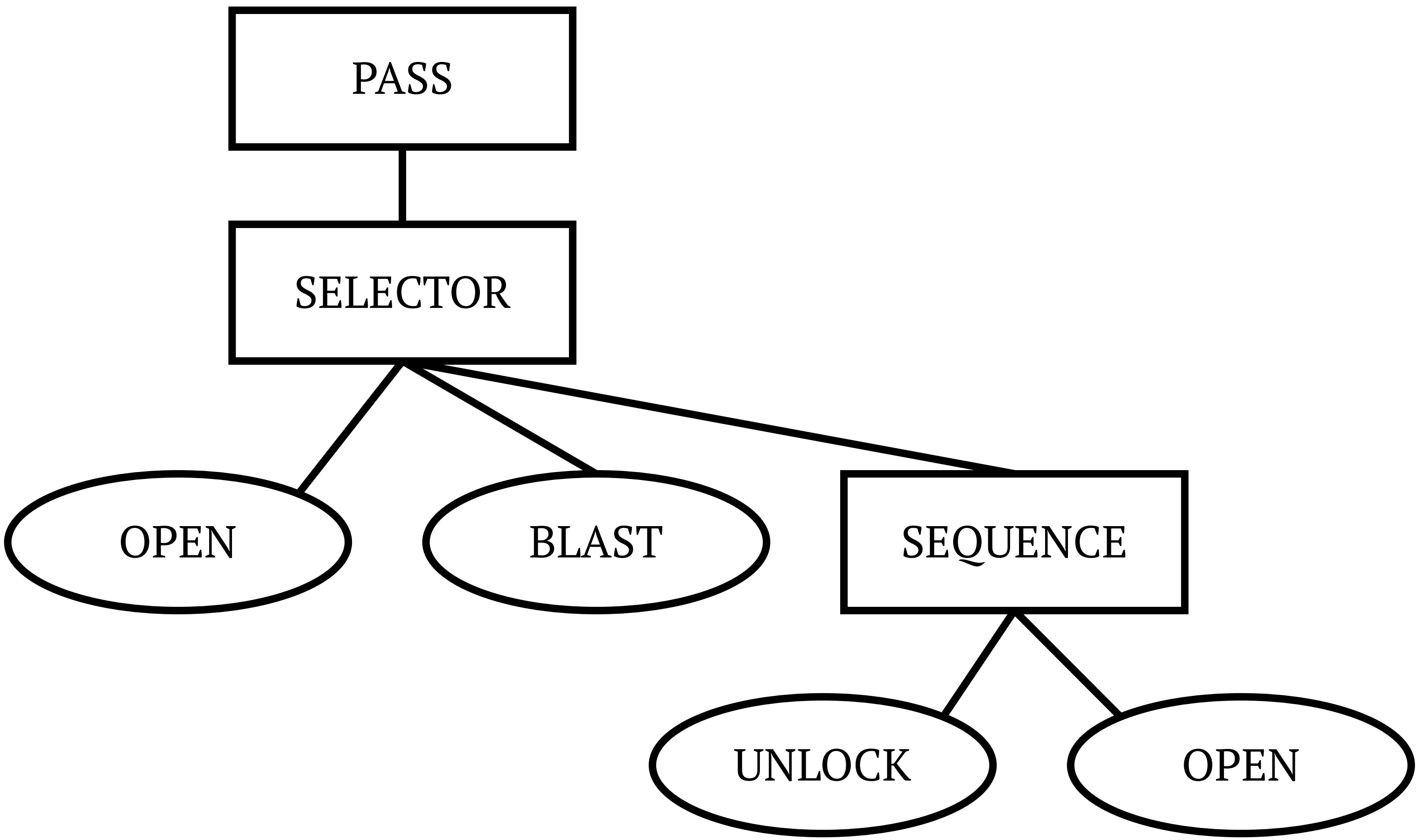
"Keep trying things until something works."
(The opposite of a sequence.)

Selector Example



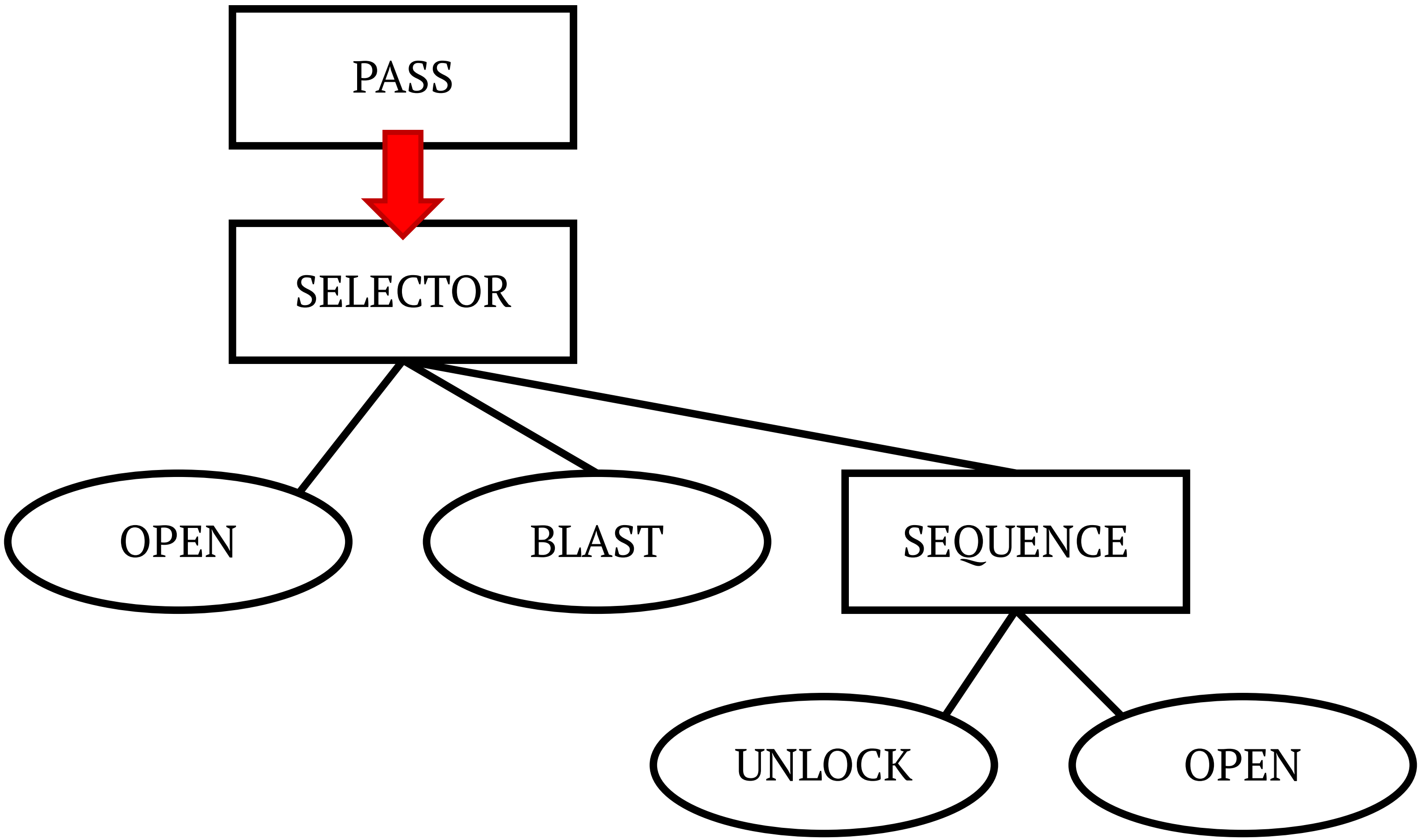
PASS LOCK AT [9,4]

