A Synchronous View on Behavior Trees

Reinhard von Hanxleden, Kiel University

Joint Work with Alexander Schulz-Rosengarten (U Kiel),
Benjamin Asch, Soroush Bateni, Marten Lohstroh, Edward Lee (UC Berkeley)

SYNCHRON 2022, Nov. 29, Fréjus, France

[Clemens.Ratte-Polle]

Praise for Behavior Trees

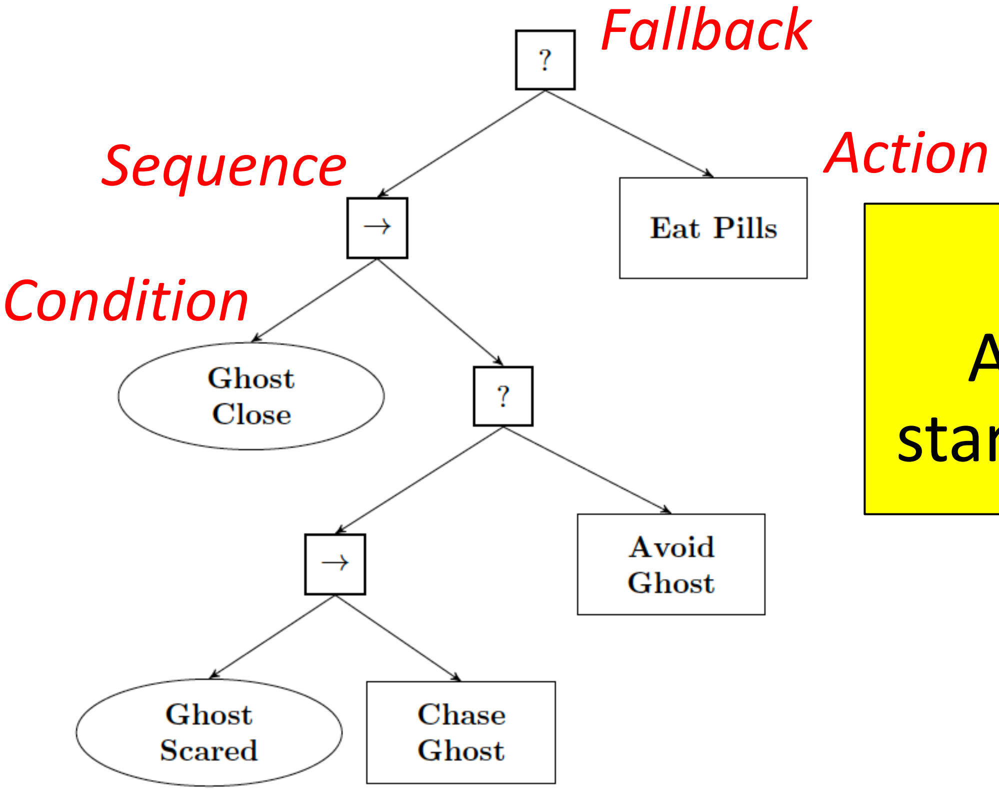
"[...]. Sure you could build the very same behaviors with a finite state machine (FSM). But anyone who has worked with this kind of technology in industry knows how fragile such logic gets as it grows. A finely tuned ***hierarchical FSM*** *before a game ships is often a temperamental work of art not to be messed with!"*

Alex J. Champandard
Editor in Chief & Founder AiGameDev.com,
Senior AI Programmer Rockstar Games

This quote and parts of the following material taken from
[Colledanchise Ogren '20]
Michele Colledanchise and Petter Ogren,
Behavior Trees in Robotics and AI - An Introduction, 2020

https://arxiv.org/pdf/1709.00084.pdf

```
if ghost is close
    then if ghost is scared
        then chase ghost
        else avoid ghost
    else eat pills
```



Key Point:

At every tick, start at **root** of BT

Figs from [Colledanchise Ogren ’20]

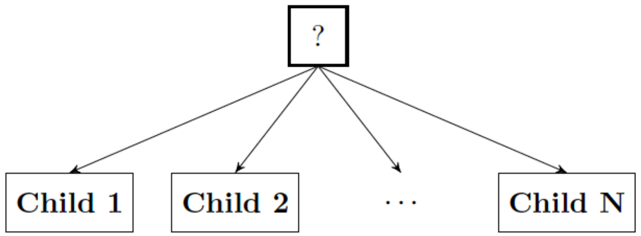


Fig. 1.3: Graphical representation of a Fallback node with N children.

Algorithm 2: Pseudocode of a Fallback node with N children

```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Success$ then
        return $Success$
return $Failure$
```

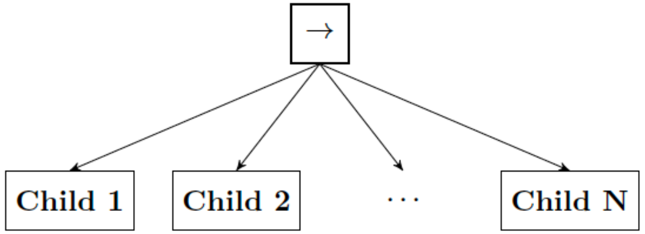


Fig. 1.2: Graphical representation of a Sequence node with N children.

Algorithm 1: Pseudocode of a Sequence node with N children

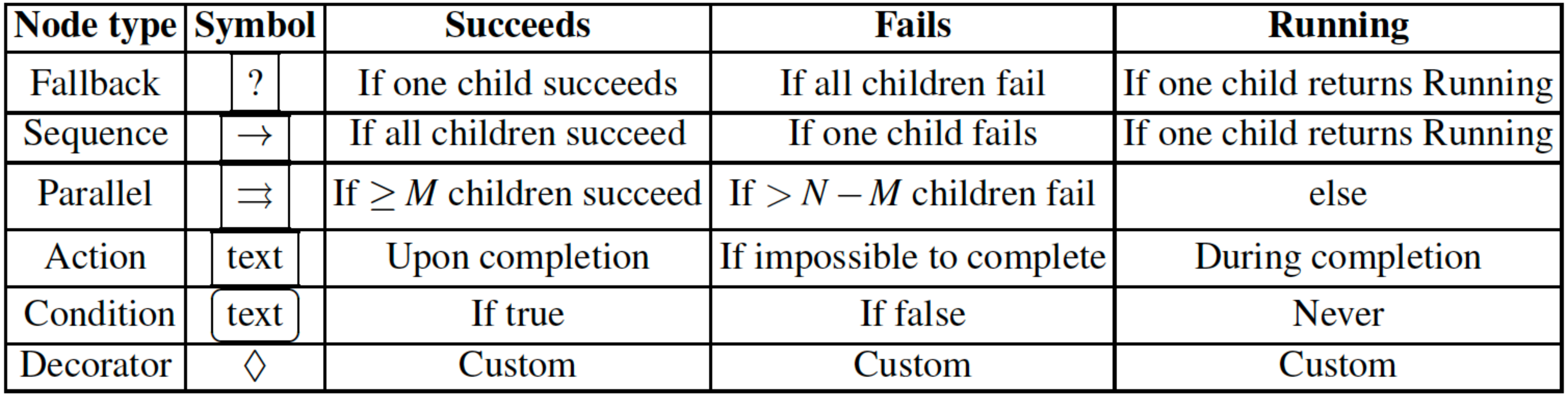
```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Failure$ then
        return $Failure$
return $Success$
```

Behavior Tree Building Blocks

Control Flow nodes: Sequence, Fallback/Selector, Parallel, Decorator

Execution nodes: Action/Task, Condition

Possible return values: Success, Running, Failure



Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Table from [Colledanchise Ogren ’20]

Behavior Trees in Lingua Franca

Preliminary work …

Modes in BTs – The “Select Mode Pattern”

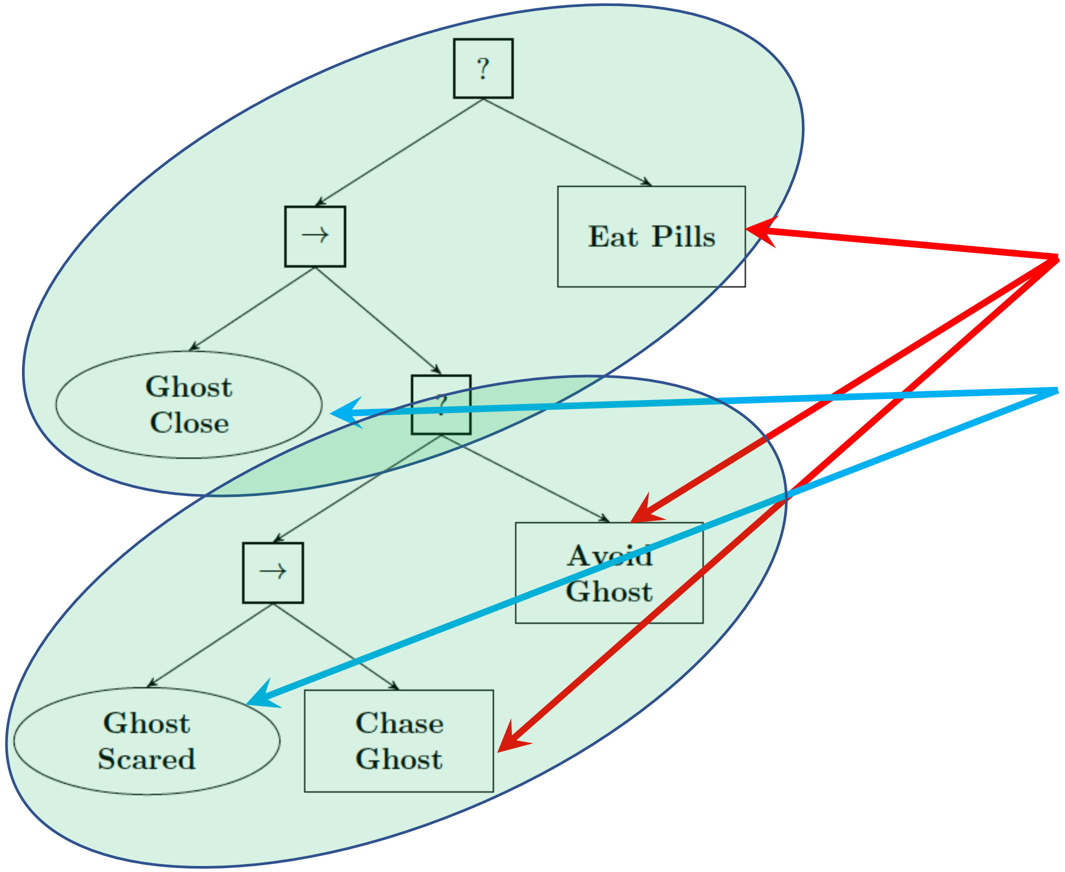
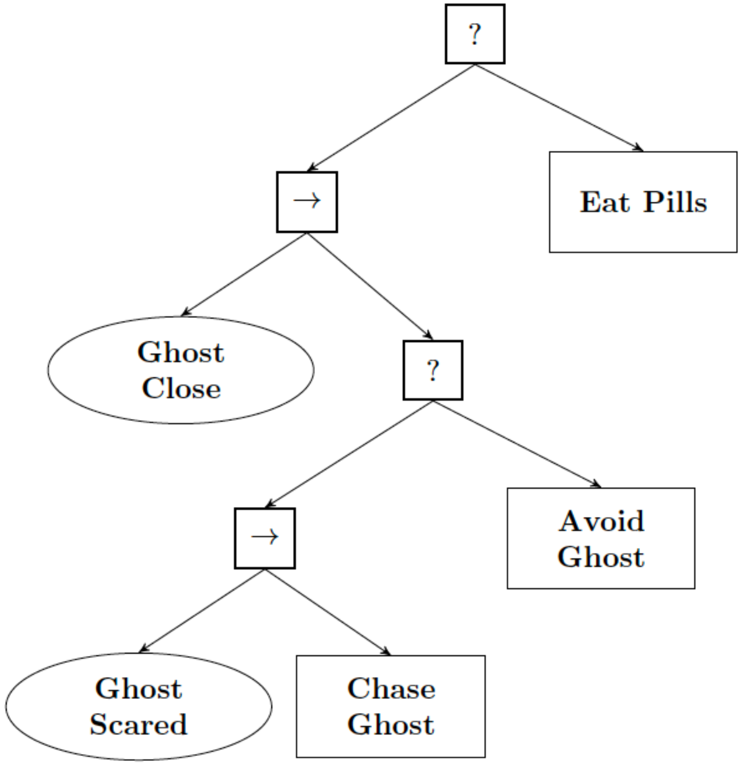


Fig from [Colledanchise Ogren ’20]

Observations:

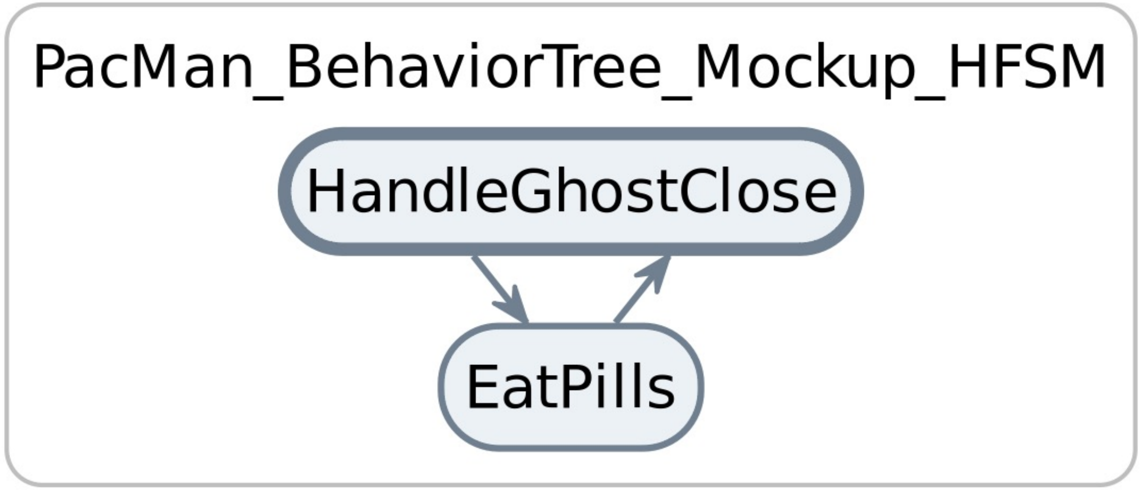
- System has three modes (always return *Running*)
- Current mode determined by two conditions (return *Success* or *Failure*)
- Nested use of “select mode pattern” (our term), where one condition switches between two (inner) modes

Select Mode Pattern in LF

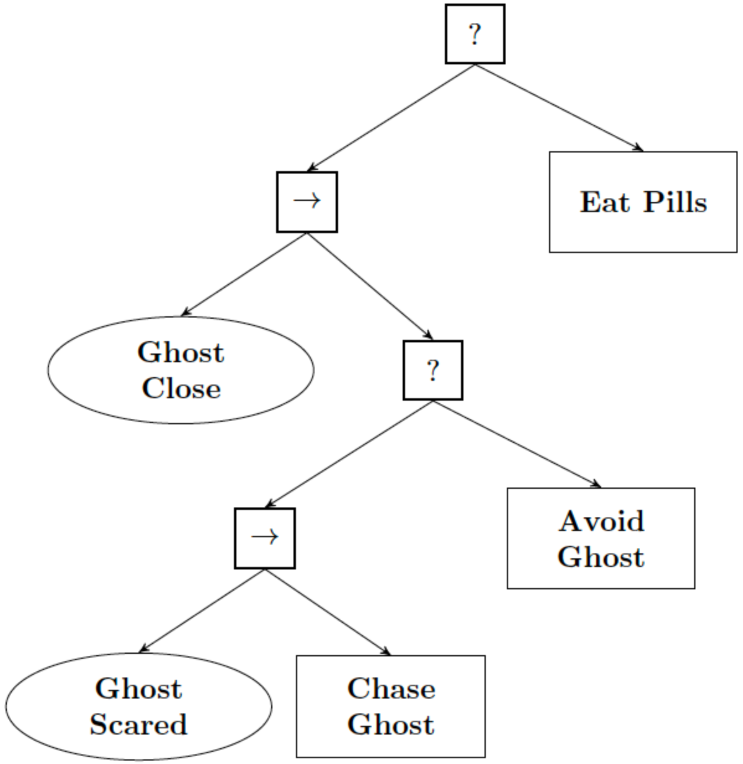


[Colledanchise Ogren '20]

Inner modes collapsed:

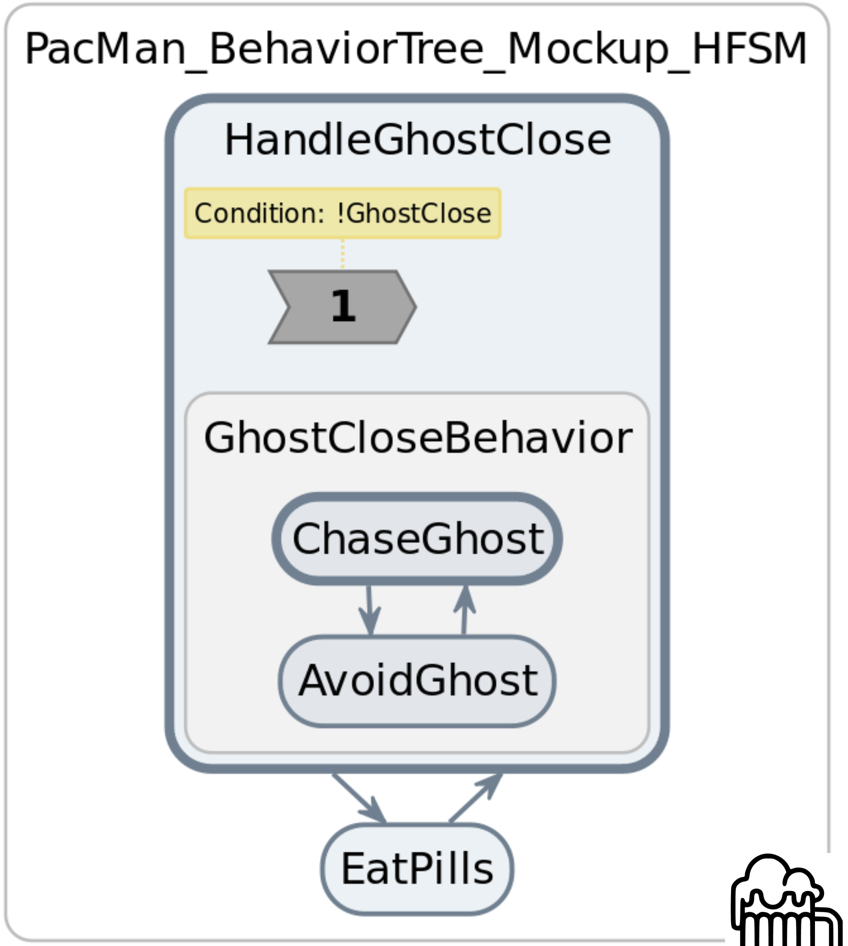


Select Mode Pattern in LF

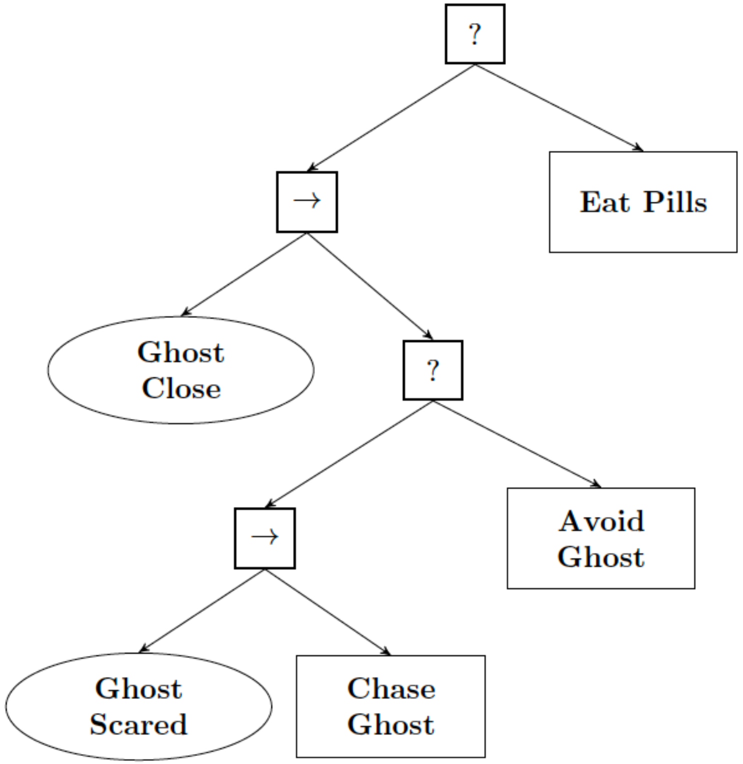


[Colledanchise Ogren ’20]

Inner modes expanded:

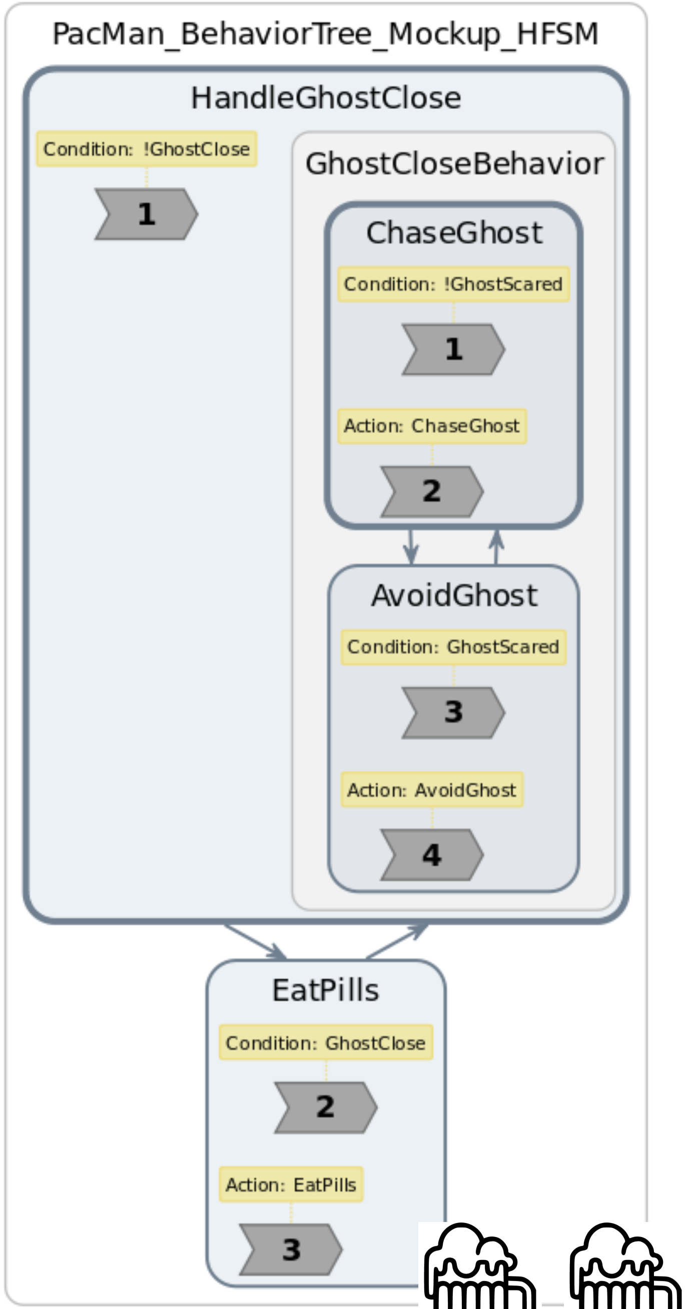


Select Mode Pattern in LF

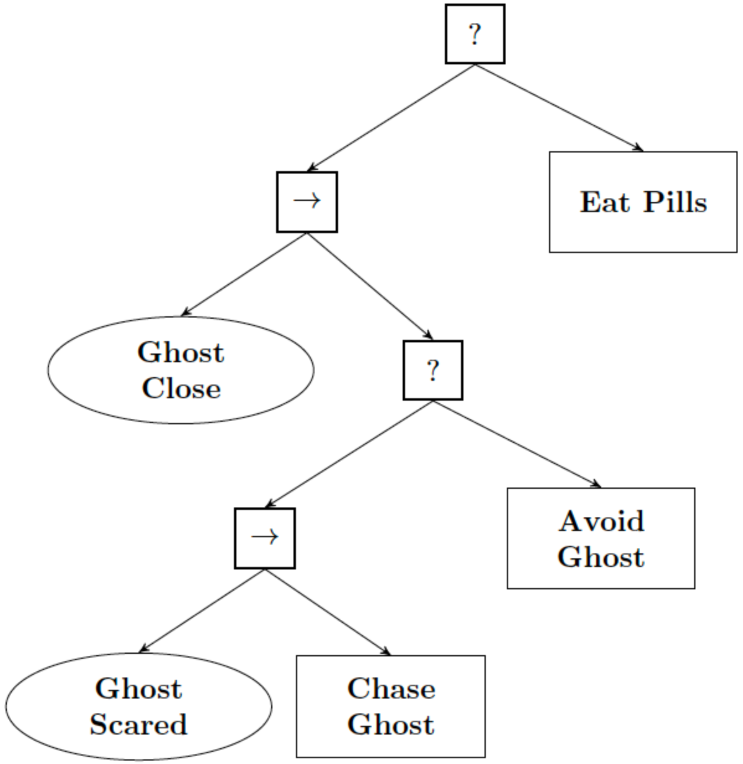


[Colledanchise Ogren '20]

With reactions:

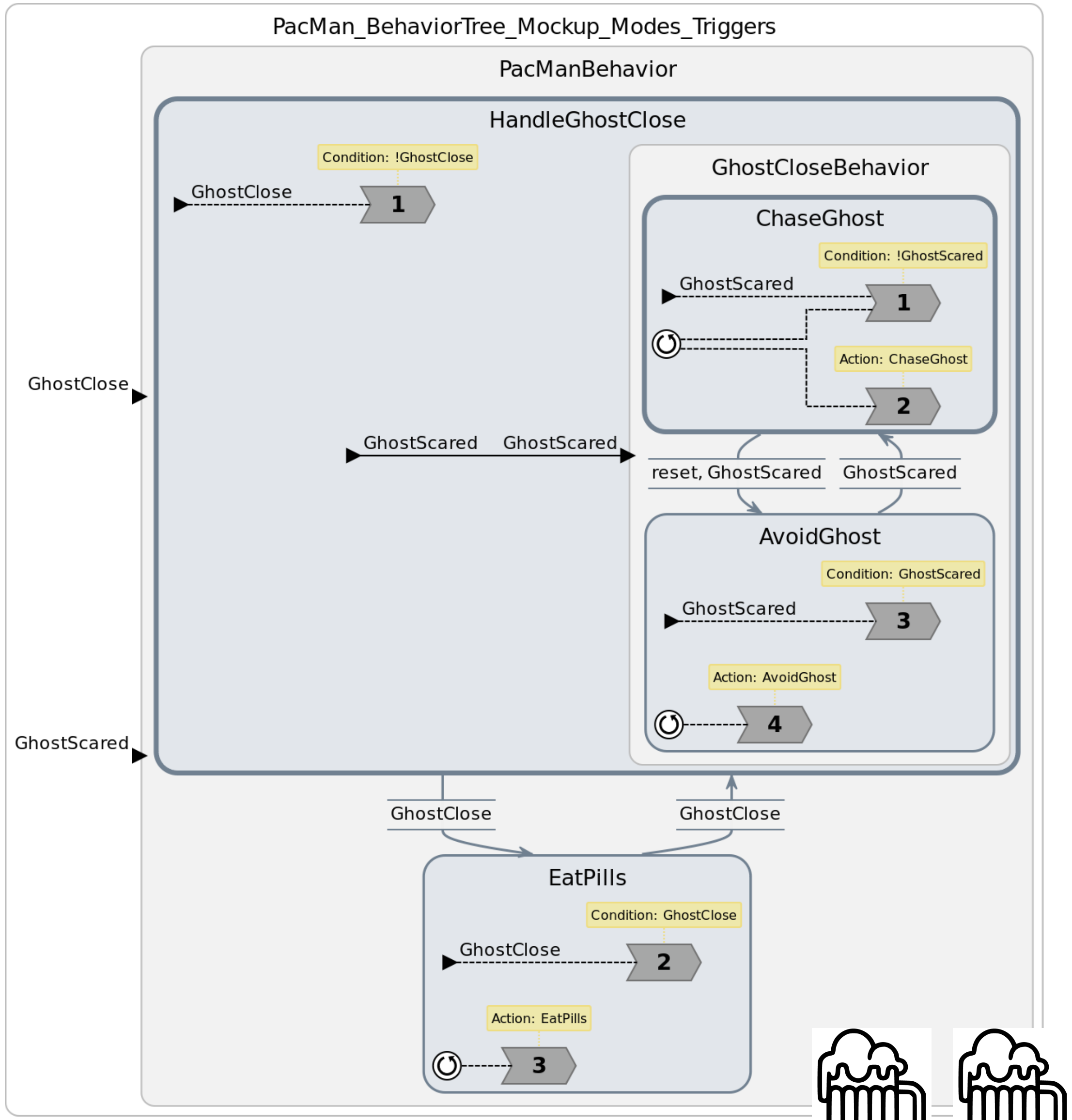


Select Mode Pattern in LF



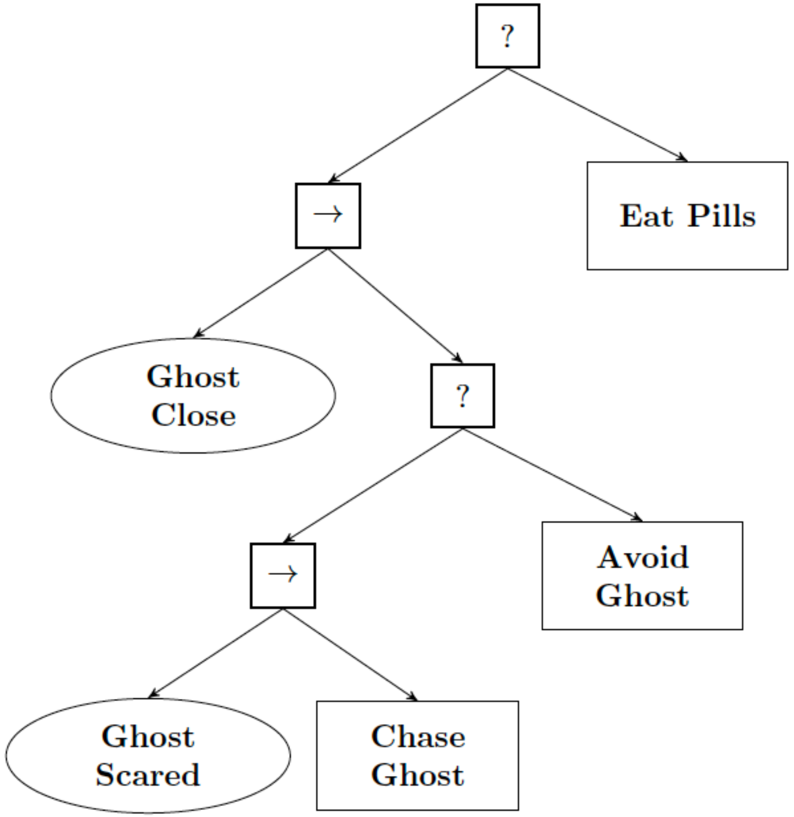
[Colledanchise Ogren ’20]

With conditions:

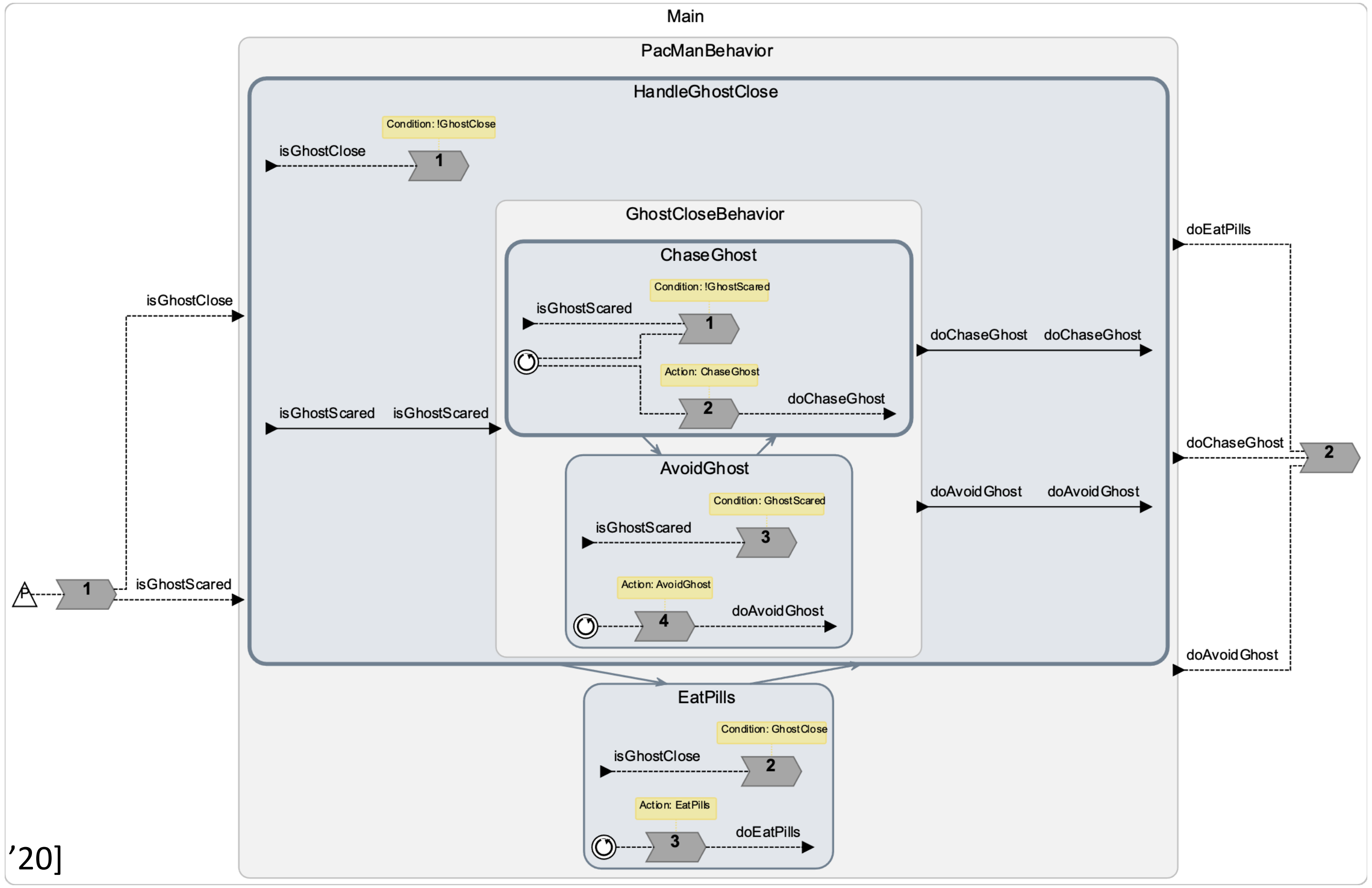


With Boolean outputs:

Select Mode Pattern in LF

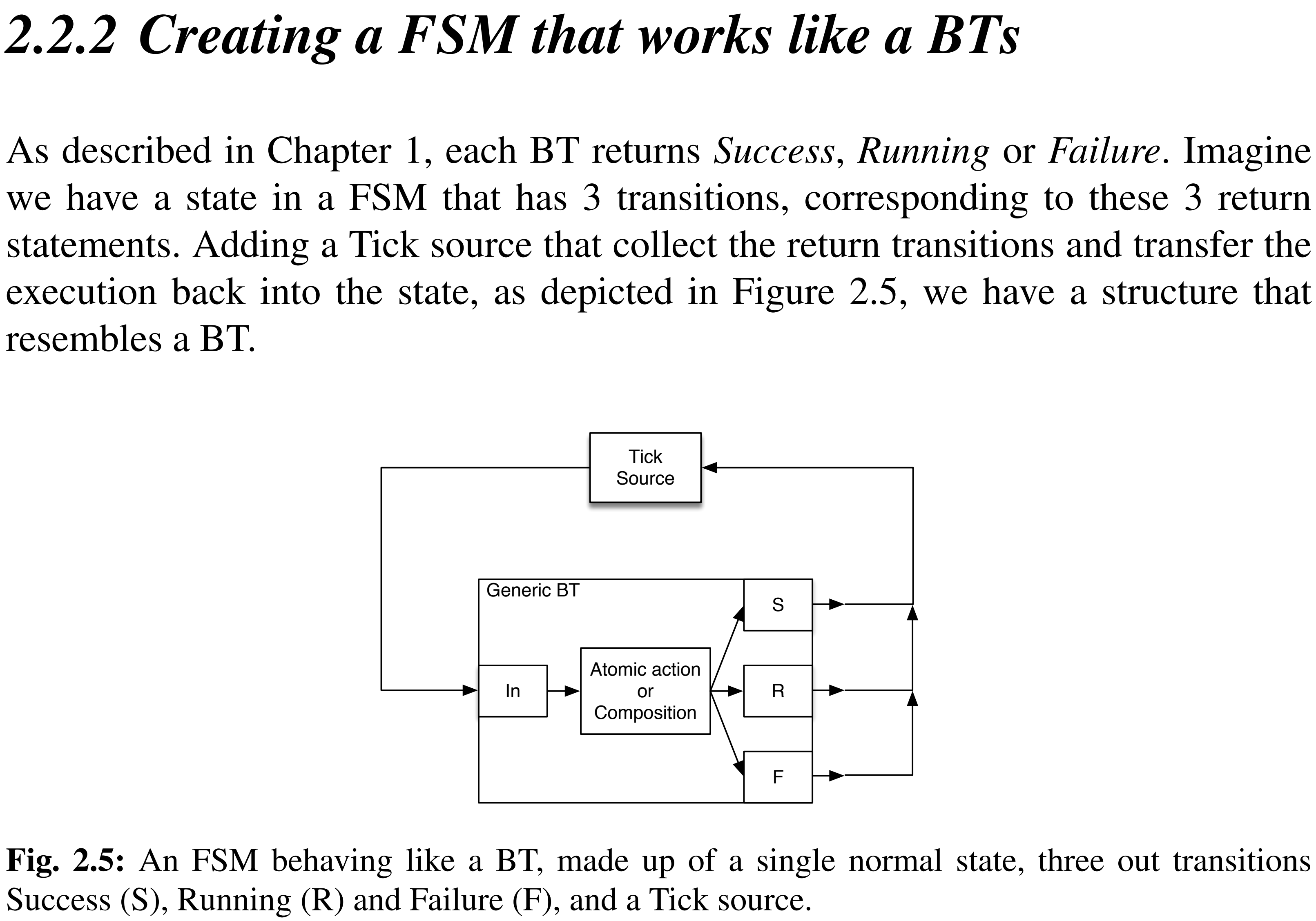


[Colledanchise Ogren '20]



This works …
But is specific for this pattern

FSM Construction by Colledanchise and Ogren



2.2.2 *Creating a FSM that works like a BTs*

As described in Chapter 1, each BT returns *Success*, *Running* or *Failure*. Imagine we have a state in a FSM that has 3 transitions, corresponding to these 3 return statements. Adding a Tick source that collect the return transitions and transfer the execution back into the state, as depicted in Figure 2.5, we have a structure that resembles a BT.

