BEHAVIOR TREES

ARTIFICIAL INTELLIGENCE | COMP 131

TODAY ON AI

- Reactive planning and Behavior Trees
- Behavior Tree elements
- Behavior Tree features
- Design patterns
- Readings
- Questions?

Round 1
1v1

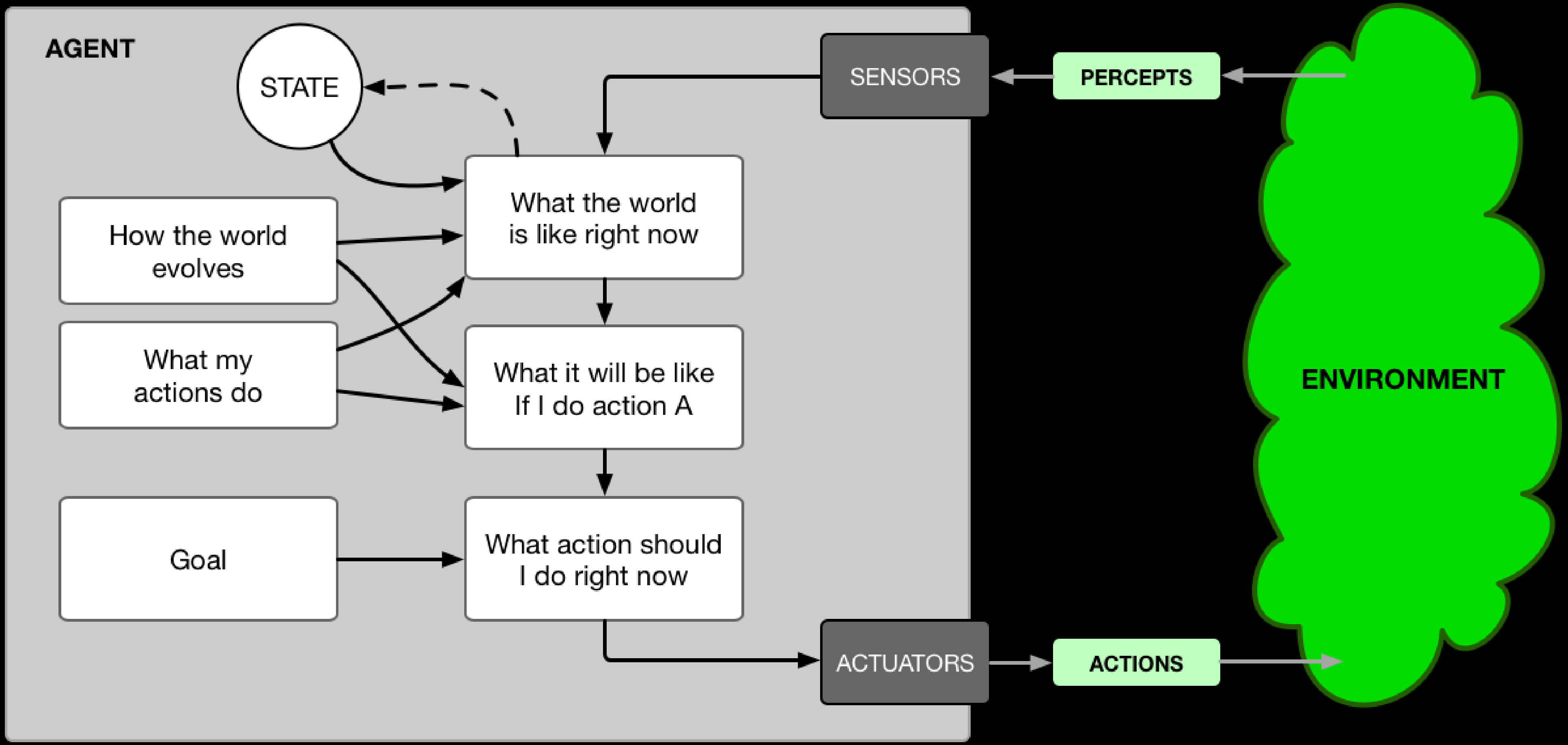
SECTION 01

Reactive planning and Behavior Trees

Planning defines a branch of Artificial Intelligence devoted to find a sequence of actions that will lead to a goal.