Execution



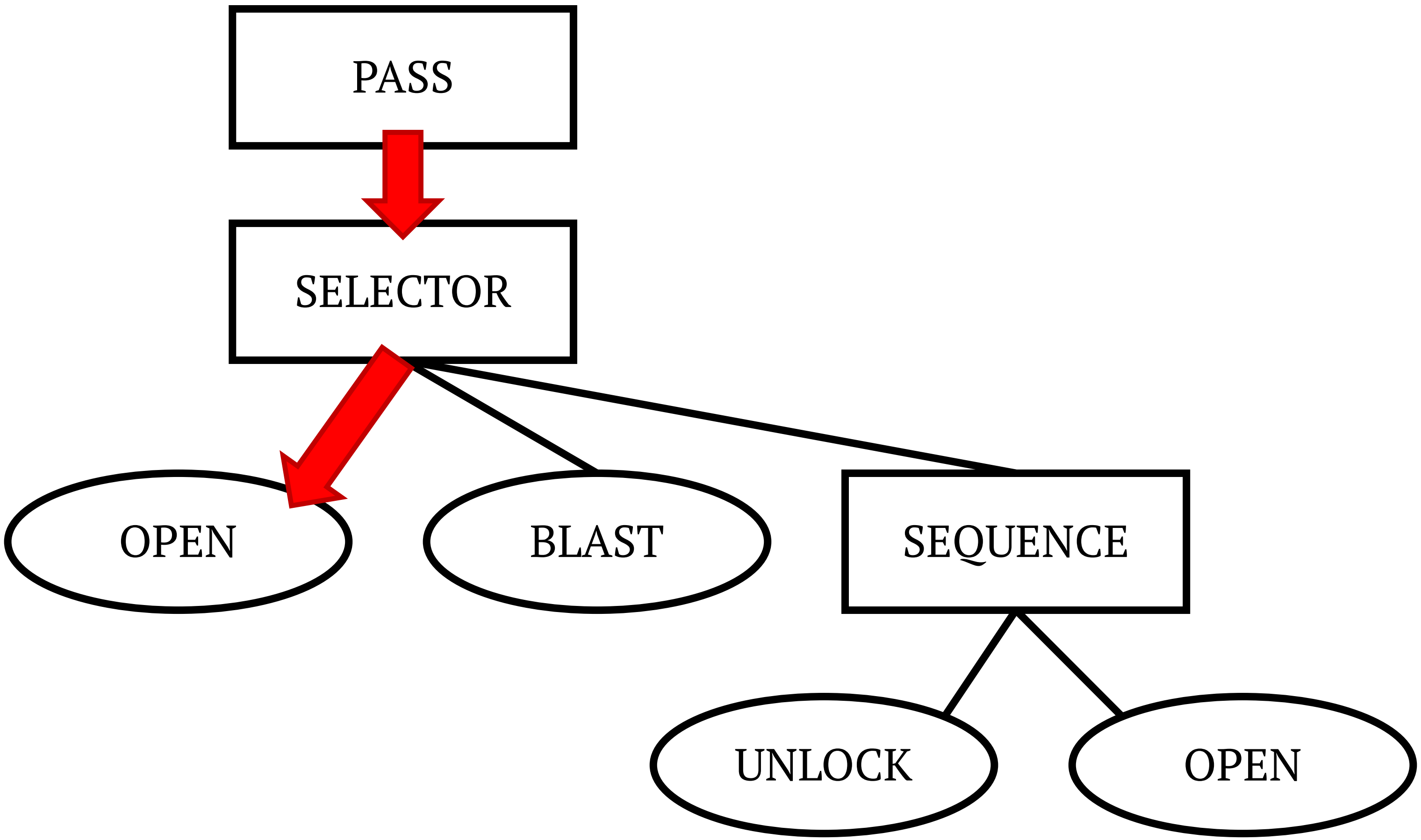
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
```



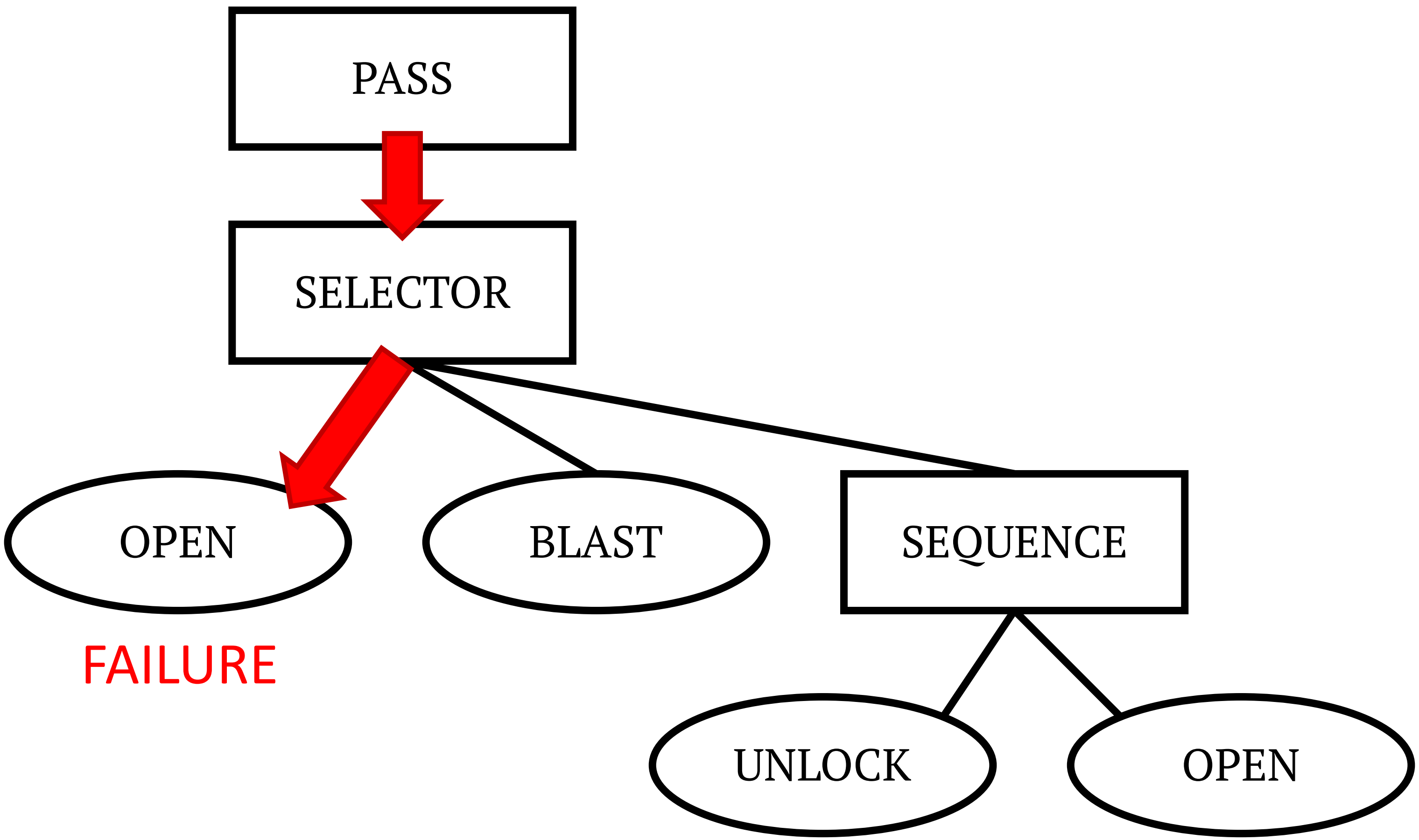
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN:
```



