strategoutil

Release 0.1.0

Mihhail Samusev and Martijn Goorden

Jun 13, 2023

INSTALLATION

Strategoutil is a collection of utility functions and classes to interface UPPAAL Stratego controllers with Python. It furthermore provides an interface to perform model-predictive control or online-control using *STOMPC*.

CHAPTER

ONE

INSTALLATION GUIDE

This part of the documentation provides detailed instruction on the installation of *strategoutil* and UPPAAL Stratego.

1.1 Strategoutil

strategoutil is available through pip:

```
pip install strategoutil
```

Now *strategoutil* can be directly used in python with

```
import strategoutil
```

1.2 UPPAAL Stratego

strategoutil cannot run without an installation of UPPAAL Stratego itself. *strategoutil* comes not with UPPAAL Stratego, so you have to install this by yourself. Follow the instructions below for the recommended way of installing UPPAAL Stratego for different operation systems.

Note: Since UPPAAL 5, UPPAAL Stratego is part of UPPAAL itself. It is recommended to download UPPAAL 5 instead of UPPAAL Stratego. The installation instructions below still apply for UPPAAL 5.

1.2.1 Linux

Go to the download page of UPPAAL Stratego. On this page, choose the latest release and the right build (32-bit versus 64 bit Linux version). If you are doubting about your OS version, go to *Settings* → *about* → *OS Type*. After accepting the license, the download should start.

Open a terminal and move to the folder where UPPAAL Stratego has been downloaded to. Now we will unzip the file with

```
unzip <name of uppaal stratego file>.zip -d $HOME/.local/bin
```

where `<name of uppaal stratego file>.zip` is the name of the downloaded file (depending on the version you downloaded), for example `uppaal-4.1.20-stratego-9-linux64.zip`.

Navigate to the `$HOME/.local/bin` folder with

```
cd $HOME/.local/bin
```

and verify that UPPAAL Stratego has been unzipped correctly to this folder with

```
ls
```

and look for a folder named <name of uppaal stratego file>.

Now we will create a symbolic link to the UPPAAL Stratego engine:

```
ln -s <name of uppaal stratego file>/bin/verifyta <short name>
```

<short name> can be any name you like, but it will become the command that you (and, in fact, *strategoutil*) call from a terminal. The suggestion is to always include the version number of the downloaded UPPAAL Stratego. For example, if you downloaded `uppaal-4.1.20-stratego-9-linux64.zip`, then `verifyta-stratego-9` can be a good name, and the full command for the symbolic link becomes

```
ln -s uppaal-4.1.20-stratego-9-linux64/bin/verifyta verifyta-stratego-9
```

Note: In UPPAAL Stratego versions 7 and lower, the folder structure was slightly different. For these lower versions you need to create the symbolic link with:

```
ln -s <name of uppaal stratego file>/bin-Linux/verifyta <short name>
```

Verify that the symbolic link is created correctly by typing

```
ls -l
```

and check that the symbolic link is not colored red, i.e., red indicates that the link is broken.

Finally, we check whether the symbolic link we created is recognized anywhere in the system. First, navigate to another random folder, for example your home folder

```
cd
```

and type

```
<short name> -h
```

with the short name of the symbolic link, for example `verifyta-stratego-9`. The command line manual of UPPAAL Stratego should now be printed. If so, we are ready to go. Otherwise, look at *Problems with the installation*.

1.2.2 Windows

Todo: No instructions yet.

1.2.3 MacOS

Go to the download page of UPPAAL Stratego. On this page, choose the latest release and the right build. After accepting the license, the download should start.

Unzip the downloaded file. As a result, you should now have a `.app` file. Move the `.app` file to your `Applications` folder.

You can now UPPAAL app from the Applications folder. macOS may put Uppaal into quarantine and suggest to "Move to Bin".

1) Dismiss the popup with error message "UPPAAL" can't be opened because Apple cannot check it for malicious software or "UPPAAL" can't be opened because it was not downloaded from the App store by clicking OK.
2) Go to the Apple menu, select System Settings…, click Privacy & Security in the sidebar, and then scroll down to the Security section on the right.
3) Under Allow apps downloaded from, click the Open Anyway button just after "UPPAAL.app" was blocked from use because it is not from an identified developer, to allow the UPPAAL app to be executed.
4) Approve the application with your credentials in the next popup.
5) Start UPPAAL again (if not restarted automatically), and when the popup appears with the same error message as before, click Open.

Once successfully opened, we will now create a symbolic link to the UPPAAL engine:

```
cd /usr/local/bin
ln -s /Applications/<name of uppaal app file>/Contents/Resources/uppaal/bin/verifyta
↪<short name>
```

`<short name>` can be any name you like, but it will become the command that you (and, in fact, *strategoutil*) call from a terminal. The suggestion is to always include the version number of the downloaded UPPAAL Stratego. For example, if you downloaded `uppaal-4.1.20-stratego-9-app.zip`, then `verifyta-stratego-9` can be a good name, and the full command for the symbolic link becomes

```
ln -s /Applications/uppaal-4.1.20-stratego-9.app/Contents/Resources/uppaal/bin/verifyta␣
↪verifyta-stratego-9
```

Note: UPPAAL Stratego versions 7 and lower are not build for macOS.

Verify that the symbolic link is created correctly by typing

```
<short name> -h
```

with the short name of the symbolic link, for example `verifyta-stratego-9`. The command line manual of UPPAAL Stratego should now be printed.

Finally, we check whether the symbolic link we created is recognized anywhere in the system. First, navigate to another random folder, for example your home folder

```
cd
```

and type

```
<short name> -h
```

with the short name of the symbolic link, for example `verifyta-stratego-9`. Again, the command line manual of UPPAAL Stratego should now be printed. If so, we are ready to go. Otherwise, look at *Problems with the installation*.

1.3 Problems with the installation

We will provide some known solutions to problems you might encounter setting up the *strategoutil* package and UPPAAL Stratego.