Reactive planning denotes a class of algorithms for **action selection** by autonomous agents that differs from **classical planning**:

- They are **time-bound** so they can quickly deal dynamic and unpredictable environments
- They compute **just one (or few more) next action** in every instant, based on the current context



UNPLANNING AGENT: REFLEX AGENTS, MODEL-BASED REFLEX AGENTS, LEARNING AGENTS
PLANNING AGENT: MODEL-BASED, GOAL-BASED AGENTS

Wikipedia defines a Behavior Tree as: **Behavior trees are a formal, graphical modeling language used in Systems and Software Engineering**. Behavior trees employ a well-defined notation to unambiguously represent natural language requirements for large-scale software-integrated systems.

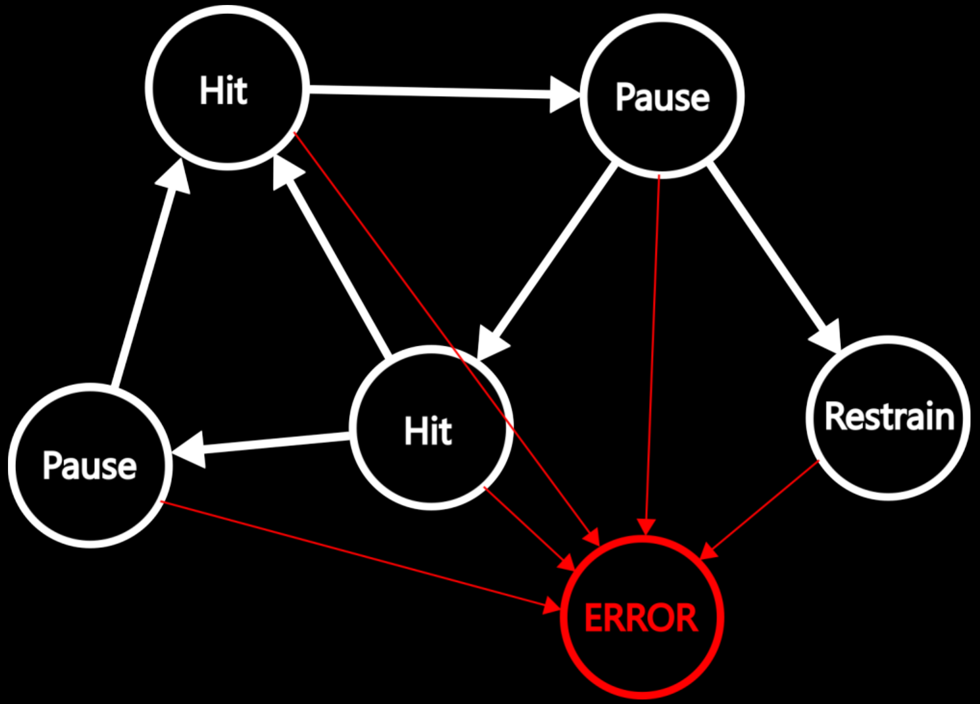
- Developed by **R. G. Dromey** with some key concepts published in **2001**
- Used to describe large-scale systems, embedded systems, role-based access control, biological systems, etc.

