DTU

Software for Autonomous Systems
SFfAS-31391:

Robot Acting with SkiROS

Guest Lecturer
Francesco Rovida, PhD
f.rovida@riact.eu

Outline

Planning and acting: a bit of history

FSMs, Behavior Trees

SkiROS

Exercise

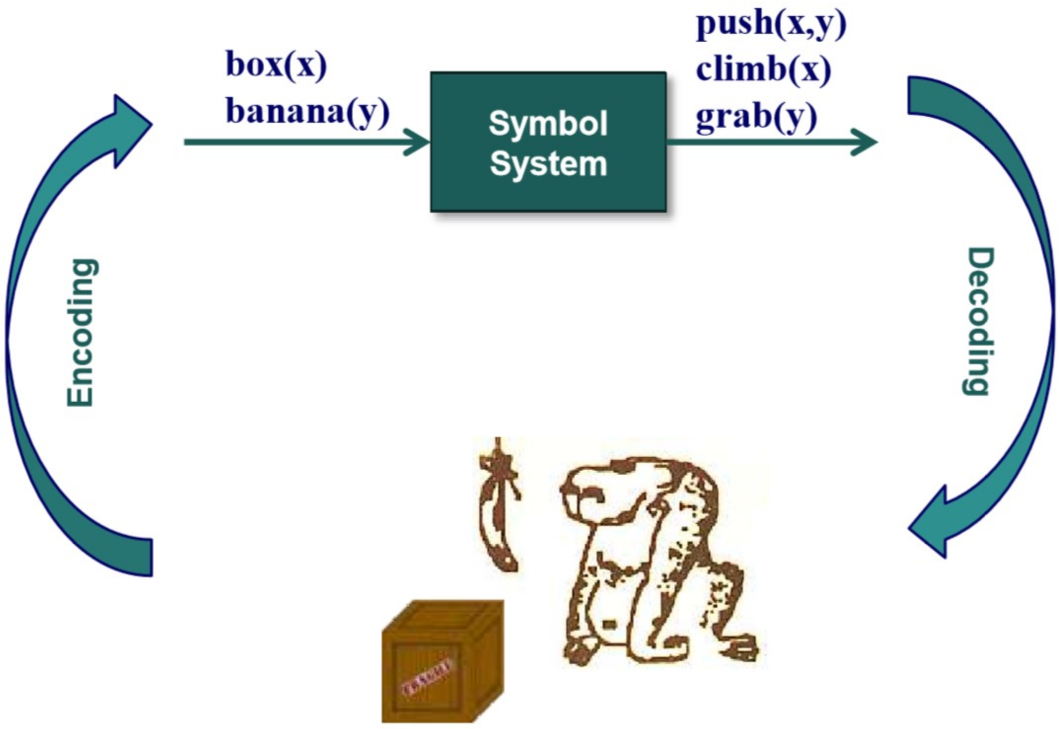
1949: Enter the computer

“ Perform mechanical operations on symbols ”

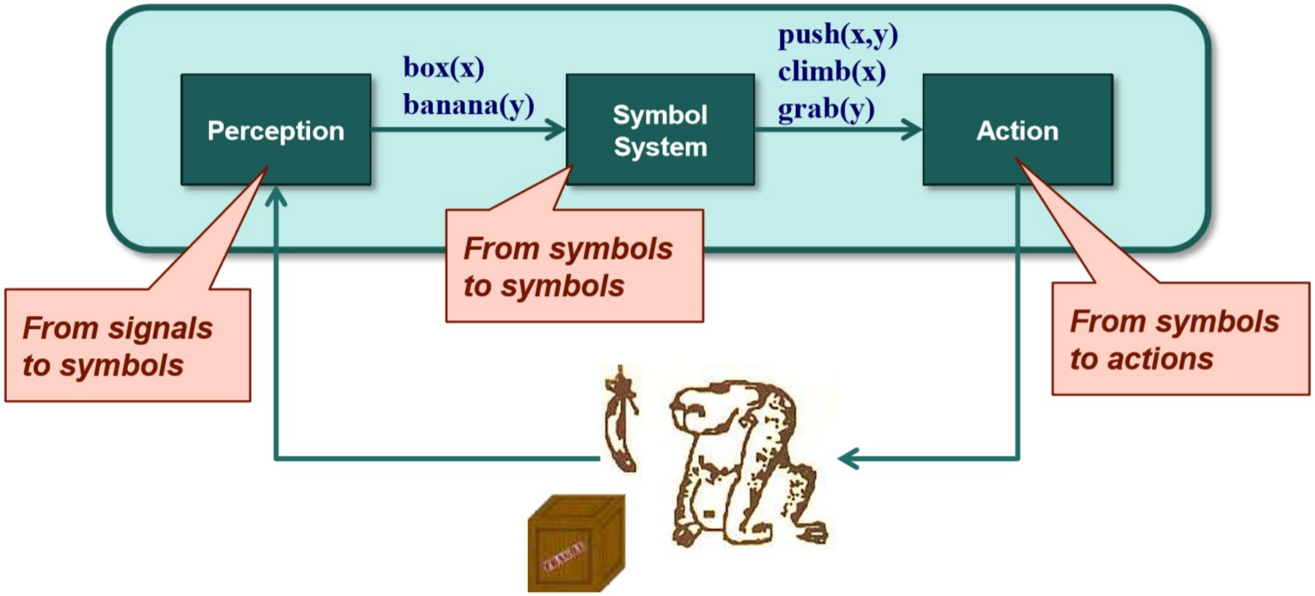
$3x - 6 = 0$ → **Computer** → $x = 2$

1956: Artificial Intelligence

“Did you say symbols?”

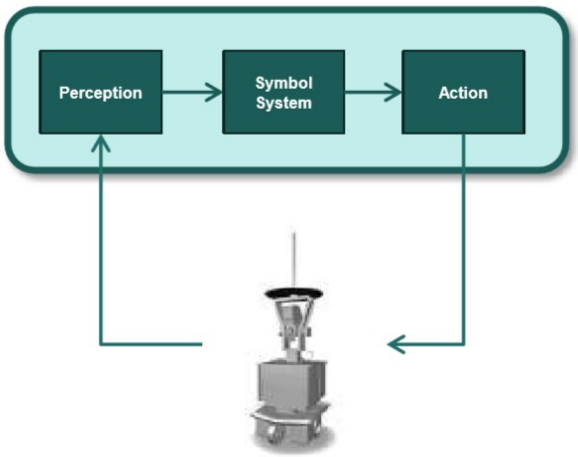


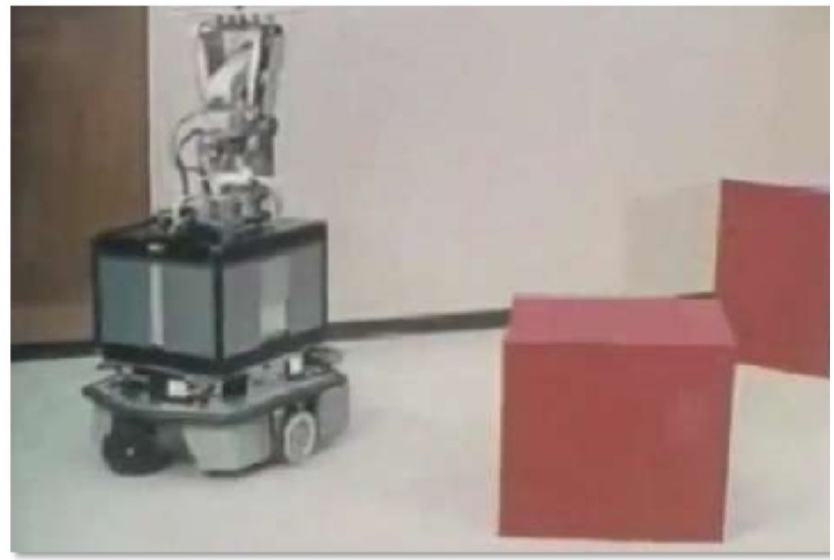
1956: Artificial Intelligence



- What robot senses is translated into symbols
- Tasks are subdivided into atomic actions
- Task planning techniques are used to find a sequence of action, given a desired goal state