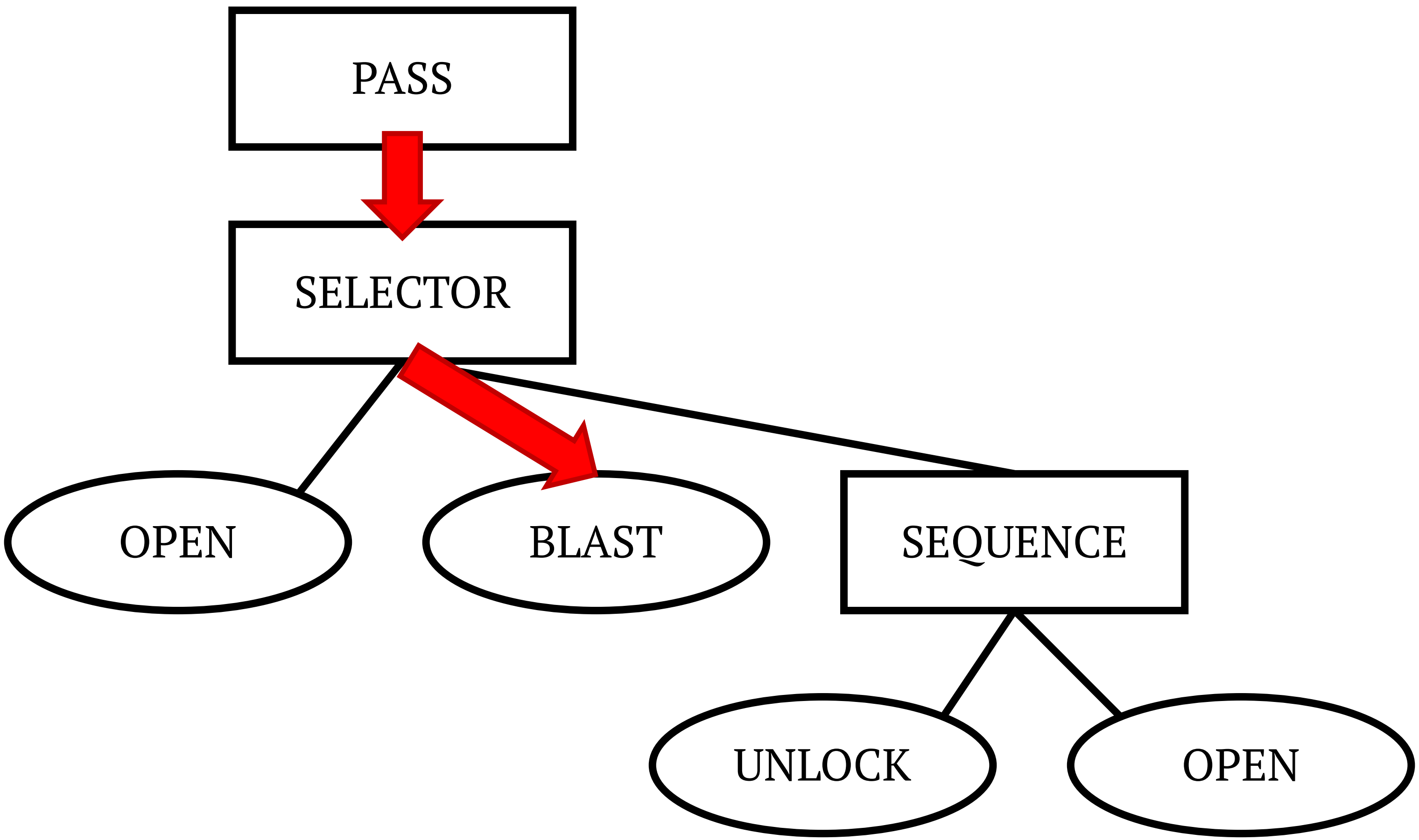
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
```



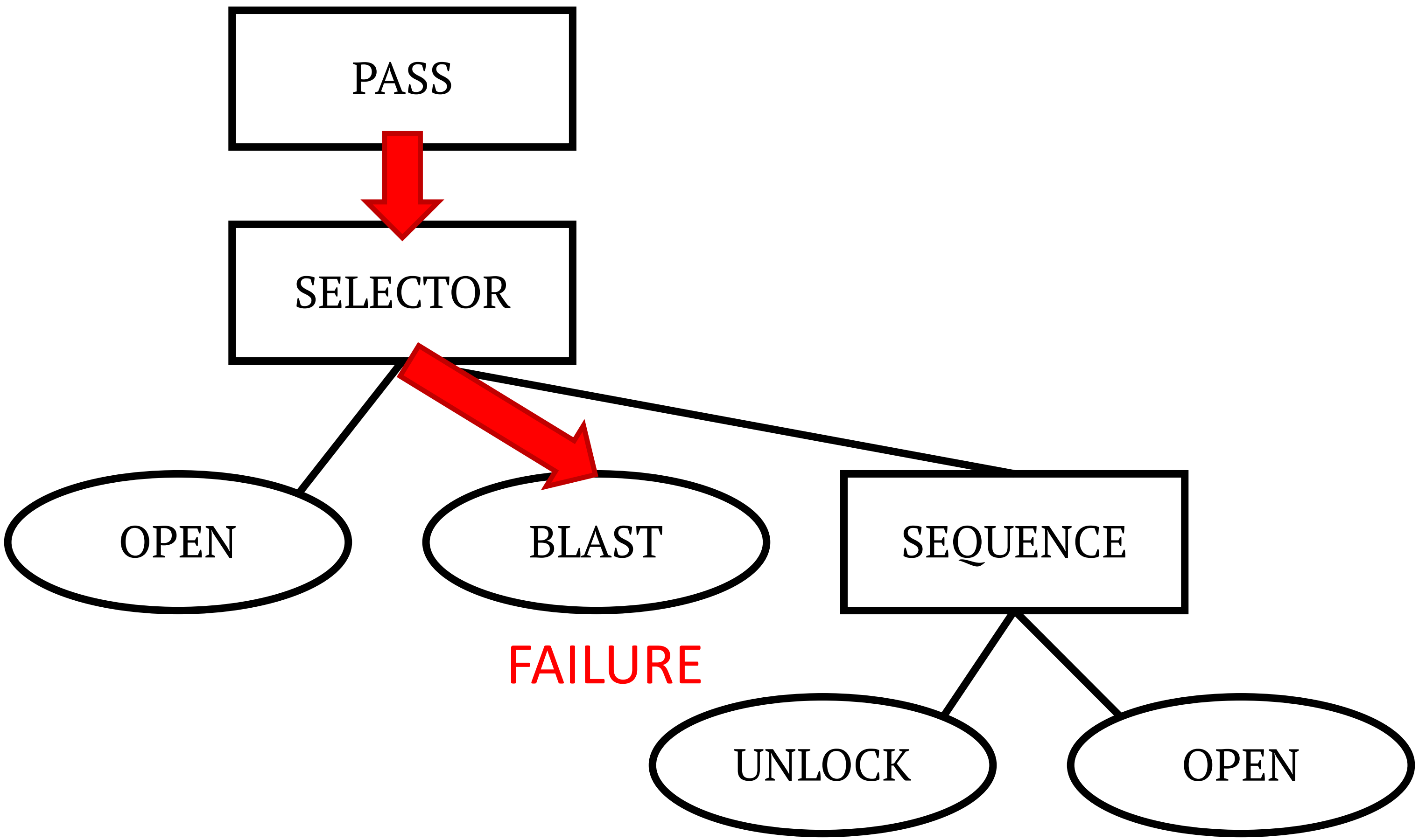
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST:
```