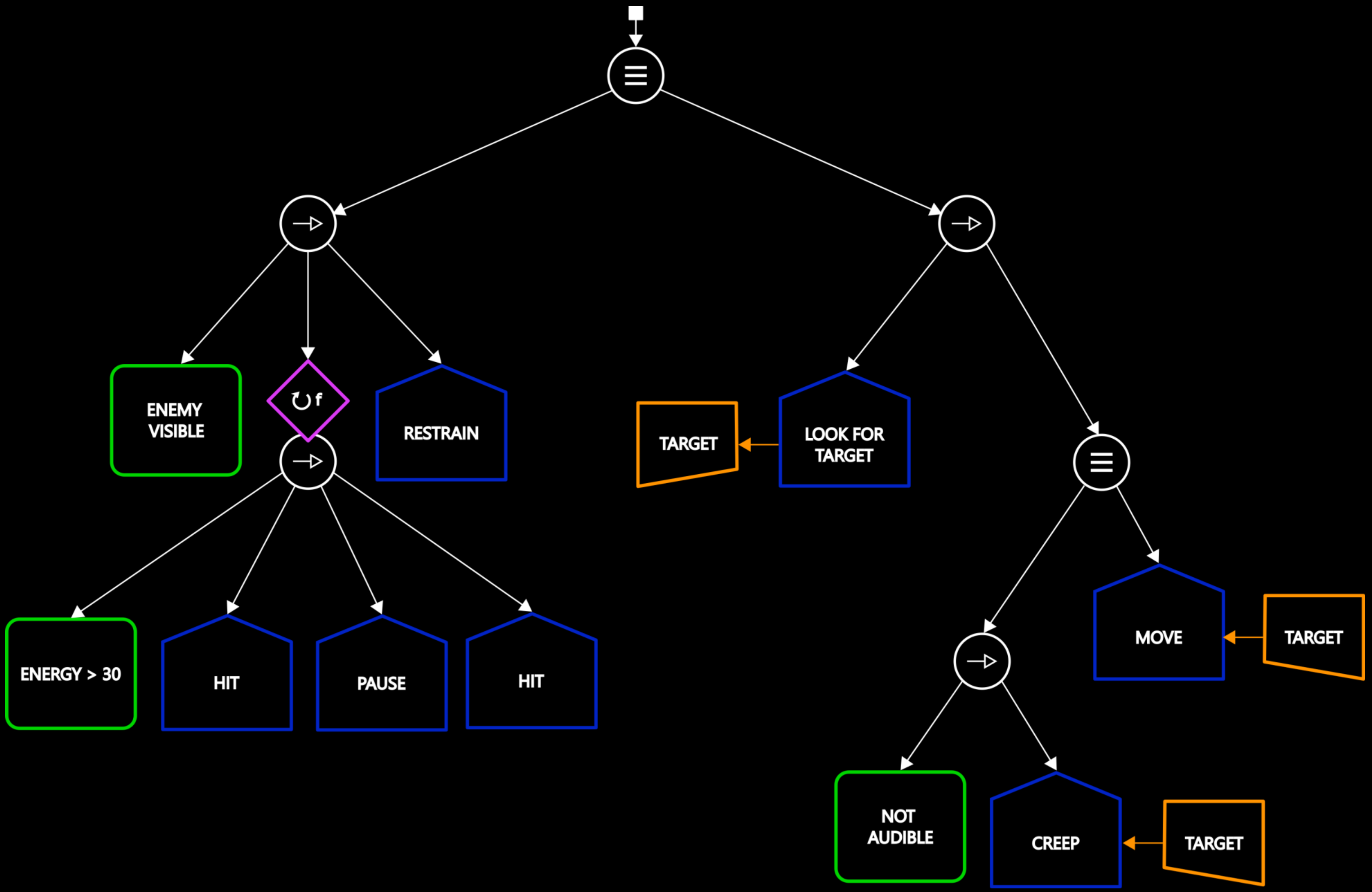
In 2004 and 2005, **Halo 2** and **Façade** AI designers adopted a similar graphical representation and name for a different formalism.

They synthesized several techniques and algorithms in one manageable tool:

- Finite State Machines
- Hierarchical Finite State Machines
- Scheduling
- Search
- Resource Conflict Resolution

- While finite state machines are **reasonably intuitive** for simple cases, as they become more **complex**, they are hard to keep goal-oriented
- As the number of states increases, the transitions between states become exponentially complex
- Hierarchical state machines can help, but many of the same issues remain





Example of Behavior Tree

SECTION 02

Behavior Tree nodes

Behavior trees organize their **nodes** into a **tree** or, more generally, **directed acyclic graph (DAG)**:

NODE	REPRESENTATION	RESULT
A **task** alters the state of the system		SUCCEEDED, FAILED, or RUNNING
A **condition** tests some property of the system		SUCCEEDED, or FAILED
A **composite** aggregates tasks and conditions		SUCCEEDED, FAILED, or RUNNING
A **decorator** alters the basic behavior of the tree-node it is associated with		SUCCEEDED, FAILED, or RUNNING
Sub-trees as a reference to complex behaviors		SUCCEEDED, FAILED, or RUNNING
Operations on the blackboard (read or write)		n/a

A single evaluation of a behavior tree runs as follow:

- **Order of evaluation**: BTs are evaluated from left to right in a depth-first tree traversal fashion
- **Result of the evaluation**: A node is evaluated (according to its nature) and an evaluation result (**SUCCEEDED**, **FAILED**, or **RUNNING**) is calculated
- **Application of a result**: The result of a node evaluation is provided to the parent of the node, which in return recursively uses it according to its nature
- **Premature conclusion**: If the result is **RUNNING**, the node evaluation concludes immediately, and it is sent to the node parent

The simplest nodes are:

- **Task**: A task is complex function executed on the system. There is usually a task implementation per type of action. The result of the execution can be either **SUCCEEDED** or **FAILED**. If the completion of the task cannot be achieved within the allotted time, the result will be **RUNNING**.