Fig. 2.5: An FSM behaving like a BT, made up of a single normal state, three out transitions Success (S), Running (R) and Failure (F), and a Tick source.

[Colledanchise Ogren ’20]

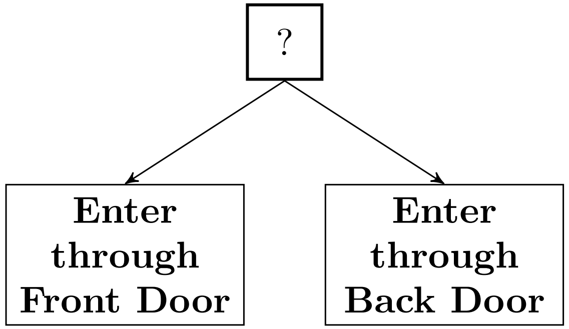


Fig. 2.6: A Fallback is used to create an *Enter Building* BT. The back door option is only tried if the front door option fails.

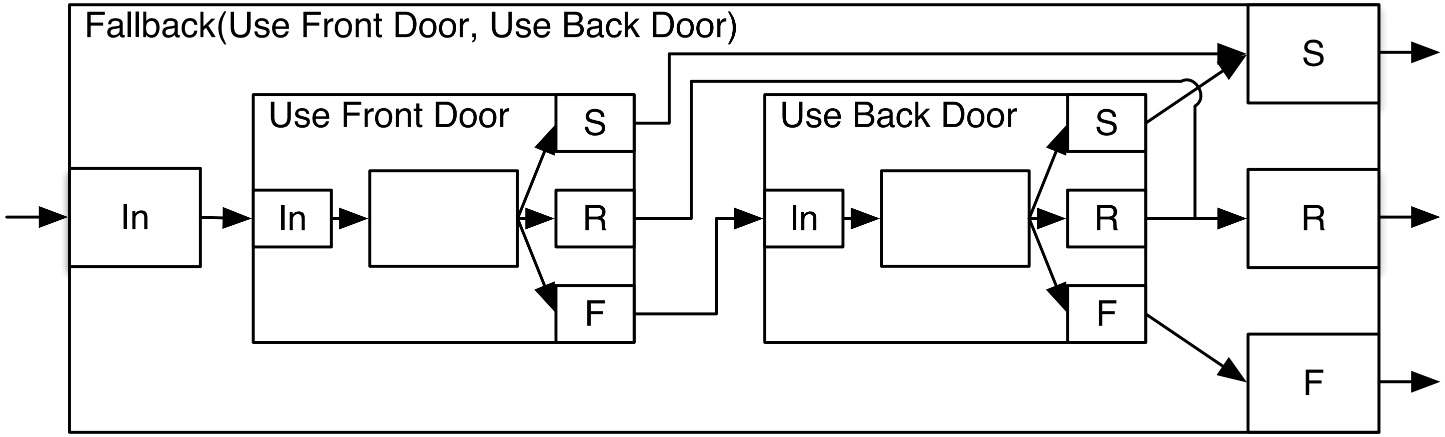


Fig. 2.7: A FSM corresponding to the Fallback BT in Figure 2.6. Note how the second state is only executed if the first fails.

[Colledanchise Ogren ’20]

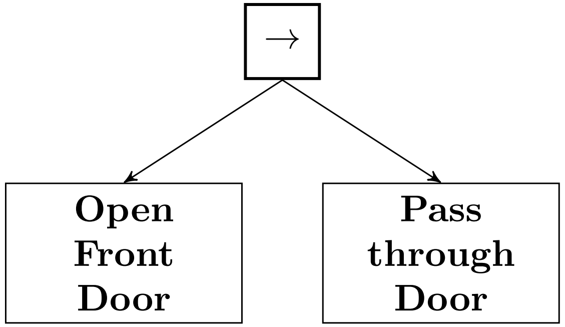


Fig. 2.8: A Sequence is used to to create an *Enter Through Front Door* BT. Passing the door is only tried if the opening action succeeds.

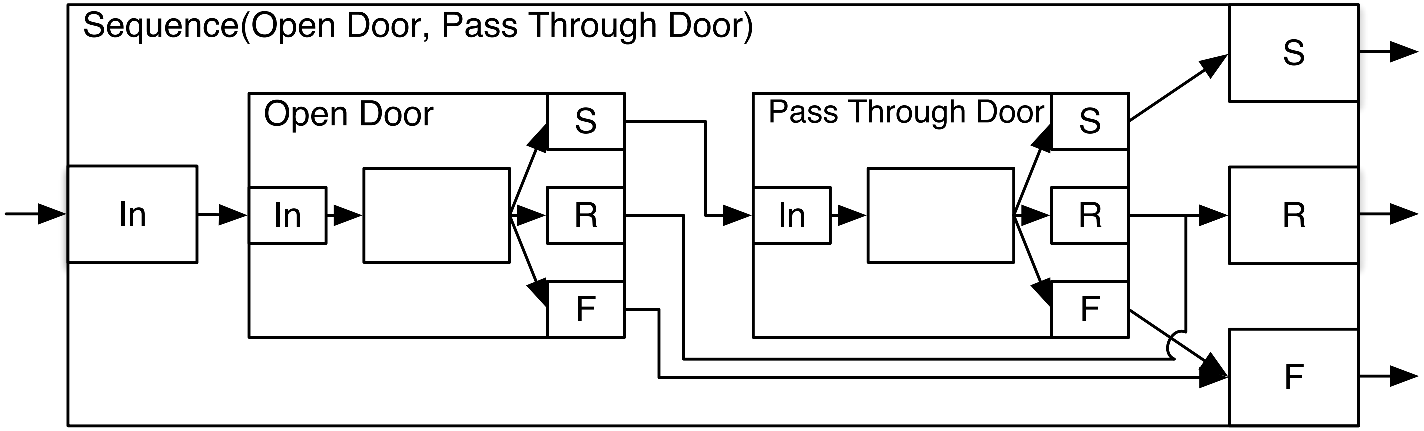


Fig. 2.9: An FSM corresponding to the Sequence BT in Figure 2.8. Note how the second state is only executed if the first succeeds.

[Colledanchise Ogren '20]

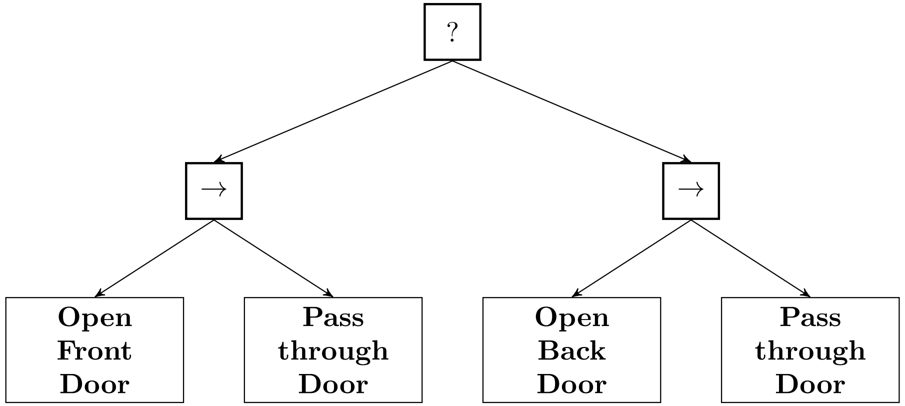


Fig. 2.10: The two BTs in Figures 2.6 and 2.8 are combined to larger BT. If e.g. the robot opens the front door, but does not manage to pass through it, it will try the back door.

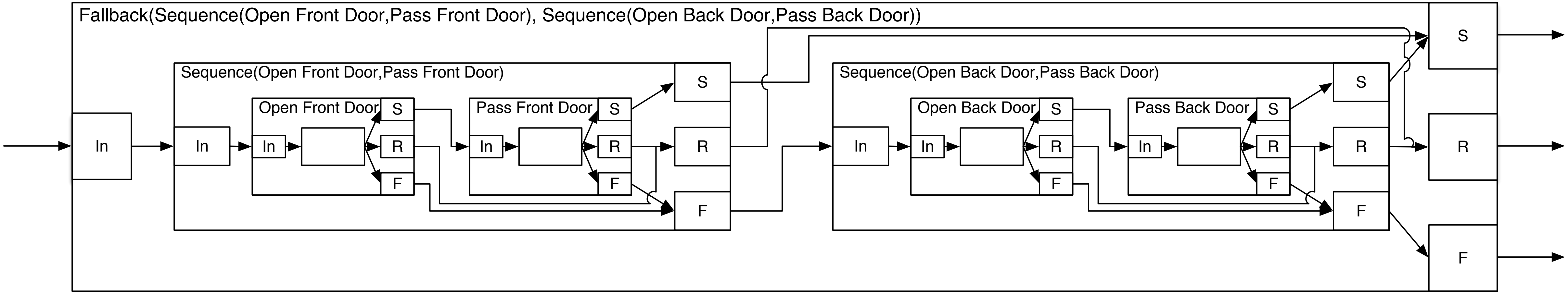

Fallback(Sequence(Open Front Door,Pass Front Door), Sequence(Open Back Door,Pass Back Door))
Sequence(Open Front Door,Pass Front Door)
Open Front Door
Pass Front Door
Sequence(Open Back Door,Pass Back Door)
Open Back Door
Pass Back Door
In
S
R
F

Fig. 2.11: An FSM corresponding to the BT in Figure 2.10.

[Colledanchise Ogren ’20]

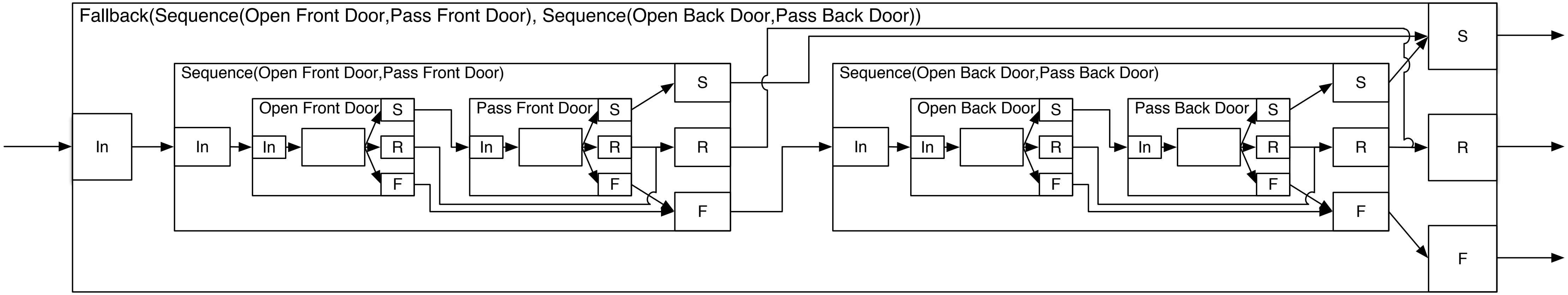

Fallback(Sequence(Open Front Door,Pass Front Door), Sequence(Open Back Door,Pass Back Door))
Sequence(Open Front Door,Pass Front Door)
Sequence(Open Back Door,Pass Back Door)
Open Front Door
Pass Front Door
Open Back Door
Pass Back Door
In
S
R
F

Fig. 2.11: An FSM corresponding to the BT in Figure 2.10.