1.3.1 Running verifyta command produces a `No such file or directory` error

Older versions of UPPAAL Stratego (up to version 7) cannot properly cope with being installed in `$HOME/.local/bin` folder and having a symbolic link to it. You can solve this by replacing the content of `<uppaal stratego folder>/bin-Linux/verifyta` to

```
#!/usr/bin/env bash
# Use this script when the native dynamic linker is incompatible
SOURCE="${BASH_SOURCE[0]}"
while [ -h "$SOURCE" ]; do
  HERE=$(cd -P $(dirname "$SOURCE") >/dev/null 2>&1 && pwd)
  SOURCE=$(readlink "$SOURCE")
  [[ "$SOURCE" != /* ]] && SOURCE="$HERE/$SOURCE"
done
HERE=$(cd -P $(dirname "$SOURCE") > /dev/null 2>&1 && pwd)
export LD_LIBRARY_PATH="$HERE"
exec -a verifyta "$HERE"/ld-linux.so "$HERE"/verifyta.bin "$@"
```

1.3.2 UPPAAL Stratego command cannot be found

If *strategoutil* generates the runtime error `Cannot find the supplied verifyta command:` or a terminal fails with `command not found`, the symbolic link to UPPAAL Stratego is not in the path variable.

We will first verify that the path variable is indeed the problem by opening a terminal and type

```
echo $PATH
```

Check whether along the printed folders is the folder `$HOME/.local/bin`, where `$HOME` is your home folder (if you do not know this folder, type `echo $HOME` in the terminal). If this folder is missing, both *strategoutil* and the terminal cannot find it.

Note: *strategoutil* has only access to commands through the path variable and not through any aliases defined in, for example, your `.bashrc` or `.zshrc` files. So it might be the case that UPPAAL Stratego is working when you call it with your terminal while *strategoutil* produces this runtime error.

You could temporarily solve the problem by running your script with *strategoutil* from the command line with

```
PATH=$HOME/.local/bin:$PATH python3 <name of script>
```

If you want to do a more thorough investigation, we will inspect the file */etc/skel/.profile*, where the user's bin folder is added to the path variable. Make sure that this file contains

```
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/.local/bin" ] ; then
    PATH="$HOME/.local/bin:$PATH"
fi
```

If the file is missing this part, something might be wrong with your Linux OS, as this file comes with the standard installation of Linux and will not be altered by most users. Therefore, instead of adding the above lines to the file, it might be better to just reinstall your Linux OS.

CHAPTER

TWO

EXAMPLES OF USING THE *STOMPC* TOOL

This part of the documentation provides several examples of how *STOMPC* is used in different case studies. The following examples are currently available:

- *Floor heating*
- *Storm water detention pond*
- *Traffic light control*

CHAPTER

THREE

FLOOR HEATING

This part of the documentation demonstrates how *STOMPC* is used to perform online control for floor heating in a family house.

3.1 Installing the tools

For this case, we need to have two tools available:

- *strategoutil*, the tool as described in this documentation, and
- UPPAAL Stratego, the generic tool that will synthesize strategies.

Information on how to install *strategoutil* and UPPAAL Stratego can be found in the *Installation guide*.

3.2 Preparing the UPPAAL Stratego model

In most cases, a UPPAAL Stratego model suitable for offline strategy synthesis is adjusted to be suitable for online control. That also has been the situation for this case. The model from this paper has been modified for this online model-predictive control setup. In this section, we do not discuss how to adjust a model suitable for offline control to one for online control, but we indicate what you have to do specifically for using *STOMPC* tool to perform this online control.

In the UPPAAL Stratego model, we need to insert placeholders at the variables that will have different values at the start of each MPC step, for example, the room temperature T. These placeholders are strings with the format `//TAG_<varname>`, where `<varname>` is the name of the variable. So, for the clock variable T representing the room temperature, we will rewrite

```
clock T = 18; // room temperature in deg. C
```

into

```
clock T = //TAG_T; // room temperature in deg. C
```

Notice that after the tag there is still the semicolon `;`, as only the placeholder will be replaced by the initial value of that variable. The UPPAAL Stratego GUI will now also start to give a syntax error on the next line, as it cannot find the closing semicolon.

After inserting all the placeholders in the UPPAAL Stratego model, we have to create a model configuration file. This file tells the *STOMPC* tool which variables it need to keep track of during MPC, and what their initial values are for the very first step. The model configuration file has to be a yaml file, but you can use a custom name. For this case, we have the following `floor-heating_config.yaml` file:

```
t: 0.0
T: 18.0
D: 0.0
heatLoc: 0
winLoc: 0
w: 0.0
i: 0
```

Finally, we have to specify the learning and other query parameters. This is also done in a separate yaml-file. Below you can find the content of the `verifyta_config.yaml` file for the floor heating example (with some arbitrarily numbers that ensure fast calculations). In *Define experiment variables* we will indicate in python which files contain the model and strategy configurations. This file contains pairs of the setting name and its value, where the setting name is the one used for the command line interface of UPPAAL Stratego. In case a certain parameter does not have a value, for example `nosummary`, you just leave the value field empty.

```
learning-method: 4
good-runs: 25
total-runs: 25
runs-pr-state: 15
eval-runs: 25
discretization: 1.0
filter: 2
nosummary:
silence-progress:
```

3.3 Specializing the MPCSetup class from *STOMPC*

The *STOMPC* tool provides several classes that can be tailored for the case you want to use it for.

- `MPCsetup`. This class is the primarily class an end-user should specialize for his or her case. It implements the basic MPC scheme. It assumes that UPPAAL Stratego will always success in synthesizing a safe and optimal strategy.
- `SafeMPCSetup`. This class inherits from `MPCsetup`, yet it monitors and detects whether UPPAAL Stratego has successfully synthesized a strategy. If not, it will run UPPAAL Stratego with an alternative query, which has to be specified by the user, as it depends on the model what a safe query would be.

