HSolver
User Manual

Stefan Ratschan and Tomáš Dzetkulič and Zhikun She

June 17, 2013

Contents

Chapter 1

Introduction

HSolver is a software package for the formal verification of safety properties of continuous time hybrid systems over unbounded time. It allows hybrid systems with non-linear ordinary differential equations, and non-linear jumps. Even though it is based on fast machine-precision floating point arithmetic, it uses sound rounding, and hence the correctness of its results cannot be hampered by round-off errors. HSolver does not only verify safe hybrid systems, but—in addition—it also computes abstractions of the input system. So, even for input systems that are unsafe, or for which exhaustive formal verification is too difficult, it will compute abstractions that can be used by other tools. For example, the abstractions could be used for guiding search for error trajectories of unsafe systems. Efficiency improvements of the tool are guided by a continuously expanding database of benchmark examples.

HSolver currently is not yet optimized for special classes of hybrid systems (e.g., systems such as linear hybrid automata that have very simply continuous dynamics). Moreover it does not yet provide mature support for finding counter-examples for unsafe input systems.

The method used by HSolver is interval constraint propagation based abstraction refinement [3]. This method incrementally refines an abstraction of the input systems. Special care is taken to reflect as much information as possible into the abstraction without increasing its size.

Chapter 2

Invocation

In the default mode, the software asks for input from the user in a format described in the next section. Usually one will write the input to a file and let the software read it from this file using a pipe (./hsolver <file).

In interactive mode (./hsolver -i), the software asks for input from the user using detailed questions. This mode is less flexible and is only provided for compatibility with earlier versions.

In the default mode, the software does verification by removing points from the state space that are not on any trajectory from an initial to an unsafe state, and hence it also may remove points in the reach set. In order to get reach set information, call HSolver with the parameter -f. Then the software only does forward checking—removing points that are not on any trajectory from an initial state, but does not do backward checking—removing points that do not lead to an unsafe state. It is also possible to do reasoning in the opposite direction: After calling HSolver with the parameter -b it computes backward reach set information, that is, points that might lead to an unsafe state.

For examples with simple continuous dynamics (e.g., clocks), slice computation improves the performance of HSolver . This feature can be turned on with the flag -s.

Call HSolver with the parameter -h to get information on further parameters.

Chapter 3

The Input

The following is an example input file:

```
VARIABLES [ x, y ]
MODES [ m1 ]
STATESPACE
  m1[[0, 4], [0, 4]]
INITIAL
  m1{x>=2.5/\x<=3/\y=0}
FLOW
  m1{x_d=x-y}{y_d=x+y}
JUMP
UNSAFE
  m1{x<=2}
```

As can be seen, the input allows several keywords followed by additional information. The keywords are:

- VARIABLES: a list of the names of the variables spanning the continuous state space
- MODES: a list of the names of the discrete modes
- STATESPACE: for each mode, the hyper-rectangle spanning the corresponding continuous state space
- INITIAL: for each mode, a constraint describing the set of initial states in this mode
- FLOW: For each mode, some constraints restricting the possible flow in this mode:
  - In braces, for each state space variable, a constraint describing the continuous evolution of this variable. The i-th constraint may

contain the variables as specified using the keyword VARIABLES, and the i-th variable followed by _d. The latter represents the derivative of this variable.

- For linear differential equations, it is also possible to specify some eigenvalue and eigenvector related information here [2, 4] (for left eigenvalues λ an orthonormal basis of $\{c : A^T c = \lambda c\}$, and for factors of the form y^2+ay+b of the left characteristic polynomial an orthonormal basis of $\{c : ((A^T)^2 - 2aA^T + (a^2 + b^2)I)c = 0\}$). This will provide the solver with additional optimization possibilities. Some examples of input files using this feature can be found in the distribution.
- Delimited by {| and |} one can specify a constraint that specifies the possible flows explicitly (i.e., as a function of time instead of a differential equation). In addition to the state space variables, this constraint may contain the state space variables with the additional postfix `_s`, and the time variable `t`. It specifies that a flow taking time `t` is possible starting from the variables with the postfix `_s` to the variables without postfix.

Note that the notion of "trajectory" is defined in such a way that along flows, such a trajectory has to be differentiable. Hence one cannot use the constraints under the keyword FLOW to specificy non-differentiable evolution, for example, the specification of trajectories that change from following one differential equation to a different one.

- JUMP: for each pair of modes, a constraint describing discontinuous jumps of trajectories. For example, we use the string $m1->m2\{x = 1 \wedge y = 2 \wedge x' = 3 \wedge y' = 5\}$ for a jump from mode m_1 to mode m_2. The constraint may contain the variables as specified using the keyword VARIABLES, and their primed versions. The unprimed versions describe the jump source and the primed versions the jump target. Jumps may occur between all pairs of states that are solutions of this constraint.
- UNSAFE: for each mode, a constraint describing the set of unsafe states in this mode

The constraints should be formulated in the syntax of RSolver . More example inputs are contained in the distribution package.

Throughout, the input allows comments in the form `(* comment *)`.

For debugging syntax errors, it may help to produce a parser trace by setting the Shell environment variable `OCAMLRUNPARAM` to `p` (for example, in Bash, using `export OCAMLRUNPARAM=p`).

Note that the formalism allows the modeling of invariants using constraints on the state space variables in the FLOW and JUMP sections.

When doing so, the start- and the endpoint of a jump should satisfy the invariant of the source and the target mode, respectively. For example, an invariant $x < 1$ in mode $m1$ and $x \geq 1$ in mode $m2$ does *not* allow jumps from mode $m1$ to mode $m2$ that leave x unchanged. If such jumps are intended, the invariant of mode $m1$ should be $x \leq 1$.