1968: Shakey, the first robot using AI programs



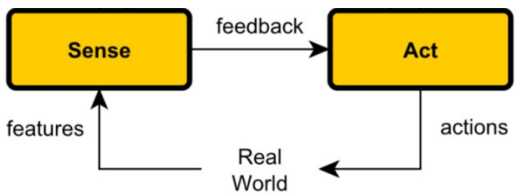


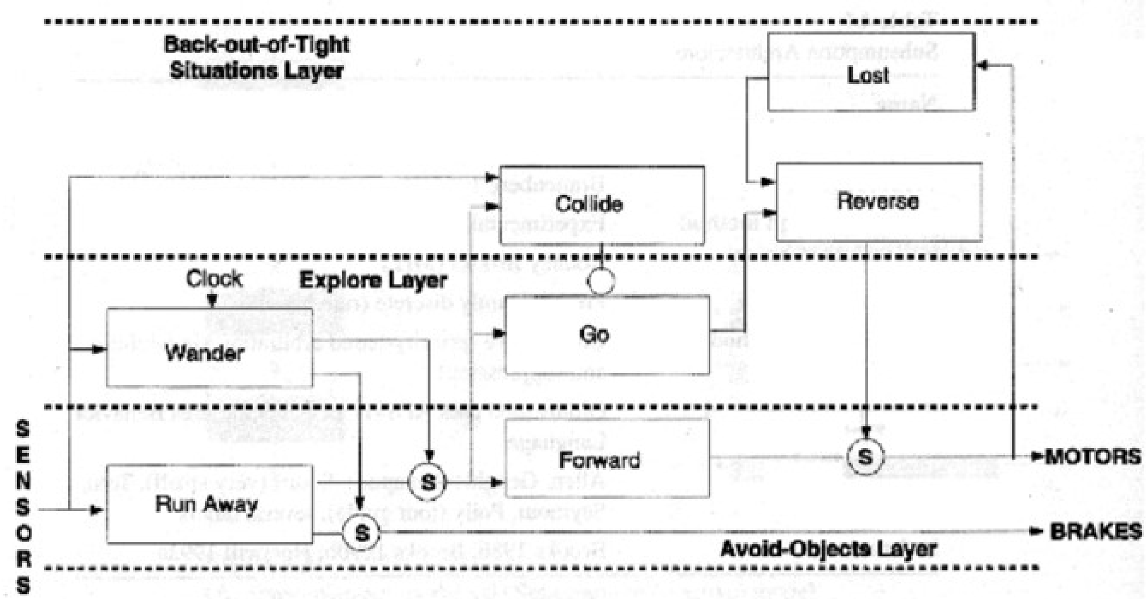
[Source: SRI International]

Limitations

- Faked world (perfect shapes, uniform light, etc.)
- Slow reactions
- Difficult to model control loops

1986: Reactive robots





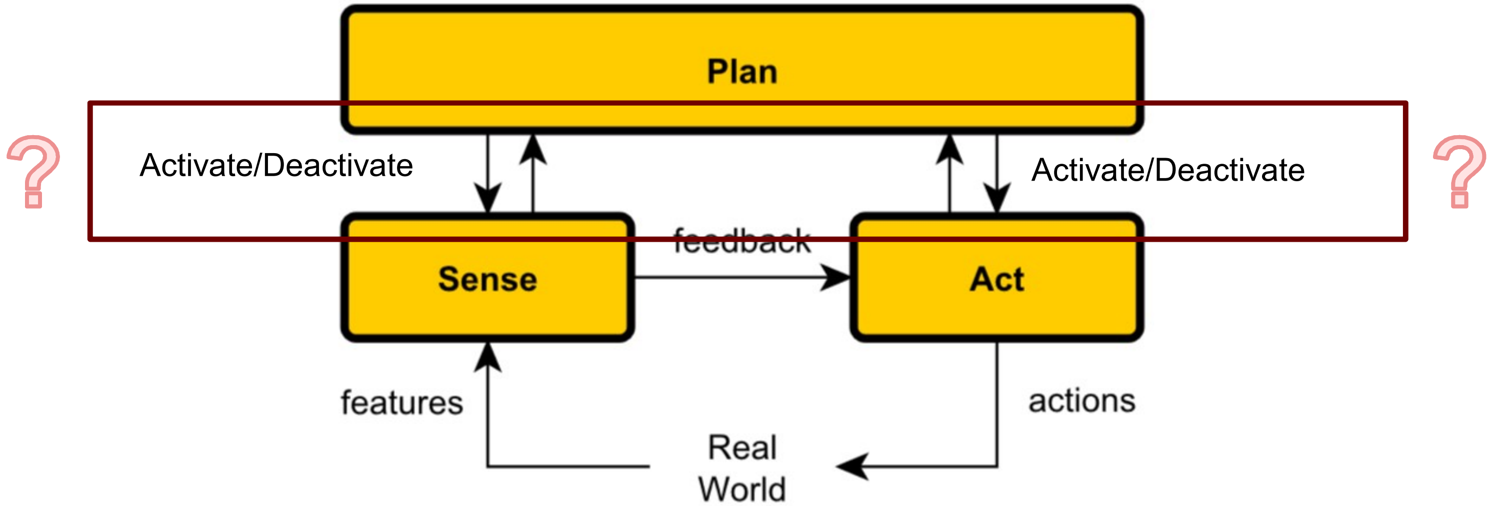
Characteristics

- Composition of concurrently running behaviors
- *Inhibition mechanism*: behaviors from the higher layers can inhibit the output of the ones from the lowest layers

Limitation

- Insect-like behavior, difficult to pursue goals

1989-Now: Layered architectures



Characteristics
Takes the best of 2 approaches: deliberative high layer, reactive low layer

Active research topic
Model to integrate deliberative and reactive layer coherently (they can't just be patched together)

Planning vs. acting

Task	Specification of objectives to be achieved by the robot, in possibly different forms, e.g., as goal states.
Planning	Given one task, generate a sequence of actions to achieve it
Acting	Refine planned actions into commands appropriate for the current context and reacts to events; both refinement and reaction may rely on skills, i.e., a collection of closed-loop functions.