For the floor heating, the primary goal is to synthesize a strategy that minimizes the cumulative distance between the room temperature and target temperature (optimality). As there is no safety requirement, UPPAAL Stratego will always synthesize a strategy. Therefore, the `MPCSetup` class should be specialized.

Below the specialized class `MPCSetupFloorHeating` is defined. As can be seen, we override the `create_query_file` method.

```
import strategoutil as stompc

class MPCSetupFloorHeating(stompc.MPCSetup):
    def create_query_file(self, horizon, period, final):
        """
        Create the query file for each step of the room heating model. Current
        content will be overwritten.

        Overrides MPCsetup.create_query_file().
```

(continues on next page)

(continued from previous page)

```
        """
        with open(self.query_file, "w") as f:
            line1 = f"strategy opt = minE (D) [<={horizon}*{period}]: <> (t=={final})\n"
            f.write(line1)
            f.write("\n")
            line2 = f"simulate 1 [<={period}+1] {{ " \
                f"{self.controller.get_var_names_as_string()} }} under opt\n"
            f.write(line2)
```

In method `create_query_file` we specify the strategy synthesis query. For the floor heating case, we have this defined with `line1`. It states that we want to synthesize a strategy that we call `opt` that minimizes the expected value of clock variable D (representing the cost in the model) where all runs have a maximum duration of the number of periods (denoted by `horizon`) and UPPAAL Stratego time units per period (denoted by `period`) such that eventually the time variable t reaches its final value.

Furthermore, we have a simulate query in this method. Only the first period is simulated to obtain the first control action of the synthesized strategy `opt` and the system's state after one period.

3.4 Define experiment variables

We can now define and set all the experiment variables. These include, for example, file paths to the UPPAAL Stratego model.

```
import yaml

if __name__ == "__main__":
    # We specify the Uppaal files.
    modelTemplatePath = "floor-heating-online.xml"
    queryFilePath = "floor-heating-online_query.q"
    outputFilePath = "results.txt"
    modelConfigPath = "floor-heating_config.yaml"
    learningConfigPath = "verifyta_config.yaml"
    verifytaCommand = "verifyta-stratego-9"

    # Define MPC model variables.
    debug = True  # Whether to run in debug mode.
    period = 15  # Period in time units (minutes).
    horizon = 5  # How many periods to compute strategy for.
    duration = 96  # Duration of experiment in periods.
```

After this we load the two configuration files:

```
# Get model and learning config dictionaries from files.
with open(model_config_path, "r") as yamlfile:
    model_cfg_dict = yaml.safe_load(yamlfile)
with open(learning_config_path, "r") as yamlfile:
    learning_cfg_dict = yaml.safe_load(yamlfile)
```

Finally, we can create the MPC object from our `MPCSetupFloorHeating` class and call `run` with the MPC inputs:

```
# Construct the MPC object.
controller = MPCSetupFloorHeating(modelTemplatePath, query_file=queryFilePath,
                                  output_file_path=outputFilePath, model_cfg_dict=model_
↪cfg_dict,
                                  learning_args=learning_cfg_dict,
                                  verifyta_command=verifytaCommand, debug=debug)

controller.run(controller.run(period, horizon, duration)
```

CHAPTER

FOUR

STORM WATER DETENTION POND

This part of the documentation demonstrates how *STOMPC* is used to perform online control for storm water detention ponds.

4.1 Installing the tools

For this case, we need to have three tools available:

- *strategoutil*, the tool as described in this documentation,
- UPPAAL Stratego, the generic tool that will synthesize strategies, and
- pySWMM the Python API for SWMM, the domain-specific tool that will perform detailed simulations of the pond.

Information on how to install *strategoutil* and UPPAAL Stratego can be found in the *Installation guide*.

pySWMM is available though pip:

```
pip install pyswmm
```

4.2 Preparing the UPPAAL Stratego model

In most cases, a UPPAAL Stratego model suitable for offline strategy synthesis is adjusted to be suitable for online control. That also has been the situation for this case. The model from this paper has been modified for this online model-predictive control setup. In this section, we do not discuss how to adjust a model suitable for offline control to one for online control, but we indicate what you have to do specifically for using *STOMPC* tool to perform this online control.

In the UPPAAL Stratego model, we need to insert placeholders at the variables that will have different values at the start of each MPC step, for example, the water level w. These placeholders are strings with the format `//TAG_<varname>`, where `<varname>` is the name of the variable. So, for the clock variable w representing the water level, we will rewrite

```
clock w = 100; // water level in pond [cm]
```

into

```
clock w = //TAG_w; // water level in pond [cm]
```

Notice that after the tag there is still the semicolon `;`, as only the placeholder will be replaced by the initial value of that variable. The UPPAAL Stratego GUI will now also start to give a syntax error on the next line, as it cannot find the closing semicolon.

After inserting all the placeholders in the UPPAAL Stratego model, we have to create a model configuration file. This file tells the *STOMPC* tool which variables it need to keep track of during MPC, and what their initial values are for the very first step. The model configuration file has to be a yaml file, but you can use a custom name. For this case, we have the following `pond_experiment_config.yaml` file:

```
w: 0.0
```

Finally, we have to specify the learning and other query parameters. This is also done in a separate yaml-file. Below you can find the content of the `verifyta_config.yaml` file for the storm water pond (with some arbitrarily numbers that ensure fast calculations). In *Define experiment variables* we will indicate in python which files contain the model and strategy configurations. This file contains pairs of the setting name and its value, where the setting name is the one used for the command line interface of UPPAAL Stratego. In case a certain parameter does not have a value, for example `nosummary`, you just leave the value field empty.

```
learning-method: 4
good-runs: 10
total-runs: 20
runs-pr-state: 5
eval-runs: 5
discretization: 0.5
filter: 2
nosummary:
silence-progress:
```

4.3 Specializing the SafeMPCSetup class from *STOMPC*

The *STOMPC* tool provides several classes that can be tailored for the case you want to use it for.

