Fawkes Behavior Engine

Till Hofmann, Victor Mataré, Johannes Rothe, Tim Niemueller

1 Introduction

2 Behavior Engine and Skills

3 Hybrid State Machine

4 Skill Execution

5 Skill Implementation

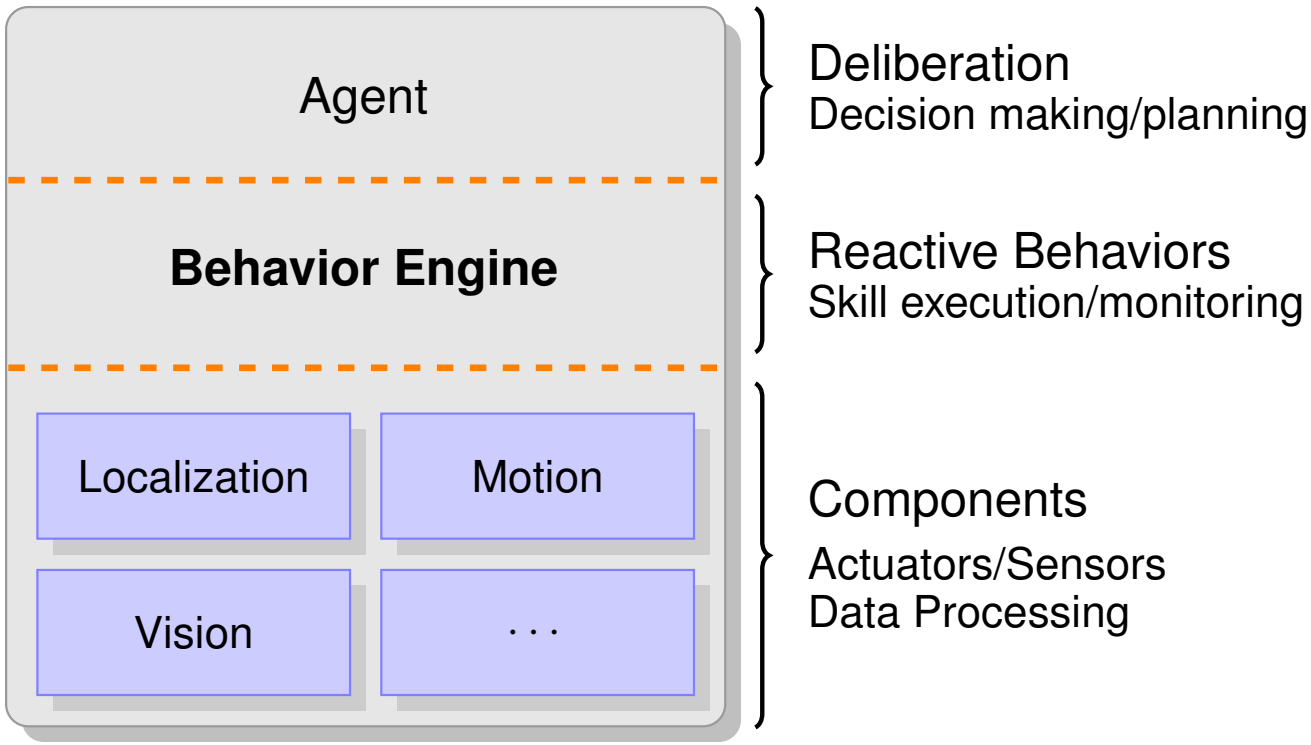
6 Hands-On

Introduction – Problem statement

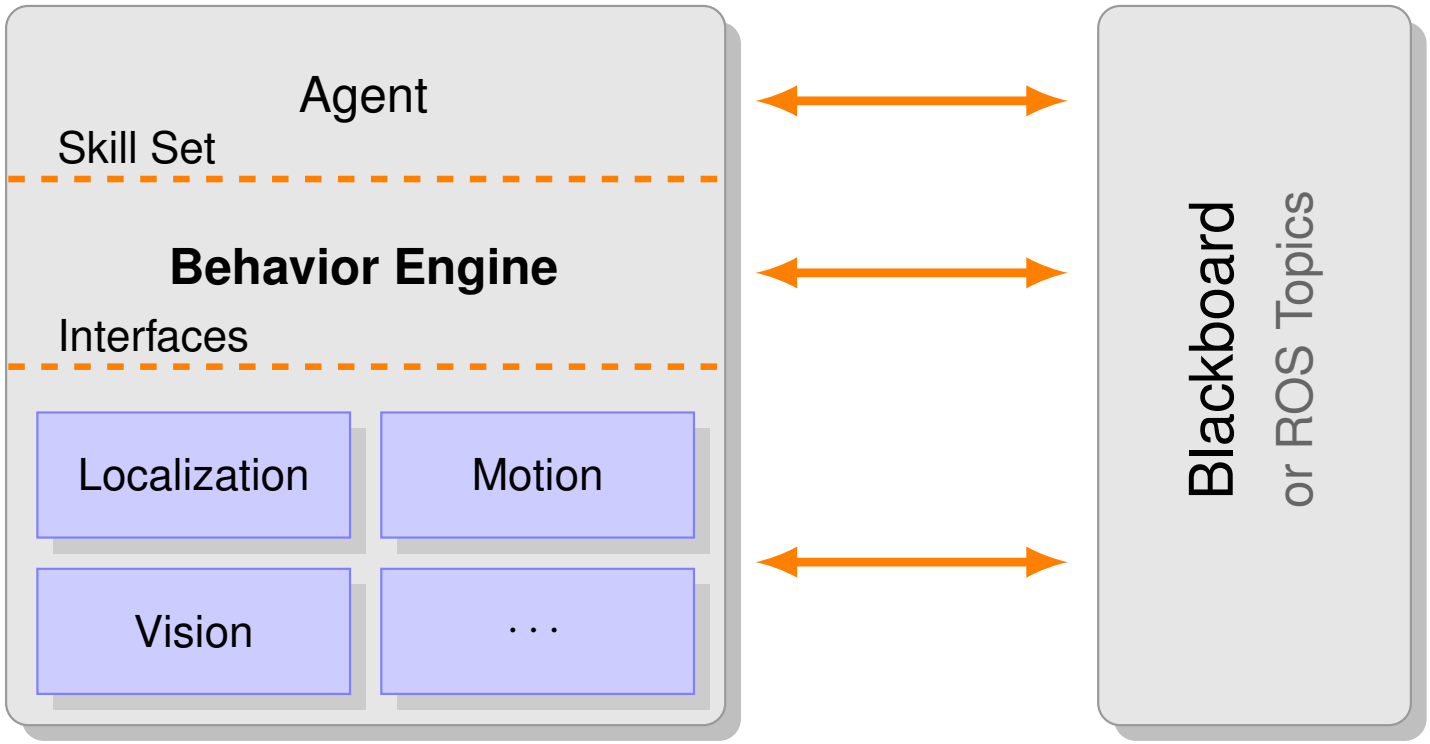
High-level reasoning generates sequence of primitive actions

