IO Robustness Feature

# Context

## Motivation

The robustness value of a system is hard to capture. One useful metric is the quantitative semantics (or robustness) of a given trace relative to a Signal Temporal Logic (STL) specification. However, this metric can be difficult to interpret, because a single real value does not tell with respect to what quantity (input, output) or even in which physical unit the robustness is measured.

## Objective

The proposed Breach feature of IO Robustness computation, which we describe in this document, attempts to remedy this situation by focusing the analysis on a subset of the signal variables, either *input* variables or *output* variables.

# Description

## Principle

Given a set of variables, when computing the robustness in variables the other variables are treated qualitatively. We consider two possible qualitative treatments: either as having zero () robustness, or as having infinite () robustness. These interpretations are called *absolute* and *relative* robustness, respectively.

The analysis we implemented yields 4 new robustness measures:

* Input robustness, absolute;
* Input robustness, relative to fixed output;
* Output robustness, absolute;
* Output robustness, relative to fixed input.

## Interpretation

The relative robustness gives a safe approximation of the distance to satisfaction/violation based on a subset of signal variables, considering other variables fixed. The absolute robustness gives a safe approximation of this distance considering arbitrary changes in other variables.

The notion of relative robustness is most intuitive, but when input and output both contribute to the satisfaction/violation in an essential manner it cannot be measured. In this case the relative robustness does not provide a quantitative truth value, but the absolute robustness may.

For output variables, relative robustness and absolute robustness capture two extreme assumptions about the system.

1. Relative: the output is unaffected by under input perturbations;
2. Absolute: the output can change in an arbitrary fashion with input perturbations.

# Implementation

We provided Breach with a proof-of-concept implementation of the IO robustness feature. The implementation provides two (almost independent) enhancements to Breach. The first enhancement associates with an STL formula two lists of input and output signals. The second enhancement consists in computing the robustness according to inputs or outputs, using the relative or absolute interpretation.

## Principles

We modified the parsing of STL formulas to include information about input and output signals, which is then associated with every STL subformula. This way the names of input and output signals can be queried at any point when parsing an STL formula, typically during the robustness computation itself. This was done by directly modifying the STL formula constructor and class to include IO information, and by implementing new setter and getter function to access this IO information.

The robustness computation involves additional parameters controlling the set of signals over which the analysis is focused (*input* or *output*) and the interpretation (*relative* or *absolute*), and changing the semantics of atomic predicates that are not the focus of the analysis to either zero or infinity. This was done by modifying the Breach System class to add additional methods suffixed with IO and accepting the additional parameters. This is then propagated to the core computation (via plotting subroutines) in the evaluation functions, which are also duplicated with IO suffix and accepting the additional parameters.

## Use model

### Declare interface in requirement file

The intended use of the IO robustness indicators is as follows. The user adds information about input and output signals in the .stl file. The syntax of .stl files have been extended with the keywords

input signal   
output signal

Then the file can be parsed using STL\_Readfile. Signals whose declaration is not decorated with IO labels will be treated as normal, and those decorated with IO labels will be additionally marked as input or output.

Alternatively, the user may call the functions:

set\_in\_signal\_names(phi, X)   
set\_out\_signal\_names(phi, Y)

over the formula phi and a cell array of signal names X or Y. If needed, the corresponding getter methods can be used to show lists of input and output signals.

### Compute and plot IO robustness

The robustness can be computed by using the method

GetIORobustSat(inout, relabs, phi, params, values, t\_phi)

of the class BreachSystem, where the new fields inout and relabs are string switches with possible values ‘in’, ‘out’ and ‘rel’, ‘abs’ respectively.

The robustness can also be computed and displayed using the method

PlotIORobustSat(phi, inout, relabs, depth, tau, ipts)

of the class BreachSystem, where depth, tau, ipts are optional parameters and inout and relabs are strings with the meaning as previously.

### Use IO robustness for falsification

The four notions of input/output robustness can be used either in isolation or in combination to drive the Breach falsification of a Simulink system. We considered four different ways to define a falsification problem with IO robustness indicators. Before calling the solve method on some Breach falsification problem, one sets the objective function and constraints using the method set\_IO\_robustness\_mode with one string argument specifying the mode, among the following values: ‘default’, ‘random’, ‘in’, ‘out’, ‘constrained’, or ‘combined’. They have the following signification:

* ‘default’: set the standard robustness as objective function.
* ‘random’: set the Boolean satisfaction value as the objective function. Since this provides no meaningful information to the solver in the case where all values are true, the behavior of the solver will default to random sampling of the parameter space. This mode is provided as a baseline for comparision.
* ‘in’: set the relative input robustness as objective function.
* ‘out’: set the relative output robustness as objective function.
* ‘constrained: set the relative output robustness as objective function and set the constraint to that the absolute input robustness should be nonnegative.
* ‘combined’: use as objective function. This objective function is an angle in rad with the following meaning:
  + if then the formula is vacuously false;
  + if then the formula is nonvacuously false;
  + if then the formula is nonvacuously true;
  + if then the formula is vacuously true;

## Changes to Breach Codebase

### Files modified

**./@STL\_Formula/STL\_Formula.m**

We added the ability to store names of input and output signals, in each subformula. The following fields were added to the STL\_Formula class:

in\_signal\_names = {};

out\_signal\_names = {};

**./Core/STL\_ReadFile.m**

Added support for parsing the STL formula from file and treats additional keywords input signal and output signal.

**./Core/BreachSystem.m**

We added the functions GetIORobustSat, GetIORobustSatFn, and PlotIORobustSat. These are copies of GetRobustSat, GetRobustSatFn, and PlotRobustSat respectively. The new functions take additional arguments inout and relabs that are strings valued as ‘in’ or ‘out’, ‘rel’ or ‘abs’ respectively, and call I/O aware robustness computation functions with matching arguments.

**./Core/BreachRequirement.m**

We added the functions Eval\_IO and evalAllTracesIO. These are copies of Eval and evallAllTraces respectively. The new functions take additional arguments inout and relabs as for methods of BreachSystem.

**./Core/OutputGen/stl\_monitor.m**

We added two attributes to the class, initialized by

this.inout = '';

this.relabs = '';

and setmode (getmode) methods to write (read) both attributes at once.

The function get\_standard\_rob now switches between several possible semantics according to the mode:

switch this.inout

case {'in','out'}

switch this.relabs

case {'rel','abs'}

[rob, time] = STL\_Eval\_IO(this.Sys, phi, this.P0, this.P.traj{1}, this.inout, this.relabs, time);

otherwise

[rob, time] = STL\_Eval\_IO(this.Sys, phi, this.P0, this.P.traj{1}, this.inout, 'rel', time);

end

otherwise

[rob, time] = STL\_Eval(this.Sys, phi, this.P0,

this.P.traj{1}, time);

end

**./Core/Algos/FalsificationProblem.m**

We created a method set\_robust\_fn(this, inout, absrel) with arguments inout and absrel and sets the abbribute robust\_fn of the BreachProblem class to Eval\_IO(this, inout, absrel).

### Files created

**./@STL\_Formula/get\_in\_signal\_names.m  
./@STL\_Formula/get\_in\_signal\_names.m**

Getter methods that return the corresponding attributes of the formula class. Called internally by evaluation functions.

**./@STL\_Formula/is\_in\_signal.m  
./@STL\_Formula/is\_out\_signal.m**

Take as argument phi and signal\_name and scan the set of strings phi.in\_signal\_names (phi.out\_signal\_names) returning 0 or 1 depending whether signal\_name appears in the list.  
Not called internally by any other function.

**./@STL\_Formula/set\_in\_signal\_names.m  
./@STL\_Formula/set\_out\_signal\_names.m**

Take as argument a list of signal names and set the in\_signal\_name (out\_signal\_names) attributes of the formula object accordingly. Note that every subformula has such attributes and the procedure is recursive, setting the list of input (output) signals in each subformula after filtering them according to those that occur.

**./@STL\_Formula/STL\_EvalThom\_Gen.m**

This is a copy of STL\_EvalThom.m implementing a function that takes two additional arguments. The partition argument is a set of signal names, in which the robustness is computed. The relabs argument indicates how signal names not in the partition should be treated, and can take two string values: ‘rel’ for relative and ‘abs’ for absolute.

The specific treatment is implemented in the function GetValues. In the predicate case, after the robustness has been computed it is altered for the set of signals not in the partition through essentially scaling by 0 or by infinity:

switch relabs

case 'rel'

% Check whether the predicate talks only about the signals

% that are not in the partition. If yes, treat the predicate

% qualitatively

all\_outside\_partition = isempty(intersect(STL\_ExtractSignals(phi), partition));

if(all\_outside\_partition)

valarray = Inf\*(2\*(valarray>0)-1);

end

case 'abs'

% Checks whether the predicate has at least one signal that

% is outside the partition. If yes, treat the predicate

% qualitatively

one\_outside\_partition = isempty(setdiff(STL\_ExtractSignals(phi), partition));

if (one\_outside\_partition)

valarray = realmin('double')\*(2\*(valarray>0)-1);

end

end

**./@STL\_Formula/STL\_Eval\_Gen.m**

This is a copy of ./@STL\_Formula/STL\_EvalThom\_Gen.m that takes two optional arguments, partition and relabs. This function calls in turn ./@STL\_Formula/STL\_EvalThom\_Gen.m with the same arguments after checking their validity and fixing them as needed.

**./@STL\_Formula/STL\_Eval\_IO.m**

This is a copy of ./@STL\_Formula/STL\_EvalThom\_Gen.m that takes two optional arguments, inout and relabs. This function calls in turn ./@STL\_Formula/STL\_EvalThom\_Gen.m with the partition argument correctly set, and the relabs argument unchanged. Beforehand arguments are checked and fixed. The additional code is as follows:

partition = [];

if (strcmp(inout, 'in'))

partition = get\_in\_signal\_names(phi);

elseif (strcmp(inout, 'out'))

partition = get\_out\_signal\_names(phi);

end

switch nargin

case 6

[val, tau] = STL\_EvalThom\_Gen(Sys, phi, P, trajs, partition, relabs);

case 7

[val, tau] = STL\_EvalThom\_Gen(Sys, phi, P, trajs, partition, relabs, taus);

end

**./Plots/SplotSatIO.m**

This is a copy of SplotSat.m, but involving additional inout and relabs arguments, and calling STL\_Eval\_IO instead of STL\_Eval.