Planning and acting are closely interrelated and can't always be separated

[F. Ingrand et Al. 2015 - Deliberation for Autonomous Robots: A Survey]

(Some) Popular approaches

Deterministic task planning

- Classical (PDDL)
- Partial-order
- Hierarchical Task Networks

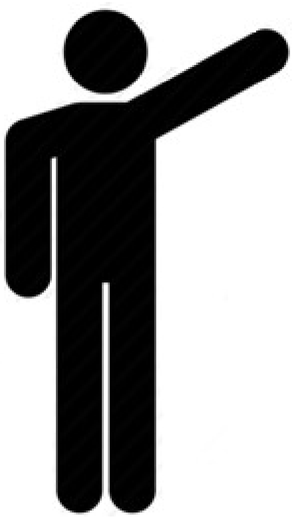
Acting

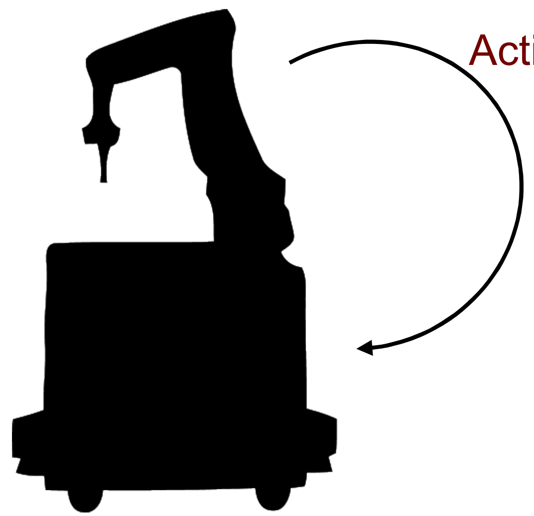
- (Hierarchical) Finite state machines
- Behaviour trees
- Petri nets
- Domain Specific Languages (TDL, CRAM)
- Other approaches (Probabilistic, Logic, etc.)

STRIPS planning

- First formalism to plan sequences of actions, nowadays still vastly used
- Planning Domain Definition Language (PDDL) is the format to define a STRIPS planning problem

Goal:
InKit(Alternator, Kit-7)

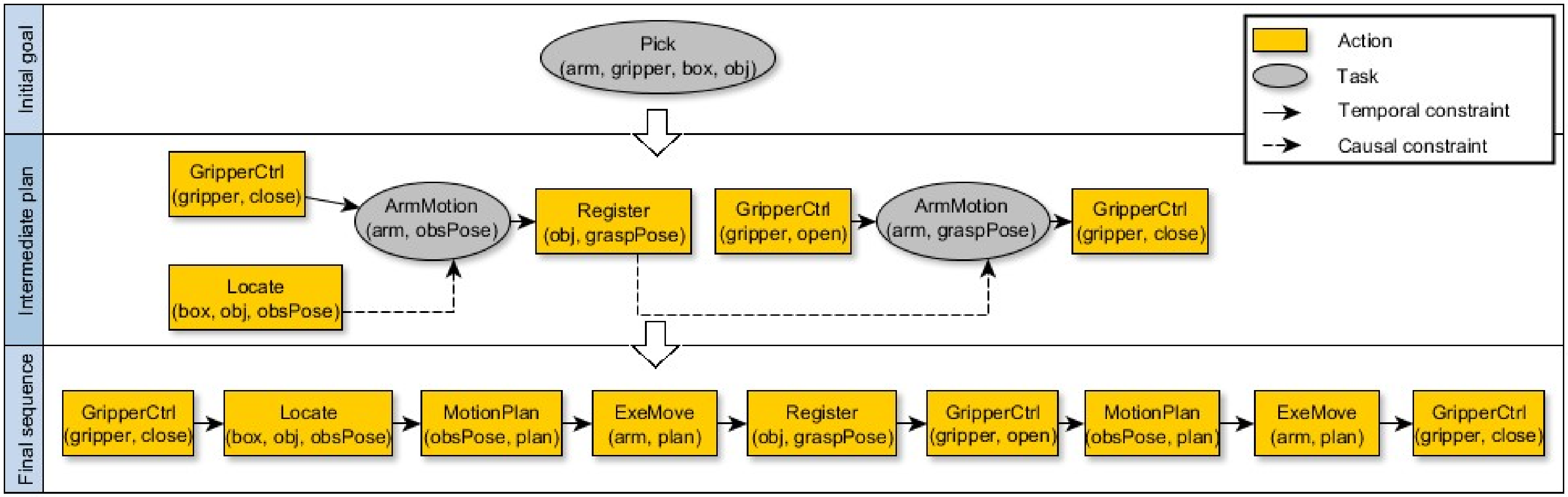




Action sequence:
drive(Room-25)
pick(Alternator)
place(Kit-7)

Hierachical Task Networks (HTNs)

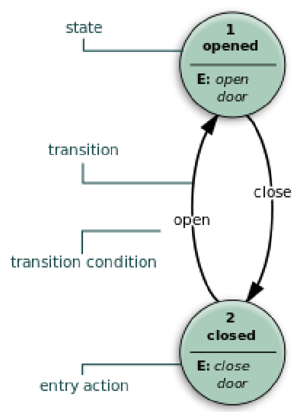
[Gallab et Al. 2004 – Chap.11]

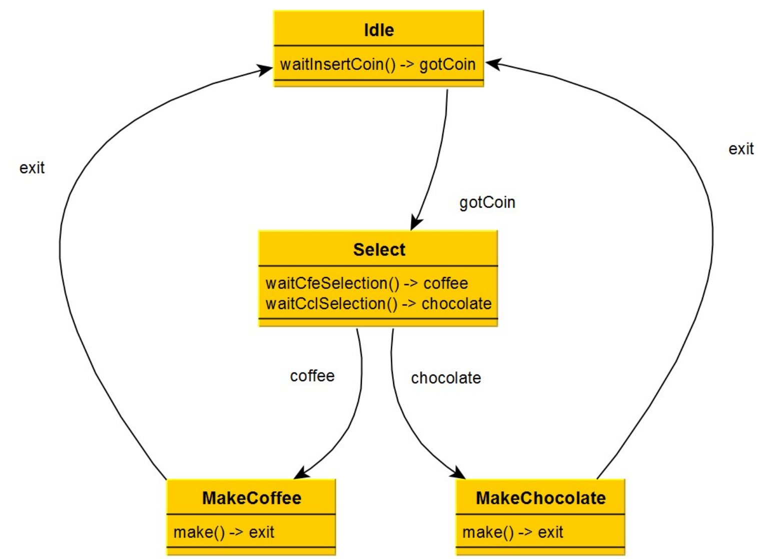


- Hierarchical task network (**HTN**) planning, uses **abstract actions** to **incrementally** decompose a planning problem from a **high-level goal** statement to a **primitive plan network**
- **Primitive operators** represent actions that are **executable**, and can appear in the final plan
- **Abstract actions** are **tasks** that require further decomposition to be executed
- Recent evolution, used in many compute games

