A Tour of Breach API

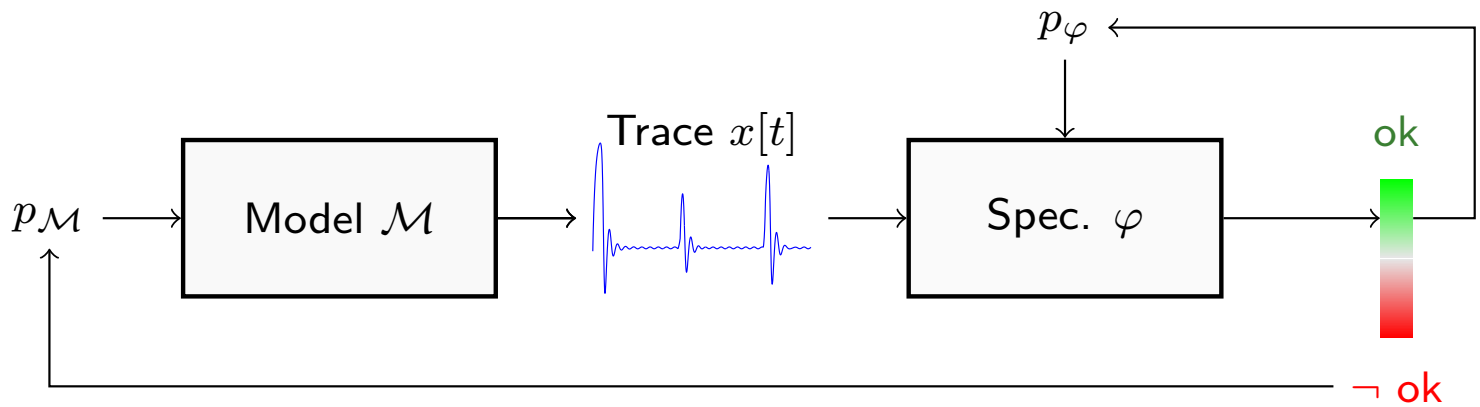
Version 1.0-beta

Alexandre Donzé

University of California, Berkeley

March 28, 2016

Simulation-Based Design with Formal Specifications



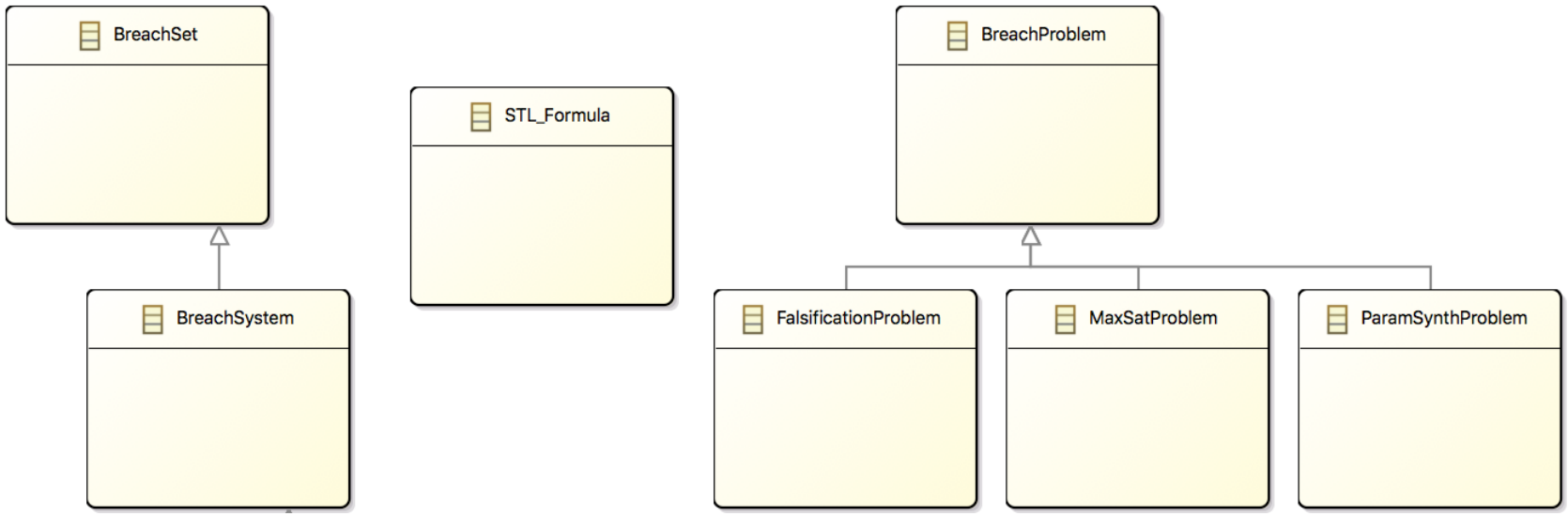
Given a model $\mathcal{M}$ producing simulation traces $t \mapsto x[t]$,

Breach can solve various problems involving

- model parameters $p_{\mathcal{M}}$,
- specifications φ expressed in Signal Temporal Logic (STL),
- and specification parameters p_φ.

A good deal of it uses **quantitative satisfaction** of φ by x.

Architecture



- **BreachSets/BreachSystem**: main interface with model and simulator. Defines sets of parameters, store/run/plot traces, etc.
- **STL_Formula**: class for (P)STL formulas
- **BreachProblem**: maps parameters, models and specifications to generic optimization problem

Interfacing a Simulink Model

Input Generation

Sets of parameters and traces

Signal Temporal Logic (STL) specifications

Breach Problems
- Falsification
- Parameter Synthesis
- Maximizing Satisfaction
- Requirement Mining

Sensitivity Analysis (work-in-progress)

Interfacing a Simulink Model

Input Generation

Sets of parameters and traces

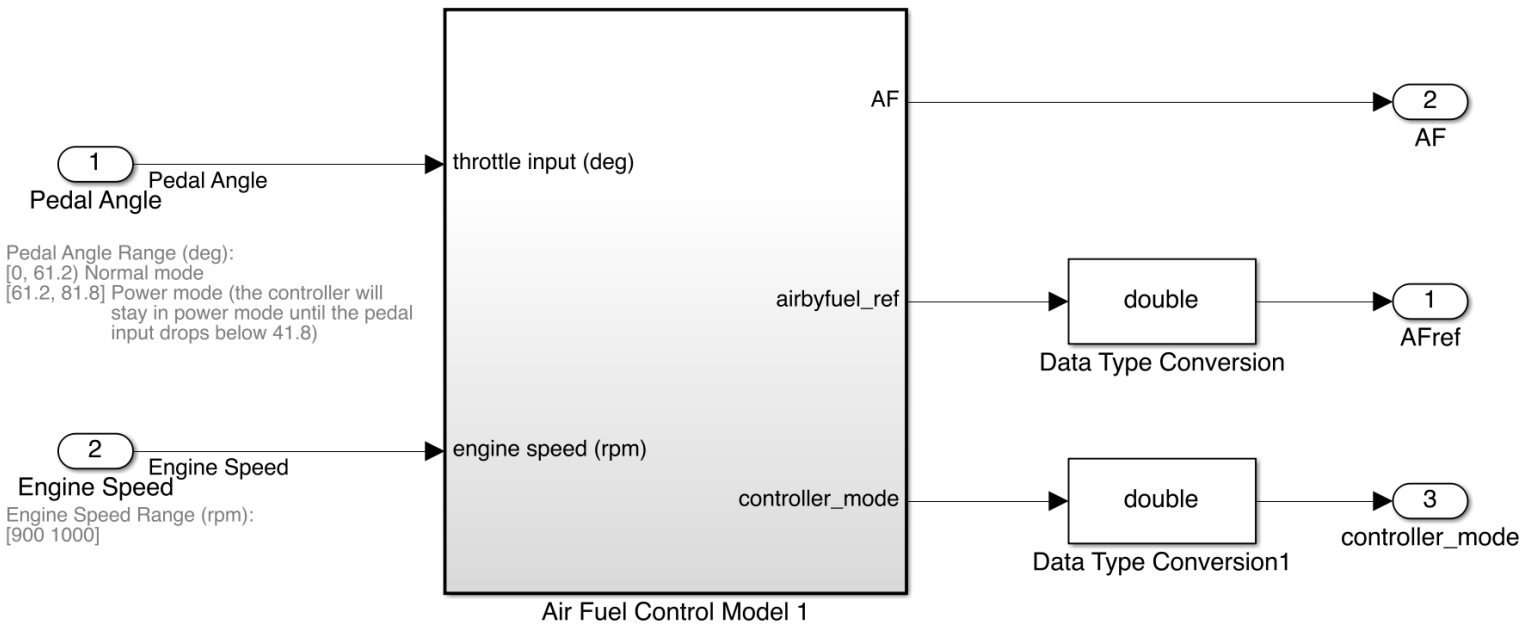
Signal Temporal Logic (STL) specifications

Breach Problems
- Falsification
- Parameter Synthesis
- Maximizing Satisfaction
- Requirement Mining

Sensitivity Analysis (work-in-progress)

Interfacing a Simulink Model

where model is:



Interfacing a Simulink model with Breach

Breach can interact with a Simulink model by

- Changing parameters defined in the base workspace
- Generating input signals
- Running simulations and collecting signals for analysis

We use the model `AbstractFuelControl_M1` as running example.

Interfacing a Simulink model with Breach

Breach can interact with a Simulink model by

- Changing parameters defined in the base workspace
- Generating input signals
- Running simulations and collecting signals for analysis

We use the model `AbstractFuelControl_M1` as running example.

First, we initialize Breach and instantiate parameters for the model to run properly:

```
InitBreach;
fuel_inj_tol = 1.0;
MAF_sensor_tol = 1.0;
AF_sensor_tol = 1.0;
pump_tol = 1.;
kappa_tol=1;
tau_ww_tol=1;
fault_time=50;
kp = 0.04;
ki = 0.14;
```

Interfacing a Simulink model with Breach

Breach can interact with a Simulink model by

- Changing parameters defined in the base workspace
- Generating input signals
- Running simulations and collecting signals for analysis

We use the model `AbstractFuelControl_M1` as running example.

First, we initialize Breach and instantiate parameters for the model to run properly:

```
InitBreach;
fuel_inj_tol = 1.0;
MAF_sensor_tol = 1.0;
AF_sensor_tol = 1.0;
pump_tol = 1.;
kappa_tol=1;
tau_ww_tol=1;
fault_time=50;
kp = 0.04;
ki = 0.14;
```

```
Initializing Breach...
```

They will be discovered automatically and included in the interface.

Creating an Interface Object

Running the following will create a BreachSystem interface with the model:

```
mdl = 'AbstractFuelControl_M1';
BrAFC = BreachSimulinkSystem(mdl)
```

Creating an Interface Object

Running the following will create a BreachSystem interface with the model:

```
mdl = 'AbstractFuelControl_M1';
BrAFC = BreachSimulinkSystem(mdl)
```

```
BrAFC =

BreachSystem with name AbstractFuelControl_M1_breach.
```

Note that Breach created a copy of the model in `AbstractFuelControl_M1_breach`.

```
% We can check which parameters are interfaced via the following command:
BrAFC.PrintParams()
```

Creating an Interface Object

Running the following will create a BreachSystem interface with the model:

```
mdl = 'AbstractFuelControl_M1';
BrAFC = BreachSimulinkSystem(mdl)
```

```
BrAFC =

BreachSystem with name AbstractFuelControl_M1_breach.
```

Note that Breach created a copy of the model in `AbstractFuelControl_M1_breach`.

```
% We can check which parameters are interfaced via the following command:
BrAFC.PrintParams()
```

```
Parameters:
-----------
AF_sensor_tol=1
MAF_sensor_tol=1
fault_time=50
fuel_inj_tol=1
kappa_tol=1
ki=0.14
kp=0.04
pump_tol=1
tau_ww_tol=1
Pedal_Angle_u0=0
Engine_Speed_u0=0
```

Creating an Interface Object

Sometimes a model contains many tunable parameters. In that case, we can explicitly specify those we want to tune, e.g., some PI parameters:

Creating an Interface Object

Sometimes a model contains many tunable parameters. In that case, we can explicitly specify those we want to tune, e.g., some PI parameters:

```
BrAFC_less_params = BreachSimulinkSystem(mdl, {'ki', 'kp'}, [0.14 0.04]);
BrAFC_less_params.PrintParams
```

Creating an Interface Object

Sometimes a model contains many tunable parameters. In that case, we can explicitly specify those we want to tune, e.g., some PI parameters:

```
BrAFC_less_params = BreachSimulinkSystem(mdl, {'ki', 'kp'}, [0.14 0.04]);
BrAFC_less_params.PrintParams
```

```
Parameters:
-----------
ki=0.14
kp=0.04
Pedal_Angle_u0=0
Engine_Speed_u0=0
```

Interfacing Signals

By default, Breach collects the following signals:

- Signals attached to input ports
- Signals attached to output ports
- Logged signals

We can check which signals are interfaced via the following command:

```
BrAFC.PrintSignals()
```

Interfacing Signals

By default, Breach collects the following signals:

- Signals attached to input ports
- Signals attached to output ports
- Logged signals

We can check which signals are interfaced via the following command:

```
BrAFC.PrintSignals()
```

```
Signals:
________

AFref
AF
controller_mode
cyl_air
manifold_pressure
MAF
cyl_fuel
Engine_Speed
Pedal_Angle
```

Note that signal and parameter names in the model should remain simple. Preferably they shoul be valid variable id names, i.e., using only underscores and alphanumerical characters (a-z, A-Z, 0-9 or '_').

Input Signals

Input signals of different types can be generated. They are defined by parameters of the form `input1_u0`, `input1_u1` etc.

By default, input signals are simply constant and we have two inputs, which are `Engine_Speed` and `Pedal_Angle`. Next, we set their constant values:

```
BrAFC.SetParam('Engine_Speed_u0',1000)
BrAFC.SetParam('Pedal_Angle_u0',30)
BrAFC.PrintParams()
```

Input Signals

Input signals of different types can be generated. They are defined by parameters of the form `input1_u0`, `input1_u1` etc.

By default, input signals are simply constant and we have two inputs, which are `Engine_Speed` and `Pedal_Angle`. Next, we set their constant values:

```
BrAFC.SetParam('Engine_Speed_u0',1000)
BrAFC.SetParam('Pedal_Angle_u0',30)
BrAFC.PrintParams()
```

```
Parameters:
-----------
AF_sensor_tol=1
MAF_sensor_tol=1
fault_time=50
fuel_inj_tol=1
kappa_tol=1
ki=0.14
kp=0.04
pump_tol=1
tau_ww_tol=1
Pedal_Angle_u0=30
Engine_Speed_u0=1000
```

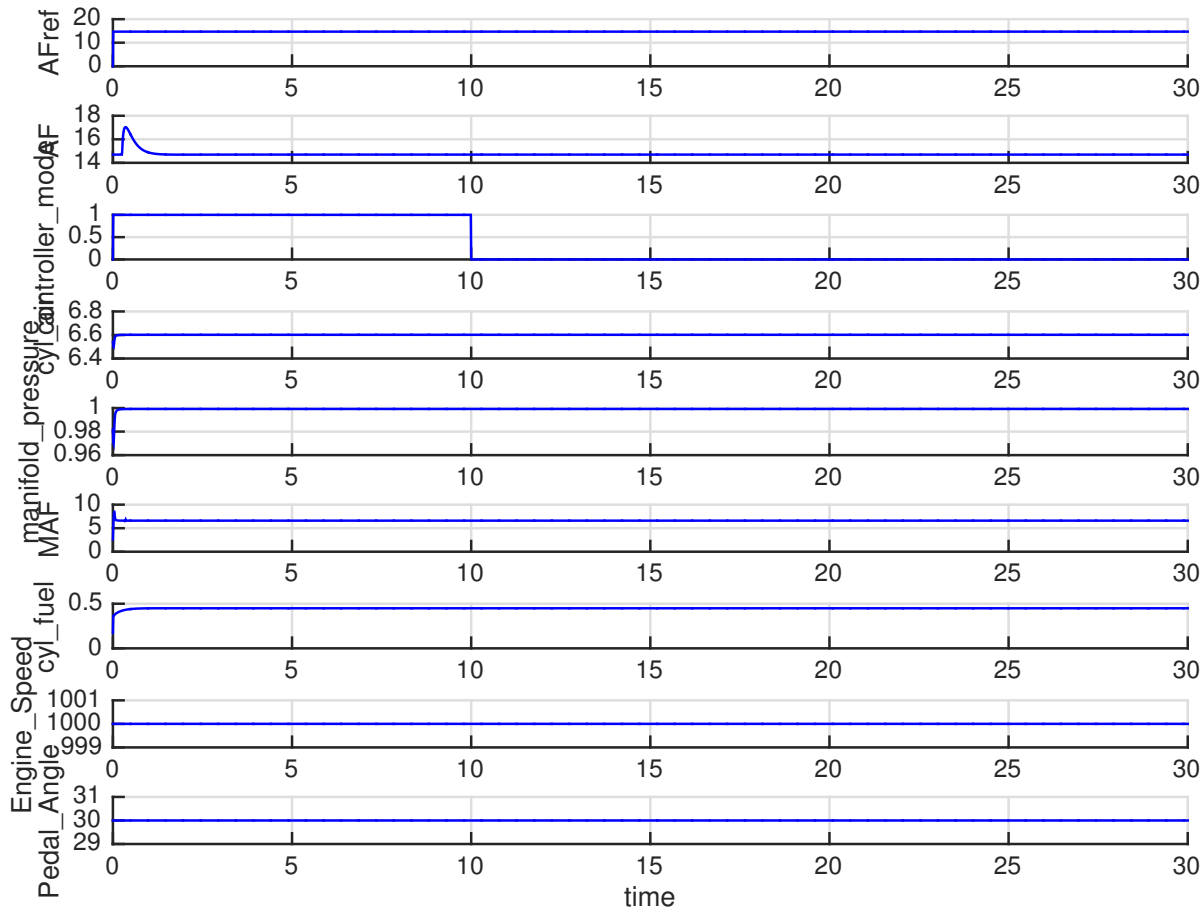
We can now run simulations.

Simulation with Nominal Parameters

```
AFC_Nominal= BrAFC.copy(); % Creates a copy of the interface object
AFC_Nominal.Sim(0:.01:30); % Run Simulink simulation until time=30s
AFC_Nominal.PlotSignals(); % plots the signals collected after the simulation
```

Simulation with Nominal Parameters

```
AFC_Nominal= BrAFC.copy(); % Creates a copy of the interface object
AFC_Nominal.Sim(0:.01:30); % Run Simulink simulation until time=30s
AFC_Nominal.PlotSignals(); % plots the signals collected after the simulation
```



Simulation with Nominal Parameters

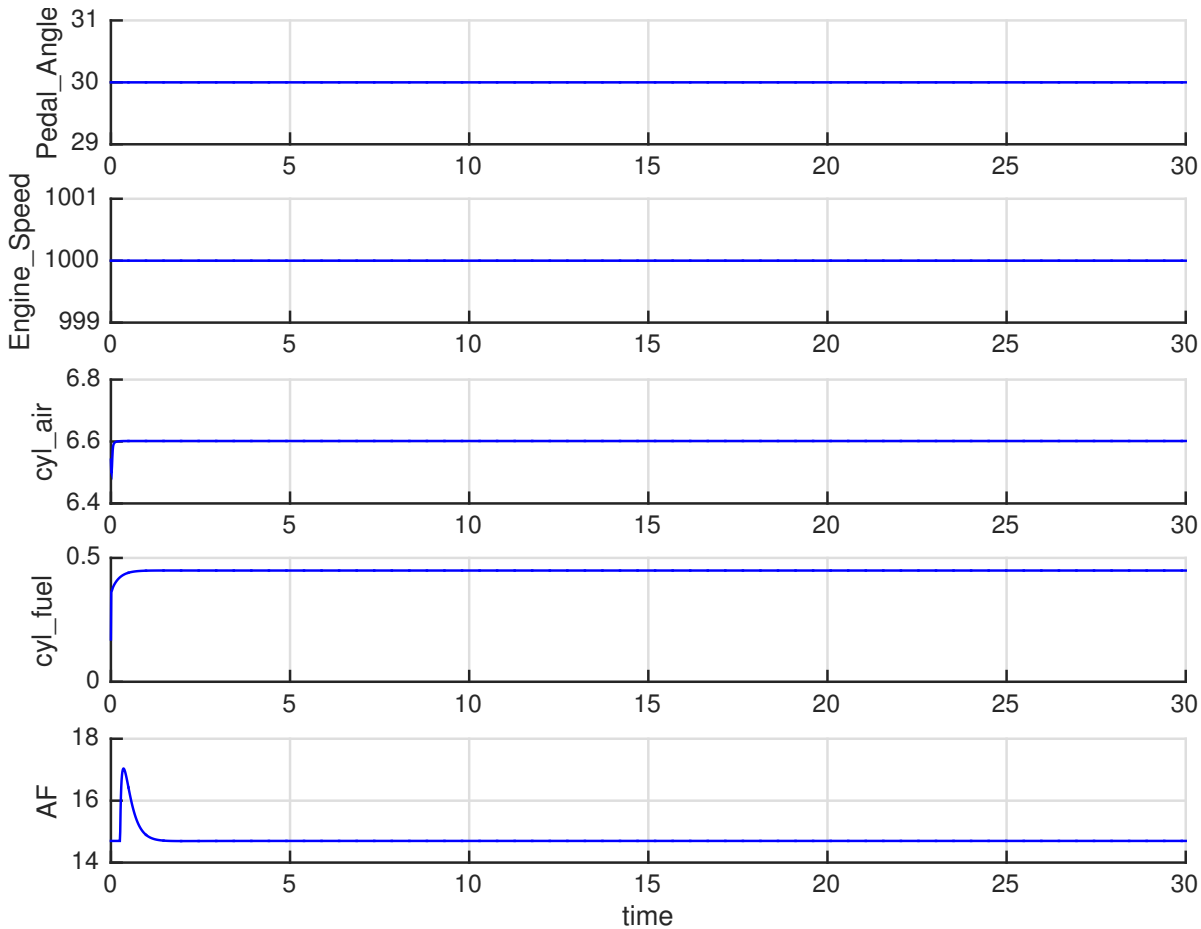
The plotting methods accepts a number of arguments. The first one and most useful simply allows to select which signals to plot.

```
AFC_Nominal.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```

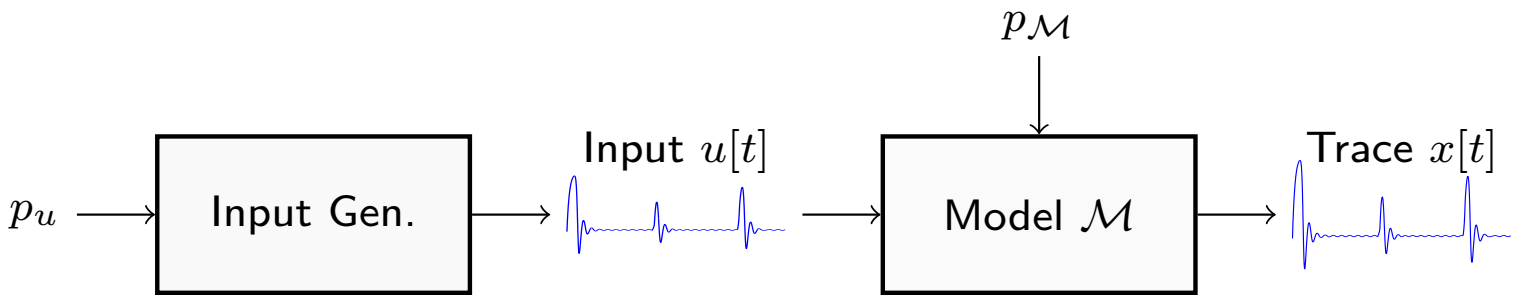
Simulation with Nominal Parameters

The plotting methods accepts a number of arguments. The first one and most useful simply allows to select which signals to plot.

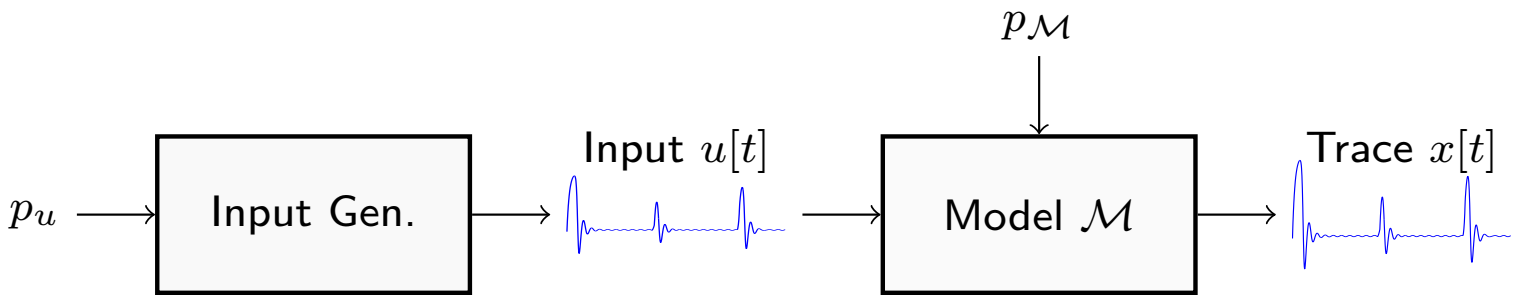
```
AFC_Nominal.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```



