gabrieltool Documentation

Release 1.1.0

Junjue Wang

May 07, 2020

Contents

OpenWorkflow is a suite of development tools to facilitate the creation and implementation of Wearable Cognitive Assistants (WCA). They key idea of OpenWorkflow is to represent WCA application logic as a finite state machine (FSM). You can read more about the fast prototyping methodology of OpenWorkflow in this document (Section 6.2).

OpenWorkflow provides the following tools.

- gabrieltool: A Python library to create and persist finite state machine based wearable cognitive assistants.
- OpenWorkflow State Machine Web Editor: A browser-based GUI to view and edit state machines.
- gabrieltool CLI (gbt): A command-line tool to launch a gabriel server given an FSM.

CHAPTER 1

User Guide

1.1 Installation

1.1.1 gabrieltool Python Library

First, make sure Docker is installed. You can follow the Docker installation guide.

Option 1. From PyPI (recommended)

```
$ pip install -U gabrieltool
```

Option 2. From Source

```
$ git clone https://github.com/cmusatyalab/OpenWorkflow.git
$ cd OpenWorkflow/
$ python setup.py install
```

1.1.2 OpenWorkflow State Machine Web Editor

Visit https://cmusatyalab.github.io/OpenWorkflow. Or You can download the build from the repo's gh-pages branch and open the index.html.

1.2 Quickstart

1.2.1 Python Library gabrieltool

Create a Two State FSM

```python
from gabrieltool.statemachine import fsm, predicate_zoo, processor_zoo

# create a two state state machine
st_start = fsm.State(
    name='start',
    processors=[fsm.Processor(
        name='proc_start',
        callable_obj=processor_zoo.DummyCallable()
    )],
    transitions=[
        fsm.Transition(
            name='tran_start_to_end',
            predicates=[
                fsm.TransitionPredicate(
                    callable_obj=predicate_zoo.Always()
                )
            ]
        )
    ]
)
st_end = fsm.State(
    name='end'
)
st_start.transitions[0].next_state = st_end
```

Save the FSM to a file

```python
# save to disk
with open('simple.pbfsm', 'wb') as f:
    f.write(fsm.StateMachine.to_bytes(
        name='simple_fsm',
        start_state=st_start
    ))
```

Launch a gabriel server using the FSM.

```
$ gbt run ./simple.pbfsm
```

See *Tutorial* for a detailed example.

1.2.2 OpenWorkflow State Machine Editor

The editor is hosted at https://cmusatyalab.github.io/OpenWorkflow/. An instruction video is available here.

This web editor provides the following functionalities.

1. Import to view a saved FSM file. This FSM file can be created either from *gabrieltool* or from the web editor.
2. Export a FSM to a file.
3. Edit FSM states and transitions. Note that only supported operations from processor_zoo and predicate_zoo can be edited. Custom defined functions can not be created or modified in the web editor.

Compared to the gabrieltool Python library, the web editor provides better visualization and is great for creating small FSMs. For more complicated FSMs, consider using the gabrieltool for better reproducibility and efficiency.

1.2.3 Gabrieltool CLI (gbt)

The gabrieltool cli (gbt) provides a convenient method to launch a gabriel server given an FSM, created by the python library or the web editor.

```
$ gbt run <path-to-fsm>
$ # for usage details, see gbt -h
```

1.3 Tutorial

We will create a wearable cognitive assistant that recognize a person or a chair in this tutorial. First, let's get the example code running before going into its implementation.

1. Install gabrieltool.
2. Download gabriel_example.py and the object detector into the same directory. This object detector is the SSD MobileNet v2 DNN from Tensorflow Object Detection API model zoo. In this example, we will use this object detector to detect people and chairs.
3. Unzip the downloaded object detector into the same directory.

```
$ unzip ssd_mobilenet_v2_saved_model.zip
```

4. Launch the gabriel server.

```
$ ./gabriel_example run_gabriel_server
```

5. In the console, you should see log messages of building the FSM, starting gabriel server, and then launching a docker container.
6. You should also be able to see the container started using docker commands. Note that it may take a few minutes to download the container image before the container is started.

```
$ docker ps -a --filter="name=GABRIELTOOL"
```

7. Once you see the container is up, the server is ready for connection. Download Gabriel client from Android Play Store to connect to it and try it out. When the client has a person or a chair in view, the application should say "Found Person" or "Found chair" correspondingly.
8. Once you're done with the demo, make sure to clean up the created docker container with the following commands.

```
$ docker stop -t 0 $(docker ps -a -q --filter="name=GABRIELTOOL")
```

Now you've gotten the code running, let see what is happening under the hood. We will focus on explaining how to create a gabriel server in this tutorial while the example code contains a few more use cases of gabrieltool package.

You can create the same FSM using either the gabrieltool python library or the web GUI. In general, the web GUI is good for simple applications while the python library provides more flexibility and supports more customizable application logic. Below we will create the same application using both methods.

1.3.1 Using Python Library

The FSM has two states. The first state is *st_start*. We want to send a welcome message when a user first connects. Therefore, *st_start* doesn't have any processing involved and will always transition immediately to the next state and return a welcome message to the client.

The second state *st_tf* is the core of this application. When in this state, input sensor data, which is an image in this example, is analyzed by our object detector to see if there is a person or a chair. This is specified by a fsm.Processor with a processor_zoo.TFServingContainerCallable. Since we want to recognize either a person or a chair, we define two transitions, one for person, another for chair. These transitions have predicates checking whether the person or the chair object class exist in the output of our TFServingContainerCallable processor. If a person is found, the person transition will be taken and return an instruction of ‘Found Person’ to the Gabriel client.

```python
import cv2
import fire
from logzero import logger
from gabriel_server.local_engine import runner as gabriel_runner
from gabrieltool.statemachine import fsm, predicate_zoo, processor_zoo, runner


def _build_fsm():
    """Build an example FSM for detecting a person or a chair.

    Returns:
        gabrieltool.statemchine.fsm.State -- The start state of the generated FSM.
    """
    st_start = fsm.State(
        name='start',
        processors=[],
        transitions=[
            fsm.Transition(
                name='tran_start_to_proc',
                predicates=[
                    fsm.TransitionPredicate(
                        callable_obj=predicate_zoo.Always()
                    )
                ],
                instruction=fsm.Instruction(audio='This app will tell you if a␣
↪person or a chair is present.')
            )
        ]
    )

    st_tf = fsm.State(
        name='tf_serving',
        processors=[fsm.Processor(
            name='proc_start',
            callable_obj=processor_zoo.TFServingContainerCallable('ssd_mobilenet_v2',
                                                                   'ssd_mobilenet_v2_
↪saved_model',
                                                                   conf_threshold=0.8
                                                                   )
        )],
        transitions=[
            fsm.Transition(
                name='tf_serving_to_tf_serving_person',
                predicates=[
                    fsm.TransitionPredicate(
                        # person id is 1 in coco labelmap
                        callable_obj=predicate_zoo.HasObjectClass(class_name='1')
                    )
                ],
                instruction=fsm.Instruction(audio='Found Person!')
            ),
```

(continues on next page)

(continued from previous page)

```
            fsm.Transition(
                name='tf_serving_to_tf_serving_chair',
                predicates=[
                    fsm.TransitionPredicate(
                        # You can also use the custom transition predicate we
                        # created in _add_custom_transition_predicate here. e.g.
                        # callable_obj=predicate_zoo.HasChairClass()
                        callable_obj=predicate_zoo.HasObjectClass(class_name='62')
                    )
                ],
                instruction=fsm.Instruction(audio='Found Chair!')
            )
        ]
    )

    # We need the state objects to mark the destinations of transitions
    st_start.transitions[0].next_state = st_tf
    st_tf.transitions[0].next_state = st_tf
    st_tf.transitions[1].next_state = st_tf
    return st_start
```

The *st_tf* state uses a custom transition predicate defined by the following function. To learn more about the how to use and create FSM components, see its API documentation.

```
def _add_custom_transition_predicates():
    """Here is how you can add a custom transition predicate to the predicate zoo

    See _build_fsm to see how this custom transition predicate is used
    """

    from gabrieltool.statemachine import callable_zoo

    class HasChairClass(callable_zoo.CallableBase):
        def __call__(self, app_state):
            # id 62 is chair
            return '62' in app_state

    predicate_zoo.HasChairClass = HasChairClass
```