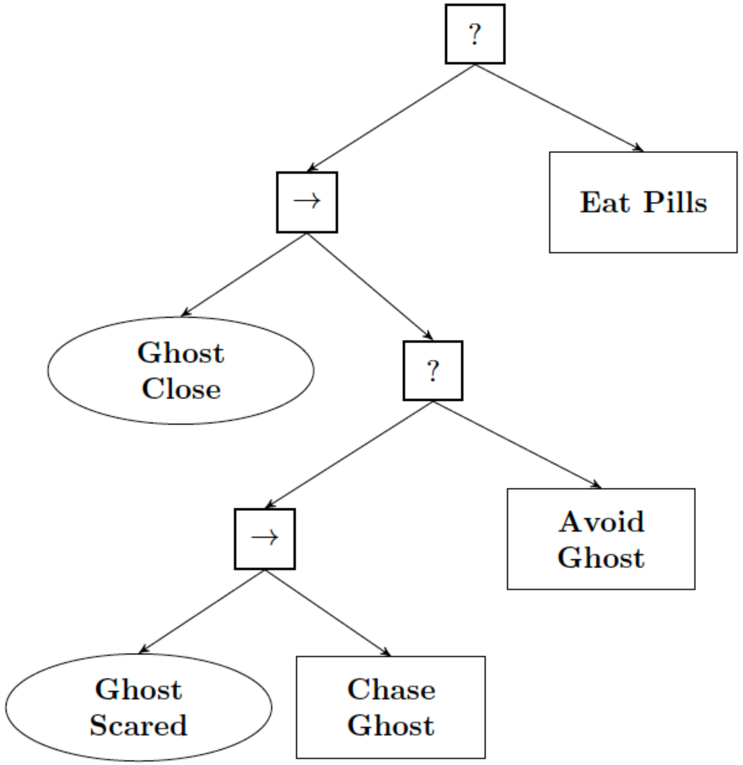
[Colledanchise Ogren ’20]

Observations/Claims:

- The nodes are not really “states,” but actors that fire when receiving an input
- The edges are not really “transitions,” but denote data (token) flow
- “Running” just denotes completion of reaction in absence of Success/Failure

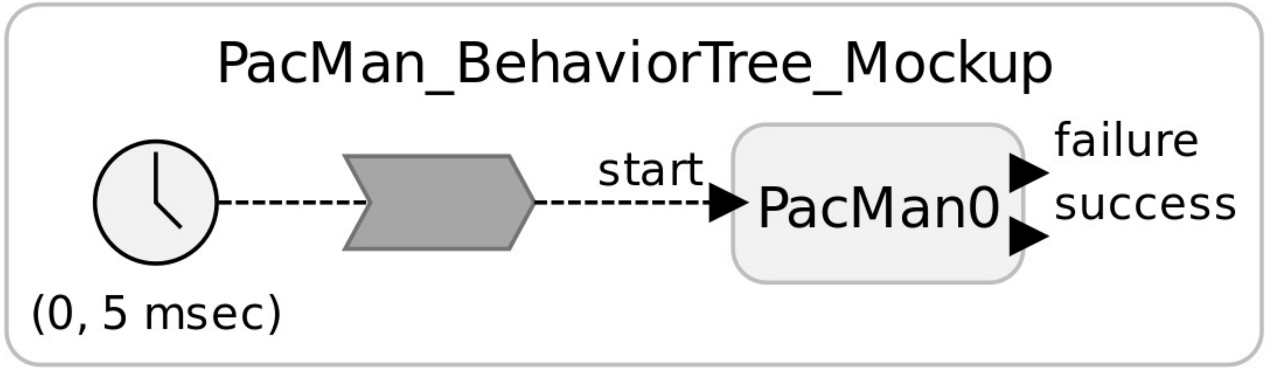
Conclusion:

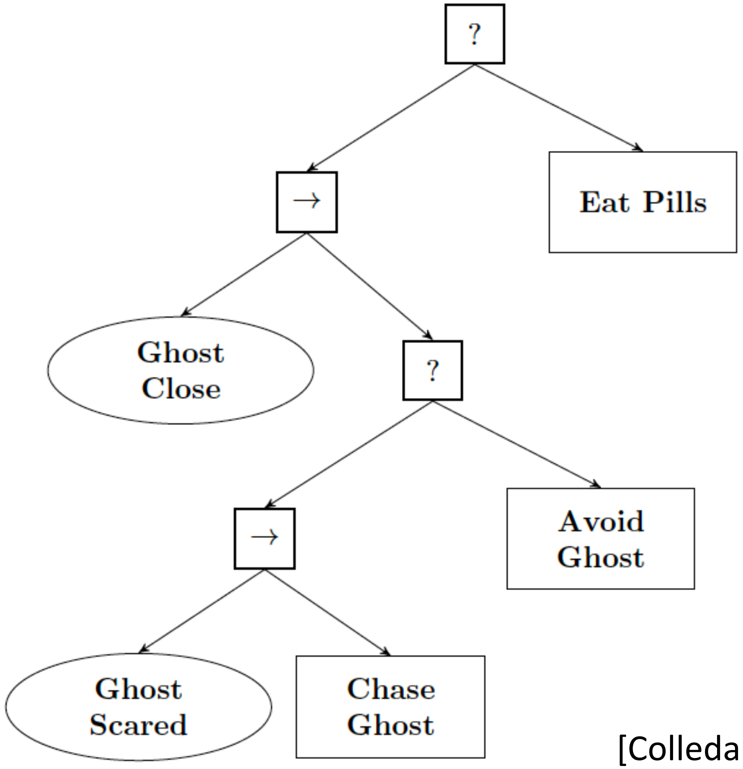
- Can map this directly to LF reactors!



[Colledanchise Ogren ’20]

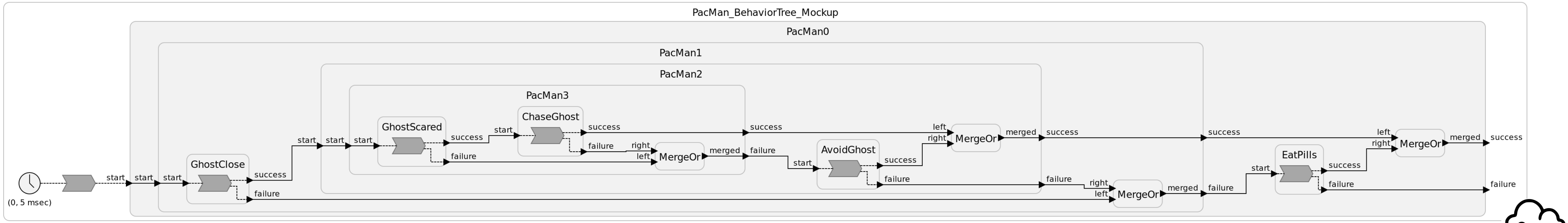
Top-Level:



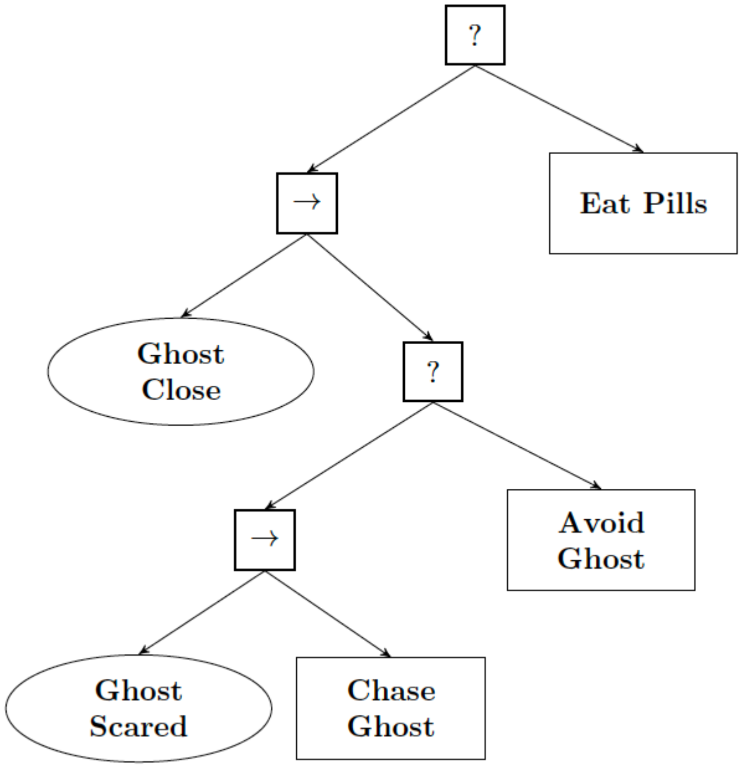


[Colledanchise Ogren ’20]

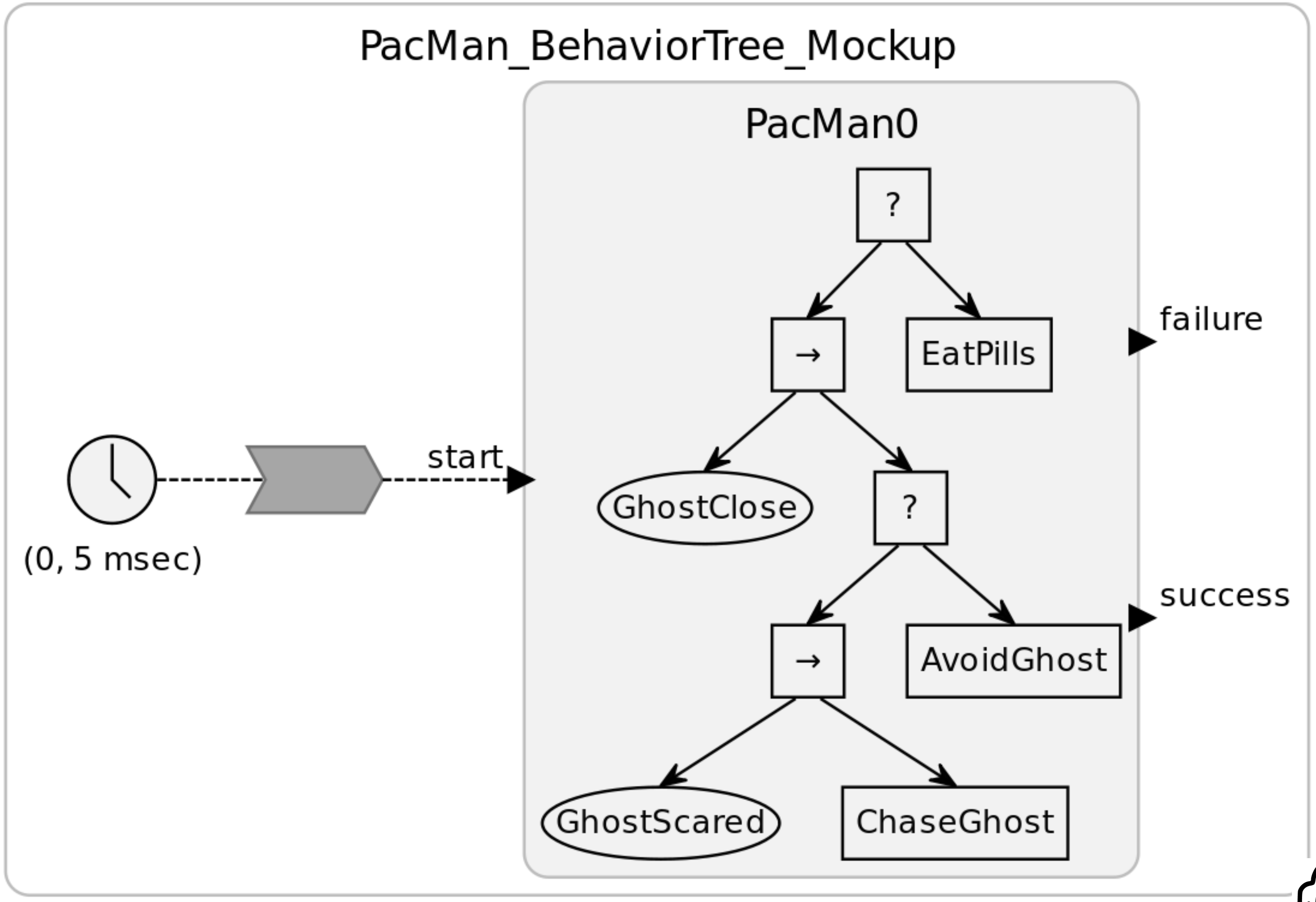
All Reactors:



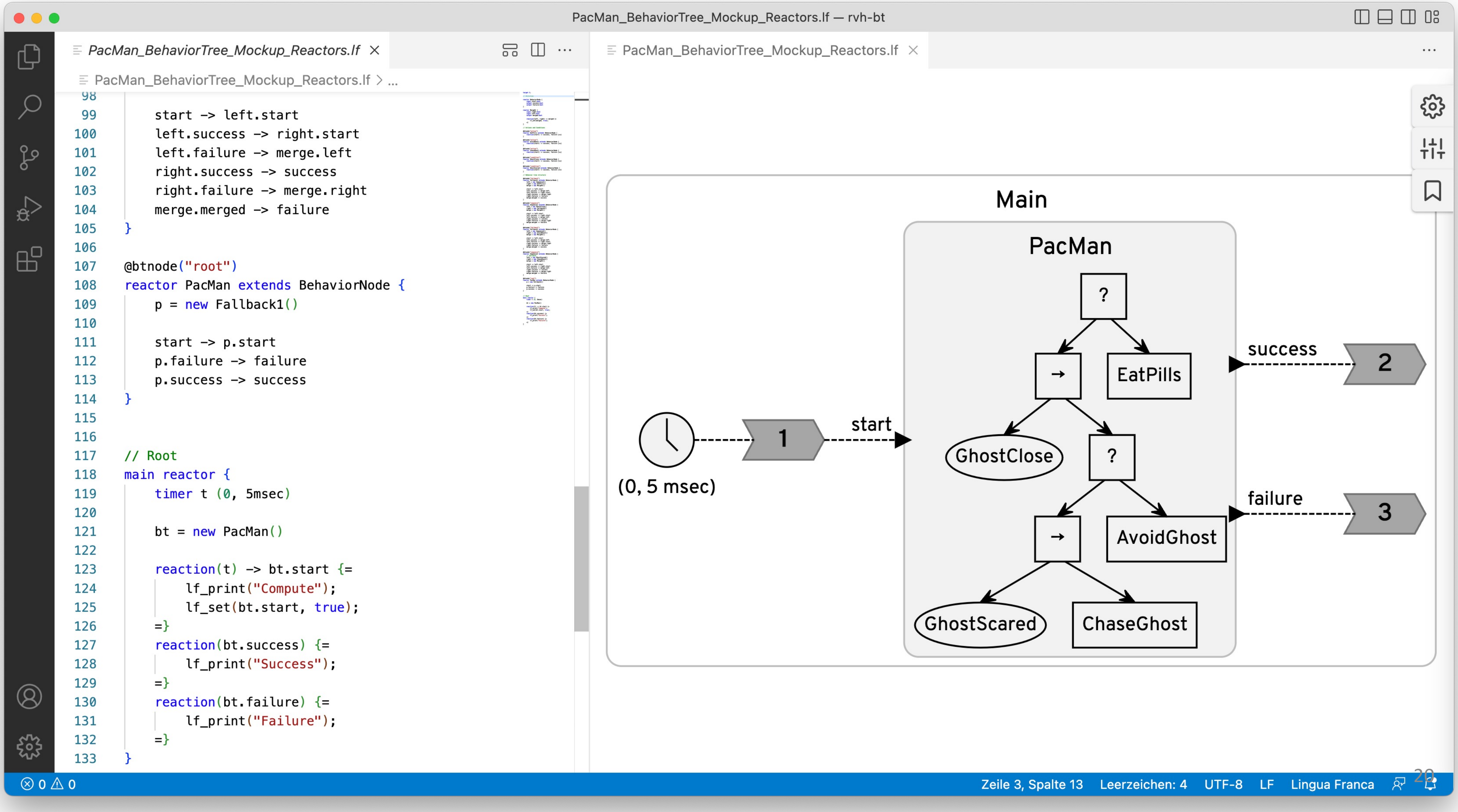
With “Behavior Tree View”:



[Colledanchise Ogren ’20]



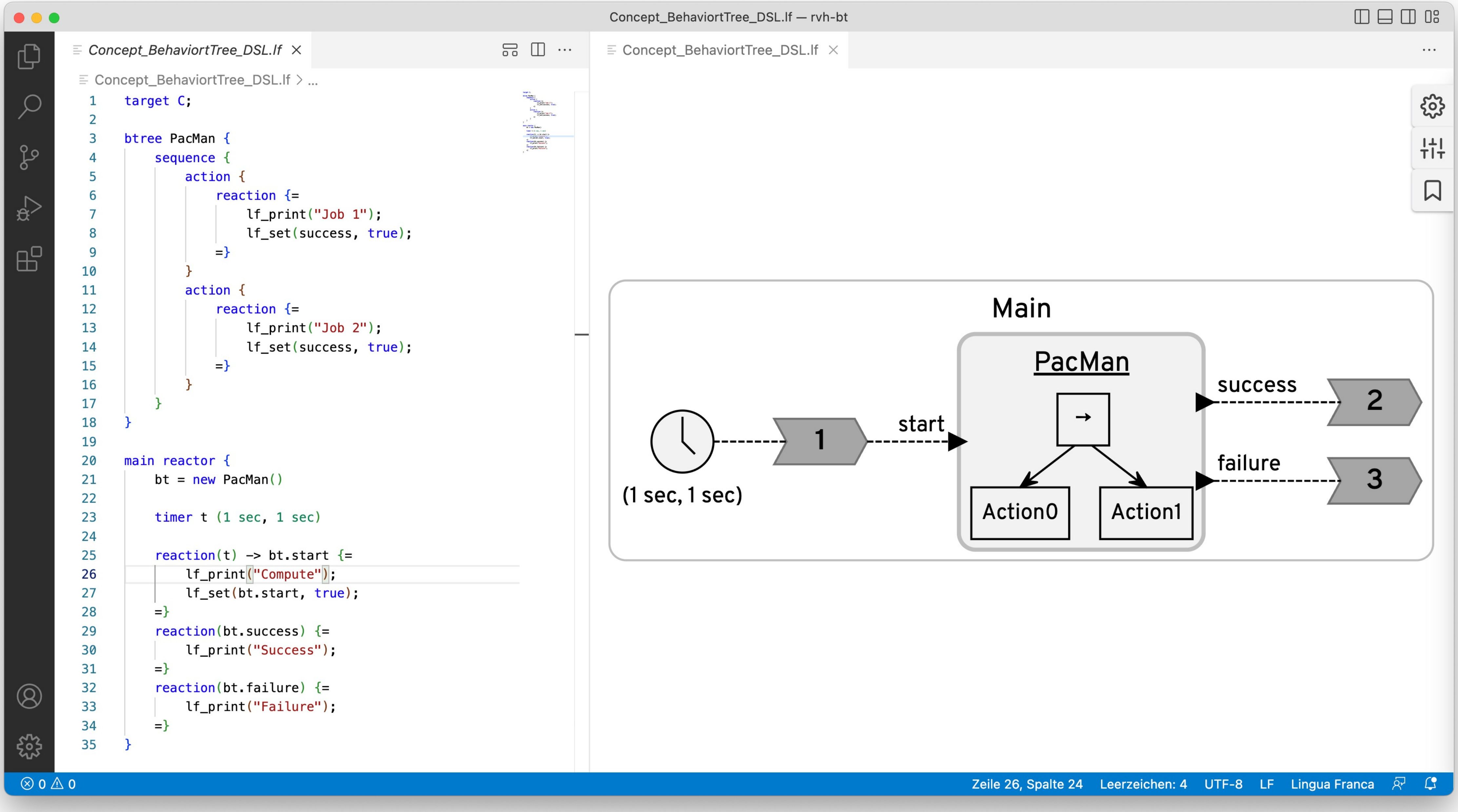
Behavior Trees/Lingua Franca in VS Code



PacMan_BehaviorTree_Mockup_Reactors.lf — rvh-bt
PacMan_BehaviorTree_Mockup_Reactors.lf
PacMan_BehaviorTree_Mockup_Reactors.lf > ...
PacMan_BehaviorTree_Mockup_Reactors.lf
Main
PacMan
?
→
EatPills
GhostClose
?
→
AvoidGhost
GhostScared
ChaseGhost
(0, 5 msec)
1
start
success
2
failure
3
⊗ 0 ⚠ 0
Zeile 3, Spalte 13
Leerzeichen: 4
UTF-8
LF
Lingua Franca

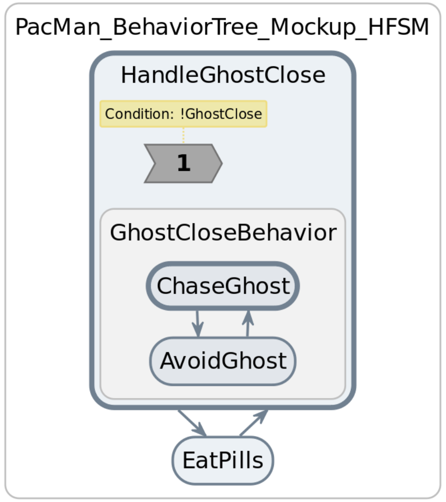
```
    start -> left.start
    left.success -> right.start
    left.failure -> merge.left
    right.success -> success
    right.failure -> merge.right
    merge.merged -> failure
}

@btnode("root")
reactor PacMan extends BehaviorNode {
    p = new Fallback1()

    start -> p.start
    p.failure -> failure
    p.success -> success
}


// Root
main reactor {
    timer t (0, 5msec)

    bt = new PacMan()

    reaction(t) -> bt.start {=
        lf_print("Compute");
        lf_set(bt.start, true);
    =}
    reaction(bt.success) {=
        lf_print("Success");
    =}
    reaction(bt.failure) {=
        lf_print("Failure");
    =}
}
```

Ongoing Work: DSL for Behavior Trees in Lingua Franca

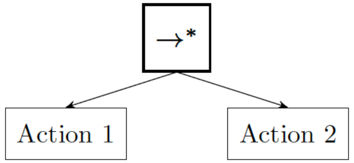

Concept_BehaviortTree_DSL.lf — rvh-bt
Concept_BehaviortTree_DSL.lf
Concept_BehaviortTree_DSL.lf > ...
Concept_BehaviortTree_DSL.lf
Main
PacMan
→
Action0
Action1
1
start
success
2
failure
3
(1 sec, 1 sec)
⊗ 0 ⚠ 0
Zeile 26, Spalte 24
Leerzeichen: 4
UTF-8
LF
Lingua Franca

```
target C;

btree PacMan {
    sequence {
        action {
            reaction {=
                lf_print("Job 1");
                lf_set(success, true);
            =}
        }
        action {
            reaction {=
                lf_print("Job 2");
                lf_set(success, true);
            =}
        }
    }
}

main reactor {
    bt = new PacMan()

    timer t (1 sec, 1 sec)

    reaction(t) -> bt.start {=
        lf_print("Compute");
        lf_set(bt.start, true);
    =}
    reaction(bt.success) {=
        lf_print("Success");
    =}
    reaction(bt.failure) {=
        lf_print("Failure");
    =}
}
```

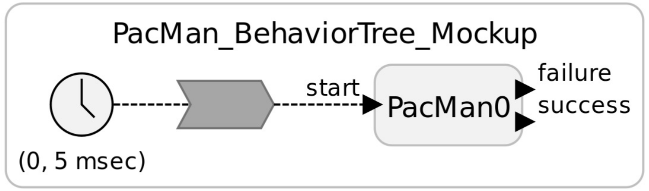
Wrap-Up – BTs in Lingua Franca

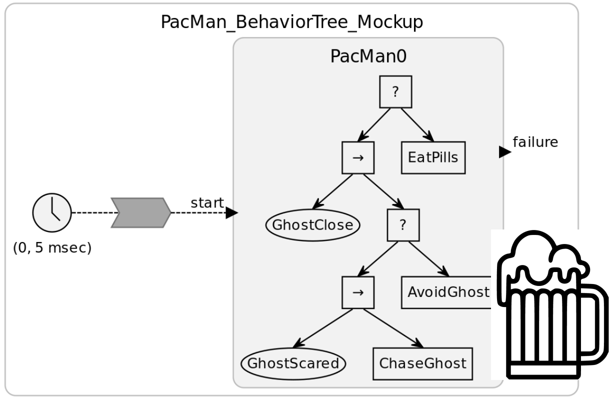


- Truly modal BTs can be expressed with modal reactors
  - Suitable for “select mode pattern”
  - Must consider that modal transitions are by default “deferred”
  - May also apply for BT nodes “with memory” (asterisk decorators)



- For general, “reactive” BTs, an actor model may be more suitable
  - Modular construction
  - Success + Failure communicated to neighbor/parent
  - Can use existing LF language
  - Can synthesize pictorial BTs within LF diagrams, resulting in hybrid data flow/BT views





Behavior Trees in Esterel

Preliminary work, barely …

Recall (Some) Basic Esterel Operators

`s1 ; s2`	Run s1, s2 sequentially
`s1 || s2`	Run s1, s2 in parallel
`pause`	Finish tick (terminate with completion code 1)
`trap T in s end`	Declare trap scope
`exit T`	Exit trap (terminate with completion code 2 or higher)

Example:

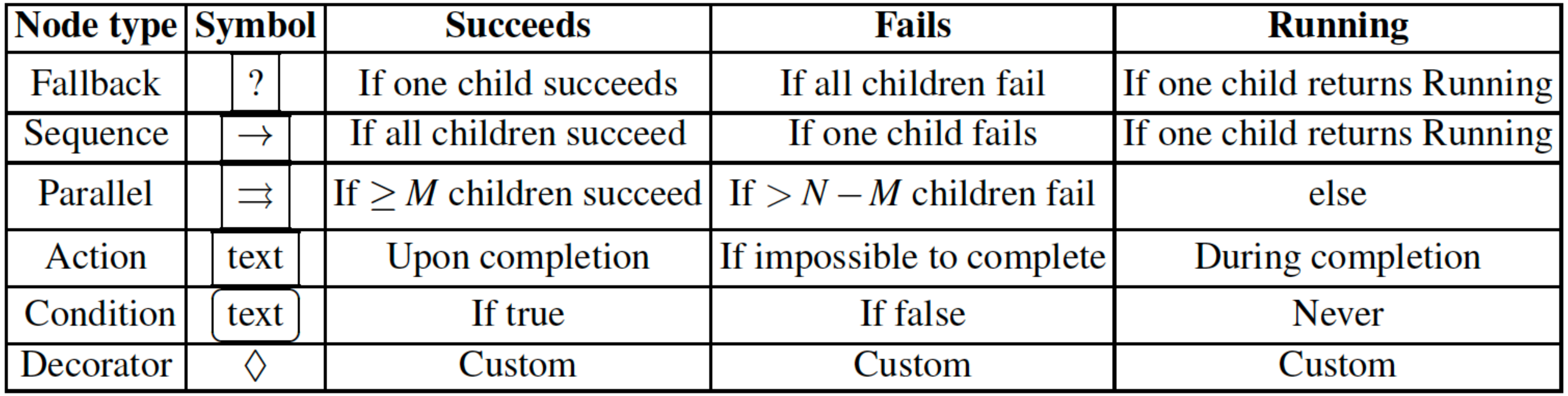
```
trap T in
  present I then exit T end;      // If I holds in first tick: terminate whole program
  pause;
  emit O                          // Otherwise: emit O in second tick
end
```

Mapping BTs to Esterel

Observation: return values correspond nicely to completion codes in Esterel.