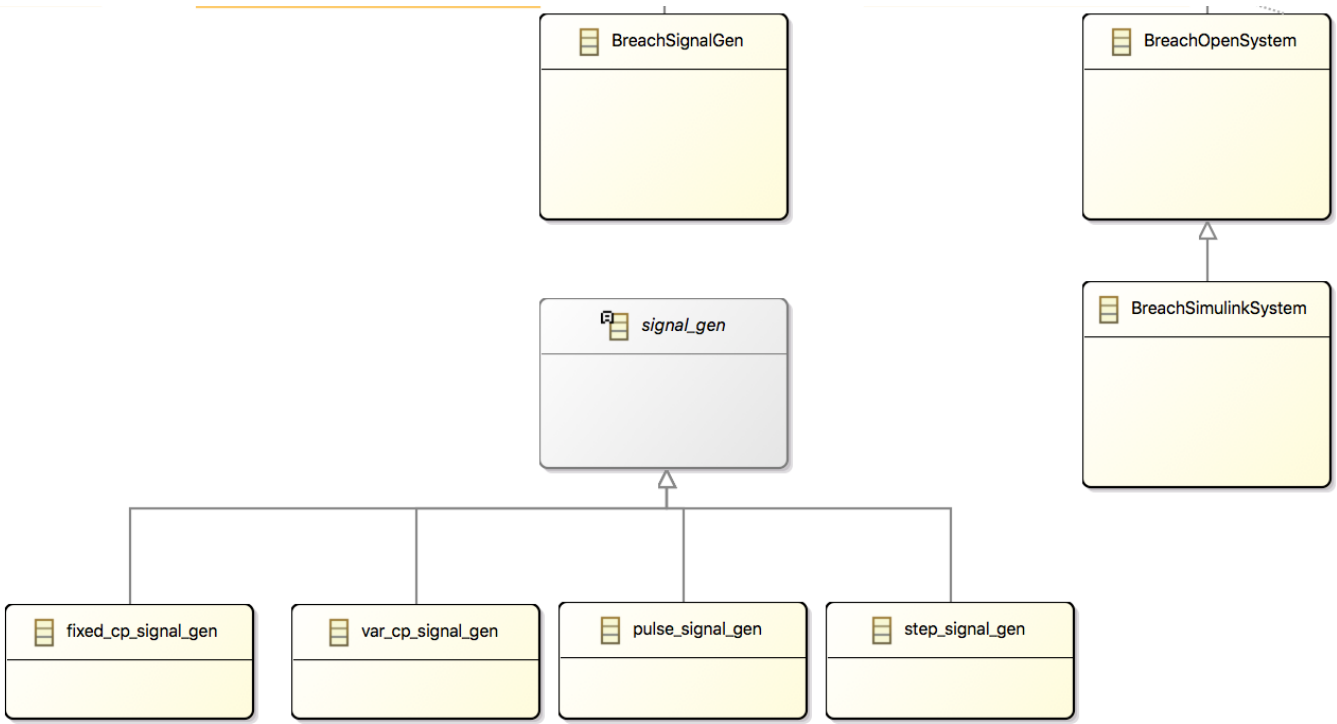
Simulation and Input Generation



Simulation and Input Generation



Corresponding Breach classes:



Using input signals

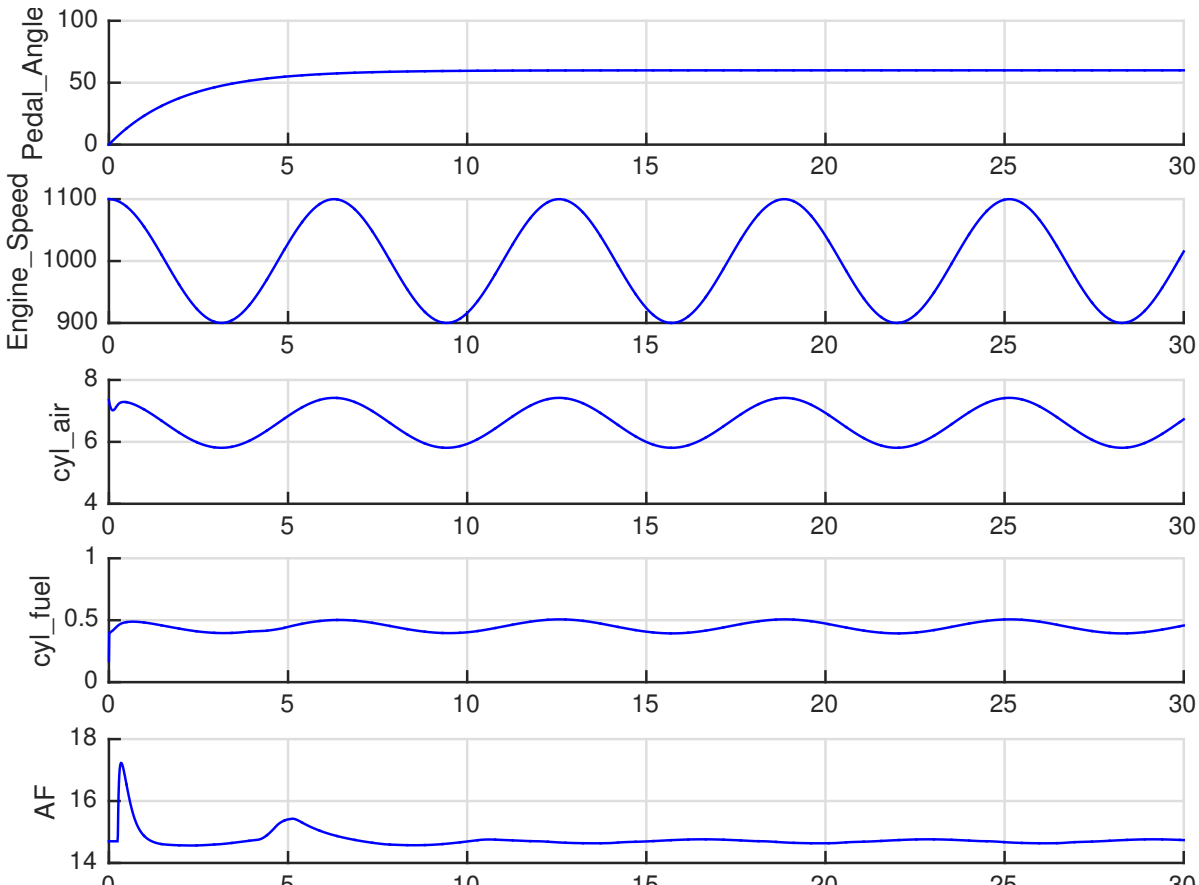
We can provide non-constant inputs to the system by directly giving them as argument to the sim command.

```
time_u = 0:.1:30;
pedal_angle = 60 - 60*exp(-0.5*time_u);
engine_speed = 100*cos(time_u) + 1000;
U = [time_u' pedal_angle' engine_speed'];  % order matters!

AFC_In = BrAFC.copy();
AFC_In.Sim(0:.01:30,U);
AFC_In.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```

Using input signals

We can provide non-constant inputs to the system by directly giving them as argument to the sim command.

```
time_u = 0:.1:30;
pedal_angle = 60 — 60*exp(—0.5*time_u);
engine_speed = 100*cos(time_u) + 1000;
U = [time_u' pedal_angle' engine_speed'];  % order matters!

AFC_In = BrAFC.copy();
AFC_In.Sim(0:.01:30,U);
AFC_In.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```



Fixed Step Input Generation

To interact with the system we can also use parameterized input generators. For instance, parameters can represent control points, and the input generated by some interpolation between these points.

```
AFC_UniStep3 = BrAFC.copy();  % Creates a copy of the system interface
input_gen.type = 'UniStep';   % uniform time steps
input_gen.cp = 3;             % number of control points
AFC_UniStep3.SetInputGen(input_gen);
```

This created a signal generator parameterized with 3 control points for each input. The corresponding parameters have been added to the interface:

```
AFC_UniStep3.PrintParams();
```

Fixed Step Input Generation

To interact with the system we can also use parameterized input generators. For instance, parameters can represent control points, and the input generated by some interpolation between these points.

```
AFC_UniStep3 = BrAFC.copy();  % Creates a copy of the system interface
input_gen.type = 'UniStep';   % uniform time steps
input_gen.cp = 3;             % number of control points
AFC_UniStep3.SetInputGen(input_gen);
```

This created a signal generator parameterized with 3 control points for each input. The corresponding parameters have been added to the interface:

```
AFC_UniStep3.PrintParams();
```

```
Parameters:
__________
AF_sensor_tol=1
MAF_sensor_tol=1
fault_time=50
fuel_inj_tol=1
kappa_tol=1
ki=0.14
kp=0.04
pump_tol=1
tau_ww_tol=1
Pedal_Angle_u0=0
Pedal_Angle_u1=0
Pedal_Angle_u2=0
Engine_Speed_u0=0
Engine_Speed_u1=0
Engine_Speed_u2=0
```

Fixed Steps Input Functions

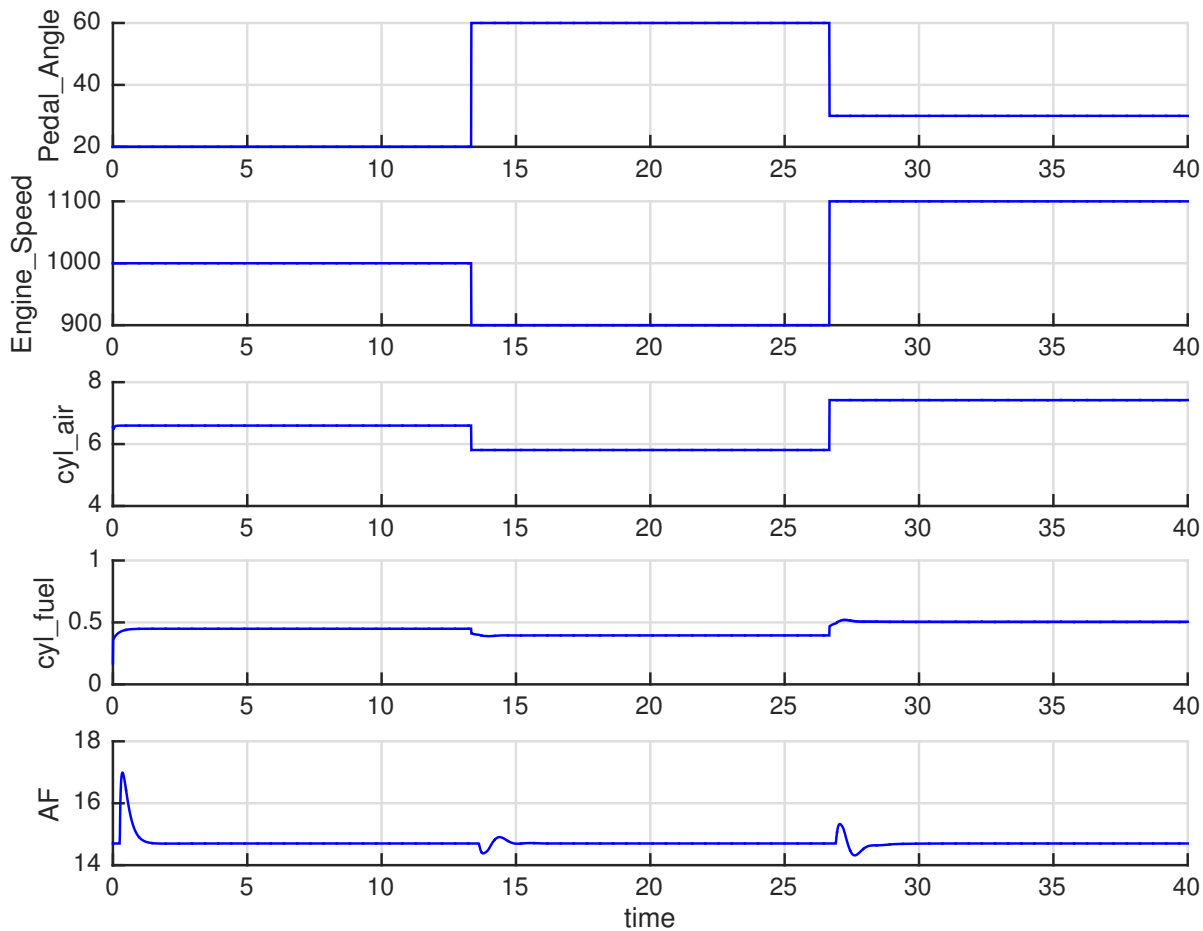
We set values for the control points and plot the result.

```
AFC_UniStep3.SetParam({'Engine_Speed_u0','Engine_Speed_u1','Engine_Speed_u2'}, [ 1000 900 1100]);
AFC_UniStep3.SetParam({'Pedal_Angle_u0','Pedal_Angle_u1','Pedal_Angle_u2'}, [ 20 60 30]);
AFC_UniStep3.Sim(0:.01:40);
AFC_UniStep3.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```

Fixed Steps Input Functions

We set values for the control points and plot the result.

```
AFC_UniStep3.SetParam({'Engine_Speed_u0','Engine_Speed_u1','Engine_Speed_u2'}, [ 1000 900 1100]);
AFC_UniStep3.SetParam({'Pedal_Angle_u0','Pedal_Angle_u1','Pedal_Angle_u2'}, [ 20 60 30]);
AFC_UniStep3.Sim(0:.01:40);
AFC_UniStep3.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```



Variable Steps Input Functions

We can have variable step inputs and also different numbers of control points and interpolation methods.

```
AFC_VarStep = BrAFC.copy();                % Creates a copy of the system interface
input_gen.type = 'VarStep';                % uniform time steps
input_gen.cp = [2 3];                      % number of control points
input_gen.method = {'previous', 'linear'};  % interpolation methods — see help of interp1.
AFC_VarStep.SetInputGen(input_gen);
```

This creates a new input parameterization:

```
AFC_VarStep.PrintParams();
```

Variable Steps Input Functions

We can have variable step inputs and also different numbers of control points and interpolation methods.

```
AFC_VarStep = BrAFC.copy();                % Creates a copy of the system interface
input_gen.type = 'VarStep';                % uniform time steps
input_gen.cp = [2 3];                      % number of control points
input_gen.method = {'previous', 'linear'};  % interpolation methods — see help of interp1.
AFC_VarStep.SetInputGen(input_gen);
```

This creates a new input parameterization:

```
AFC_VarStep.PrintParams();
```

```
Parameters:
----------
AF_sensor_tol=1
MAF_sensor_tol=1
fault_time=50
fuel_inj_tol=1
kappa_tol=1
ki=0.14
kp=0.04
pump_tol=1
tau_ww_tol=1
Pedal_Angle_u0=0
Pedal_Angle_dt0=1
Pedal_Angle_u1=0
Engine_Speed_u0=0
Engine_Speed_dt0=1
Engine_Speed_u1=0
Engine_Speed_dt1=1
Engine_Speed_u2=0
```

Changing Input Functions

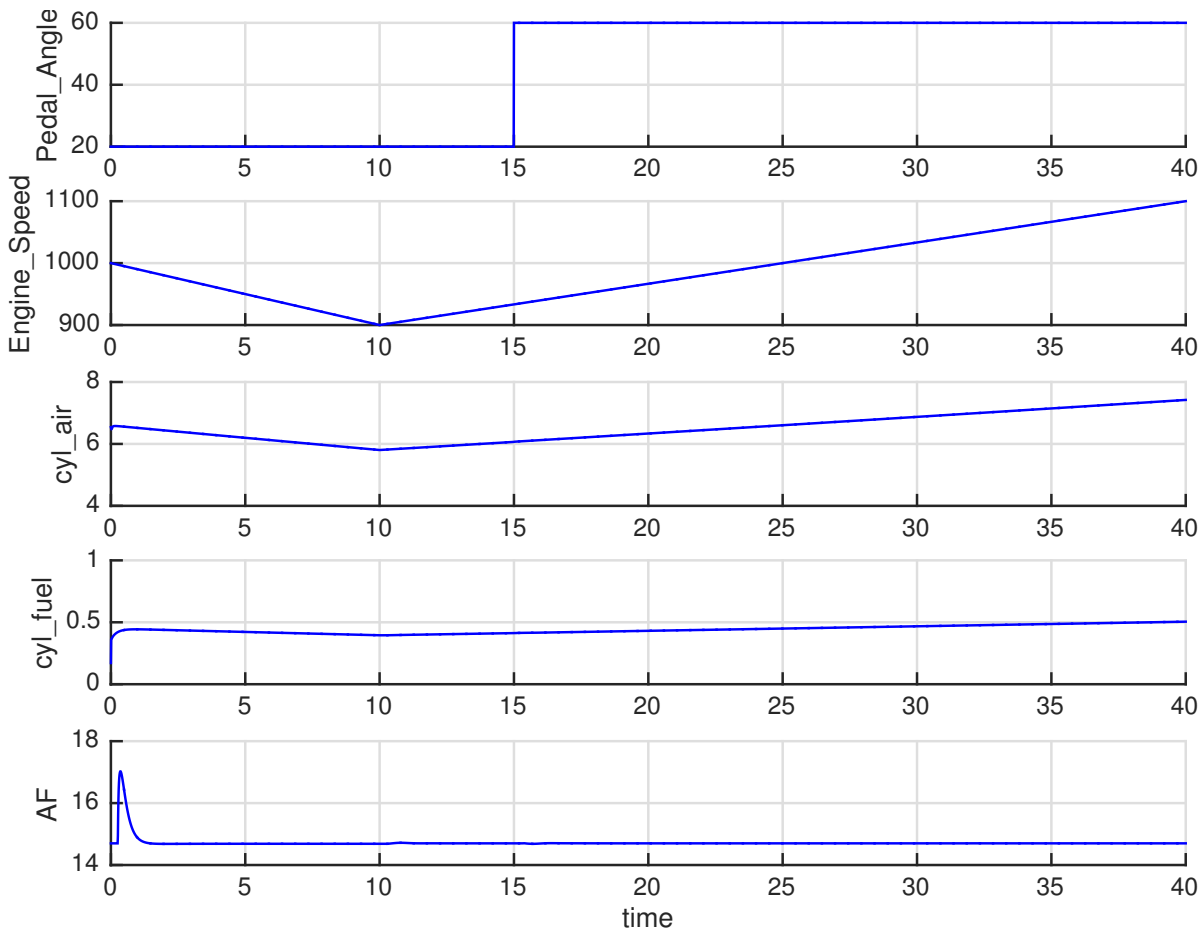
We set values for the control points and plot the result. The semantics is input_ui holds for input_dti seconds

```
AFC_VarStep.SetParam({'Engine_Speed_u0','Engine_Speed_dt0','Engine_Speed_u1', 'Engine_Speed_dt1', '
      Engine_Speed_u2'},...
                    [ 1000 10 900 30 1100]);
AFC_VarStep.SetParam({'Pedal_Angle_u0','Pedal_Angle_dt0','Pedal_Angle_u1'}, [20 15 60]);
AFC_VarStep.Sim(0:.01:40);
AFC_VarStep.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```

Changing Input Functions

We set values for the control points and plot the result. The semantics is input_ui holds for input_dti seconds

```
AFC_VarStep.SetParam({'Engine_Speed_u0','Engine_Speed_dt0','Engine_Speed_u1', 'Engine_Speed_dt1', '
      Engine_Speed_u2'},...
                    [ 1000 10 900 30 1100]);
AFC_VarStep.SetParam({'Pedal_Angle_u0','Pedal_Angle_dt0','Pedal_Angle_u1'}, [20 15 60]);
AFC_VarStep.Sim(0:.01:40);
AFC_VarStep.PlotSignals({'Pedal_Angle','Engine_Speed','cyl_air', 'cyl_fuel', 'AF'});
```



Mixing Signal Generators for Inputs

It is possible to mix different ways to generate signals for the different inputs. To do this, Breach provides a number of classes of signal generators. For example:

```
pedal_angle_gen = pulse_signal_gen({'Pedal_Angle'}); % Generate a pulse signal for pedal angle
engine_gen      = fixed_cp_signal_gen({'Engine_Speed'}, ... % signal name
                                      3,...                 % number of control points
                                     {'spline'});         % interpolation method
```

Several signal generators can be glued together in a special Breach System:

```
InputGen = BreachSignalGen({pedal_angle_gen, engine_gen});
```

InputGen is a Breach System in its own right. Meaning we can change parameters, plot signals, etc, independantly from a Simulink model

```
InputGen.SetParam({'Engine_Speed_u0','Engine_Speed_u1','Engine_Speed_u2'},...
                          [1000 1100 500]);
InputGen.SetParam({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', ...
                           'Pedal_Angle_pulse_amp','Pedal_Angle_pulse_width'}, ...
                           [0 15 30 .5]);
InputGen.PrintParams();
```

Mixing Signal Generators for Inputs

It is possible to mix different ways to generate signals for the different inputs. To do this, Breach provides a number of classes of signal generators. For example:

```
pedal_angle_gen = pulse_signal_gen({'Pedal_Angle'}); % Generate a pulse signal for pedal angle
engine_gen      = fixed_cp_signal_gen({'Engine_Speed'}, ... % signal name
                                      3,...                 % number of control points
                                     {'spline'});         % interpolation method
```

Several signal generators can be glued together in a special Breach System:

```
InputGen = BreachSignalGen({pedal_angle_gen, engine_gen});
```

InputGen is a Breach System in its own right. Meaning we can change parameters, plot signals, etc, independantly from a Simulink model

```
InputGen.SetParam({'Engine_Speed_u0','Engine_Speed_u1','Engine_Speed_u2'},...
                          [1000 1100 500]);
InputGen.SetParam({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', ...
                           'Pedal_Angle_pulse_amp','Pedal_Angle_pulse_width'}, ...
                           [0 15 30 .5]);
InputGen.PrintParams();
```

```
Parameters:
----------
Pedal_Angle_base_value=0
Pedal_Angle_pulse_period=15
Pedal_Angle_pulse_width=0.5
Pedal_Angle_pulse_amp=30
Engine_Speed_u0=1000
Engine_Speed_u1=1100
Engine_Speed_u2=500
```

Mixing Signal Generators for Inputs

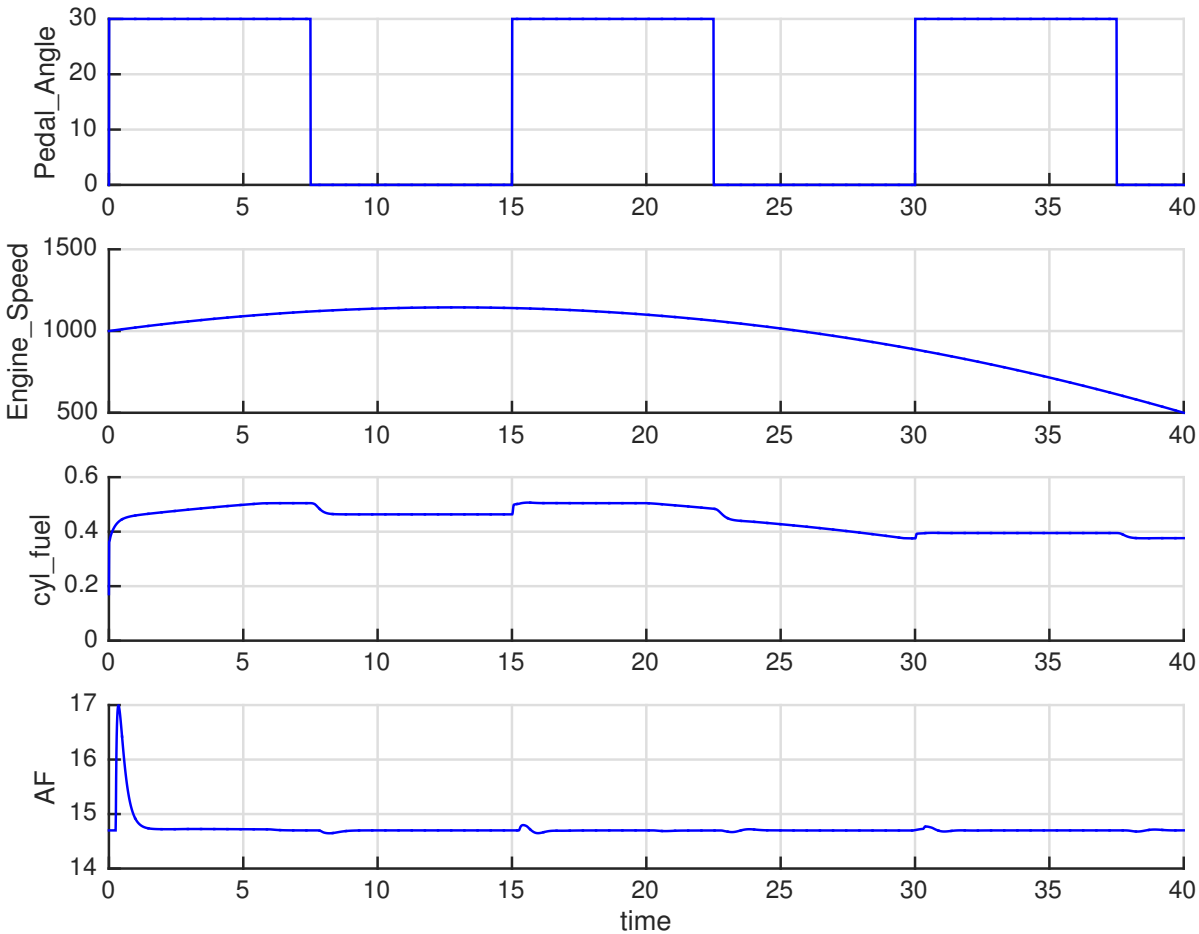
We can attach InputGen to the Simulink model and simulate as follows.

```
BrAFC.SetInputGen(InputGen);
BrAFC.Sim(0:.01:40);
BrAFC.PlotSignals({'Pedal_Angle', 'Engine_Speed','cyl_fuel', 'AF'});
```

Mixing Signal Generators for Inputs

We can attach InputGen to the Simulink model and simulate as follows.

```
BrAFC.SetInputGen(InputGen);
BrAFC.Sim(0:.01:40);
BrAFC.PlotSignals({'Pedal_Angle', 'Engine_Speed','cyl_fuel', 'AF'});
```



Writing a New Signal Generator

If the available signal generators are not enough. It is easy to write a new one. The idea is to write a simple new class implementing a computeSignals method.

```
type my_signal_generator;
```

Writing a New Signal Generator

If the available signal generators are not enough. It is easy to write a new one. The idea is to write a simple new class implementing a computeSignals method.

```
type my_signal_generator;
```

```
classdef my_signal_generator < signal_gen
    properties
       lambda  % some parameter for the signal generator — this won't be visible from Breach API
    end
    methods
        % The constructor must name the signals and the parameters needed to construct them.
        function this = my_signal_generator(lambda)
            this.lambda = lambda;
            this.signals = {'Engine_Speed', 'Pedal_Angle'};
            this.params = {'my_param_for_Engine', 'my_param_for_Pedal'};
            this.p0 = [1000 50]; % default values
        end

        % The class must implement a method with the signature below
        function [X, time] = computeSignals(this, p, time)
            % p contains values for the parameters in the declared order
            my_param_for_Engine = p(1);
            my_param_for_Pedal = p(2);

            % Constructs signals as some function of time and parameters
            Engine_Speed = this.lambda*cos(time)+ my_param_for_Engine;
            Pedal_Angle = this.lambda*sin(time)+ my_param_for_Pedal;

            % The signals must be returned as rows of X, in the declared order
            X = [ Engine_Speed; Pedal_Angle ];
        end
    end
end
```

Writing a New Signal Generator

We can use the new signal generator as the builtin ones

```
my_input_gen = my_signal_generator(2)         % creates an instance with lambda=2
MyInputGen = BreachSignalGen({my_input_gen}); % Makes it a Breach system

% Then plug it to the Simulink model:
BrAFC_MyGen = BrAFC.copy();
BrAFC_MyGen.SetInputGen(MyInputGen);
BrAFC_MyGen.PrintParams();  % Makes sure the new parameters are visible
```

Writing a New Signal Generator

We can use the new signal generator as the builtin ones

```
my_input_gen = my_signal_generator(2)         % creates an instance with lambda=2
MyInputGen = BreachSignalGen({my_input_gen}); % Makes it a Breach system

% Then plug it to the Simulink model:
BrAFC_MyGen = BrAFC.copy();
BrAFC_MyGen.SetInputGen(MyInputGen);
BrAFC_MyGen.PrintParams();  % Makes sure the new parameters are visible
```

```
my_input_gen =

  my_signal_generator with properties:

     lambda: 2
    signals: {'Engine_Speed'  'Pedal_Angle'}
     params: {'my_param_for_Engine'  'my_param_for_Pedal'}
         p0: [1000 50]

Parameters:
----------
AF_sensor_tol=1
MAF_sensor_tol=1
fault_time=50
fuel_inj_tol=1
kappa_tol=1
ki=0.14
kp=0.04
pump_tol=1
tau_ww_tol=1
my_param_for_Engine=1000
my_param_for_Pedal=50
```

Writing a New Signal Generator

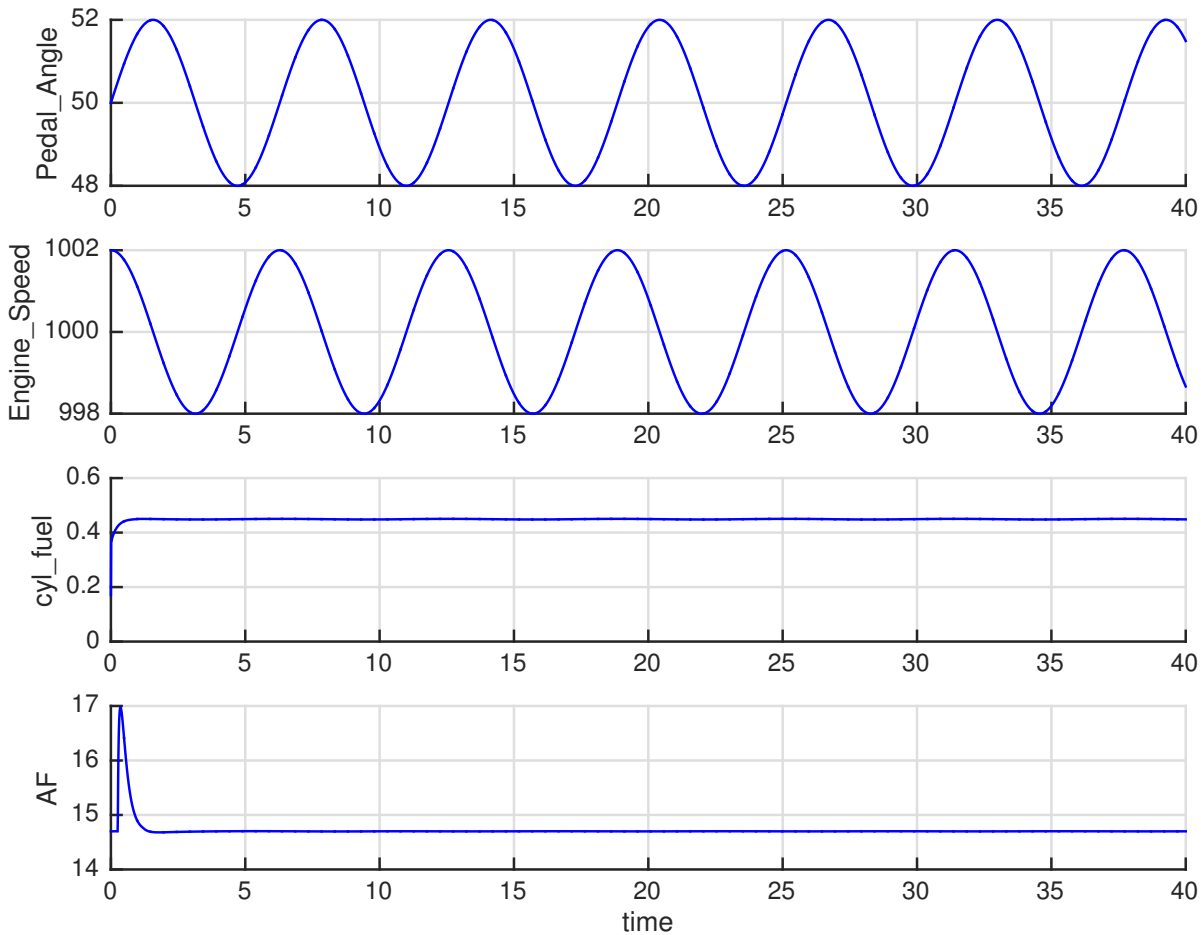
And simulate to see what happens.

```
BrAFC_MyGen.Sim(0:.01:40);
BrAFC_MyGen.PlotSignals({'Pedal_Angle', 'Engine_Speed','cyl_fuel', 'AF'});
```

Writing a New Signal Generator

And simulate to see what happens.

```
BrAFC_MyGen.Sim(0:.01:40);
BrAFC_MyGen.PlotSignals({'Pedal_Angle', 'Engine_Speed','cyl_fuel', 'AF'});
```



Setting Time for Simulation

If only the end time is specified, Breach collects signals at time instants decided by the solver used by Simulink. We can also force Simulink to provide outputs at specific times, e.g., every fixed time steps.

```
AFC_Time= BrAFC.copy(); % Creates a copy of the interface object
AFC_Time.Sim(0:.01:40); % Run Simulink simulation from time 0 to time 40
                        % and collect outputs at time steps of 0.01.
```

Note that Simulink will still use a variable step solver in this case. This only affects the output times. To change solver options (e.g., switch to a fixed step solver), the proper way is to open the original model, update solver options, and re-create the Breach interface.

Also we don't need to specify fixed time steps, e.g., the following will work too:

```
AFC_Time.Sim([0 1 5 10 20 21 30]); % Run Simulink simulation from time 0 to time 30
                                   % and collect outputs at times 0, 1, 5, etc
```

Finally, we can set a default time to run simulations:

```
AFC_Time.SetTime(0:.1:40);
AFC_Time.Sim();
```

Grid Sampling of Parameters

Each BreachSystem is also a BreachSet object, i.e., a collection of parameter values and traces. Some basic operations are available to generate new values, such as grid sampling and quasi-random sampling which we demonstrate next.

```
% We first need to set the ranges of parameters that we want to vary.
AFC_Grid= BrAFC.copy();
AFC_Grid.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'},...
                        [10 15; 0 61.1]);
% Next we creates a 5x5 grid sampling and plot it.
AFC_Grid.GridSample([5 5]);
AFC_Grid.PlotParams();
```

Grid Sampling of Parameters

Each BreachSystem is also a BreachSet object, i.e., a collection of parameter values and traces. Some basic operations are available to generate new values, such as grid sampling and quasi-random sampling which we demonstrate next.

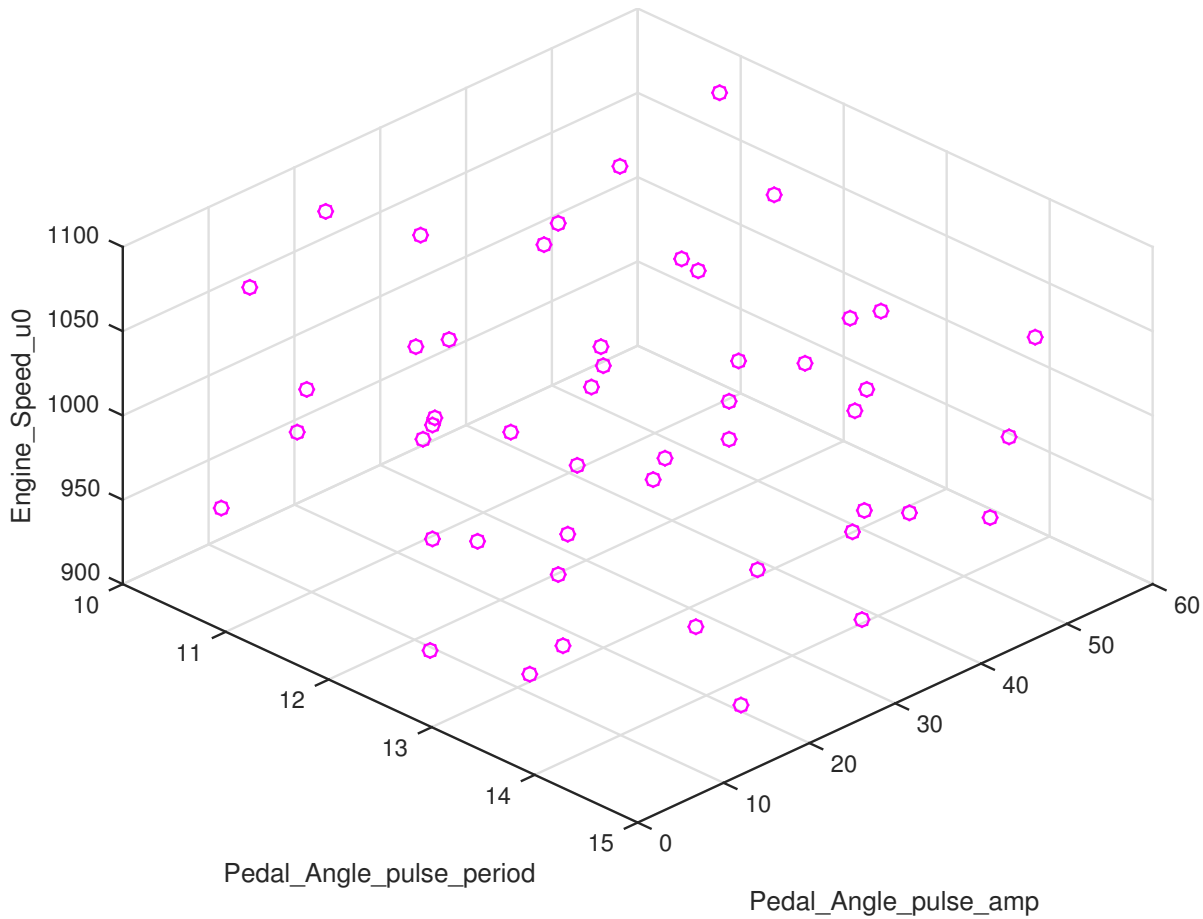
```
% We first need to set the ranges of parameters that we want to vary.
AFC_Grid= BrAFC.copy();
AFC_Grid.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'},...
                        [10 15; 0 61.1]);
% Next we creates a 5x5 grid sampling and plot it.
AFC_Grid.GridSample([5 5]);
AFC_Grid.PlotParams();
```

Random Sampling of Parameters

```
% Again, we first set the ranges of parameters that we want to sample.
AFC_Rand= BrAFC.copy();
AFC_Rand.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp', 'Engine_Speed_u0'},...
                        [10 15; 0 61.1;900 1100]);
% Next we creates a random sampling with 50 samples and plot it.
AFC_Rand.QuasiRandomSample(50);
AFC_Rand.PlotParams();
set(gca,'View', [45 45]);
```

Random Sampling of Parameters

```
% Again, we first set the ranges of parameters that we want to sample.
AFC_Rand= BrAFC.copy();
AFC_Rand.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp', 'Engine_Speed_u0'},...
                        [10 15; 0 61.1;900 1100]);
% Next we creates a random sampling with 50 samples and plot it.
AFC_Rand.QuasiRandomSample(50);
AFC_Rand.PlotParams();
set(gca,'View', [45 45]);
```



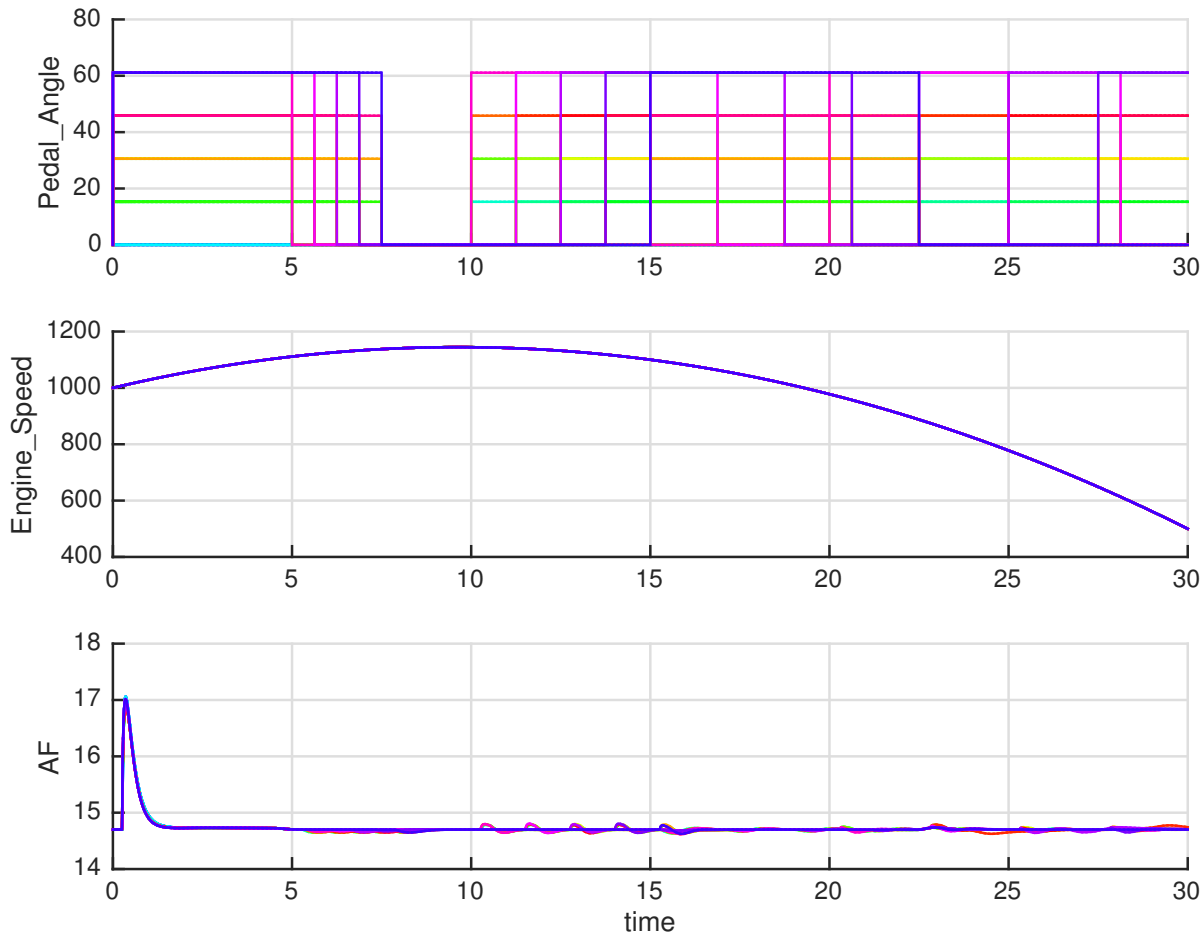
Simulation from Multiple Parameters

```
AFC_Grid.Sim(30);
AFC_Grid.PlotSignals({'Pedal_Angle','Engine_Speed', 'AF'}); % we plot only the input signals and AF
```

Simulation from Multiple Parameters

```
AFC_Grid.Sim(30);
AFC_Grid.PlotSignals({'Pedal_Angle','Engine_Speed', 'AF'}); % we plot only the input signals and AF
```