- Separate strategic from local decisions
- Middle layer between agent and low-level components
- Define elementary behavioral units
- Instruct and then monitor the functional components
- Quick execution to operate at soft-realtime
- Easy to learn and use

Introduction – System Architecture



Introduction – System Architecture



Behavior Engine

- Framework to develop, execute, and monitor skills
- Provides a skill set, which defines the available skills
- Runs in a single plugin called skiller (**skill** **e**xecution **r**un-time)
- Access to config, blackboard, logging, clock . . .
- No compiling, automated reload (via inotify)

Behavior Engine and Skills

Skills

- Performs a simple, isolated task (e.g. close gripper)
- Hierarchical calling (allows ‘complex’ skills)
- Each implemented as a Lua module
- Execution state updated @ 15 Hz
- Three possible statuses:
  `S_RUNNING`, `S_FINAL`, `S_FAILED`

Behavior Engine and Skills

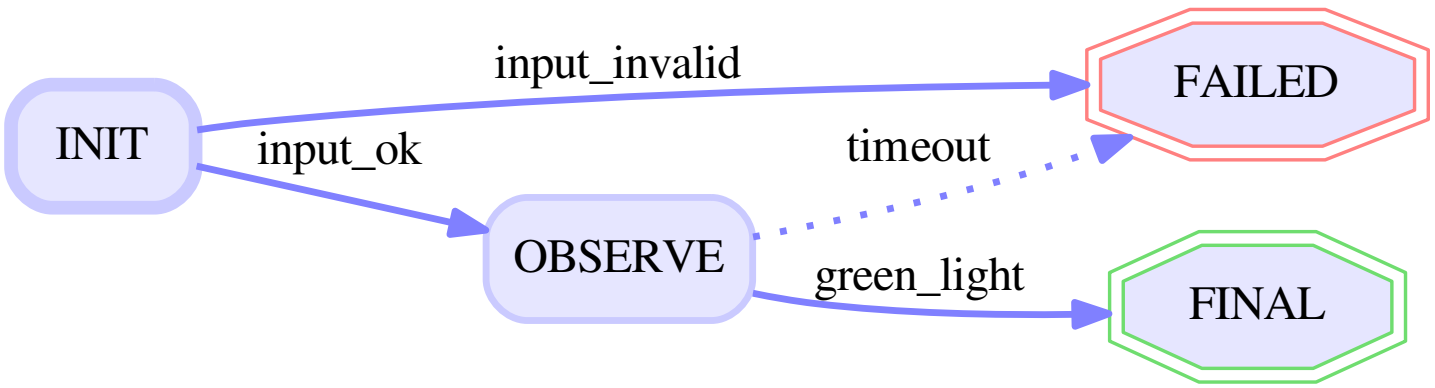
Skills

- Performs a simple, isolated task (e.g. close gripper)
- Hierarchical calling (allows ‘complex’ skills)
- Each implemented as a Lua module
- Execution state updated @ 15 Hz
- Three possible statuses:
  `S_RUNNING`, `S_FINAL`, `S_FAILED`

Modeled as **Hybrid State Machines**

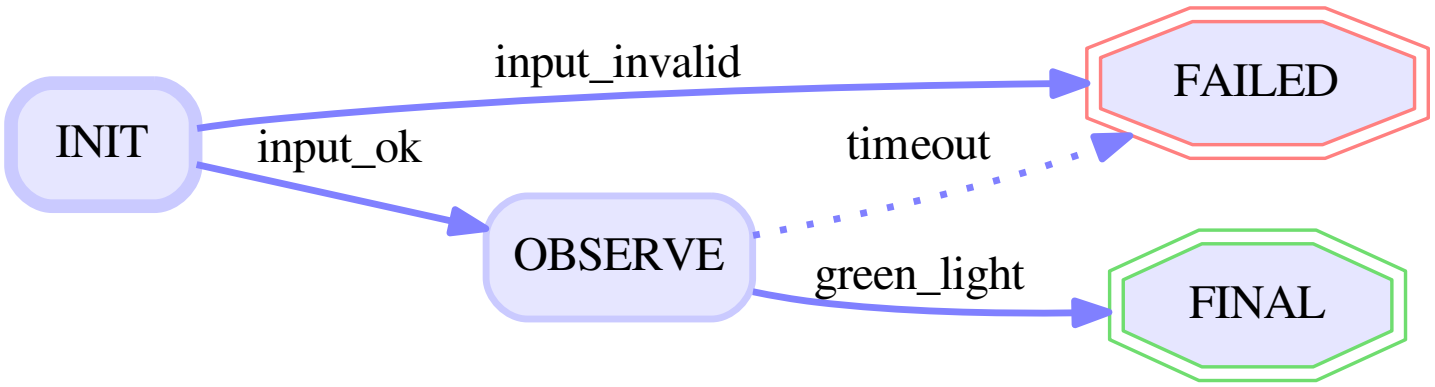
Hybrid State Machine (HSM)

