iSAT3 Manual (`isat3 0.03-20161213`)*

Karsten Scheibler

December 13, 2016

1 Introduction

This document provides a brief introduction into the usage of the iSAT3 binary and the iSAT3 library. iSAT3 is a satisfiability checker for Boolean combinations of arithmetic constraints over real- and integer-valued variables. Those constraints may contain linear and non-linear arithmetic as well as transcendental functions. iSAT3 is the third implementation of the iSAT algorithm [FHT+07]. All three implementations (with HySAT [FH06, HEFT08] and iSAT [EKKT08] being the first two) share the same major core principles of tightly integrating ICP into the CDCL framework. But while the core solvers of HySAT and iSAT operate on simple bounds, the core of iSAT3 uses literals and additionally utilizes an ASG for more advanced formula preprocessing [SKB13]. Furthermore, iSAT3 includes support for accurate floating-point reasoning (since version 0.03).

2 The iSAT3 Binary

2.1 Modes of Operation

The command-line tool iSAT3 has two different modes of operation: It can be used (1) as a satisfiability checker for a single formula or (2) for finding a trace of a hybrid system via bounded model checking (BMC). In the following the usage of iSAT3 is illustrated by means of some examples. iSAT3 understands the same input language as HySAT and iSAT.

2.1.1 Single Formula Mode

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 iSAT3 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;
```

*This work has been partially funded by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org/`).

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, in this case consisting of a single arithmetic constraint only. After calling iSAT3 the following output is generated, reporting $a = 3, b = 4$ and $c = 5$ as a satisfying valuation.

```
# isat3 -I -v sample1.hys
opening 'sample1.hys' for reading
parsing (user time 0.000000 seconds)
rewrite (user time 0.000000 seconds)
cnf generation (user time 0.000000 seconds)
starting to solve
a:
    [3, 3] -- point interval
b:
    [4, 4] -- point interval
c:
    [5, 5] -- point interval
SATISFIABLE [satisfiable with all values in the given intervals]
```

You might have noticed, that iSAT3 writes the result in form of *intervals* instead of single values. This is due to the fact that calculations in iSAT3 are carried out in *interval arithmetic*. In contrast to the examples presented in this brief introduction, the solution intervals computed by iSAT3 will in general be non-point intervals.

2.1.2 Bounded Model Checking Mode

Bounded model checking (BMC) of a hybrid system aims at finding 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, iSAT3 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).
- `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 0.5, 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 (`x'`) or unprimed (`x`) 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` in the example 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.

```
DECL
        define f = 2.0;
        real [0, 1000] x;
        boole jump;

INIT
        x = 0.5;
        !jump;

TRANS
        jump' <-> !jump;

        jump -> f * x' = x;
        !jump -> x' = x + 2;

TARGET
        x > 3.5;
```

When calling iSAT3 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 excerpts of 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 iSAT3 reports the values of `jump` and `x` for each step k of the system. After the last transition, as required, `x > 3.5` holds.

```
# isat3 -I -v -v sample2.hys
opening 'sample2.hys' for reading
parsing (user time 0.000000 seconds)
rewrite (user time 0.000000 seconds)
cnf generation (user time 0.000000 seconds)
starting to solve
depth 0 is UNSATISFIABLE (user time 0.000000 seconds)
depth 1 is UNSATISFIABLE (user time 0.000000 seconds)
depth 2 is UNSATISFIABLE (user time 0.000000 seconds)
depth 3 is UNSATISFIABLE (user time 0.000000 seconds)
depth 4 is UNSATISFIABLE (user time 0.000000 seconds)
depth 5 is SATISFIABLE [satisfiable with all values in the given intervals] ...
jump@0:
    false
x@0:
    [0.5, 0.5] -- point interval
x@1:
    [2.5, 2.5] -- point interval
jump@1:
    true
x@2:
    [1.25, 1.25] -- point interval
jump@2:
    false
x@3:
    [3.25, 3.25] -- point interval
jump@3:
    true
x@4:
    [1.625, 1.625] -- point interval
jump@4:
    false
x@5:
    [3.625, 3.625] -- point interval
jump@5:
    true
UNSAFE [target is reachable with all values in the given intervals]
```

2.2 Input Language: Classical Types, Operators, and Expressions

2.2.1 Types

- Types supported by iSAT3 are: `boole`, `int`, and `real`. In order to maintain compatibility with the input language of all earlier versions `real` is an alias for `float`. Historically, `float` is the type for all real-valued variables, but because this term is misleading, the alias `real` was added.

- When declaring an integer or a real-valued variable you have to specify the range of this variable. Due to the internal working of iSAT3 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.
- Boolean, integer, real-valued variables can be mixed within the same arithmetic constraint.

2.2.2 Operators

- Boolean operators:

Operator	Type	Num. Args.	Meaning
`and`	infix	2	conjunction
`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
`ite`⋆	prefix	3	if-then-else
`exp`	prefix	1	exponential function regarding base e
`exp2`⋆	prefix	1	exponential function regarding base 2
`exp10`⋆	prefix	1	exponential function regarding base 10
`log`	prefix	1	logarithmic function regarding base e
`log2`⋆	prefix	1	logarithmic function regarding base 2
`log10`⋆	prefix	1	logarithmic function regarding base 10
`sin`	prefix	1	sine (unit: radian)
`cos`	prefix	1	cosine (unit: radian)
`pow`	prefix	2	nth power, n (2nd argument) has to be an integer, $n \geq 0$
`^`	infix	2	nth power, n (2nd argument) has to be an integer, $n \geq 0$
`nrt`	prefix	2	nth root, n (2nd argument) has to be an integer, $n \geq 1$

  Operators `abs`, `min`, `max`, `ite`, `exp`, `exp2`, `exp10`, `log`, `log2`, `log10`, `sin`, `cos`, `pow` and `nrt` have to be called with their arguments being separated by commas and enclosed in brackets, e.g. `min(x, y)`. Operators marked with $\star$ are only available when the option `--extended-hys-syntax` is set.

- Relational operators: `>`, `>=`, `<`, `<=`, `=`, `!=` (all infix). Note that `(1 < x < 2)` **is not the same** as `((1 < x) and (x < 2))`. `(1 < x < 2)` is interpreted as `((1 < x) < 2)` with `(1 < x)` being either `0` or `1` depending if the condition is false or true.
- 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 plus, unary minus
- ▷ `*`
- ▷ `+`, `-`
- ▷ `abs`, `min`, `max`, `ite`, `exp`, `exp2`, `exp10`, `log`, `log2`, `log10`, `sin`, `cos`, `pow`, `nrt`
- ▷ `>`, `>=`, `<`, `<=`, `=`, `!=`
- ▷ `and`, `nand`
- ▷ `xor`, `nxor`
- ▷ `or`, `nor`
- ▷ `impl`

2.2.3 Expressions

- Let `a` and `b` be Boolean variables and `x` and `y` be real-valued 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(3.1 * min(x^2 + y^2, -x)) <= 10.3`
- Note that the individual constraints occurring 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.

2.3 Input Language: Constants, Types, Operators, and Expressions for Accurate Floating-Point Reasoning

The new types and operations were created with the basic C data types in mind (`double`, `float`, `char`, `short`, `int`, `long long`). Therefore, all types and operators start with the prefix `cl` which stands for C language. Furthermore, nearly all constants, types or operators targeting:

- the C-`double` data type start with the prefix `cl_double`
- the C-`float` data type start with the prefix `cl_float`
- the C-integer data types start with the prefix `cl_genint` (`genint` = generic integer)

Note that all listed types and operators are only available when the option `--extended-hys-syntax` is set.

2.3.1 Constants

Operator	Meaning
`cl_double_nan`	`cl_double` not-a-number (NaN)
`cl_double_neginf`	`cl_double` negative infinity
`cl_double_posinf`	`cl_double` positive infinity
`cl_double_min`	`cl_double` smallest normal number

Operator	Meaning
cl_double_max	cl_double largest normal number
cl_double_negzero	cl_double negative zero
cl_double_poszero	cl_double positive zero
cl_double_constant(number)	convert number to cl_double
cl_float_nan	cl_float not-a-number (NaN)
cl_float_neginf	cl_float negative infinity
cl_float_posinf	cl_float positive infinity
cl_float_min	cl_float smallest normal number
cl_float_max	cl_float largest normal number
cl_float_negzero	cl_float negative zero
cl_float_poszero	cl_float positive zero
cl_float_constant(number)	convert number to cl_float
cl_sint8_min	smallest number for a signed integer with 8 bits (-0x80)
cl_sint8_max	largest number for a signed integer with 8 bits (0x7f)
cl_uint8_min	smallest number for an unsigned integer with 8 bits (0x00)
cl_uint8_max	largest number for an unsigned integer with 8 bits (0xff)
cl_sint16_min	smallest number for a signed integer with 16 bits
cl_sint16_max	largest number for a signed integer with 16 bits
cl_uint16_min	smallest number for an unsigned integer with 16 bits
cl_uint16_max	largest number for an unsigned integer with 16 bits
cl_sint32_min	smallest number for a signed integer with 32 bits
cl_sint32_max	largest number for a signed integer with 32 bits
cl_uint32_min	smallest number for an unsigned integer with 32 bits
cl_uint32_max	largest number for an unsigned integer with 32 bits
cl_sint64_min	smallest number for a signed integer with 64 bits
cl_sint64_max	largest number for a signed integer with 64 bits
cl_uint64_min	smallest number for an unsigned integer with 64 bits
cl_uint64_max	largest number for an unsigned integer with 64 bits
cl_genint_constant(number)	convert number to cl_genint

Note that `cl_double_constant(-0)` and `cl_double_constant(0)` both represent the positive zero `cl_double_poszero`, because the unary minus is applied before the number conversion. Use `cl_double_negzero` to represent the negative zero. Further note that `cl_double_constant()`, `cl_float_constant()` and `cl_genint_constant()` expect exactly representable numbers, e.g. `cl_double_constant(0.1)` is not allowed. It is recommended to use the hexadecimal bit-precise representation of floating-point numbers, e.g. `cl_float_constant(0x1.99999ap-4)`. The following C-snippet demonstrates how `printf` could be used to output this format:

```
#include <stdio.h>

int main(void)
        {
        float   f = 0.1;
        double  d = 0.1;

        printf("%1.6a\n", f);
        printf("%1.13a\n", d);
        return (0);
        }
```

2.3.2 Types

- The following new types are supported by iSAT3: `cl_double`, `cl_float`, `cl_genint`.
- When declaring variables of these types you have to specify the initial range, e.g.

```
DECL
        -- f1 and d1 allow all values (NaN is always allowed and can not be
        -- excluded in the initial interval)
        cl_double [cl_double_neginf, cl_double_posinf] d1;
        cl_float [cl_float_neginf, cl_float_posinf] f1;

        -- d2 and f2 cutoff some values (but still allow NaN)
        cl_double [cl_double_negero, cl_double_constant(0x1.0p+1020)] d2;
        cl_float [cl_float_min, cl_float_negzero] f2;

        -- g1 and g2 have the same initial interval
        cl_genint [cl_genint_constant(-0x80), cl_genint_constant(0x7f)] g1;
        cl_genint [cl_sint8_min, cl_sint8_max] g2;

...
```

2.3.3 Operators

- Arithmetic operators:

Operator	Meaning
`cl_double_cast_genint(g)`	cast sub-expression `g` of type `cl_genint` to `cl_double`
`cl_double_cast_float(f)`	cast sub-expression `f` of type `cl_float` to `cl_double`
`cl_double_abs(d)`	absolute value
`cl_double_minus(d)`	unary minus
`cl_double_ite(b, d1, d2)`	if `b` is `true` then select `d1` otherwise `d2`
`cl_double_add(d1, d2)`	addition
`cl_double_sub(d1, d2)`	subtraction
`cl_double_mult(d1, d2)`	multiplication
`cl_double_div(d1, d2)`	division
`cl_float_cast_genint(g)`	cast sub-expression `g` of type `cl_genint` to `cl_float`
`cl_float_cast_double(d)`	cast sub-expression `f` of type `cl_float` to `cl_float`
`cl_float_abs(f)`	absolute value
`cl_float_minus(f)`	unary minus
`cl_float_ite(b, f1, f2)`	if `b` is `true` then select `f1` otherwise `f2`
`cl_float_add(f1, f2)`	addition
`cl_float_sub(f1, f2)`	subtraction
`cl_float_mult(f1, f2)`	multiplication
`cl_float_div(f1, f2)`	division
`cl_genint_scast_float(f, bw)`	cast sub-expression `f` of type `cl_float` to `cl_genint` (signed integer with constant bitwidth `bw`)
`cl_genint_ucast_float(f, bw)`	cast sub-expression `f` of type `cl_float` to `cl_genint` (unsigned integer with constant bitwidth `bw`)
`cl_genint_scast_double(d, bw)`	cast sub-expression `d` of type `cl_double` to `cl_genint` (signed integer with constant bitwidth `bw`)
`cl_genint_ucast_double(d, bw)`	cast sub-expression `d` of type `cl_double` to `cl_genint` (unsigned integer with constant bitwidth `bw`)
`cl_genint_scast(g, bw)`	cast sub-expression `g` to a signed integer with constant bitwidth `bw`
`cl_genint_ucast(g, bw)`	cast sub-expression `g` to an unsigned integer with constant bitwidth `bw`
`cl_genint_sanot(g, bw)`	bitwise arithmetic and of `g` with constant bitwidth `bw`, result is signed
`cl_genint_uanot(g, bw)`	bitwise arithmetic not of `g` with constant bitwidth `bw`, result is unsigned
`cl_genint_saand(g1, g2, bw)`	bitwise arithmetic and of `g1`, `g2` with constant bitwidth `bw`, result is signed
`cl_genint_uaand(g1, g2, bw)`	bitwise arithmetic and of `g1`, `g2` with constant bitwidth `bw`, result is unsigned
`cl_genint_saor(g1, g2, bw)`	bitwise arithmetic or of `g1`, `g2` with constant bitwidth `bw`, result is signed
`cl_genint_uaor(g1, g2, bw)`	bitwise arithmetic or of `g1`, `g2` with constant bitwidth `bw`, result is unsigned
`cl_genint_saxor(g1, g2, bw)`	bitwise arithmetic xor of `g1`, `g2` with constant bitwidth `bw`, result is signed
`cl_genint_uaxor(g1, g2, bw)`	bitwise arithmetic xor of `g1`, `g2` with constant bitwidth `bw`, result is unsigned

Operator	Meaning
`cl_genint_lshift(g, sv)`	bitwise left-shift of `g` with constant shift-value `sv`
`cl_genint_rshift(g, sv)`	bitwise right-shift of `g` with constant shift-value `sv`
`cl_genint_abs(g)`	absolute value
`cl_genint_minus(g)`	unary minus
`cl_genint_ite(b, g1, g2)`	if `b` is `true` then select `g1` otherwise `g2`
`cl_genint_add(g1, g2)`	addition
`cl_genint_sub(g1, g2)`	subtraction
`cl_genint_mult(g1, g2)`	multiplication
`cl_genint_div(g1, g2)`	division

- Relational operators:

Operator	Meaning
`cl_double_isnan(d)`	true if `d` is NaN
`cl_double_less(d1, d2)`	C-semantics of less comparison
`cl_double_less_equal(d1, d2)`	C-semantics of less-equal comparison
`cl_double_greater(d1, d2)`	C-semantics of greater comparison
`cl_double_greater_equal(d1, d2)`	C-semantics of greater-equal comparison
`cl_double_equal(d1, d2)`	C-semantics of equal comparison
`cl_double_not_equal(d1, d2)`	C-semantics of not-equal comparison
`cl_double_total_equal(d1, d2)`	C-semantics of an assignment
`cl_float_isnan(f)`	true if `f` is NaN
`cl_float_less(f1, f2)`	C-semantics of less comparison
`cl_float_less_equal(f1, f2)`	C-semantics of less-equa comparison
`cl_float_greater(f1, f2)`	C-semantics of greater comparison
`cl_float_greater_equal(f1, f2)`	C-semantics of greater-equal comparison
`cl_float_equal(f1, f2)`	C-semantics of equal comparison
`cl_float_not_equal(f1, f2)`	C-semantics of not-equal comparison
`cl_float_total_equal(f1, f2)`	C-semantics of an assignment
`cl_genint_less(g1, g2)`	less comparison
`cl_genint_less_equal(g1, g2)`	less-equal comparison
`cl_genint_greater(g1, g2)`	greater comparison
`cl_genint_greater_equal(g1, g2)`	greater-equal comparison
`cl_genint_equal(g1, g2)`	equal comparison
`cl_genint_not_equal(g1, g2)`	not-equal comparison
`cl_genint_total_equal(g1, g2)`	alias for `cl_genint_equal`

Note that the floating-point comparison operators mimic the behaviour of the comparison operators in C and do not distinguish between the signed zeros. In order to model an assignment use `cl_double_total_equal` and `cl_float_total_equal`. These operators are only true if both arguments represent the same value (the signed zeros are distinguished).

Most of the floating-point operations use round-to-nearest tie-to-even as rounding mode. Only the following four operations use round-to-zero: `cl_genint_scast_float()`, `cl_genint_ucast_float()`, `cl_genint_scast_double()` and `cl_genint_ucast_double()`. Furthermore, several aspects of casts from floating-point values to integer values are not specified in the C standard. Therefore, the following assumptions are made:

1. special floating-point values like the infinities and NaN are mapped to zero
2. if the integral part of the floating-point value is too large and does not fit into a `cl_genint` with n bits, then the n least significant bits of the integral part are taken

The generic operators listed below are provided as syntactic sugar. These operators analyze the types of their sub-expressions and insert casts when needed, e.g.

1. `cl_less(g, f)` with `g` being of type `cl_genint` and `f` being of type `cl_float` will be rewritten to `cl_float_less(cl_float_cast_genint(g), f)`,

2. `cl_add(d1, d2)` with the `cl_double` arguments `d1` and `d2` will be rewritten to `cl_double_add(d1, d2)`.

If one of the arguments is of type `cl_double`, all other arguments are casted to `cl_double` as well (if they have a different type). Similarly, if one of the arguments is of type `cl_float` (and there is no `cl_double` argument), all other arguments are casted to `cl_float`. This mimics the type propagation in C.

- Generic arithmetic operators:

Operator	Meaning
`cl_cast_to_double(dfg)`	cast sub-expression `dfg` to `cl_double`
`cl_cast_to_float(dfg)`	cast sub-expression `dfg` to `cl_float`
`cl_scast_to_genint(dfg, bw)`	cast sub-expression `dfg` to a signed integer with constant bitwidth `bw`
`cl_ucast_to_genint(dfg, bw)`	cast sub-expression `dfg` to an unsigned integer with constant bitwidth `bw`
`cl_abs(dfg)`	absolute value
`cl_minus(dfg)`	unary minus
`cl_ite(b, dfg1, dfg2)`	if `b` is `true` then select `dfg1` otherwise `dfg2`
`cl_add(dfg1, dfg2)`	addition
`cl_sub(dfg1, dfg2)`	subtraction
`cl_mult(dfg1, dfg2)`	multiplication
`cl_div(dfg1, dfg2)`	division

- Generic relational operators:

Operator	Meaning
`cl_isnan(dfg)`	true if argument is NaN
`cl_less(dfg1, dfg2)`	C-semantics of less comparison
`cl_less_equal(dfg1, dfg2)`	C-semantics of less-equal comparison
`cl_greater(dfg1, dfg2)`	C-semantics of greater comparison
`cl_greater_equal(dfg1, dfg2)`	C-semantics of greater-equal comparison
`cl_equal(dfg1, dfg2)`	C-semantics of equal comparison
`cl_not_equal(dfg1, dfg2)`	C-semantics of not-equal comparison
`cl_total_equal(dfg1, dfg2)`	C-semantics of an assignment

The following small example should illustrate the usage of the operators. Consider the following C-snippet:

```
float           f1, f2;
int             g1, g2;
unsigned short  g3, g4;

if (f1 < 0.1) f2 = f1;
g3 = g2 + (((int) f1) + g1);
g4 = g1 & g3;
```

The benchmark listed below encodes the example. Note that every cast (implicit or explicit) has to be encoded in the benchmark, e.g. an assignment in C between integer variables with different bitwidths or signedness contains an implicit cast which has to be encoded in the benchmark.

Please note that the interval-based solving strategy of iSAT3 is tailored for arithmetic operations (e.g. addition, subtraction, multiplication and division). If your benchmarks mainly consist of bitwise operations (e.g. not, and, or, xor), a bitblasting-based solver could be perhaps a better choice.

```
DECL
        cl_float [cl_float_neginf, cl_float_posinf] f1, f2;
        cl_genint [cl_sint32_min, cl_sint32_max] g1, g2;
        cl_genint [cl_uint16_min, cl_uint16_max] g3, g4;

EXPR
        -- for better readability some subexpressions are defined here
        define float01  = cl_float_constant(0x1.99999ap-4);
        define cast1    = cl_genint_scast_float(f1, cl_genint_constant(32));
        define add1     = cl_genint_add(cast1, g1);
        define cast2    = cl_genint_scast(add1, cl_genint_constant(32));
        define add2     = cl_genint_add(g2, cast2);
        define cast3    = cl_genint_ucast(add2, cl_genint_constant(16));
        define aand1    = cl_genint_saand(g1, g3, cl_genint_constant(32));
        define cast4    = cl_genint_ucast(aand1, cl_genint_constant(16));

        -- if (f1 < 0.1) f2 = f1;
        cl_float_less(f1, float01) -> cl_float_total_equal(f2, f1);

        -- g3 = g2 + (((int) f1) + g1);
        cl_genint_total_equal(g3, cast3);

        -- g4 = g1 & g3;
        cl_genint_total_equal(g4, cast4);
```

2.4 How iSAT3 works

To be able to interprete iSAT3's output and the tool options presented in the next section you need some basic understanding of how the tool works internally. For further details regarding the iSAT algorithm refer to [FHT+07] and regarding iSAT3 internals refer to [SKB13]. The iSAT3 solver 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, iSAT3 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 part 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 (non-conflicting) 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 $|b_c - b_d| \leq \delta_{\text{abs}}$, where b_c is the current lower (upper) bound of the respective variable and δ_{abs} is a parameter which we call *minimum progress*. The values of ε and δ can be set with the command-line options `--msw` and `--mpr`.

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

- If iSAT3 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 tool outputs result '`SATISFIABLE`', then the input formula is satisfiable.

  We note however that iSAT3 is in general not able to compute a definite answer for all input formulae. This is due to the fact that interval arithmetic combined with splitting (floating-point) intervals yields a highly incomplete deduction calculus. We are currently working on techniques to certify satisfiability in more cases.
- As mentioned before, iSAT3 cannot decide all given formulae due to the methods employed. To nevertheless provide termination with at least some quantitative result, the tool may stop with result `CANDIDATE SOLUTION`. This means that for the given parameters ε and δ_{abs} the solver could not detect any conflicts within the reported interval valuation. It does *not* mean, however, that the box actually contains a solution, but it can be seen as an *approximative* solution wrt. parameters ε and δ_{abs}. Actually, in most cases there *will* be a solution within the box or at least nearby. If you think that iSAT3 has reported a spurious solution you should re-run the solver with smaller ε and δ_{abs} in order to confirm (or to refute) the previous result.

2.5 Tool Options

The options influence the solver behavior and thereby its performance. Understanding the solving algorithm on the abstract level described in the previous section is important when using most of the options listed below.

2.5.1 General Options

These options are the ones you will most certainly want to experiment with when solving your models.

`--msw` Set the minimum splitting width, i.e. iSAT3 will not split intervals if their current width is below this threshold.

`--mpr` Set the absolute progress. Deduced consistent bounds that refine the valuation by less than this value are neglected.

`--extended-hys-syntax` Enable support for `exp2`, `exp10`, `log`, `log2` and `log10`. Without this option these strings are treated as variable names.

2.5.2 Print Options

Print options only affect the output of the solver, not the actual solving process.

`-v` increase verbosity level (without `-v` iSAT3 will only give a very brief output).

`--print-hex` all numbers are outputted in hexadecimal format. This might be interesting to get a bit-precise representation of floating-point numbers. You may also use such numbers in the input files given to iSAT3.

2.5.3 BMC-related Options

The BMC-related options refer only to input formulae in the bounded model checking format. They have no influence on the solver's behavior if the formula is given in the `EXPR` format.

`--start-depth` The first unwinding depth for which a BMC formula is generated and checked for satisfiability.

`--max-depth` The last unwinding depth for which a BMC formula is generated and checked for satisfiability.

3 The iSAT3 Library

The iSAT3 library provides a C-API (also usable in C++). The following code snippets will give an idea how the API should be used. Please look also at the example program `example.c` which is contained in the `isat3.tar.bz2` file. Please note that the library interface does not provide the new operators for accurate floating-point reasoning at the moment.

3.1 Creating a solver instance

Before using the library you have to call `isat3_setup()` once. Before your program exits, you should call `isat3_cleanup()` once. The function `isat3_init()` will return a new instance of iSAT3. An solver instance is needed to create a formula, solve it and asking for a solution – if there is any. A formula is created with the help of `isat3_nodes`. Every solver instance is independent of other already created instances. If you use multiple instances at the same time, be careful to not mix `isat3_nodes` from different instances. If an instance is not needed any more, use `isat3_deinit()` to destroy it. The following listing shows the basic steps of setting up the iSAT3 library and creating an iSAT3 instance.

```
#include <stdio.h>

#include "isat3.h"

void do_something(int parameter)
        {
        struct isat3 *is3 = isat3_init(NULL);

        /* ... create a formula and solve it ... */

        isat3_deinit(is3);
        }

int main(void)
        {
        isat3_setup();
        do_something(0);
        do_something(1);
        do_something(2);

        /* ... */

        isat3_cleanup();
        return (0);
        }
```

3.2 Creating a formula

A formula is created with the help of `isat3_nodes`. Starting with the variables the formula is build bottom-up. For example the constraint `x + y * z < 7;` can be created with these four steps:

1. create sub-formula for `(y * z)`, the associated `isat3_node` is `t1`
2. create sub-formula for `(x + t1)`, the associated `isat3_node` is `t2`
3. create an integer constant 7, the associated `isat3_node` is `c1`
4. create sub-formula for `(t2 < c1)`, the associated `isat3_node` is `t3`

If sub-formulas are no longer needed you may destroy the reference to them with `isat3_node_destroy()`.

```
struct isat3            *is3;
struct isat3_node       *b, *x, *y, *z;
struct isat3_node       *t1, *t2, *t3, *t4, *t5, *t6;
struct isat3_node       *c1, *c2, *c3;
struct isat3_node       *expr;

is3 = isat3_init(NULL);

/*
 * DECL
 * boole b;
 * int [-10,10] x,y,z;
 */

b = isat3_node_create_variable_boole(is3, "b");

x = isat3_node_create_variable_integer(is3, "x", -10, 10);
y = isat3_node_create_variable_integer(is3, "y", -10, 10);
z = isat3_node_create_variable_integer(is3, "z", -10, 10);

/*
 * EXPR
 * x + y * z < 7;
 * b or (x^2 = 4);
 */

t1 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_MUL, y, z);
t2 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_ADD, x, t1);
c1 = isat3_node_create_constant_integer(is3, 7);
t3 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_LESS, t2, c1);

c2 = isat3_node_create_constant_integer(is3, 2);
t4 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_POWER, x, c2);
c3 = isat3_node_create_constant_integer(is3, 4);
t5 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_EQUAL, t4, c3);
t6 = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_OR, b, t5);

expr = isat3_node_create_binary_operation(is3, ISAT3_NODE_BOP_AND, t3, t6);

/* destroy temporary nodes, not needed any more */

isat3_node_destroy(is3, t6);
isat3_node_destroy(is3, t5);
isat3_node_destroy(is3, t4);
isat3_node_destroy(is3, t3);
isat3_node_destroy(is3, t2);
isat3_node_destroy(is3, t1);
isat3_node_destroy(is3, c3);
isat3_node_destroy(is3, c2);
isat3_node_destroy(is3, c1);

/* ... solving ... */

isat3_deinit(is3);
```

The function `isat3_node_create_unary_operation()` creates unary operations of the following types:

Parameter	Operation
`ISAT3_NODE_UOP_PRIME`	to create a primed variable in `TRANS` (for example `x'`)

Parameter	Operation
ISAT3_NODE_UOP_NOT	boolean negation
ISAT3_NODE_UOP_ABS	absolute value
ISAT3_NODE_UOP_MINUS	unary minus
ISAT3_NODE_UOP_SIN	sine (unit: radian)
ISAT3_NODE_UOP_COS	cosine (unit: radian)
ISAT3_NODE_UOP_EXP	exponential function regarding base e
ISAT3_NODE_UOP_EXP2	exponential function regarding base 2
ISAT3_NODE_UOP_EXP10	exponential function regarding base 10
ISAT3_NODE_UOP_LOG	logarithmic function regarding base e
ISAT3_NODE_UOP_LOG2	logarithmic function regarding base 2
ISAT3_NODE_UOP_LOG10	logarithmic function regarding base 10

The function `isat3_node_create_binary_operation()` creates binary operations of the following types:

Parameter	Operation
ISAT3_NODE_BOP_AND	conjunction
ISAT3_NODE_BOP_NAND	negated and
ISAT3_NODE_BOP_OR	disjunction
ISAT3_NODE_BOP_NOR	negated or
ISAT3_NODE_BOP_XOR	exclusive or
ISAT3_NODE_BOP_XNOR	negated xor, i.e. equivalence
ISAT3_NODE_BOP_IMPLIES	implication
ISAT3_NODE_BOP_IFF	equivalence
ISAT3_NODE_BOP_LESS	`<`
ISAT3_NODE_BOP_LESS_EQUAL	`<=`
ISAT3_NODE_BOP_GREATER	`>`
ISAT3_NODE_BOP_GREATER_EQUAL	`>=`
ISAT3_NODE_BOP_EQUAL	`=`
ISAT3_NODE_BOP_NOT_EQUAL	`!=`
ISAT3_NODE_BOP_MIN	minimum
ISAT3_NODE_BOP_MAX	maximum
ISAT3_NODE_BOP_ADD	addition
ISAT3_NODE_BOP_SUB	subtraction
ISAT3_NODE_BOP_MULT	multiplication
ISAT3_NODE_BOP_POWER	nth power, n (2nd argument) has to be an integer $n \geq 0$
ISAT3_NODE_BOP_ROOT	nth root, n (2nd argument) has to be an integer $n \geq 1$

For easier creation of nodes with three (ternary) to nine (nonary) arguments, the functions `isat3_node_create_ternary_operation()`, ..., `isat3_node_create_nonary_operation()` may be used. The created nodes are of the following types:

Parameter	Operation
ISAT3_NODE_NOP_AND	conjunction
ISAT3_NODE_NOP_NAND	negated and
ISAT3_NODE_NOP_OR	disjunction
ISAT3_NODE_NOP_NOR	negated or
ISAT3_NODE_NOP_XOR	exclusive or
ISAT3_NODE_NOP_XNOR	negated xor, i.e. equivalence
ISAT3_NODE_NOP_ADD	addition
ISAT3_NODE_NOP_MULT	multiplication

Please consult `isat3.h` for the detailed function declarations.

3.3 Solving

The two different operation modes of the binary, namely (1) satisfiability checking of a single formula or (2) finding a trace of a hybrid system via bounded model checking (BMC) are also supported through the library interface.

3.3.1 Single Formula Mode

```
struct isat3            *is3;
struct isat3_node       *expr;
i3_type_t               result;
i3_type_t               result;
i3_truthval_t           truthval;
i3_bool_t               lb_is_strict, ub_is_strict;
i3_double_t             lb, ub;

is3 = isat3_init(NULL);

/* declare variables */

b = isat3_node_create_variable_boole(is3, "b");
x = isat3_node_create_variable_float(is3, "x", -100, 1000);

/* ... create expr ... */

/*
 * expr         is the node representing the formula to be solved
 *              corresponding to EXPR in a .hys files
 * timeout      in micro-seconds
 * result       could be ISAT3_RESULT_UNKNOWN, ISAT3_RESULT_UNSAT,
 *              ISAT3_RESULT_SAT, ...
 */

result = isat3_solve_expr(is3, expr, timeout);

/* get solution or candidate solution */

if ((isat3_result_contains_possible_solution(result)) ||
        (isat3_result_contains_solution(result)))
        {
        truthval = isat3_get_truth_value(is3, b, 0);

        /*
         * truthval could be I3_TRUTHVAL_FALSE, I3_TRUTHVAL_TRUE or
         * I3_TRUTHVAL_UNDEF
         */

        lb_is_strict = isat3_is_lower_bound_strict(is3, x, 0);
        ub_is_strict = isat3_is_upper_bound_strict(is3, x, 0);
        lb = isat3_get_lower_bound(is3, x, 0);
        ub = isat3_get_upper_bound(is3, x, 0);

        /* print the result for x */

        printf("%s:␣%s%1.40f,␣%1.40f%s\n",
                isat3_node_get_variable_name(is3, x),
                lb_is_strict ? "(" : "[",
                lb,
                ub,
                ub_is_strict ? ")" : "]");
        }

/* ... */

isat3_deinit(is3);
```

3.3.2 Bounded Model Checking Mode

```
struct isat3            *is3;
struct isat3_node       *b, *x;
struct isat3_node       *expr;
i3_truthval_t           truthval;
i3_bool_t               lb_is_strict, ub_is_strict;
i3_double_t             lb, ub;
i3_tframe_t             t, tframe;

is3 = isat3_init(NULL);

/* declare variables */

b = isat3_node_create_variable_boole(is3, "b");
x = isat3_node_create_variable_float(is3, "x", -100, 1000);

/* ... create init, trans, target ... */

/*
 * init, trans, target  nodes representing the sub-formulas of INIT, TRANS,
 *                      TARGET
 * start_tframe         before solving unroll up to this time frame
 * max_tframe           unroll not more than this number of time frames
 * timeout              in micro-seconds
 * result               could be ISAT3_RESULT_UNKNOWN, ISAT3_RESULT_UNSAT,
 *                      ISAT3_RESULT_SAT, ...
 */

result = isat3_solve_bmc(is3, init, trans, target, start_tframe, end_tframe, timeout);

/* get solution or candidate solution */

if ((isat3_result_contains_possible_solution(result)) ||
        (isat3_result_contains_solution(result)))
        {
        tframe = isat3_get_tframe(is3);
        for (t = 0; t <= tframe; t++)
                {
                truthval = isat3_get_truth_value(is3, b, t);

                /*
                 * truthval could be I3_TRUTHVAL_FALSE, I3_TRUTHVAL_TRUE or
                 * I3_TRUTHVAL_UNDEF
                 */

                lb_is_strict = isat3_is_lower_bound_strict(is3, x, t);
                ub_is_strict = isat3_is_upper_bound_strict(is3, x, t);
                lb = isat3_get_lower_bound(is3, x, t);
                ub = isat3_get_upper_bound(is3, x, t);

                /* print the result for x */

                printf("%s@%d: %s%1.40f, %1.40f%s\n",
                        isat3_node_get_variable_name(is3, x),
                        tframe,
                        lb_is_strict ? "(" : "[",
                        lb,
                        ub,
                        ub_is_strict ? ")" : "]");
                }
        }

/* ... */

isat3_deinit(is3);
```

3.3.3 Incremental Solving

The functions `isat3_solve_expr()` and `isat3_solve_bmc()` start the solver every time from scratch. If you want to solve a set of formulas with common sub-formulas, it is more beneficial to use incremental solving. You create constraints as before. With the function `isat3_add_constraint()` you add the them to the set of constraints to be solved with `isat3_solve_constraints()`. Every call to `isat3_solve_constraints()` will re-use conflict-clauses learnt during previous calls.

Additionally *backtrack-points* are supported. Imagine the set of constraints as a stack. New constraints are added to the top. A backtrack-point is a marker in this stack. Going back to a backtrack-point removes (pops) the constraints above this backtrack-point. The function `isat3_push_btpoint()` will set a backtrack-point – in other words it will set a marker in the stack of the constraints. The function `isat3_pop_btpoint()` will remove all constraints until the top-most backtrack-point. Please consult `example.c` for an example illustrating the usage of the incremental interface.

References

[EKKT08] Andreas Eggers, Natalia Kalinnik, Stefan Kupferschmid, and Tino Teige. Challenges in constraint-based analysis of hybrid systems. In Angelo Oddi, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 5655 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2008.

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

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *Journal on Satisfiability, Boolean Modeling, and Computation*, 1(2007):209–236, 2007.

[HEFT08] Christian Herde, Andreas Eggers, Martin Fränzle, and Tino Teige. Analysis of hybrid systems using hysat. In *ICONS*, pages 196–201. IEEE Computer Society, 2008.

[SKB13] Karsten Scheibler, Stefan Kupferschmid, and Bernd Becker. Recent improvements in the smt solver isat. In Christian Haubelt and Dirk Timmermann, editors, *MBMV*, pages 231–241. Institut für Angewandte Mikroelektronik und Datentechnik, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2013.