The gabriel cognitive engine is created using a FSM cognitive engine runner.

```
def run_gabriel_server():
    """Create and execute a gabriel server for detecting people.

    This gabriel server uses a gabrieltool.statemachine.fsm to represents
    application logic. Use Gabriel Client to stream images and receive feedback.
    """
    logger.info('Building Person Detection FSM...')
    start_state = _build_fsm()
    logger.info('Initializing Cognitive Engine...')
    # engine_name has to be 'instruction' to work with
    # gabriel client from App Store. Someone working on Gabriel needs to fix this.
    engine_name = 'instruction'
    logger.info('Launching Gabriel server...')
    gabriel_runner.run(
        engine_setup=lambda: runner.BasicCognitiveEngineRunner(
```

(continues on next page)

(continued from previous page)

```
                engine_name=engine_name, fsm=start_state),
            engine_name=engine_name,
            input_queue_maxsize=60,
            port=9099,
            num_tokens=1
        )
```

Gabrieltool currently doesn't support cleaning up the launched containers automatically. You can stop and remove all gabrieltool related containers using the following command.

```
$ docker stop -t 0 $(docker ps -a -q --filter="name=GABRIELTOOL")
```

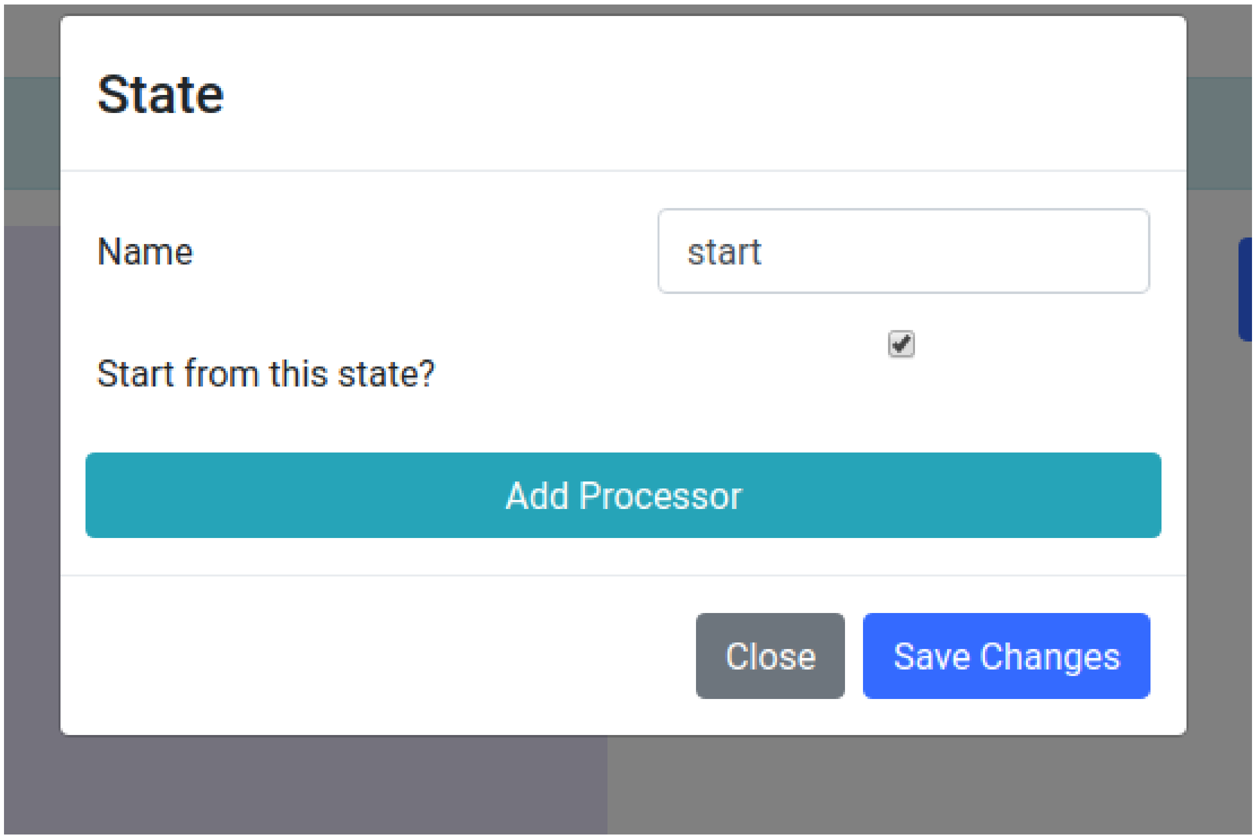
1.3.2 Using OpenWorkflow State Machine Web Editor

Let's use the Web Editor to create the same application.

First, let's create states. Go to Add >> State.

start state:

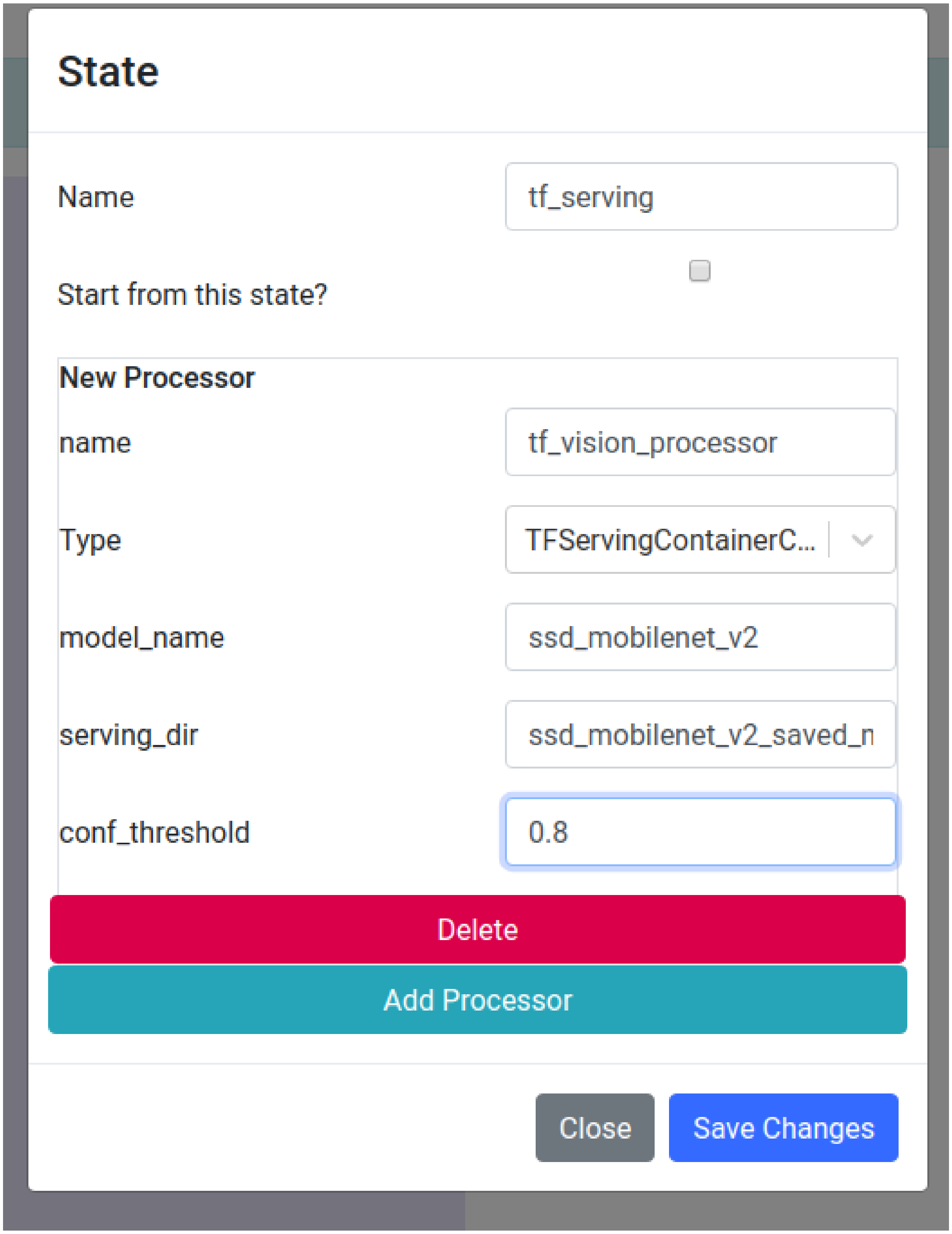
- name: "start"
- Check Start from this state.



tf_serving state:

- name: "tf_serving"
- add a new processor

- name: “tf_vision_processor”
- type: “TFServingContainerCallable”
- model_name: “ssd_mobilenet_v2”
- serving_dir: “ssd_mobilenet_v2_saved_model”. This is directory of the downloaded and unzipped model.
- conf_threshold: 0.8