- **Condition**: A condition is a check on the state of the system. There is usually a condition implementation per type of condition. The result of the execution can be either **SUCCEEDED** or **FAILED**.

The most common **composites** are:

- **Sequence**: Children are evaluated in order (left to right). It fails as soon as one of the children fails, otherwise it succeeds
- **Selection**: Children are evaluated in order (left to right). It fails if all children have failed, otherwise it succeeds
- **Priority**: Like selection, but the children are evaluated in order of priority
- **Random sequence**: Like sequence, but the children are evaluated in random order
- **Random selection**: Like selection but the children are evaluated in random order

The most common decorators are:

- **Logical negation**: It executes the attached node and then it negates its result

- **Until Succeeds**: It executes the attached node while it fails returning **RUNNING**. It returns **SUCCEEDED** at the first success.

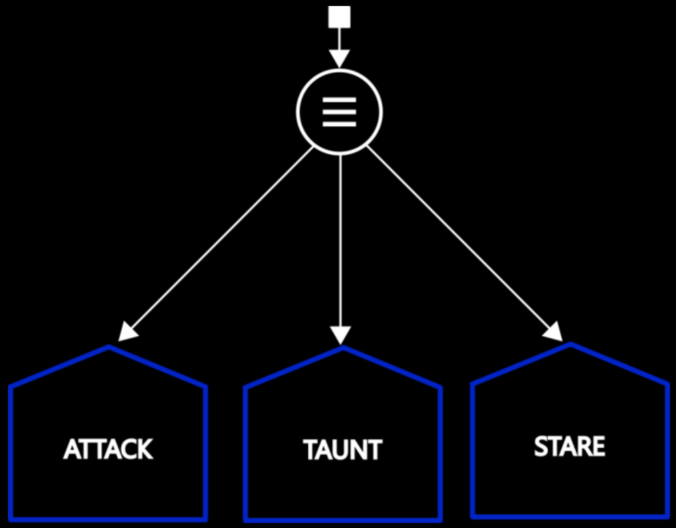
- **Until Fails**: It executes the attached node while it succeeds returning **RUNNING**. It returns **SUCCEEDED** at the first failure.

- **Resource semaphore**: It resolves conflicts between nodes associated with the same resource. It returns **RUNNING** while the resource is not available.

- **Timer**: It executes the attached node for a specific amount of time. It returns **RUNNING** while the timer is running. It returns **SUCCEEDED** after the expiration.

There is **never a representation of the system state** in a behavior tree. Also, they normally require an inter-node communication mechanism called **blackboards**.

- The **blackboard** implementation is one big design choice:
  - **Single**: There is only one blackboard for the whole tree. It includes state of the system and the tree nodes
  - **Global/Local**: The blackboard is divided in two areas: state of the system, and state of the tree nodes
- The simplest implementation of a blackboard is a **hash-table** or **dictionary**:
  - The **key** is the variable name
  - The **value** is the variable value



Try all the children **until one** succeeds. It fails otherwise.

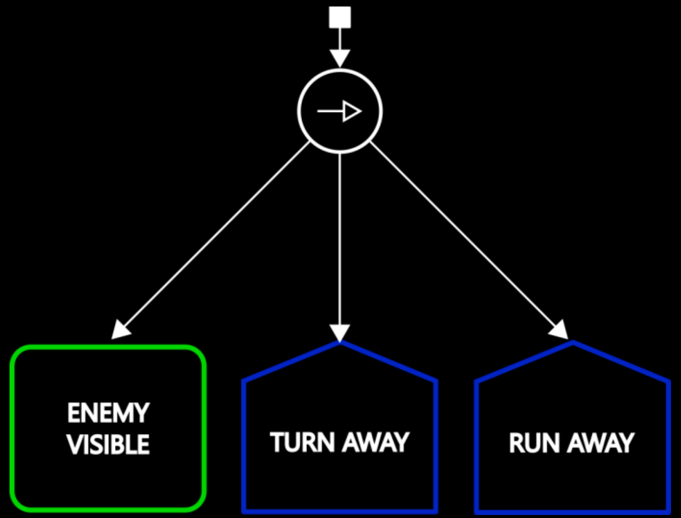
```
class Node:
  function run()
    return FAILED
```

```
class Attack(Node):
  function run()
    if Execute attack:
      return SUCCEEDED
    else
      return FAILED
```

...

