STOMPC: Stochastic Model-Predictive Control with Uppaal Stratego *

Martijn A. Goorden[1][0000−0002−0641−7240], Peter G. Jensen[1][0000−0002−9320−9991], Kim G. Larsen[1], Mihhail Samusev[1,2], Jiří Srba[1], and Guohan Zhao[2]

[1] Deparment of Computer Science, Aalborg University, Aalborg, Denmark
{mgoorden,pgj,kgl,srba}@cs.aau.dk
[2] Deparment of the Built Environment, Aalborg University, Aalborg, Denmark
{msam,guohanz}@build.aau.dk

Abstract. We present the new co-simulation and synthesis integrated-framework STOMPC for stochastic model-predictive control (MPC) with Uppaal Stratego. The framework allows users to easily set up MPC designs, a widely accepted method for designing software controllers in industry, with Uppaal Stratego as the controller synthesis engine, which provides a powerful tool to synthesize safe and optimal strategies for hybrid stochastic systems. STOMPC provides the user freedom to connect it to external simulators, making the framework applicable across multiple domains.

1 Introduction

Controller software has become increasingly dominant in cyber-physical systems. Functionality that previously was implemented by hardware is now being shifted towards software. Often cyber-physical systems are safety-critical, hence strong safety-related requirements are formulated for them. At the same time, quality objectives need to be considered, such as being as fast as possible or minimizing resource usage. Designing safe and optimal controller software manually is a challenge, and several formal methods have been developed to synthesize controller strategies automatically [1, 14, 15].

For stochastic hybrid systems, the tool Uppaal Stratego [5,10] is the newly emerged branch of the leading tool Uppaal that can automatically synthesize safe and near-optimal controller strategies. It combines statistical model checking, synthesis for timed games, and reinforcement learning. Uppaal Stratego has been applied successfully to several case studies [3, 6, 8, 11, 12].

Within industry, model predictive control (MPC) is a widely adopted method for designing controllers [7]. MPC schemes are popular as they yield high-performing control systems without expert intervention over long periods of time. This is achieved by periodically using a model to predict the system's

* This work is partly supported by the Villum Synergy project CLAIRE and the ERC Advanced Grant LASSO.

future behavior and calculate an optimal control strategy for the next time-bounded period [4]. Therefore, MPC schemes are also called *online control*, as they can adapt control strategies while the system is running.

Uppaal Stratego conceptually fits well within MPC designs. Yet it lacks the ability to periodically update the model's state and synthesize a new strategy. In previous work [11], bash scripts are created utilizing the command line interface of Uppaal Stratego to do all the calculations periodically. Unfortunately, these bash scripts are very case specific and not well adaptable to other case studies. Furthermore, we noticed that for each new case study, researchers were repeatedly rediscovering MPC schemes for Uppaal Stratego.

We present the co-simulation and synthesis integrated-framework STOMPC, which implements a basic MPC scheme using Uppaal Stratego as the core engine for synthesizing the strategies. With this framework, we aim to greatly simplify the setup for different case studies by implementing standard functionalities for MPC schemes with Uppaal Stratego in Python classes. Furthermore, STOMPC can be connected to external, domain specific, simulators (or in fact again Uppaal Stratego) that represent the real world. This makes the framework applicable to cases from different domains. Our framework is accessible on GitHub[3], can be installed through pip, and its documentation is available[4]. An artifact for evaluation can be downloaded from Zenodo[5].

2 Framework Overview

MPC captures a particular way of designing controllers for a broad range of systems and processes. It has the following three characteristics [4]: a model, which is used to predict the future of the system within a certain horizon, the calculation of a control sequence (or strategy) that optimizes some objective, and a receding approach, where all calculations are repeated after executing the first control action from the sequence and observing the true state as a consequence of that.

Fig. 1 provides a conceptual overview of the key ingredients of MPC that are implemented by STOMPC. Up to time $t = k$, we have observed the true state of the system x and provided control input u to it. Using a model of the system, we can predict the future state $\hat{x}_k$ within the control horizon. The evolution of the state depends on the control sequence being applied $\hat{u}_k$, where the applied control action can be switched after each control period. To determine which control sequence to choose, the objective is optimized. Often the objective is to minimize the difference between the state of the system and a reference signal.

Once the optimal control sequence is obtained, the first control action of this sequence is applied. When the end of the control period is reached, the process mentioned above is repeated. At time $t = k + p$, where p is the duration of the control period, the true value of the state of the system $x(k + p)$ is observed,

[3] `https://github.com/DEIS-Tools/strategoutil`
[4] `https://strategoutil.readthedocs.io/en/latest/`
[5] `https://doi.org/10.5281/zenodo.6519909`

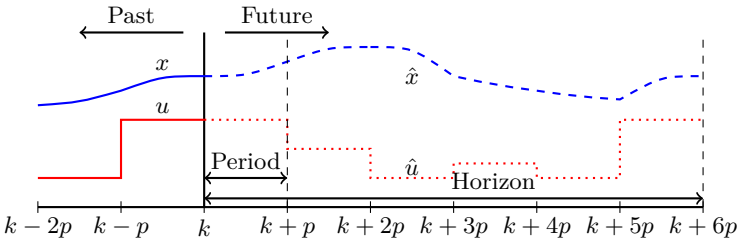

Fig. 1. Conceptual overview of model predictive control. In blue (dashed line) is the continuous evolution of the state in the past x and for the future $\hat{x}$, while red (dotted line) shows the periodically switched control signal in the past u and for the future $\hat{u}$.

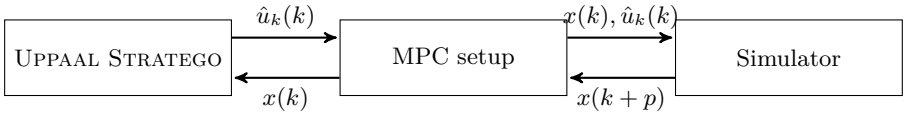

Fig. 2. Global architecture of STOMPC, where the MPC setup starts a new step at time $t = k$. After each step, k is replaced by $k + p$ and everything is repeated.

which, most likely, is different from the predicted state $\hat{x}_k(k+p)$. Repeating the calculation with the new true state $x(k + p)$ might result in a different control sequence $\hat{u}_{k+p}$ than the one calculated before $\hat{u}_k$.

STOMPC implements this MPC scheme using Python, hiding as much details as possible, such that a user can focus more on the application itself. Fig. 2 shows the architecture of STOMPC. It provides the component MPC setup, which orchestrates the MPC scheme. At time $t = k$ for some k, it supplies the current true state of the system $x(k)$ to Uppaal Stratego. It does this by inserting the state values into the Uppaal Stratego model. Subsequently, the MPC setup runs Uppaal Stratego with this model to calculate the optimal control strategy. From the report generated by Uppaal Stratego, the MPC setup identifies the calculated control action $\hat{u}_k(k)$ for the next control period.

After this, the MPC setup switches to the simulator. This simulator can be again Uppaal Stratego or an external, domain specific one (see Section 3 for examples), or the actual physical system. The MPC setup supplies the simulator with the calculated control action $\hat{u}_k(k)$ for the next control period and, for memory-less simulators, also the last recorded true state $x(k)$ from which the simulator should continue. Subsequently, the simulator returns the true state $x(k+p)$ at the end of the control period. After that, the above procedure repeats until the end of the experiment.

More information on the setup of the tool, including a detailed example, can be found in the tool's documentation[6].

[6] `https://strategoutil.readthedocs.io`

3 Use Cases

An advantage of STOMPC is its general applicability across different application domains. We now discuss three use cases from different application domains: floorheating in a family house, storm water detention ponds, and traffic light control.

3.1 Floorheating in a Family House

The MPC scheme from Section 2 is in collaboration with the company Seluxit applied to controlling floor heating in a family house located in Northern Jutland, Denmark. Fig. 3 shows a screenshot of a digital twin of the house, displaying all its 10 rooms and the water pipes supplying heat to the rooms. Each room has its individually controlled target temperature (the upper digits in the rooms) and the thermodynamic equations used in the model consider the heat exchange between the rooms, between the rooms and their outside envelope, as well as the heat exchange from the water pipes passing under rooms.

In each 15 minute period, temperature sensors in each room report the current readings to the central control unit. During the following 15 minutes, the server gathers a 24-hour weather forecast and computes an optimal control strategy for the next 75 minutes using Uppaal Stratego. The computed strategy optimizes the comfort in each room.

Fig. 3. Digital twin of a floor heating system

Simulations on the digital twin using the Uppaal Stratego online controller (where the real house behavior is replaced by a Simulink model) show an average 40% improvement in comfort, compared to the controller that was used in the house before. As a side effect of the predictive control, the new Uppaal Stratego control saves about 10% of energy. Further details about this concrete application of MPC can be found in [2, 11].

3.2 Stormwater Detention Ponds

Stormwater Detention Ponds are critical real-time control assets in urban stormwater management systems. They reduce the considerable hydraulic impact towards the natural stream, as well as avoid significant pollutant loads being discharged. However, only passive control of the stormwater pond outlet valves is currently used in Danish engineering practice.

We implement a co-simulation by combining Uppaal Stratego with the domain specific simulator EPA-SWMM [9], as shown in Fig. 4. EPA-SWMM is an open-source physical-based dynamic rainfall-runoff model that has been implemented for decades in the urban stormwater management [9].

Pyswmm [13], a python interface wrapper, is used for the interfacing of EPA-SWMM with STOMPC. In each 15 minute control period, EPA-SWMM extracts the current water level in stormwater ponds, and feeds it towards Uppaal Stratego. From thereon until the end of the upcoming control horizon (48 hours), Uppaal Stratego synthesizes the optimal control strategy for the outlet valves taking weather forecasting data into account. Two objectives are involved: guarantee the safe operation of the stormwater pond without any overflow and maximize the sedimentation process to improve the water quality. Our approach increased the control performance by 22%. Further details can be found in [8].

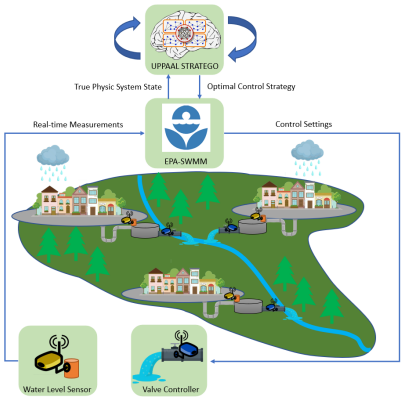

Fig. 4. Digital twin of an urban stormwater management system

3.3 Traffic Light Control

The application of MPC is widespread in the domain of traffic control. Recently Uppaal Stratego has been successfully used to minimize the delays, queue lengths, number of stops, and fuel consumption of vehicles traveling on the arterial street Hobrovej in Aalborg simulated in VISSIM [6]. The street consists of 4 signalized intersections as shown in Fig. 5. The original traffic light controllers are pre-timed or detector time-gap based.

Every second Uppaal Stratego is called to solve a traffic light configuration sequence planning problem that minimizes the total intersection delay. The vehicle information communicated to Uppaal Stratego are the estimated times of arrival extracted from VISSIM's area sensors for each vehicle within 200m of the intersection. The first step in the resulting optimal control sequence is then sent back to VISSIM. Compared to the original control, and considering an intersection with smallest improvements, the described MPC approach manages to reduce the delays by 27%, queue lengths by 42%, number of stops by 20% and fuel consumption by 19%.