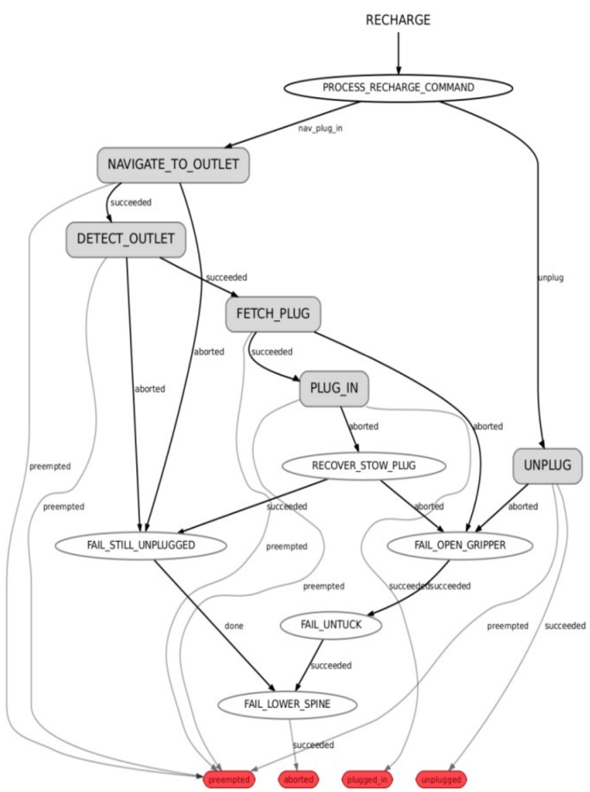
Finite State Machines (FSMs)

- Graph with a finite number of nodes (**states**) and edges(**transitions**)
- Historical and nowadays most widely used approach
- A transition is a set of actions to be executed when an event is received

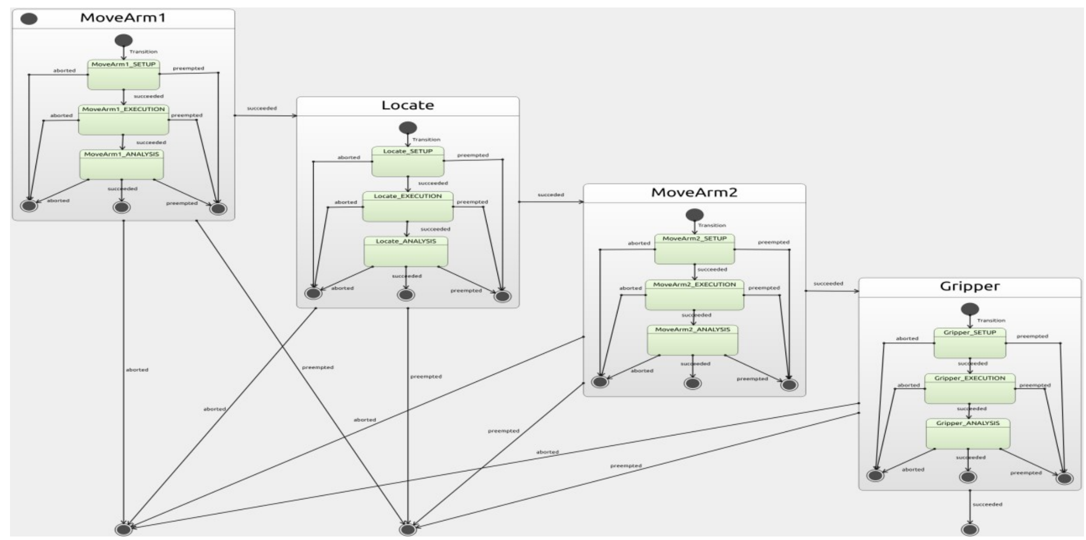




On ROS: Smach



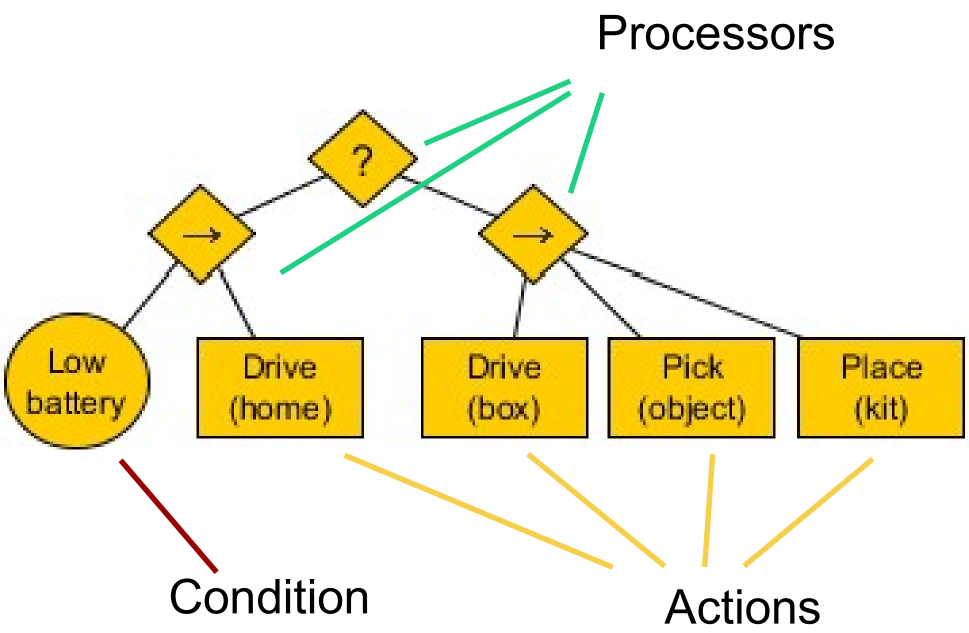
Can be grouped hierarchically (H-FSMs)



[http://wiki.ros.org/smach]

Behavior trees

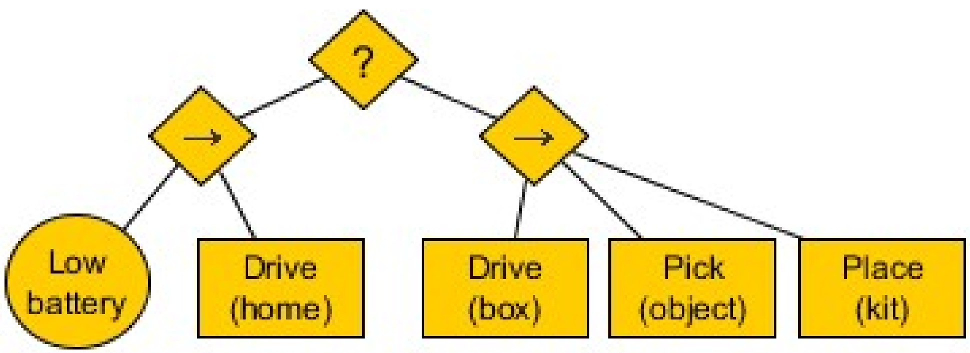
- Execution is divided into discrete *ticks*, which propagate from the root node periodically
- Inner nodes are **processors,** defining which branch to tick
- Tree where leafs are ***actions*** to execute or **conditions** to evaluate
- Actions return 1 of 3 states: Success, Failure or Running
- Conditions return either Success or Failure