Implementation of all types of actions allowed in the system.

```
class Selector(Node):
  function run()
    for child in children
      result = child.run()
      if result SUCCEEDED:
        return result

    return FAILED
```



Execute all the children sequentially, **succeeding if all succeed**. It fails otherwise.

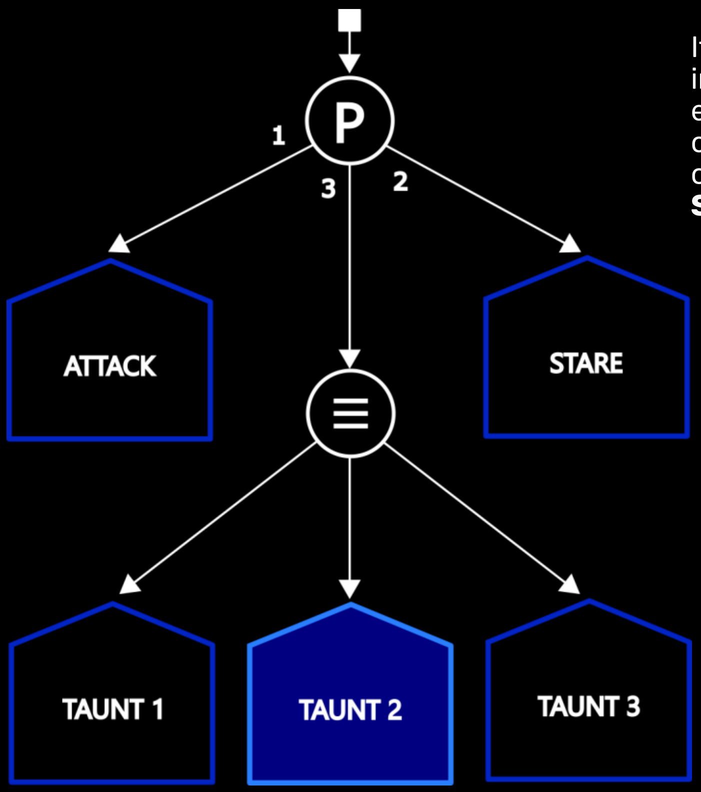
```
class Node:
  function run()
    return FAILED
```

```
class Enemy_Visible(Node):
  function run()
    if Enemy visible:
      return SUCCEEDED
    else
      return FAILED
```

...

Implementation of all types of actions allowed in the system.

```
class Sequence(Node):
  function run()
    for child in children
      result = child.run()
      if result FAILED:
        return result

    return SUCCEEDED
```

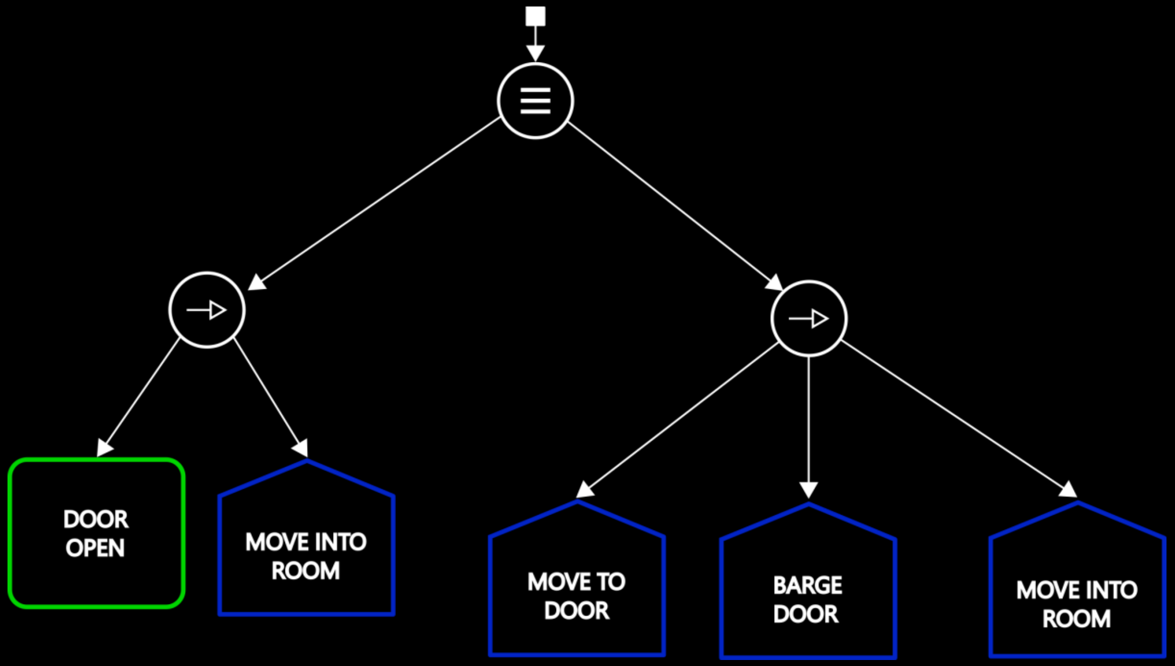


If **TAUNT 2** is marked as in a running state, its execution will continue only after the execution of both **ATTACK** and **STARE**.

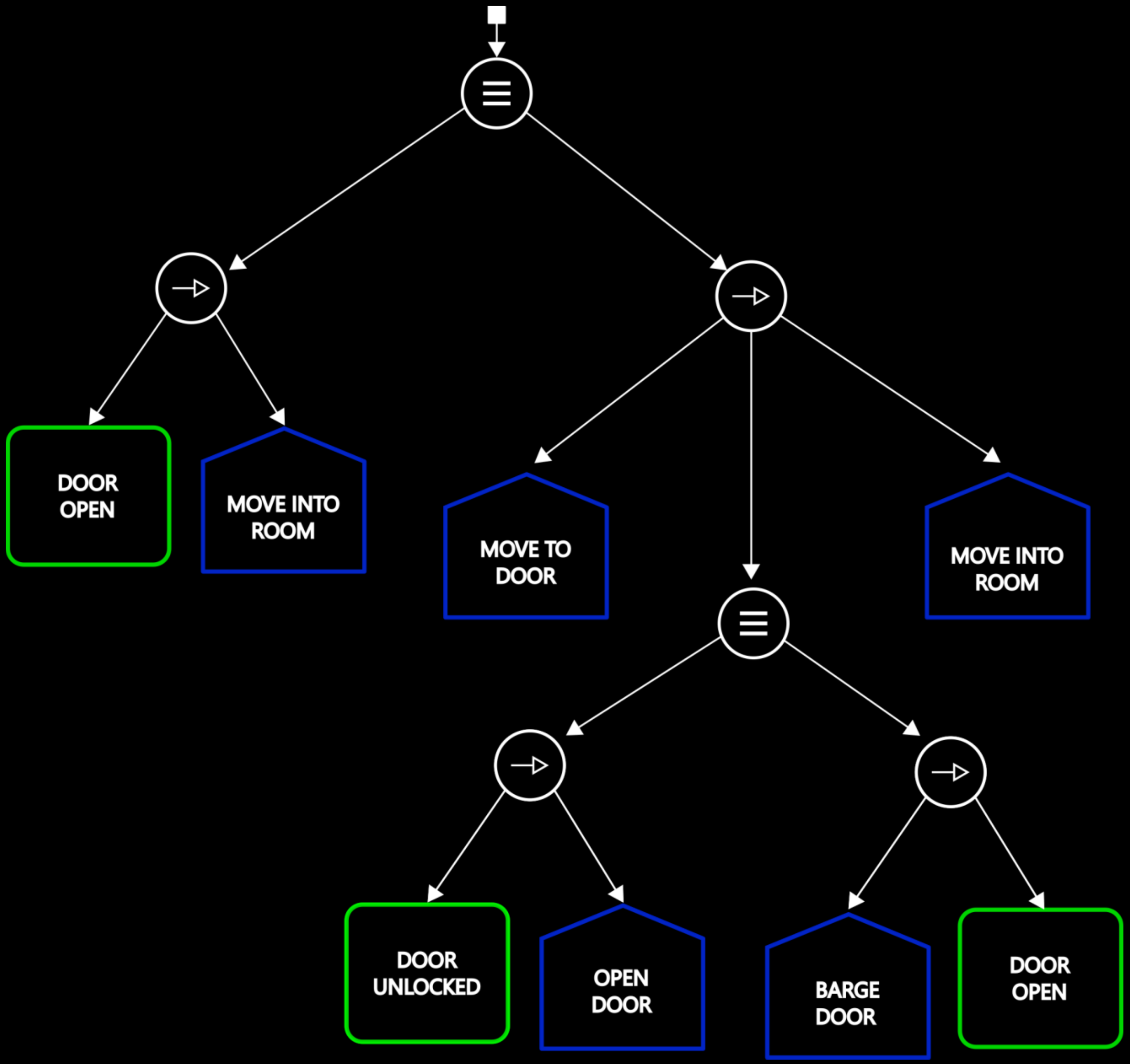
Try all the children **until one** succeeds in order of priority.

A priority node **forces** the tree to evaluate higher-priority nodes before continuing the evaluation of the one marked as **running**.

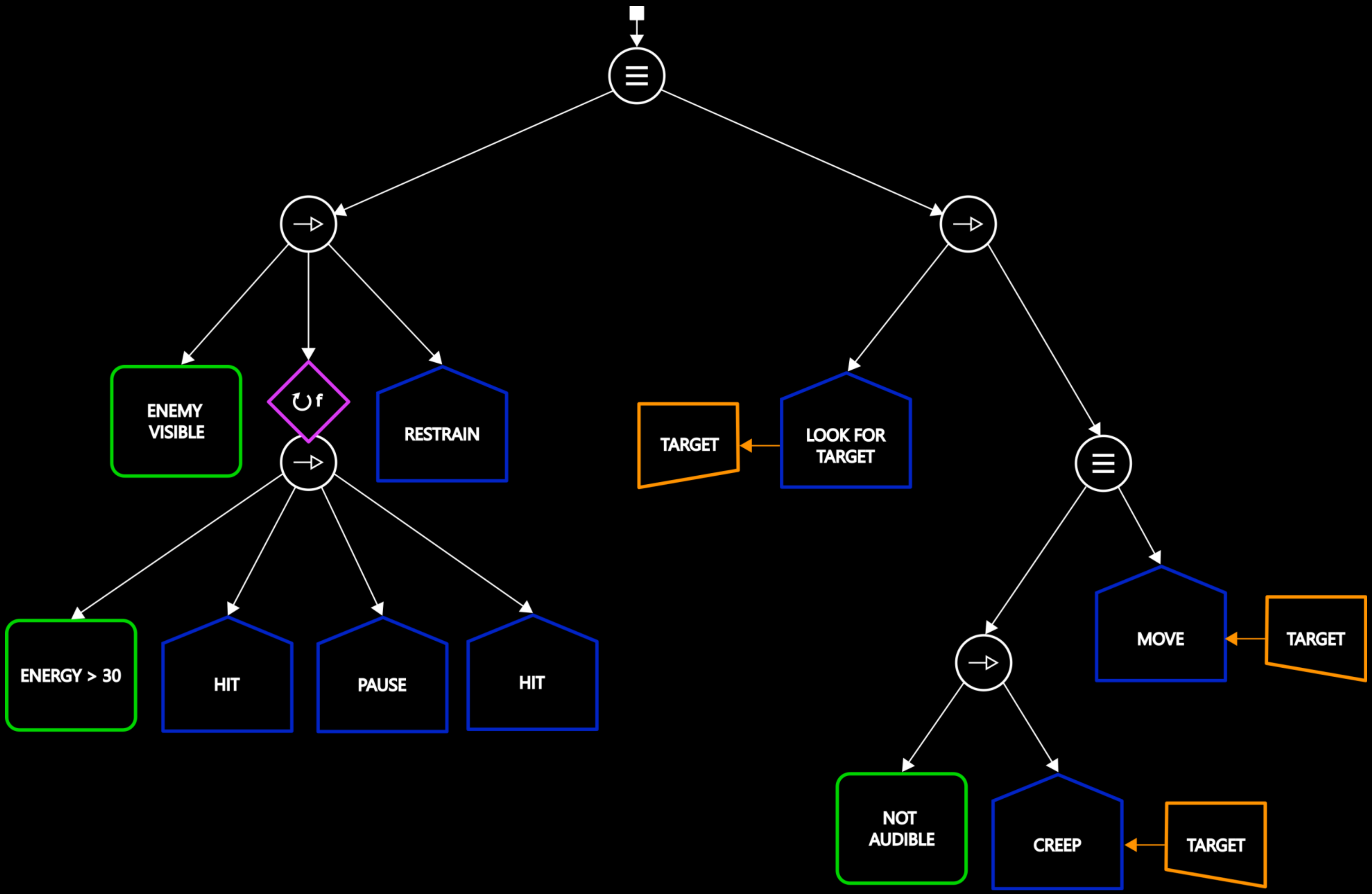
```
class Priority(Node):
  function run
  for child in sort(children)
    result = child.run()
    if result SUCCEEDED:
      return result
    if result RUNNING:
      return result

  return FAILED
```


DOOR OPEN
MOVE INTO ROOM
MOVE TO DOOR
BARGE DOOR
MOVE INTO ROOM

Example 1


DOOR
OPEN
MOVE INTO
ROOM
MOVE TO
DOOR
MOVE INTO
ROOM
DOOR
UNLOCKED
OPEN
DOOR
BARGE
DOOR
DOOR
OPEN

Example 2


ENEMY VISIBLE
↻ f
RESTRAIN
ENERGY > 30
HIT
PAUSE
HIT
TARGET
LOOK FOR TARGET
MOVE
TARGET
NOT AUDIBLE
CREEP
TARGET

23

Example 3

SECTION 03

Behavior Tree features

Behavior trees have several highly desirable **features**:

- The basic components are reusable
- Agents can be goal directed and autonomous
- Agents can respond to events
- The knowledge base is easy to read and debug
- The knowledge base is easy to maintain

Behavior trees also have several **improvements** over hierarchical finite state machines:

- The history of the state transitions is clear
- Easy to build sequences
- Easy to add new behaviors without rewiring

A valid criticism about behavior trees is that they do not implement a full search in the search space, but a **reactive search** or **planning**.

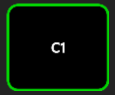
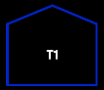
It is hard to **produce** a minimal set of tasks, conditions, and decorators

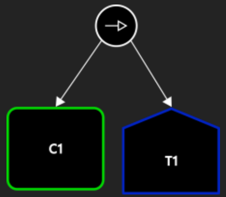
Less important criticism:

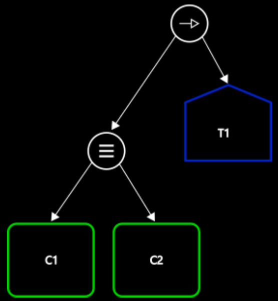
- Slow to react to changes in strategy
- It is hard do verify that the set of tasks, conditions, and decorators is powerful enough to describe the AI requirements

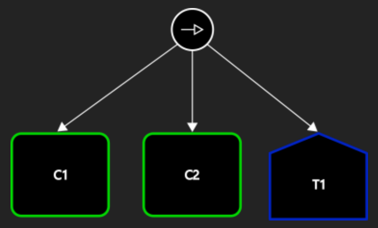
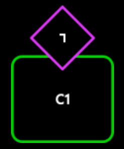
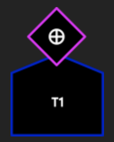
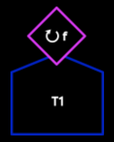
SECTION 04

Design patterns

TASKS	IF-THEN-ELSE	RULE NAME
	**return** C1	`Empty`
	**return** T1	`Always`

	**if** C1:
	    return T1
	else
	    return failed
		`Conditional execution`

	**if** C1 | C2:
	    return T1
	else
	    return failed
		`Conditional execution OR`

TASKS	IF-THEN-ELSE	RULE NAME
	**if** C1 & C2: **return** T1 **else** **return** failed	Conditional Execution AND
	**return not** C1	Negation
	**if not** Ti1.has_expired(): **if** T1 == succeeded **return** running **else** **return** failed **else** **return** succeeded	Timer
	**if** T1 == succeeded **return** running **else** **return** succeeded	Until fail

READINGS

Chapters 10 and 11

https://arxiv.org/abs/1709.00084

https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

QUESTIONS ?

ARTIFICIAL INTELLIGENCE

COMP 131

FABRIZIO SANTINI

VERSION 4.2