- `MPCsetup`. This class is the primarily class an end-user should specialize for his or her case. It implements the basic MPC scheme. It assumes that UPPAAL Stratego will always success in synthesizing a safe and optimal strategy.
- `SafeMPCSetup`. This class inherits from `MPCsetup`, yet it monitors and detects whether UPPAAL Stratego has successfully synthesized a strategy. If not, it will run UPPAAL Stratego with an alternative query, which has to be specified by the user, as it depends on the model what a safe query would be.

For the storm water detention pond, the primary goal is to synthesize a strategy that ensures no overflow (safety) while maximizing particle sedimentation (optimality). Nonetheless, it might be the case that overflow cannot be prevented by any strategy, thus UPPAAL Stratego will fail. Therefore, the `SafeMPCSetup` class should be specialized.

Below the specialized class `MPCSetupPond` is defined. As can be seen, we override three methods for the pond case: `create_query_file`, `create_alternative_query_file`, and `perform_at_start_iteration`.

```
import strategoutil as stompc
import weather_forecast_generation as weather
import datetime

class MPCSetupPond(stompc.SafeMPCSetup):
    def create_query_file(self, horizon, period, final):
        """
        Create the query file for each step of the pond model.
        Current content will be overwritten.

        Overrides SafeMPCsetup.create_query_file().
```

(continues on next page)

(continued from previous page)

```
        """
        with open(self.queryfile, "w") as f:
            line1 = f"strategy opt = minE (c) [<={horizon}*{period}]: " \
                    f"<> (t=={final} && o <= 0)\n"
            f.write(line1)
            f.write("\n")
            line2 = f"simulate 1 [<={period}+1] {{" \
                    f"{self.controller.get_var_names_as_string()} }} under opt\n"
            f.write(line2)

    def create_alternative_query_file(self, horizon, period, final):
        """
        Create an alternative query file in case the original
        query could not be satisfied by Stratego, i.e., it could
        not find a strategy. Current content will be overwritten.

        Overrides SafeMPCsetup.create_alternative_query_file().
        """
        with open(self.queryfile, "w") as f:
            line1 = f"strategy opt = minE (w) [<={horizon}*{period}]: <> (t=={final})\n"
            f.write(line1.format(horizon, period, final))
            f.write("\n")
            line2 = f"simulate 1 [<={period}+1] " \
                    f"{{ {self.controller.get_var_names_as_string()} }} under opt\n"
            f.write(line2)

    def perform_at_start_iteration(self, controlperiod, horizon, duration, step,␣
↪**kwargs):
        """
        Performs some customizable preprocessing steps at the
        start of each MPC iteration.

        Overrides SafeMPCsetup.perform_at_start_iteration().
        """
        current_date = kwargs["start_date"] + datetime.timedelta(hours=step)
        weather.create_weather_forecast(kwargs["historical_rain_data_path"],
                                        kwargs["weather_forecast_path"],
                                        current_date,
                                        horizon * controlperiod,
                                        kwargs["uncertainty"])
```

In method `create_query_file` we specify the strategy synthesis query. For the pond case, we have this defined with `line1`. It states that we want to synthesize a strategy that we call `opt` that minimizes the expected value of clock variable c (representing the cost in the model) where all runs have a maximum duration of the number of periods (denoted by `horizon`) and UPPAAL Stratego time units per period (denoted by `period`) such that eventually the time variable t reaches its final value and accumulated overflow duration o is zero or less.

Furthermore, we have a simulate query in this method. Only the first period is simulated to obtain the first control action of the synthesized strategy `opt`.

The second method `create_alternative_query_file` specifies the query in case there is overflow and UPPAAL Stratego fails to synthesize a safe strategy. We have almost the same strategy synthesis query, except we removed the requirement that no overflow can occur ($o \leq 0$) and we want to minimize the water level w instead of the cost c.

Finally, at the start of each MPC iteration, we need to create a weather forecast. These are generated from historical rain data and, similarly to real weather forecasts, these change over time. Therefore, we create new ones each iteration. A separate custom library contains methods to generate weather forecasts.

4.4 Define experiment variables

We can now define and set all the experiment variables. These include, for example, file paths to the UPPAAL Stratego and SWMM models.

```
import yaml

if __name__ == "__main__":
    # SWMM files.
    swmm_inputfile = "swmm_simulation.inp"
    rain_data_file = "swmm_5061.dat"

    # Other variables of swimm.
    orifice_id = "OR1"
    basin_id = "SU1"
    time_step = 60 * 60 # duration of SWMM simulation step in seconds.
    swmm_results = "swmm_results_online.csv"

    # Now we specify the Uppaal files.
    model_template_path = "pond_experiment.xml"
    query_file_path = "pond_experiment_query.q"
    model_config_path = "pond_experiment_config.yaml"
    learning_config_path = "verifyta_config.yaml"
    weather_forecast_path = "weather_forecast.csv"
    output_file_path = "stratego_result.txt"
    verifyta_command = "verifyta-stratego-9"

    # Define MPC model variables.
    action_variable = "Open"  # Name of the control variable.
    debug = True  # Whether to run in debug mode.
    period = 60  # Control period in Stratego time units (minutes).
    horizon = 12  # How many periods to compute strategy for.
    uncertainty = 0.1  # The uncertainty in the weather forecast generation.
```

After this we load the two configuration files:

```
# Get model and learning config dictionaries from files.
with open(model_config_path, "r") as yamlfile:
    model_cfg_dict = yaml.safe_load(yamlfile)
with open(learning_config_path, "r") as yamlfile:
    learning_cfg_dict = yaml.safe_load(yamlfile)
```

Finally, we can create the MPC object from our `MPCSetupPond` class:

```
# Construct the MPC object.
controller = MPCSetupPond(model_template_path, output_file_path, queryfile=query_file_
↪path,
                          model_cfg_dict=model_cfg_dict, learning_args=learning_cfg_dict,
```

(continues on next page)

(continued from previous page)

```
                            verifyta_command=verifyta_command, external_simulator=False,
                            action_variable=action_variable, debug=debug)
```

4.5 Combining strategy synthesis and simulation

Finally, we need to actually define how *STOMPC* should combine UPPAAL Stratego and SWMM together. Because SWMM is a stateful simulator from which we cannot extract the full state through the pySWMM API, we cannot use the default `SafeMPCSetup.run` method to perform MPC. Therefore, we will 'pause' the SWMM simulator after each step and let `SafeMPCSetup` perform a single MPC step instead.

