PySMT
A Python Interface to Satisfiability Modulo Theories Solvers*

Version 0.5

Tommi Junttila
Helsinki University of Technology
Laboratory for Theoretical Computer Science
http://www.tcs.hut.fi/~tjunttil

December 17, 2007

1 Introduction

This document describes an experimental Python programming language interface to the Satisfiability Modulo Theories (SMT) approach (see e.g. [SMTLIB 2007]). Such an interface will, I hope, enable people who are not too familiar with first-order logic to successfully apply modern SMT solvers to solve problems in their own application areas. The interface has been developed as a part of the SMUML project [1]. In that project, the purpose of the interface has been to facilitate prototyping of SMT-based Bounded Model Checking (BMC) encodings of UML state machines as well as some other related tasks, such as checking whether the guards of two transitions can be true at the same time.

Note that efficiency has not been the main concern when developing the interface but applicability for fast prototyping and support for multiple SMT solvers. Therefore, if the efficiency of the interface becomes a bottleneck at some point, consider (i) fixing a specific, efficient SMT solver to be used, (ii) interfacing it with a programming language more efficient than Python, and, (iii) most importantly, using the SMT solver in an incremental mode if such is provided and your problem is incremental by its nature.

In the current version the following features are supported.

- Boolean variables and the common Boolean operators like "and", "or", "not", "if-then-else", and "if-and-only-if".
- Arbitrary precision integer variables and arithmetic operators for addition, multiplication (with a constant), subtraction, negation, and comparison of integers.

*This work has been done as a part of the SMUML project funded by the Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, Conformiq, and Mipro. The author also thanks Jori Dubrovin for his feedback during the development of the PySMT tool.

[1] http://www.tcs.hut.fi/Research/Logic/SMUML.shtml

- Bounded integer variables (both signed and unsigned) and arithmetic operators for addition, multiplication, subtraction, negation, and comparison.
- Finite enumeration data types.
- Definition and application of uninterpreted functions (UIFs).
- Array data types.
- Translating the constructed problems into the SMT-LIB file format [SMTLIB 2007] (only a very limited subset supported at the moment) or to the native input formats of the state-of-the-art SMT solvers CVC3 [CVC3 2007], MathSAT [Bruttomesso et al. 2007; MathSAT 2007], STP [Ganesh and Dill 2006; Ganesh 2007], and Yices [Dutertre and de Moura 2006; Yices 2007]. The supported subset of data types and operators depends on the applied solver.
- The state-of-the-art SMT solvers CVC3 [CVC3 2007], MathSAT [Bruttomesso et al. 2007; MathSAT 2007], STP [Ganesh and Dill 2006; Ganesh 2007], and Yices [Dutertre and de Moura 2006; Yices 2007] can be called to solve the constructed problems and the solution can be queried by using the Python interface.

Notice that not all the logics described in the SMT-LIB format [SMTLIB 2007] are covered in the proposed circuit format and Python API at the moment. In fact, some logics are not covered at all, e.g. those with quantifiers. In addition, please note that SMT solvers are not restricted to the theories, logics, or syntax described in the SMT-LIB but can provide their own input formats with constructs not found in SMT-LIB. For instance, some solvers support so-called lambda functions not found in the SMT-LIB format. In addition, problems mixing some theories cannot always be described in the current version of the SMT-LIB format (e.g. unbounded and bounded integers cannot be used in the same problem) but

2 An Example

Suppose that we want to check whether the two programming language conditions, `x > y` and `x+1 < y`, can both evaluate to true for some values of x and y, where x are y are 32-bit signed integers. An equivalent, more mathematically expressed formulation of the problem is: assuming that (i) x and y are 32-bit signed integers, and (ii) $+_{32}, <_{32}, >_{32}, 1_{32}$ are the standard 32-bit signed addition, less than comparison, greater than comparison, and constant one, respectively, is the formula

$$((x +_{32} 1_{32}) <_{32} y) \wedge (x >_{32} y)$$

satisfiable, i.e., are there values in the range $\{-2^{31}, \ldots, -1, 0, 1, \ldots, 2^{31} - 1\}$ for x and y such that the formula evaluates to true? In fact, the above formula is satisfiable when $x = 2147483647 = 2^{31} - 1$ and $-2^{31} < y < 2^{31} - 1$ because $x +_{32} 1_{32} = 2147483647 +_{32} 1_{32} = -2147483648 = -2^{31}$ in 32-bit signed arithmetics.