```
Computing 25 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



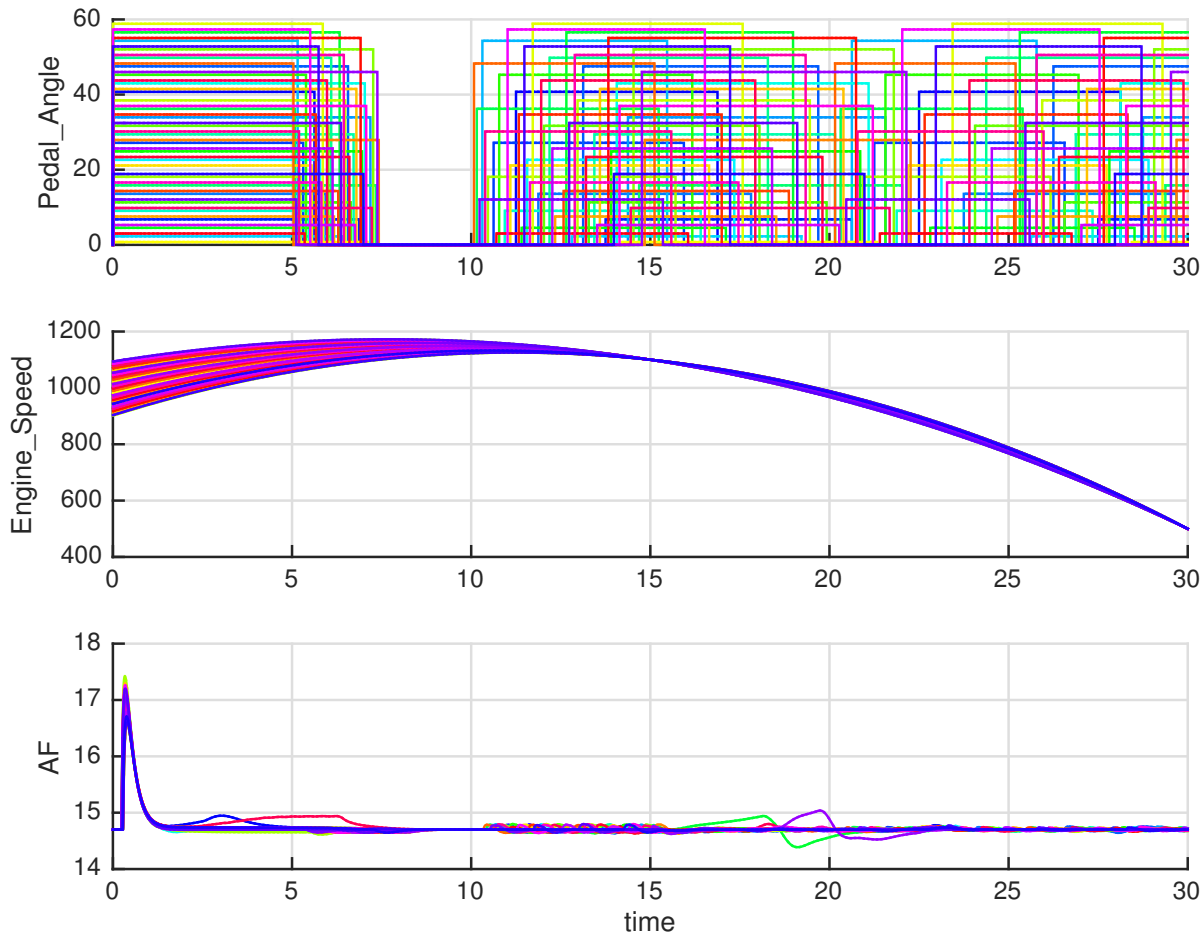
Simulation from Multiple Parameters

```
AFC_Rand.Sim(30);
AFC_Rand.PlotSignals({'Pedal_Angle','Engine_Speed','AF'}); % we plot only the input signals and AF
```

Simulation from Multiple Parameters

```
AFC_Rand.Sim(30);
AFC_Rand.PlotSignals({'Pedal_Angle','Engine_Speed','AF'}); % we plot only the input signals and AF
```

```
Computing 50 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Estimating Signal Ranges from Simulations

```
AFC_Rand.UpdateSignalRanges(); % Computes min and max values for all signals
AFC_Rand.PrintSignals();
```

Estimating Signal Ranges from Simulations

```
AFC_Rand.UpdateSignalRanges(); % Computes min and max values for all signals
AFC_Rand.PrintSignals();
```

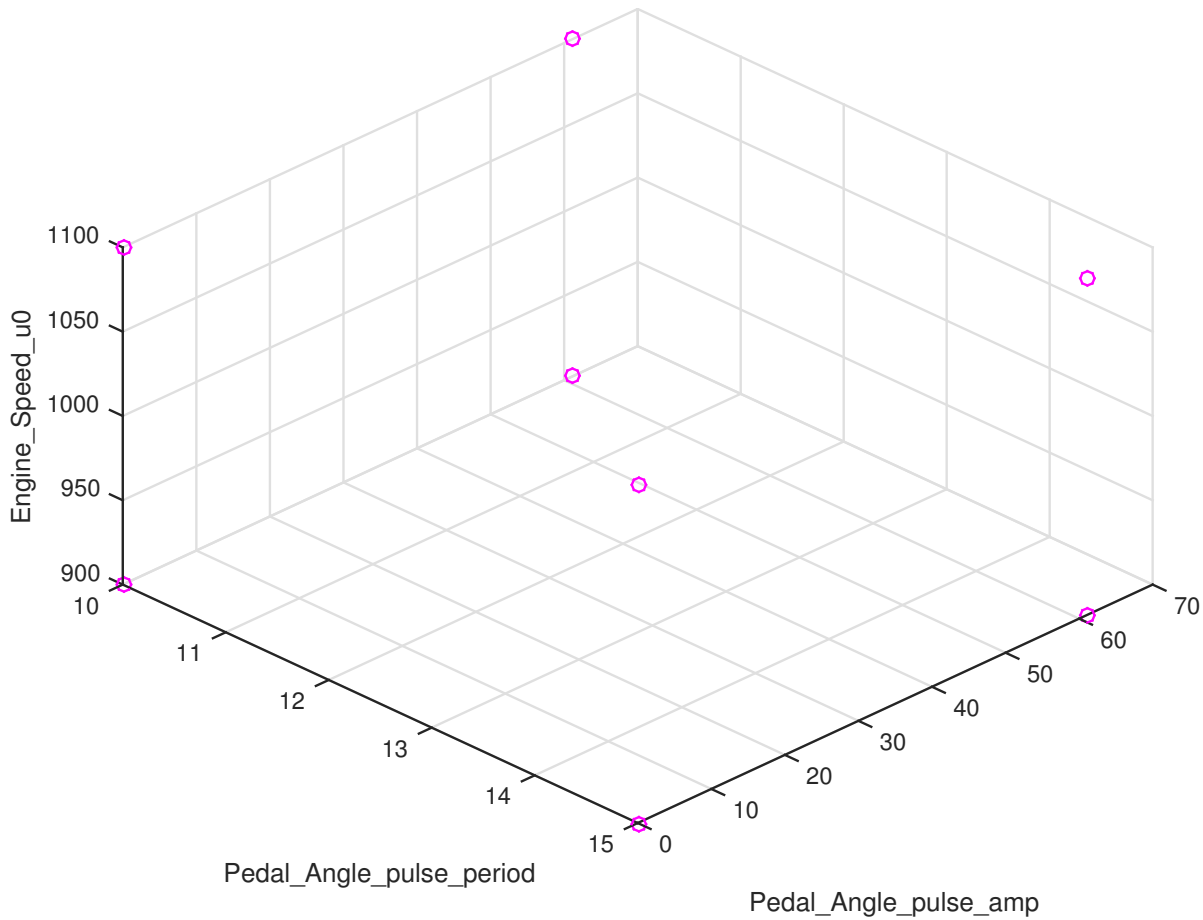
```
Signals (in range estimated over 50 simulations):
________
AFref in  [0, 14.7]
AF in  [14.3879, 17.4242]
controller_mode in  [0, 1]
cyl_air in  [5.5279, 7.42155]
manifold_pressure in  [0.862957, 0.999915]
MAF in  [2.63426, 20.7167]
cyl_fuel in  [0.170233, 0.507597]
Engine_Speed in  [500, 1172.63]
Pedal_Angle in  [0, 58.837]
```

Simulations from Corners

```
% Again, we first set the ranges of parameters that we want to sample.
AFC_Corners= BrAFC.copy();
AFC_Corners.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp', 'Engine_Speed_u0'},
      ...
                        [10 15; 0 61.1;900 1100]);
% Next we obtain the corner parameters
AFC_Corners.CornerSample();
AFC_Corners.PlotParams();
set(gca,'View', [45 45]);
```

Simulations from Corners

```
% Again, we first set the ranges of parameters that we want to sample.
AFC_Corners= BrAFC.copy();
AFC_Corners.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp', 'Engine_Speed_u0'},
      ...
                        [10 15; 0 61.1;900 1100]);
% Next we obtain the corner parameters
AFC_Corners.CornerSample();
AFC_Corners.PlotParams();
set(gca,'View', [45 45]);
```



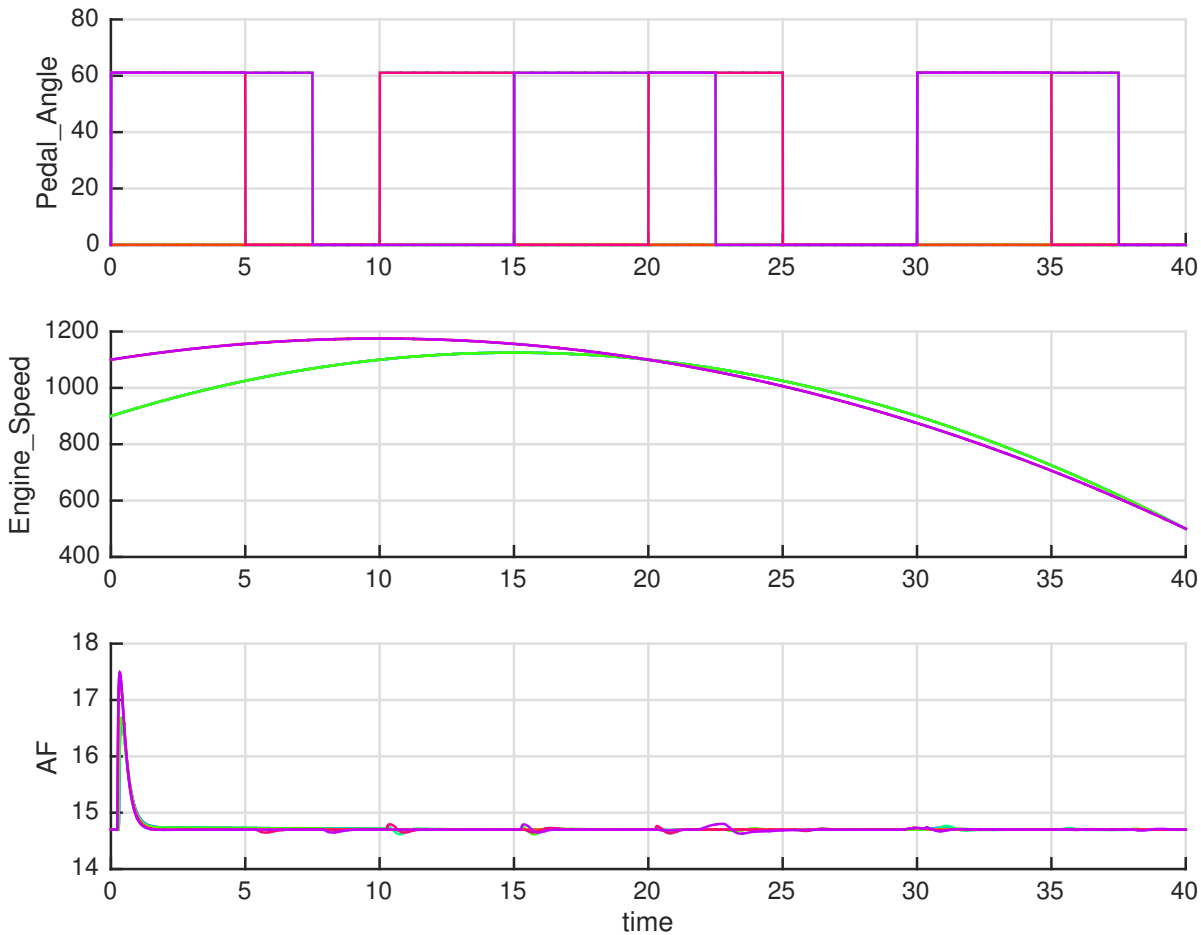
Simulations from Corners

```
AFC_Corners.Sim();
AFC_Corners.PlotSignals({'Pedal_Angle','Engine_Speed','AF'});
```

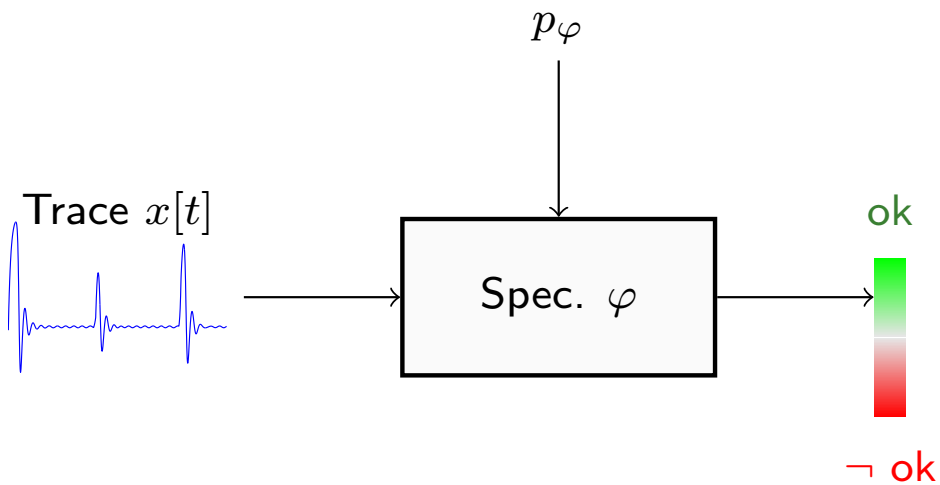
Simulations from Corners

```
AFC_Corners.Sim();
AFC_Corners.PlotSignals({'Pedal_Angle','Engine_Speed','AF'});
```

```
Computing 8 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%                ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Signal Temporal Logic (STL) Specifications



Writing Simple STL Properties

First we define a predicate stating that AF is above 1% of AFref

```
AF_not_ok = STL_Formula('AF_not_ok', 'AF[t]— AFref[t] > 0.01*14.7')
```

Writing Simple STL Properties

First we define a predicate stating that AF is above 1% of AFref

```
AF_not_ok = STL_Formula('AF_not_ok', 'AF[t]— AFref[t] > 0.01*14.7')
```

```
AF[t]— AFref[t] > 0.01*14.7
```

The first argument 'AF_not_ok' is an id for the formula needed so that it can be referenced as a sub-formula.

The second argument is a string describing the formula. The syntax AF[t] refers to the value of signal AF at a time t. If we evaluate this formula, t will be instantiated with 0. We need temporal operators to examine other time instants.

For example, we define next a formula stating "some time in the future, AF_not_ok is true".

```
AF_ev_not_ok = STL_Formula('AF_ev_not_ok', 'ev (AF_not_ok)')
```

Writing Simple STL Properties

First we define a predicate stating that AF is above 1% of AFref

```
AF_not_ok = STL_Formula('AF_not_ok', 'AF[t]— AFref[t] > 0.01*14.7')
```

```
AF[t]— AFref[t] > 0.01*14.7
```

The first argument 'AF_not_ok' is an id for the formula needed so that it can be referenced as a sub-formula.

The second argument is a string describing the formula. The syntax AF[t] refers to the value of signal AF at a time t. If we evaluate this formula, t will be instantiated with 0. We need temporal operators to examine other time instants.

For example, we define next a formula stating "some time in the future, AF_not_ok is true".

```
AF_ev_not_ok = STL_Formula('AF_ev_not_ok', 'ev (AF_not_ok)')
```

```
ev (AF_not_ok)
```

ev is a shorthand for 'eventually'. Other temporal operators are 'alw' or 'always' and 'until'.

Checking Simple Properties with Nominal Parameters

Temporal operators can specify a time range to operate on. For our system, AF does not need to be checked before 10s, and we simulate until 30s so we modify the formula as:

```
AF_ev_not_ok = STL_Formula('AF_ev_not_ok', 'ev_[10,30] (AF_not_ok)')
```

Checking Simple Properties with Nominal Parameters

Temporal operators can specify a time range to operate on. For our system, AF does not need to be checked before 10s, and we simulate until 30s so we modify the formula as:

```
AF_ev_not_ok = STL_Formula('AF_ev_not_ok', 'ev_[10,30] (AF_not_ok)')
```

```


ev_[10,30] (AF_not_ok)
```

Then we check our formula on a simulation with nominal parameters:

```
AFC_w_Specs= BrAFC.copy();
Time = 0:.01:30;
AFC_w_Specs.Sim(Time);
AFC_w_Specs.CheckSpec(AF_ev_not_ok)
```

Checking Simple Properties with Nominal Parameters

Temporal operators can specify a time range to operate on. For our system, AF does not need to be checked before 10s, and we simulate until 30s so we modify the formula as:

```
AF_ev_not_ok = STL_Formula('AF_ev_not_ok', 'ev_[10,30] (AF_not_ok)')
```

```


ev_[10,30] (AF_not_ok)
```

Then we check our formula on a simulation with nominal parameters:

```
AFC_w_Specs= BrAFC.copy();
Time = 0:.01:30;
AFC_w_Specs.Sim(Time);
AFC_w_Specs.CheckSpec(AF_ev_not_ok)
```

```


ans =

   -0.0590
```

Negative results means the formula is false, i.e., the system does not overshoot.

Checking Simple Properties with Nominal Parameters

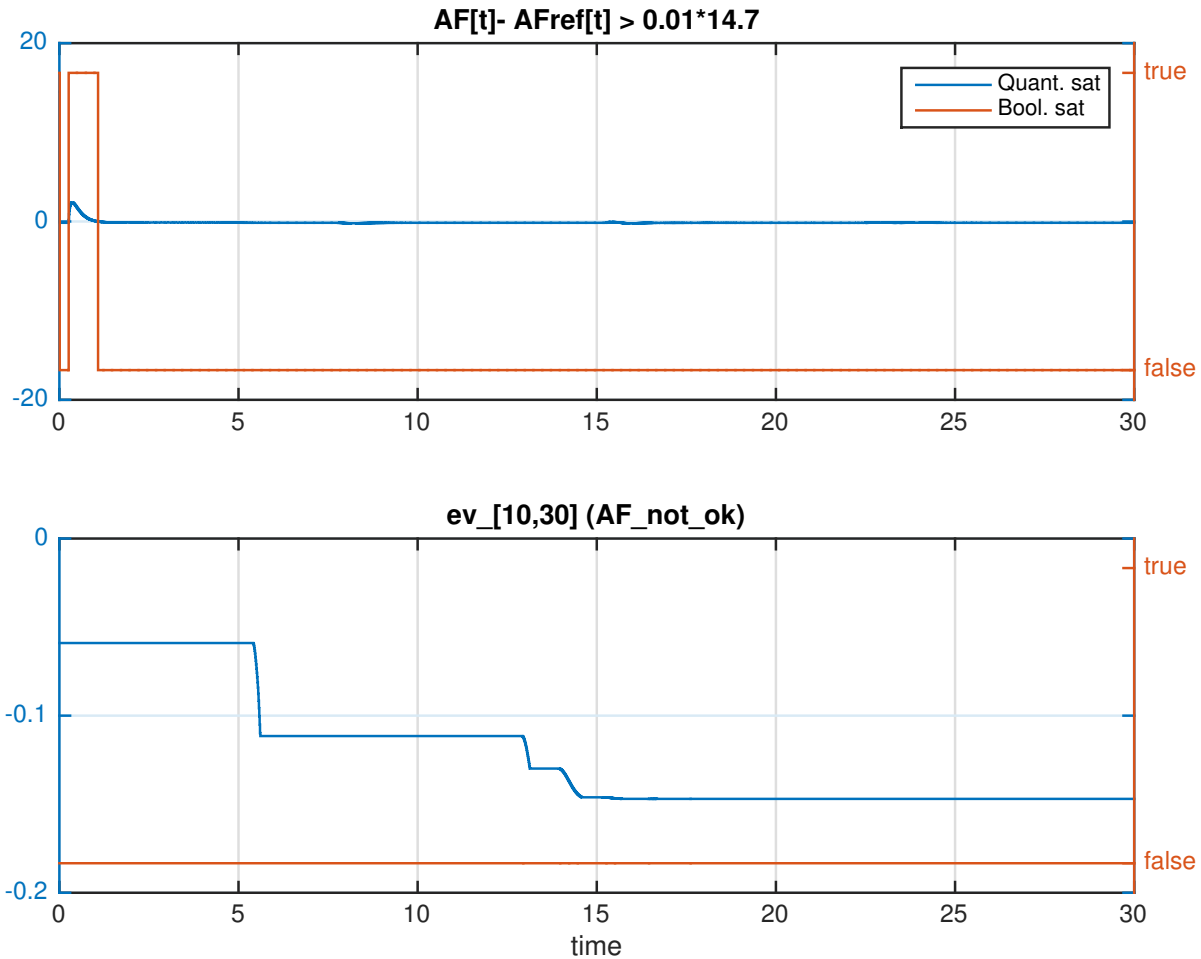
We can plot the satisfaction function with

```
AFC_w_Specs.PlotRobustSat(AF_ev_not_ok);
```

Checking Simple Properties with Nominal Parameters

We can plot the satisfaction function with

```
AFC_w_Specs.PlotRobustSat(AF_ev_not_ok);
```



Checking Simple Properties with Nominal Parameters

The reason we are not interested in AF between 0 and 10s is because the controller is not in a mode where it tries to regulate it at this time. We can implement this explicitly using the controller_mode signal:

```
AF_ev_not_ok2 = STL_Formula('AF_ev_not_ok2', 'ev (controller_mode[t]==0 and AF_not_ok)')
```

Checking Simple Properties with Nominal Parameters

The reason we are not interested in AF between 0 and 10s is because the controller is not in a mode where it tries to regulate it at this time. We can implement this explicitly using the controller_mode signal:

```
AF_ev_not_ok2 = STL_Formula('AF_ev_not_ok2', 'ev (controller_mode[t]==0 and AF_not_ok)')
```

```
ev ((controller_mode[t]==0) and (AF_not_ok))
```

Then check the new formula on the simulation we performed already:

```
AFC_w_Specs.CheckSpec(AF_ev_not_ok2)
```

Checking Simple Properties with Nominal Parameters

The reason we are not interested in AF between 0 and 10s is because the controller is not in a mode where it tries to regulate it at this time. We can implement this explicitly using the controller_mode signal:

```
AF_ev_not_ok2 = STL_Formula('AF_ev_not_ok2', 'ev (controller_mode[t]==0 and AF_not_ok)')
```

```
ev ((controller_mode[t]==0) and (AF_not_ok))
```

Then check the new formula on the simulation we performed already:

```
AFC_w_Specs.CheckSpec(AF_ev_not_ok2)
```

```
ans =

     1
```

We should still get a negative result.

