— HySAT Quick Start Guide —

Christian Herde

December 20, 2009

1 Introduction

HySAT is a satisfiability checker for Boolean combinations of arithmetic constraints over real– and integer–valued variables. A peculiarity of HySAT, which sets it apart from many other solvers, is that it is not limited to linear arithmetic, but can also deal with nonlinear constraints involving transcendental functions.

The algorithmic core of HySAT is the iSAT algorithm, a tight integration of recent SAT solving techniques with interval-based arithmetic constraint solving. For technical details, see [?]. HySAT is the successor tool and shares the name of the solver described in [?]. Do not mix up the two.

A Linux binary of HySAT is available at `http://hysat.informatik.uni-oldenburg.de`. On the website you will also find some sample input files and this manual. HySAT is an ongoing project. Please check back regularly for updated versions of the tool.

This document shall provide a brief introduction into the usage of HySAT. Unfortunately, it is still far from being complete. If your question is not answered here, send an email to `christian.herde@informatik.uni-oldenburg.de`.

2 Modes of operation

HySAT is a command-line tool which has two different modes of operation: It can be used a) as a satisfiability checker for a single formula, which in particular can seek an optimal solution to the given formula, and b) for finding a trace of a hybrid system via bounded model checking (BMC). The usage of HySAT is best illustrated by means of examples.

2.1 Single formula mode

We first explain the use of HySAT as satisfiability checker, thereafter how to solve optimization problems with it.

2.1.1 (Un)satisfiability checking

Assume you want to find a pythagorean triple, i.e. a triple (a, b, c) of integer values which satisfies $a^2 + b^2 = c^2$. To use HySAT for this purpose, create a file, say `sample1.hys`, containing the following lines (without the line numbers).

```
DECL
     -- The range of each variable has to be bounded.
     int [1, 100] a, b, c;

EXPR
     -- Constraint to be solved.
     a*a + b*b = c*c;
```

In single formula mode the input file has two sections: The first section, starting with the keyword DECL, contains declarations of all variables occuring in the formula to be solved. The second section, starting with EXPR, contains the formula itself, the latter in this example consisting of a single arithmetic constraint only. After calling '`hysat sample1.hys`', HySAT generates the output displayed below[1], reporting $a = 80, b = 60$, and $c = 100$ as a satisfying valuation.

```
# This is HySAT 0.8.5, compiled on Sat Nov 14, 2009.

Reading input file 'sample1.hys'.
Preprocessing input formulae.

SOLVING:
    input formula

RESULT:
    candidate solution box found

SOLUTION:
    a (int):
        @0: [51, 51]

    b (int):
        @0: [68, 68]

    c (int):
        @0: [85, 85]
```

You might have noticed, that HySAT writes the result in form of *intervals* instead of single values. This is due to the fact that calculations in HySAT are carried out in *interval arithmetic*. In contrast to the examples presented in this brief introduction, the solution intervals computed by HySAT will in general be non-point intervals.

2.1.2 Optimization

In single formula mode, HySAT can also be used to determine a solution which minimizes the value of an arbitrary variable occuring in the input formula. Internally, this is achieved by calling the satisfiability checker within a binary search scheme. To compute, for example, a pythagorean triple which minimized the value of variable c, call '`hysat --minimize c sample1.hys`', which will generate the following output:

```
# This is HySAT 0.8.5, compiled on Sat Nov 14, 2009.

Reading input file 'sample1.hys'.
Preprocessing input formulae.

MINIMIZING:
    value of variable 'c'
    objective value: 85
    objective value: 34
    objective value: 15
    objective value: 5
    objective value: 5

SOLUTION:
    a (int):
        @0: [3, 3]

    b (int):
        @0: [4, 4]

    c (int):
        @0: [5, 5]
```

That is, the optimum is attained at $c = 5$.

Since the input formula may be built from Boolean, integer, and real–valued variables using all standard Boolean and arithmetic operations, the class of optimization problems which can

[1]For brevity, we have omitted the sections 'SETTINGS' and 'STATISTICS' of the tool output here and in the following examples.

be solved by HySAT is very general and in particular includes nonlinear, nonconvex, and mixed–integer problems.

Being based on interval propagation, HySAT will find the optimum also in pathological cases. Consider, e.g., the function $y = x^2 - 0.3e^{-(a(x-\frac{1}{2}))^2}$ for $a = 200$ over the interval $[-200, 200]$[2]. A rough plot of the function using gnuplot, for example, seems to indicate that the global optimum is at $(x, y) = (0, 0)$. A closer inspection of the graph however reveals that the function has a sharp bend in the interval $[0.48, 0.52]$ which contains the global miminum $(x, y) \approx (0.5, -0.05)$. Being run on the input file

```
DECL
    define a = 200;
    float [-a, a] x, y;

EXPR
    y = x^2 - 0.3 * exp(-(a*(x - 0.5))^2);
```

(using command line option `--minimize y`), HySAT encloses this mimimum correctly:

```
# This is HySAT 0.8.5, compiled on Sat Nov 14, 2009.

Reading input file 'sample2.hys'.
Preprocessing input formulae.

MINIMIZING:
    value of variable 'y'
    objective value: 2.70385
    objective value: 1.10819
    objective value: 0.268511
    objective value: 0.00964123
    objective value: -0.0460648
    objective value: -0.0498516
    objective value: -0.0500207
    objective value: -0.0500208

SOLUTION:
    x (float):
        @0: [0.49995866529807836409, 0.4999586656049818667]

    y (float):
        @0: [-0.050020831307689694878, -0.050020830695775175823]
```

Note, by the way, that the declaration section of the input file contains a definition of a symbolic constant ($a = 200$), which thereafter can be used in declarations and constraints.

2.2 Bounded model checking mode

Bounded model checking (BMC) aims at checking whether a system has a run of bounded length k which

- starts in an initial state of the system,
- obeys the system's transition relation and
- ends in a state in which a certain (desired or undesired) property holds.

The idea of BMC is to construct a formula which is satisfiable if and only if a trace with above properties exists. In case of satisfiability, any satisfying valuation of this formula corresponds to such a trace.

For specifying BMC tasks iSAT has a second input file format which consists of four parts:

- `DECL`: As above, this part contains declarations of all variables. Furthermore, you can define symbolic constants in this section (see the definition of `f` in line 2 in the example below).

[2]This example was taken from the lecture 'Validated Numerics' by Warwick Tucker, Uppsala University.

- `INIT`: This part is a formula describing the initial state(s) of the system to be investigated. In the example below, `x` is initialized to 0.6, and `jump` is set to `false`, since this is the only valuation which satisfies the constraint `!jump`, where '`!`' stands for '`not`'.
- `TRANS`: This formula describes the transition relation of the system. Variables may occur in primed or unprimed form. A primed variable represents the value of that variable in the successor step, i.e. after the transition has taken place. Thus, line 14 of the example states, that if `jump` is `false` in the current state, then the value of `x` in the next state is given by its current value plus 2.

  The semicolon which terminates each constraint can be read as an AND-operator. Hence, `TRANS` is a conjunction of three constraints.
- `TARGET`: This formula characterises the state(s) whose reachability is to be checked. In the example below, we want to find out if a state is reachable in which `x > 3.5` holds.

```
 1  DECL
 2      define f = 2.0;
 3      float [0, 1000] x;
 4      boole jump;
 5
 6  INIT
 7      x = 0.6;
 8      !jump;
 9
10  TRANS
11      jump' <-> !jump;
12
13      jump -> f * x' = x;
14      !jump -> x' = x + 2;
15
16  TARGET
17      x > 3.5;
```

When calling HySAT with the input file above, it successively unwinds the transition relation $k = 0, 1, 2, \ldots$ times, conjoins the resulting formula with the formulae describing the initial state and the target states, and thereafter solves the formula thus obtained.

From the tool ouput below you can see that for $k = 0, 1, 2, 3, 4$, the formulae are all unsatisfiable, for $k = 5$ however, a solution is found. Note, that HySAT reports the values of `jump` and `x` for each step k of the system. After the last transition, as required, `x > 3.5` holds.

```
# This is HySAT 0.8.5, compiled on Sat Nov 14, 2009.

Reading input file 'sample3.hys'.
Normalizing input formulae.

SOLVING:
    k = 0

RESULT:
    unsatisfiable

SOLVING:
    k = 1

RESULT:
    unsatisfiable

SOLVING:
    k = 2

RESULT:
    unsatisfiable

SOLVING:
    k = 3

RESULT:
    unsatisfiable
SOLVING:
    k = 4

RESULT:
    unsatisfiable

SOLVING:
    k = 5

RESULT:
    candidate solution box found

SOLUTION:
    jump (boole):
        @0: [0, 0]
        @1: [1, 1]
        @2: [0, 0]
        @3: [1, 1]
        @4: [0, 0]
        @5: [1, 1]

    x (float):
        @0: [0.6, 0.6]
        @1: [2.6, 2.6]
        @2: [1.3, 1.3]
        @3: [3.3, 3.3]
        @4: [1.65, 1.65]
        @5: [3.65, 3.65]
```

3 Input language: types, operators and expressions

3.1 Types

- Types supported by HySAT are `float`, `int` and `boole`.
- Boolean variables are identified with integer variables of range $[0, 1]$.
- When declaring a float or an integer variable you have to specify the range of this variable. Due to the internal working of HySAT these ranges have to be bounded, i.e. you have to specify a lower and an upper bound. To reduce solving time, ranges should be chosen as small as possible.
- Integer and float variables can be mixed within the same arithmetic constraint.

3.2 Operators

- Boolean operators:

Operator	Type	Num. Args.	Meaning
`and`	infix	2	conjunction
`;`	infix	2	alternative notation for '`and`', cf. precedence rules, however
`or`	infix	2	disjunction
`nand`	infix	2	negated and
`nor`	infix	2	negated or
`xor`	infix	2	exclusive or
`nxor`	infix	2	negated xor, i.e. equivalence
`<->`	infix	2	alternative notation for '`nxor`'
`impl`	infix	2	implication
`->`	infix	2	alternative notation for '`impl`'
`not`	prefix	1	negation
`!`	prefix	1	alternative notation for '`not`'

- Arithmetic operators:

Operator	Type	Num. Args.	Meaning
`+`	infix	1 or 2	unary 'plus' and addition
`-`	infix	1 or 2	unary 'minus' and subtraction
`*`	infix	2	multiplication
`abs`	prefix	1	absolute value
`min`	prefix	2	minimum
`max`	prefix	2	maximum
`exp`	prefix	1	exponential function
`sin`	prefix	1	sine (unit: radian)
`cos`	prefix	1	cosine (unit: radian)
`pow`	prefix	2	nth power, n (2nd argument) has to be a positive integer
`^`	infix	2	alternative notation for '`pow`', infix however
`nrt`	prefix	2	nth root, n (2nd argument) has to be a positive integer

  Operators `abs`, `min`, `max`, `exp`, `sin`, `cos`, `pow`, and `nrt` have to be called with their arguments being separated by commas and enclosed in brackets. Example: `pow(x, 3)`

- Relational operators:

  `>`, `>=`, `<`, `<=`, `=`, `!=` (all infix)

- Precedence of operators: The following list shows all operators ordered by their precedence, starting with the one that binds strongest. You can use brackets in the input file to modify the order of term evaluation induced by these precedence rules.

▷ unary not, `^` (infix version of '`pow`')

▷ unary plus, unary minus

▷ multiplication

▷ plus, minus

▷ `abs`, `min`, `max`, `exp`, `sin`, `cos`, `pow`, `nrt`

▷ `>`, `>=`, `<`, `<=`, `=`, `!=`

▷ `and`, `nand`

▷ `xor`, `nxor`

▷ `or`, `nor`

▷ `impl`

▷ `;`

3.3 Expressions

- Let `a`, `b` be Boolean variables and `x`, `y` be float variables. Examples for expressions are:

  ▷ `[x * y + 2 * (x - 4) >= 5 - 2 * (x - 4)]`

  ▷ `(x > 20 and !b) xor a`

  ▷ `b <-> {3.18 * (-5 - y * y * y) = 7}`

  ▷ `sin(x + max(3, y)) < 0.4`

  ▷ `abs(nrt(x^2 + y^2), 2)) <= 10.3`

- Note that the individual constraints occuring in a formula have to be conjoined using the '`;`'-operator. In addition, the last line of each formula has to be terminated with a semicolon.

4 How it works

To be able to interprete HySAT's output you need some basic understanding of how the tool works internally. HySAT performs a backtrack search to prune the search space until it is left with a 'sufficiently small' portion of the search space for which it cannot derive any contradiction with respect to the constraints occuring in the input formula.

Initially, the search space consists of the cartesian product of the ranges of all variables occuring in the formula to be solved. Just like an ordinary (purely Boolean) SAT solver, HySAT operates by alternating between two steps:

- The *decision step* involves selecting a variable 'blindly', splitting its current interval (e.g. by using the midpoint of the interval as split point) and temporarily discarding either the lower or the upper half of the interval from the search. The solver will ignore the discarded part of the search space until the decision is undone by backtracking.

- Each decision is followed by a *deduction step* in which the solver applies a set of deduction rules that explore all the consequences of the previous decision. Informally speaking, the deduction rules carve away portions of the search space that contain non-solutions only.

Assume, for example, that the input formula consists of the single constraint $x \cdot y = 8$ and initially $x \in [2, 4]$ and $y \in [2, 4]$ holds. The solver might now decide to split the interval of x by assigning the new lower bound $x \geq 3$ to x. In the subsequent deduction phase, the solver will deduce that, due to the increased lower bound of x, the upper bound of y can be reduced to $\frac{8}{3}$ because for all other values of y the constraint $x \cdot y = 8$ is violated. After asserting $y \leq \frac{8}{3}$, thereby contracting the search space to $[3, 4] \times [2, \frac{8}{3}]$, no further deductions are possible, and the solver goes on with taking the next decision. Deduction may also yield a conflict, i. e. a variable whose interval is empty, indicating the need to backtrack.

To enforce termination of the algorithm, the solver only selects a variable x for splitting if the width $\overline{x} - \underline{x}$ of its interval $[\underline{x}, \overline{x}]$ is above a certain threshold ε, which we call *minimal splitting width*. Furthermore, the solver discards a deduced bound if it only yields a comparatively small progress with respect to the bound already in place. More precisely, a deduced lower (upper) bound b_d is ignored if $|b_c - b_d| \leq \delta$, where b_c is the current lower (upper) bound of the respective variable and δ is a parameter which we call *minimal bound delta*. The values of ε and δ can be set with the command-line options `--msw` and `--mbd`. Their default values are $\varepsilon = 0.1$ and $\delta = 0.01$.

These measures taken to enforce termination have some consequences which are important to understand:

- If HySAT terminates with result '`unsatisfiable`', then – assuming that there are no bugs in the implementation – you can be sure, that the formula is actually unsatisfiable.

- If the solver stops with the result '`candidate solution box found`', this means that for the given ε and δ the solver could not detect any conflicts within the reported box. It does *not* mean, however, that the box actually contains a solution. None the less, you will find, that in most cases there *will* be a solution within the box or at least nearby.

If you think that HySAT has reported a spurious solution you should re-run the solver with a smaller ε and δ in order to confirm (or refute) the previous result. In [**?**] we have proposed methods which allow in certain cases to report solutions which provably satisfy the input formula. For the paper we have benchmarked prototypic implementations of these methods. Because of their experimental character we have opted for removing these methods from the version of the solver which is available online.

5 Tool options

5.1 Restarting

HySAT mimics the restart policy implemented in PicoSAT[3]. Restarting is disabled by default. To enable restarting use the command line option `--rst`, followed by four parameters: a_{init} (integer), b_{init} (integer), f_a (float) and f_b (float). The parameters have to satisfy $a_{\text{init}} \leq b_{\text{init}}$ and $1 < f_a \leq f_b$. Doing so, the n-th restart will happen a_n conflicts after the $(n-1)$-th restart, i.e. after $\sum_{i=1}^{n} a_i$ conflicts in total, where a_n is defined as follows:

$$
\begin{aligned}
a_1 &= a_{\text{init}} \\
b_1 &= b_{\text{init}} \\
a_n &= \begin{cases} a_{\text{init}} & \text{if } a_{n-1} \geq b_{n-1} \\ \lfloor f_a \cdot a_{n-1} \rfloor & \text{otherwise} \end{cases} \\
b_n &= \begin{cases} \lfloor f_b \cdot b_{n-1} \rfloor & \text{if } a_{n-1} \geq b_{n-1} \\ b_{n-1} & \text{otherwise} \end{cases}
\end{aligned}
$$

Choosing $a_{\text{init}} = 100$, $b_{\text{init}} = 200$, $f_a = 1.5$ and $f_b = 2.0$ yields the following restart schedule:

n	1	2	3	4	5	6	7	8	9	10
a_n	100	150	225	100	150	225	337	505	100	150
b_n	200	200	200	400	400	400	400	400	800	800
$\sum_{i=1}^{n} a_i$	100	250	475	575	725	950	1287	1792	1892	2042

PicoSAT uses the following settings: $a_{\text{init}} = b_{\text{init}} = 100$ and $f_a = f_b = 1.1$. To run HySAT with PicSAT's restart schedule start it with option '`--rst 100 100 1.1 1.1`'.

[3]http://fmv.jku.at/picosat/

6 Frequently asked questions (FAQ)

- *When applied to purely Boolean problems, would HySAT be competitive with state-of-the-art SAT solvers like MiniSAT or Chaff?*

  No, clearly not. The main reason is that the data structures employed in HySAT are rather general and not optimized w.r.t. Boolean variables. Indeed Booleans, integers, and floats are internally represented by the same type of object which makes handling of Boolean atoms much less efficient than it could be. Moreover, HySAT lacks all the low-level code optimizations (e.g. those which aim at improving the caching behaviour of the code) that are common in todays SAT solvers. This said, I can see no principle reason why a more elaborate implementation of the iSAT algorithm should be significantly slower on purely Boolean problems than a standard SAT solver would be.

- *Is there a difference between writing, e.g., '`y*y*y`' and '`y^3`' in an input file?*

  Yes, there is. The deduction rules for the power operator are much more powerful than those for multiplication. When you have the choice, you should always opt for the power operator.

- *Does HySAT support pseudo–Boolean constraints?*

  In contrast to its predecessor tool, the new HySAT does not contain any optimizations that are specific for pseudo–Boolean constraint solving. Nevertheless you can use pseudo–Boolean constraints in your model, because HySAT can deal with arbitrary integer constraints. Since Boolean variables are 0–1 integer variables, the only thing you have to do is to slightly rewrite your pseudo–Boolean constraints by replacing each negative literal $\neg b$ with $(1 - b)$. The constraint $2\neg a + b + c \geq 2$, for example, becomes $2 \cdot (1 - a) + b + c \geq 2$, i.e. $-2a + b + c \geq 0$ this way.

7 Final remarks

I implemented the current version of HySAT as part of my PhD work. The main purpose of the implementation was to evaluate the concepts described in [**?**], rather than to provide the most efficient code implementing them. Furthermore, the current implementation still lacks several important features of modern SAT solvers (e.g. a more sophisticated heuristics for decision-making, forgetting of learned clauses, progress saving, etc.) which probably would make the tool much faster. This said, HySAT is suprisingly efficient and has solved formulas with some ten thousands of float variables. I'd be interested to hear which problems you could solve with it.

Solvers are fairly complex pieces of software; as such, their implementation is prone to errors. Though I did a lot of testing, HySAT is probably not free from bugs. Please be aware that neither I nor the University of Oldenburg are liable for any losses or damages arising from the use of the tool or its documentation. If you encounter errors or strange behaviour, please drop me an email with a problem description, your input file, and instructions on how to reproduce the error. I will try to fix the problem as soon as I can devote time to it.

ChangeLog

- Version 0.8.1:
  - ▷ First release.
- Version 0.8.2:
  - ▷ Fixed a bug in the deduction rule of the `pow`-operator.
  - ▷ Added restarts.
- Version 0.8.3:
  - ▷ Fixed a bug concerning shifting of clauses.
  - ▷ Added variable order 'shuffle'.
- Version 0.8.4:
  - ▷ Fixed some minor bugs.
- Version 0.8.5:
  - ▷ Added optimization mode (command line option `--minimize`).
- Version 0.8.6:
  - ▷ Fixed a bug in the decision procedure: The problem occured if the solver tried to split an interval $[\underline{x}, \overline{x}]$, where $\underline{x}$ and $\overline{x}$ are huge float values. In this case it may happen that no floating point number x with $\underline{x} < x < \overline{x}$ exists, although the width $w = \overline{x} - \underline{x}$ of the interval is still fairly large. This situation was not handled properly in the source code and could misguide the solver into an 'almost-infinite loop' which was only left by chance and caused a rapidly increasing memory usage.

References

[FH06] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 2006.

[FHR$^+$07] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT Special Issue on Constraint Programming and SAT*, 2007. Accepted for publication.