In the SMT-LIB format we can express the above problem as:

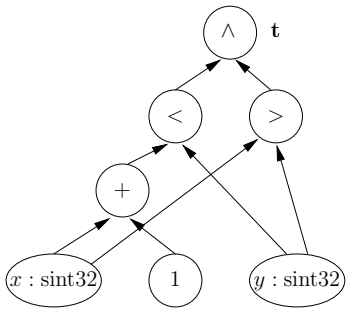

Figure 1: The constrained circuit corresponding to the formula $((x +_{32} 1_{32}) <_{32} y) \wedge (x >_{32} y)$.

```
(benchmark problem
 :logic QF_UFBV[32]
 :extrafuns ((x BitVec[32]))
 :extrafuns ((y BitVec[32]))
 :formula (and (bvsgt x y) (bvslt (bvadd (extract[31:0] bv1) x) y))
)
```

If we invoke Yices on this problem, it prints:

```
sat
(= x 0b01111111111111111111111111111111)
(= y 0b00111000110011000101100011110111)
```

Similarly, STP prints:

```
Invalid.
ASSERT( x  = 0hex7FFFFFFF  );
ASSERT( y  = 0hex80000003  );
```

The Python interface described in the next section provides an API for building problem instances, outputting them in the SMT-LIB format or in the native input format of an SMT solver, invoking an SMT solver on the instance, and parsing the result back so that it can be queried through the Python API. As an example, the above problem can be solved by the following Python program.

```
import PySMT
solver = PySMT.Yices("~/Local/Progs/yices-1.0.9/bin/yices")
WIDTH=32
c = PySMT.Circuit()
x = c.add_sint_var(WIDTH)
y = c.add_sint_var(WIDTH)
eqn1 = c.add_lt(c.add_plus(x, c.add_sint_const(WIDTH,1)), y)
eqn2 = c.add_gt(x, y)
problem = c.add_and(eqn1, eqn2)
c.constrain(problem, True)
result = c.solve(solver)
print result
if result == 'sat':
```

```
  print "x = "+PySMT.sint2hex(WIDTH, c.get_value(x))
  print "y = "+PySMT.sint2hex(WIDTH, c.get_value(y))
  print "eqn1 = "+str(c.get_value(eqn1))
```

When executed, it outputs:

```
sat
x = 0h7fffffff
y = 0h377c4ab9
eqn1 = True
```

Note that it is possible to query the solution values of *all* subformulae (such as `eqn1` in the example) in the problem through the Python interface, not only the values of the variables.

Internally, the formulae are presented as parse trees whose all common subformulae are shared, i.e. as extended Boolean circuits. Figure 1 shows the circuit for the above formula with **t** next to the top gate constraining that gate to always evaluate to true.

3 The Python Interface

The Python language API is as follows.[2]

```
Help on module PySMT:

NAME
    PySMT - A circuit based Python front-end for some SMT solvers and the SMT-LIB format.

FILE
    /lhome/tjunttil/Priv/SMUML/PySMT.py

DESCRIPTION
    Author: Tommi Junttila

    Copyright (c) 2007 Helsinki University of Technology
    Copyright (c) 2008 Tommi Junttila
    Released under the GNU General Public License version 2

CLASSES
    Circuit
    Gate
    Solver
        CVC3
        MathSAT
        STP
        Yices
        Z3

    class CVC3(Solver)
     |  A front-end for CVC3.
     |
     |  Methods defined here:
     |
     |  __init__(self, solver_binary='./cvc3')
     |      Create a front-end for CVC3.
     |      'solver_binary' is the file name for a CVC3 executable binary.
```

[2] Produced by the Python command `help(PySMT)`.

```
class Circuit
 |  An extended Boolean circuit.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      Create an empty circuit.
 |
 |  add_and(self, *children)
 |      Add and return a new boolean and-gate over the children.
 |      The gate evaluates to true iff all child gates evaluate to true.
 |      If no children are given, return the boolean true constant gate.
 |      If the children list is empty, the gate is equivalent to true.
 |
 |  add_array_read(self, array, index)
 |      Add and return a gate computing the value 'array'['index'],
 |      where 'array' is a gate of array type and 'index' is a gate of
 |      the index type of the array type.
 |
 |  add_array_var(self, t)
 |      Add and return a variable of an array type 't'.
 |
 |  add_array_write(self, array, index, value)
 |      Add and return a new array gate equal to 'array' except that
 |      'array'['index'] = 'value',
 |      where 'array' is a gate of array type,
 |      'index' is a gate of the index type of the array type, and
 |      'value' is a gate of the range type of the array type.
 |
 |  add_bitwise_and(self, child1, child2)
 |      Add and return a new gate for the bitwise and of child1 and child2.
 |      The types of the children should be bounded integers of same width.
 |
 |  add_bitwise_not(self, child)
 |      Add and return a new gate for the bitwise not of child.
 |      The type of the child should be a bounded integer.
 |
 |  add_bitwise_or(self, child1, child2)
 |      Add and return a new gate for the bitwise or of child1 and child2.
 |      The types of the children should be bounded integers of same width.
 |
 |  add_bitwise_xor(self, child1, child2)
 |      Add and return a new gate for the bitwise xor of child1 and child2.
 |      The types of the children should be bounded integers of same width.
 |
 |  add_bool_var(self)
 |      Add and return a new boolean variable.
 |
 |  add_enum_const(self, t, c)
 |      Add and return a new constant 'c' of enumeration type 't'.
 |      'c' should be a literal in the domain of 't'.
 |
 |  add_enum_var(self, t)
 |      Add and return a variable of enumeration type 't'.
 |
 |  add_eq(self, child1, child2)
 |      Add and return a new equality gate.
 |      The types of the two children should match and be either Boolean,
 |      enumeration, integer, rational, or bounded integers of same width.
 |      The gate evaluates to true iff the children evaluate to the same
 |      value.
```

```
|
|  add_false(self)
|      Get the boolean false constant gate.
|
|  add_ge(self, child1, child2)
|      Add and return a new greater-or-equal gate.
|      The types of the two children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_gt(self, child1, child2)
|      Add and return a new greater-than gate.
|      The types of the two children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_iff(self, child1, child2)
|      Add and return a new boolean iff-gate over the boolean child gates.
|      The gate evaluates to true iff the child gates evaluate to
|      the same value.
|
|  add_implies(self, child1, child2)
|      Add and return a new boolean implies-gate over the boolean child gates.
|      The gate evaluates to true iff (a) child1 evaluates to false or (b)
|      child2 evaluates to true.
|
|  add_int_const(self, c)
|      Add and return a new unbounded integer constant c.
|
|  add_int_var(self)
|      Add and return a new unbounded integer variable.
|
|  add_ite(self, if_child, then_child, else_child)
|      Add and return a new if-then-else-gate
|      if-then-else(if_child, then_child, else_child), where
|      (i) if_child must be Boolean, and
|      (ii) then_child and else_child are of same type, the type being
|           boolean, enumeration, integer, rational, array,
|           bounded signed integer, or bounded unsigned integer.
|      The type of the gate is the same as the type of the then and else
|      childs.
|      The gate evaluates to
|      (i) the value of then_child, if if_child is true, and
|      (ii) the value of else_child, if_child is false.
|
|  add_le(self, child1, child2)
|      Add and return a new less-or-equal gate.
|      The types of the two children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_lt(self, child1, child2)
|      Add and return a new less-than gate.
|      The types of the two children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_minus(self, child1, child2)
|      Add and return a new gate for the subtraction child1 - child2.
|      The types of the children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_neg(self, child)
|      Add and return a new gate for the negation of child.
|      The type of child must be either integer, rational, or bounded integer.
|
```

```
|  add_neq(self, child1, child2)
|      Add and return a new negated equality gate.
|      The types of the two children should match and be either Boolean,
|      enumeration, integer, rational, or bounded integers of same width.
|      The gate evaluates to true iff the children evaluate to the different
|      values.
|
|  add_not(self, child)
|      Add and return a new boolean not-gate over the child.
|
|  add_or(self, *children)
|      Add and return a new boolean or-gate over the children.
|      The gate evaluates to true iff at least one of the child gates
|      evaluates to true.
|      If no children are given, return the boolean false constant gate.
|      If the children list is empty, the gate is equivalent to false.
|
|  add_plus(self, *children)
|      Add and return a new addition gate over the children.
|      The types of the children should match and be either integer,
|      rational, or bounded integers of same width.
|
|  add_sint_const(self, w, c)
|      Add and return a new bounded signed integer constant c of width w,
|      1 <= w <= 32 and -2^{w-1} <= c <= 2^{w-1}-1
|
|  add_sint_var(self, w)
|      Add and return a new bounded signed integer variable of width w.
|      Required: 1 <= w <= 32.
|
|  add_subcircuit(self, subcircuit, input_map, tracked_gates)
|      Instantiate the circuit 'subcircuit' into this circuit.
|      'subcircuit' may not equal to 'self'.
|      'input_map' should be a dictionary from (a subset of) the variable
|      gates of 'subcircuit' to the gates already in this circuit.
|      If a variable 'v' gate is a key in 'input_map', then it is
|      instantiated as 'input_map[v]', otherwise it is instantiated
|      as a new variable.
|      'tracked_gates' should be a list of gates in 'subcircuit';
|      this method returns a dictionary from them to
|      the corresponding gates instantiated into this circuit.
|
|  add_times(self, *children)
|      Add and return a new multiplication gate over the children.
|      The types of the children should match and be either (i) integer,
|      (ii) rational, or (iii) bounded integers of same width.
|
|  add_true(self)
|      Get the boolean true constant gate.
|
|  add_uif(self, uif, *children)
|      Add and return a new uninterpreted function application gate
|      over the children.
|      The types of the children must match the argument types of the
|      uninterpreted function 'uif'.
|      The type of the returned gate is the range type of 'uif',
|      and its value is uif(value(child1),...,value(childn)).
|
|  add_uint_const(self, w, c)
|      Add and return a new bounded unsigned integer constant c of width w.
|      Required: 0 <= w and 0 <= c < 2^w.
|
```

```
|  add_uint_var(self, w)
|      Add and return a new bounded unsigned integer variable of width w.
|      Required: 0 <= w.
|
|  add_var(self, t=None)
|      Add and return a new variable of type 't'.
|      If 't' is None, a Boolean variable is returned.
|
|  add_xor(self, child1, child2)
|      Add and return a new boolean xor-gate over the boolean child gates.
|
|  constrain(self, gate, value)
|      Constrain the value of a Boolean gate to False or True.
|
|  get_nof_gates(self)
|      Get the number of gates in the circuit.
|
|  get_nof_relevant_gates(self)
|      Get the number of gates in the circuit that are constrained
|      gates or descendants of constrained gates.
|
|  get_type(self, gate)
|      Return the type of the gate 'gate'.
|
|  get_value(self, gate)
|      Return the value of the gate 'gate' in the latest solution.
|      Thus the previous solve_* API call to this circuit must have
|      returned 'sat'.
|      The solution is invalidated when any add_* or constrain_* function
|      is called.
|
|  is_constant_false(self, gate)
|      Returns True iff 'gate' corresponds to the Boolean constant False
|      and the circuit is not inconsistent.
|
|  is_constant_true(self, gate)
|      Returns True iff 'gate' corresponds to the Boolean constant True
|      and the circuit is not inconsistent.
|
|  is_inconsistent(self)
|      Returns True if it has been detected that the circuit is
|      permanently inconsistent.
|
|  solve(self, solver, mode='native', produce_model=True)
|      Solve whether the circuit is satisfiable by using the Solver 'solver'.
|      The 'mode' should be either 'native' or 'smtlib'.
|      If 'mode' is 'native', use the native input format of 'solver',
|      otherwise the SMT-LIB format is used.
|
|      Returns either 'error', 'unsat', or 'sat'.
|
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_cvc3(self, cvc3_binary='./cvc3', produce_model=True)
|      Solve the circuit by using CVC3 and its native input format.
|      Returns either 'error', 'unsat', or 'sat'.
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_mathsat(self, mathsat_binary='./mathsat_model', produce_model=True)
|      Solve the circuit by using MathSAT and its native input format.
```

```
|      Returns either 'error', 'unsat', or 'sat'.
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_native(self, solver, produce_model=True)
|      Solve whether the circuit is satisfiable by using the Solver 'solver'.
|      with its native input format.
|
|      Returns either 'error', 'unsat', or 'sat'.
|
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_smtlib(self, solver)
|      Solve the circuit by using Solver solver
|      with the SMT-LIB input format.
|
|      Returns either 'error', 'unsat', or 'sat'.
|
|      If 'sat' is returned, the satisfying solution can be queried by
|      calling the get_value function.
|
|  solve_stp(self, stp_binary='./stp', produce_model=True)
|      Solve the circuit by using STP and its native input format.
|      Returns either 'error', 'unsat', or 'sat'.
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_yices(self, yices_binary='./yices', produce_model=True)
|      Solve the circuit by using Yices and its native input format.
|      Returns either 'error', 'unsat', or 'sat'.
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  solve_z3(self, z3_binary='./z3', produce_model=True)
|      Solve the circuit by using Z3 and its native input format.
|      Returns either 'error', 'unsat', or 'sat'.
|      If 'produce_model' is True and 'sat' is returned,
|      the satisfying model can be queried by calling the get_value function.
|
|  to_cvc3(self, f, produce_model)
|      Output the circuit in CVC3 format to the file object 'f'.
|      If 'produce_model' is True, then the 'COUNTERMODEL' command
|      is inserted in the generated CVC3 problem file.
|
|  to_dot(self, f, only_relevant=False)
|      Print the circuit in the graphviz dotty format to
|      the file object f.
|      If 'only_relevant' is True, only the constrained gates and
|      their descendants are printed.
|
|  to_mathsat(self, f)
|      Output the circuit in MathSAT format to the file object 'f'.
|
|  to_smtlib(self, f)
|      Output the circuit as an SMT-LIB format problem to
|      the file object f.
|
|  to_stp(self, f, produce_model=True)
|      Output the circuit in STP format to the file object 'f'.
|      If 'produce_model' is True, then the 'COUNTEREXAMPLE' command
|      is inserted in the generated STP problem file.
```

```
 |
 |  to_yices(self, f, produce_model=True)
 |      Output the circuit in Yices format to the file object f.
 |      If 'produce_model' is True, then the 'set-evidence!' flag is set
 |      to true in the generated Yices problem file.
 |
 |  to_z3(self, f)
 |      Output the circuit in Z3 native format to the file object f.

class Gate
 |  Just a handle front-end, the actual gates are not visible to the user.
 |  Use Circuit.add_* functions to create gates.
 |
 |  Methods defined here:
 |
 |  __del__(self)
 |
 |  __str__(self)
 |
 |  get_type(self)
 |      Return the type of the gate.

class MathSAT(Solver)
 |  A front-end for MathSAT.
 |
 |  Methods defined here:
 |
 |  __init__(self, solver_binary='./mathsat')
 |      Create a front-end for MathSAT.
 |      'solver_binary' is the file name for a MathSAT executable binary.

class STP(Solver)
 |  A front-end for STP.
 |
 |  Methods defined here:
 |
 |  __init__(self, solver_binary='./stp')
 |      Create a front-end for STP.
 |      'solver_binary' is the file name for an STP executable binary.

class Solver
 |  An abstract base class for SMT solver front-ends.

class Yices(Solver)
 |  A front-end for Yices.
 |
 |  Methods defined here:
 |
 |  __init__(self, solver_binary='./yices')
 |      Create a front-end for Yices.
 |      'solver_binary' is the file name of the Yices executable binary.

class Z3(Solver)
 |  A front-end for Z3.
 |
 |  Methods defined here:
 |
 |  __init__(self, solver_binary='./z3')
 |      Create a front-end for Z3.
 |      'solver_binary' is the file name for a Z3 executable binary.

FUNCTIONS
```

```
bin2sint(s)
    Converts a non-empty string of 0's and 1's in two's complement notation
    to a long integer, the first bit is interpreted as a sign bit.

bin2uint(s)
    Converts a non-empty string of 0's and 1's to a long integer.

get_bool()
    Return the boolean type.

get_int()
    Return the integer type.

get_rat()
    Return the rational type.

get_sint(W)
    Return the W-bits bounded signed integer type.
    W must be a positive integer.

get_uint(W)
    Return the W-bits bounded unsigned integer type.
    W must be a positive integer.

new_array(index, range)
    Introduce a new array type from 'index' type to the 'range' type.
    Note that if you call this function two times with the same index and
    range types, you will get the same array type.

new_enum(literals, name=None)
    Introduce a new enumeration primitive type.
    'literals' should be a non-empty list or set of disjoint names,
    name being a string that matches the regular expression
    [a-zA-Z][a-zA-Z0-9]*.
    Note that if you call this function two times with the same list
    of literals, you will get two distinct types, not the same one.

    The name of the type, 'name', has no semantic meaning but
    is only used for verbose output and debugging purposes.

new_uif(domain, range, name=None)
    Introduce a new uninterpreted function (UIF) with signature
    D1 * ... * Dn -> 'range',
    where
    'domain' is a non-empty list [D1,...,Dn] of domain types, and
    'range'  is the range type.
    Note that if you call this function two times with the same domain and
    range types, you will get two distinct UIFs, not the same one.

    The name of the UIF, 'name', has no semantic meaning but
    is only used for verbose output and debugging purposes.

sint2bin(width, n, prefix='0b')
    Converts a signed, 'width'-bit integer 'n' (given as int or long)
    to binary string.
    Required: width >= 1
    Required: -2^{width-1} <= n < 2^{width-1}

sint2hex(width, n, prefix='0h')
    Converts a signed, 'width'-bit integer 'n' (given as int or long)
    to hexadecimal string.
    Required: width >= 1 and is a multiple of four
```

```
    Required: -2^{width-1} <= n < 2^{width-1}

uint2bin(width, n, prefix='0b')
    Converts an unsigned, 'width'-bit integer 'n' (given as int or long)
    to binary string.
    Required: width >= 1
    Required: 0 <= n < 2^{width}

uint2hex(width, n, prefix='0h')
    Converts an unsigned, 'width'-bit integer 'n' (given as int or long)
    to hexadecimal string.
    Required: width >= 1 and is a multiple of four
    Required: -2^{width-1} <= n < 2^{width-1}
```

References

Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., and Sebastiani, R. 2007. System description: MathSAT 4. http://www.smtcomp.org/descriptions/mathsat_system_description.pdf. SMT-COMP 2007 system description paper.

CVC3 2007. CVC3 page. http://www.cs.nyu.edu/acsys/cvc3/.

Dutertre, B. and de Moura, L. 2006. System description: Yices 1.0. http://www.csl.sri.com/users/demoura/smt-comp/descriptions/yices-smtcom%p06.pdf. SMT-COMP 2006 system description paper.

Ganesh, V. 2007. STP — a decision procedure for bitvectors and arrays. http://cag.csail.mit.edu/~vganesh/stp.html.

Ganesh, V. and Dill, D. L. 2006. System description of STP. http://www.csl.sri.com/users/demoura/smt-comp/descriptions/stp.ps. SMT-COMP 2006 system description paper.

MathSAT 2007. Mathsat web page. http://mathsat.itc.it/.

SMTLIB 2007. SMT-LIB — the satisfiability modulo theories library. http://combination.cs.uiowa.edu/smtlib/.

Yices 2007. Yices: An SMT solver. http://yices.csl.sri.com/.