Checking Simple Properties with Nominal Parameters

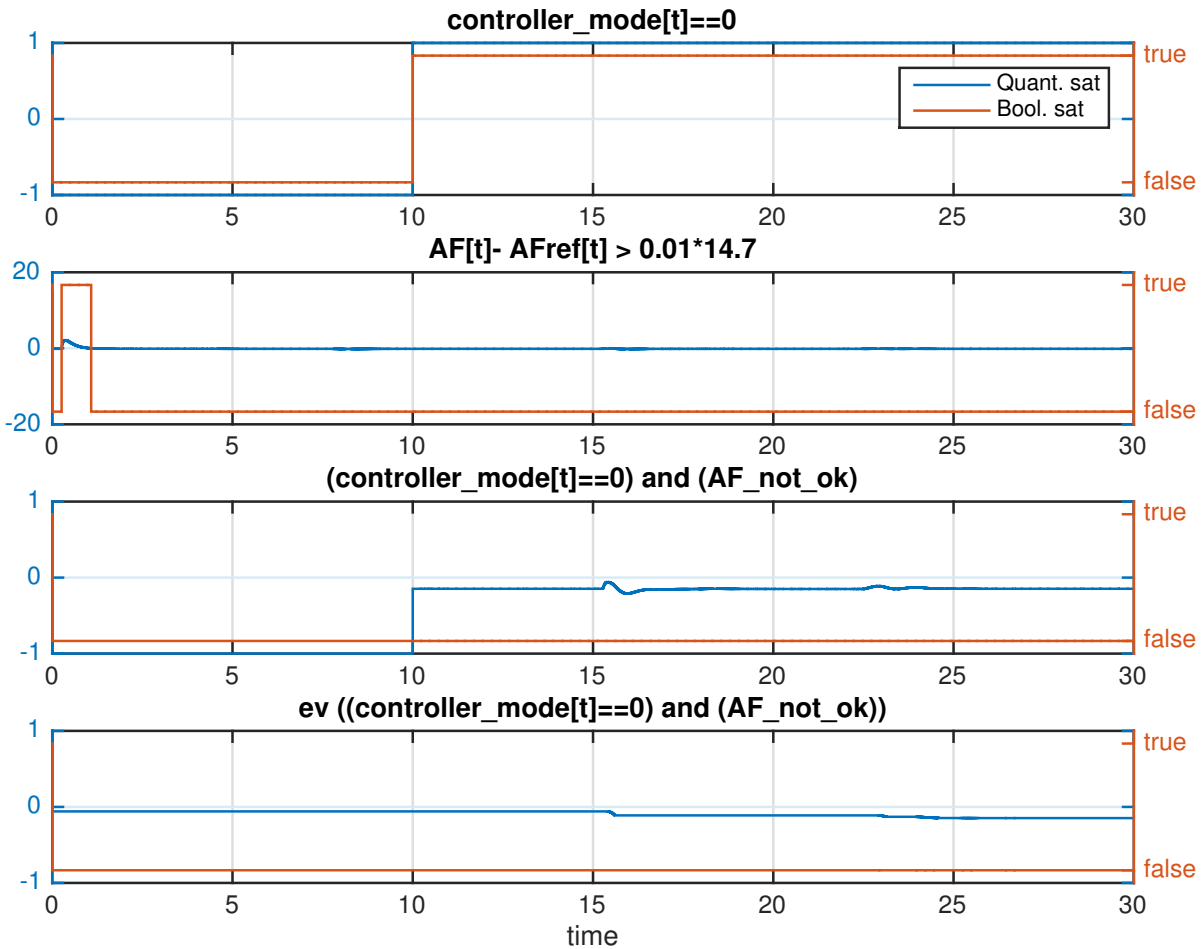
We can plot the satisfaction function again to interpret the result.

```
AFC_w_Specs.PlotRobustSat(AF_ev_not_ok2);
```

Checking Simple Properties with Nominal Parameters

We can plot the satisfaction function again to interpret the result.

```
AFC_w_Specs.PlotRobustSat(AF_ev_not_ok2);
```



Reading a formula from an STL File

STL formulas can be defined in a file. This makes it much easier to write complex properties. Also easier to defined parametric STL formulas, e.g.:

```
type simple_spec.stl
```

Reading a formula from an STL File

STL formulas can be defined in a file. This makes it much easier to write complex properties. Also easier to defined parametric STL formulas, e.g.:

```
type simple_spec.stl
```

```
# We declare parameters with default values
param tol=0.01, af_ref=14.7

# Slightly more complex definition of Air—Fuel Ratio being OK
AF_above_ref := AF[t]— AFref[t] >  tol*af_ref
AF_below_ref := AF[t]— AFref[t] < —tol*af_ref
AF_ok := not (AF_above_ref or AF_below_ref)

# Top formula, using time parameters
param ti=10, tf=30
AF_alw_ok := alw_[ti,tf] (AF_ok)
```

STL files are loaded using the following command:

```
STL_ReadFile('simple_spec.stl'); % this loads all formulas and sub—formulas defined in the file
AFC_w_Specs.CheckSpec(AF_alw_ok)
```

Reading a formula from an STL File

STL formulas can be defined in a file. This makes it much easier to write complex properties. Also easier to defined parametric STL formulas, e.g.:

```
type simple_spec.stl
```

```
# We declare parameters with default values
param tol=0.01, af_ref=14.7

# Slightly more complex definition of Air-Fuel Ratio being OK
AF_above_ref := AF[t]- AFref[t] >  tol*af_ref
AF_below_ref := AF[t]- AFref[t] < -tol*af_ref
AF_ok := not (AF_above_ref or AF_below_ref)

# Top formula, using time parameters
param ti=10, tf=30
AF_alw_ok := alw_[ti,tf] (AF_ok)
```

STL files are loaded using the following command:

```
STL_ReadFile('simple_spec.stl'); % this loads all formulas and sub-formulas defined in the file
AFC_w_Specs.CheckSpec(AF_alw_ok)
```

```
ans =

    0.0590
```

Plotting satisfaction of complex formulas

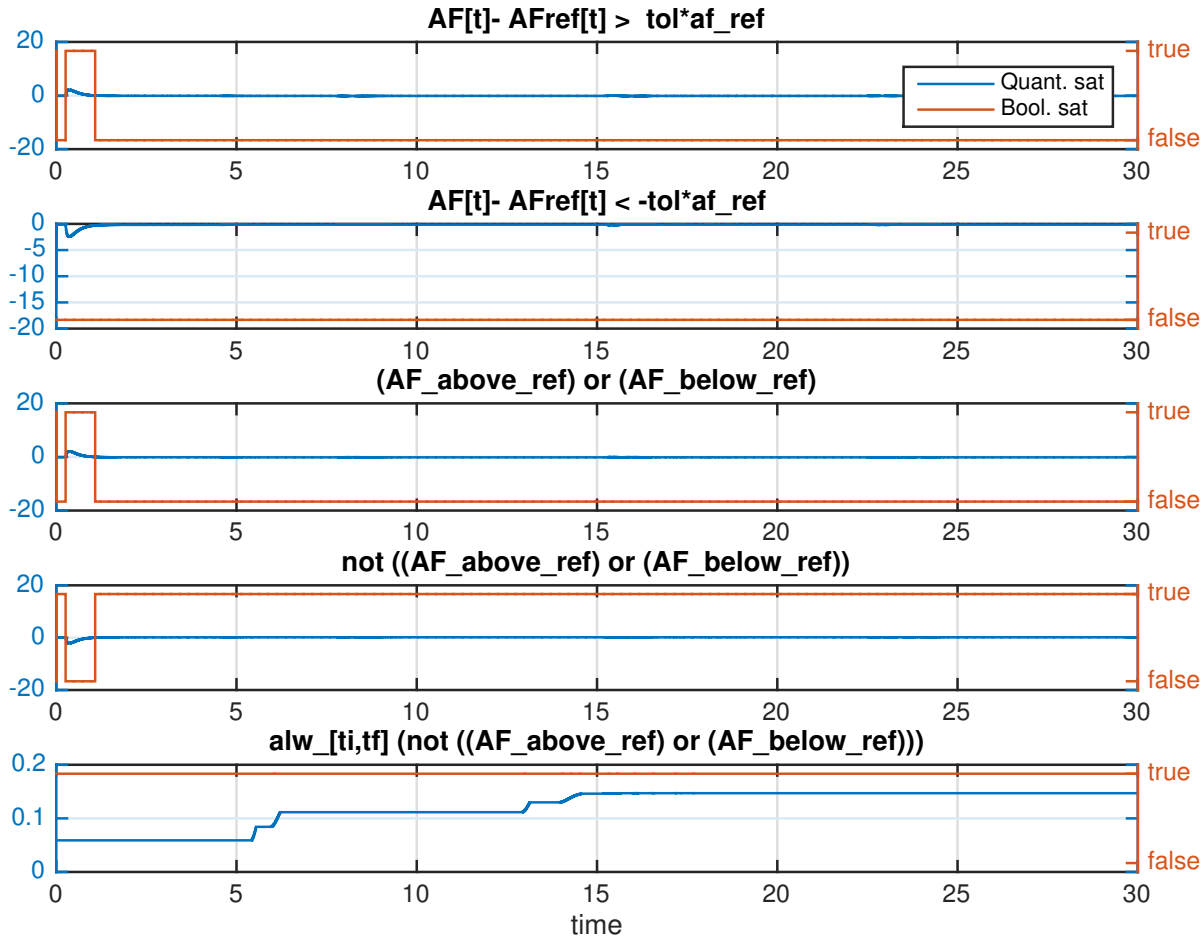
We can plot the satisfaction function again to interpret the result.

```
AFC_w_Specs.PlotRobustSat(AF_alw_ok);
```

Plotting satisfaction of complex formulas

We can plot the satisfaction function again to interpret the result.

```
AFC_w_Specs.PlotRobustSat(AF_alw_ok);
```



Plotting satisfaction of complex formulas

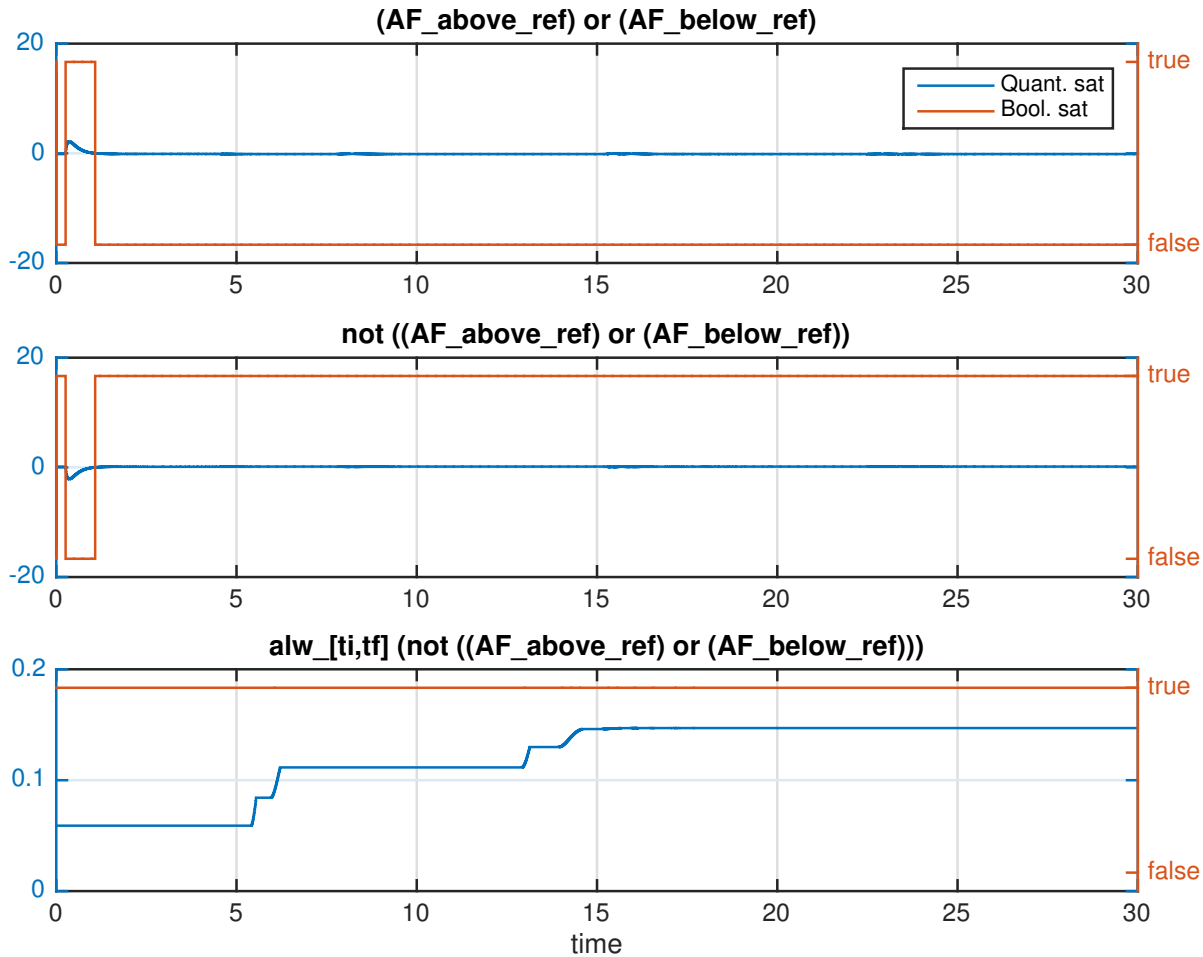
PlotRobustSat decomposes the formula into subformulas. We can specify the depth of decomposition, e.g.,

```
AFC_w_Specs.PlotRobustSat(AF_alw_ok,3); % shows satisfaction of top formula and 2 subformulas.
```

Plotting satisfaction of complex formulas

PlotRobustSat decomposes the formula into subformulas. We can specify the depth of decomposition, e.g.,

```
AFC_w_Specs.PlotRobustSat(AF_alw_ok,3); % shows satisfaction of top formula and 2 subformulas.
```



Dealing with Formula Parameters

Parameters for formulas can be accessed and changed using get_params and set_params functions:

```
get_params(AF_alw_ok)
```

Dealing with Formula Parameters

Parameters for formulas can be accessed and changed using get_params and set_params functions:

```
get_params(AF_alw_ok)
```

```
ans =

       tol: 0.0100
    af_ref: 14.7000
        ti: 10
        tf: 30
```

```
AF_alw_ok2 = set_params(AF_alw_ok, {'tol'}, [1e-3]) % change tolerance from 1% to 0.1%
```

Dealing with Formula Parameters

Parameters for formulas can be accessed and changed using get_params and set_params functions:

```
get_params(AF_alw_ok)
```

```
ans =

       tol: 0.0100
    af_ref: 14.7000
        ti: 10
        tf: 30
```

```
AF_alw_ok2 = set_params(AF_alw_ok, {'tol'}, [1e-3]) % change tolerance from 1% to 0.1%
```

```
alw_[ti,tf] (not ((AF_above_ref) or (AF_below_ref)))
```

```
AFC_w_Specs.CheckSpec(AF_alw_ok2)
```

Dealing with Formula Parameters

Parameters for formulas can be accessed and changed using get_params and set_params functions:

```
get_params(AF_alw_ok)
```

```
ans =

       tol: 0.0100
    af_ref: 14.7000
        ti: 10
        tf: 30
```

```
AF_alw_ok2 = set_params(AF_alw_ok, {'tol'}, [1e-3]) % change tolerance from 1% to 0.1%
```

```
alw_[ti,tf] (not ((AF_above_ref) or (AF_below_ref)))
```

```
AFC_w_Specs.CheckSpec(AF_alw_ok2)
```

```
ans =

   -0.0733
```

Dealing with Formula Parameters

Another way to access and modify formula parameters is by defining them using the methods GetParam and SetParam of class BreachSystem. This actually overrides the values defined for the STL_Formula object.

```
AFC_w_Specs.SetParam('tol', 1e-3, 'spec'); % the third argument tells Breach that we set
      specification parameter.
                                          % Ommitting it will work but issue a warning since tol is
                                                 not parameter of the system.
AFC_w_Specs.CheckSpec(AF_alw_ok)
```

Dealing with Formula Parameters

Another way to access and modify formula parameters is by defining them using the methods GetParam and SetParam of class BreachSystem. This actually overrides the values defined for the STL_Formula object.

```
AFC_w_Specs.SetParam('tol', 1e-3, 'spec'); % the third argument tells Breach that we set
      specification parameter.
                                          % Ommitting it will work but issue a warning since tol is
                                                 not parameter of the system.
AFC_w_Specs.CheckSpec(AF_alw_ok)
```

```
ans =

   -0.0733
```

The advantage of the second method is that we can explore (sample) the formula parameter space the same way we did with system parameters, as demonstrated next.

Checking Multiple Formula Parameters

```
AFC_w_mult_Specs= AFC_w_Specs.copy();
% we define a range for parameter tol and and grid sample it with 10 values
AFC_w_mult_Specs.SetParamRanges('tol', [0 1e−2]);
AFC_w_mult_Specs.GridSample(10); AFC_w_mult_Specs.GetParam('tol')
```

Checking Multiple Formula Parameters

```
AFC_w_mult_Specs= AFC_w_Specs.copy();
% we define a range for parameter tol and and grid sample it with 10 values
AFC_w_mult_Specs.SetParamRanges('tol', [0 1e−2]);
AFC_w_mult_Specs.GridSample(10); AFC_w_mult_Specs.GetParam('tol')
```

```
ans =

  Columns 1 through 7

         0    0.0011    0.0022    0.0033    0.0044    0.0056    0.0067

  Columns 8 through 10

    0.0078    0.0089    0.0100
```

```
AFC_w_mult_Specs.CheckSpec(AF_alw_ok)
```

Checking Multiple Formula Parameters

```
AFC_w_mult_Specs= AFC_w_Specs.copy();
% we define a range for parameter tol and and grid sample it with 10 values
AFC_w_mult_Specs.SetParamRanges('tol', [0 1e−2]);
AFC_w_mult_Specs.GridSample(10); AFC_w_mult_Specs.GetParam('tol')
```

```
ans =

  Columns 1 through 7

         0    0.0011    0.0022    0.0033    0.0044    0.0056    0.0067

  Columns 8 through 10

    0.0078    0.0089    0.0100
```

```
AFC_w_mult_Specs.CheckSpec(AF_alw_ok)
```

```
ans =

  Columns 1 through 7

   −0.0880   −0.0717   −0.0553   −0.0390   −0.0227   −0.0063    0.0100

  Columns 8 through 10

    0.0263    0.0427    0.0590
```

Checking a Property for Multiple System Parameters

```
% Create new interface object with some ranges for pedal angle inputs and sample randomly
AFC_Rand_w_Specs = BrAFC.copy();
AFC_Rand_w_Specs.SetParamRanges({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, [0 20; 10 15; 0 40]);
AFC_Rand_w_Specs.QuasiRandomSample(10);
AFC_Rand_w_Specs.Sim(Time);
% Set property parameter
AFC_Rand_w_Specs.SetParam({'ti', 'tf', 'tol'}, [10 30 .003], 'spec');
% Checks property for all simulations
AFC_Rand_w_Specs.CheckSpec(AF_alw_ok)
```

Checking a Property for Multiple System Parameters

```
% Create new interface object with some ranges for pedal angle inputs and sample randomly
AFC_Rand_w_Specs = BrAFC.copy();
AFC_Rand_w_Specs.SetParamRanges({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, [0 20; 10 15; 0 40]);
AFC_Rand_w_Specs.QuasiRandomSample(10);
AFC_Rand_w_Specs.Sim(Time);
% Set property parameter
AFC_Rand_w_Specs.SetParam({'ti', 'tf', 'tol'}, [10 30 .003], 'spec');
% Checks property for all simulations
AFC_Rand_w_Specs.CheckSpec(AF_alw_ok)
```

```
Computing 10 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

ans =

  Columns 1 through 7

    0.0058    0.0064   -0.0088    0.0061    0.0066    0.0061   -0.0156

  Columns 8 through 10

   -0.0400    0.0059    0.0061
```

Note that for some simulations, the property is false, meaning the system overshoots.

Checking a Property for Multiple System Parameters

To find out which parameter value lead to a positive or negative satisfaction, we can do the following:

```
sat_values = AFC_Rand_w_Specs.CheckSpec(AF_alw_ok);
i_pos = find(sat_values<0);
param_values = AFC_Rand_w_Specs.GetParam({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, i_pos)
```

Checking a Property for Multiple System Parameters

To find out which parameter value lead to a positive or negative satisfaction, we can do the following:

```
sat_values = AFC_Rand_w_Specs.CheckSpec(AF_alw_ok);
i_pos = find(sat_values<0);
param_values = AFC_Rand_w_Specs.GetParam({'Pedal_Angle_base_value', 'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, i_pos)
```

```
param_values =

    2.5000    1.2500   11.2500
   12.2222   14.4444   10.1852
   32.0000   25.6000   33.6000
```

Monitoring a formula on a trace

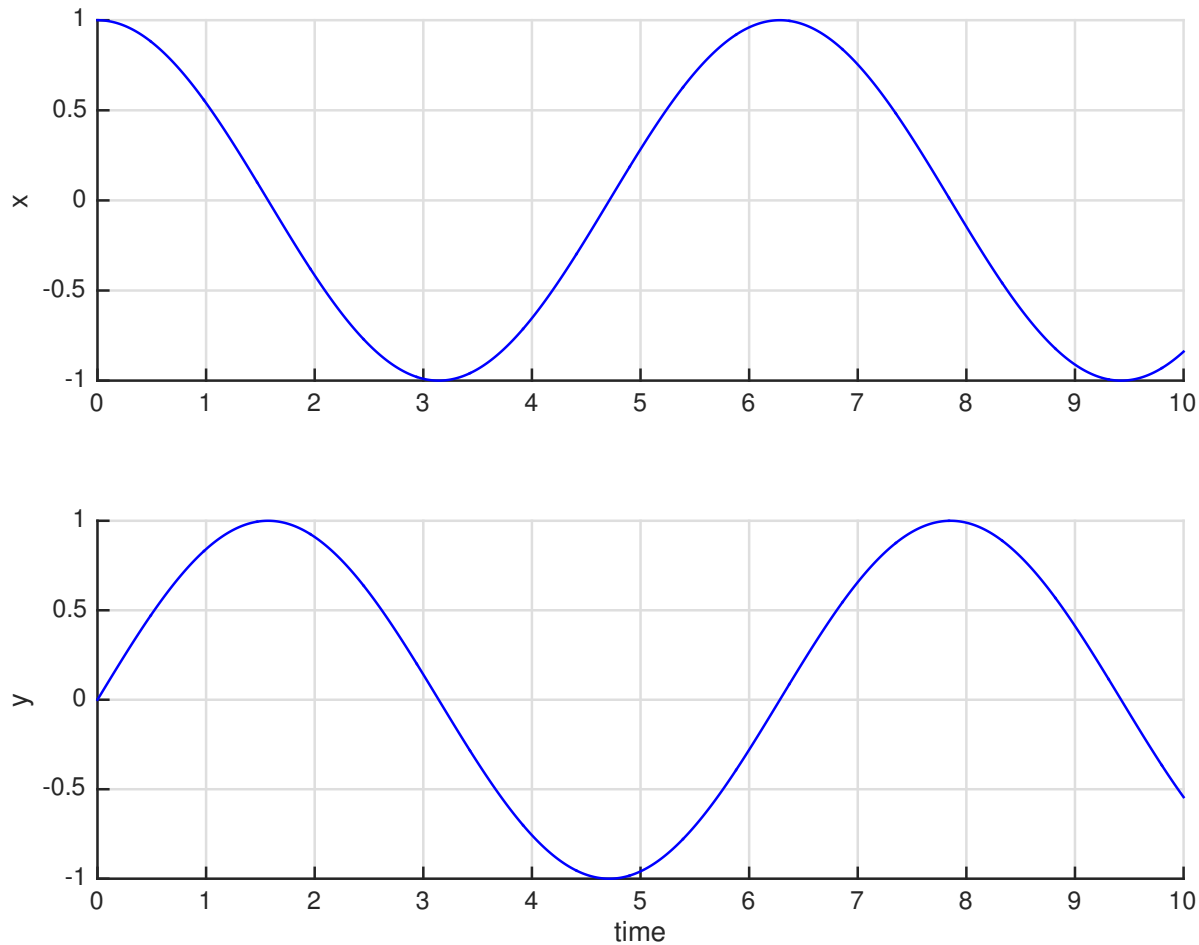
To monitor STL formulas on existing traces, one can use the BreachTraceSystem class.

```
% create a trace with signals x and y
time = 0:.01:10; x = cos(time); y = sin(time);
trace = [time' x' y']; % trace is in column format, first column is time
BrTrace = BreachTraceSystem({'x','y'}, trace);
BrTrace.PlotSignals();
```