Processors

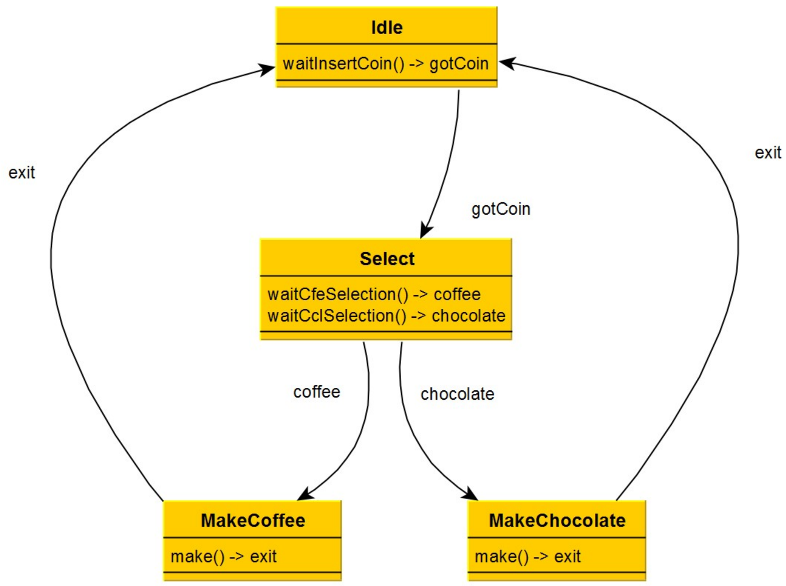
Processors defines the policy that a *compound* skill uses to access its children. We define the following processors:

- **Serial** (->): Process children in sequence until all succeed. Returns on first occurrence of a running of failed skill.
- **Selector** (?): Process children in sequence until one succeeds, ignoring failures. Returns on first occurrence of a running or successful skill.
- **ParallelFf** (Parallel First Fail): Process children in parallel until all succeed. Stop all processes if a child fails.
- **ParallelFs** (Parallel First Stop): Process children in parallel until one succeed. Stop all processes if a child finishes (succeeded/fail).



Exercise

FSM vs. BT



FSM

?

BT

SkiROS

Software platform for robots coordination

Main features

Reactive execution engine based on Behavior Trees

Keep robot behaviors organized into modular skill libraries

A semantic database to manage environmental knowledge

ROS - enabled

Install SkiROS

(run these commands from your ROS workspace src)

```
git clone https://github.com/RVMI/skiros2
git clone https://github.com/RVMI/skiros2_std_lib
git clone https://github.com/RVMI/skiros2_examples
sudo apt install python-pip ros-melodic-rosmon
catkin build && source ../devel/setup.bash
roscd skiros2/..
pip install -r requirements.txt --user
```

Verify:

```
roslaunch skiros2_examples simple_params_example.launch
```

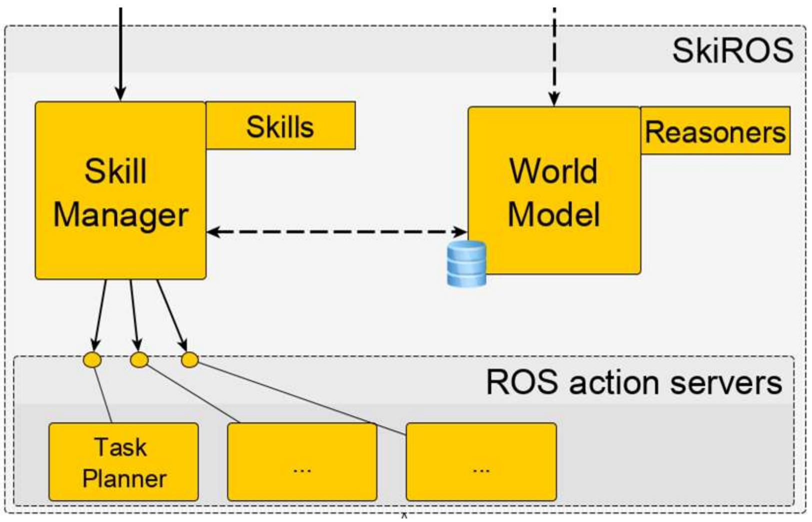
Break

SkiROS architecture

World model
The semantic database, holds the world state and offers services to modify it and reason on it

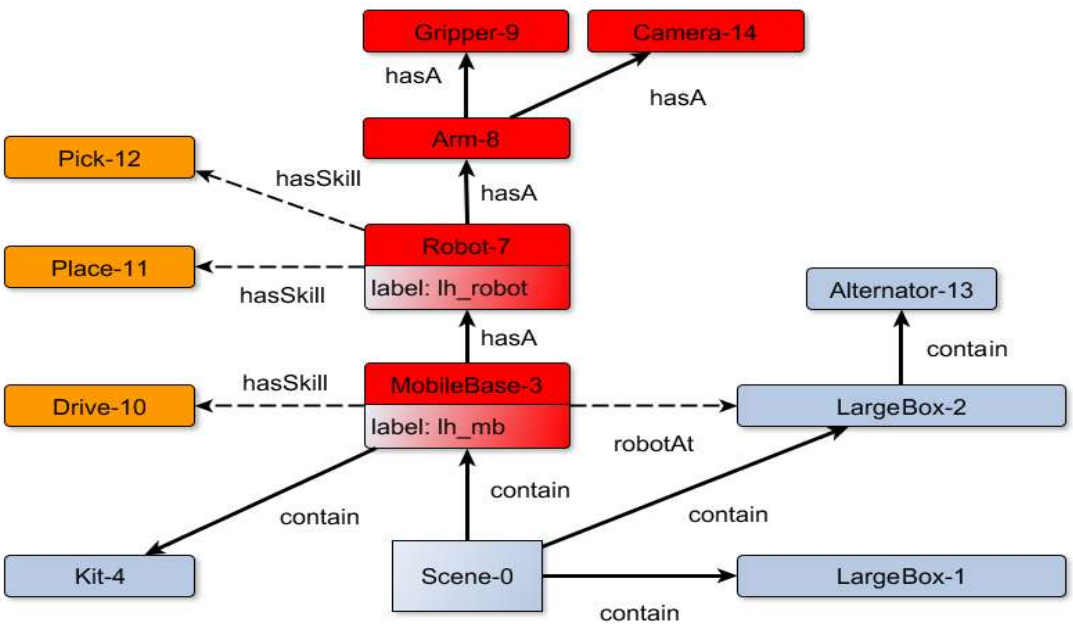
Skill manager
The execution engine, manages information about available skills and offers services to execute and monitoring

Skills
Complex primitive skills can be exposed as ROS actions



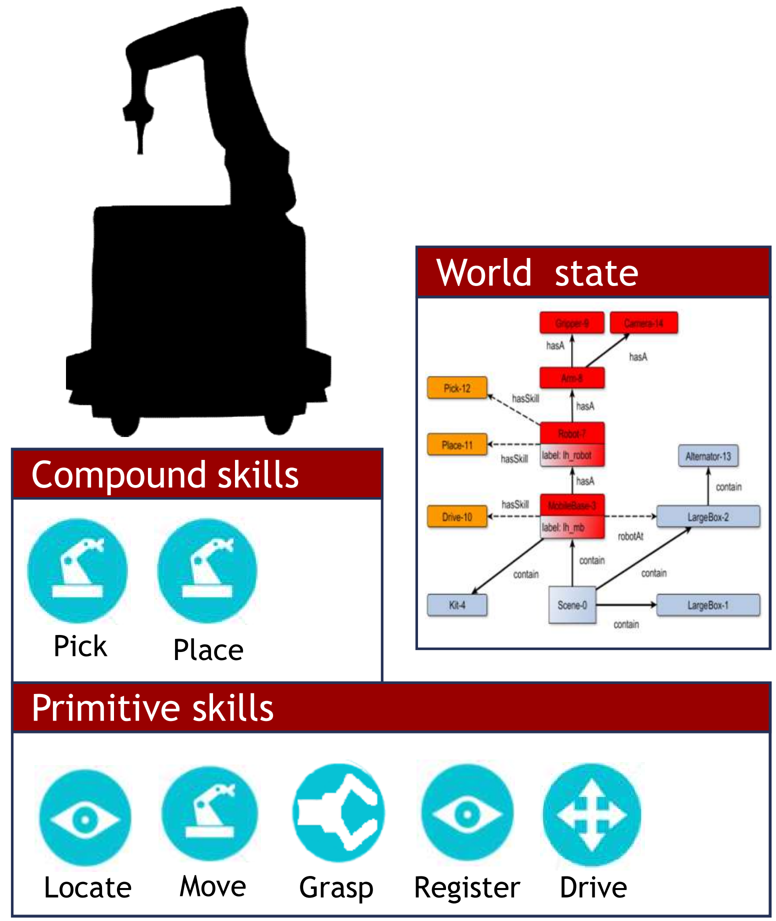
World model

Scene graph
Model of the current world state for reasoning, planning and execution



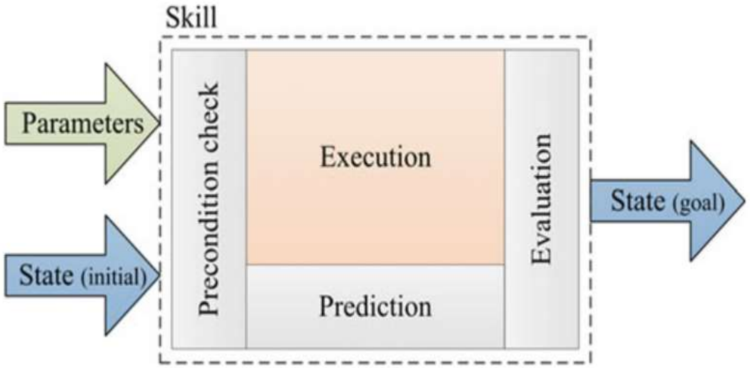
Skill concept

Skill	A process that can change the state of the robot and its environment.
Primitive skill	A command that resides at the lowest level of the hierarchy and can be directly executed by the robot platform
Compound skill	A hierarchically organized collection of skills.
World state	Contains the current state of the world known by the robot



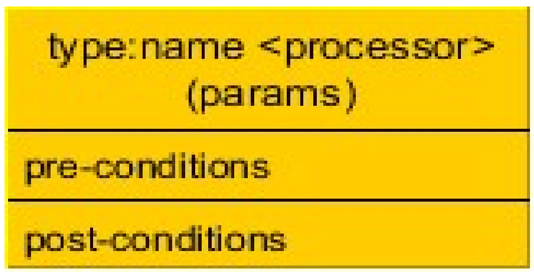
Skills

Definition:
A skill allows to transition from one world state to another, if its preconditions are met