The method below will start and run the SWMM simulation, and after each step ask for the next control setting.

```
from pyswmm import Simulation, Nodes, Links
import csv

def swmm_control(swmm_inputfile, orifice_id, basin_id, time_step, swmm_results,
                 controller, period, horizon, rain_data_file, weather_forecast_path,
                 uncertainty):
    # Arrays for storing simulation results before writing it to file.
    time_series = []
    water_depth = []
    orifice_settings = []

    with Simulation(swmm_inputfile) as sim:
        # Get the pond and orifice objects from the simulation.
        pond = Nodes(sim)[basin_id]
        orifice = Links(sim)[orifice_id]

        sim.step_advance(time_step)
        current_time = sim.start_time

        # Ask for the first control setting.
        orifice.target_setting = get_control_strategy(pond.depth, current_time,␣
↪controller,
                                                      period, horizon, rain_data_file,
                                                      weather_forecast_path, uncertainty)

        # Get the initial data points.
        orifice_settings.append(orifice.target_setting)
        time_series.append(sim.start_time)
        water_depth.append(pond.depth)

        for step in sim:
            current_time = sim.current_time
            time_series.append(current_time)
            water_depth.append(pond.depth)

            # Get and set the control parameter for the next period.
            orifice.target_setting = get_control_strategy(pond.depth, current_time,
                                                          controller, period, horizon,
```

(continues on next page)

(continued from previous page)

```
                                                        rain_data_file,
                                                        weather_forecast_path,␣
↪uncertainty)
            orifice_settings.append(orifice.target_setting)

    # Write results to file.
    with open(swmm_results, "w") as f:
        writer = csv.writer(f)
        for i, j, k in zip(time_series, water_depth, orifice_settings):
            i = i.strftime('%Y-%m-%d %H:%M')
            writer.writerow([i, j, k])
```

The method `get_control_strategy` that gets the next control setting is defined below. It first updates the state of the controller by updating the value of the water level w as obtained by the SWMM simulation. Subsequently, it performs the `run_single` method that performs a single MPC step. This method returns the control setting for the next period.

```
def get_control_strategy(current_water_level, current_time, controller, period, horizon,
                         rain_data_file, weather_forecast_path, uncertainty):
    # The 100 is due to conversion from m to cm.
    controller.controller.update_state({'w':current_water_level * 100})
    control_setting = controller.run_single(period, horizon, start_date=current_time,
                                            historical_rain_data_path=rain_data_file,
                                            weather_forecast_path=weather_forecast_path,
                                            uncertainty=uncertainty)

    return control_setting
```

Finally, we have to start everything in our `main` block. We do this by simply calling `swmm_control` with the necessary inputs.

```
swmm_control(swmm_inputfile, orifice_id, basin_id, time_step, swmm_results, controller,
             period, horizon, rain_data_file, weather_forecast_path, uncertainty)
```

CHAPTER

FIVE

TRAFFIC LIGHT CONTROL

This part of the documentation demonstrates how *STOMPC* is used to perform online control for a traffic light intersection.

5.1 Installing the tools

For this case, we need to have two tools available:

- *strategoutil*, the tool as described in this documentation, and
- UPPAAL Stratego, the generic tool that will synthesize strategies.

Information on how to install *strategoutil* and UPPAAL Stratego can be found in the *Installation guide*.

5.2 Preparing the UPPAAL Stratego model

In the UPPAAL Stratego model, we need to insert placeholders at the variables that will have different values at the start of each MPC step, for example, the traffic light phase $phase$. These placeholders are strings with the format `//TAG_<varname>`, where `<varname>` is the name of the variable. So, for the integer variable $phase$ representing the traffic light phase, we will rewrite

```
int phase = 0; // 0 east green, 1 south green
```

into

```
int phase = //TAG_phase; // 0 east green, 1 south green
```

Notice that after the tag there is still the semicolon `;`, as only the placeholder will be replaced by the initial value of that variable. The UPPAAL Stratego GUI will now also start to give a syntax error on the next line, as it cannot find the closing semicolon.

After inserting all the placeholders in the UPPAAL Stratego model, we have to create a model configuration file. This file tells the *STOMPC* tool which variables it need to keep track of during MPC, and what their initial values are for the very first step. The model configuration file has to be a yaml file, but you can use a custom name. For this case, we have the following `traffic-light_config.yaml` file:

```
t: 0.0
E: 0
S: 0
phase: 0
Q: 0.0
```

Finally, we have to specify the learning and other query parameters. This is also done in a separate yaml-file. Below you can find the content of the `verifyta_config.yaml` file for the traffic light example (with some arbitrarily numbers that ensure fast calculations). In *Define experiment variables* we will indicate in python which files contain the model and strategy configurations. This file contains pairs of the setting name and its value, where the setting name is the one used for the command line interface of UPPAAL Stratego. In case a certain parameter does not have a value, for example `nosummary`, you just leave the value field empty.

```
learning-method: 4
good-runs: 100
total-runs: 100
runs-pr-state: 100
eval-runs: 100
max-iterations: 30
filter: 0
nosummary:
silence-progress:
```

5.3 Specializing the MPCSetup class from *STOMPC*

The *STOMPC* tool provides several classes that can be tailored for the case you want to use it for.