State

Name: tf_serving

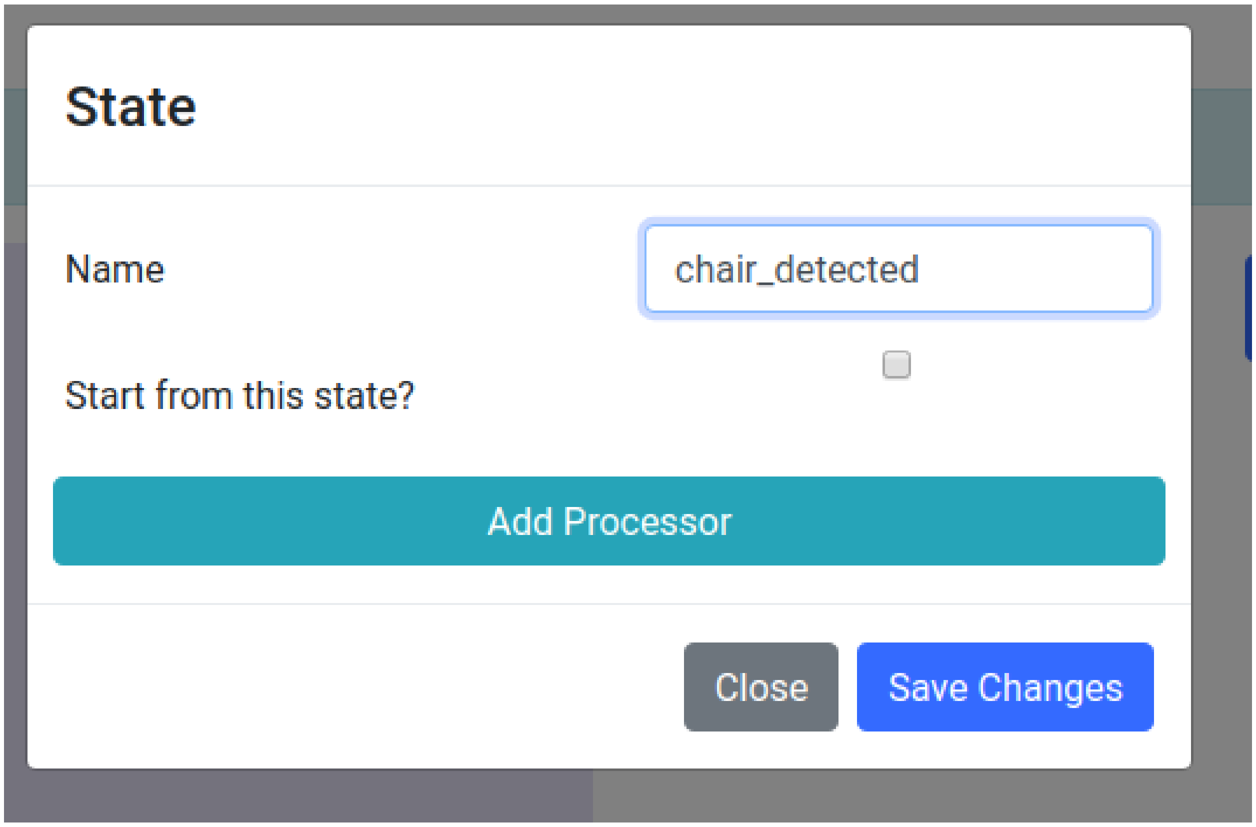
Start from this state? ☐

New Processor

name	tf_vision_processor
Type	TFServingContainerC...
model_name	ssd_mobilenet_v2
serving_dir	ssd_mobilenet_v2_saved_n
conf_threshold	0.8

chair_detected state:

- name: "chair_detected"

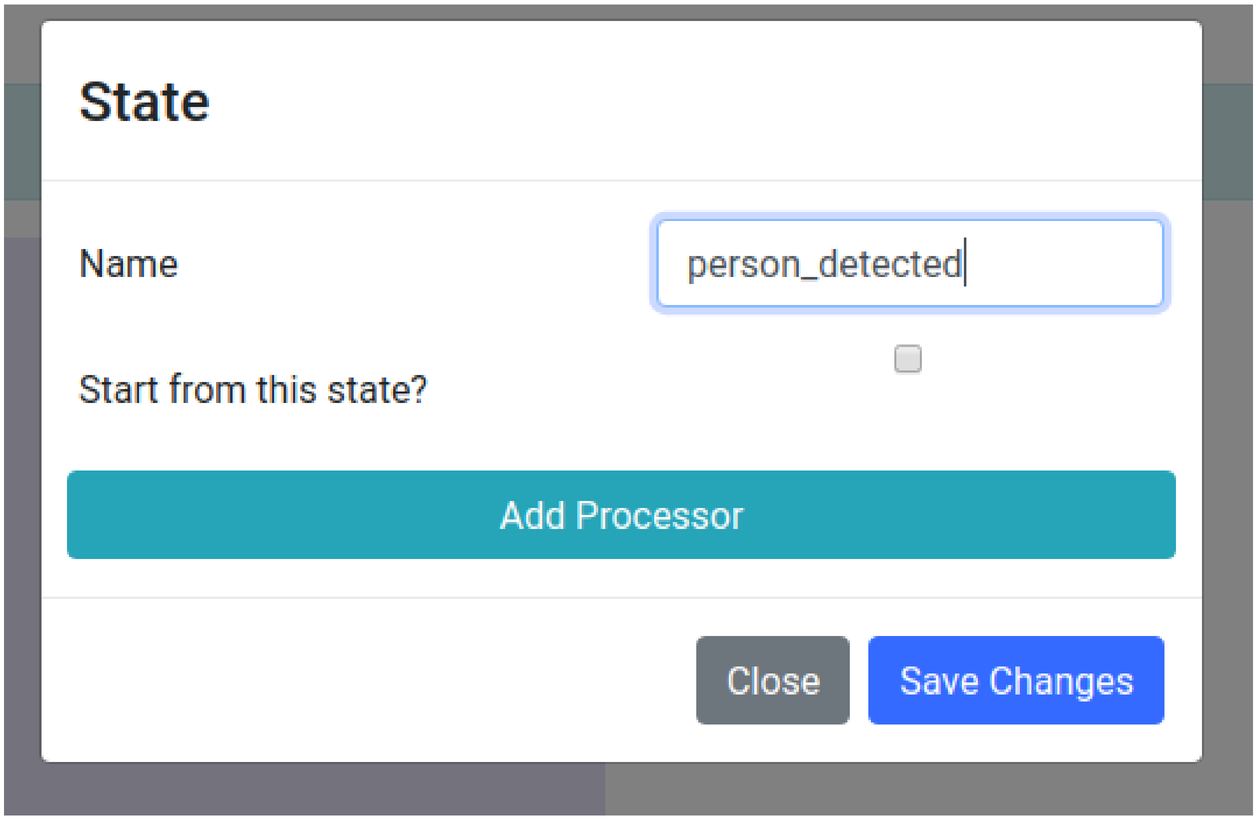

State

Name: chair_detected

Start from this state? ☐

person_detected state:

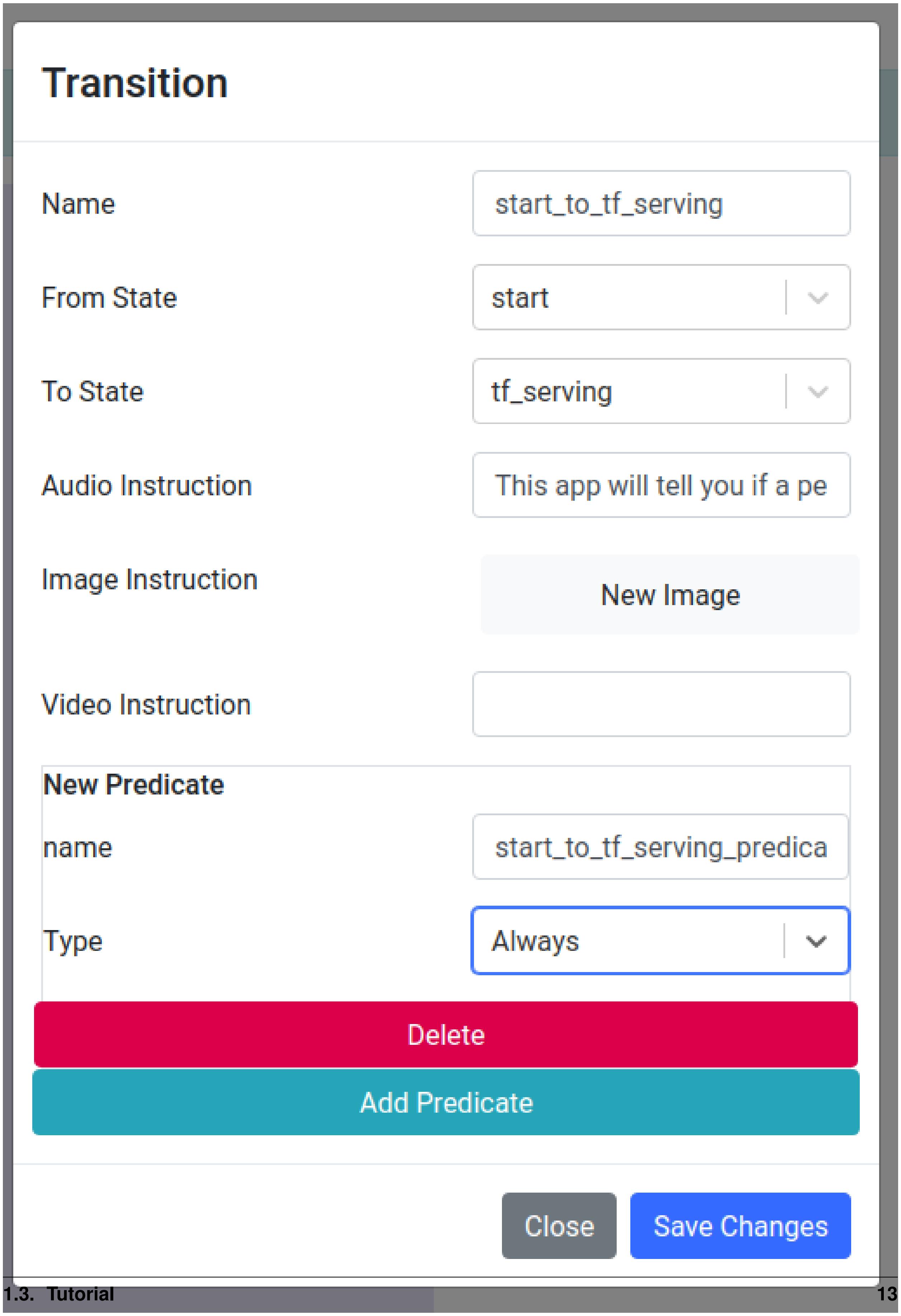
- name: "person_detected"



Now, let's add transitions. Go to Add >> Transition

start to tf_serving

- name: "start_to_tf_serving"
- From State: start
- To State: tf_serving
- Audio Instruction: "This app will tell you if a person or a chair is present."
- Add Predicate
  - name: "start_to_tf_serving_predicate"
  - type: Always



Transition

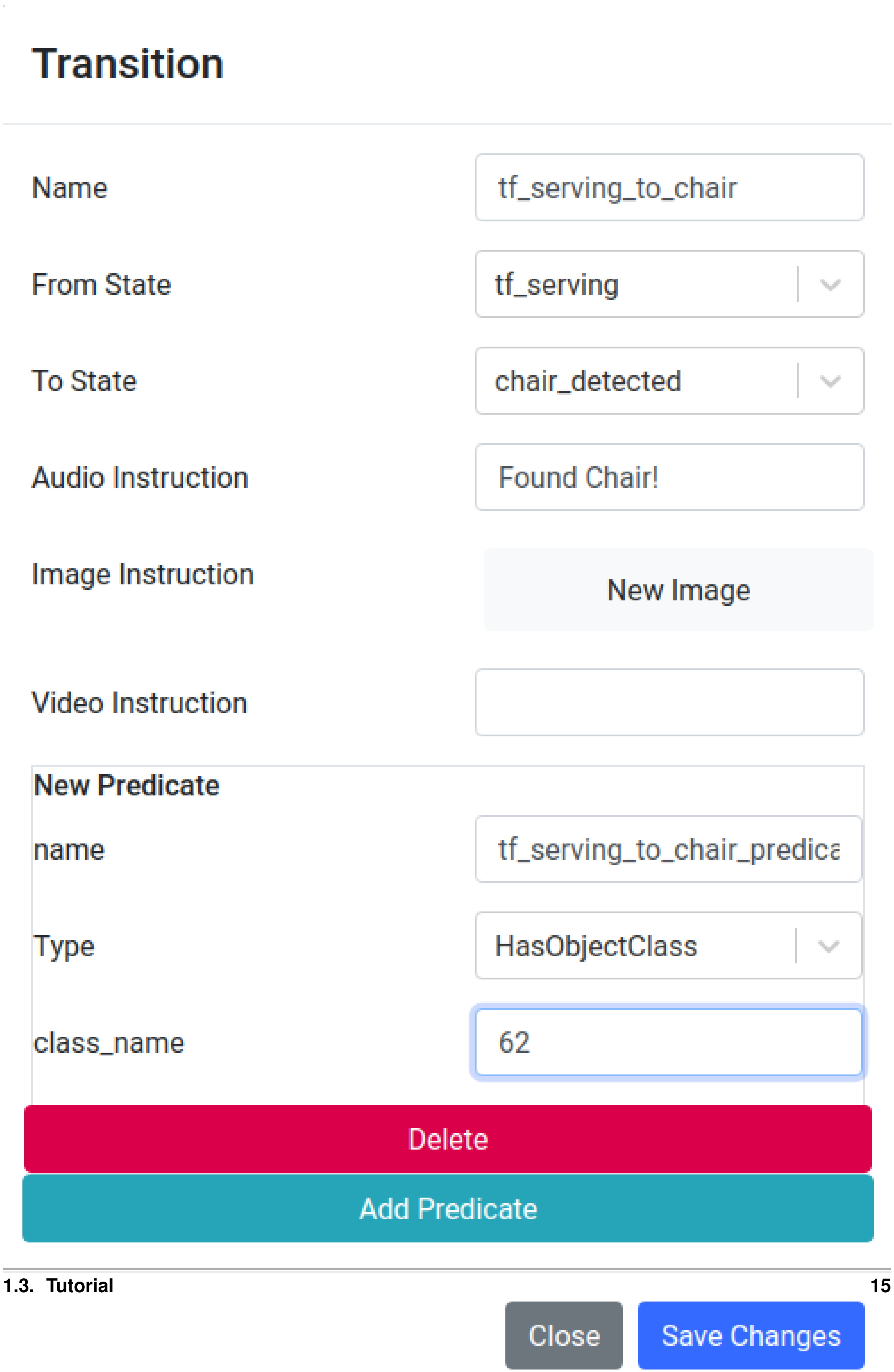
Name	start_to_tf_serving
From State	start
To State	tf_serving
Audio Instruction	This app will tell you if a pe
Image Instruction	New Image
Video Instruction	

New Predicate

name	start_to_tf_serving_predica
Type	Always

tf_serving to chair_detected

- name: "tf_serving_to_chair"
- From State: tf_serving
- To State: chair_detected
- Audio Instruction: "Found Chair!"
- Add Predicate
  - name: "tf_serving_to_chair_predicate"
  - type: HasObjectClass
  - class_name: 62



