STLmc Manual

Geunyeol Yu, Jia Lee, and Kyungmin Bae

Pohang University of Science and Technology, Pohang, Korea

1 Introduction

The STLmc tool is a robust model checker for signal temporal logic (STL) properties of hybrid system. STLmc can perform STL model checking up to a robustness threshold $\epsilon > 0$ for a wide range of nonlinear hybrid systems with ordinary differential equations.

The tool provides an expressive input format to easily specify a wide range of hybrid automata, and a visualization command to give an intuitive description of counterexamples. The tool is implemented in around 9,500 lines of Python code. It support various SMT solvers, such as Z3 [3], Yices2 [4], dReal [5] as an underlying solver to solve the robust STL model checking problem.

The architecture of the tool is illustrated in Figure 1. The tool takes a hybrid automaton H, an STL formula φ, a variable point bound N, time bound τ, and a threshold ϵ. The given STL formula φ is first translated into the ϵ-strengthening of the negated formula $\neg(\varphi^{+\epsilon})$. The SMT encoding $\Psi^{k,\tau}_{H,\neg(\varphi^{+\epsilon})}$ for $1 \leq k \leq N$ is then built using the STL bounded model checking algorithm [2, 8]. The satisfiability of $\Psi^{k,\tau}_{H,\neg(\varphi^{+\epsilon})}$ can be checked directly using an SMT solver or using the two-step solving algorithm. If the tool find a counterexample, the tool provides visualization graphs of counterexample signals and robustness degrees.

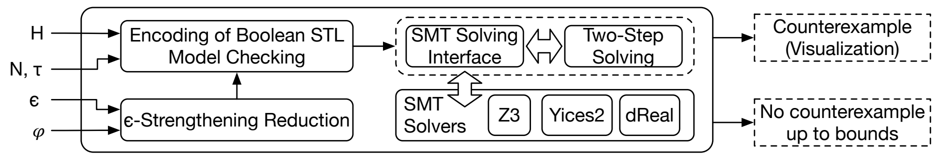
H
N, τ
ϵ
φ
Encoding of Boolean STL Model Checking
ϵ-Strengthening Reduction
SMT Solving Interface
Two-Step Solving
SMT Solvers
Z3
Yices2
dReal
Counterexample (Visualization)
No counterexample up to bounds

Fig. 1: The STLmc architecture

The tool is available at `https://stlmc.github.io/download` which explains how to download the tool.

The rest of the manual is organized as follows. Section 2 describes a simple running example used to explain our tool. Section 3 explains the input model language of STLmc. Section 4 explains how to set analysis parameters for robust STL model checking using configuration files and command-line options. Finally, Section 5 shows how to visualize counterexample signals and robustness degrees.

2 Running example

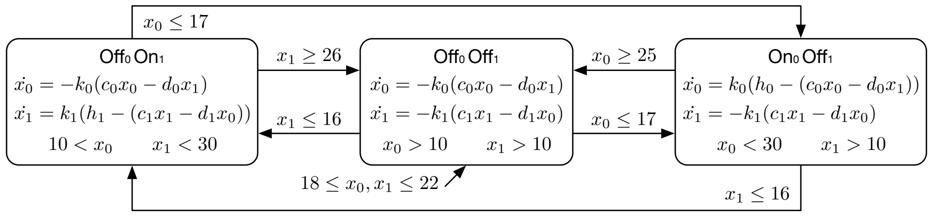
$x_0 \leq 17$
$\mathsf{Off}_0\,\mathsf{On}_1$
$\dot{x_0} = -k_0(c_0x_0 - d_0x_1)$
$\dot{x_1} = k_1(h_1 - (c_1x_1 - d_1x_0))$
$10 < x_0$ $x_1 < 30$
$x_1 \geq 26$
$x_1 \leq 16$
$\mathsf{Off}_0\,\mathsf{Off}_1$
$\dot{x_0} = -k_0(c_0x_0 - d_0x_1)$
$\dot{x_1} = -k_1(c_1x_1 - d_1x_0)$
$x_0 > 10$ $x_1 > 10$
$x_0 \geq 25$
$x_0 \leq 17$
$\mathsf{On}_0\,\mathsf{Off}_1$
$\dot{x_0} = k_0(h_0 - (c_0x_0 - d_0x_1))$
$\dot{x_1} = -k_1(c_1x_1 - d_1x_0)$
$x_0 < 30$ $x_1 > 10$
$18 \leq x_0, x_1 \leq 22$
$x_1 \leq 16$

Fig. 2: A hybrid automaton for the networked thermostats.

We consider networked thermostat controllers adapted from [1,6]. There are two rooms connected by an open door. The temperature x_i of each room $i \in \{0, 1\}$ is controlled by each thermostat, depending on the heater's mode $q_i \in \{\mathsf{On}, \mathsf{Off}\}$ and the other room's temperature. The continuous dynamics of x_i can be given as ODEs as follows:

$$\dot{x_i} = \begin{cases} K_i(h_i - (c_ix_i - d_ix_{1-i})) & (\mathsf{On}) \\ -K_i(c_ix_i - d_ix_{1-i}) & (\mathsf{Off}), \end{cases}$$

where K_i, h_i, c_i, d_i are constants depending on the size of the room, the heater's power, and the size of the door. Fig. 2 shows a hybrid automaton of our thermostat controllers. Initially, both heaters are off and the temperatures are between 18 and 22. The jumps between modes then define a control logic to keep the temperatures within a certain range using only one heater. We are interested in robust model checking of the following nontrivial STL properties:

(i) $\phi_1 = \Diamond_{[0,3]}(x_0 \geq 13 \; \mathbf{U}_{[0,\infty)} \; x_1 \leq 22)$,
(ii) $\phi_2 = \Box_{[2,4]}(x_0 - x_1 \geq 4 \rightarrow \Diamond_{[3,10]}(x_0 - x_1 \leq -3))$, and
(iii) $\phi_3 = \Box_{[0,10]}(x_0 > 23 \; \mathbf{R}_{[0,\infty)} \; (x_0 - x_1 \geq 4))$.

3 Modeling language

The STLmc tool supports modeling language to define hybrid system. The *model file* input format of STLmc, inspired by dReach [7], consists of five sections: variable declarations, mode definitions, initial conditions, state propositions, and STL properties. Mode definitions specifies flow, jump, and invariant conditions. Initial conditions specifies initial states of hybrid automata. STL formulas to be analyzed are defined in the STL properties section using proposition variables declared in state propositions. Fig. 3 shows the input model of the hybrid automaton described in the running example above.

```
const k0 = 0.015;       const k1 = 0.045;
const h0 = 100;         const h1 = 200;
const c0 = 0.98;        const c1 = 0.97;
const d0 = 0.01;        const d1 = 0.03;

int on0;                int on1;
[10, 35] x0;            [10, 35] x1;

{ mode: on0 = 0;         on1 = 1;
  inv:  10 < x0; x1 < 30;
  flow: d/dt[x0] = - k0 * (c0 * x0 - d0 * x1);
        d/dt[x1] = k1 * (h1 - (c1 * x1 - d1 * x0));
  jump: x0 <= 17 => (and (on0' = 1) (on1' = 0)
                         (x0' = x0) (x1' = x1));
        x1 >= 26 => (and (on1' = 0) (on0' = on0)
                         (x0' = x0) (x1' = x1));
}
{ mode: on0 = 1;         on1 = 0;
  inv:  x0 < 30; x1 > 10;
  flow: d/dt[x0] = k0 * (h0 - (c0 * x0 - d0 * x1));
        d/dt[x1] = - k1 * (c1 * x1 - d1 * x0);
  jump: x1 <= 16 => (and (on0' = 0) (on1' = 1)
                         (x0' = x0) (x1' = x1));
```

```
   x0 >= 25 => (and (on0' = 0) (on1' = on1)
                    (x0' = x0) (x1' = x1));
}
{ mode: on0 = 0;         on1 = 0;
  inv:  x0 > 10; x1 > 10;
  flow: d/dt[x0] = - k0 * (c0 * x0 - d0 * x1);
        d/dt[x1] = - k1 * (c1 * x1 - d1 * x0);
  jump:
    x0 <= 17 => (and (on0' = 1) (on1' = on1)
                     (x0' = x0) (x1' = x1));
    x1 <= 16 => (and (on1' = 1) (on0' = on0)
                     (x0' = x0) (x1' = x1));
}

init: on0 = 0;  18 <= x0;  x0 <= 22;
      on1 = 0;  18 <= x1;  x1 <= 22;

proposition:
  [p1]: x0 - x1 >= 4;    [p2]: x0 - x1 <= -3;

goal:
  [f1]: <>[0, 3](x0 >= 13 U[0, inf) x1 <= 22);
  [f2]: [][2, 4](p1 -> <>[3, 10] p2);
  [f3]: [][0, 10](x0 > 23 R[0, inf) p1);
```

Fig. 3: An input model example

3.1 Variable declarations

STLmc uses mode and continuous variables to specify discrete and continuous states of a hybrid automaton. *Mode variables* define a set of discrete modes Q, and *continuous variables* define a set of real-valued variables X.[1] We can also declare named constants whose values do not change.

Mode variables are declared with one of three types: `bool` variables with `true` and `false` values, `int` variables with integer values, and `real` variables with real values. E.g., the following declares a `bool` variable `b` and an `int` variable `i`:

```
bool b;    int i;
```

Continuous variables are declared with domain intervals. Domains can be any intervals of real numbers, including open, closed, and half-open intervals, E.g., the following declares two continuous variable `x` and `y`:

```
[0, 50] x;    (-1.1, 1) y;
```

Finally, constants are introduced with the `const` keyword. Constants can have Boolean, integer, or rational values. For example, the following declares a constant `k1` with a rational value 0.015:

```
const k1 = 0.015;
```

[1] An assignment to mode and continuous variables corresponds to a single state.

3.2 Mode definitions

In STLmc, *mode blocks* define mode, jump, invariant, and flow conditions for a group of modes in a hybrid automaton. Each mode block consists of four components:

```
{
    mode: ...
    inv:  ...
    flow: ...
    jump: ...
}
```

A mode block represents a set of discrete modes of a hybrid automaton that satisfies conditions in the `mode` component. All states of the mode block satisfy conditions in the `inv` component. The continuous variables of the states in the mode block evolve according to conditions in the `flow` component. A discrete transition can be possible if a state satisfies conditions in the `jump` component.

A `mode` component contains a semicolon-separated set of Boolean conditions over mode variables. The conjunction of these conditions represents a set of discrete modes.[2] E.g., the following represents a set of two discrete modes $\{(\mathit{true}, 1), (\mathit{true}, 2)\}$, provided there are two mode variables `b` (of `bool` type) and `i` (of `int` type):

```
b = true;    i > 0;    i < 3;
```

An `inv` component contains a semicolon-separated set of Boolean formulas over continuous variables. The conjunction of these conditions represents the invariant condition for each mode in the mode block. For example, the following declares an invariant condition for two continuous variables `x` and `y`:

```
x < 30;    y > - 0.5;
```

A `flow` component contains either a system of ordinary differential equations (ODEs) or a closed-form solution of ODEs. In STLmc, a system of ODEs over continuous variables $x_1, \ldots, x_n$ is written as a semicolon-separated equation of the following form, where e_i denotes an expression over $x_1, \ldots, x_n$:

```
d/dt[x_1] = e_1(x_1, ..., x_n) ;
...
d/dt[x_n] = e_n(x_1, ..., x_n) ;
```

For example, continuous dynamics of x and y that are specified as the ODEs: $\dot{x} = \sin(y)$ and $\dot{y} = y^2$ are defined as follows:

```
d/dt[x] = sin(y);  d/dt[y] = y ** 2;
```

[2] We assume that the mode conditions of different mode blocks cannot be satisfied at the same time, which can be automatically detected by our tool.

A closed-form solution of ODEs is written as a set of continuous functions, parameterized by a time variable `t` and the initial values $x_1(0), ..., x_n(0)$ for each discrete mode:

```
x_1(t) = e_1(t, x_1(0),...,x_n(0)) ;
...
x_n(t) = e_n(t, x_1(0),...,x_n(0)) ;
```

For example, let continuous dynamics of x and y be specified in closed-form solutions: $x(t) = t^2 + 3 * t + x(0)$ and $y(t) = t + y(0)$. These are defined as follows:

```
x(t) = t ** 2 + 3 * t + x(0);  y(t) = t + y(0);
```

A `jump` component contains a set of jump conditions *guard* `=>` *reset*, where *guard* and *reset* are Boolean conditions over mode and continuous variables. We use "primed" variables to denote variables after jumps have occurred. E.g., the following defines a jump with four variables `b`, `i`, `x`, and `y` (declared above):

```
(and (i = 1) (x > 10)) => (and (b' = false) (i' = 3) (x' = x) (y' = 0));
```

3.3 Initial conditions

In the `init` section, initial conditions are declared as a set of Boolean formulas over mode and continuous variables. The conjunction of these conditions represents a set of initial discrete modes. E.g., the following defines a set of initial discrete modes over variables `b`, `i`, `x`, and `y`:

```
init: not b; i = 0; 19.9 <= x; y = 0;
```

3.4 STL Properties

In the `goal` section, STL properties can be declared with or without labels. When an STL property is declared with a label, the STLmc tool can execute this specific formula by specifying its label.

For example, the following declares two STL formulas in the running example. The first STL formula is declared without a label and the second STL formula is declared with label `f2`:

```
<>[0, 3](x0 >= 13 U[0, inf) x1 <= 22);
[f2]: [][2, 4]((x0 - x1 >= 4) -> <>[3, 10] (x0 - x1 <= -3));
```

To make it easy to write repeated propositions, "named" state propositions can be declared in the `proposition` section. For example, the second STL formula can be rewritten using two propositions `p1` and `p2` as follows:

```
proposition:
[p1]: x0 - x1 >= 4;
[p2]: x0 - x1 <= -3;

goal:
...
[f2]: [][2, 4](p1 -> <>[3, 10] p2);
[f3]: [][0, 10](x0 > 23 R[0, inf) p1);
```

4 Configuration

In this section, we explain the STLmc tool parameters in more details and show how to set the tool parameters using configuration files and command-line. STLmc performs robust STL model checking, using the analysis parameters explained in Table 1. For example, consider the input model in Fig. 3. The following command verifies the formula `f1` up to bounds $N = 5$ and $\tau = 30$ with respect to robustness threshold $\epsilon = 1$ in 1006 seconds using dReal:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 -threshold 1 \
         -goal f1 -solver dreal -two-step -parallel -time-horizon 7
goal : (<>[0.0,3.0] ((x0 >= 13) U[0.0,inf) (x1 <= 22)))
result : True up to bound 5 (time bound: 30)
running time 1005.49810 seconds
```

4.1 The STLmc Tool Parameters

We explain the STLmc tool parameters in more details. There are two types of tool parameters: one for the STLmc algorithms and the other for the underlying SMT solvers (i.e., `z3`/`yices`/`dreal`). For example, the `bound` parameter sets a discrete bound used for the STLmc model checking algorithm, while the `logic` parameter sets a background logic for SMT solvers (`z3`/`yices`). In total, STLmc supports 14 parameters. The summary of all available parameters are in Table 1.

The basic parameters for the analysis are `bound` and `time-bound`. The `bound` parameter limits the number of mode changes and the number of *variable points*—at which the truth value of some STL subformula changes—in the trajectories of input hybrid automaton. The `time-bound` parameter sets the maximum time bound of the trajectories.

For labeled STL formulas, we can analyze the labeled goals by specifying the labels to the `goal` parameter. For example, the following command performs analysis on the `f2` and `f3` formulas:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 \
         -goal f2,f3 -solver dreal
```

Name	Explanation	Default
`bound`	a discrete bound $N \in \mathbb{N}$	-
`time-bound`	a time bound $\tau \in \mathbb{Q}^+$	-
`threshold`	a robustness threshold $\epsilon \in \mathbb{Q}^+$	0.01
`solver`	an SMT solver to be used (`z3`/`yices`/`dreal`)	`auto`
`time-horizon`	a mode duration bound	τ
`goal`	a list of STL goals to be analyzed	all
`two-step`	enable the two-step solving algorithm	disabled
`parallel`	parallelize the two-step solving algorithm	disabled
`visualize`	generate extra visualization data	disabled
`verbose`	print more information of the execution	disabled
`logic`	set a background logic (`z3`/`yices`)	`QF_NRA`
`precision`	a δ-precision parameter (`dreal`)	0.001
`ode-step`	set a time step of numerical calculation (`dreal`)	0.001
`ode-order`	set a maximum talyer expansion order (`dreal`)	20

Table 1: The STLmc tool parameters.

We can use the `threshold` and `time-horizon` parameters for fine analysis. For example, we can increase or decrease `threshold` to change the robust threshold of the robustness STL model checking. The `time-horizon` parameter can be used to limit the maximum time duration of a single mode of the trajectories. E.g., the following command performs analysis with a threshold 2 and a time-horizon 7 for the goal `f3`:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 -threshold 2 \
        -goal f3 -solver dreal -time-horizon 7
```

To use the two-step solving algorithm, we can enable the `two-step` parameter. We can also parallelize the two-step solving algorithm by enabling the `parallel` parameter. For example, the following command uses the parallelized two-step algorithm for the analysis:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 -threshold 2 \
        -goal f3 -solver dreal -time-horizon 7 \
        -two-step -parallel
```

When the `visualize` parameter is enabled, STLmc generates extra data for visualization during the analysis. STLmc generates visualization data only if it finds counterexample. For example, the following command generates a visualization data file:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 -threshold 1 \
        -goal f3 -solver dreal -time-horizon 7 \
        -two-step -parallel -visualize
```

There are four solver-specific parameters: `logic`, `precision`, `ode-step`, and `ode-order`. The `logic` parameter sets the background logic for the `z3` and `yices`

```
# STLmc configuration
common {
    # mandatory arguments
    # bound =
    # time-bound =

    threshold = 0.01            # positive rational number
    solver = "auto"             # dreal, yices, z3
    time-horizon = "time-bound"
    goal = "all"                # STL formula labels
    two-step = "false"          # on
    parallel = "false"          # on
    visualize = "false"         # on
    verbose = "false"           # print verbose messages
}

# underlying solver
z3     {
    logic = "QF_NRA" # QF_LRA

}
yices  {
    logic = "QF_NRA" # QF_LRA
}
dreal {
    precision = 0.001            # positive rational number
    ode-order = 5                # natural number
    ode-step = 0.001             # positive rational number
    executable-path = "../dreal" # dreal executable path
}
```

Fig. 4: The default configuration file "default.cfg" of STLmc.

solvers. The other parameters are for the dreal solver. The precision parameter sets the δ precision for the δ-complete decision procedures. The ode-order and ode-step parameters set the maximum order of taylor expansion for ordinary differential equations and the time step of the dreal's numerical calculation, respectively.

4.2 Setting the Tool Parameters with Configuration Files

In this section, we explain the STLmc configuration files and show how to set the tool parameters using the configuration files. A STLmc configuration file is a file having the set of predefined tool parameters. STLmc can take multiple STLmc configuration files to set its tool parameters. A STLmc configuration file consists of four sections: common, z3, yices, and dreal. The common section defines the common tool parameters for the STLmc algorithms. The other three sections (i.e., `z3`, `yices`, and `dreal`) define solver-specific parameters. In default, we provide one configuration file `default.cfg` for STLmc. STLmc first uses this configuration file to set its parameters. Before taking the tool parameters via command-line, STLmc first uses this configuration file to set its parameters. Figure 4 shows the default configuration file of STLmc.

The basic STLmc tool parameters can be defined in a `common` section. For example, we can set a time horizon to 7.8, underlying SMT solver to Yices2, and enabling the two-step alogrithm as follows:

```
time-horizon = 7.8
solver = "yices"
two-step = "true"
```

When the `solver` parameter is set to `"auto"`, STLmc chooses an appropriate underlying solver, by analyzing the flow conditions of a hybrid automaton.

The solver-specific parameters, such as `logic`, can be defined in each solver section (i.e., `z3`, `yices`, and `dreal` sections). For example, consider that we set the solver parameter to `yices` in the `common` section. Then, we can set the background logic of the `yices` solver to QF_LRA by setting the `logic` parameter of the `yices` section to `QF_LRA` as follows:

```
logic = "QF_LRA"
```

There can be many analysis scenarios for a given input model. In this case, we can set the common parameters used throughout all the scenarios in the same configuration file, while setting the parameters specific to each scenario in another configuration file. For example, consider the input model in Fig. 3. We can set the following as the common analysis parameters:

```
time-bound = 30
bound = 5
solver = "dreal"
verbose = "true"
two-step = "true"
parallel = "true"
```

We can set different parameters for each goal formula. For the goal `f1`, we can set the following parameters:

```
threshold = 1.0
time-horizon = 7
goal = "f1"
```

For the goal `f2`, we can set the following parameters:

```
threshold = 2.0
time-horizon = 30
goal = "f2"
```

STLmc provides command line arguments to take configuration files (i.e., `default-cfg`, `model-cfg` and `model-specific-cfg`). The `default-cfg` parameter is used to take the `default.cfg` file which is explained in Fig. 4. The `model-cfg` and `model-specific-cfg` parameters are used to take the common configuration file and the configuration file specific to each analysis scenario, respectively.

There can be the same parameters set differently in multiple configuration files. To select one of them, STLmc reads the configuration files and takes the last one. STLmc reads the command line arguments in the following order: reading the `default-cfg` parameter first, the `model-cfg` parameter second, and the `model-specific-cfg` parameter last.

4.3 Setting the Tool Parameters with Command-line

The STLmc tool provides a command-line interface. The tool takes a model file, configuration files, and all parameters in the `default.cfg`. The model file argument is required and other configuration arguments are optional.

```
$./stlmc [path to model file] \
         -defaul-cfg [path to default config file]\
         -model-cfg [path to model config file] \
         -model-specific-cfg [path to model specific config file] \
         -bound [int] ...
```

If the `default-cfg` parameter is not given, the tool uses the predefined default configuration (i.e., `default.cfg`). If the `model-cfg` parameter is not given, the tool searches for the configuration file, named ⟨model_file⟩.cfg, in the same directory of the input model file. STLmc reads the parameters given by th command-line last For example, if some parameters set in the configuration files are also given through the command-line, STLmc selects the values of the parameters set with the command-line.

Example 1. Consider the input model in Fig. 3 (`therm.model`), we can define the model configuration file `therm.cfg` as follows:

```
common { time-bound = 30
         bound = 10
         solver = "dreal"
         verbose = "true"
         two-step = "true"
         parallel = "true" }
```

For the goal `f2`, we can define the configuration file `therm-f2.cfg`:

```
common { threshold = 2.0
         time-horizon = 30
         goal = "f2" }
```

The following command found the counterexample of the goal `f2` at bound 5 with respect to robustness threshold $\epsilon = 2$ in 8 seconds using dReal:

```
$./stlmc ./therm.model -model-cfg therm.cfg  \
        -model-specific-cfg therm-f2.cfg -bound 5
goal: []_[2.0,4.0] (p1 -> (<>_[3.0,10.0] p2))
result: counterexample found (bound 2)
running time: 7.46335 seconds
```

5 Visualization

The STLmc tool provides a script to visualize counterexamples for robust STL model checking. The visualization script takes a counterexample file and a visualization configuration file and returns graphs representing counterexample trajectories and robustness degrees.

```
./stlmc-vis [path to counterexample file] \
          -cfg [path to configuration file]
```

The counterexample file is specified in a `.counterexample` file. The file is created when there is a counterexample and the `visualize` option is set. The configuration file contains following things: (1) continuous variables, (2) STL subformula labels, (3) output format, and (4) group of variables. The file contains variable information of continuous variables and STL subformulas. The STLmc tool supports "html" and "pdf" as output formats. The states of variables in a group are plotted on one graph. Only variables of the same type can be grouped together. [3] Variables not assigned to a group are grouped together according to their type. Figure 5 shows a visualization configuration file of our thermostat controllers.

```
{   # continuous variables: x0, x1
    # STL subformula labels:
    # f2 --> [][2, 4]((x0 - x1 >= 4) -> <>[3,10](x0 - x1 <= -3))
    # f2_1 --> (x0 - x1 >= 4) -> <>[3,10](x0 - x1 <= -3)
    # f2_2 --> not (x0 - x1 >= 4)
    # f2_3 --> <>[3,10](x0 - x1 <= -3)
    # p_1 --> x0 - x1 >= 4
    # p_2 --> x0 - x1 <= -4

    # output = pdf # html
    group { (x0, x1), (f2, f2_1) (f2_2, f2_3) (p_1, p_2) }
```

Fig. 5: A visualization configuration example

[3] For example, if there is a group $(x0, f2)$, the tool will throw an error because the types of $x0$ and $f2$ are real and bool, respectively.

Example 2. Consider the conunterexample file for `f2` in Example 1 (`therm.model`) and the visualization configuration file in Fig. 5 (`therm_vis.cfg`). The following command generate a PDF image. in Fig. 6.

```
./stlmc-vis therm.counterexample -cfg therm_vis.cfg
```

The robustness degree of `f2` is less than ϵ at time 0, since the robustness degree of $\texttt{f2}_1$ goes below ϵ in the interval $[2, 4]$, which is because both the degrees of $\texttt{f2}_2$ and $\texttt{f2}_3$ are less than ϵ in $[2, 4]$. The robustness degree of $\texttt{f2}_3$ is less than ϵ in $[2, 4]$, since the robustness degree of $\texttt{p}_2$ is less than ϵ in $[5, 14] = [2, 4] + [3, 10]$.

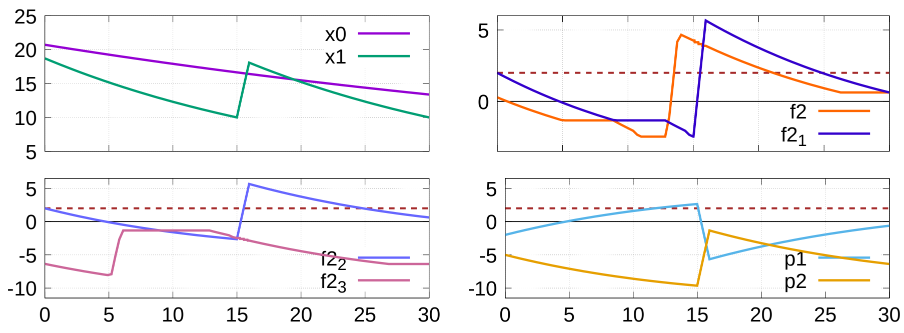

Fig. 6: Visualization of a counterexample (horizontal dotted lines denote $\epsilon = 2$).

References

1. Bae, K., Gao, S.: Modular smt-based analysis of nonlinear hybrid systems. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 180–187. FMCAD '17, FMCAD, Austin, TX (2017), `http://dl.acm.org/citation.cfm?id=3168451.3168490`
2. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3, POPL**(51), 1–30 (2019)
3. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
4. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer (2014)
5. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. LNCS, vol. 7898, pp. 208–214. Springer (2013)
6. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)
7. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: δ-reachability analysis for hybrid systems. In: Proc. TACAS. LNCS, vol. 7898, pp. 200–205. Springer (2015)
8. Lee, J., Yu, G., Bae, K.: Efficient smt-based model checking for signal temporal logic. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 343–354 (2021). https://doi.org/10.1109/ASE51524.2021.9678719