Monitoring a formula on a trace

To monitor STL formulas on existing traces, one can use the BreachTraceSystem class.

```
% create a trace with signals x and y
time = 0:.01:10; x = cos(time); y = sin(time);
trace = [time' x' y']; % trace is in column format, first column is time
BrTrace = BreachTraceSystem({'x','y'}, trace);
BrTrace.PlotSignals();
```



Monitoring a formula on a trace

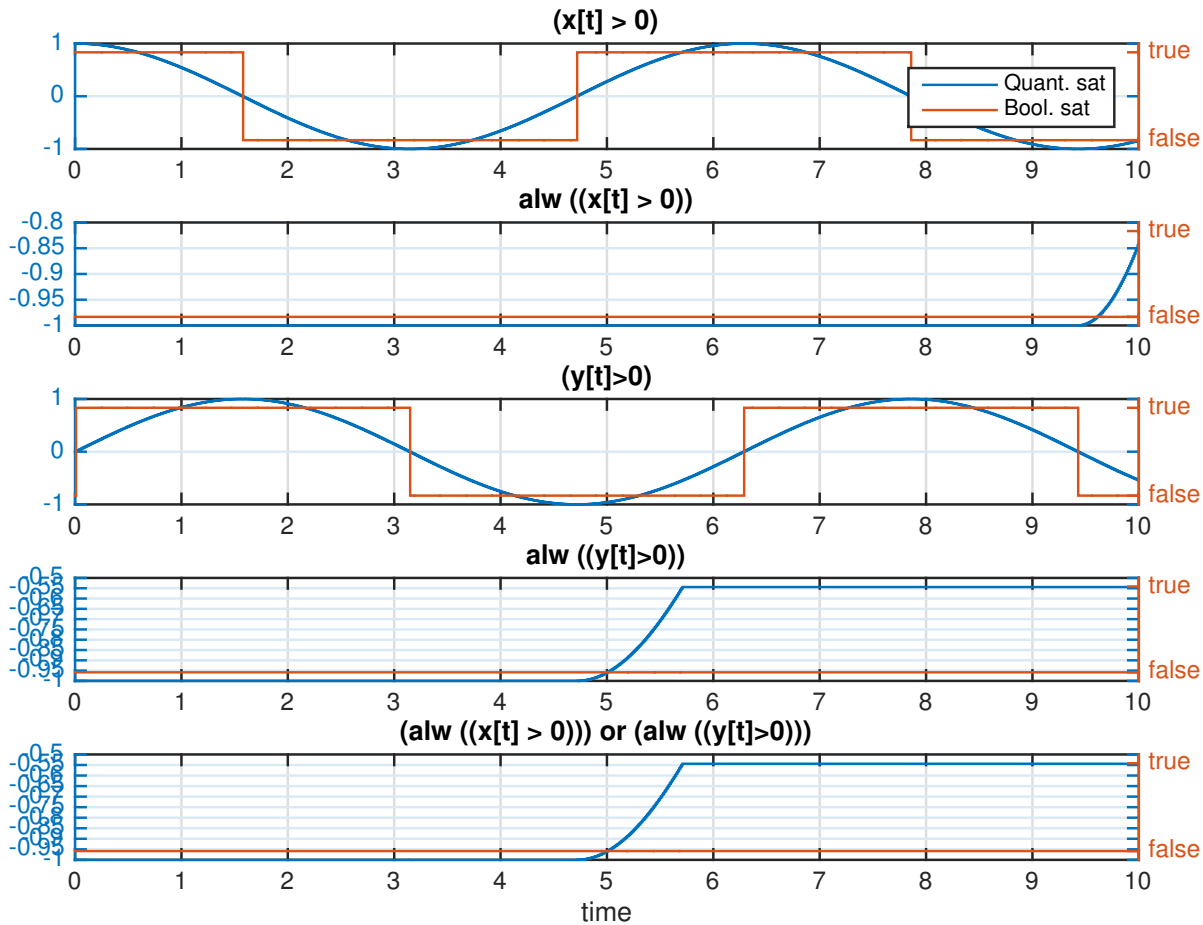
Checks (plots) some formula on imported trace:

```
BrTrace.PlotRobustSat('alw (x[t] > 0) or alw (y[t]>0)')
```

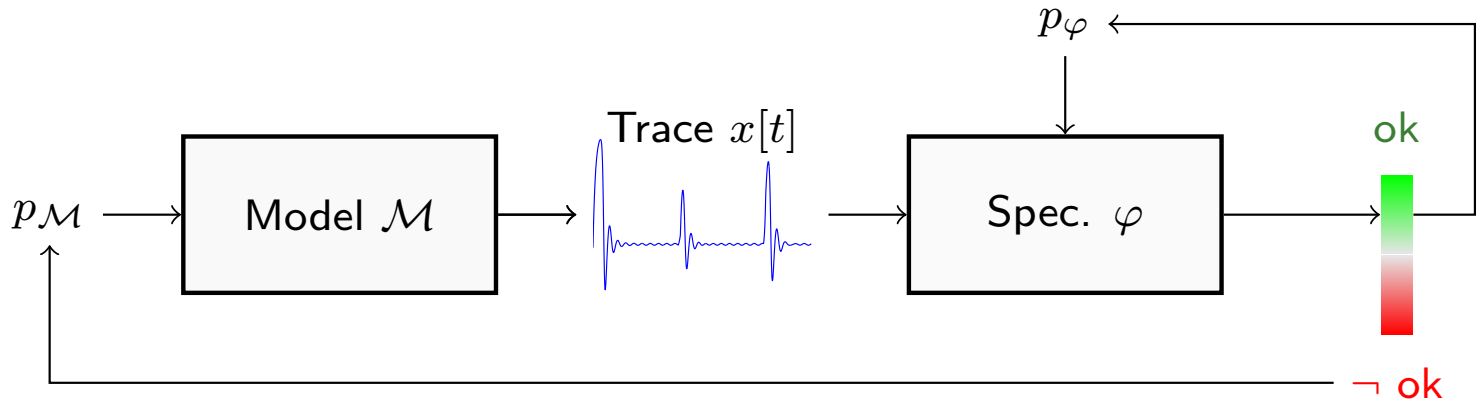
Monitoring a formula on a trace

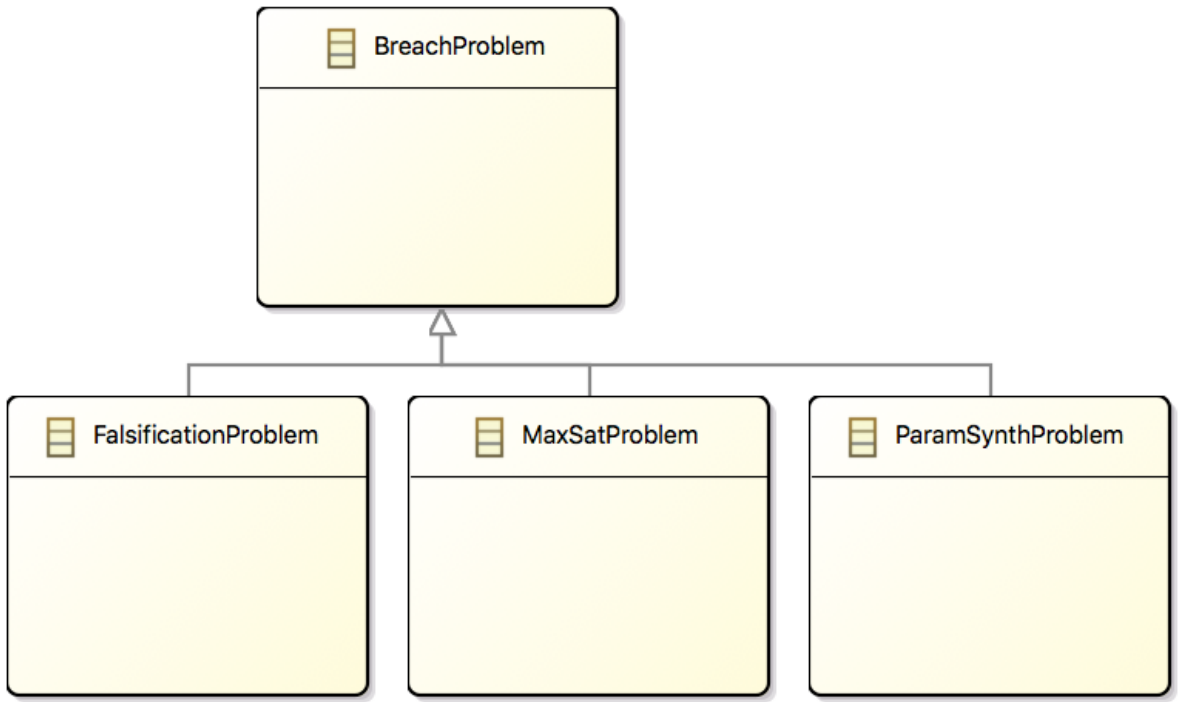
Checks (plots) some formula on imported trace:

```
BrTrace.PlotRobustSat('alw (x[t] > 0) or alw (y[t]>0)')
```



Solving Problems





Creating Falsification Problems

Given a specification φ and some parameter range, we want to find a parameter value for which the system violates φ.
First we create the parameter search domain and load specifications.

```
AFC_Falsify = BrAFC.copy();
AFC_Falsify.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'}, [10 20; 10 60]);
STL_ReadFile('simple_spec.stl');
```

Then we create the falsification problem and solve it.

```
falsif_pb = FalsificationProblem(AFC_Falsify, AF_alw_ok);
res = falsif_pb.solve();
```

Creating Falsification Problems

Given a specification φ and some parameter range, we want to find a parameter value for which the system violates φ.
First we create the parameter search domain and load specifications.

```
AFC_Falsify = BrAFC.copy();
AFC_Falsify.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'}, [10 20; 10 60]);
STL_ReadFile('simple_spec.stl');
```

Then we create the falsification problem and solve it.

```
falsif_pb = FalsificationProblem(AFC_Falsify, AF_alw_ok);
res = falsif_pb.solve();
```

```
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
    3      +6.49516e-02      +6.49516e-02          2.9

----- Falsified with obj=-0.0722088 at
        Pedal_Angle_pulse_period = 20
        Pedal_Angle_pulse_amp = 60
```

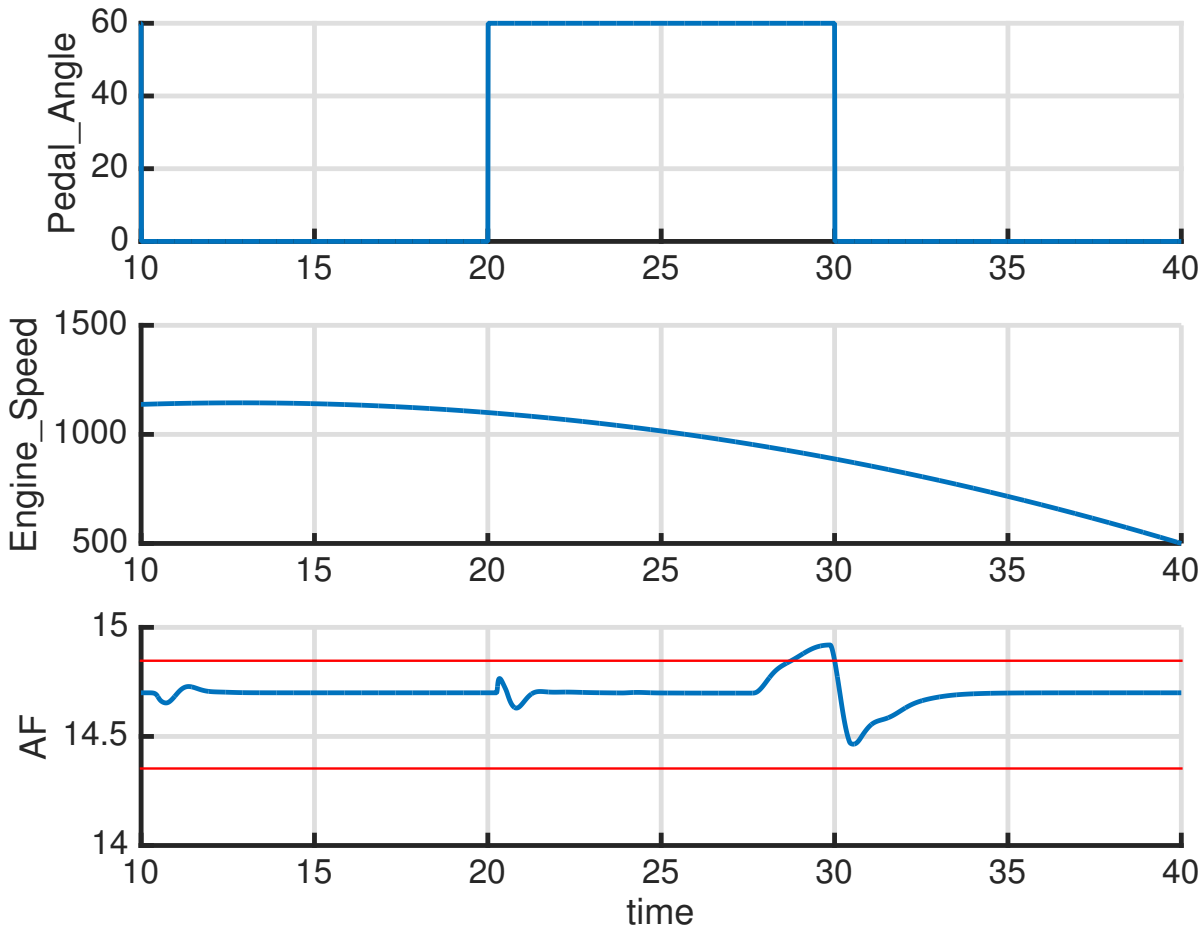
That's it. The default solver found a trace violating the specification AF_alw_ok.

Getting and Plotting the Falsifying Trace

```
AFC_False = falsif_pb.GetBrSet_False(); % AFC_False contains the falsifying trace
AFC_False.PlotSignals({'Pedal_Angle','Engine_Speed','AF'}, [], {'LineWidth', 2});
% Some figure cosmetics
subplot(3,1,1); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
subplot(3,1,2); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
subplot(3,1,3); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1-0.01)*[14.7 14.7]-0.2,'r');
```

Getting and Plotting the Falsifying Trace

```
AFC_False = falsif_pb.GetBrSet_False(); % AFC_False contains the falsifying trace
AFC_False.PlotSignals({'Pedal_Angle','Engine_Speed','AF'}, [], {'LineWidth', 2});
% Some figure cosmetics
subplot(3,1,1); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
subplot(3,1,2); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
subplot(3,1,3); set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1-0.01)*[14.7 14.7]-0.2,'r');
```



Getting and Plotting the Falsifying Trace

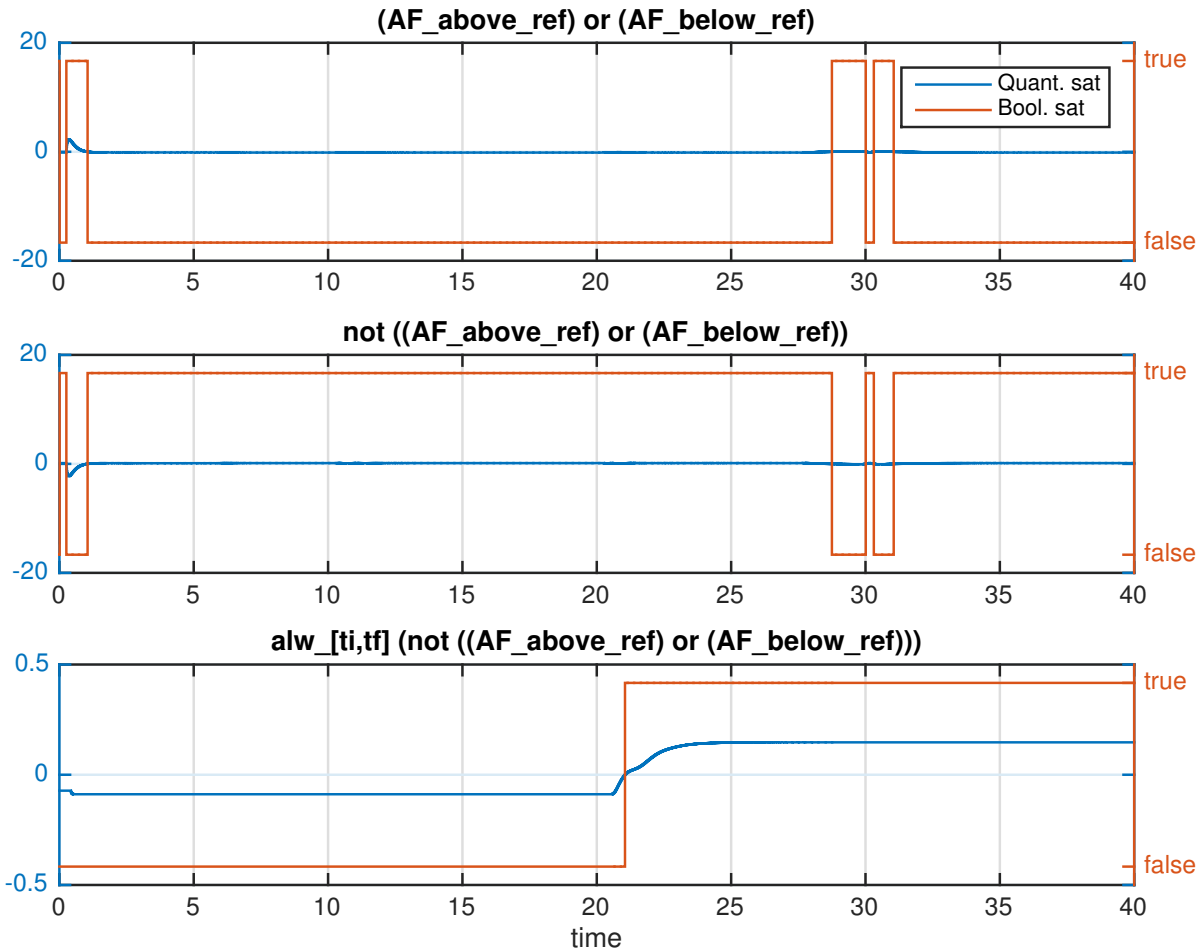
We can also examine more closely at the violation by plotting the robust satisfaction function for the formula and its subformulas:

```
AFC_False.PlotRobustSat(AF_alw_ok,3); % second argument is depth of formula decomposition
```

Getting and Plotting the Falsifying Trace

We can also examine more closely at the violation by plotting the robust satisfaction function for the formula and its subformulas:

```
AFC_False.PlotRobustSat(AF_alw_ok,3); % second argument is depth of formula decomposition
```



Trying harder falsification instances

We were able to falsify AF_alw_ok which says that AF should stay within 1% of AFref. What if we soften the requirement and make it 1.5% ?

```
AF_alw_ok2 = set_params(AF_alw_ok, 'tol', 0.015);
falsif_pb2 = FalsificationProblem(AFC_Falsify, AF_alw_ok2);
res = falsif_pb2.solve();
```

Trying harder falsification instances

We were able to falsify AF_alw_ok which says that AF should stay within 1% of AFref. What if we soften the requirement and make it 1.5% ?

```
AF_alw_ok2 = set_params(AF_alw_ok, 'tol', 0.015);
falsif_pb2 = FalsificationProblem(AFC_Falsify, AF_alw_ok2);
res = falsif_pb2.solve();
```

```
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
   20      +1.39226e-01      +1.29116e-03         17.4

No falsifying trace found.

----- Best value 0.00129116 found with
        Pedal_Angle_pulse_period = 20
        Pedal_Angle_pulse_amp = 60
```

BreachProblem Default Solver

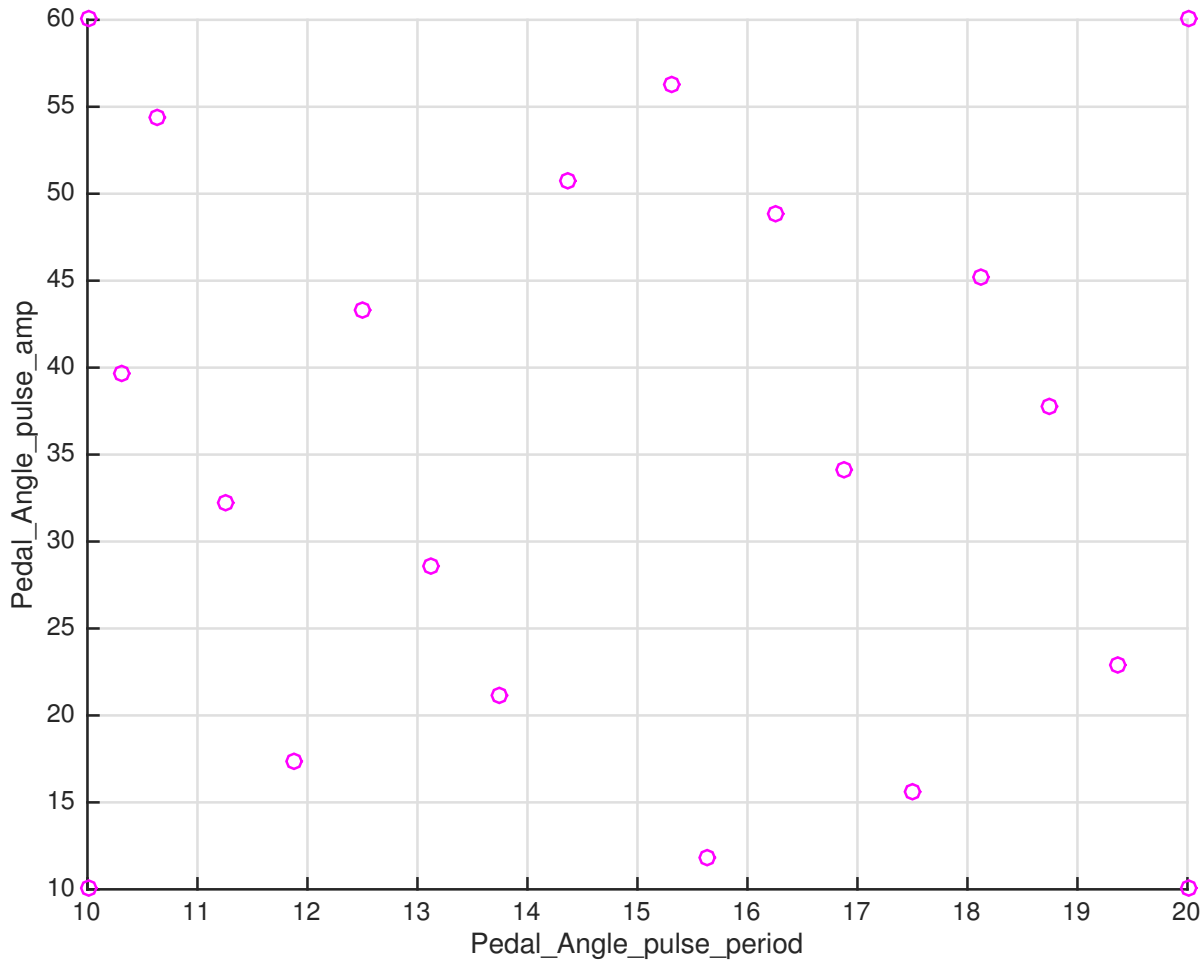
The default solver is very simple. It computes the objective function at the corners of the parameter domain and then uniform random finite sampling inside. We can examine which parameter value was tested as follows:

```
BrLogged = falsif_pb2.GetBrSet_Logged();
BrLogged.PlotParams();
```

BreachProblem Default Solver

The default solver is very simple. It computes the objective function at the corners of the parameter domain and then uniform random finite sampling inside. We can examine which parameter value was tested as follows:

```
BrLogged = falsif_pb2.GetBrSet_Logged();
BrLogged.PlotParams();
```



BreachProblem Default Solver

The basic solver has two nice features though: (1) it can be resumed and (2) It will eventually cover the search domain in a dense way.
It has two options:

```
falsif_pb2.solver_options
```

BreachProblem Default Solver

The basic solver has two nice features though: (1) it can be resumed and (2) It will eventually cover the search domain in a dense way.
It has two options:

```
falsif_pb2.solver_options
```

```
ans =

          step: 20
    nb_samples: 20
```

- the first is the number of steps to skip. It is 20 here because this is the number of simulations that were performed in the previous run.
- the second is the number of simulations to run in the next call to solve().

