Behavior Architecture

Bas Hickendorff

February 20, 2013

1 Introduction

This document describes the behavior architecture used by the BORG team. The architecture is used to define behavior of the robot. Code to do perception (using the robots sensors) and controlling the robot is not part of this architecture, altough modules for those tasks can be used from the behavior architecture.

2 Structure

The behavior architecture is based on two classes, *Behavior* and *BehaviorImplementation*. A *Behavior* class represents a task we want the robot to perform, like, 'gotoLocation'. For each *Behavior*, there is at least one *BehaviorImplementation* that specifies how the behavior works. There can be more than one implementation of the same behavior. Whenever you want to use a behavior, you should create an instance of the *Behavior* class, instead of creating an instance of one of the implementations of that behavior. Then, at runtime, one of the implementations of that behavior is automatically chosen, based on machine learning.

3 Functions of the Behavior class

A *Behavior* class (not a *BehaviorImplementation* class) implements the *check_postcondition* method. The *check_postcondition* method checks the postcondition, so the postcondition is the same for all the different implementations of a behavior. The abstract *Behavior* class takes care of selecting one of the different implementations of a behavior, based on the results of the machine learning. It also makes sure that the results of the behavior are passed back to the machine learning.

All these classes are automatically generated from a file called *behaviors_config* file, which is the place changes to the behaviors should be made. The specification of this file will be explained later.

4 Conditions

In the system, preconditions, exceptions and postconditions are used. They all have the same format. They are represented as string that, when evaluated in python, returns a boolean.

Usually these conditions will either be using a lookup in the memory, a check on the status of a behavior, or a combination of these two.

An example of a memory lookup is:

```
m.isnow(bluecup.ingripper == True)
```

This checks whether this is currently true. Instead of the *m.isnow* function, you can also use the *m.wasever* function, to check whether something was ever true, according to the memory.

An example of a behavior check is:

```
gotoLocation.finished() == True
```

They can be combined like this:

```
bluecup.ingripper == True and gotoLocation.finished() == True
```

5 Functions of the BehaviorImplementation class

A *BehaviorImplementation* can implement the *implementation_update* method. In this method you specify what the behavior should do. You need to return from this function fast, so you can for example send a command to the *BodyController* to drive forward one meter, but you cannot wait until the robot has actually done that. If you need to wait for the robot to do something, you need to use pre- and postconditions for that (see below). This *implementation_update* method is called several times per second on all running behaviors, so it cannot wait for the robot to do something.

The *behaviorImplementation* class registers whether the behavior is running, finished, or whether it has failed (see below). The *behaviorImplementation* class also has references to the singleton *AllBehaviors* and *BodyController* classes, respectively to get other behaviors and to control the robots.

If you build a behavior you will usually define that behavior in terms of other behaviors. If you want to build upon existing behaviors, you implement the *implementation_init()* and optionally the *implementation_update()* methods. In the *implementation_init()* method you can specify a list of behaviors and preconditions, in the form of *(behavior, precondition)*.

An example *implementation_init()* looks like this:

```
def implementation_init(self):

    #create an object for each behavior we want to use:
    self.gotolocation1 = self.ab.gotoLocation({'location': "kitchen"})
    self.getCup = self.ab.grabObject({'object': "cup"})
    self.gotolocation2 = self.ab.gotoLocation({'location': "living room"})

    #list the behaviors we want to use with their preconditions:
    self.selected_behaviors = [ \
    ( "gotolocation1", "self.redcup.visible == True and self.alreadyGrabbed = F
    ( "getCup", "self.gotolocation1.finished == True" )  , \
    ( "gotolocation2", "self.redcup.ingripper == True" ), \
```

```
]
```

gotolocation1 and *gotolocation2* are different objects, both instantiated from the *gotolocation* behavior, but possibly with different parameters.

Each of these behaviors is run when its corresponding precondition is valid. They are run before the *implementation_update()* function is run. In the *implementation_update()* function it is possible to check whether some of the behaviors in the list started, failed or finished, with the methods *finished_behaviors()*, *running_behaviors()*, *failed_behaviors()*.

In the *implementation_init()* you should also create an object for each behavior you want to use in the behavior. You can use the *AllBehaviors class* for this (see below). The names of these objects are the names you use in the *selected_behaviors* list.

In the *implementation_update()* function it is possible to call functions that are defined in the behavior. This is not possible in the *selected_behaviors* list.

6 The failed state

Each behavior has a 'failed' flag, that is used to indicate that something unexpected has happened, and that the behavior should probably not continue. The decision whether the behavior should stop when it fails is made by the behavior calling the behavior. The failed state is for example used when the behavior takes too long reaching its postcondition.

Other cases where the behavior should fail should be defined in the implementation. There is a *set_failed* function in the *BehaviorImplementation* class that can be used for this.

7 Exceptions

With each behavior, you can specify exceptions. They are the same for each implementation of that behavior. Exceptions are conditions that can happen while the behavior is running and tell the behavior that something has happend that might cause problems if the behavior keeps running. The syntax of exceptions is the same as the syntax of the preconditions and postconditions. The difference between exceptions and precondition is that exceptions are checked while the behavior is running, while the preconditions are checked when the behavior is not running. You can check for exceptions in the *implementation_update()* function.

When an exception happens, the failed-flag is set on the behavior. You can also say that the behavior should stop when an exception happens by calling *self.stop_in_case_of_exception(True)* in the *implementation_init()* function of the behavior. If you do not do this the behavior will keep on running, but still set the failed-flag. You set this option like this:

```
self.getCup.stop_in_case_of_exception(True)
```

In case of a failure because of an exception you can get the reason for the failure from the get_all_exceptions_behaviors() function. This gives you a list of all behaviors that failed, with the corresponding failure reasons.

8 The configuration of behaviors

The list of behaviors available to the robot is defined in the *config/behaviors_config* file. This file defines at least the name and the postcondition of the behaviors.

It is also possible to list exceptions with the behavior. There can be multiple exceptions per behavior.

An example *behaviors_config* file is:

```
[goto_location]
postcondition = "m.isnow(locations == self.location)"
description = "Some optional but recommended informative description."

[grab_object]
postcondition = "closegripper.finished()"
exception = "m.isnow(emptytable == true)"

[walk_and_grab]
postcondition = "walkToLocations.finished()"

[follow_me]
postcondition = "False"
```

It is possible to not have a postcondition, when you have a behavior that should continue forever. For example, in the case of *follow_me*, the robot should just keep following the person, until another behavior tells the *follow_me* behavior to stop. In this case you just define a postcondition *False*

9 The AllBehaviors class

When you need to create a behavior (for example to use in your *selected_behaviors* list), you need to do that via the *AllBehaviors* class. The *AllBehavior* class is imported in each behavior automatically, and is called *ab*. This class will give you a new behavior object that you can use. The *AllBehaviors* class is generated automatically from the *behaviors_config* file and should therefore not be modified. It has a function for each defined behavior to create a new instance of that behavior.

10 Passing parameters to behaviors

Parameters can be passed to a behavior. An example would be the location you want to go to in the *goto* behavior, or which object to grab in a grab behavior. The parameters should be passed as a python dictionary to the function you use to create the behavior (the function in the *AllBehaviors* class).

For example, to create a new behavior that makes the robot navigate to the kitchen, you use:

```
gotoKitchen = self.ab.gotoLocation({'location': 'kitchen'})
```

This gives you a behavior that, when started has a variable self.location that is equal to "kitchen". This variable can be used in the *implementation_update()* function.

These parameters are then accessible in the behavior implementation as datamembers of that behavior (so to access *parameter*, you use *self.parameter*).

The parameters can also be set on a behavior that already exists, by *calling set_params()* on the behavior. You can use the same parameter name if you want to overwrite an existing one, for example:

```
gotoKitchen.setParams({'location': 'hallway'})
```

These parameters can also be used in the conditions, for example:

```
self.location == 'hallway'
```

would be a valid condition that would be true when the *gotoKitchen* behavior has accomplished its goal.

A Example behaviors

A.1 A simple behaviorImplementation

This is an example of a behavior that just drives forward. The command to go forward is sent to the pioneer every update step. This is not a problem, the bodycontroller will make sure that those commands are integrated like expected.

```
import behaviorimplementation
import body.pioneer


class FollowMe_1(behaviorimplementation.BehaviorImplementation):

    def implementation_init(self):
        pass

    def implementation_update(self):

        speed = 20
        turn_speed = 15
        pioneer = self.body.pioneer(0)

        #tell the pioneer to move forward
        pioneer.do(body.pioneer.MOVE_FORWARD, speed)
```

A.2 A behaviorImplementation consisting of other behaviors

This is an example of a behavior that consists of other behaviors. It Grabs an object and brings it to the kitchen.

```
import behaviorimplementation


class GotoLocation_1(behaviorimplementation.BehaviorImplementation):
```

```
def implementation_init(self):

    #all behaviors should be initialized here!
    #we create them once, and run them only if their precondition is valid
    self.grab = self.ab.grabObject({'object': "cup"})
    self.walk_to_kitchen = self.ab.walk({'location': "kitchen"})

    self.grab.stop_in_case_of_exception(True)

    #in the list below, the behaviors should be specified as strings!!
    self.selected_behaviors = [ \
    ( "grab", "m.isnow(cup.holding == true)") , \
    ( "walk_to_kitchen", "self.grab.is_finished()" ),  \
        ]


def implementation_update(self):

    #in this function you can check what behaviors have failed or finished
    #and do possibly other things when something has failed
    pass
```