- `MPCsetup`. This class is the primarily class an end-user should specialize for his or her case. It implements the basic MPC scheme. It assumes that UPPAAL Stratego will always success in synthesizing a safe and optimal strategy.
- `SafeMPCSetup`. This class inherits from `MPCsetup`, yet it monitors and detects whether UPPAAL Stratego has successfully synthesized a strategy. If not, it will run UPPAAL Stratego with an alternative query, which has to be specified by the user, as it depends on the model what a safe query would be.

For the traffic light system, the primary goal is to synthesize a strategy that minimizes the cumulative number of waiting cars (optimality). As there is no safety requirement, UPPAAL Stratego will always synthesize a strategy. Therefore, the `MPCSetup` class should be specialized.

Below the specialized class `MPCSetupTrafficLight` is defined. As can be seen, we override the `create_query_file` method.

```
import strategoutil as stompc

class MPCSetupTrafficLight(stompc.MPCsetup):
    # Overriding parent method.
    def create_query_file(self, horizon, period, final):
        """
        Create the query file for each step of the traffic light model. Current
        content will be overwritten.
        """
        with open(self.query_file, "w") as f:
            line1 = f"strategy opt = minE (Q) [<={horizon}*{period}]: <> (t=={final})\n"
            f.write(line1)
            f.write("\n")
            line2 = f"simulate 1 [<={period}+1] {{ " \
                    f"{self.controller.get_var_names_as_string()} }} under opt\n"
            f.write(line2)
```

In method `create_query_file` we specify the strategy synthesis query. For the traffic light case, we have this defined

with `line1`. It states that we want to synthesize a strategy that we call `opt` that minimizes the expected value of clock variable Q (representing the cost in the model) where all runs have a maximum duration of the number of periods (denoted by `horizon`) and UPPAAL Stratego time units per period (denoted by `period`) such that eventually the time variable t reaches its final value.

Furthermore, we have a simulate query in this method. Only the first period is simulated to obtain the first control action of the synthesized strategy `opt` and the system's state after one period.

5.4 Define experiment variables

We can now define and set all the experiment variables. These include, for example, file paths to the UPPAAL Stratego model.

```
import yaml

if __name__ == "__main__":
    # Define location of the relevant files and commands.
    modelTemplatePath = "traffic-light_template.xml"
    queryFilePath = "traffic-light_query.q"
    outputFilePath = "results.txt"
    modelConfigPath = "traffic-light_config.yaml"
    learningConfigPath = "verifyta_config.yaml"
    verifytaCommand = "verifyta-stratego-9"

    # Define MPC model variables.
    debug = True  # Whether to run in debug mode.
    period = 60  # Period in time units (minutes).
    horizon = 1  # How many periods to compute strategy for.
    duration = 30  # Duration of experiment in periods.
```

After this we load the two configuration files:

```
# Get model and learning config dictionaries from files.
with open(model_config_path, "r") as yamlfile:
    model_cfg_dict = yaml.safe_load(yamlfile)
with open(learning_config_path, "r") as yamlfile:
    learning_cfg_dict = yaml.safe_load(yamlfile)
```

Finally, we can create the MPC object from our `MPCSetupTrafficLight` class and call `run` with the MPC inputs:

```
# Construct the MPC object.
controller = MPCSetupTrafficLight(modelTemplatePath, query_file=queryFilePath,
                                  output_file_path=outputFilePath, model_cfg_dict=model_
↪cfg_dict,
                                  learning_args=learning_cfg_dict,
                                  verifyta_command=verifytaCommand, debug=debug)

controller.run(controller.run(period, horizon, duration)
```

CHAPTER

SIX

API DOCUMENTATION

This part of the documentation covers the interfaces used to develop with `strategoutil`.

6.1 StrategoController

class strategoutil.**StrategoController**(*model_template_file*, *model_cfg_dict*, *cleanup=True*)

Bases: `object`

Controller class to interface with UPPAAL Stratego through python.

Parameters

- **model_template_file** (*str*) – The file name of the template model.
- **model_cfg_dict** (*dict*) – Dictionary containing pairs of state variable name and its initial value. The state variable name should match the tag name in the template model.
- **cleanup** (*bool*) – Whether or not to clean up the temporarily simulation file after being used.

Variables

- **states** (*bool*) – Dictionary containing the current state of the system, where a state is a pair of variable name and value. It is initialized with the values from *model_cfg_dict*.
- **tagRule** (*str*) – The rule for each tag in the template model. Currently, the rul is set to be "`//TAG_{}`". Therefore, tags in the template model should be "`//TAG_<variable name>`", where `<variable name>` is the global name of the variable.

debug_copy(*debug_file*)

Copy UPPAAL simulationfile.xml file for manual debug in Stratego.

Parameters
debug_file (*str*) – The file name of the debug file.

get_state(*key*)

Get the current value of the provided state variable.

Parameters
key (*str*) – The state variable name.

Returns
The currently stored value of the state variable.

Return type
int or float

`get_state_as_string()`

Print the values of the state variables separated by a ‘,’.

Returns
All the variable values joined together with a ‘,’.

Return type
str

`get_states()`

Get the current states.

Returns
The current state dictionary.

Return type
dict

`get_var_names_as_string()`

Print the names of the state variables separated by a ‘,’.

Returns
All the variable names joined together with a ‘,’.

Return type
str

`init_simfile()`

Make a copy of a template file where data of specific variables is inserted.

`insert_state()`

Insert the current state values of the variables at the appropriate position in the simulation *.xml file indicated by the `tagRule`.

`remove_simfile()`

Clean created temporary files after the simulation is finished.

`run`(*query_file=''*, *learning_args=None*, *verifyta_command='verifyta'*)

Runs verifyta with requested queries and parameters that are either part of the *.xml model file or explicitly specified.

Parameters

- **query_file** (*str*) – The file name of the query file where the queries are written to.
- **learning_args** (*dict*) – Dictionary containing the learning parameters and their values. The learning parameter names should be those used in the command line interface of Uppaal Stratego. You can also include non-learning command line parameters in this dictionary. If a non-learning command line parameter does not take any value, include the empty string "" as value.
- **verifyta_command** (*str*) – The command name for running Uppaal Stratego at the user’s machine.

