

IROBOT-Projekt

Einführung in FlexBE – Oktober 2019

Autor: Alvaro Fernández Garcia

Kontakt: fernandezgalvaro@uniovi.es

Datum: 9. Oktober 2019

Das FlexBE-Tool hilft bei der Entwicklung vollständig integrierter Zustandsmaschinen in ROS. Neben der Bereitstellung vieler bereits implementierter Grundzustände (Nachrichten auf dem Bildschirm drucken, eine Nachricht aus einem Thema lesen, eine bestimmte Zeit warten usw.) bietet es auch eine Benutzeroberfläche, die die Erstellung und Überwachung des Automaten erleichtert.

1 Aktionen. Server und Clients Das Aktionskonzept in

ROS ist sehr wichtig bei der Implementierung von Zustandsmaschinen, da es die schnelle Anpassung von C++-Code zur Ausführung in einem in Python programmierten FlexBE-Zustand ermöglicht.

Aktionen sind eine Kommunikationsschnittstelle, die vom *Paket actionlib*[1] implementiert wird. Seine Funktionsweise ähnelt der von ROS-Diensten, diese sind jedoch für kurze Operationen ausgelegt, während Aktionen bessere Werkzeuge für längere und komplexere Ausführungen darstellen.

Wie in Abbildung 1 zu sehen ist, ist das allgemeine Schema dem der Dienstleistungen sehr ähnlich. Die Clientanwendung ruft die Funktion mit bestimmten Parametern auf. Die Ausführung dieser Funktion erfolgt auf dem Server, auf dem die entsprechenden Operationen ausgeführt werden, wobei das entsprechende Ergebnis oder die entsprechenden Ergebnisse an die Client-Anwendung gesendet werden.

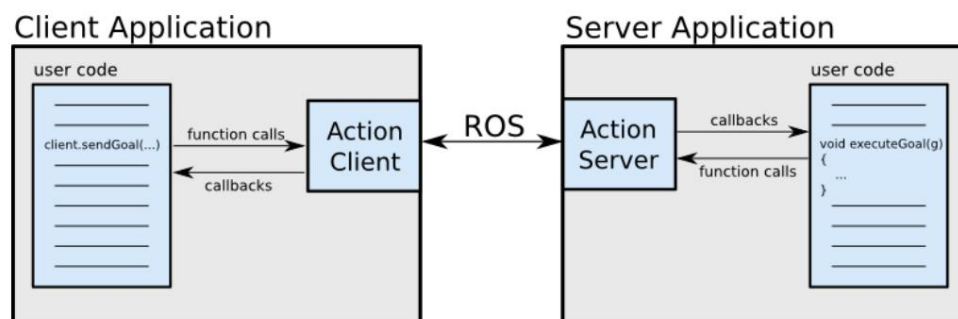
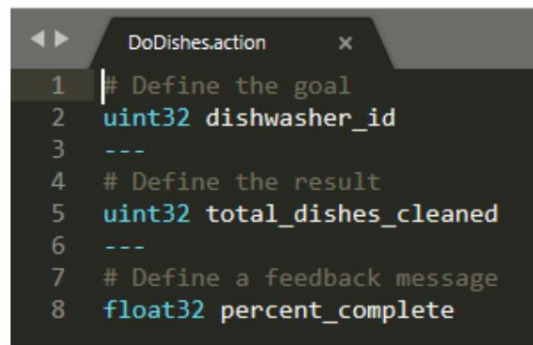


Abbildung 1: Betriebsdiagramm eines Clients und eines Servers, die eine Aktion verwenden. Abbildung aus [1]

Um diese Art der Kommunikation in ROS einzurichten, sind drei Dateien erforderlich: die *.action-Datei*, der Server und der Client.

1.1 .action- Datei Die

Action-Datei ist eine Datei ähnlich wie *.msg* oder *.srv*. Es definiert das *Ziel* oder den Zweck, das *Ergebnis* oder Ergebnis nach Durchführung der Aktion und das *Feedback* oder die Informationen, die vor Abschluss der Aktion an den Kunden gesendet werden können.



```

1 | # Define the goal
2 | uint32 dishwasher_id
3 | ---
4 | # Define the result
5 | uint32 total_dishes_cleaned
6 | ---
7 | # Define a feedback message
8 | float32 percent_complete

```

Abbildung 2: Beispiel einer *.action*-Datei

Ein Beispiel für eine Aktionsdatei ist in Abbildung 2 zu sehen. In diesem Fall ist der Funktionsaufrufparameter die *ID* des Geschirrspülers. Während der Ausführung der Funktion wird der Prozentsatz des gespülten Geschirrs gemeldet. Sobald die Aktion beendet ist, wird die Anzahl des gespülten Geschirrs zurückgegeben.

Die Aktionsfelder müssen in der beschriebenen Reihenfolge deklariert und durch *---* *getrennt werden*. Im Beispiel wird für jedes Feld eine Variable dargestellt, aber sie können aus mehreren Variablen bestehen oder leer bleiben.

Aus diesen einfachen Dateien kann ROS C++-Klassenstrukturen und Nachrichten für jede Aktion generieren. Um mit dem Beispiel von *DoDishes.action* fortzufahren, müssen Sie zum Kompilieren der Aktion vor der Funktion *catkin_package()* in der *CMakeList.txt* des Pakets, in dem sie gefunden wird, Folgendes hinzufügen:

```

find_package(catkin REQUIRED genmsg actionlib_msgs) add_action_files(DIRECTORY
action FILES DoDishes.action) generate_messages(ABHÄNGIGKEITEN actionlib_msgs)

```

Die folgenden Abhängigkeiten müssen auch zur Datei *package.xml* hinzugefügt werden:

```

<build_depend>actionlib_msgs</build_depend>
<exec_depend>actionlib_msgs</exec_depend>

```

1.2 Server

Der Server ist das Programm, das jederzeit parallel laufen muss und die Aktion ausführt, wenn sie von einem anderen Knoten angefordert wird.

Ein Beispiel für ein solches Programm findet sich in Abbildung 3. Wie man sieht, enthält es einen *Callback* (Zeile 6), in dem die Aktion so ausgeführt würde. In der *Hauptsache* wird der Server gestartet und eine Ausführungsschleife wird aufrechterhalten, die darauf wartet, dass ein Anruf getätigt wird

```

1 #include <chores/DoDishesAction.h> // Note: "Action" is appended
2 #include <actionlib/server/simple_action_server.h>
3
4 typedef actionlib::SimpleActionServer<chores::DoDishesAction> Server;
5
6 void execute(const chores::DoDishesGoalConstPtr& goal, Server* as) // Note: "Action" is not appended to DoDishes here
7 {
8     // Do lots of awesome groundbreaking robot stuff here
9     as->setSucceeded();
10 }
11
12 int main(int argc, char** argv)
13 {
14     ros::init(argc, argv, "do_dishes_server");
15     ros::NodeHandle n;
16     Server server(n, "do_dishes", boost::bind(&execute, _1, &server), false);
17     server.start();
18     ros::spin();
19     return 0;
20 }

```

Abbildung 3: Serverseitiges Beispiel der *DoDishes*-Aktion in C++

1.3 Auftraggeber

Abschließend obliegt es dem Auftraggeber, den Auftrag für die auszuführende Aktion zu übermitteln. Da wir es in diesem Fall in einen *FlexBE-Zustand integrieren*, wird ein Beispiel in *Python-Sprache* gezeigt. Sie müssen lediglich einen Client deklarieren und ihm die entsprechenden Parameter übergeben, also das/die Ziel(e), das/die die *Aktion* erreichen soll. In demselben Code wird auf den Abschluss der Aktion gewartet und ihr Ergebnis kann gelesen werden, oder die *Rückmeldung*, wenn sie noch nicht abgeschlossen ist.

Ein Beispiel für einen FlexBE-Zustand, in dem ein Client für Aktionen implementiert ist, finden Sie in Abschnitt 4.

```

1 #!/usr/bin/env python
2
3 import roslib
4 roslib.load_manifest('my_pkg_name')
5 import rospy
6 import actionlib
7
8 from chores.msg import DoDishesAction, DoDishesGoal
9
10 if __name__ == '__main__':
11     rospy.init_node('do_dishes_client')
12     client = actionlib.SimpleActionClient('do_dishes', DoDishesAction)
13     client.wait_for_server()
14
15     goal = DoDishesGoal()
16     # Fill in the goal here
17     client.send_goal(goal)
18     client.wait_for_result(rospy.Duration.from_sec(5.0))

```

Abbildung 4: Client-Beispiel der *DoDishes*-Aktion in Python

Die in diesem Abschnitt behandelten Themen können ausführlicher in der Dokumentation zum Paket *actionlib* [1] nachgelesen werden.

2 Installation

Für die Installation folgen Sie am bequemsten den auf der FlexBE-Seite [2] beschriebenen Schritten. Obwohl mehr Pakete verfügbar sind, die zusätzliche Zustände implementieren, hat die Installation der beiden grundlegendsten Metapakete für die Arbeit, die ich geleistet habe, ausgereicht. Unterhalb dieses Absatzes sehen Sie die darin enthaltenen Ordner.

- flexbe_behavior_engine:

Verhaltensweisen

flexbe_core

flexbe_input

flexbe_mirror
flexbe_states

flexbe_msgs
flexbe_testing

flexbe_onboard
flexbe_widget

- generische_flexbe_states:

flexbe_manipulation_states flexbe_navigation_states flexbe_utility_states

Sobald Sie FlexBE haben, besteht der nächste Schritt darin, FlexBeApp zu installieren, das die grafische Benutzeroberfläche bereitstellt. Anweisungen finden Sie auch in der oben angegebenen Referenz.

Der nächste Schritt kann darin bestehen, einen Ordner zu erstellen, der unseren Code speichert (die implementierten Zustände und die Zustandsmaschine selbst). Navigieren Sie dazu einfach im Terminal zum gewünschten Ordner und führen Sie aus:

```
roslaunch flexbe_widget create_repo Maschinennamen (_Verhalten)
```

Sobald dies geschehen ist, besteht der letzte Schritt darin, die Konfiguration vorzunehmen. Öffnen Sie dazu die FlexBEApp und gehen Sie auf die *Registerkarte Konfiguration*. Dort müssen Sie im Feld *StateLibrary* den Pfad angeben, in dem sich der Statuscode befindet. Der Einfachheit halber habe ich mich entschieden, den Inhalt der Pakete „flexbe_behavior_engine“ und „generic_flexbe_states“ zu extrahieren und in einem neuen Metapaket namens „flexbe_things“ zu *speichern*, Abbildung 5.



Abbildung 5: Inhalt meines Ordners *flexbe_things*.

In das Feld „Arbeitsbereich“ müssen Sie die Ordnerpfade „Verhalten“ und „flexbe_behaviors“ einfügen, die automatisch in unserem persönlichen Ordner erstellt wurden. In Abbildung 6 sehen Sie, wie die Konfigurationsregisterkarte nach diesen Schritten aussieht.

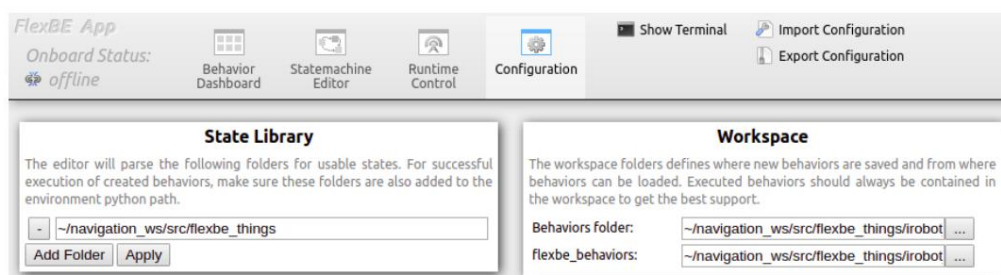


Abbildung 6: Konfiguration der Registerkarte *Konfiguration*.

3 FlexBeApp-Schnittstelle

Im folgenden Abschnitt werden wir versuchen, die Handhabung der FlexBEApp-Schnittstelle einzuführen und versuchen, die wichtigsten Aspekte ihrer Handhabung hervorzuheben.

3.1 Verhaltens-Dashboard

Nach der Installation und Konfiguration können Sie mit der Erstellung des ersten Zustands- oder *Verhaltensautomaten fortfahren*, für den Sie die drei verbleibenden Registerkarten der Anwendung besuchen müssen. Sie beginnen mit der Registerkarte *Verhaltens-Dashboard*.

Der erste Schritt besteht darin, unserem neuen *Verhalten* einen Namen zu geben, wobei eine gültige Nomenklatur für Python-Code, eine Beschreibung, einige Tags und der Name des Autors zu Dokumentationszwecken verwendet werden.

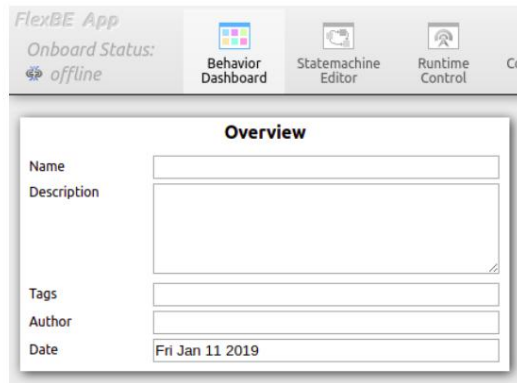


Abbildung 7: Benennung und Dokumentation unserer Zustandsmaschine im *Übersichtsregister*.

Nachdem das *Verhalten beschrieben wurde*, können wir auf der Registerkarte „*Private Konfiguration*“ konstante Werte erstellen, die wir dann als Eingaben an unseren Zustand übergeben können. In *State Machine Userdata* können Variablen deklariert werden, die in diesem Fall in den Zuständen modifiziert werden können und nur einen Default-Wert erhalten.

Private Configuration

Variables enable easy configuration of constant internal values which are used multiple times. They are read-only and cannot be used in private functions.

topic_rviz_goal = "/my_2d_goal"

dist_laserrobot = 1

width_laser = 2

step_path = 4

vel_path = 0.5

inspection_frame = "inspection_frame"

State Machine Userdata

The userdata of a state machine can be used to pass any data from one state to another. Userdata values may be changed by states during runtime. Make sure you define default values for all userdata keys.

rviz_goal_posestamped = []

rviz_goal_pose2d = []

grados = -95.0

pose_1x = Pose2D(1,0,0)

Add

Abbildung 8: Registerkarten „Private Configuration“ und „State Machine Userdata“.

Verwenden:

Wenn Sie eine Variable hinzufügen möchten, die zu einem Paket gehört, z. B. `geometry_msgs::Pose2D` wie in Abbildung 9, müssen Sie zum Python-Code der Zustandsmaschine wechseln, der sich unter befindet:

Ordnername_Verhalten\

Behaviors\Behavior_Name_Machine_States\src\Behavior_Name_Machine_States\Name_Machine_States_sm.py

Und importieren Sie den Nachrichtentyp:

```
21 from flexbe_states.log_key_state import LogKeyState
22 from irobot_flexbe_states.getsimpleinspectionposes_state import GetSimpleInspectionPoses_State
23 from irobot_flexbe_states.moverobotcoord_action_state import MoveRobotCoordActionState
24 # Additional imports can be added inside the following tags
25 # [MANUAL_IMPORT]
26 from geometry_msgs.msg import Pose2D
27 # [/MANUAL_IMPORT]
28
```

Abbildung 9: Aufnahme des Nachrichtentyps `geometry_msgs::Pose2D`

Der letzte Variablentyp sind Parameter. Im Feld „Verhaltensparameter“, Abbildung 10, können Sie Variablen hinzufügen, die vom Bediener auf der Registerkarte „Laufzeitsteuerung“ einfach geändert werden können. Grenzwerte, Standardwerte und andere Aspekte des Parameters können konfiguriert werden, indem auf neben seiner Definition geklickt wird, Abbildung 11.



Behavior Parameters

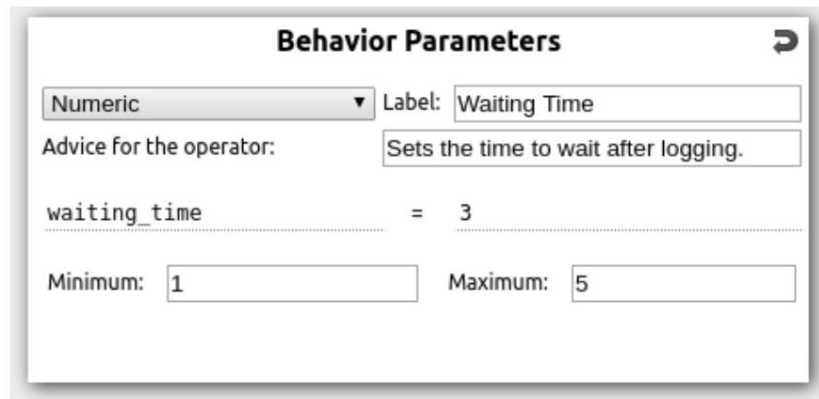
These parameters can be set by the operator when this behavior is started via Runtime Control. Each parameter is identified by a unique variable name and displayed by using a label and providing usage advice. Depending on their type, some parameters may require additional specification. Parameters can be accessed as `self.parameter_name`.

Numeric ▼ waiting_time

Text ▼ nombre_operario

Enum ▼ **Add**

Abbildung 10: Feld Verhaltensparameter



Behavior Parameters

Numeric Label: **Waiting Time**

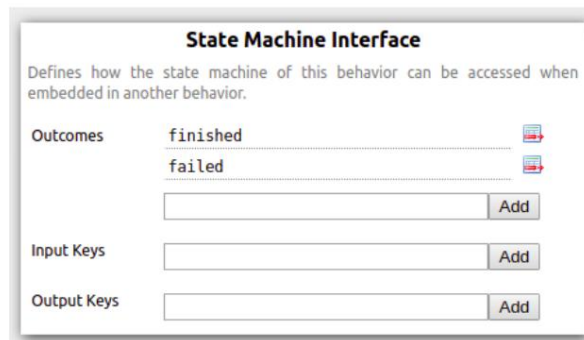
Advice for the operator: **Sets the time to wait after logging.**

waiting_time = **3**

Minimum: **1** Maximum: **5**

Abbildung 11: Setzen des Parameters wait_time

Schließlich finden wir im unteren rechten Teil das *Feld State Machine Interface*. Darin können wir die Ausgänge unserer Zustandsmaschine definieren, die ihre Ausführung beenden. Auch die Definition von Eingangs- und Ausgangsvariablen ist möglich, da die Software die Möglichkeit bietet, Superzustände aus bereits erstellten Automaten zu implementieren.



State Machine Interface

Defines how the state machine of this behavior can be accessed when embedded in another behavior.

Outcomes **finished** **failed**

Input Keys

Output Keys

Add

Abbildung 12: Zustandsmaschinenschnittstelle

State-Machine-Editor

Diese Registerkarte bietet vielleicht die meisten Vorteile im Vergleich zur Erstellung eines Automaten ohne grafische Oberfläche. Auf diesem Bildschirm können wir Zustände in unseren Automaten einfügen, sie über ihre Ausgänge miteinander verbinden und die Übertragung von Daten und Variablen organisieren. Viele grundlegende Funktionalitäten sind bereits in Zuständen implementiert, andere müssen von jedem Benutzer programmiert werden.

Der Editor verfügt über ein Werkzeug zum Kopieren, Ausschneiden und Einfügen sowie Rückgängigmachen und Wiederherstellen, das durch Drücken von **STRG+LEERTASTE** angezeigt werden kann, Abbildung 13.

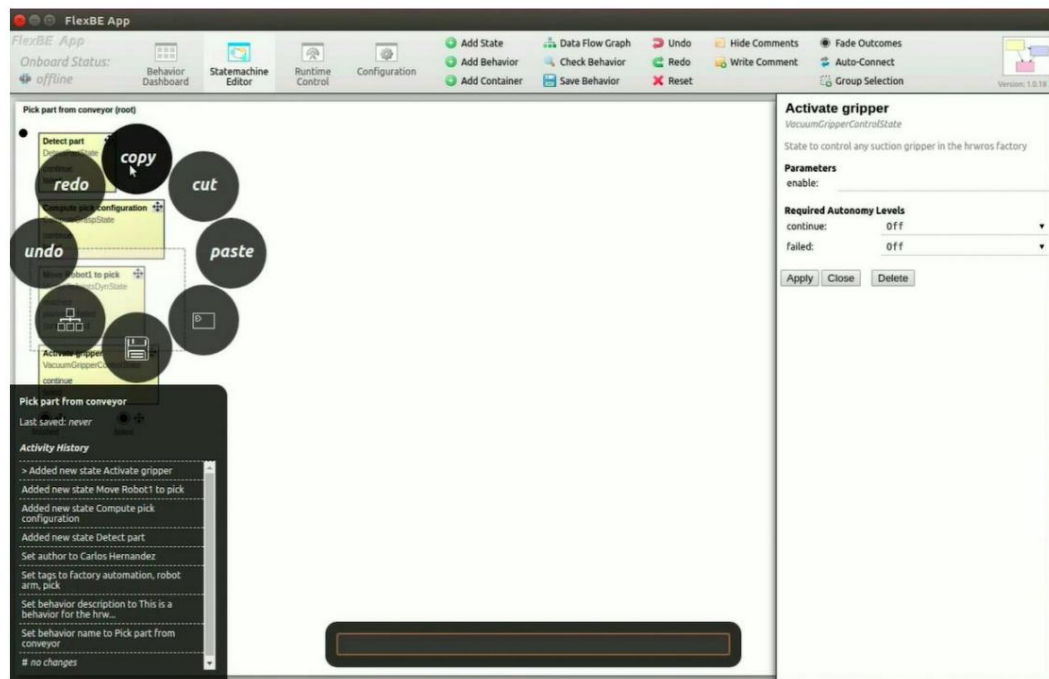


Abbildung 13: Werkzeug, das im Zustandsmaschineneditor angezeigt wird

Verwenden:

Es sei darauf hingewiesen, dass die Zustandsmaschine das Speichern des erzielten Fortschritts nicht zulässt, bis ihre Kompilierung korrekt ist. Aus diesem Grund ist es angebracht, keine großen Fortschritte zu machen, ohne gespart zu haben.

Zusätzlich zu Zuständen können wir auch das hinzufügen, was FlexBE als *Container bezeichnet*. Diese können von drei Arten sein:

- **Zustandsmaschine**

Es funktioniert wie der klassische Superstaat. Darin können Sie Zustände und sogar eine hinzufügen neuer Behälter.

- **Concurrency**

State, der die gleichzeitige Ausführung mehrerer Zustände erlaubt und auf Wunsch nicht aufgegeben wird, bis alle parallel laufenden Zustände beendet sind. Bisher war ihre Verwendung nicht notwendig, aber eine einfache Implementierung wurde getestet und scheint zu funktionieren. Es kann nützlich sein, wenn zwei Aktivitäten gleichzeitig ausgeführt werden, z. B. in unserem Projekt, das Blatt durchzugehen und es zu inspizieren, indem eine Fehlerkarte aufgezeichnet wird. Eine Dokumentation zu dieser Art von Containern findet sich in [3].

- **Priorität**

Spezieller Superzustand, weil er Vorrang vor dem Rest des Automaten genießt. Wenn dies aktiviert ist, stoppt die Ausführung der restlichen Zustände.

Der Editor ermöglicht auch das Hinzufügen von *Verhaltensweisen* und die Funktion als *StateMachine-Container*.

3.2 RunTimeControl

In dieser Registerkarte wird der erstellte Code gestartet und ermöglicht auch die Überwachung der Ausführung und die Interaktion mit den Übergängen zwischen den Zuständen. Dazu reicht es nicht, die normale FlexBEApp zu öffnen, es muss mit dem Befehl geschehen

```
roslaunch flexbe_app flexbe_full.launch
```

und warten Sie, bis es sich mit ROS verbindet, Abbildung 14.

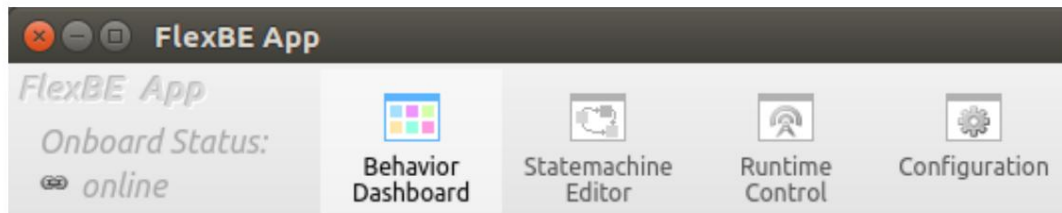


Abbildung 14: FlexBEApp-Schnittstelle, wenn es eine Verbindung mit ROS hergestellt hat.

Im Editor kann zuvor der Grad der Interaktivität seiner Ausgänge für jeden Zustand festgelegt werden, Abbildung 15. Der Autonomiegrad unserer Maschine kann auf dieser Registerkarte „Ausführung“ definiert werden, Abbildung 16. Sobald der Grad in der Ausführung festgelegt wurde müssen alle Transaktionen, deren Ebene darunter liegt, manuell aktiviert werden, damit der Benutzer sie ausführen kann. Die Schnittstelle meldet in jedem Fall, ob der Ausgang durch den Zustand aktiviert worden wäre oder nicht.

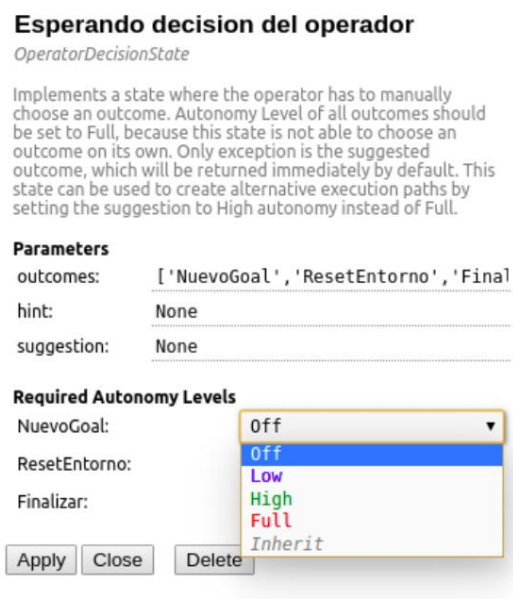


Abbildung 15: Auswahl der Autonomiestufe jedes Ausgangs im StateMachineEditor

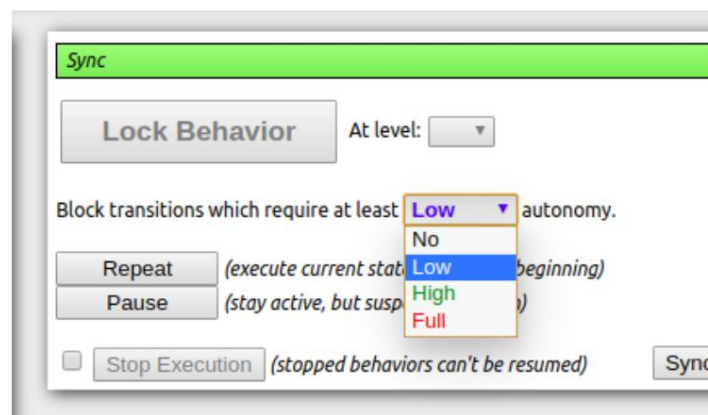


Abbildung 16: Auswahl des minimalen Autonomiegrades, um die Übergänge in der Registerkarte nicht zu blockieren
RunTimeControl

Unabhängig davon kann jeder Übergang jederzeit von dieser Registerkarte aus erzwungen werden. Diese Features ermöglichen es bis zu einem gewissen Grad, eine Benutzerschnittstelle einzurichten. Dies wird von bereits implementierten Zuständen verwendet, wie beispielsweise dem *OperatorDecisionState*, der eine Auswahl zwischen den im Editor definierten Ausgängen ermöglicht, wie in Abbildung 15 gezeigt.

Verwenden:

Fehler im Zustandscode können dazu führen, dass der Automat fehlschlägt, obwohl das Verhalten ohne Probleme kompiliert wurde. In diesem Fall müssen Sie die Anwendung schließen, die .py -Datei ändern und den folgenden Befehl im Codepfad ausführen, bevor Sie FlexBEApp erneut öffnen:

```
chmod +x mi_archivo.py
```

Andernfalls wird die Maschine weiterhin mit der Zustandsversion ausgeführt, mit der sie ausgeführt werden sollte.

Es sollte auch berücksichtigt werden, dass es für das korrekte Funktionieren von Zuständen, die Aktionen verwenden, erforderlich ist, dass Ihr Server vor der Ausführung des Automaten läuft.

4 EventState-Klasse

Schließlich wird es eine kleine Nachverfolgung der Funktionen geben, die in den Klassen vorhanden sind, die die FlexBE-Zustände implementieren, die von EventState erben müssen. Dazu wird die *example_action_state.py* analysiert, die leicht modifiziert automatisch im Ordner `xxxx_flexbe_states` unseres Codes hinzugefügt wird.

Dieses Beispiel sowie *example_state.py* und die in den FlexBE-Paketen enthaltenen Zustände sind gut dokumentiert und stellen eine gute Möglichkeit dar, sich mit der Programmierung dieser Klassen vertraut zu machen.

Weitere Informationen zum Lebenszyklus von FlexBE-Zuständen finden Sie in [4].

Für den korrekten Betrieb müssen die notwendigen Bibliotheken eingebunden werden, wie in Abbildung 17 zu sehen ist. *EventState* und *Logger* sind für den Betrieb jedes FlexBE-Zustands unerlässlich.

Auf der anderen Seite der Kommentar

```
#!/usr/bin/env Python
```

Es ist für die Ausführung von Python-Code in ROS erforderlich.

In der Klassendeklaration wird ihr ein Name gegeben und sie wird dazu gebracht, von EventState zu erben. Es ist dann üblich, den Zustand zu dokumentieren, um ihn in der FlexBEApp zu erkennen und den Code benutzerfreundlicher zu gestalten. Die zur Identifizierung der Eingänge und Ausgänge des Staates verwendete Nomenklatur ist diejenige, die in Tabelle 1 zu sehen ist.

```

1  #!/usr/bin/env python
2  from flexbe_core import EventState, Logger
3  from flexbe_core.proxy import ProxyActionClient
4
5  # example import of required action
6  from chores.msg import DoDishesAction, DoDishesGoal
7
8
9  class ExampleActionState(EventState):
10     """
11     Actionlib actions are the most common basis for state implementations
12     since they provide a non-blocking, high-level interface for robot capabilities.
13     The example is based on the DoDishes-example of actionlib (see http://wiki.ros.org/actionlib).
14     This time we have input and output keys in order to specify the goal and possibly further evaluate the result in a later state.
15
16     -- dishes_to_do int    Expected amount of dishes to be cleaned.
17
18     ># dishwasher int    ID of the dishwasher to be used.
19
20     #> cleaned int        Amount of cleaned dishes.
21
22     <= cleaned_some       Only a few dishes have been cleaned.
23     <= cleaned_enough     Cleaned a lot of dishes.
24     <= command_error      Cannot send the action goal.
25
26     """

```

Abbildung 17: Einbindung von Bibliotheken, Klassendeklaration und deren Dokumentation.

Parameter	--
Eingabevariablen	>#
Ausgangsvariablen	#>
Staatsabgänge	<=

Tabelle 1: Nomenklatur, die in der Dokumentation der Staaten verwendet wird

Als nächstes werden die neu kommentierten Ein- und Ausgänge im Konstruktor `_init_` deklariert. Die Parameter müssen dem Konstruktor als Argument übergeben werden (Zeile 28), während die restlichen Variablen im Superkonstruktor deklariert werden (Zeile 30), Abbildung 18.

```

28  def _init_(self, dishes_to_do):
29      # See example_state.py for basic explanations.
30      super(ExampleActionState, self).__init__(outcomes = ['cleaned_some', 'cleaned_enough', 'command_error'],
31                                              input_keys = ['dishwasher'],
32                                              output_keys = ['cleaned'])
33
34      self.dishes_to_do = dishes_to_do
35
36      # Create the action client when building the behavior.
37      # This will cause the behavior to wait for the client before starting execution
38      # and will trigger a timeout error if it is not available.
39      # Using the proxy client provides asynchronous access to the result and status
40      # and makes sure only one client is used, no matter how often this state is used in a behavior.
41      self._topic = 'do_dishes'
42      self._client = ProxyActionClient({self._topic: DoDishesAction}) # pass required clients as dict (topic: type)
43
44      # It may happen that the action client fails to send the action goal.
45      self._error = False
46

```

Abbildung 18: `_init_-`Funktion in `example_action_state.py`

In dieser `_init_`-Funktion können Attribute definiert werden, die später in den restlichen Methoden verwendet werden können, wie etwa dem Aktionsclient (Zeile 42). Es ist auch gängige Praxis und gute Praxis, eine Variable für Parameter oder Eingabevariablen zu erstellen, falls der ursprüngliche Wert nicht versehentlich geändert werden soll.

Die *Ausführungsfunktion*, Bild 19, wird zyklisch ausgeführt, während der Zustand aktiv ist. Im Beispiel der Abbildung wird überprüft, ob die Aktion abgeschlossen wurde oder einen Fehler ergab. Am Ende ist dies die Methode, die den Ausgängen die entsprechenden Werte zuordnet (Zeile 61) und die Statusausgabe aktiviert (Zeile 65), wodurch die Ausführungsschleife unterbrochen wird.

```

48 def execute(self, userdata):
49     # While this state is active, check if the action has been finished and evaluate the result.
50
51     # Check if the client failed to send the goal.
52     if self._error:
53         return 'command_error'
54
55     # Check if the action has been finished
56     if self._client.has_result(self._topic):
57         result = self._client.get_result(self._topic)
58         dishes_cleaned = result.total_dishes_cleaned
59
60         # In this example, we also provide the amount of cleaned dishes as output key.
61         userdata.cleaned = dishes_cleaned
62
63         # Based on the result, decide which outcome to trigger.
64         if dishes_cleaned > self._dishes_to_do:
65             return 'cleaned_enough'
66         else:
67             return 'cleaned_some'
68
69     # If the action has not yet finished, no outcome will be returned and the state stays active.

```

Abbildung 19: Funktion in example_action_state.py ausführen

Vor der `execute`-Schleife wurde der in `on_enter`, Bild 20, enthaltene Code ausgeführt, allerdings in diesem Fall nur einmal, in diesem Beispiel zum Lesen der Eingabevariablen (Zeile 77) und zum Aufruf der Aktion `progress` davon wird in *Ausführung eingecheckt*.

```

72 def on_enter(self, userdata):
73     # When entering this state, we send the action goal once to let the robot start its work.
74
75     # As documented above, we get the specification of which dishwasher to use as input key.
76     # This enables a previous state to make this decision during runtime and provide the ID as its own output key.
77     dishwasher_id = userdata.dishwasher
78
79     # Create the goal.
80     goal = DoDishesGoal()
81     goal.dishwasher_id = dishwasher_id
82
83     # Send the goal.
84     self._error = False # make sure to reset the error state since a previous state execution might have failed
85     try:
86         self._client.send_goal(self._topic, goal)
87     except Exception as e:
88         # Since a state failure not necessarily causes a behavior failure, it is recommended to only print warnings, not errors.
89         # Using a linebreak before appending the error log enables the operator to collapse details in the GUI.
90         logger.logwarn('Failed to send the DoDishes command:\n%s' % str(e))
91     self._error = True

```

Abbildung 20: on_enter-Funktion in example_action_state.py

Auch die Methode `on_exit` wird nur einmal ausgeführt, nachdem einer der Exits aktiviert und ein neuer Zustand passiert wurde. Im Beispiel, Abbildung 21, wird es verwendet, um sicherzustellen, dass die Aktion nicht mehr ausgeführt wird.

```

94 def on_exit(self, userdata):
95     # Make sure that the action is not running when leaving this state.
96     # A situation where the action would still be active is for example when the operator manually triggers an outcome.
97
98     if not self._client.has_result(self._topic):
99         self._client.cancel(self._topic)
100         logger.loginfo('Cancelled active action goal.')

```

Abbildung 21: Funktion on_exit in example_action_state.py

Schließlich, und obwohl diese nicht in `example_action_state` definiert sind (und daher nicht berücksichtigt werden), ist es wert, die Existenz von `on_start` und `on_stop` zu erwähnen, Abbildung 22. Die erste Methode ermöglicht das Ausführen einer Aktion, wenn die Zustandsmaschine startet, unabhängig davon, ob der Zustand aktiv ist oder nicht. Der zweite hat ein ähnliches Verhalten, aber er beginnt zu arbeiten, wenn der Zustandsautomat seine Ausführung stoppt.

```
def on_start(self):
    # This method is called when the behavior is started.
    # If possible, it is generally better to initialize used resources in the constructor
    # because if anything failed, the behavior would not even be started.

    # In this example, we use this event to set the correct start time.
    self._start_time = rospy.Time.now()

def on_stop(self):
    # This method is called whenever the behavior stops execution, also if it is cancelled.
    # Use this event to clean up things like claimed resources.
```

Abbildung 22: Aus example_state.py extrahierte Funktionen on_start und on_enter

5 Referenzen

- [1] "actionlib - ROS-Wiki". [Online]. Verfügbar unter: <http://wiki.ros.org/actionlib>. [Zugriff: 11. Jan 2019].
- [2] «FlexBE Verhaltensmaschine». Linie]. [In Verfügbar In: unter <http://philserver.bplaced.net/fbe/download.php#states>. [Zugriff: 14. Januar 2019].
- [3] „flexbe/Tutorials/Parallel State Execution – ROS Wiki“. [Online]. Verfügbar unter: <http://wiki.ros.org/flexbe/Tutorials/Parallel%20State%20Execution>. [Zugriff: 16. Januar 2019]. [4] "flexbe/Tutorials/The State Lifecycle - ROS Wiki". [Online]. Verfügbar unter: <http://wiki.ros.org/flexbe/Tutorials/The%20State%20Lifecycle>. [Zugriff: 15. Januar 2019].