Example



Hybrid State Machine (HSM)

Example

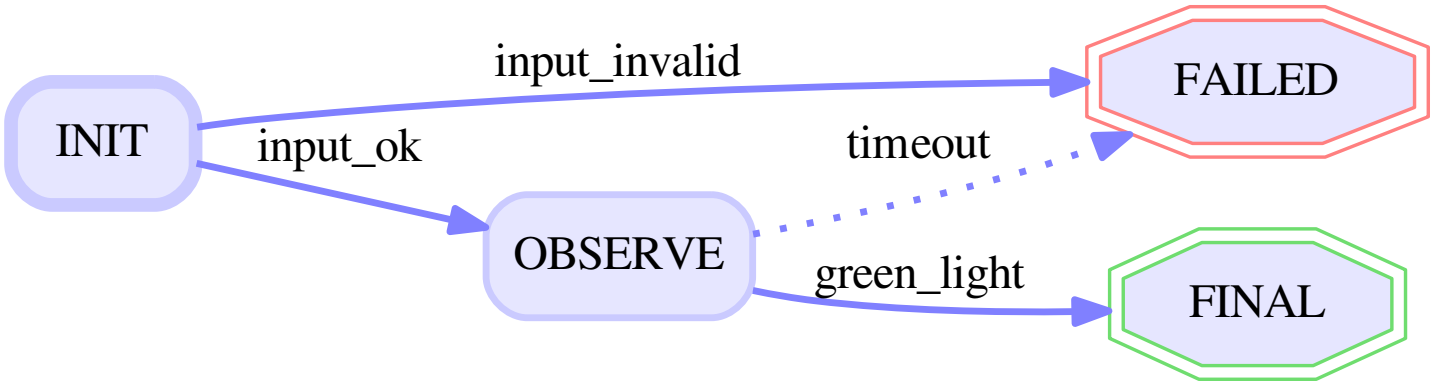


Differences from normal state machines

- Action occurs in the states
- Can access & modify global variables
- Both continuous processes and discrete state changes

Hybrid State Machine (HSM)

Example

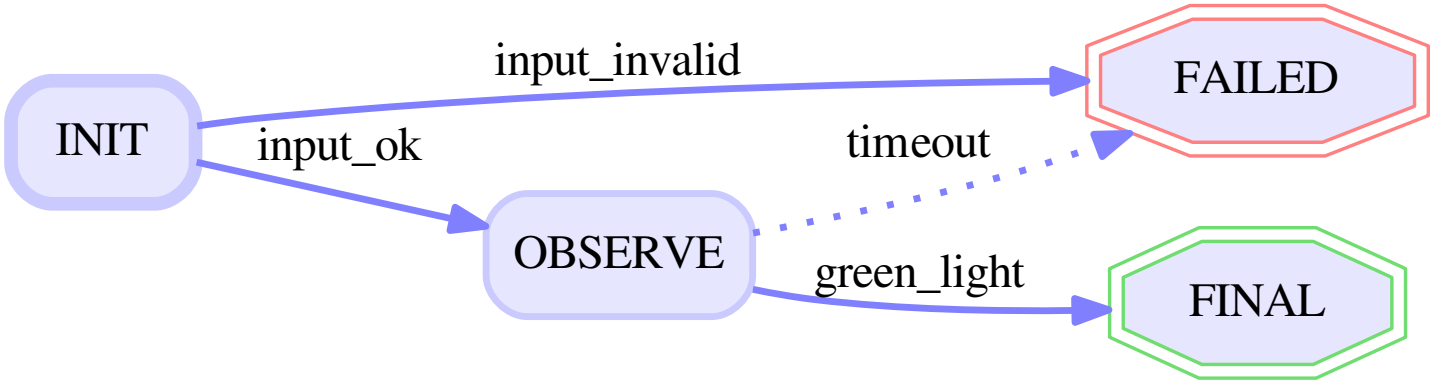


States

- Incoming/outgoing transitions
- Various hook functions
- Special **skill**-states that execute subskills

Hybrid State Machine (HSM)