Returns
The output generated by Uppaal Stratego.

Return type
str

update_state(*new_values*)

Update the state of the MPC controller.

Parameters

new_values (*dict*) – Dictionary containing new values for the state variables.

6.2 MPCsetup

class strategoutil.**MPCsetup**(*model_template_file*, *output_file_path=None*, *query_file=''*, *model_cfg_dict=None*, *learning_args=None*, *verifyta_command='verifyta'*, *external_simulator=False*, *action_variable=None*, *debug=False*)

Bases: object

Class that performs the basic MPC scheme for Uppaal Stratego.

The class parameters are also available as attributes.

Parameters

- **model_template_file** (*str*) – The file name of the template model.
- **output_file_path** (*str*) – The file name of the output file where the results are printed to.
- **query_file** (*str*) – The file name of the query file where the queries are written to.
- **model_cfg_dict** (*dict*) – Dictionary containing pairs of state variable name and its initial value. The state variable name should match the tag name in the template model.
- **learning_args** (*dict*) – Dictionary containing the learning parameters and their values. The learning parameter names should be those used in the command line interface of Uppaal Stratego. You can also include non-learning command line parameters in this dictionary. If a non-learning command line parameter does not take any value, include the empty string "" as value.
- **verifyta_command** (*str*) – The command name for running Uppaal Stratego at the user's machine.
- **external_simulator** (*bool*) – Whether an external simulator is used to obtain the true state after applying the synthesized control strategy for a single control period.
- **action_variable** (*str*) – Name of the variable in the model that captures the control actions to choose from. Only relevant if an external simulator is used, as we need to get the chosen control action from Uppaal Stratego and pass it on to the external simulator. It should be a variable in *model_cfg_dict*.
- **debug** (*bool*) – Whether or not to run in debug mode.

Variables

controller (*StrategoController*) – The controller object used for interacting with Uppaal Stratego.

create_query_file(*horizon*, *period*, *final*)

Create a basic query file for each step of the MPC scheme. Current content will be overwritten.

You might want to override this method for specific models.

Parameters

- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of periods.
- **period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **final** (*int*) – The time that should be reached by the synthesized strategy, given in Uppaal Stratego time units. Most likely this will be current time + *horizon* x *period*.

extract_control_action_from_stratego(*stratego_output*)

Extract the chosen control action for the first control period from the simulation output of Stratego.

Parameters
stratego_output (*str*) – The output as generated by Uppaal Stratego.

Returns
The control action chosen for the first control period.

Return type
float

extract_states_from_stratego(*result*, *control_period*)

Extract the new state values from the simulation output of Stratego.

The extracted values are directly saved in the `controller`.

Parameters

- **result** (*str*) – The output as generated by Uppaal Stratego.
- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.

perform_at_start_iteration(**args*, ***kwargs*)

Perform some customizable preprocessing steps at the start of each MPC iteration. This method can be overwritten for specific models.

print_state()

Print the current state to output file if provided. Otherwise, it will be printed to the standard output.

print_state_vars()

Print the names of the state variables to output file if provided. Otherwise, it will be printed to the standard output.

run(*control_period*, *horizon*, *duration*, ***kwargs*)

Run the basic MPC scheme where the controller can changes its strategy once every period, where the strategy synthesis looks the horizon ahead, and continues for the duration of the experiment.

The control period is in Uppaal Stratego time units. Both horizon and duration have control period as time unit.

Parameters

- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of control periods.
- **duration** (*int*) – The number of times (steps) the MPC scheme should be performed, given as the number of control periods.

- ****kwargs** – Any additional parameters are forwarded to *perform_at_start_iteration()*.

run_external_simulator(*chosen_action*, **args*, ***kwargs*)

Run an external simulator to obtain the 'true' state after applying the synthesized control action for a single control period.

This method should be overridden by the user. The method should return the new 'true' state as a dictionary containing pairs where the key is a variable name and the value is its new value.

Parameters
chosen_action (*int or float*) – The synthesized control action for the first control period.

Returns
The 'true' state of the system after simulation a single control period. The dictionary containings pairs of state variable name and their values. The state variable name should match the tag name in the template model.

Return type
dict

run_single(*control_period*, *horizon*, ***kwargs*)

Run the basic MPC scheme a single step where a single controller strategy is calculated, where the strategy synthesis looks the horizon ahead, and continues for the duration of the experiment.

The control period is in Uppaal Stratego time units. Horizon have control period as time unit.

Parameters

- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of control periods.
- ****kwargs** – Any additional parameters are forwarded to *perform_at_start_iteration()*.

Returns
The control action chosen for the first control period.

run_verifyta(**args*, ***kwargs*)

Run verifyta with the current data stored in this class.

Parameters

- ***args** – Is not used in this method; it is used in the overriding method *run_verifyta()* in *SafeMPCSetup*.
- ****kwargs** – Is not used in this method; it is used in the overriding method *run_verifyta()* in *SafeMPCSetup*.

Returns
The output generated by Uppaal Stratego.

Return type
str

step_without_sim(*control_period*, *horizon*, *duration*, *step*, ***kwargs*)

Perform a step in the basic MPC scheme without the simulation of the synthesized strategy.

Parameters

- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of control periods.
- **duration** (*int*) – The number of times (steps) the MPC scheme should be performed, given as the number of control periods. Is only forwarded to *perform_at_start_iteration()*.
- **step** (*int*) – The current iteration step in the basic MPC loop.
- **kwargs** – Any additional parameters are forwarded to *perform_at_start_iteration()*.

Returns
The output generated by Uppaal Stratego.

Return type
str

6.3 SafeMPCSetup

class strategoutil.**SafeMPCSetup**(*model_template_file*, *output_file_path=None*, *query_file=''*, *model_cfg_dict=None*, *learning_args=None*, *verifyta_command='verifyta'*, *external_simulator=False*, *action_variable=None*, *debug=False*)

Bases: *MPCsetup*

Class that performs the basic MPC scheme for Uppaal Stratego.