Transition

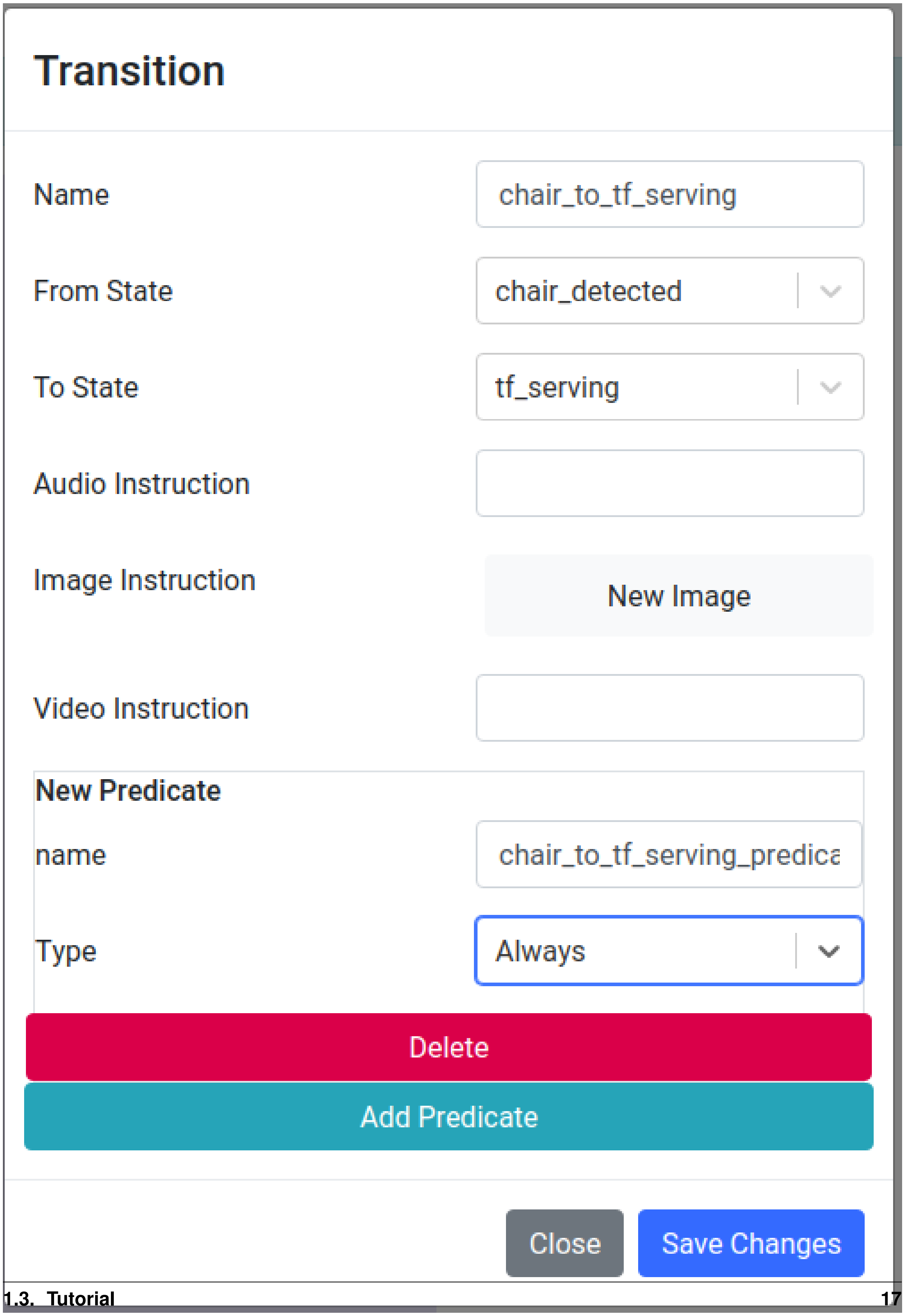
Name	tf_serving_to_chair
From State	tf_serving
To State	chair_detected
Audio Instruction	Found Chair!
Image Instruction	New Image
Video Instruction	

New Predicate

name	tf_serving_to_chair_predica
Type	HasObjectClass
class_name	62

chair_detected back to tf_serving

- name: “chair_to_tf_serving”
- From State: chair_detected
- To State: tf_serving
- Add Predicate
  - name: “chair_to_tf_serving_predicate”
  - type: Always



Transition

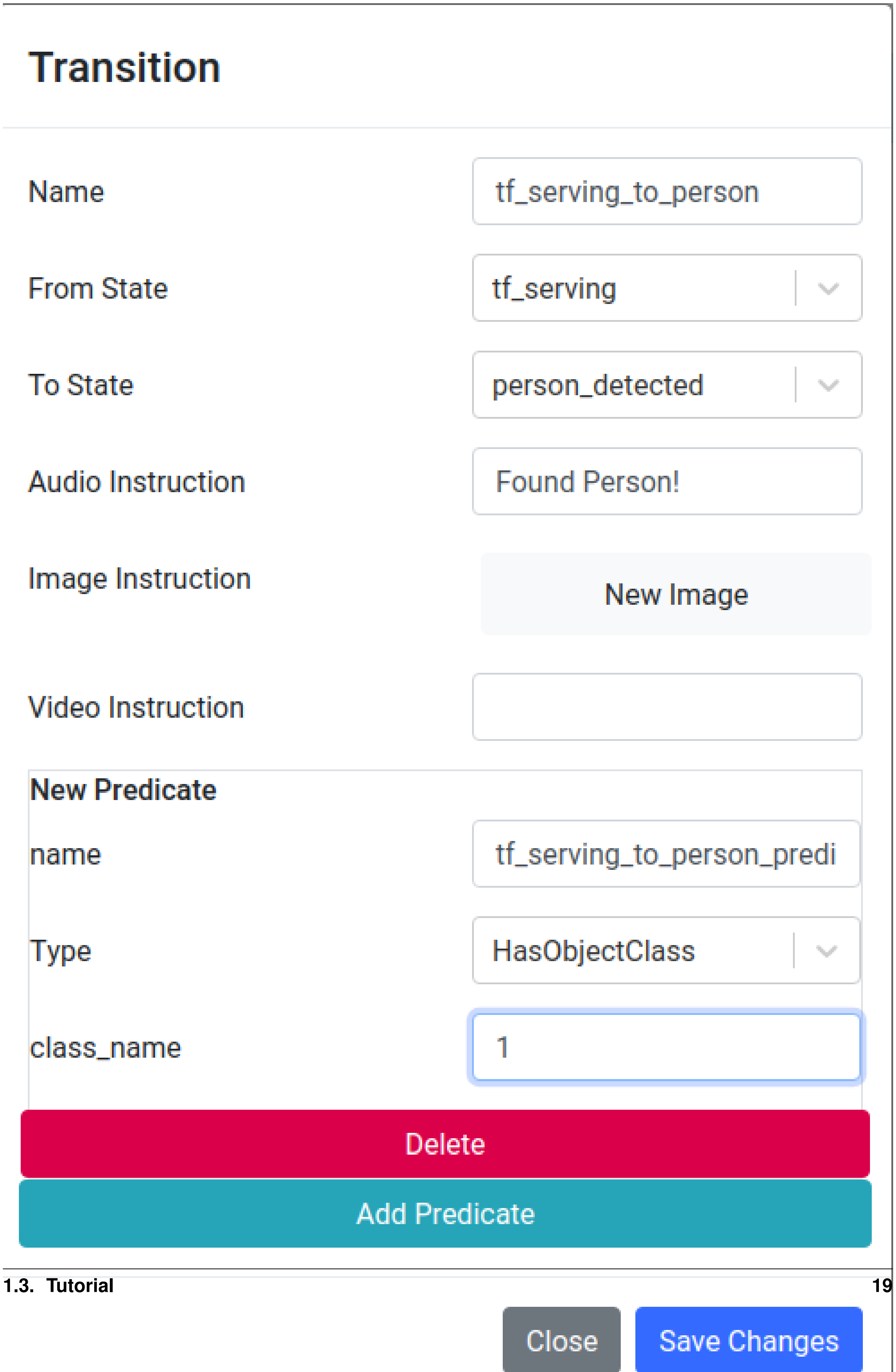
Name	chair_to_tf_serving
From State	chair_detected
To State	tf_serving
Audio Instruction	
Image Instruction	New Image
Video Instruction	

New Predicate

name	chair_to_tf_serving_predica
Type	Always

tf_serving to person_detected

- name: “tf_serving_to_person”
- From State: tf_serving
- To State: person_detected
- Audio Instruction: “Found Person!”
- Add Predicate
  - name: “tf_serving_to_chair_predicate”
  - type: HasObjectClass
  - class_name: 1