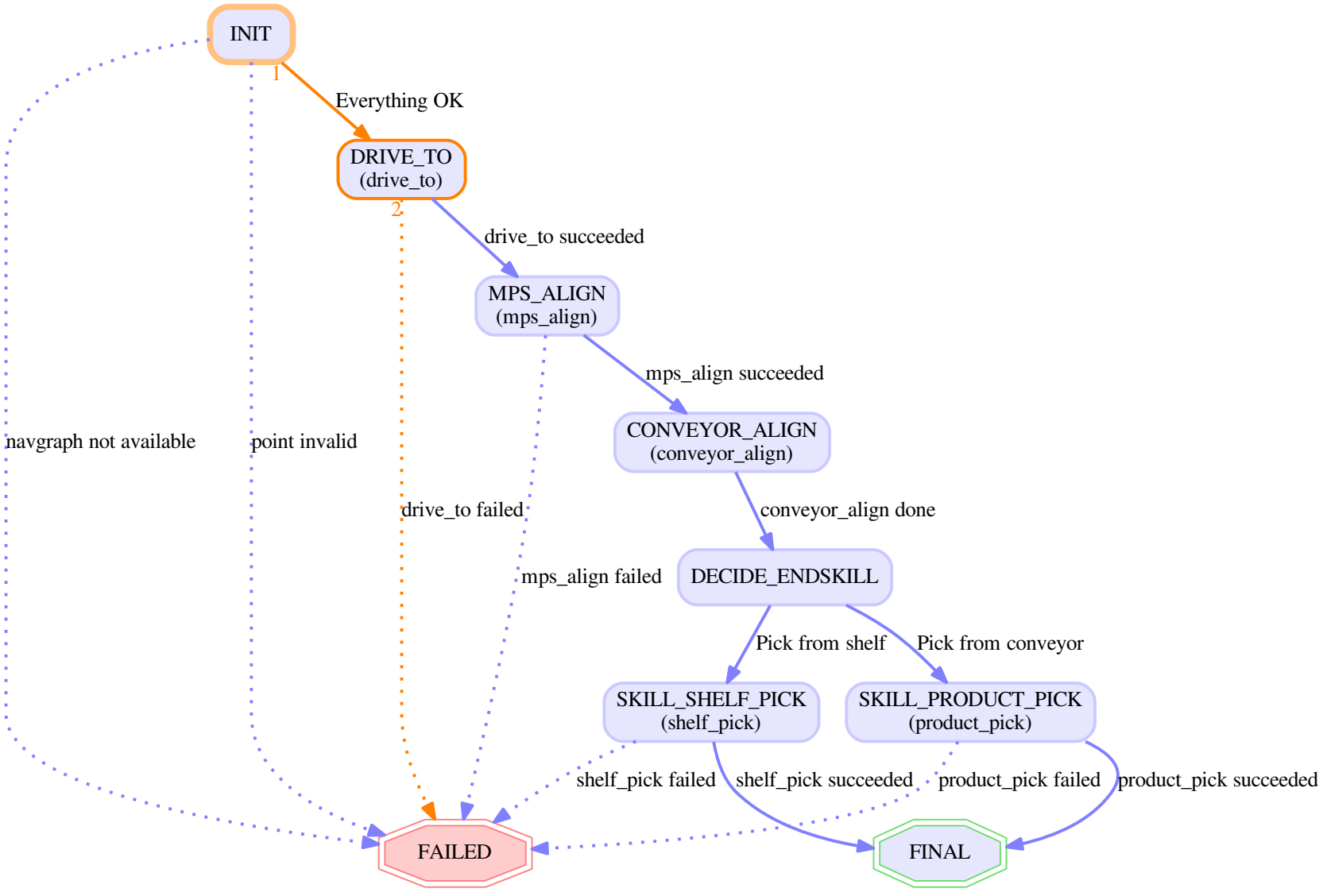
Example



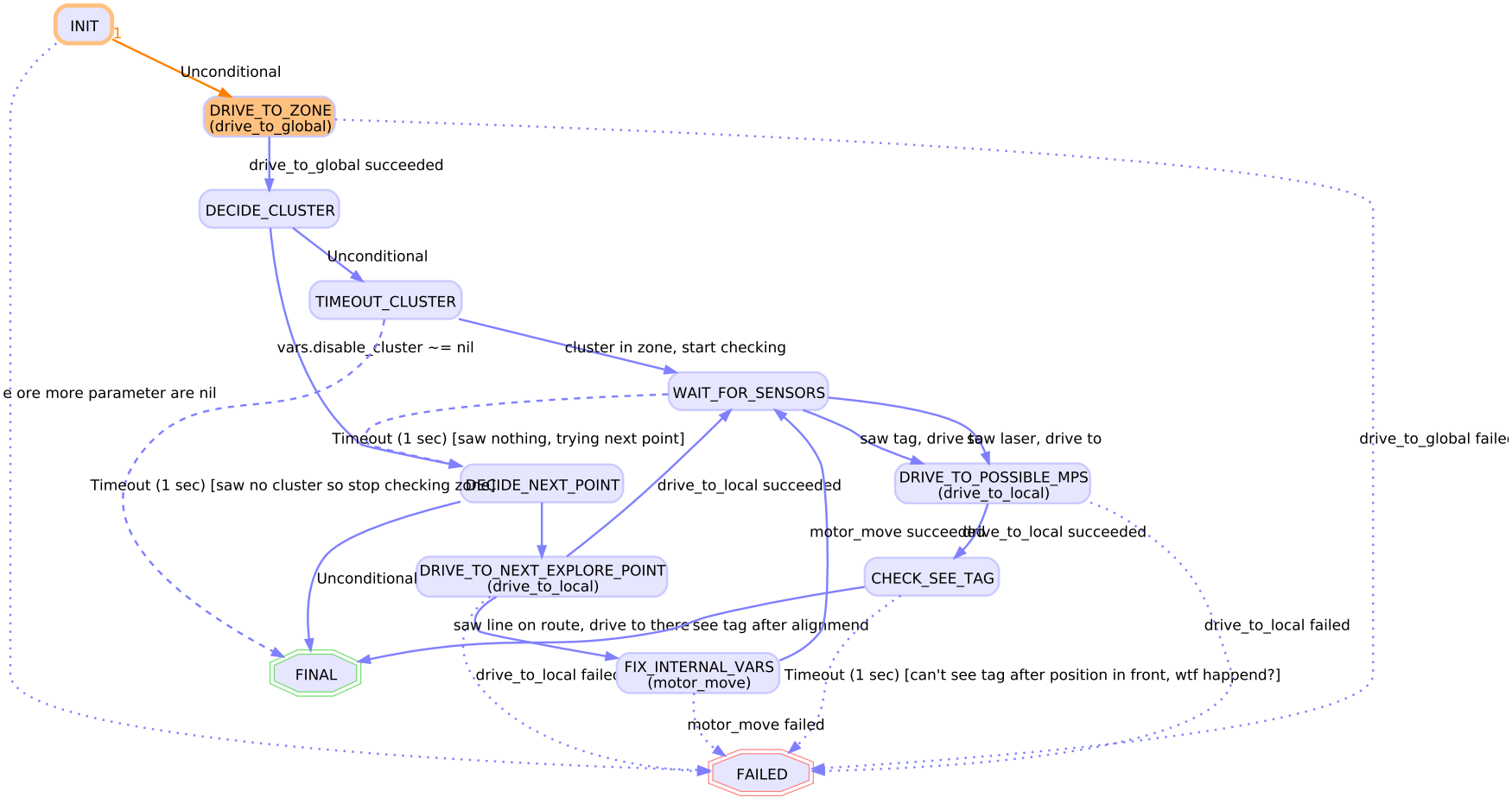
Transitions

- Originating state
- Target state
- Condition (boolean function)
- Timeout

Example Graph: `get_product_from`



Example Graph: `explore_zone`



Skill Execution: Control Flow

Given: Current skill, current state `CUR_STATE`

1. Evaluate skill module
2. Execute `CUR_STATE:loop()`
3. Test all outgoing transitions of `CUR_STATE`
4. If a condition `c` is `true`:
   - Execute `CUR_STATE:exit()`
   - `CUR_STATE` is now `c`'s target state
   - Execute `CUR_STATE:init()`
5. Repeat from 2. @15 Hz!