The class monitors and detects whether Uppaal Stratego has sucessfully synthesized a strategy. If not, it will run Uppaal Stratego with an alternative query, which has to be specified by the user, as it depends on the model what a safe query would be.

create_alternative_query_file(*horizon*, *period*, *final*)

Create an alternative query file in case the original query could not be satisfied by Stratego, i.e., it could not find a strategy.

Parameters

- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of periods.
- **period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **final** (*int*) – The time that should be reached by the synthesized strategy, given in Uppaal Stratego time units. Most likely this will be current time + *horizon* x *period*.

run_verifyta(*horizon*, *control_period*, *final*, **args*, ***kwargs*)

Run verifyta with the current data stored in this class.

It verifies whether Stratego has successfully synthesized a strategy. If not, it will create an alternative query file and run Stratego again.

Overrides *run_verifyta()* in *MPCsetup*.

Parameters

- **horizon** (*int*) – The interval duration for which Uppaal stratego synthesizes a control strategy each MPC step. Is given in the number of periods.
- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.
- **final** (*int*) – The time that should be reached by the synthesized strategy, given in Uppaal Stratego time units. Most likely this will be current time + *horizon* x *period*.
- ***args** – Is not used in this method; it is included here to safely override the original method.
- ****kwargs** – Is not used in this method; it is included here to safely override the original method.

6.4 Static methods

The folowing static methods are available.

strategoutil.**array_to_stratego**(*arr*)

Convert python array string to C style array used in UPPAAL Stratego. NB, does not include ';' in the end.

Parameters
arr (*str*) – The array string to convert.

Returns
An array string where "[" and "]" are replaced by "{" and "}", respectively.

Return type
str

strategoutil.**check_tool_existence**(*name*)

Check whether 'name' is on PATH and marked executable.

From https://stackoverflow.com/questions/11210104/check-if-a-program-exists-from-a-python-script.

Parameters
name (*str*) – the name of the tool.

Returns
True when the tool is found and executable, false otherwise.

Return type
bool

strategoutil.**extract_state**(*text*, *var*, *control_period*)

Extract the state from the Uppaal Stratego output at the end of the simulated control period.

Parameters

- **text** (*str*) – The input string containing the Uppaal Stratego output.
- **var** (*str*) – The variable name.
- **control_period** (*int*) – The interval duration after which the controller can change the control setting, given in Uppaal Stratego time units.

Returns
The value of the variable at the end of *control_period*.

Return type
float

strategoutil.**get_duration_action**(*tuples*, *max_time=None*)

Get tuples (duration, action) from tuples (time, variable) resulted from simulate query.

strategoutil.**get_float_tuples**(*text*)

Convert Stratego simulation output to list of tuples (float, float).

Parameters
text (*str*) – The input string containing the Uppaal Stratego output.

Returns
A list of tuples (float, float).

Return type
list

strategoutil.**get_int_tuples**(*text*)

Convert Stratego simulation output to list of tuples (int, int).

Parameters
text (*str*) – The input string containing the Uppaal Stratego output.

Returns
A list of tuples (int, int).

Return type
list

strategoutil.**insert_to_modelfile**(*model_file*, *tag*, *inserted*)

Replace tag in model file by the desired text.

Parameters

- **model_file** (*str*) – The file name of the model.
- **tag** (*str*) – The tag to replace.
- **inserted** (*str*) – The value to replace the tag with.

strategoutil.**merge_verifyta_args**(*cfg_dict*)

Concatenate and format a string of verifyta arguments given by the configuration dictionary.

Parameters
cfg_dict (*dict*) – The configuration dictionary.

Returns
String containing all arguments from the configuration dictionary.

Return type
str

strategoutil.**print_progress_bar**(*i*, *max*, *post_text*)

Print a progress bar to sys.stdout.

Subsequent calls will override the previous progress bar (given that nothing else has been written to sys.stdout).

From https://stackoverflow.com/a/58602365.

Parameters

- **i** (*int*) – The number of steps already completed.
- **max** (*int*) – The maximum number of steps for process to be completed.
- **post_text** (*str*) – The text to display after the progress bar.

strategoutil.**run_stratego**(*model_file*, *query_file=''*, *learning_args=None*, *verifyta_command='verifyta'*)

Run command line version of Uppaal Stratego.

Parameters

- **model_file** (*str*) – The file name of the model.
- **query_file** (*str*) – The file name of the query.
- **learning_args** (*dict*) – Dictionary containing the learning parameters and their values. The learning parameter names should be those used in the command line interface of Uppaal Stratego. You can also include non-learning command line parameters in this dictionary. If a non-learning command line parameter does not take any value, include the empty string "" as value.
- **verifyta_command** (*str*) – The command name for running Uppaal Stratego at the user's machine.

Returns

The output as produced by Uppaal Stratego.

Return type

str

strategoutil.**successful_result**(*text*)

Verify whether the stratego output is based on the successful synthesis of a strategy.

Parameters

text (*str*) – The output generated by Uppaal Stratego.

Returns

Whether Uppaal Stratego has successfuly ran all queries.

Return type

bool

CHAPTER

SEVEN

GETTING STARTED

1) Use pip or clone this git repo to install *strategoutil* to your environment

```
pip install strategoutil
```

or

```
git clone https://github.com/DEIS-Tools/strategoutil.git
cd strategoutil
pip install -e .
```

2) Look how *strategoutil* is used in *Examples of using the STOMPC tool*.

CHAPTER

EIGHT

FUNCTIONALITY

Currently, *strategoutil* contains the tool *STOMPC* that is capable of performing the following actions:

- Write input variables to Stratego model **.xml* files
- Parse outputs of *simulate* queries to get timeseries of important variables
- Run *verifyta* with chosen query **.q* and run parameters
- Create model predictive control (MPC) routines where plant is either defined within the same Stratego model, or plant is defined as external process, simulataor, etc.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX