HySIA Manual (version 0.1.1)

Daisuke Ishii dsksh@acm.org

March 3, 2017

HySIA is a reliable simulator and verifier for hybrid systems. HySIA supports nonlinear *hybrid automata* (HAs) whose ODEs, guards, and reset functions are specified with nonlinear expressions. It assumes a deterministic class of HA; a transition to another location happens whenever a guard condition holds. Main functionalities of HySIA are as follows:

Simulation. HySIA simulates an HA based on interval analysis; it computes an overapproximation of a bounded trajectory that is composed of *boxes* (i.e., closed interval vectors) and *parallelotopes* (linear transformation of boxes). Intensive use of interval analysis techniques distinguishes HySIA from other reachability analysis tools. First, the simulation process carefully reduces *wrapping effect* that can expand an enclosure interval. As a result, HySIA is able to simulate HA for more number of steps than other overapproximation-based tools; e.g., it can simulate a periodic bouncing ball for more than a thousand steps. Second, HySIA assures the soundness of each interval computation, so that the resulting overapproximation is verified to contain a theoretical trajectory. See Reference [2] for details of the underlying method.

Monitoring. HySIA takes a temporal property as an input and monitors whether a simulated trajectory satisfies the property; otherwise, HySIA computes (an interval overapproximation of) a robustness signal for the property. The verification process is based on a monitoring procedure of the *signal temporal logic* (STL) formulas [5], which is extended to handle overapproximation of trajectories. See Reference [3] for more details.

1 Short Tutorial

HySIA provides a specification language for hybrid automata and temporal logic properties. An HA is simulated and analyzed about a property using the `hysia` command. This section exemplifies the modeling, simulation, and verification of an HA using HySIA.

1.1 Modeling Hybrid Automata

Below is an example specification that describes a simple bouncing ball model (bb.ha):

```
1  (* first part *)
2  let   g = 1
3  let   c = 0.9
4  let   pertb = [-1e-5, 1e-5]
5
6  (* second part *)
7  var   y, vy
8
9  init  Loc, 1+pertb, 0+pertb
10
11 at Loc wait vy, -g
12       once (y, -vy) goto Loc then y, -c*vy
13 end
14
15 (* third part *)
16 param order = 20
17 param t_max = 100
18 param dump_interval = 0.1
```

The specification consists of three parts. The first part (Lines 1–4) defines three constants g, c, and pertb to be used in the second part. The values of constants can be either real values or intervals.

The second part (Lines 6–13) describes an HA. Line 7 declared the state variables y and vy of the HA, which are evaluated over timeline. Line 9 describes the initial state as a comma-separated list of a location and values for each state variable. Lines 11–13 defines a location named Loc. After the keyword wait, the derivatives of the state variable are specified as $\frac{d}{dt}$y = vy and $\frac{d}{dt}$vy = −g. At Line 12, an inter-location transition is specified. The tuple after once tells that the guard condition for this transition is y = 0 ∧ vy > 0. (Only the left-hand side of the implicit form is described.) Whenever the guard is satisfied, an execution will transit to the same location Loc with a reset of the state variables as y := y and vy := −c vy.

The last part (Lines 15–18) configures some parameters of the simulator implementation, i.e., the order order of Taylor coefficient expansion, the time horizon t_max of a continuous state evolution, and dump_interval that bounds the step size when computing a dumped data.

In general, an HA consists of multiple locations. The bouncing ball system

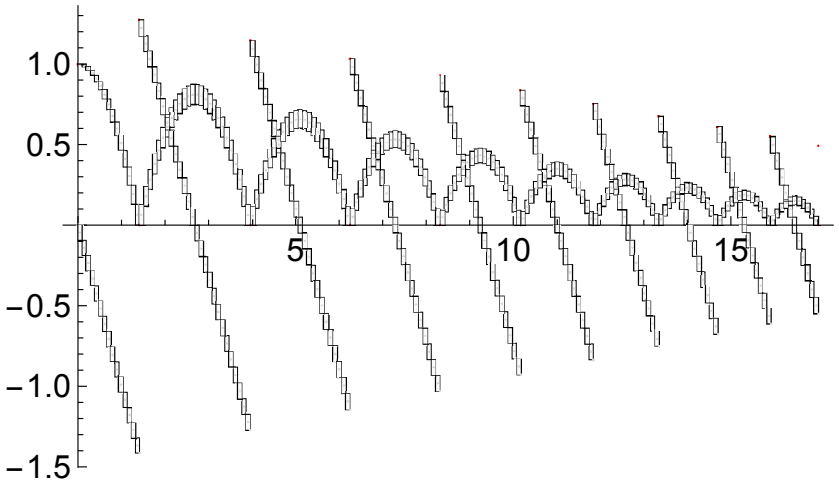

Figure 1: Dumped trajectory enclosure of the bouncing ball example

can be modeled with two locations by modelling the second part as follows:

```
init  Fall, 1+pertb, -0+pertb

at Fall wait vy, -g
      once (y, vy) goto Rise then y, -c*vy
end

at Rise wait vy, -g
      once (vy, -y) goto Fall then y, vy
end
```