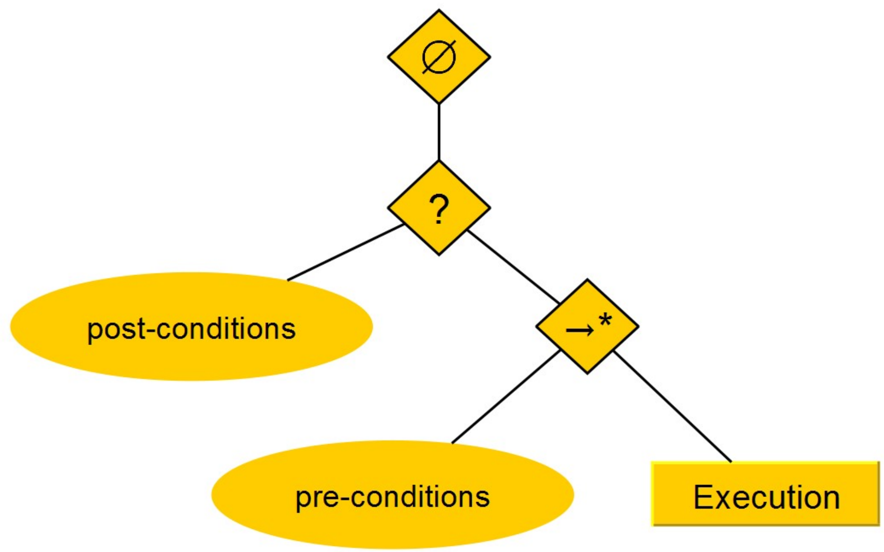


Skill vs. BT

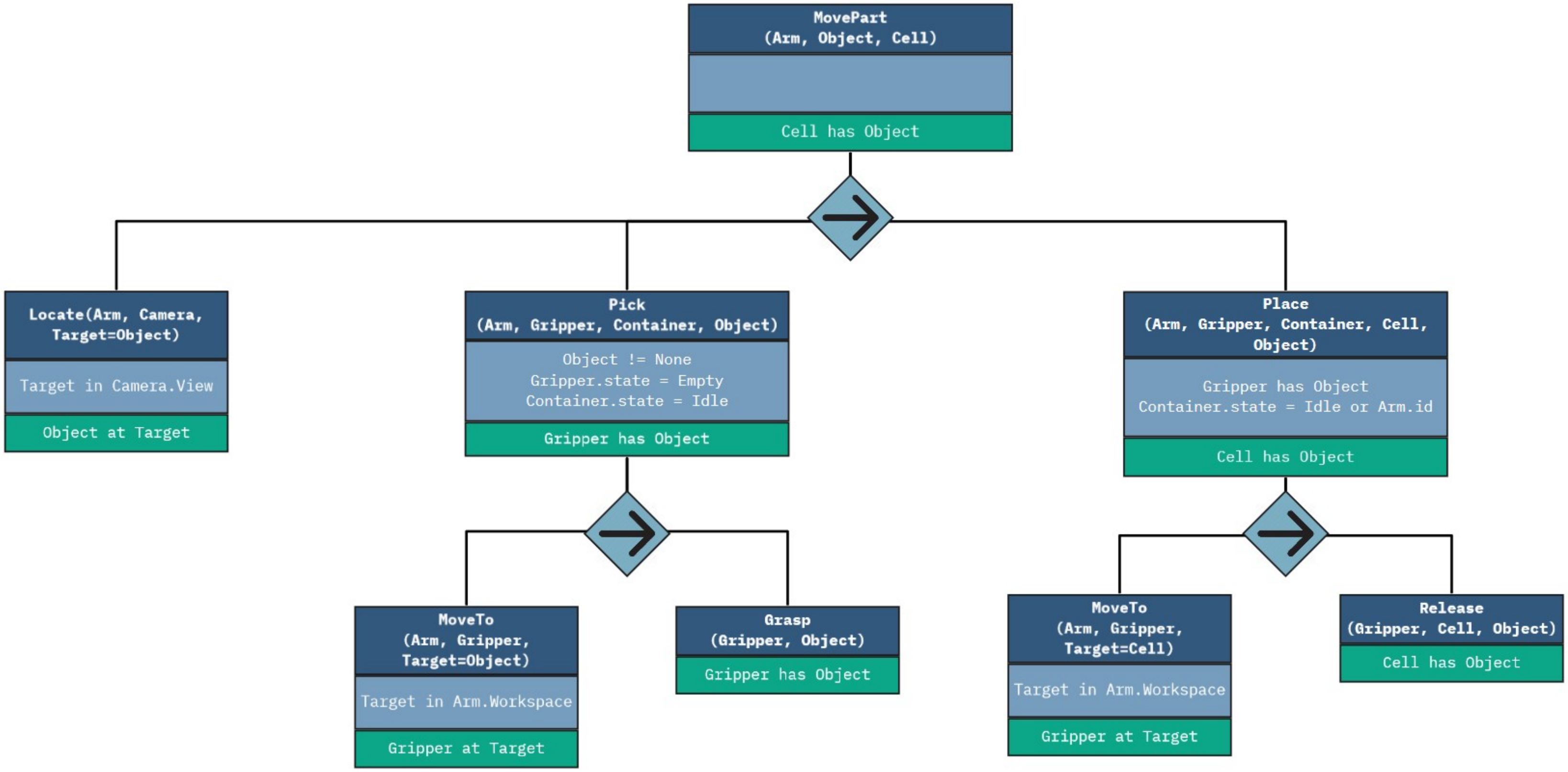
Skill Template



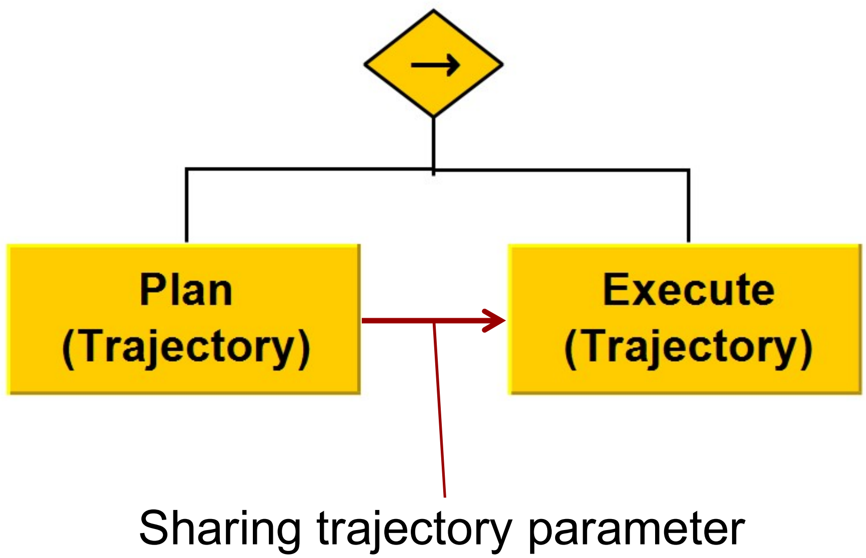
Equivalent classical BT

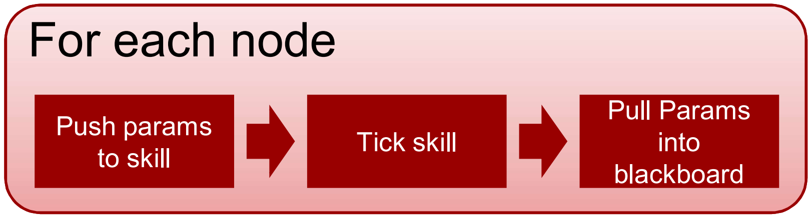


Behavior trees with skills

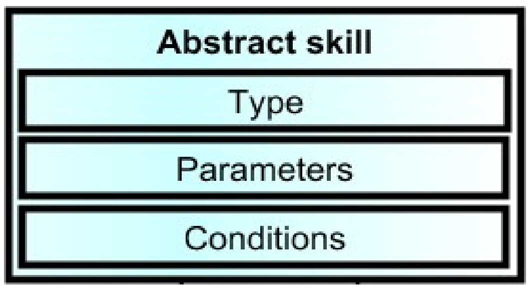


Skill communication: Blackboard





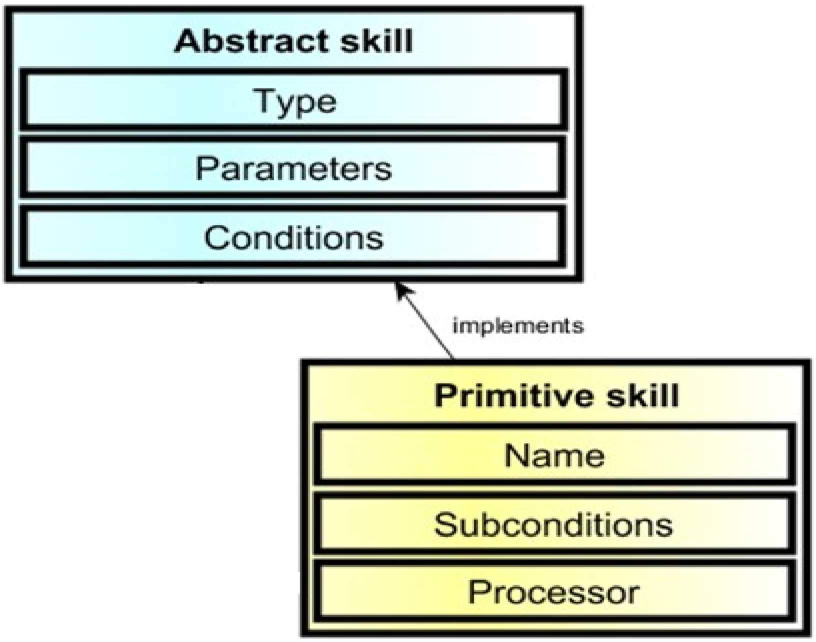
Abstract skill



Composed of

- a *type*, a unique symbol identifying the abstract skill
- a set of *parameters* $\mathbf{x}_1, \ldots, \mathbf{x}_n$ that maps into objects in the world model (world) or configure the behavior (config)
- *conditions*, a set of constraints over the input parameters. Further divide into pre-conditions and post-conditions