Esterel in turn can be mapped to hierarchical FSMs,
which should also work for LF modal models

0	– (normal) termination	– “Succeeds”
2 (and higher)	– throw exception	– “Fails”
1	– pause operation	– “Running”



Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	$\rightarrow$	If all children succeed	If one child fails	If one child returns Running
Parallel	$\rightrightarrows$	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	$\Diamond$	Custom	Custom	Custom

[Colledanchise Ogren ’20]

“Parallel” in Esterel

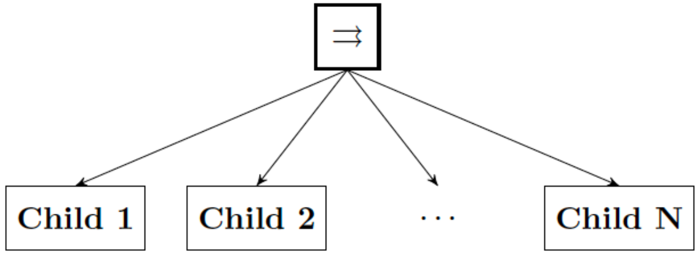
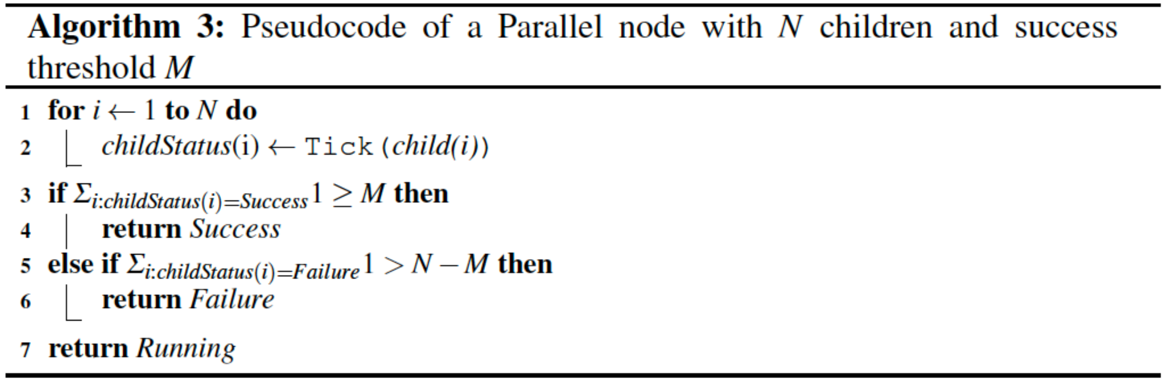


Fig. 1.4: Graphical representation of a Parallel node with N children.



Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

```
for $i \leftarrow 1$ to $N$ do
    $childStatus(i) \leftarrow$ Tick $(child(i))$
if $\Sigma_{i:childStatus(i)=Success} 1 \geq M$ then
    return $Success$
else if $\Sigma_{i:childStatus(i)=Failure} 1 > N - M$ then
    return $Failure$
return $Running$
```

[Colledanchise Ogren ’20]

```
// Children signal failure with “exit Failure”
Parallel(child1, child2, …, childN):
int SuccessCnt = 0, FailureCnt = 0;
trap SuccessPar in
[
  trap Success in
    trap Failure in
      child1;
      SuccessCnt++;  // Increment SuccessCnt if child succeeds
      exit Success;
    end trap;
   FailureCnt++;   // Increment FailureCnt if child fails
  end trap
||
…
||
  loop
    if (SuccessCnt >= M)
      exit SuccessPar;
    if (FailureCnt > N-M)
      exit Failure;
    pause;
  end loop
]
end trap
```

“Parallel” in Esterel

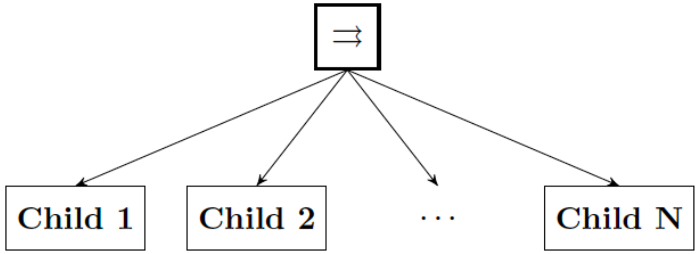
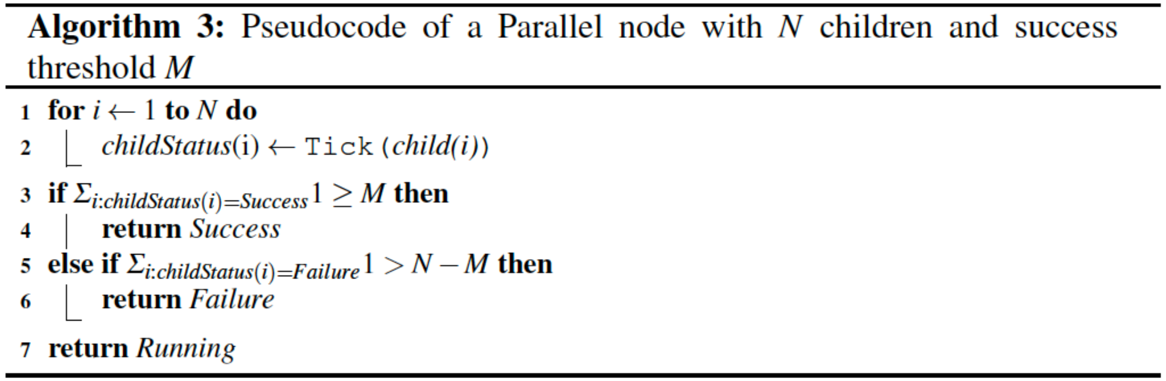


Fig. 1.4: Graphical representation of a Parallel node with N children.



Algorithm 3: Pseudocode of a Parallel node with N children and success threshold M

```
for $i \leftarrow 1$ to $N$ do
    $childStatus(\mathrm{i}) \leftarrow$ Tick ($child(i)$)
if $\Sigma_{i:childStatus(i)=Success} 1 \geq M$ then
    return $Success$
else if $\Sigma_{i:childStatus(i)=Failure} 1 > N - M$ then
    return $Failure$
return $Running$
```

[Colledanchise Ogren ’20]

// SPECIAL CASE M = N
// Children signal failure with “exit Failure”
// If any child fails, the parallel fails
// If all children succeed, the parallel succeeds
Parallel(child1, child2, …, childN):

```
[
 child1;
||
…
||
 childN;
]
```

“Fallback” in Esterel

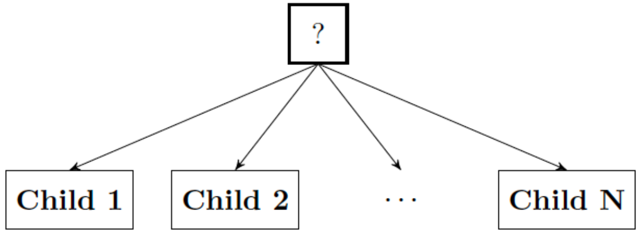
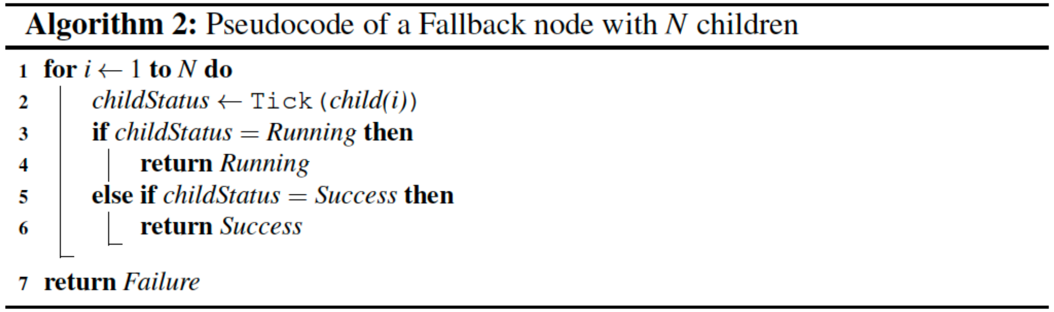


Fig. 1.3: Graphical representation of a Fallback node with N children.



Algorithm 2: Pseudocode of a Fallback node with N children

```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Success$ then
        return $Success$
return $Failure$
```

[Colledanchise Ogren ’20]

```
// Children signal failure with “exit Failure”
```

Fallback(child1, child2, …, childN):

```
trap Success in
  trap Failure in
    child1;
    exit Success; // Success when child1 terminates normally
  end trap;
 trap Failure in
    child2;
    exit Success; // Success when child2 terminates normally
  end trap
  …
  childN;  // If childN fails, propagate that out
  // Otherwise, terminate normally (= Success)
end trap
```

“Sequence” in Esterel

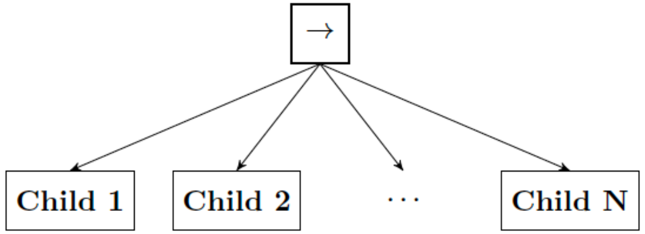
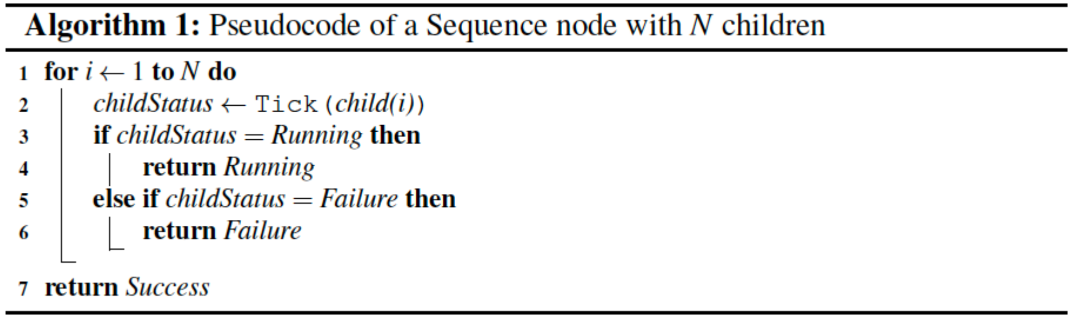


Fig. 1.2: Graphical representation of a Sequence node with N children.

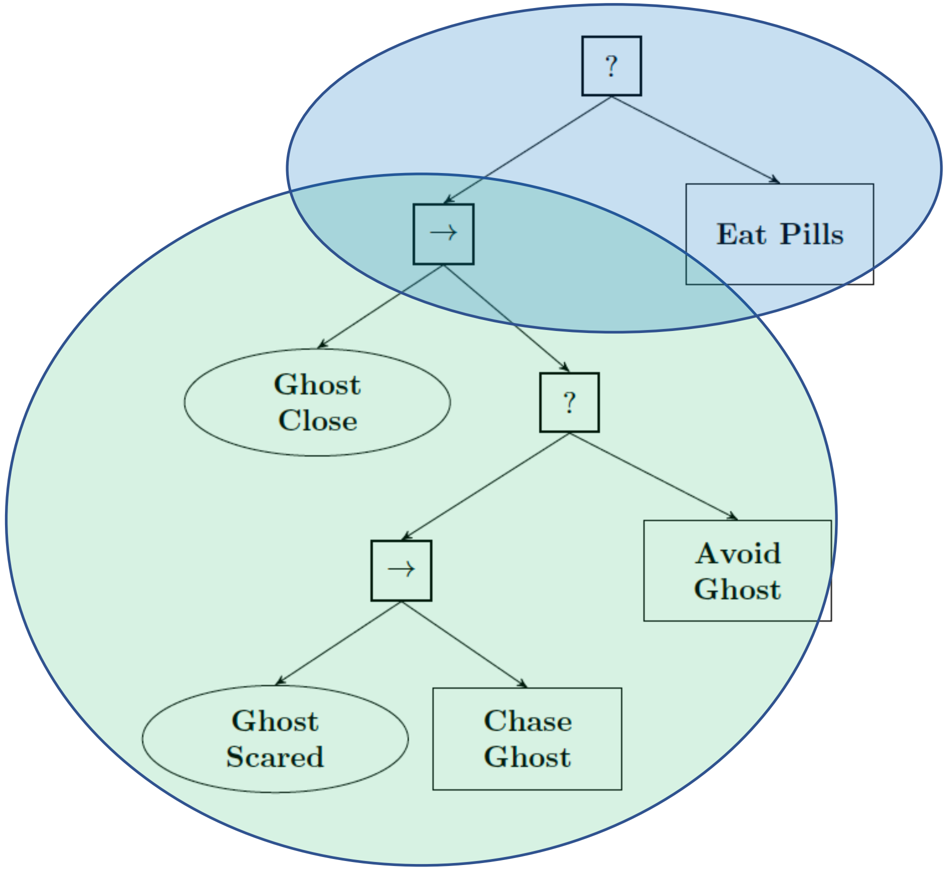


Algorithm 1: Pseudocode of a Sequence node with N children

```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Failure$ then
        return $Failure$
return $Success$
```