1.2 Simulation of a Model

Once a specification is prepared, a user can simulate the model for 10 transitions with the following command:

```
$ hysia bb.ha -n 10 -dump
```

The option `-dump` let HySIA to output the result of the simulation to the file `pped.dat` in a JSON format. The output data consists of a set of *boxes* (interval vectors) that encloses the trajectories of the HA. (In this example, the interval value `pertb` allows perturbation of trajectories; these trajectories are enclosed within the boxes.) The result can be visualized as shown in Figure 1.

Simulation and verification of HySIA are all computed with validated interval analysis. Thus, a resulting interval enclosure of a system's state expands as longer

the simulation length and more the *wrapping effect* occurs. However, thanks to the underlying *parallelotope*-based simulation method for wrapping effect reduction, HySIA is able to simulate a large number of jumps for various HA. Indeed, for the bouncing ball model, when we modify the parameter values as `c = 1` and `pertb = 0`, HySIA can simulate for more than a thousand steps.

As we have seen so far, HySIA allows models to involve some uncertainties derived by interval values. HySIA computes (the overapproximation of) the reachable region with respect to the uncertainties. However, here we may encounter a drawback of HySIA; a simulation is not always successful but may result in an error. When we modify the value of `pertb` to a slightly large interval `[-1e-4, 1e-4]` and run a simulation, `hysia` will output as follows:

```
$ hysia bb.ha -n 10 -dump
step 0 (0.000000 < inf) at Loc
step 1 (1.414043 < inf) at Loc
step 2 (3.959155 < inf) at Loc
step 3 (6.249714 < inf) at Loc
step 4 (8.310768 < inf) at Loc
step 5 (10.163552 < inf) at Loc
step 6 (11.809965 < inf) at Loc
libc++abi.dylib: terminating with uncaught exception of type std::
runtime_error: zero in the derivative
Abort trap: 6
```

The simulation fails after the sixth jump. The output implies that HySIA fails in the detection of a discrete change because a state enclosure becomes too large. In fact, in each detection, HySIA checks whether the orientation between the trajectory and the guard is regular enough so that it is sure that each trajectory within an enclosure satisfies the guard.

1.3 Verification of STL Properties

HySIA provides a function for verifying properties described in the *signal temporal logic* (STL). For example, we can add an STL property

```
1 prop G[0,10] F[0,1] y-0.3
```

in the above mentioned model. (It should be added in between the second and third parts.) The property intuitively claims that, for the duration of 10 time units, the height of the ball (`y`) goes beyond 0.3 (`y-0.3` is interpreted as $y - 0.3 > 0$) within every duration of 1 time unit.

The verification can be done with the following command:

```
$ hysia bb.ha -a
```

The `hysia` command calculates the *necessary* simulation length for the verification (in this case, 11 time units), performs a simulation, and evaluates the satisfiability of this property. Accordingly, the verification succeeds with the following output:

```
$ hysia bb.ha -a
step 0 (0.000000 < 11.000000) at Loc
step 1 (1.414196 < 11.000000) at Loc
step 2 (3.959734 < 11.000000) at Loc
step 3 (6.250722 < 11.000000) at Loc
step 4 (8.312608 < 11.000000) at Loc
step 5 (10.168295 < 11.000000) at Loc
true, 0.036239
```

2 Getting the Tool

The HySIA tool can be obtained and used in three ways.

2.1 Web Demonstration

A web demonstration site is available at:

`http://bit.ly/hysia`

Through a web browser, you can access a GUI, load the basic examples, modify the HA/STL specification, simulate, and verify the specification.

2.2 Docker Image

For those who are familier with *Docker*,[1] Docker containers are available at:

`https://hub.docker.com/r/dsksh/`

The image `dsksh/hysia-web` contains the server program for the web demonstration. A container can be launched by:

```
$ docker run -p 8080:80 dsksh/hysia-web
```

Then, the server will be available at:

`http://localhost:8080`

[1] `https://www.docker.com/`

2.3 Source Distribution

The HySIA source code is distributed via GitHub:

`https://github.com/dsksh/hysia`.