Skill Execution: The Lua Programming Language

- Powerful, fast, light-weight, embeddable
- Tables as ubiquitous data structure
- Easy C++ integration
- Flexible, yet simple programming environment
- Usage in other projects
  - NASA Space Shuttle Hazardous Gas Detection System
  - CryEngine Game AI, World of Warcraft Extension API

Skill Implementation: Overview

```
--------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--             2015  Tobias Neumann
--------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2


function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end


fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
    fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
    final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me !=)
}
```

```
fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}
```

```
function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Skill Implementation: Overview

```
--------------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--              2015  Tobias Neumann
--------------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name                = "goto"
fsm                 = SkillHSM:new{name=name, start="INIT"}
depends_skills      = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation       = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2
```

```
function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end


fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me :=)
}


fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO", cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}


function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
            ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Boilerplate

Skill Implementation: Overview

```
--------------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--             2015  Tobias Neumann
--------------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2

function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end

fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me ;-)
}

fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}


function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Jump-Conditions

```
function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end
```

Skill Implementation: Overview

```
----------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--             2015  Tobias Neumann
----------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2


function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end

fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me ;=)
}

fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}


function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

States

```
fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me :=)
}
```

Skill Implementation: States

State definition

```
fsm:define_states{ export_to = _M,
  { "INIT", JumpState },
  { "OBSERVE", JumpState}
}
```

Syntactic explanation

- `fsm` = global Lua object representing state machine
- `define_states` = method called “on” `fsm` object
- One table passed as argument to `define_states`:
  - One element named `export_to`
  - Two unnamed elements that are again tables

Skill Implementation: States

State definition

```
fsm:define_states{ export_to = _M,
  { "INIT", JumpState },
  { "OBSERVE", JumpState}
}
```

Sugar-free syntax

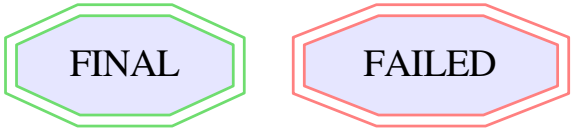
```
fsm.define_states( fsm,
  { export_to = _M,
    { "INIT", JumpState },
    { "OBSERVE", JumpState }
  }
)
```

Skill Implementation: States

State definition

```
fsm:define_states{ export_to = _M,
  { "INIT", JumpState },
  { "OBSERVE", JumpState}
}
```

Effect

- Predefined states already exist: 

- Create two new state objects: 

Skill Implementation: Subskills

State definition

```
fsm:define_states{ export_to = _M,
  { "MOVE_FORWARD", SkillJumpState,
    skills={{"motor_move"}},
    final_to="FINAL", fail_to="FAILED"
  }
}
```

Meaning

- Execute `motor_move` skill in `MOVE_FORWARD` state
- Transition to `FINAL` if & when `motor_move` ends with `FINAL`
- Transition to `FAILED` if & when `motor_move` ends with `FAILED`

Skill Implementation: Overview

```
----------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--             2015  Tobias Neumann
----------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2


function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end


fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me ;-)
}

fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
       final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}

function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Transitions

```
fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
       final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}
```

Skill Implementation: Transitions

Transition Definition

```
fsm:add_transitions{
  {"INIT", "OBSERVE", cond=input_ok},
  {"INIT", "OBSERVE", cond="not input_ok()"},
  {"OBSERVE", "FAILED", timeout=5},
  {"OBSERVE", "FINAL", cond=green_light}
}
```

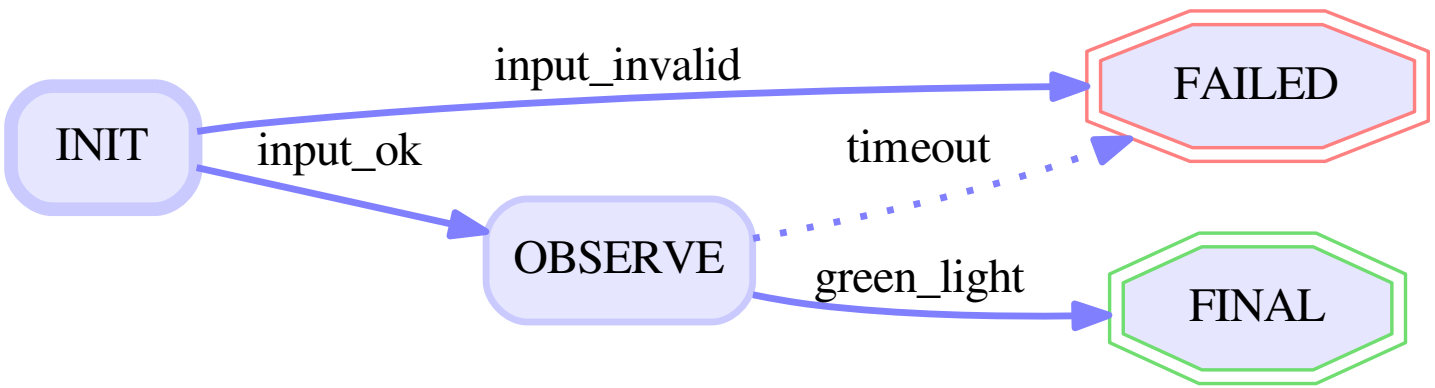
Three ways of defining a condition

1. Function reference (i.e. a function name)
2. Interpreted Lua expression (as a string)
3. Timeout

Skill Implementation: Transitions

Transition Definition