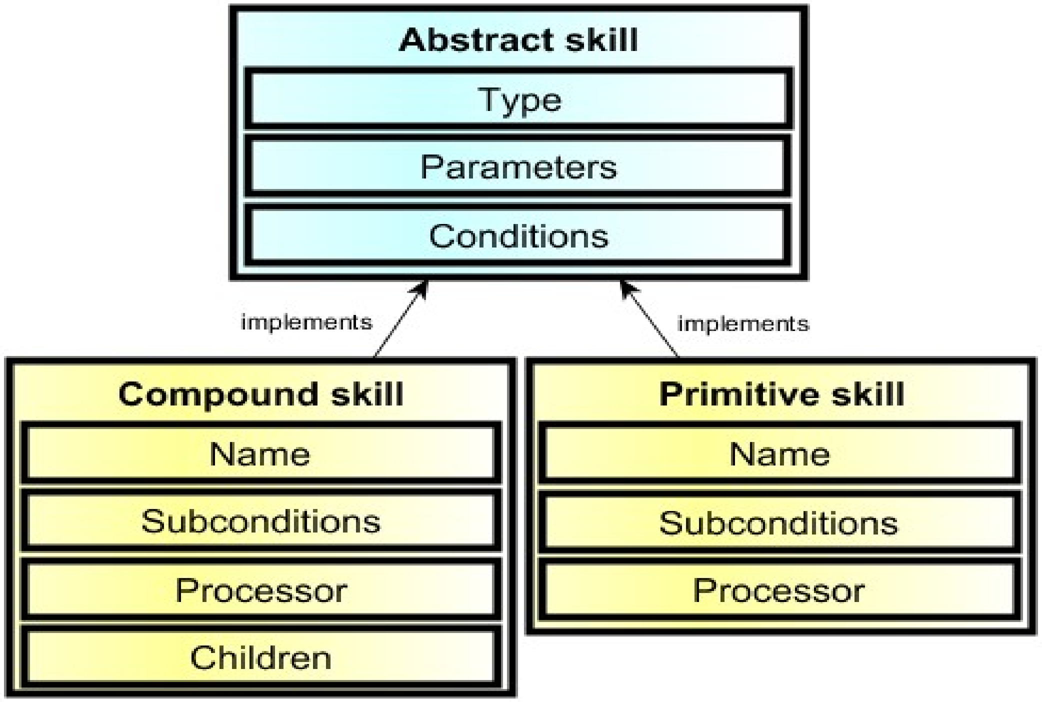
Primitive skill



Composed of

- a *name*, a unique symbol identifying the implementation
- *subconditions*, a set of additional constraints over the input parameters.
- *processor*, an executable

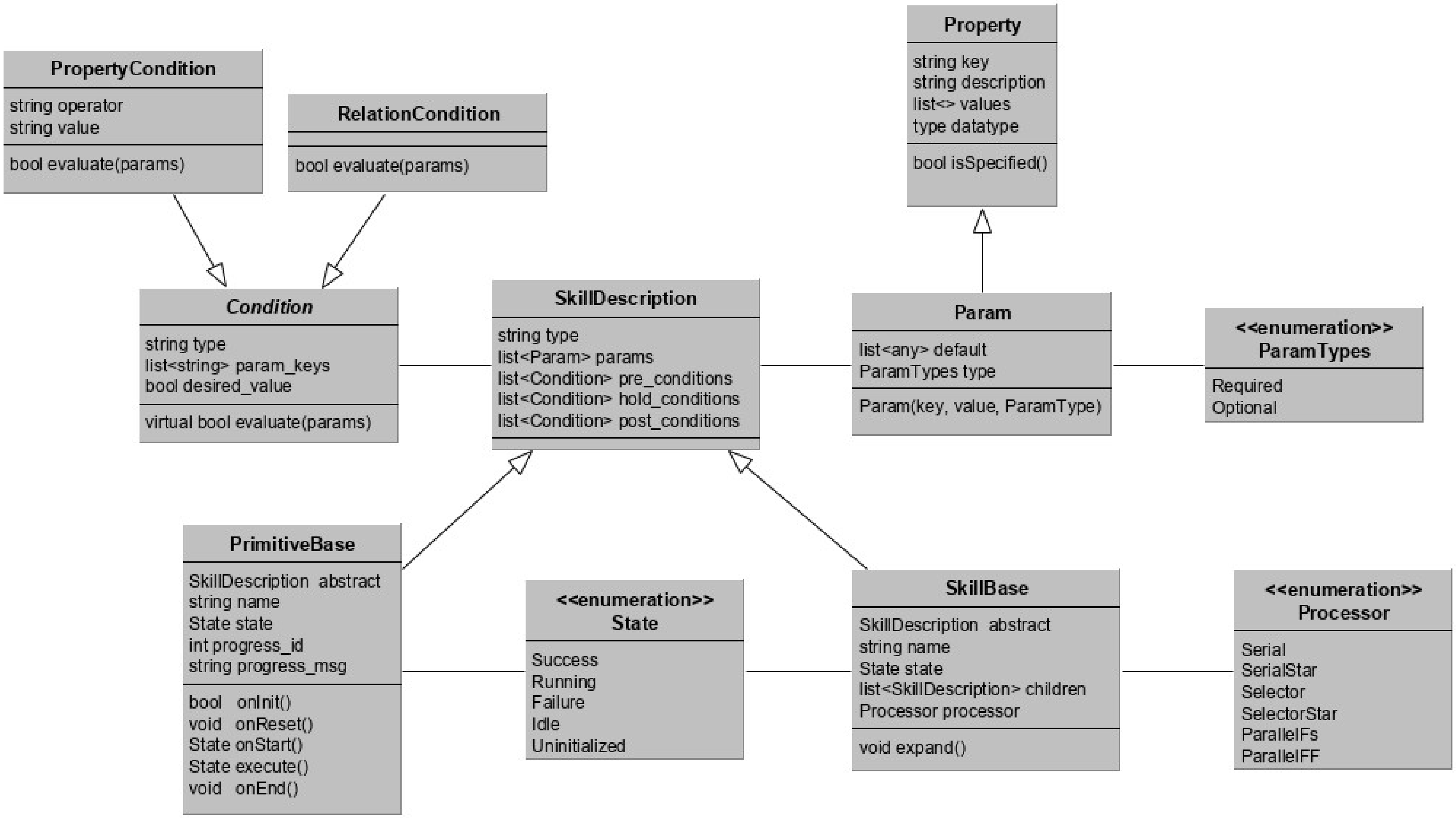
Compound skill



Composed of

- *processor*, defines the way the children are coordinated
- *children*, a set of skills to coordinate

UML diagram



PropertyCondition
string operator
string value
bool evaluate(params)
RelationCondition
bool evaluate(params)
Property
string key
string description
list<> values
type datatype
bool isSpecified()
Condition
string type
list<string> param_keys
bool desired_value
virtual bool evaluate(params)
SkillDescription
string type
list<Param> params
list<Condition> pre_conditions
list<Condition> hold_conditions
list<Condition> post_conditions
Param
list<any> default
ParamTypes type
Param(key, value, ParamType)
<<enumeration>>
ParamTypes
Required
Optional
PrimitiveBase
SkillDescription abstract
string name
State state
int progress_id
string progress_msg
bool onInit()
void onReset()
State onStart()
State execute()
void onEnd()
<<enumeration>>
State
Success
Running
Failure
Idle
Uninitialized
SkillBase
SkillDescription abstract
string name
State state
list<SkillDescription> children
Processor processor
void expand()
<<enumeration>>
Processor
Serial
SerialStar
Selector
SelectorStar
ParallelFs
ParallelFF

Exercise

Setup exercise

```
roscd skiros2_examples
git checkout feature/AvoidTurtlesDemo
rosed skiros2_examples avoid_the_turtles_demo.py
```

Exercise

Follow tutorials on: https://github.com/RVMI/skiros2/wiki. Have a look in particular to the turtlesim tutorial (last one).

Create a Behavior Tree in SkiROS to control SuperT, a turtle operating in the kitting area of an aquatic manufacturing line. The turtle has to collect 3 different type of parts: shells, corals and algae, located at the turtles Nina, Pinta and SantaMaria. A part is considered collected when the turtle reaches the related turtles. Consider the following tasks:

1. Collect 3 different parts and deliver to MissT.
2. Get a dynamic order with different type/number of parts, collect them and deliver to MissT.