[Colledanchise Ogren ’20]

// Children signal failure with “exit Failure”
// If any child fails, this is propagated out
// Otherwise, terminate normally (success)
Sequence(child1, child2, …, childN):
child1;
child2;
…
childN;



[Colledanchise Ogren ’20]

Fallback(child1, child2, …, childN):

```
trap Success in
 trap Failure in
  child1;
  exit Success;
 end trap;  // Failure
 trap Failure in
  child2;
  exit Success;
 end trap // Failure
 …
 childN;
end trap
```

Sequence(child1, child2, …, childN):

```
child1;
child2;
…
childN;
```

Pac-Man()

```
trap Success in
 trap Failure in
  if (GhostClose) exit Failure;
  trap Success in
   trap Failure in
    if (GhostScared) exit Failure;
    ChaseGhost()
    exit Success;
   end trap;
   AvoidGhost()
  end;
  exit Success;
 end trap;
 EatPills()
end trap
```

But Remember:

At every tick,
start at **root** of BT

“Sequence” in Esterel – Not

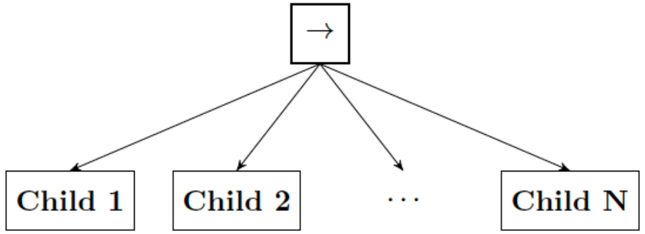
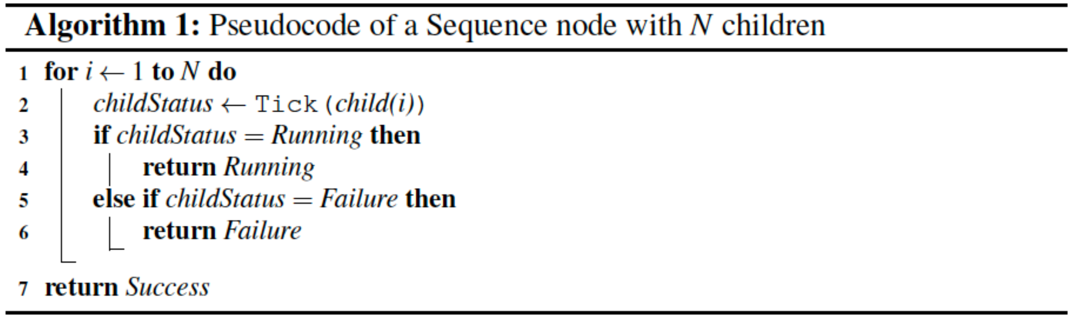


Fig. 1.2: Graphical representation of a Sequence node with N children.



Algorithm 1: Pseudocode of a Sequence node with N children

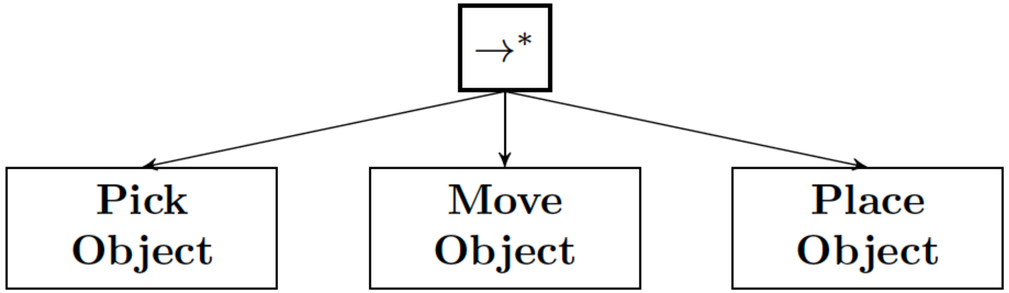
```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick ($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Failure$ then
        return $Failure$
return $Success$
```

```
// Children signal failure with “exit Failure”
// If any child fails, this is propagated out
// Otherwise, terminate normally (success)
```

Sequence(child1, child2, …, childN):

```
child1;
child2;
…
childN;
```

The problem: this translation implements an “un-reactive” *sequence with memory*, where we resume at running children, instead of re-starting each tick at first child again



[Colledanchise Ogren ’20]

“Sequence” in Esterel – With Weak Suspend?

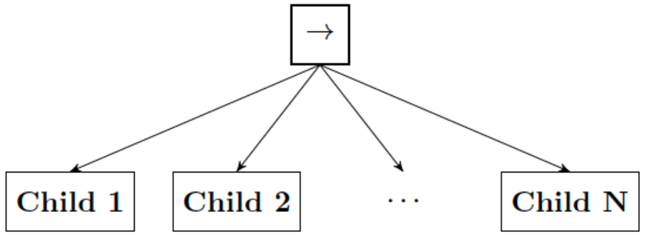
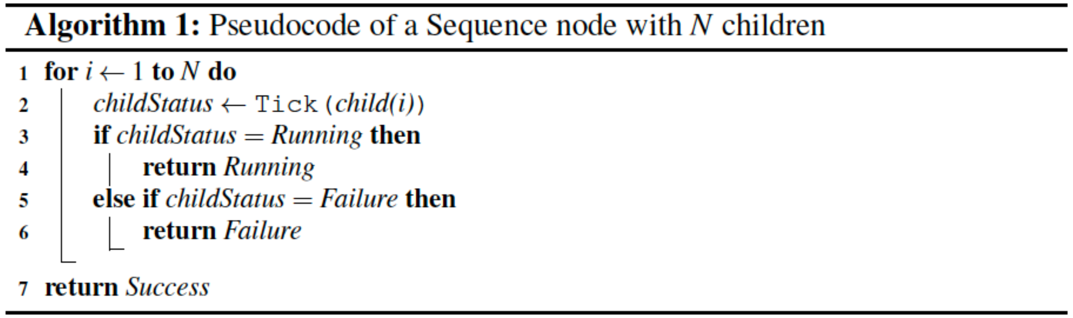


Fig. 1.2: Graphical representation of a Sequence node with N children.



Algorithm 1: Pseudocode of a Sequence node with N children

```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Failure$ then
        return $Failure$
return $Success$
```

[Colledanchise Ogren ’20]

```
// Children signal failure with “exit Failure”
// If any child fails, this is propagated out
// Otherwise, terminate normally (success)
Sequence(child1, child2, …, childN):
weak suspend
  child1;
  child2;
  …
  childN;
when true
```

Note: should also consider nestings of reactive (no memory) and non-reactive (with memory) constructs.

“Fallback” in Esterel – With Weak Suspend

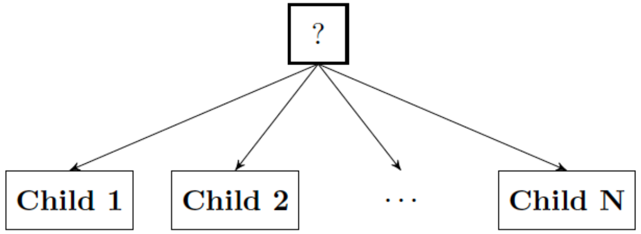
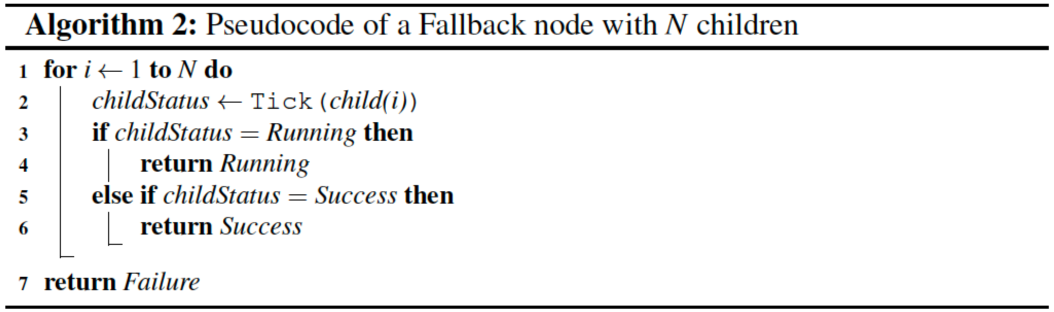


Fig. 1.3: Graphical representation of a Fallback node with *N* children.



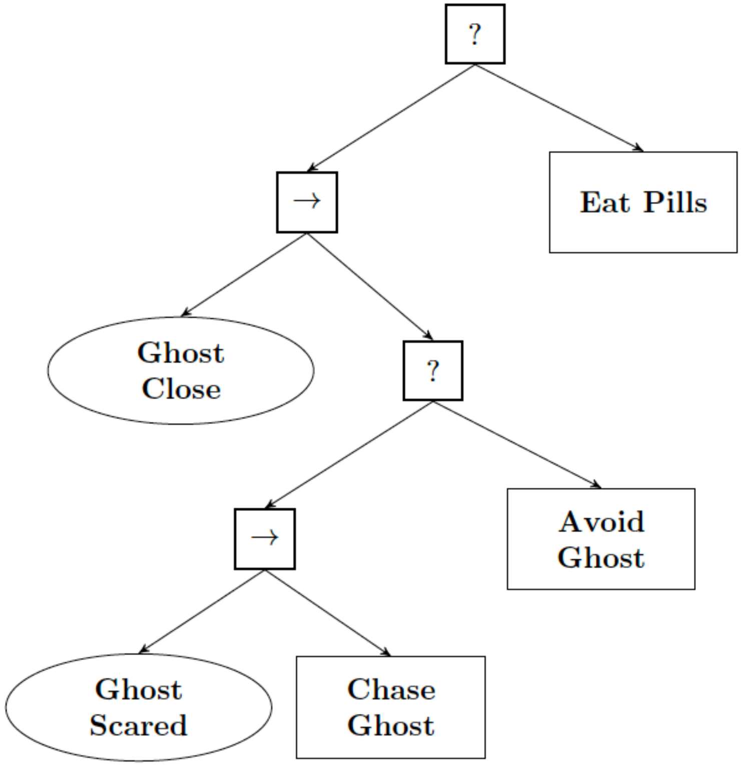
Algorithm 2: Pseudocode of a Fallback node with *N* children

```
for $i \leftarrow 1$ to $N$ do
    $childStatus \leftarrow$ Tick($child(i)$)
    if $childStatus = Running$ then
        return $Running$
    else if $childStatus = Success$ then
        return $Success$
return $Failure$
```

[Colledanchise Ogren ’20]

```
// Children signal failure with “exit Failure”
Fallback(child1, child2, …, childN):
weak suspend
  trap Success in
    trap Failure in
      child1;
      exit Success; // Success when child1 terminates normally
    end trap;
   trap Failure in
      child2;
      exit Success; // Success when child2 terminates normally
    end trap
    …
    childN;  // If childN fails, propagate that out
    // Otherwise, terminate normally (= Success)
  end trap
when true
```

Esterel Sketches for Select Mode Pattern



Sequence special case:

- Only 2 children
- child1 is a condition, i.e., "instantaneous", returns either Failure or Success

```
// Option 1: child1 encodes
// Success/Failure as true/false:
Sequence(child1, child2):
abort
  child2
when immediate !child1

// Option 2: child1 encodes
// Success/Failure with
// as termination/exit Failure:
Sequence(child1, child2):
loop
  trap Success in
    trap Failure in
      child1;
      exit Success;
    end trap; // Failure
    emit abortChild2;
    exit Failure
  end trap; // Success
  pause;
end
||
abort
  child2
when immediate abortChild2
```

Fallback special case:

- Only 2 children
- child1 returns either Failure or Running

```
// child1 encodes Failure/Running
// as termination/exit Failure:
Fallback(child1, child2):
trap Success in
[
loop
  trap Failure in
    child1;
    // Usually don't get here
    exit Success;
   end trap; // Failure
   emit abortChild2;
  end trap; // Success
  pause;
end
||
abort
  child2
when immediate abortChild2
]
end trap
```

Wrap-Up – BTs in Esterel

- As in Esterel, individual BT nodes do maintain (internal) state
- However, “reactive” BT does **not** maintain state;
  e.g., sequence always starts at first child
- BT return values resemble Esterel completion codes
- However, emulating “reactiveness” with Esterel (v5) appears non-trivial
- Possible approaches (?):
  - weak suspension, to avoid changing state
  - Explicit control structure, based on existing primitives such as (weak) aborts, loops, (weak) suspend, gotopause, …
  - New primitive(s) designed explicitly for reactiveness
  - Dataflow approach, as in “FSM pattern” also used in Lingua Franca

Thanks!