BreachProblem Default Solver

To resume falsification with the basic solver, we run solve() again.

```
falsif_pb2.solver_options.nb_samples = 100; % try harder with more samples
falsif_pb2.solve();
```

BreachProblem Default Solver

To resume falsification with the basic solver, we run solve() again.

```
falsif_pb2.solver_options.nb_samples = 100; % try harder with more samples
falsif_pb2.solve();
```

```
Eval objective function on 100 initial parameters.
#init      obj               best           time spent
   26      +1.22712e-01      +5.35440e-02         40.1

----- Falsified with obj=-0.00303559 at
        Pedal_Angle_pulse_period = 19.5312
        Pedal_Angle_pulse_amp = 47.6543
```

Other Solvers

We can get a list of available solvers by

```
BreachProblem.list_solvers()
```

Other Solvers

We can get a list of available solvers by

```
BreachProblem.list_solvers()
```

```
ans =

  Columns 1 through 5

    'init'    'basic (default)'    'fmincon'    'optimtool'    'binsearch'

  Column 6

    'cmaes'
```

Each of these solvers can be setup with default options using setup_solver()

E.g, CMA-ES is a popular evolutionary algorithm included in the list of solvers

```
falsif_pb3 = FalsificationProblem(AFC_Falsify, AF_alw_ok2,{'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, [10 20; 10 60]);
falsif_pb3.setup_solver('cmaes');
```

Other Solvers

We can get a list of available solvers by

```
BreachProblem.list_solvers()
```

```
ans =

  Columns 1 through 5

    'init'    'basic (default)'    'fmincon'    'optimtool'    'binsearch'

  Column 6

    'cmaes'
```

Each of these solvers can be setup with default options using setup_solver()

E.g, CMA-ES is a popular evolutionary algorithm included in the list of solvers

```
falsif_pb3 = FalsificationProblem(AFC_Falsify, AF_alw_ok2,{'Pedal_Angle_pulse_period', '
      Pedal_Angle_pulse_amp'}, [10 20; 10 60]);
falsif_pb3.setup_solver('cmaes');
```

```
Setting options for cmaes solver — use help cmaes for details
Default options returned (type "help cmaes" for help).
```

Running CMAES

```
falsif_pb3.solver_options.MaxFunEvals = 100; % max number of simulations
falsif_pb3.solve();
```

Running CMAES

```
falsif_pb3.solver_options.MaxFunEvals = 100; % max number of simulations
falsif_pb3.solve();
```

```
  n=2: (3,6)-CMA-ES(w=[64 28 8]%, mu_eff=2.0) on objective function
Iterat, #Fevals:   Function Value    (median,worst) |Axis Ratio|idx:Min SD idx:Max SD
    1 ,      7 : 1.3711399259636e-01 +(2e-03,6e-03) | 4.91e+00 |  1:2.3e+00  2:1.1e+01
    2 ,     13 : 1.2249843599852e-01 +(1e-02,3e-02) | 4.92e+00 |  1:2.1e+00  2:1.0e+01
   17 ,    103 : 1.2133189250765e-01 +(5e-03,2e-02) | 2.17e+00 |  1:3.8e-01  2:8.1e-01
#Fevals:   f(returned x)   |    bestever.f     | stopflag (saved to variablescmaes.mat)
    104: 1.21331892508e-01 | 1.21331892508e-01 | maxfunevals
mean solution: +1.3e+01 +3.2e+01
std deviation:  3.8e-01  8.1e-01

No falsifying trace found.

----- Best value 0.121332 found with
        Pedal_Angle_pulse_period = 12.5332
        Pedal_Angle_pulse_amp = 32.6172
```

Customizing/Interfacing a New Solver

New solvers can be interfaced (or new interface created) by creating a new class:

```
type MyFalsificationProblem.m
```

Customizing/Interfacing a New Solver

New solvers can be interfaced (or new interface created) by creating a new class:

```
type MyFalsificationProblem.m
```

```
classdef MyFalsificationProblem < FalsificationProblem
    % Example of a class deriving from FalsificationProblem specialized for one solver
    methods
        function this = MyFalsificationProblem(BrSys, phi)
            this = this@FalsificationProblem(BrSys, phi); % calls parent constructor
            this.StopAtFalse = false; % don't stop when a falsifying trace is found
        end

        % setup options — will be called by the constructor
        function setup_solver(this)
            this.solver_options = optimset('Display', 'iter');
        end

        % custom objective function — robust_fn is obtained by the constructor from BrSys and phi
        function fval = objective_fn(this,x)
            rob = min(this.robust_fn(x)); % note: robust_fn might return an array of values
            fval = rob*norm(x);  % also tries to maximize the norm of x
        end

        % call solver and return a structure with results.
        function res = solve(this)
            [x, fval] = fmincon(this.objective, ... % objective calls objective_fn plus some
                  bookkeeping
            this.x0, ... % obtained from BrSys
             [], [], [],[], ...   % no linear (in)—equalities
            this.lb, this.ub, ... % obtained by constructor from ranges
            [], this.solver_options);
            res = struct('x_best',x, 'fval', fval);
        end
    end
```

Customizing/Interfacing a New Solver

```
myfalsif_problem = MyFalsificationProblem(AFC_Falsify, AF_alw_ok);
myfalsif_problem.solve();
```

Customizing/Interfacing a New Solver

```
myfalsif_problem = MyFalsificationProblem(AFC_Falsify, AF_alw_ok);
myfalsif_problem.solve();
```

```
                                        First-order      Norm of
 Iter F-count            f(x)  Feasibility   optimality         step
    0       3    2.344048e+00    0.000e+00    2.380e+04
    1      10    2.108942e+00    0.000e+00    1.615e+05    1.412e+00
    2      15    2.087242e+00    0.000e+00    1.127e+06    7.060e-01
    3      18    2.068179e+00    0.000e+00    7.320e+04    6.176e-01
    4      23    2.062973e+00    0.000e+00    7.172e+04    1.194e-02
    5      27    2.059490e+00    0.000e+00    5.962e+04    1.182e-01
    6      30    2.033455e+00    0.000e+00    1.044e+05    5.905e-01
    7      34    2.022373e+00    0.000e+00    4.053e+03    1.837e-01
    8      39    1.989718e+00    0.000e+00    1.642e+05    5.295e-03
    9      44    1.973150e+00    0.000e+00    8.978e+02    3.512e-01
   10      48    1.971169e+00    0.000e+00    1.635e-01    8.261e-03
   11      52    1.971020e+00    0.000e+00    8.001e+04    7.731e-03
   12      60    1.970363e+00    0.000e+00    1.232e+03    3.725e-09
   13      64    1.970363e+00    0.000e+00    1.232e+03    2.511e-08
   14      68    1.953517e+00    0.000e+00    1.057e+00    9.368e-01
   15      72    1.948591e+00    0.000e+00    6.813e-01    2.731e-01
   16      75    1.944553e+00    0.000e+00    1.005e-01    2.268e-01

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the default value of the step size tolerance and constraints are
satisfied to within the default value of the constraint tolerance.
```

Parameter Synthesis

We falsified the property that AF stays within tol=0.01 of AF_ref. We might also ask: for what minimum value of tol is this property true?

This can be solved by solving a parameter synthesis problem:

```
AFC_ParamSynth = BrAFC.copy(); AFC_ParamSynth.Sim;
synth_pb = ParamSynthProblem(AFC_ParamSynth, AF_alw_ok, {'tol'},  [0 0.1]);
synth_pb.solve();
```

Parameter Synthesis

We falsified the property that AF stays within tol=0.01 of AF_ref. We might also ask: for what minimum value of tol is this property true?

This can be solved by solving a parameter synthesis problem:

```
AFC_ParamSynth = BrAFC.copy(); AFC_ParamSynth.Sim;
synth_pb = ParamSynthProblem(AFC_ParamSynth, AF_alw_ok, {'tol'}, [0 0.1]);
synth_pb.solve();
```

```
Inferring monotonicity...
The problem appears to be monotonic in
tol (increasing)

Finding tight tol           0.0066204

------ Best objective value found: 1.13531e-06, with
       tol = 0.00662041
```

The default solver guessed that the problem was monotonic in 'tol'. Monotonicity is actually required by this solver.
If monotonicity cannot be guessed it can be set up manually:

```
synth_pb.solver_options.monotony = [1]; % +1/-1 resp. increasing/decreasing
```

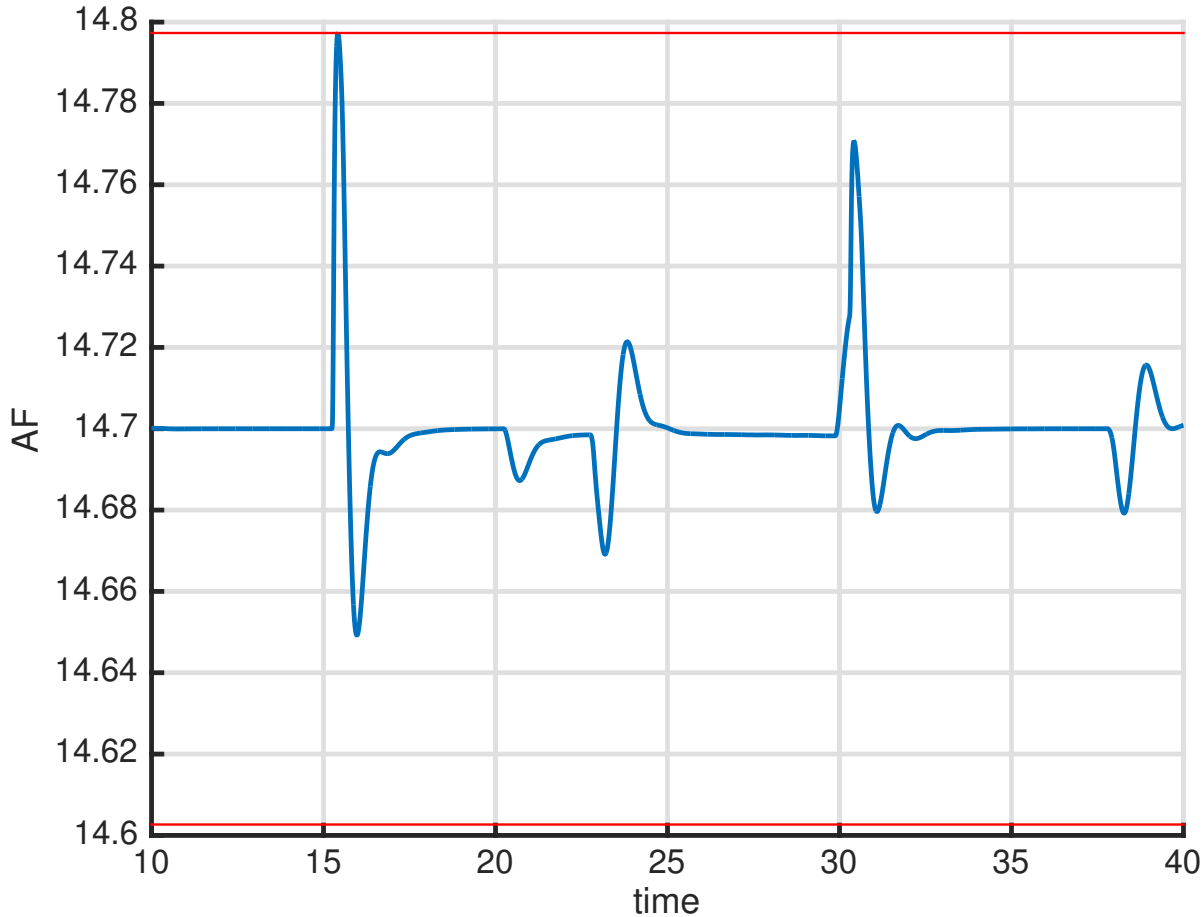
If the problem is not monotonic, another solver has to be chosen.

Parameter Synthesis - Plot

```
AFC_ParamSynth.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);

% plotting the new tolerance region
tol_best  = synth_pb.x_best;
plot([0 41], (1+tol_best)*[14.7 14.7],'r'); plot([0 41], (1-tol_best)*[14.7 14.7],'r');
```

Parameter Synthesis - Plot

```
AFC_ParamSynth.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);

% plotting the new tolerance region
tol_best  = synth_pb.x_best;
plot([0 41], (1+tol_best)*[14.7 14.7],'r'); plot([0 41], (1-tol_best)*[14.7 14.7],'r');
```



Parameter Synthesis For Multiple Traces

Parameter synthesis can be solved for multiple traces.

```
AFC_ParamSynth.SetParamRanges({'Pedal_Angle_pulse_period','Pedal_Angle_pulse_amp'},[10 20;10 60]);
AFC_ParamSynth.QuasiRandomSample(10);
AFC_ParamSynth.Sim();
synth_pb = ParamSynthProblem(AFC_ParamSynth, AF_alw_ok, {'tol'},  [0 0.1]);
synth_pb.solve();
```

Parameter Synthesis For Multiple Traces

Parameter synthesis can be solved for multiple traces.

```
AFC_ParamSynth.SetParamRanges({'Pedal_Angle_pulse_period','Pedal_Angle_pulse_amp'},[10 20;10 60]);
AFC_ParamSynth.QuasiRandomSample(10);
AFC_ParamSynth.Sim();
synth_pb = ParamSynthProblem(AFC_ParamSynth, AF_alw_ok, {'tol'},  [0 0.1]);
synth_pb.solve();
```

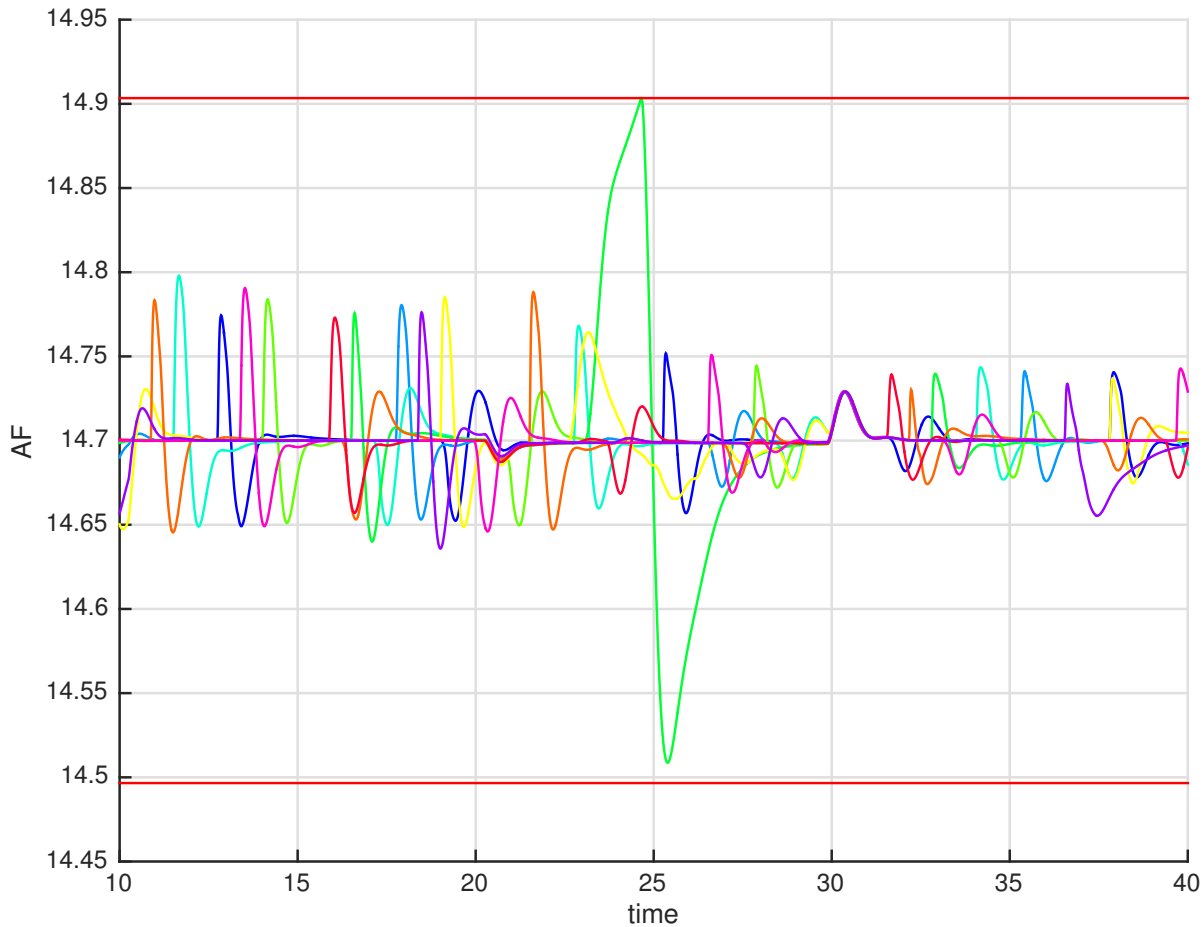
```
Computing 10 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Inferring monotonicity...
The problem appears to be monotonic in
tol (increasing)

Finding tight tol           0.013837

 ————— Best objective value found: 5.39365e—06, with
        tol = 0.0138374
```

Parameter Synthesis For Multiple Traces - Plot

```
AFC_ParamSynth.PlotSignals({'AF'});
set(gca, 'XLim', [10 40]);
% plotting the new tolerance region
tol_best  = synth_pb.x_best;
plot([0 41], (1+tol_best)*[14.7 14.7],'r'); plot([0 41], (1-tol_best)*[14.7 14.7],'r');
```

Parameter Synthesis For Multiple Traces - Plot

```
AFC_ParamSynth.PlotSignals({'AF'});
set(gca, 'XLim', [10 40]);
% plotting the new tolerance region
tol_best  = synth_pb.x_best;
plot([0 41], (1+tol_best)*[14.7 14.7],'r'); plot([0 41], (1-tol_best)*[14.7 14.7],'r');
```



Maximizing Satisfaction

Instead of minimizing satisfaction for falsification, we might want to maximize it to make it positive and as robust as possible.
One possible use case is tuning control parameters.

```
AFC_MaxSat = BrAFC.copy();
% Create the max sat problem and solve it
max_sat_problem = MaxSatProblem(AFC_MaxSat, AF_alw_ok,...
                            {'ki', 'kp'}, ...       % we look for a better PI controller.
                            [0. 0.3; 0.01 0.1]);
max_sat_problem.solve();
```

Maximizing Satisfaction

Instead of minimizing satisfaction for falsification, we might want to maximize it to make it positive and as robust as possible.
One possible use case is tuning control parameters.

```
AFC_MaxSat = BrAFC.copy();
% Create the max sat problem and solve it
max_sat_problem = MaxSatProblem(AFC_MaxSat, AF_alw_ok,...
                            {'ki', 'kp'}, ...       % we look for a better PI controller.
                            [0. 0.3; 0.01 0.1]);
max_sat_problem.solve();
```

```
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
   20      −5.26462e−02      −5.26462e−02         17.7

 ──── Best robustness value 0.0526462 found at
        ki = 0.159375
        kp = 0.0933333
```

The best robustness value obtained is positive, which means we found 'better' parameters for PI controller that satisfy the specification.

Maximizing Satisfaction - Plot

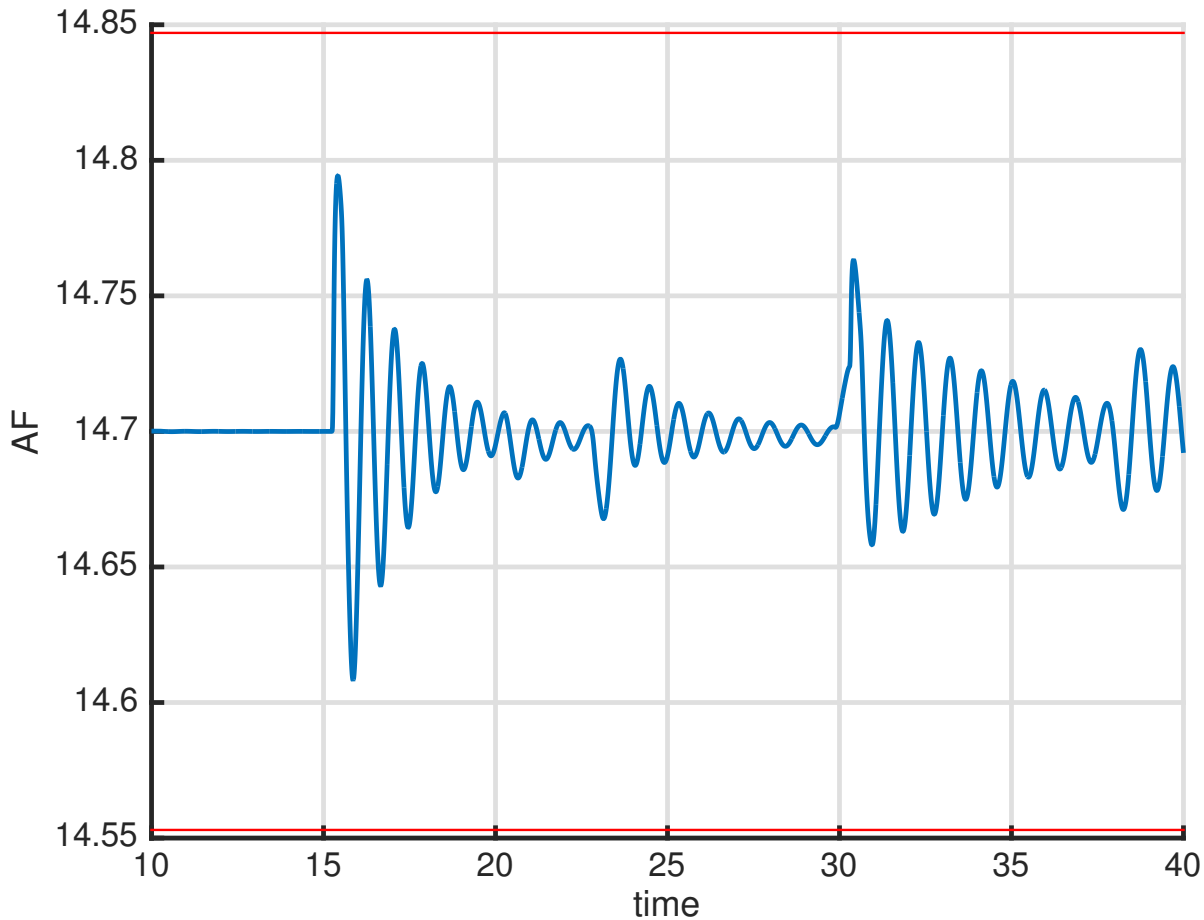
Maybe it is not so good though.

```
AFC_Best = max_sat_problem.GetBrSet_Best();
AFC_Best.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1—0.01)*[14.7 14.7],'r');
```

Maximizing Satisfaction - Plot

Maybe it is not so good though.

```
AFC_Best = max_sat_problem.GetBrSet_Best();
AFC_Best.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1—0.01)*[14.7 14.7],'r');
```



Maximizing Satisfaction - Plot

We maximized the satisfaction of AF_alw_tol, but this resulted in an undesired oscillatory behavior. To fix this, we use a new formula that requires that AF settles for at least 1s every 10s.

```
STL_ReadFile('settling_spec.stl')
type settling_spec.stl
```

Maximizing Satisfaction - Plot

We maximized the satisfaction of AF_alw_tol, but this resulted in an undesired oscillatory behavior. To fix this, we use a new formula that requires that AF settles for at least 1s every 10s.

```
STL_ReadFile('settling_spec.stl')
type settling_spec.stl
```

```
ans =

    'AF_settled'    'AF_stable'    'AF_will_be_stable'    'AF_alw_settle'


# settling
param dt=0.1,epsi=1e-2

AF_settled := abs(AF[t+dt]-AF[t])<epsi*dt
AF_stable := alw_[0, 1] (AF_settled)
AF_will_be_stable := ev_[0,9] (AF_stable)
AF_alw_settle := alw_[10, 30] (AF_will_be_stable)
```

Maximizing Satisfaction - Plot

Defining and solving the max sat problem with settling property

```
AFC_MaxSat = BrAFC.copy();
% Create the max sat problem and solve it
max_sat_problem = MaxSatProblem(AFC_MaxSat, AF_alw_settle,...
                            {'ki', 'kp'}, ...
                            [0. 0.2; 0.0 0.05]);
max_sat_problem.solve();
```

Maximizing Satisfaction - Plot

Defining and solving the max sat problem with settling property

```
AFC_MaxSat = BrAFC.copy();
% Create the max sat problem and solve it
max_sat_problem = MaxSatProblem(AFC_MaxSat, AF_alw_settle,...
                           {'ki', 'kp'}, ...
                           [0. 0.2; 0.0 0.05]);
max_sat_problem.solve();
```

```
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
   20      -9.33237e-04      -9.83003e-04         16.7

------ Best robustness value 0.000983003 found at
        ki = 0.1625
        kp = 0.0351852
```

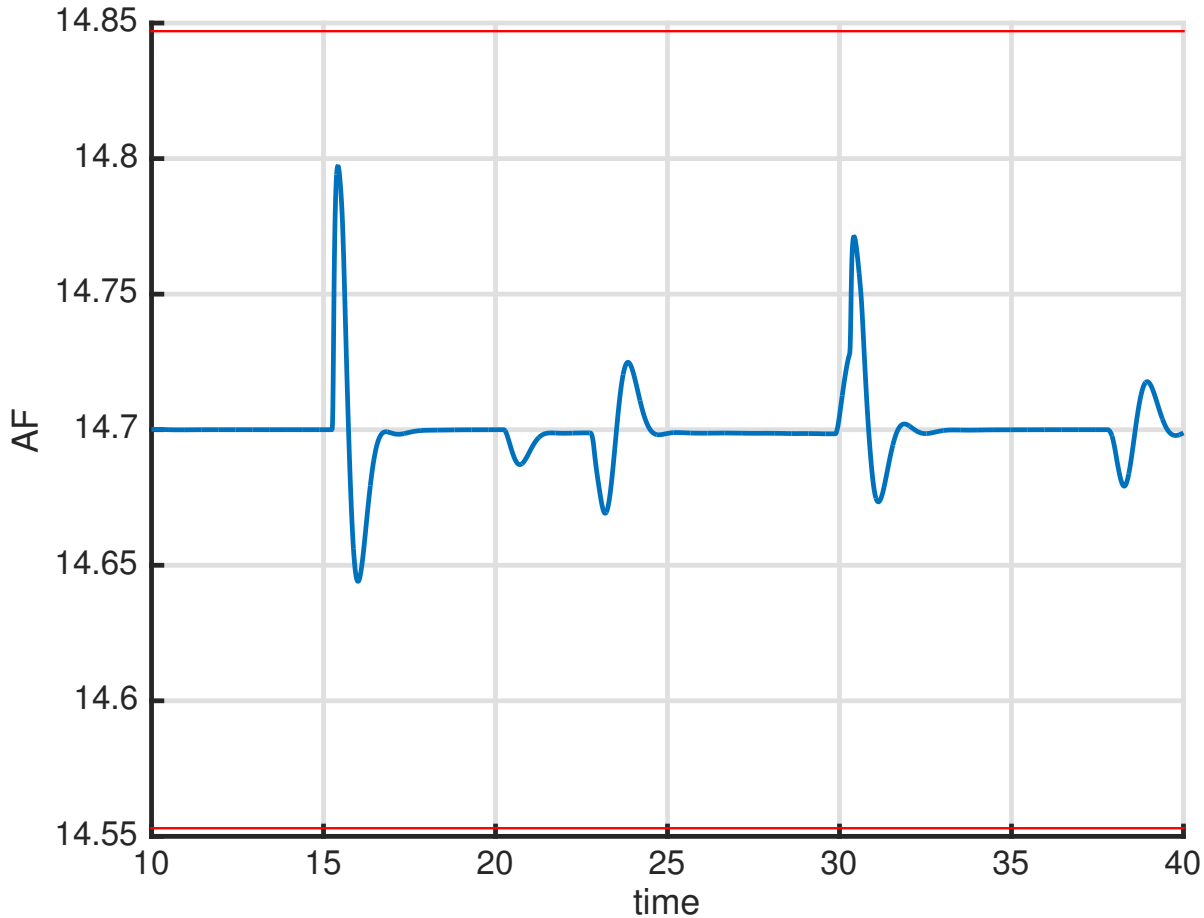
The solver found a satisfactory solution that should behave better.

Maximizing Satisfaction - Plot

```
AFC_Best = max_sat_problem.GetBrSet_Best();
AFC_Best.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1—0.01)*[14.7 14.7],'r');
```

Maximizing Satisfaction - Plot

```
AFC_Best = max_sat_problem.GetBrSet_Best();
AFC_Best.PlotSignals({'AF'}, [], {'LineWidth', 2});
set(gca, 'XLim', [10 40], 'FontSize',14, 'LineWidth', 2);
plot([0 41], (1+0.01)*[14.7 14.7],'r');
plot([0 41], (1—0.01)*[14.7 14.7],'r');
```



Maximizing Satisfaction For Multiple Inputs

Tuning PI for only one input is likely not sufficient. Below we try to find a PI controller for a set of 9 inputs.

```
AFC_Grid = BrAFC.copy();
AFC_Grid.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'}, [10, 20; 10 60]);
AFC_Grid.GridSample([3 3]);

max_sat_problem_grid = MaxSatProblem(AFC_Grid, AF_alw_settle, {'ki', 'kp'}, [0. 0.2; 0.0 0.05]);
max_sat_problem_grid.solve();
```

Maximizing Satisfaction For Multiple Inputs

Tuning PI for only one input is likely not sufficient. Below we try to find a PI controller for a set of 9 inputs.

```
AFC_Grid = BrAFC.copy();
AFC_Grid.SetParamRanges({'Pedal_Angle_pulse_period', 'Pedal_Angle_pulse_amp'}, [10, 20; 10 60]);
AFC_Grid.GridSample([3 3]);

max_sat_problem_grid = MaxSatProblem(AFC_Grid, AF_alw_settle, {'ki', 'kp'}, [0. 0.2; 0.0 0.05]);
max_sat_problem_grid.solve();
```

```
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
   20      +7.62496e-04      +2.95760e-04        155.6

----- Best robustness value -0.00029576 found at
        ki = 0.0625
        kp = 0.0185185
```

If the solver doesn't find a solution, we can again try another one, increase the simulation budget, change the problem, etc.

Requirement Mining

We falsified AF_alw_ok, then updated the tol parameter to make it true again.
A natural next step would be falsify the property with new tolerance, and iterate. We can automate this process using a ReqMiningProblem.

```
AFC_ReqMining = BrAFC.copy();

% Input parameters names and ranges for falsification
input_params = struct('names',{{'Pedal_Angle_pulse_period','Pedal_Angle_pulse_amp'}}, ...
                      'ranges', [10 20; 10 60]);

% Property with parameters names and ranges for paramter synthesis
STL_ReadFile('simple_spec.stl'); phi = AF_alw_ok;
prop_params = struct('names', {{'tol'}}, 'ranges', [0 0.1]);
```

Requirement Mining

A ReqMiningProblem is an object combining a FalsificationProblem object and a ParamSynthesis object.

```
mine_pb = ReqMiningProblem(AFC_ReqMining, phi, input_params, prop_params)
```

Requirement Mining

A ReqMiningProblem is an object combining a FalsificationProblem object and a ParamSynthesis object.

```
mine_pb = ReqMiningProblem(AFC_ReqMining, phi, input_params, prop_params)
```

```
mine_pb =

  ReqMiningProblem with properties:

     synth_pb: [1x1 ParamSynthProblem]
    falsif_pb: [1x1 FalsificationProblem]
         iter: 0
     iter_max: 10
      verbose: 1
```

We can change options for both problems as usual. E.g., specify monotonicity:

```
mine_pb.synth_pb.solver_options.monotony = 1;
```

Requirement Mining - Solving

```
mine_pb.solve();
```

Requirement Mining - Solving

```
mine_pb.solve();
```

```
Iter 0/10
Synthesis step
---------------
Finding tight tol           0.0066204

 ----- Best objective value found: 1.13531e-06, with
        tol = 0.00662041

Counter-Example step
--------------------
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
    3      +1.52716e-02      +1.52716e-02          2.6

 ----- Falsified with obj=-0.121889 at
        Pedal_Angle_pulse_period = 20
        Pedal_Angle_pulse_amp = 60

Iter 1/10
Synthesis step
---------------
Finding tight tol           0.014912

 ----- Best objective value found: 3.65022e-06, with
        tol = 0.0149124

Counter-Example step
--------------------
Eval objective function on 20 initial parameters.
#init      obj               best           time spent
   20      +1.22929e-01      +5.22565e-02         16.6

No falsifying trace found.
```

Requirement Mining

The mining ends when the falsifier fails. Though, we can change the falsifier and resume mining where it stops for more confidence:

```
mine_pb.falsif_pb.solver_options.nb_samples = 100;
mine_pb.solve(); % restart
```

Requirement Mining

The mining ends when the falsifier fails. Though, we can change the falsifier and resume mining where it stops for more confidence:

```
mine_pb.falsif_pb.solver_options.nb_samples = 100;
mine_pb.solve(); % restart
```

```
Iter 1/10
Synthesis step
--------------
Finding tight tol           0.014912

 ----- Best objective value found: 3.65022e-06, with
        tol = 0.0149124

Counter-Example step
--------------------
Eval objective function on 100 initial parameters.
#init      obj               best           time spent
    6      +1.21424e-01      +1.21424e-01          5.0

 ----- Falsified with obj=-0.0043231 at
        Pedal_Angle_pulse_period = 19.5312
        Pedal_Angle_pulse_amp = 47.6543

Iter 2/10
Synthesis step
--------------
Finding tight tol           0.015206

 ----- Best objective value found: 5.97141e-06, with
        tol = 0.0152069

Counter-Example step
--------------------
Eval objective function on 100 initial parameters.
```

Plotting Satisfaction Maps

We can visualize the satisfaction of a formula when varying parameters. For example:

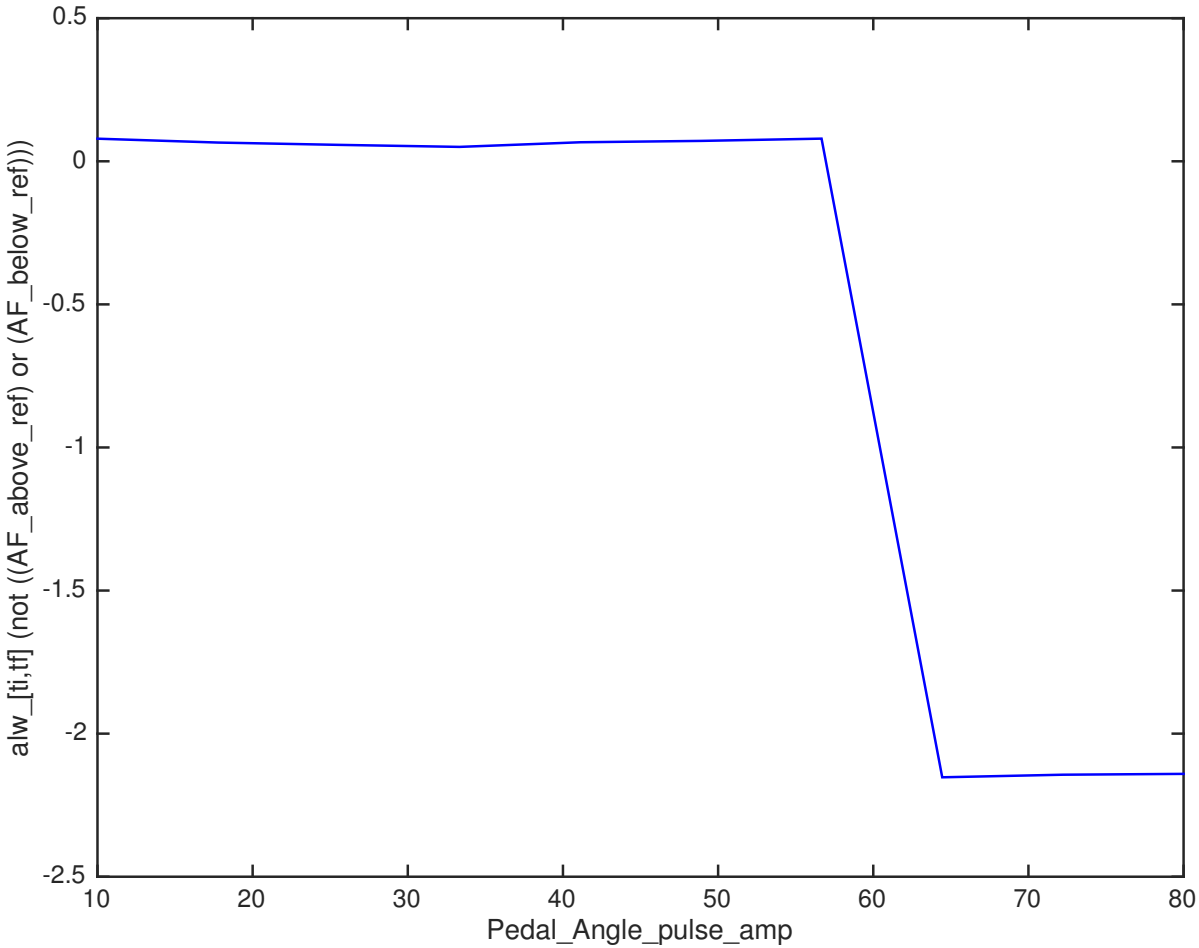
```
BrMap = BrAFC.copy();
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp'}, [10 80]);
```

Plotting Satisfaction Maps

We can visualize the satisfaction of a formula when varying parameters. For example:

```
BrMap = BrAFC.copy();
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp'}, [10 80]);
```

```
Computing 10 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Plotting Satisfaction Maps

The Robust Map can be plotted as a surface plot, varying two parameters:

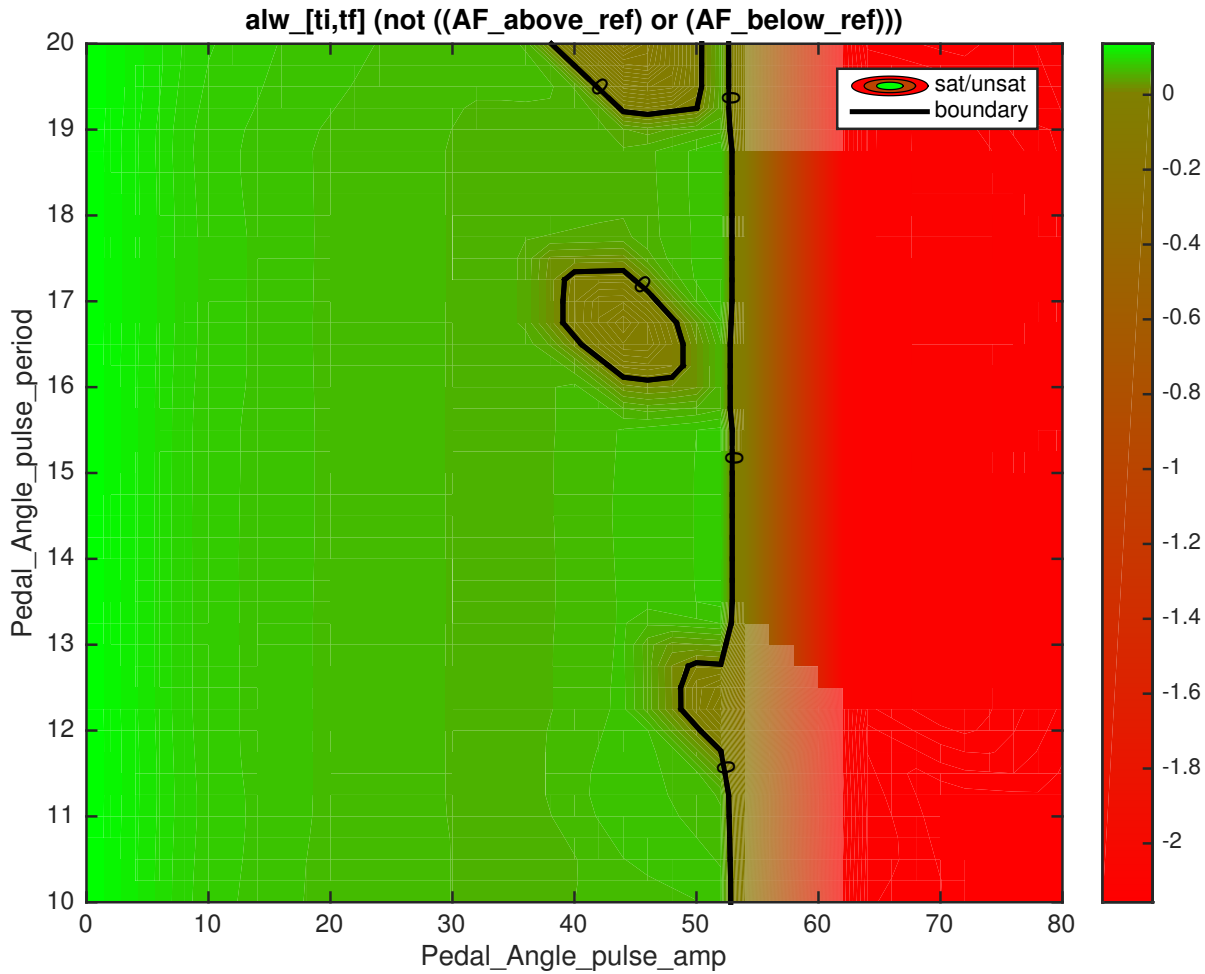
```
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp', 'Pedal_Angle_pulse_period'}, [0 80; 10 20])
      ;
```

Plotting Satisfaction Maps

The Robust Map can be plotted as a surface plot, varying two parameters:

```
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp', 'Pedal_Angle_pulse_period'}, [0 80; 10 20])
      ;
```

```
Computing 100 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Plotting Satisfaction Maps

We can zoom in to get more details. The last argument of PlotRobustMap can be used to get a more refined grid too (here 15×15).

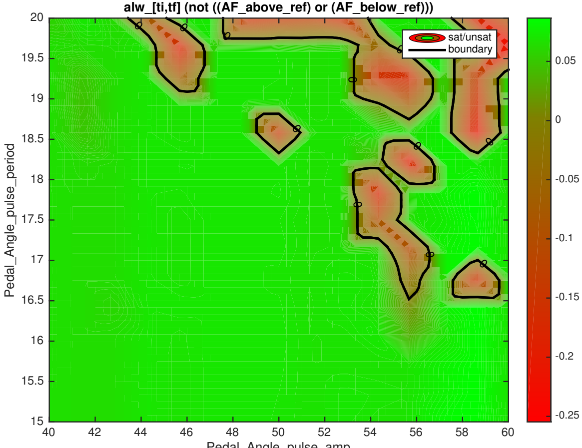
```
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp', 'Pedal_Angle_pulse_period'}, [40 60; 15
      20], [15 15]);
```

Plotting Satisfaction Maps

We can zoom in to get more details. The last argument of PlotRobustMap can be used to get a more refined grid too (here 15x15).

```
BrMap.PlotRobustMap(AF_alw_ok, {'Pedal_Angle_pulse_amp', 'Pedal_Angle_pulse_period'}, [40 60; 15
      20], [15 15]);
```

```
Computing 225 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```



Sensitivity Analysis

```
BrSensi = BrAFC.copy();
params = {'AF_sensor_tol','MAF_sensor_tol','fuel_inj_tol','kappa_tol','tau_ww_tol','pump_tol'};
ranges = [ 0.99 1.01; 0.99 1.01; 0.99 1.01;0.99 1.01; 0.99 1.01;0.99 1.01];
BrSensi.SensiSpec(AF_alw_ok, params, ranges);
```

Sensitivity Analysis

```
BrSensi = BrAFC.copy();
params = {'AF_sensor_tol','MAF_sensor_tol','fuel_inj_tol','kappa_tol','tau_ww_tol','pump_tol'};
ranges = [ 0.99 1.01; 0.99 1.01; 0.99 1.01;0.99 1.01; 0.99 1.01;0.99 1.01];
BrSensi.SensiSpec(AF_alw_ok, params, ranges);
```

```
Computing 70 trajectories of model AbstractFuelControl_M1_breach
[             25%           50%            75%               ]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```