Transition

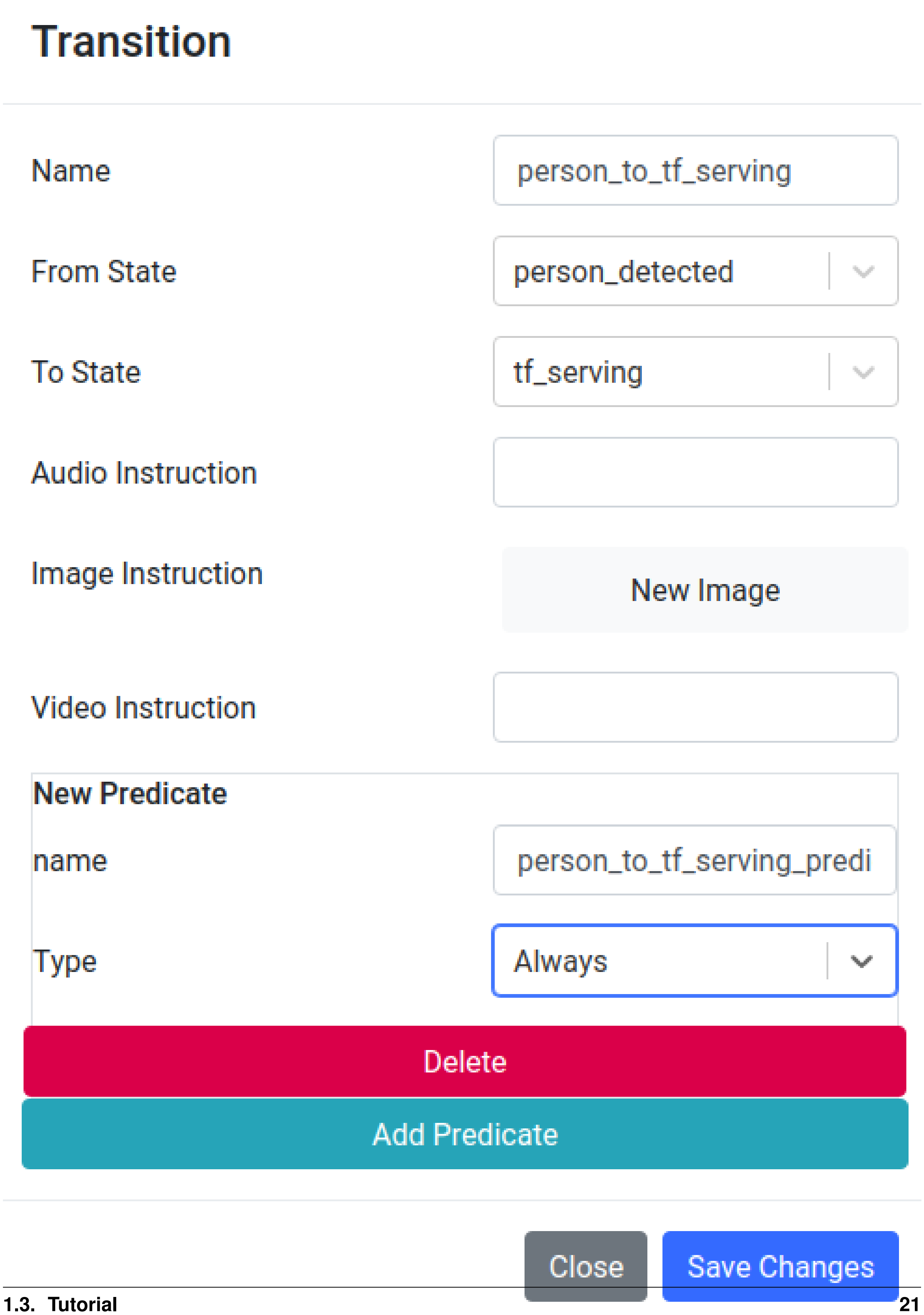
Name	tf_serving_to_person
From State	tf_serving
To State	person_detected
Audio Instruction	Found Person!
Image Instruction	New Image
Video Instruction	

New Predicate

name	tf_serving_to_person_predi
Type	HasObjectClass
class_name	1

person_detected back to tf_serving

- name: “person_to_tf_serving”
- From State: person_detected
- To State: tf_serving
- Add Predicate
  - name: “person_to_tf_serving_predicate”
  - type: Always



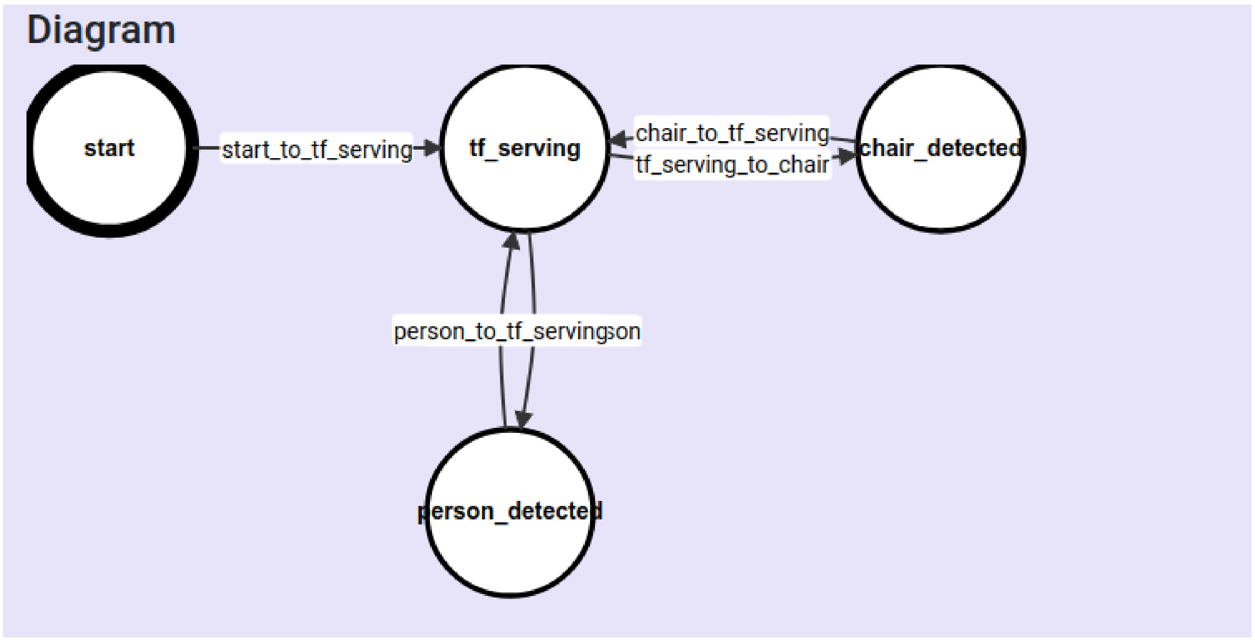
Transition

Name	person_to_tf_serving
From State	person_detected
To State	tf_serving
Audio Instruction	
Image Instruction	New Image
Video Instruction	

New Predicate

name	person_to_tf_serving_predi
Type	Always

By now, we have finished creating the FSM. The complete FSM looks as follows.



Let's export the FSM to the same directory of our downloaded object detectors. The directory structure should look like the following.

Gabrieltool CLI provides a convenient command-line tool to launch a gabriel server with the exported FSM. Connect Gabriel Client to your server. When you point the client to a person or a chair, the application should say "Found Person" or "Found chair" correspondingly.

```
$ gbt run ./app.pbfsm
```

Once you're done with the demo, make sure to clean up the created docker container with the following commands.

```
$ docker stop -t 0 $(docker ps -a -q --filter="name=GABRIELTOOL")
```

1.4 gabrieltool API

1.4.1 gabrieltool package

Subpackages

gabrieltool.statemachine package

Subpackages

gabrieltool.statemachine.callable_zoo package

Subpackages

gabrieltool.statemachine.callable_zoo.predicate_zoo package

Submodules

gabrieltool.statemachine.callable_zoo.predicate_zoo.base module

Callable classes for Transition Predicates.

All the classes here should be a callable and return either True/False when called (to indicate whether or not to take a transition). All classes should inherit from CallableBase class and annoate their constructor (if there is one) with the @record_kwargs decorator for proper serialization.

class Always

Bases: *gabrieltool.statemachine.callable_zoo.base.CallableBase*

Always take this transition.

Useful for welcome message when the application starts.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

class HasObjectClass(*class_name*)

Bases: *gabrieltool.statemachine.callable_zoo.base.CallableBase*

Check if there is an object class in the extracted information of the sensor data.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

class HasObjectClassWhileNotOthers(*has_classes=None*, *absent_classes=None*)

Bases: *gabrieltool.statemachine.callable_zoo.base.CallableBase*

Check if there are some object classes in the extracted information while some other classes are not.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

class Wait(*wait_time=None*)

Bases: *gabrieltool.statemachine.callable_zoo.base.CallableBase*

Wait for some time before turning true.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

Module contents

gabrieltool.statemachine.callable_zoo.processor_zoo package

Submodules

gabrieltool.statemachine.callable_zoo.processor_zoo.base module

Basic callable classes for Processor.

class DummyCallable(*dummy_input='dummy_input_value'*)

Bases: `gabrieltool.statemachine.callable_zoo.base.CallableBase`

A Dummy Callable class for testing and examples.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

class FasterRCNNOpenCVCallable(*proto_path*, *model_path*, *labels=None*, *conf_threshold=0.8*)

Bases: `gabrieltool.statemachine.callable_zoo.base.CallableBase`

A callable class that executes a FasterRCNN object detection model using OpenCV.

classmethod from_json(*json_obj*)

Create an object from a JSON object.

Parameters **json_obj** (*json*) – JSON object that has all the serialized constructor arguments.

Raises `ValueError` – when constructor arguments' type don't match.

Returns The deserialized FasterRCNNOpenCVCallable object.

Return type *FasterRCNNOpenCVCallable*

visualize_detections(*img*, *results*)

Visualize object detection outputs.

This is a helper function for debugging processor callables. The results should follow Gabrieltool's convention, which is

Parameters

- **{OpenCV Image}** (*img*) –
- **{Dictionary} -- a dictionary of class_idx -> [[x1, y1, x2, y2, confidence, cls_idx],..]** (*results*) –

Returns OpenCV Image – Image with detected objects annotated

gabrieltool.statemachine.callable_zoo.processor_zoo.containerized module

Callable classes for Containerized Processors.

Currently we don't offer functionalities to clean up the containers after the program finishes. Use the following commands to clean up the containers started by this module.