```
fsm:add_transitions{
   {"INIT", "OBSERVE", cond=input_ok},
   {"INIT", "OBSERVE", cond="not input_ok()"},
   {"OBSERVE", "FAILED", timeout=5},
   {"OBSERVE", "FINAL", cond=green_light}
}
```



Skill Implementation: Overview

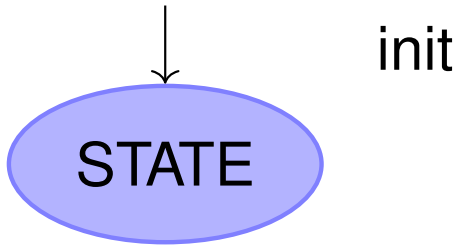
```
-- goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--             2015  Tobias Neumann
-----------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2


function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end


fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
    fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
    final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me ;=)
}


fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO",  cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
     Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}
```

```
function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
       y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

State-Functions

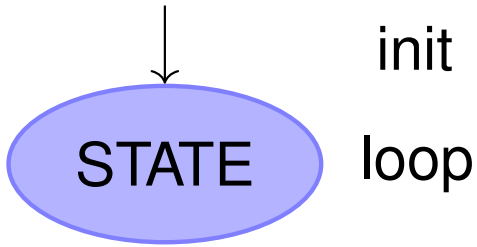
```
function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
       y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
             ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Skill Implementation: State hooks



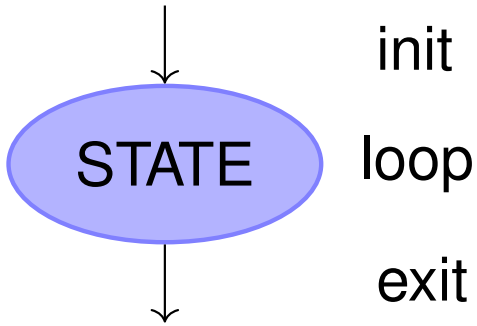
Hook function	**Executed...**
STATE:init()	once when *entering* state

Skill Implementation: State hooks



Hook function	**Executed...**
STATE:init()	once when *entering* state
STATE:loop()	15 times/second *while* state is active

Skill Implementation: State hooks



Hook function	**Executed...**
STATE:init()	once when *entering* state
STATE:loop()	15 times/second *while* state is active
STATE:exit()	once when *leaving* state

Skill Implementation: Subskills

State definition

```
fsm:define_states{ export_to = _M,
   { "MOVE_FORWARD", SkillJumpState,
     skills={{"motor_move"}},
     final_to="FINAL", fail_to="FAILED"
   }
}
```

Skill Implementation: Overview

```
--------------------------------------------------------------------------------
--  goto.lua -
--
--  Created: Thu Aug 14 14:32:47 2008
--  modified by Victor Matare
--              2015  Tobias Neumann
--------------------------------------------------------------------------------
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License as published by
--  the Free Software Foundation; either version 2 of the License, or
--  (at your option) any later version.
--
--  This program is distributed in the hope that it will be useful,
--  but WITHOUT ANY WARRANTY; without even the implied warranty of
--  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
--  GNU Library General Public License for more details.
--
--  Read the full text in the LICENSE.GPL file in the doc directory.

-- Initialize module
module(..., skillenv.module_init)

-- Crucial skill information
name               = "goto"
fsm                = SkillHSM:new{name=name, start="INIT"}
depends_skills     = { "relgoto", "global_motor_move" }
depends_interfaces = {
   {v = "pose", type="Position3DInterface", id="Pose"},
   {v = "navigator", type="NavigatorInterface", id="Navigator"},
}

documentation      = [==[Move to a known location via place or x, y, ori.
if place is set, this will be used and x, y and ori will be ignored
@param place  Name of the place we want to go to.
@param x      x we want to drive to
@param y      y we want to drive to
@param ori    ori we want to drive to
]==]
-- Initialize as skill module
skillenv.skill_module(_M)

local tf_mod = require 'tf_module'
-- Tunables
local REGION_TRANS=0.2
```

```
function check_navgraph(self)
  return self.fsm.vars.place ~= nil and not navgraph
end

function reached_target_region(self) -- function name(parameter) body end
  local region_sqr = self.fsm.vars.region_trans * self.fsm.vars.region_trans
  local distance_region_sqr = rel_pos.x * rel_pos.x + rel_pos.y * rel_pos.y
  if distance_region_sqr > region_sqr then
    return false
  else
    return true
  end
end


fsm:define_states{ export_to=_M,
  closure={check_navgraph=check_navgraph,
      reached_target_region=reached_target_region},
  {"INIT", JumpState},
  {"SKILL_RELGOTO", SkillJumpState, skills={{relgoto}}, final_to="FINAL",
      fail_to="FAILED",desc="Do a relgoto"},
  {"REGION_REACHED_STOPPING", JumpState},
  {"FINAL_ORIENTATION", SkillJumpState, skills={{global_motor_move}},
      final_to="FINAL", fail_to="FINAL"}, -- will be ok; trust me :=)
}


fsm:add_transitions{
  {"INIT", "FAILED",precond=check_navgraph, desc="no navgraph"},
  {"INIT", "FAILED",cond="not vars.target_valid", desc="target invalid"},
  {"INIT", "SKILL_RELGOTO", cond=true},
  {"SKILL_RELGOTO", "INIT", timeout=1, desc="Recalculate target"},
  {"SKILL_RELGOTO", "REGION_REACHED_STOPPING", cond=reached_target_region, desc="
      Reached target, stopping local-planner"},
  {"REGION_REACHED_STOPPING", "FINAL_ORIENTATION", cond="vars.ori", desc="Stopped, do
      final orientation"},
  {"REGION_REACHED_STOPPING", "FINAL",cond="not vars.ori", desc="Stopped, and don't
      need to orientate => FINAL"},
}