Requirements

In addition to a standard UNIX-like environment, the following softwares are required to compile HySIA:

- *C/C++ compiler*. We have compiled with both `gcc` (versions 4.7.4–4.8.4) and `clang` (Apple LLVM version 8).
- *OCaml compiler*. We have tested with versions 3.12–4.02.
- *CAPD library*.[2] HySIA is built on an old release of the CAPD-DynSys 3.0 library, which was distributed around 2014, and is not available on the official site. A (slightly modified) source package is available at:

`https://www.dropbox.com/s/3uf7t2nsizfebno/capdDynSys-201406.zip`

- *Boost library*.[3] HySIA uses `shared_ptr`.
- *OUnit*.[4] Optional for test cases compilation.
- *Eliom*.[5] Optional for building the web application.

Build

In the root directory of the source code, HySIA can be compiled with:

```
$ ./configure; make
```

When compilation succeeds, the program file

`src_ocaml/hss.opt`

is generated.

[2] http://capd.ii.uj.edu.pl/
[3] http://www.boost.org/
[4] http://ounit.forge.ocamlcore.org/
[5] http://ocsigen.org/eliom/

3 Examples

3.1 Simple Rotation System

TBD.

3.2 Bouncing Planet

TBD.

4 Reference Manual

4.1 Command-Line Tool

The basic syntax for the command-line execution is as follows:

⟨*command*⟩ ::= '`hysia`' ⟨*options*⟩ ⟨*filename*⟩

The `hysia` command accepts the following options:

`-h`, `-help`, or `-help` Displays a summary of the options accepted by the command.

`-n` Specifies the number steps to simulate (default is ∞).

`-t` Specifies the max simulation time (default is ∞).

`-a` Decides the simulation length automatically from the STL property.

`-g` Sets the debug flag.

`-dump` Activates dumping plot to the file "pped.dat."

`-cm_thres` Sets the threshold for character matrix selection.

The option `-cm_thres` specifies the character matrix B to be used in the parallelotope method (see the corresponding publication for the detail). It is selected as follows:

- -1: $B := (\mathrm{mid}\boldsymbol{J})A$.
- 0: $B := I$ (i.e., identity matrix).
- 1: $B :=$ orthogonalize$((\mathrm{mid}\boldsymbol{J})A)$.
- $n > 1$ (default):

$$B := \begin{cases} (\mathrm{mid}\boldsymbol{J})A & \text{if } \kappa((\mathrm{mid}\boldsymbol{J})A) < n \\ \text{orthogonalize}((\mathrm{mid}\boldsymbol{J})A) & \text{otherwise.} \end{cases}$$

4.2 Solving Parameters

order Order of Taylor expansion.

t_max Max time horizon assumed in the simulation of each step.

h_min Min time CAPD integration can take.

epsilon Specifies the precision of the event detection.

dump_interval Sets the precision of the dumped flowpipe data.

delta Parameter for the box inflation process.

tau Parameter for the box inflation process.

cm_thres Parameter for the character matrix selection.

4.3 Specification Language

This section describes the grammar of the specification language of HySIA. A specification consists of the definition of a hybrid automaton, an STL formula, and solving parameter configurations.

4.3.1 Lexical conventions

The lexical class of digits, letters, and identifiers is the following:

⟨*digit*⟩ ::= [‘0’–‘9’]

⟨*letter*⟩ ::= [‘a’–‘z’ ‘A’–‘Z’]

⟨*id*⟩ ::= ⟨*letter*⟩ (⟨*digit*⟩ | ⟨*letter*⟩ | ‘_’)*

The syntax of various numeral expressions is as follows:

⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*float*⟩ ::= ⟨*digit*⟩+ (‘.’ ⟨*digit*⟩*)? ((‘e’|‘E’) (‘+’|‘-’)? ⟨*digit*⟩+)?

⟨*float-pn*⟩ ::= ⟨*float*⟩ | ‘-’ ⟨*float*⟩

⟨*interval*⟩ ::= ⟨*interval-noun*⟩ | ⟨*float-pn*⟩

⟨*interval-noun*⟩ ::= ‘(’ ⟨*float-pn*⟩ ‘,’ ⟨*float-pn*⟩ ‘)’

⟨*interval-list*⟩ ::= ⟨*interval*⟩ ⟨*interval-list-rest*⟩
| '(' ⟨*interval*⟩ ⟨*interval-list-rest*⟩ ')'

⟨*interval-list-rest*⟩ ::= ',' ⟨*interval*⟩ ⟨*interval-list-rest*⟩ | ⟨*empty*⟩

Comments. Comments are either enclosed between (* and *) (can be nested) or prefixed with #.

4.3.2 Toplevel syntax

The syntax for the toplevel of specifications is the following:

⟨*specification*⟩ ::= ⟨*statement*⟩+ ⟨*property*⟩? ⟨*solver-param*⟩*

⟨*statement*⟩ ::= 'let' ⟨*id*⟩ '=' ⟨*interval*⟩
| 'let' ⟨*id*⟩ '=' 'R' ⟨*float*⟩
| 'var' ⟨*var-list*⟩
| 'init' ⟨*expr-list*⟩
| 'at' ⟨*id*⟩ ⟨*flow*⟩ ⟨*invariant*⟩? ⟨*edge*⟩* 'end'

⟨*property*⟩ ::= 'prop' ⟨*stl-formula*⟩

⟨*solver-param*⟩ ::= 'param' ⟨*id*⟩ '=' ⟨*float-pn*⟩

⟨*flow*⟩ ::= 'wait' ⟨*expr-list*⟩

⟨*invariant*⟩ ::= 'inv' ⟨*expr-list*⟩

⟨*edge*⟩ ::= ('when' | 'once') '(' ⟨*expr*⟩ ',' ⟨*expr-list*⟩ ')' 'goto' ⟨*id*⟩
'then' ⟨*expr-list*⟩

4.3.3 Expressions

The operators and function application in expressions have the priorities and associativities as shown in the table below (from lowest to greatest priority):

construct	associativity
'+', '-'	left
'*', '/'	left
function application	left
'^', '-' (unary)	—

The syntax for expressions is the following:

⟨*expr*⟩ ::= ⟨*expr*⟩ '+' ⟨*expr*⟩ | ⟨*expr*⟩ '-' ⟨*expr*⟩
| ⟨*expr*⟩ '*' ⟨*expr*⟩ | ⟨*expr*⟩ '/' ⟨*expr*⟩
| '-' ⟨*expr*⟩ | ⟨*expr*⟩ '^' ⟨*integer*⟩
| ⟨*function*⟩ ⟨*expr*⟩
| ⟨*id*⟩ | ⟨*interval*⟩
| '(' ⟨*expr*⟩ ')'

⟨*function*⟩ ::= 'sqrt' | 'exp' | 'log' | 'sin' | 'cos' | 'atan' | 'asin' | 'acos'

⟨*expr-list*⟩ ::= ⟨*expr*⟩ ⟨*expr-list-rest*⟩ | '(' ⟨*expr*⟩ ⟨*expr-list-rest*⟩ ')'

⟨*expr-list-rest*⟩ ::= ',' ⟨*expr*⟩ ⟨*expr-list-rest*⟩ | ⟨*empty*⟩

4.3.4 STL formulae

The operators in formulae have the following priorities and associativities:

construct	associativity
'->'	right
'|'	right
'&'	right
'U [·,·]'	right
'G [·,·]', 'F [·,·]'	—
'!'	—

The syntax for formulae is the following:

⟨*stl-formula*⟩ ::= 'true' | 'false'
| ⟨*id*⟩ | ⟨*expr*⟩
| '!' ⟨*stl-formula*⟩
| ⟨*stl-formula*⟩ ('&'|'|'|'->') ⟨*stl-formula*⟩
| ('F'|'G') ⟨*noun-interval*⟩ ⟨*stl-formula*⟩
| ⟨*stl-formula*⟩ 'U' ⟨*noun-interval*⟩ ⟨*stl-formula*⟩
| '(' ⟨*stl-formula*⟩ ')'

References

[1] A. Donzé, T. Ferrère, O. Maler: Efficient Robust Monitoring for STL, Proc. of CAV, pp. 264–279, *LNCS* 8044, 2013.

[2] A. Goldsztejn, D. Ishii: A Parallelotope Method for Hybrid System Simulation, Reliable Computing, 23:163–185, 2016.

[3] D. Ishii, N. Yonezaki, A. Goldsztejn: Monitoring Temporal Properties using Interval Analysis. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E99-A, 2016.

[4] D. Ishii, N. Yonezaki, A. Goldsztejn: Monitoring Bounded LTL Properties Using Interval Analysis. Proc. of 8th International Workshop on Numerical Software Verification (NSV), *ENTCS* 317, pp. 85–100, 2015.

[5] O. Maler and D. Nickovic: Monitoring Temporal Properties of Continuous Signals, Proc. of FORMATS, pp. 152–166, *LNCS* 3253, 2004.