$ docker stop -t 0 $(docker ps -a -q –filter="name=GABRIELTOOL")

class FasterRCNNContainerCallable(*container_image_url*, *conf_threshold=0.5*)

Bases: `gabrieltool.statemachine.callable_zoo.base.CallableBase`

A callable class to execute containerized FasterRCNN model in Caffe.

Use this class if your object detector is generated by TPOD v1 and the container image is hosted by cmusatyalab's gitlab container registry.

CONTAINER_NAME = 'GABRIELTOOL-FasterRCNNContainerCallable-129'

clean()

container_server_url

classmethod from_json(*json_obj*)
Deserialize.

class SingletonContainerManager(*container_name*)
Bases: `object`

Helper class to start, get, and remove a container identified by a name.

clean()
Remove the container if it exists.

container

container_name

start_container(*image_url*, *command*, ***kwargs*)
Start a container

Parameters

- **image_url** (*string*) – Container Image URL.
- **command** (*string*) – Container command.
- **kwargs** (*dictionary*) – Extra arguments to pass to Docker client.

Returns A container

Return type Container

class TFServingContainerCallable(*model_name*, *serving_dir*, *conf_threshold=0.5*)
Bases: `gabrieltool.statemachine.callable_zoo.base.CallableBase`

A callable class to execute frozen tensorflow models using TF serving container images.

Use this class if your object detector is generated by OpenTPOD and you have downloaded the model. The TF serving container is started lazily when an FSM runner starts.

CONTAINER_NAME = 'GABRIELTOOL-TFServingContainerCallable-129'

SERVED_DIRS = {}

TFSERVING_GRPC_PORT = 8500

clean()

container_external_port
Port of the TF Serving container.

classmethod from_json(*json_obj*)
Deserialize.

prepare()
Launch the TF serving container. Do not call this method directly unless debugging.

This function is called when an FSM runner starts. This enables gabrieltool to start only one TF serving container to serve many models.

gabrieltool.statemachine.callable_zoo.processor_zoo.tfutils module

Utilities for using Tensorflow models.

class TFServingPredictor(*host*, *port*)
Bases: `object`

An agent that makes request to a TF serving server to get object detection results.

This agent communicates with the TF serving server (often a container at localhost) through gRPC.

__init__(*host*, *port*)
Constructor.

Parameters

- **host** (*string*) – TF serving server hostname or IP address.
- **port** (*int*) – TF serving server port number.

infer_one(*model_name*, *rgb_image*, *conf_threshold=0.5*)
Infer one image by sending a request to TF serving server.

Parameters

- **model_name** (*string*) – Name of the Model
- **rgb_image** (*numpy array*) – Image in RGB format
- **conf_threshold** (*float, optional*) – Cut-off threshold for detection. Defaults to 0.5.

Returns keys are class ids, values are list of [x1, y1, x2, y2, confidence, label_idx]. e.g {'cat': [[0, 0, 100, 100, 0.6, 'cat']], 1: [[0, 0, 100, 100, 0.7, 1]]}

Return type Dictionary

Module contents

A collection of Callable classes to be used by Processors (in FSM states).

Submodules

gabrieltool.statemachine.callable_zoo.base module

Base class and helper functions for callable classes.

class CallableBase
Bases: `object`

Base class for Callables used in FSMs.

Custom callable classes need to inherit from this class. Inherited classes should add decorator @record_kwargs to their constructors for serialization.

classmethod from_json(*json_obj*)
Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

class Null
Bases: *gabrieltool.statemachine.callable_zoo.base.CallableBase*

A empty callable class that returns None.

Useful for initialization.

classmethod from_json(*json_obj*)

Create a CallableBase class instance from a json object.

Subclasses should overide this class depending on the input type of their constructor.

record_kwargs(*func*)

Decorator to automatically record constructor arguments

```
>>> class process:
...     @record_kwargs
...     def __init__(self, cmd, reachable=False, user='root'):
...         pass
>>> p = process('halt', True)
>>> p.cmd, p.reachable, p.user
('halt', True, 'root')
```

Module contents

A collection of Callable classes to be used by Processors and TransitionPredicates.

Submodules

gabrieltool.statemachine.fsm module

Components to Create a Finite State Machine.

See FSM's wikipedia page for its basics: https://en.wikipedia.org/wiki/Finite-state_machine.

This modules provides components to create, edit, serialize, and deserialize a finite state machine. Below is a list of key concepts.

- **State: FSM states represents the status of a cognitive assistant. States** have Processors, which are executed to analyze the input data when the application is in the state.
- **Transitions: Transitions define the conditions (TransitionPredicate) for** state change and actions (Instruction) to take when changing states.
- **Finite State Machine (StateMachine): An FSM is a set of states and** transitions. Helper functions are provided in the StateMachine class for serialization, deserialization and traversal.

class Instruction(*name=None*, *audio=None*, *image=None*, *video=None*)

Bases: `gabrieltool.statemachine.fsm._FSMObjBase`

Instruction to return when a transition is taken.

__init__(*name=None*, *audio=None*, *image=None*, *video=None*)

Instructions can have audio, image, or video.

Parameters

- **name** (*string, optional*) – Name of the instruction. Defaults to None.
- **audio** (*string, optional*) – Verbal instruction in text. Defaults to None.
- **image** (*bytes, optional*) – Encoded Image in bytes. Defaults to None.
- **video** (*url string, optional*) – Video Url in string. Defaults to None.

from_desc(*desc*)

Construct an object from its serialized description.

to_desc()
Returned the serialized description of this object as a protobuf message.

class Processor(*name=None*, *callable_obj=None*)
Bases: `gabrieltool.statemachine.fsm._FSMCallable`

Processor specifies how to process input (e.g. an image) in a state.

__init__(*name=None*, *callable_obj=None*)
Construct a processor.

Parameters

- **name** (*string, optional*) – Name of the processor. Defaults to None.
- **callable_obj** (*subclass of callable_zoo.CallableBase, optional*) – An object whose type is a subclass of callable_zoo.CallableBase. This object is called/executed when there is a new input (e.g. an image). This callable_obj should expect exact one positional argument. Defaults to None.

callable_obj
The callable object to invoke when this object is called.

from_desc(*data*)
Construct an object from its serialized description.

prepare()
Invoke prepare() method of the callable_obj if it has any.

This prepare method is called when the FSM runner starts executing to give callables an opportunity to initialize themselves.

to_desc()
Returned the serialized description of this object as a protobuf message.

class State(*name=None*, *processors=None*, *transitions=None*)
Bases: `gabrieltool.statemachine.fsm._FSMObjBase`

A FSM state represents the status of the system.

A state can have many processors and transitions.

__init__(*name=None*, *processors=None*, *transitions=None*)
Construct a FSM state.

Parameters

- **name** (*string, required*) – Name of the State. Each state in a FSM needs to have an unique name.
- **processors** (*list of Processor, optional*) – Processor to run on an input in this state. Each processor will be called with exactly one positional argument (input), and should return a dictionary that contains the extracted information. The returned dictionaries from multiple processors will be unioned together to serve as inputs to transition predicates. Defaults to None.
- **transitions** (*list of Transition, optional*) – Possible Transitions from this state. Transitions are evaluated one by one in the order of this list. The first transition that satisfies will be taken. Defaults to None.

from_desc()
Do not call this method directly.

State itself does not know enough information to build from its description. The next_state in state's transitions depends on a FSM. Use StateMachine Helper Class instead.

Raises `NotImplementedError` – Always

`prepare``()`
Prepare a state (e.g. initialize all processors and transition predicates.)

This method is called when the FSM runner first starts to give callables an opportunity to initialize themselves.

`processors`
The list of processors to be executed in this state.

`to_desc``()`
Returned the serialized description of this object as a protobuf message.

`transitions`
The list of possible transitions to take in this state.

`class StateMachine`
Bases: `object`

Helper class to serialize, deserialize, and traverse a state machine.

`classmethod bfs``(`*start_state*`)`
Generator for a breadth-first traversal on the FSM.

This method can be used to enumerate states in an FSM.

Parameters **`start_state`** (`State`) – The start state of the traversal.

Yields *State* – The current state of the traversal.

`classmethod from_bytes``(`*data*`)`
Load a State Machine from bytes.

Parameters

- **`data`** (*`bytes`*) – Serialized FSM in bytes. Format is specified in
- **`wca_state_machine.proto.`** –

Raises `ValueError` – raised when there are duplicate state names.

Returns The start state of the FSM.

Return type *State*

`classmethod to_bytes``(`*name*, *start_state*`)`
Serialize a FSM to bytes.

States in the FSM are discovered using a breadth-first search (see the bfs method in this class).