function INIT:init()
  self.fsm.vars.target_valid = true

  -- if a place is given, get the point from the navgraph and use this instead of x,
      y, ori
  if self.fsm.vars.place ~= nil then
    self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
    if self.fsm.vars.node:is_valid() then
      self.fsm.vars.x = self.fsm.vars.node:x()
      self.fsm.vars.y = self.fsm.vars.node:y()
      if self.fsm.vars.node:has_property("orientation") then
        self.fsm.vars.ori   = self.fsm.vars.node:property_as_float("orientation");
      else
        self.fsm.vars.ori   = nil   -- if orientation is not set, we don't care
      end
    else
      self.fsm.vars.target_valid = false
    end
  end
  self.fsm.vars.region_trans = self.fsm.vars.region_trans or REGION_TRANS
end

function SKILL_RELGOTO:init()
        self.args["relgoto"] = { x = self.fsm.vars.rel_x, y = self.fsm.vars.rel_y,
            ori = self.fsm.vars.rel_ori }
end

function REGION_REACHED_STOPPING:init()
  local msg = navigator.StopMessage:new( )
  navigator:msgq_enqueue_copy(msg)
end

function FINAL_ORIENTATION:init()
  self.args["global_motor_move"] = {ori=self.fsm.vars.ori}
end
```

Boilerplate

Skill Implementation: Subskill arguments

`motor_move` skill documentation

```
documentation = [==[
  Move on a (kind of) straight line relative
     to the given coordinates.
  @param x The target X coordinate, relative
     to /base_link
  @param y Dito
  [...]
]==]
```

Skill Implementation: Subskills

State definition

```
fsm:define_states{ export_to = _M,
  { "MOVE_FORWARD", SkillJumpState,
    skills={{"motor_move"}},
    final_to="FINAL", fail_to="FAILED"
  }
}
```

Subskill arguments

```
function MOVE_FORWARD:init()
  self.args["motor_move"].x = 1
end
```

Skill Implementation: Handling arguments

`motor_move` implementation (simplified)

```
function DRIVE:loop()
  -- [...]
  local vx = calc_x_speed(self.fsm.vars.x)
  motor:set_x_speed(vx)
  -- [...]
end
```

Variables

Three types of variables:

1. Lua local variable
2. Lua global variable
3. fsm.vars table

Variables

Three types of variables:

1. Lua local variable
2. Lua global variable
3. fsm.vars table

lua local

```
function my_function()
  local foo = "bar"
  -- code here can use foo
end
-- code here can't use foo
```

- local scope
- in functions, loops, conditionals

Variables

Three types of variables:

1. Lua local variable
2. Lua global variable
3. fsm.vars table

Lua global

```
local MY_CONSTANT = 10
-- code here can use MY_CONSTANT
function my_function()
  -- code here can use MY_CONSTANT
end
```

- global scope in the skill
- reset before each skiller iteration
- so be careful, only use for constants

Variables

Three types of variables:

1. Lua local variable
2. Lua global variable
3. fsm.vars table

fsm.vars table

```
self.fsm.vars.foo = "bar"
```

- table with variables defined in the state machine
- initially filled with the skill parameters
- persistent while the skill is running
- reset when the skill finished

Use blackboard interfaces

1. **In fawkes-robotino/cfg/conf.d/skiller.yaml**

```
skiller/interfaces/robotino:
  reading:
    motor: MotorInterface::Robotino
  writing:
    light: RobotinoLightInterface::Light determined
```

Use blackboard interfaces

1. **In fawkes-robotino/cfg/conf.d/skiller.yaml**

```
skiller/interfaces/robotino:
  reading:
    motor: MotorInterface::Robotino
  writing:
    light: RobotinoLightInterface::Light determined
```

2. **Skill boilerplate**

```
depends_interfaces = {
  {v = "motor",type = "MotorInterface", id="Robotino"}
}
```

Use blackboard interfaces

1. **In fawkes-robotino/cfg/conf.d/skiller.yaml**

```
skiller/interfaces/robotino:
  reading:
    motor: MotorInterface::Robotino
  writing:
    light: RobotinoLightInterface::Light determined
```

2. **Skill boilerplate**

```
depends_interfaces = {
  {v = "motor",type = "MotorInterface", id="Robotino"}
}
```

3. **Handle interface object**

```
if motor:has_writer() then
  do_stuff()
end
```

Transforms

```
-- Include tf_module
local tf = require("fawkes.tfutils")

function my_function(self)
   -- Transform (0,0,0) from /base_link to /map
   -- and save it to fsm.vars.startpos
   self.fsm.vars.startpos = tf.transform({x=0, y=0,
         ori=0},"base_link", "map")
   --check if the transform is valid
   if self.fsm.vars.startpos ~= nil then
      return self.fsm.vars.startpos
   else
      print("startpos transform not available!")
      return false
   end
end
```

Use navgraph