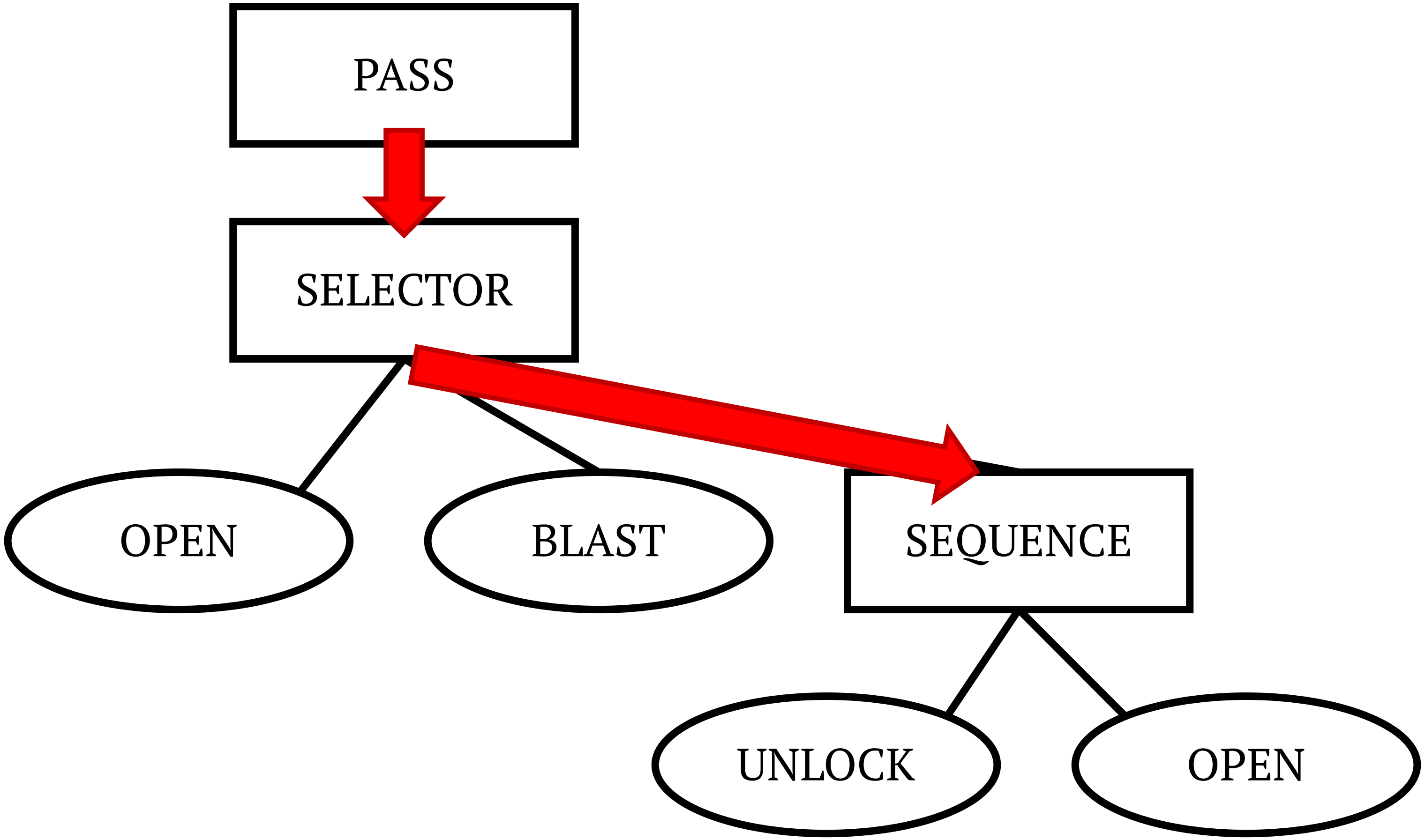
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
```



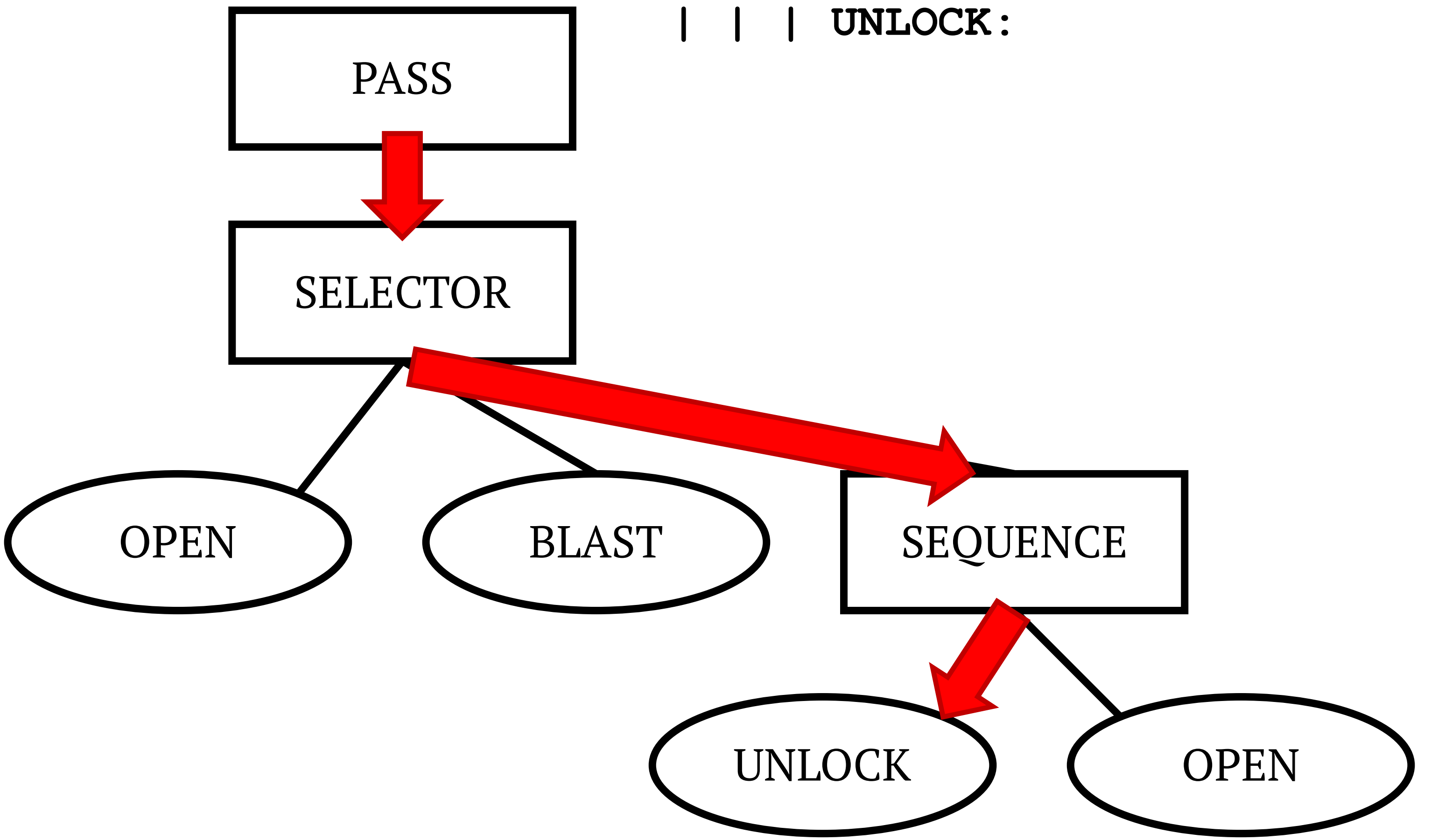
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
```