Note also that the conditions for jumping from a mode $m1$ to a mode $m2$ should be well-separated from the conditions for jumping back from mode $m2$ to $m1$. For example, in the case when a hybrid system uses dynamics $\dot{x} = f_1(x)$ whenever $x < 0$ and $\dot{x} = f_2(x)$ whenever $x \geq 0$, this should be modeled using a jump from a mode with dynamics f_1 to a mode with dynamics f_2 for $x > \varepsilon$, and a jump back for $x \leq -\varepsilon$. This takes into account switching delays occurring in actual system implementations, and avoids infinite loops between the two modes.

Chapter 4

Program Termination

The program terminates if the safety properties can be proved, or if the number of abstract states exceeds a certain bound, or if the maximal box diameter after calling the splitting and pruning algorithm is less than the given float.

Note that the default value for the maximal number of abstract states is 1000 and the default value for the maximal box diameter bound is 0.01. The user can run "./hsolver -h" for help and then choose other values.

Chapter 5

Produced Output

After every refinement step, the procedure prints the number of boxes used for representing the abstraction of the hybrid system. For example,

```
***3 s:  *** 5 ***
```

means that after the third refinement step, the abstraction consists of five boxes.

In recent versions, HSolver additionally prints the currently known upper bound on the volume of all starting points of error trajectories (`initvol:`), and the currently known upper bound on the volume of all endpoints of error trajectories (`unsafevol:`). Naturally, as soon as one of these values turns zero, the property has been proven, and HSolver terminates. Note however, that for computing these values, volumes of boxes are added in floating point arithmetic—without conservative rounding. Hence, these values should be considered as approximations.

Upon termination, HSolver outputs some statistics, and "INPUT SAFE" if the verification succeeded, or "SAFETY UNKNOWN" if it did not succeed. Our precise definition of safety of a hybrid system can be found in our publications [3].

If some boxes are left in the abstraction, they are printed, with some additional information. For example:

```
7: (1[ [ 4.875, 5.3125][ 1.5, 1.75] ]) L: 1 -> 21 13  <- 19 15 11
```

This means that the abstract state with id 7 represents the above box in mode 1. Numbers directly after the string "L: " represent labels of this abstract state (currently label 0 means that the abstract state is labeled as being initial, and label 1 means that the abstract state is labeled as being unsafe). After the string `->` follows a list of ids to which there is a transition from the current id, and after the string `<-` follows a list of ids from which there is a transition to the current one. These lists might contain some ids twice, denoting that there is both a flow and a jump transition.

In forward checking mode (command-line option -f) this list is an over-approximation of the set of state reachable from an initial state (i.e., the forward reach set), and in backward checking mode (command-line option -b) it is an over-approximation of the set of states that might lead to an unsafe state (i.e., the backward reach set).

Chapter 6

Graphical Output of State Space Analysis

The binary `hsolver_gui` is able to visualize the state space produced after each refinement step of the algorithm. For this, call HSolver as follows:

```
./hsolver -ds <input.hs | ./hsolver_gui
```

Chapter 7

Graphical Output of Abstraction

The software is able to print the output in dot language, a language that provides a simple way of describing graphs that both humans and computer programs can use.

Two command line options are supported:

- `-d` produces only one graph at the end of the computation
- `-ds` produces a graph after each splitting step of the algorithm

There are various programs that can further process such output, for example rendering it into some image format. With the tool Graphviz the user can easily render graphics from HSolver . For example, the command

```
./hsolver -d -a 5 < input.hs | dot -Tgif -o output.gif.
```

will create an image file output.gif at the moment when the number of abstract states reaches 5. Clutter can be removed by filtering the output through `tred` which removes edges implied by transitivity.

Nodes of the graph represent abstract states of the hybrid system. They are grouped by the mode they represent. The list of possible transitions follows a list of all nodes. The meaning of the coloring of nodes is as follows:

- green - the node may contain some initial points
- red - the node may contain some unsafe points
- blue - the node may contain both initial and unsafe points
- yellow border - the node occurs on the shortest abstract error trajectory

The produced output would look like this:

```
digraph "Hybrid system" {
  subgraph cluster_n
  {
    label= "n"
    0 [ label = "0[ [ 0., 0.425][ 0., 0.85] ]" color = green];
    1 [ label = "0[ [ 0.425, 0.85][ 0.15, 0.575] ]" color = black];
    2 [ label = "0[ [ 0.425, 0.85][ 0.575, 1.] ]" color = red];
  }

  0 -> 1;
  0 -> 2;
  1 -> 2;
}
```

Chapter 8

The Algorithm

A detailed description of the algorithm employed by this software can be found in our papers [1, 2, 3]. We implemented the algorithm on top of our RSolver package that provides pruning and solving of quantified constraints.

Chapter 9

Known Bugs

- The definition of safety of a hybrid systems used in our publications [3] assumes that for trajectories, on flows of length zero, the flow constraints (keyword "FLOW") does not have to hold. In our current implementation it even has to hold on flows of length zero.

Bibliography

[1] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *LNCS*, pages 573–589. Springer, 2005.

[2] S. Ratschan and Z. She. Constraints for continuous reachability in the verification of hybrid systems. In *Proc. 8th Int. Conf. on Artif. Intell. and Symb. Comp., AISC'2006*, number 4120 in LNCS, pages 196–210. Springer, 2006.

[3] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions in Embedded Computing Systems*, 6(1):1–23, 2007. article no. 8.

[4] A. Tiwari. Approximate reachability for linear systems. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 2623 of *LNCS*. Springer, 2003.