```
self.fsm.vars.node = navgraph:node(self.fsm.vars.place)
if self.fsm.vars.node:is_valid() then
  self.fsm.vars.x = self.fsm.vars.node:x()
  self.fsm.vars.y = self.fsm.vars.node:y()
  if fsm.vars.node:has_property("output_offset_y") then
    self.fsm.vars.ori = self.fsm.vars.node:
        property_as_float("output_offset_y");
  end
end
```

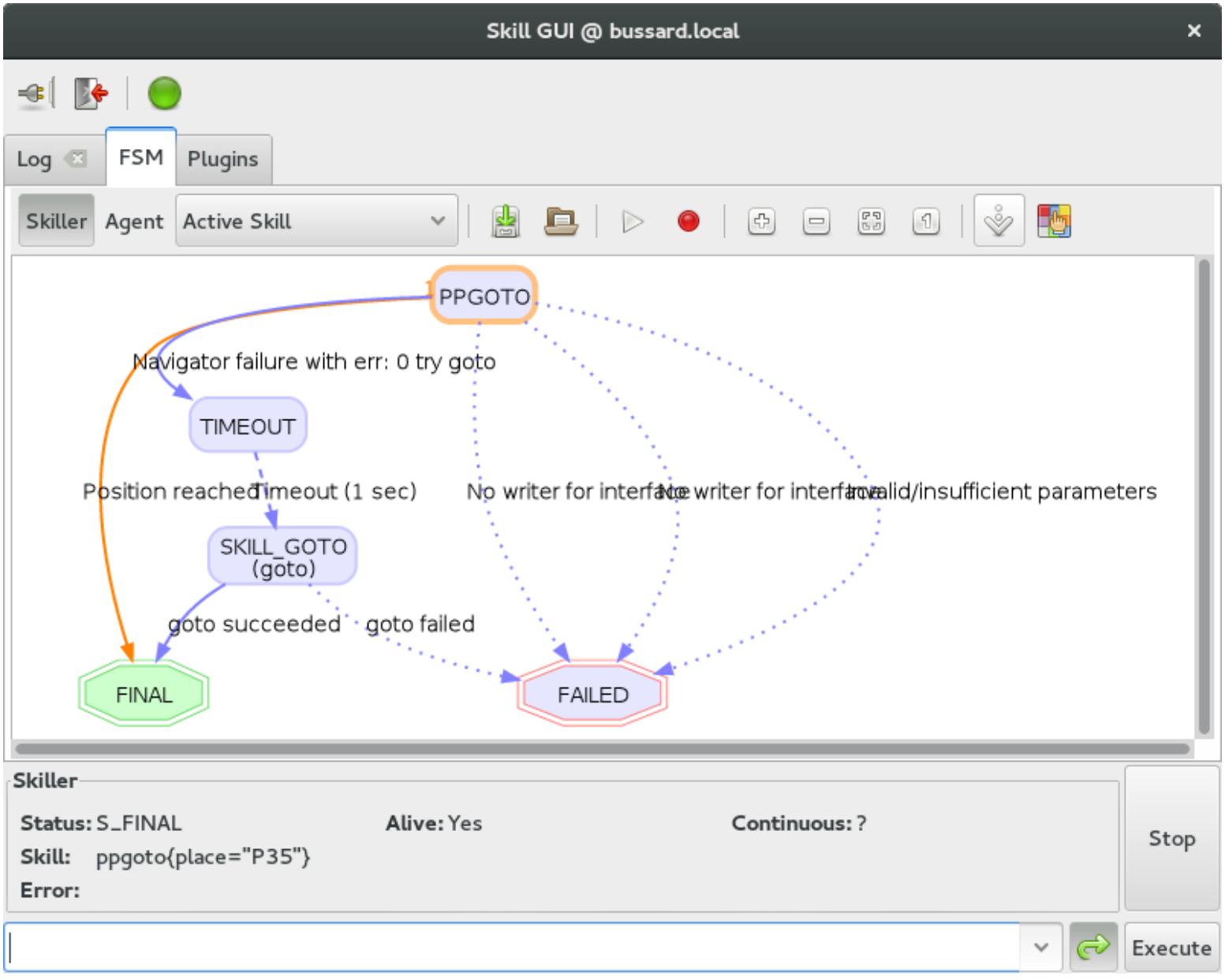
init.lua

`fawkes-robotino/src/lua/skills/robotino/init.lua`

- Initiates the skill space for robotino
- Add the skill you need (without .lua extension)
- Load order reflects the dependencies!

```
skillenv.use_skill("skills.robotino.my_skill")
skillenv.use_skill("skills.robotino.depends_on_skill")
```

SkillGUI and calling skills manually



Conclusion

- Behavior engine and skills
- Hybrid state machines
- Structure of a skill
- In-depth implementation

Separate the agent and the behavior level to separate strategic from local decisions

Conclusion

- Behavior engine and skills
- Hybrid state machines
- Structure of a skill
- In-depth implementation

Separate the agent and the behavior level to separate strategic from local decisions

Questions?

Hands-On

First Task

- Call skills manually via the skillGUI in the simulation
- Try out different skills in the skillspace and learn what they do
- Fetch a product from the cap station and put it onto the belt

Hands-On

First Task

- Call skills manually via the skillGUI in the simulation
- Try out different skills in the skillspace and learn what they do
- Fetch a product from the cap station and put it onto the belt

Second Task:

- Write a ’complex’ skill which calls other skills
  1. Pick a base with a cap from the CS shelf
  2. Instruct the CS to remove the cap
  3. Put the base on the conveyor
  4. Pick up the processed base at the output side
  5. Deliver it at the DS
- Hint: you can use navgraph node names

Hands-On

The skills are located in

`~/robotics/fawkes-robotino/src/lua/skills/robotino`

Lua reference manual

http://www.lua.org/manual/5.1/manual.html

Hands-On Hints

- Access the navgraph:
  1. Start the simulation with the agent and with navgraph generation: `./gazsim.bash -x start -r -n 1 -a -t`
  2. Select the team and start up the setup phase
  3. Unload the plugin `clips-agent` (look at *Plugins* in the SkillGUI)
- Start RVIZ:

```
export ROS_MASTER_URI=http://localhost:11321
rosrun rviz rviz
```

- Send MPS instructions:
  1. Run the simulation as before
  2. Stop the program in the last tab of the simulation (`GazsimLLSFRbCommThread`)
  3. Go to `~/robotics/llsf-refbox/bin`
  4. To instruct `C-CS1` to retrieve a cap:

```
./rcll-prepare-machine Carologistics C-CS1
RETRIEVE_CAP
```