Fig. 5. Intersections optimized by Uppaal Stratego at Hobrovej, Aalborg

In the original paper the data exchange between Uppaal Stratego and VISSIM was established using a Python script. STOMPC can with minimal adjustments wrap the complexity of the communication between those two pieces

of software and let the user focus on the more high-level problems such as the definition of input data, objective function, and MPC parameters.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable specifications of reactive systems. In: ICALP. pp. 1–17. Springer (1989)
2. Agesen, M., Larsen, K., Mikucionis, M., Muniz, M., Olsen, P., Pedersen, T., Srba, J., Skou, A.: Toolchain for user-centered intelligent floor heating control. In: IECON. pp. 5296–5301. IEEE (2016). https://doi.org/10.1109/IECON.2016.7794040
3. Ashok, P., Křetínský, J., Larsen, K.G., Le Coënt, A., Taankvist, J.H., Weininger, M.: SOS: Safe, optimal and small strategies for hybrid markov decision processes. In: Parker, D., Wolf, V. (eds.) QEST. pp. 147–164. LNCS (2019). https://doi.org/10.1007/978-3-030-30281-8_9
4. Camacho, E.F., Alba, C.B.: Model predictive control. Springer (2013)
5. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: Baier, C., Tinelli, C. (eds.) TACAS. pp. 206–211. LNCS (2015). https://doi.org/10.1007/978-3-662-46681-0_16
6. Eriksen, A., Lahrmann, H., Larsen, K., Taankvist, J.: Controlling signalized intersections using machine learning. Transportation Research Procedia **48**, 987–997 (2020). https://doi.org/10.1016/j.trpro.2020.08.127
7. García, C.E., Prett, D.M., Morari, M.: Model predictive control: Theory and practice – a survey. Automatica **25**(3), 335–348 (1989). https://doi.org/10.1016/0005-1098(89)90002-2
8. Goorden, M.A., Larsen, K.G., Nielsen, J.E., Nielsen, T.D., Rasmussen, M.R., Srba, J.: Learning safe and optimal control strategies for storm water detention ponds. IFAC-PapersOnLine **54**(5), 13–18 (2021). https://doi.org/10.1016/j.ifacol.2021.08.467
9. Huber, W.C., Rossman, L.A., Dickinson, R.E.: Epa storm water management model, swmm5. Watershed models **338**, 359 (2005)
10. Jaeger, M., Jensen, P.G., Larsen, K.G., Legay, A., Sedwards, S., Taankvist, J.H.: Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In: Chen, Y.F., Cheng, C.H., Esparza, J. (eds.) ATVA. pp. 81–97. LNCS (2019). https://doi.org/10.1007/978-3-030-31784-3_5
11. Larsen, K.G., Mikučioni, M., Muñiz, M., Srba, J., Taankvist, J.H.: Online and compositional learning of controllers with application to floor heating. In: Chechik, M., Raskin, J.F. (eds.) TACAS. pp. 244–259. LNCS (2016). https://doi.org/10.1007/978-3-662-49674-9_14
12. Larsen, K.G., Mikučioni, M., Taankvist, J.H.: Safe and optimal adaptive cruise control. In: Meyer, R., Platzer, A., Wehrheim, H. (eds.) Olderog-Festschrift, pp. 260–277. LNCS, Springer (2015). https://doi.org/10.1007/978-3-319-23506-6_17
13. McDonnell, B.E., Ratliff, K., Tryby, M.E., Wu, J.J.X., Mullapudi, A.: Pyswmm: The python interface to stormwater management model (swmm). Journal of Open Source Software **5**(52), 2292 (2020). https://doi.org/10.21105/joss.02292
14. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: ICALP. pp. 652–671. Springer (1989)
15. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM journal on control and optimization **25**(1), 206–230 (1987)

A Demonstration with the storm water detention pond

In this appendix, we demonstrate the usage of the framework with the storm water detention pond case.

A.1 Installing the framework

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

- *STOMPC*, the framework as described in this paper,
- 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.

Both STOMPC and pySWMM are available though 'pip'[7]:

```
$ pip install strategoutil
$ pip install pyswmm
```

Uppaal Stratego itself needs to be downloaded manually. A detailed installation guide is available in the STOMPC documentation at `https://strategoutil.readthedocs.io/en/latest/installation.html#uppaal-stratego`.