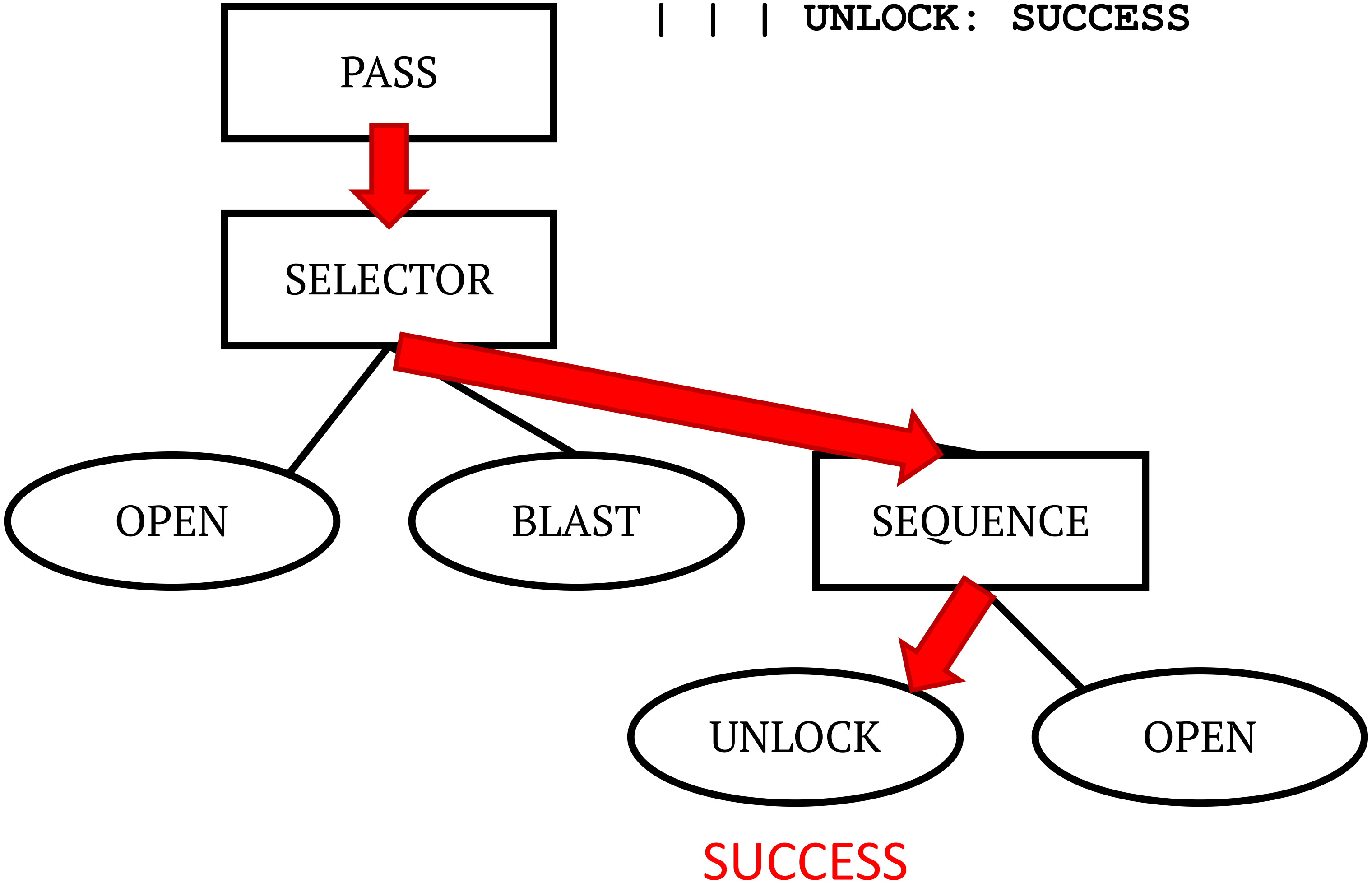
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK:
```



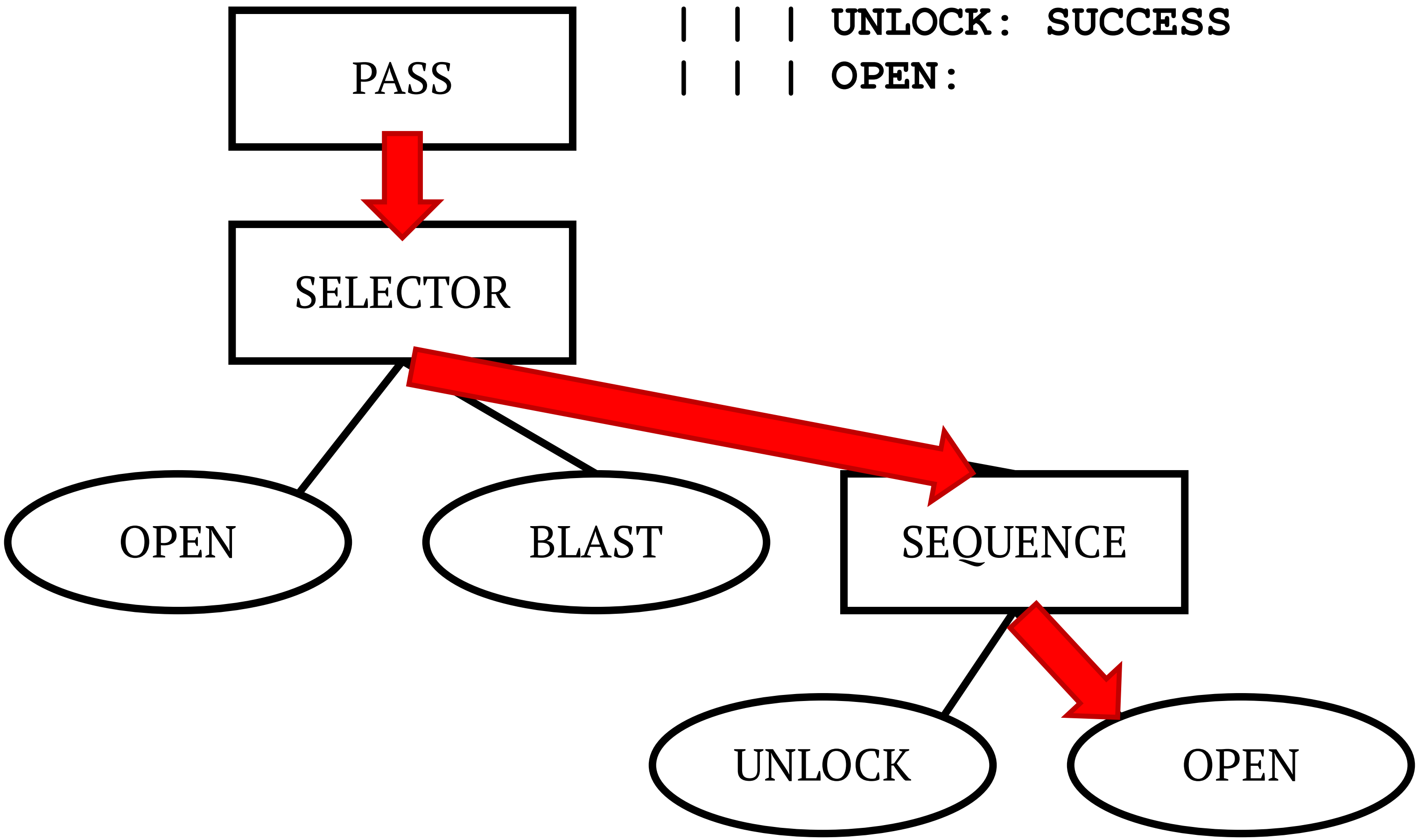
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
```



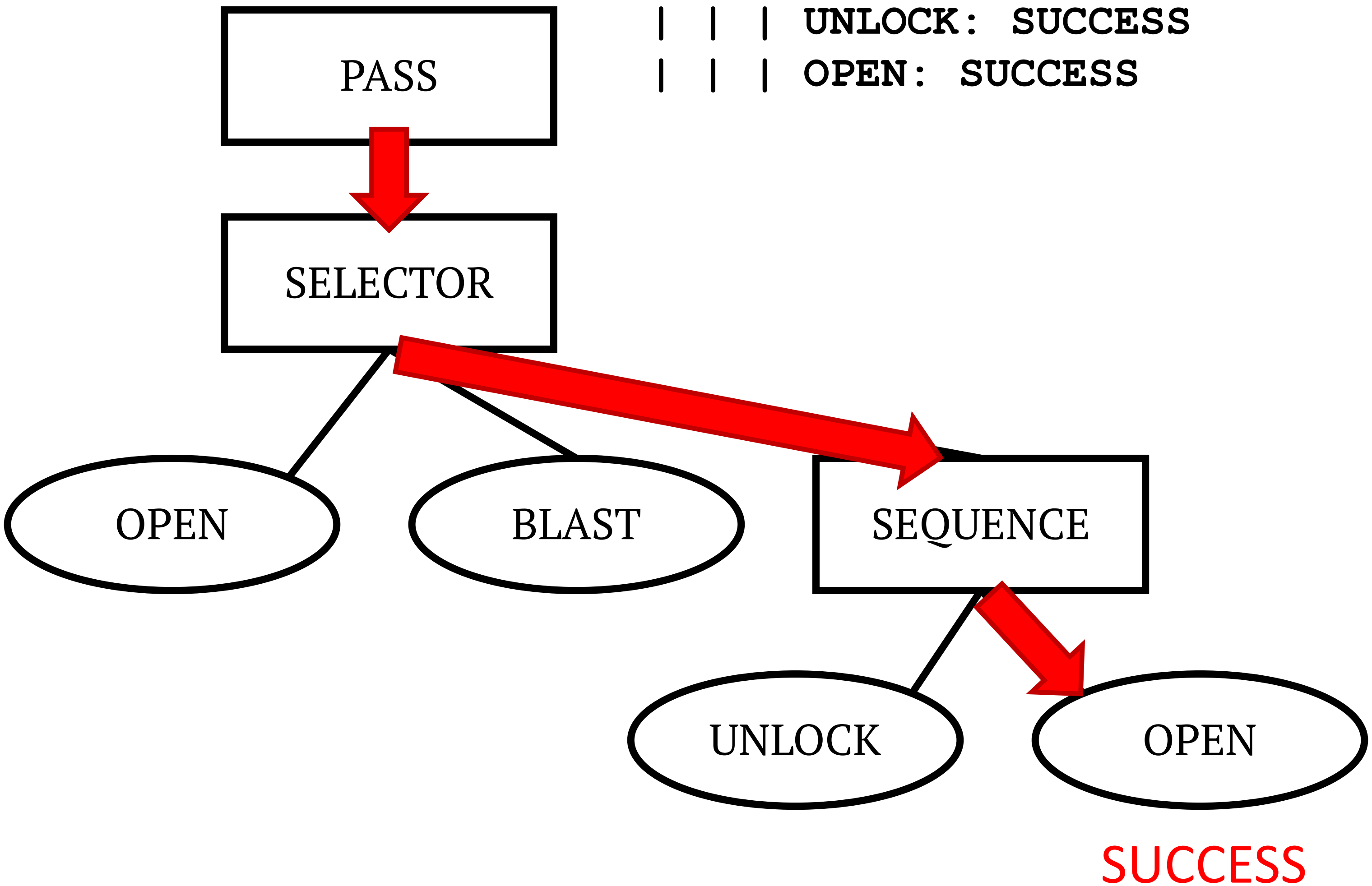
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
| | | OPEN:
```



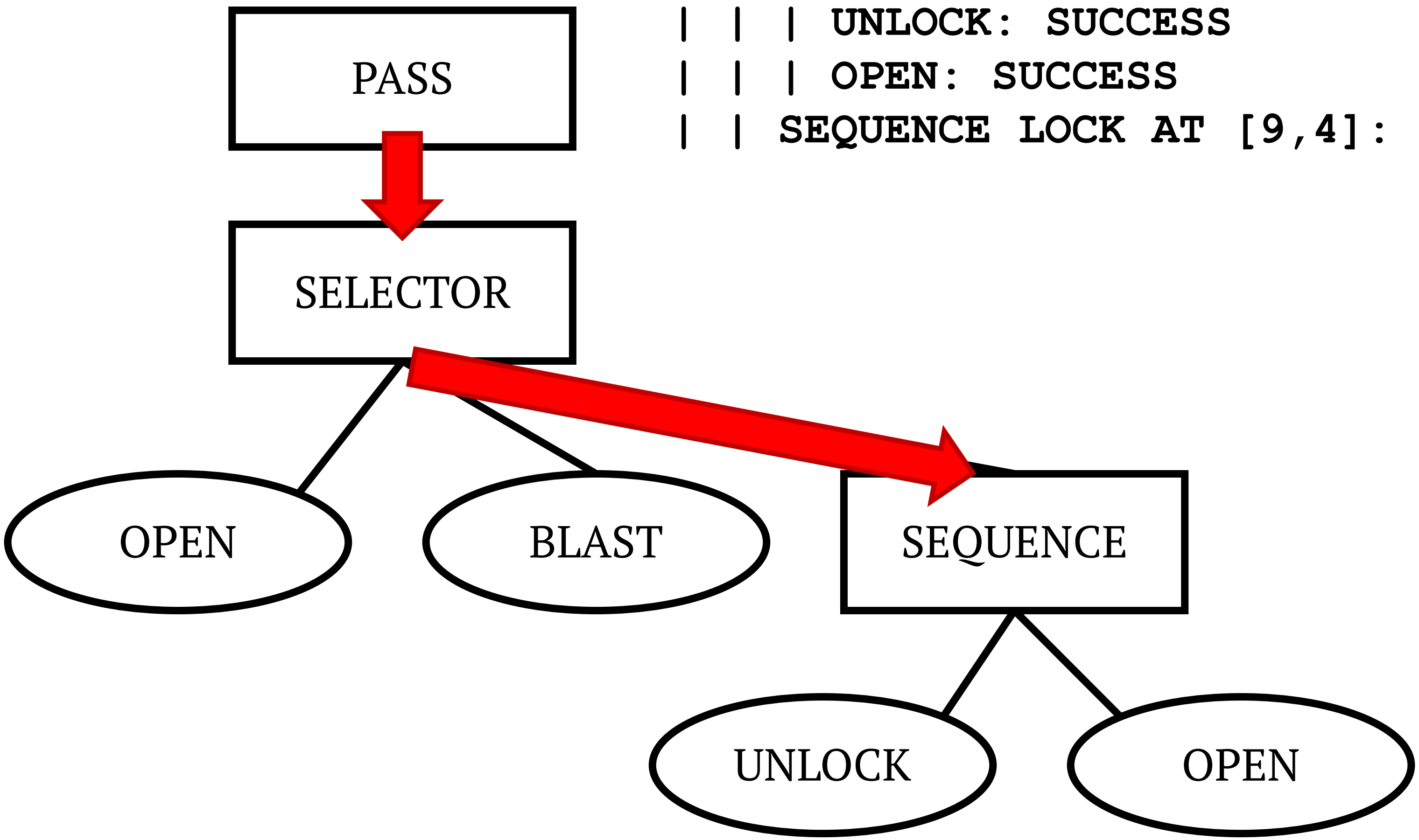
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
| | | OPEN: SUCCESS
```



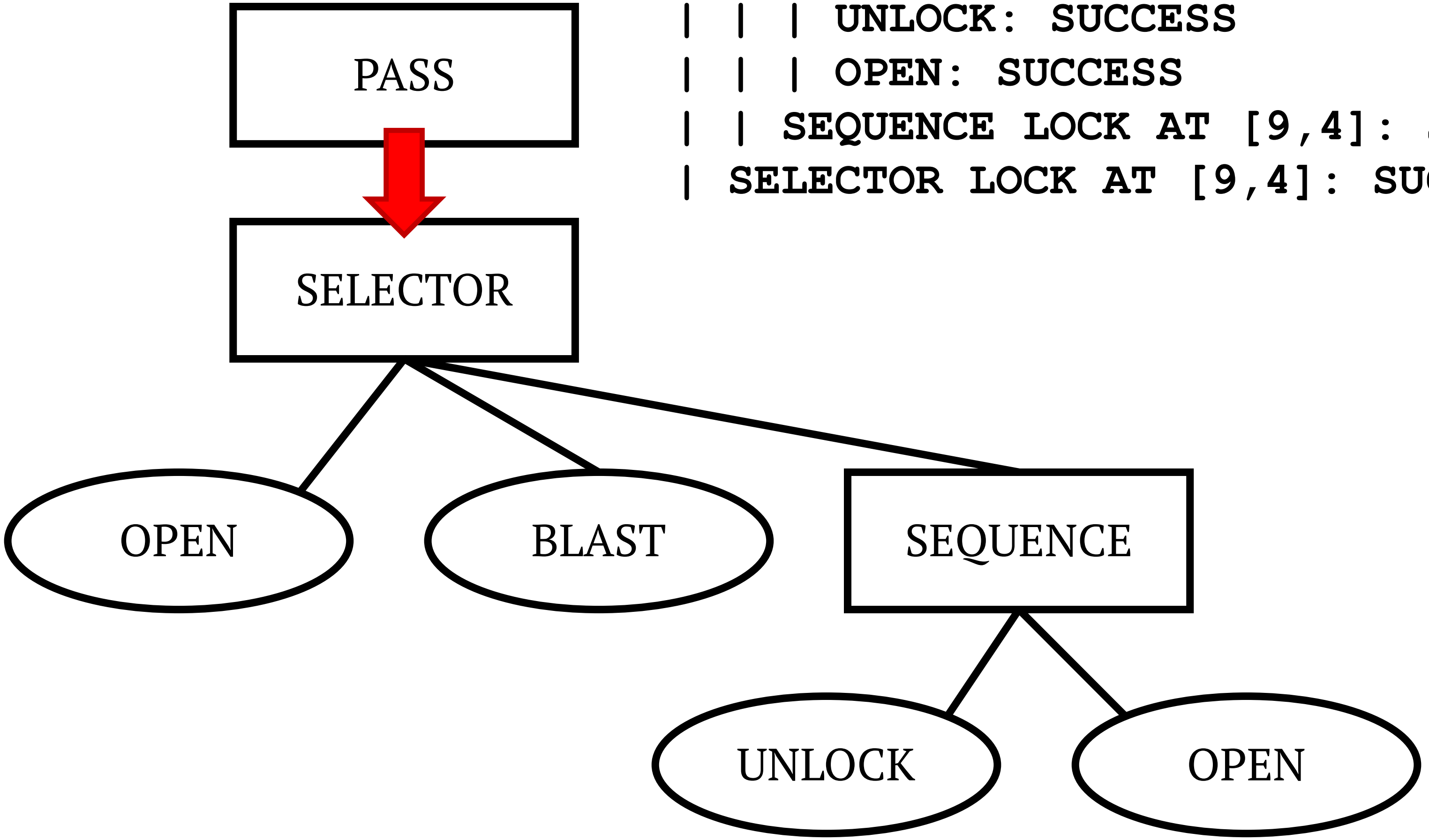
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
| | | OPEN: SUCCESS
| | SEQUENCE LOCK AT [9,4]: SUCCESS
```



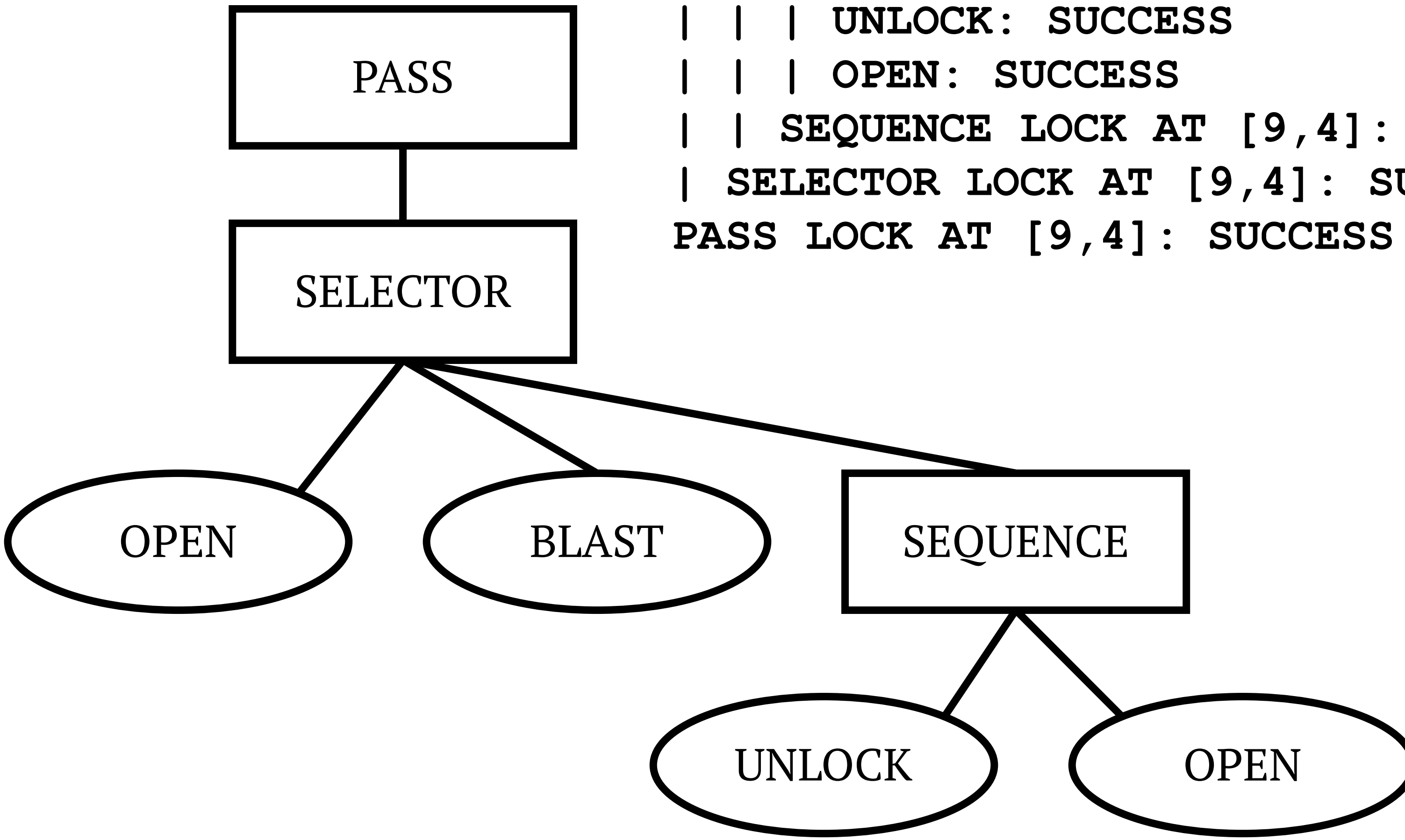
Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
| | | OPEN: SUCCESS
| | SEQUENCE LOCK AT [9,4]: SUCCESS
| SELECTOR LOCK AT [9,4]: SUCCESS
```



Execution

```
PASS LOCK AT [9,4]
| SELECTOR LOCK AT [9,4]
| | OPEN: FAILURE
| | BLAST: FAILURE
| | SEQUENCE LOCK AT [9,4]
| | | UNLOCK: SUCCESS
| | | OPEN: SUCCESS
| | SEQUENCE LOCK AT [9,4]: SUCCESS
| SELECTOR LOCK AT [9,4]: SUCCESS
PASS LOCK AT [9,4]: SUCCESS
```



Succeeder / Failer (Decotrator)

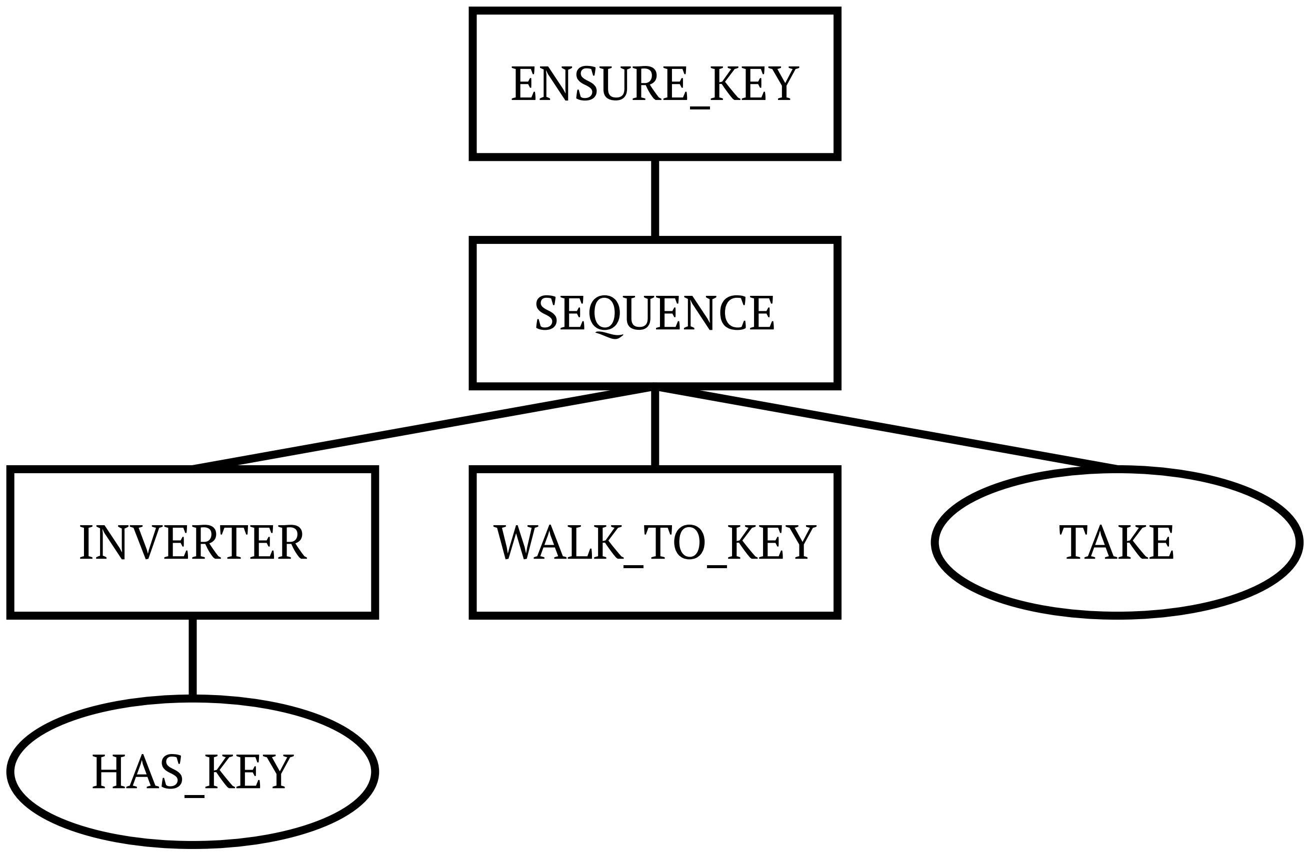
Succeeder: Call the child, but always return true no matter what.

Failer: Call the child, but always return false no matter what.

Inverter (Decorator)

Return the opposite of what the child returns.

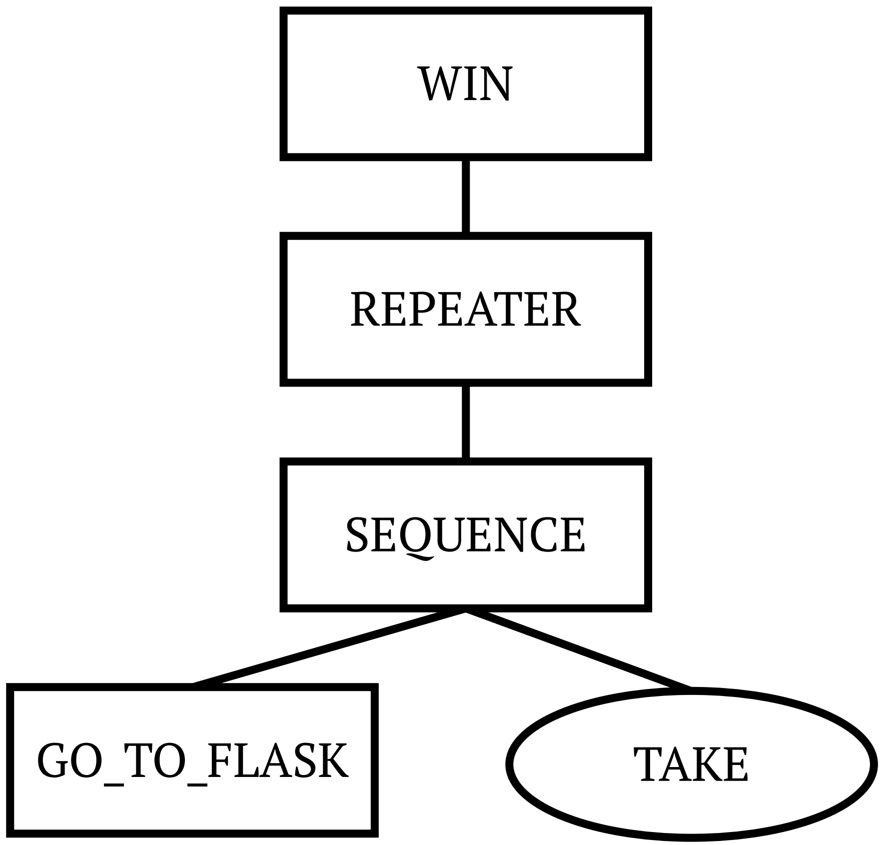
Inverter Example



Repeater (Decorator)

Call the child over and over until it returns true.

Repeater Example



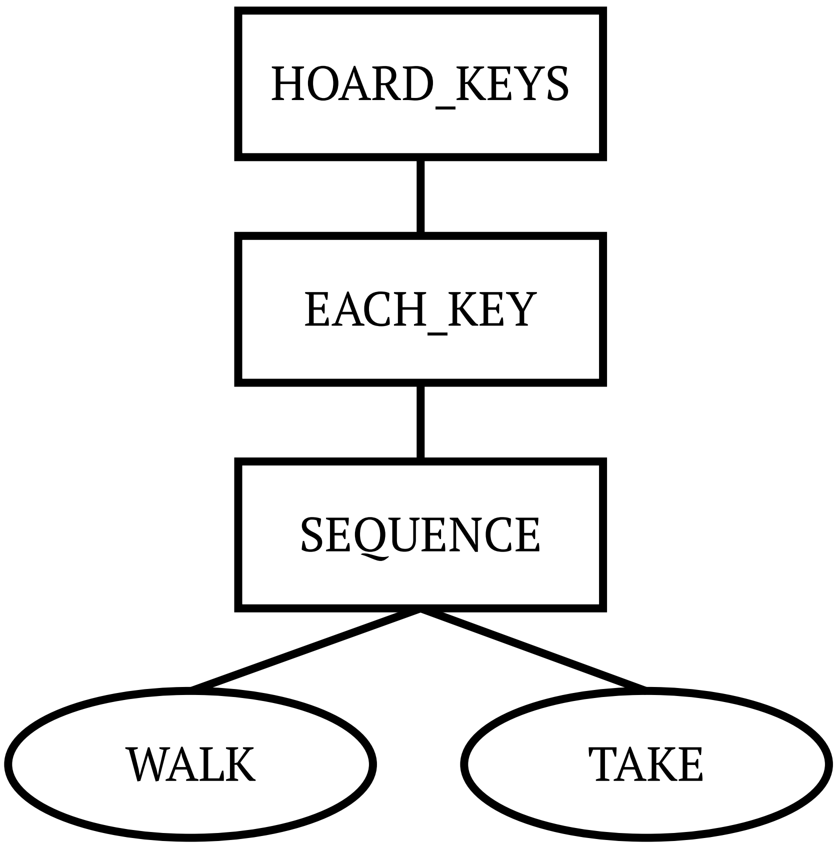
Iterator (Decorator)

Ignore the argument passed to this node. For each item of a certain type, pass that item as an argument to the child.

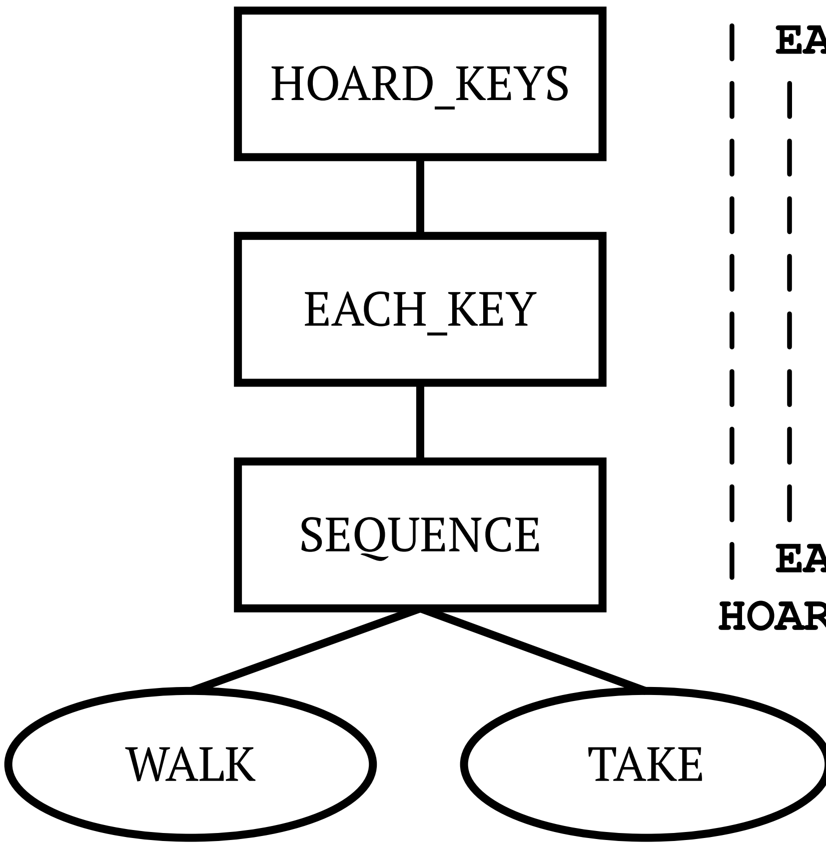
Each: If a child returns false, return false. If all children return true, return true.

Any: If a child returns true, return true. If all children return false, return false.

Iterator Example



Iterator Execution




```
HOARD
| EACH_KEY
| | SEQUENCE KEY AT [4,6]
| | | WALK: FAILURE
| | | TAKE: SUCCESS
| | SEQUENCE KEY AT [4,6]: SUCCESS
| | SEQUENCE KEY AT [7,5]
| | | WALK: SUCCESS
| | | TAKE: SUCCESS
| | SEQUENCE KEY AT [7,5]: SUCCESS
| EACH_KEY: SUCCESS
HOARD: SUCCESS
```

Node Types for this Exercise

- **Sequence** – do each until one fails
- **Selector** – do each until one succeeds
- **Inverter** – return the opposite of the child's value
- **Repeater** – repeat child until it succeeds
- **Each** – pass each item of a type to the child until the child fails
- **Any** – pass each item of a type to the child until the child succeeds