Parameters

- **`name`** (*`string`*) – The name of the FSM.
- **`start_state`** (`State`) – The start state of the FSM.

Raises `ValueError` – raised when there are duplicate state names.

Returns Serialized FSM in bytes. Format is defined in wca_state_machine.proto.

Return type bytes

class Transition(*name=None*, *predicates=None*, *instruction=None*, *next_state=None*)

Bases: `gabrieltool.statemachine.fsm._FSMObjBase`

Links among FSM states that defines state changes and results to return when changing states.

A Transition has the following components:

- transition predicates: The conditions that need to be satisfied to take this transition.
- next_state: The next FSM state to visit after taking the transition.
- instructions: Instructions returned to users when this transition is taken.

__init__(*name=None*, *predicates=None*, *instruction=None*, *next_state=None*)

Construct a Transition

Parameters

- **name** (*string, optional*) – Name of the transition. Defaults to None.
- **predicates** (*a list of TransitionPredicate, optional*) – A list of condition to satisfy. They are daisy-chained (AND) together when evaluating whether this transition takes place. Defaults to None.
- **instruction** (`Instruction`, *optional*) – Instruction to give. Defaults to None.
- **next_state** (`State`, *optional*) – Next state to move to. Defaults to None.

from_desc()

Do not call this method directly.

Transition itself does not know enough information to build from its description. next_state variable depends on a FSM. Use StateMachine Helper Class instead.

Raises `NotImplementedError` – Always

predicates

The list of TransitionPredicates.

to_desc()

Returned the serialized description of this object as a protobuf message.

class TransitionPredicate(*name=None*, *callable_obj=None*)

Bases: `gabrieltool.statemachine.fsm._FSMCallable`

Condition for state transition.

TransitionPredicate determines whether a state transition should taken. TransitionPredciate implements the callable interface so that its objects can be evaluated as a function. A state transition is taken when a TransitionPredicate evaluates to True.

__init__(*name=None*, *callable_obj=None*)

Construct a transition predicate.

Parameters

- **name** (*string, optional*) – Name of the TransitionPredicate. Defaults to None.
- **callable_obj** (*subclass of callable_zoo.CallableBase, optional*) – An object whose type is a subclass of callable_zoo.CallableBase. This object is called/executed when this transition predicate is called to determine whether the current transition should be taken. This callable_obj should expect exact one positional argument of type dictionary, which contains the output of current state's processors. Defaults to None.

callable_obj
The callable object to invoke when this object is called.

from_desc(*data*)
Construct an object from its serialized description.

prepare()
Invoke prepare() method of the callable_obj if it has any.

This prepare method is called when the FSM runner starts executing to give callables an opportunity to initialize themselves.

to_desc()
Returned the serialized description of this object as a protobuf message.

gabrieltool.statemachine.instruction_pb2 module

gabrieltool.statemachine.runner module

Finite State Machine Runner.

Runner to run the cognitive assistants that are expressed as state machines.

class BasicCognitiveEngineRunner(*engine_name*, *fsm*)
Bases: `gabriel_server.cognitive_engine.Engine`

A basic Gabriel Cognitive Engine Runner for FSM based cognitive assistants.

This runner will start a Gabriel cognitive engine that follows the logic defined in a FSM. This class is basic as it restricts the sensor input to be images and the instruction output to be audio or images.

__init__(*engine_name*, *fsm*)
Construct a Gabriel Cognitive Engine Runner.

Parameters

- **engine_name** (*string*) – Name of the cognitive engine.
- **fsm** (`State`) – The start state of an FSM.

handle(*from_client*)
Do not call directly.

This method is invoked by the gabriel framework when a user input is available.

class Runner(*start_state*, *prepare_to_run=True*)
Bases: `object`

Finite State Machine Runner.

A basic finite state machine runner. Make sure the fsm is constructed fully before creating a runner.

__init__(*start_state*, *prepare_to_run=True*)
Construct a FSM runner.

Parameters

- **start_state** (`State`) – The start state of a FSM.
- **prepare_to_run** (*bool, optional*) – Whether to call prepare() functions on all state before running. It should be set to true unless debugging. Defaults to True.

feed(*data*, *debug=False*)
Feed the FSM an input to get an output.

Parameters

- **data** (*any*) – Input data.
- **debug** (*bool, optional*) – Defaults to False.

Raises ValueError – when current state is None.

Returns Instruction from the FSM.

Return type *Instruction*

gabrieltool.statemachine.wca_state_machine_pb2 module

Module contents

Library to build application logic as a Finite State Machine.

_pb2 modules are generated python modules by Protobuf for serialization. See the proto directory for the serialization formats.

Module contents

A set of tools and libraries for fast prototyping wearable cognitive assistants.

1.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/cmusatyalab/OpenWorkflow/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

Write Documentation

OpenWorkflow could always use more documentation, whether as part of the official OpenWorkflow docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/cmusatyalab/OpenWorkflow/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.5.2 Get Started!

Ready to contribute? Here's how to set up *OpenWorkflow* for local development.

1. Fork the *OpenWorkflow* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/gabrieltool.git
   ```

3. Install your local copy into a virtualenv:

   ```
   $ python3 -m venv env
   $ . env/bin/activate
   $ cd gabrieltool/
   $ pip install -r requirements/dev.txt
   $ python setup.py install
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linter (autopep8) and the tests:

   ```
   $ python -m pytest <directory>
   ```

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

1.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. The pull request should work for Python 3.5, 3.6, and 3.7. Check https://github.com/cmusatyalab/OpenWorkflow/actions and make sure that the tests pass for all supported Python versions.

1.5.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed.

gabrieltool Python Package

[This Github workflow](https://github.com/cmusatyalab/OpenWorkflow/blob/master/.github/workflows/pythonpackage.yml.yml) will then deploy to PyPI if tests pass.

FSM Web Editor

Generate Documentation

After pushing to remove, The documentation page will automatically build the docs directory through a webhook integration.

1.6 History

1.6.1 0.0.1 (2018-11-14)

- First release on PyPI.

CHAPTER 2

Installation

2.1 gabrieltool Python Library

First, make sure Docker is installed. You can follow the Docker installation guide.

Option 1. From PyPI (recommended)

```
$ pip install -U gabrieltool
```

Option 2. From Source

```
$ git clone https://github.com/cmusatyalab/OpenWorkflow.git
$ cd OpenWorkflow/
$ python setup.py install
```

2.2 OpenWorkflow State Machine Web Editor

Visit https://cmusatyalab.github.io/OpenWorkflow. Or You can download the build from the repo's gh-pages branch and open the index.html.

CHAPTER 3

Quickstart

3.1 Python Library gabrieltool

Create a Two State FSM

```python
from gabrieltool.statemachine import fsm, predicate_zoo, processor_zoo

# create a two state state machine
st_start = fsm.State(
    name='start',
    processors=[fsm.Processor(
        name='proc_start',
        callable_obj=processor_zoo.DummyCallable()
    )],
    transitions=[
        fsm.Transition(
            name='tran_start_to_end',
            predicates=[
                fsm.TransitionPredicate(
                    callable_obj=predicate_zoo.Always()
                )
            ]
        )
    ]
)
st_end = fsm.State(
    name='end'
)
st_start.transitions[0].next_state = st_end
```

Save the FSM to a file

```python
# save to disk
with open('simple.pbfsm', 'wb') as f:
```

(continues on next page)

(continued from previous page)

```
    f.write(fsm.StateMachine.to_bytes(
        name='simple_fsm',
        start_state=st_start
    ))
```

Launch a gabriel server using the FSM.

```
$ gbt run ./simple.pbfsm
```

See *Tutorial* for a detailed example.

3.2 OpenWorkflow State Machine Editor

The editor is hosted at https://cmusatyalab.github.io/OpenWorkflow/. An instruction video is available here.

This web editor provides the following functionalities.

1. Import to view a saved FSM file. This FSM file can be created either from *gabrieltool* or from the web editor.
2. Export a FSM to a file.
3. Edit FSM states and transitions. Note that only supported operations from processor_zoo and predicate_zoo can be edited. Custom defined functions can not be created or modified in the web editor.

Compared to the gabrieltool Python library, the web editor provides better visualization and is great for creating small FSMs. For more complicated FSMs, consider using the gabrieltool for better reproducibility and efficiency.

3.3 Gabrieltool CLI (gbt)

The gabrieltool cli (gbt) provides a convenient method to launch a gabriel server given an FSM, created by the python library or the web editor.

```
$ gbt run <path-to-fsm>
$ # for usage details, see gbt -h
```

CHAPTER 4

Contribute

- Issue Tracker: https://www.github.com/cmusatyalab/OpenWorkflow/issues
- Source Code: https://www.github.com/cmusatyalab/OpenWorkflow

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index