A.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 [8] 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 the STOMPC framework 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]
```

[7] While writing the paper, the name of the framework has changed from StrategoUtil to STOMPC. In pip the framework is currently still accessible through the old name.

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 framework which variables it needs 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:

```
t: 0.0
rain: 0.0
S_UC: 0.0
w: 0.0
c: 0.0
Open : 1.0
o: 0.0
Rain.rainLoc: 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 arbitrary numbers that ensure fast calculations). In Section A.4 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:
```

A.3 Specializing the SafeMPCSetup class from STOMPC

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

- `MPCSetup`. This class is the primary class an end-user should specialize for his or her case. It implements the basic MPC scheme as explained in Section 2.

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`.

```
1  import strategoutil as stompc
2  import weather_forecast_generation as weather
3  import datetime
4
5  class MPCSetupPond(stompc.SafeMPCSetup):
6    def create_query_file(self, horizon, period, final):
7      """
8      Create the query file for each step of the pond model.
9      Current content will be overwritten.
10
11     Overrides SafeMPCsetup.create_query_file().
12     """
13     with open(self.queryfile, "w") as f:
14       line1 = f"strategy opt = minE (c) [<={horizon}*{period}]:
15                <> (t=={final} && o <= 0)\n"
16       f.write(line1)
17       f.write("\n")
18       line2 = f"simulate 1 [<={period}+1] {{
19                 {self.controller.get_var_names_as_string()} }}
20                 under opt\n"
21       f.write(line2)
22
23   def create_alternative_query_file(self, horizon, period,
24                                     final):
25     """
26     Create an alternative query file in case the original
27     query could not be satisfied by Stratego, i.e., it could
28     not find a strategy. Current content will be overwritten.
29
30     Overrides SafeMPCsetup.create_alternative_query_file().
31     """
32     with open(self.queryfile, "w") as f:
```

```
33          line1 = f"strategy opt = minE (w) [<={horizon}*{period}]:
34                   <> (t=={final})\n"
35          f.write(line1)
36          f.write("\n")
37          line2 = f"simulate 1 [<={period}+1] {{
38                    {self.controller.get_var_names_as_string()} }}
39                    under opt\n"
40          f.write(line2)
41
42      def perform_at_start_iteration(self, controlperiod, horizon,
43                                     duration, step, **kwargs):
44          """
45          Performs some customizable preprocessing steps at the
46          start of each MPC iteration.
47
48          Overrides SafeMPCsetup.perform_at_start_iteration().
49          """
50          current_date = kwargs["start_date"] +
51                         datetime.timedelta(hours=step)
52          weather.create_weather_forecast(
53                    kwargs["historical_rain_data_path"],
54                    kwargs["weather_forecast_path"], current_date,
55                    horizon * controlperiod, kwargs["uncertainty"])
```

In method `create_query_file` we specify the strategy synthesis query. For the pond case, we have this defined at line 14-15. 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.

A.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 -8 -7"

  # 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,
                            verifyta_command=verifyta_command,
                            external_simulator=False,
```

```
                                    action_variable=action_variable,
                                    debug=debug)
```

A.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)
```

```
35        water_depth.append(pond.depth)
36
37        # Get and set the control parameter for the next period.
38        orifice.target_setting = get_control_strategy(pond.depth,
39            current_time, controller, period, horizon,
40            rain_data_file, weather_forecast_path, uncertainty)
41        orifice_settings.append(orifice.target_setting)
42
43    # Write results to file.
44    with open(swmm_results, "w") as f:
45      writer = csv.writer(f)
46      for i, j, k in zip(time_series, water_depth,
47                         orifice_settings):
48        i = i.strftime('%Y-%m-%d %H:%M')
49        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.

```
1  def get_control_strategy(current_water_level, current_time,
2                           controller, period, horizon,
3                           rain_data_file, weather_forecast_path,
4                           uncertainty):
5    controller.controller.update_state(
6      {'w':current_water_level * 100}) # Conversion from m to cm
7    control_setting = controller.run_single(period, horizon,
8        start_date=current_time,
9        historical_rain_data_path=rain_data_file,
10       weather_forecast_path=weather_forecast_path,
11       uncertainty=uncertainty)
12
13   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.

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