pySMT Documentation

Release 0.6.2

Andrea Micheli and Marco Gario

January 03, 2017

Contents

CHAPTER 1

Introduction

pySMT makes working with Satisfiability Modulo Theory simple. pySMT provides an intermediate step between the SMT-LIB (that is universal but not programmable) and solvers API (that are programmable, but specific).

Among others, you can:

- Define formulae in a solver independent way
- Run the same code using multiple solvers
- Easily perform quantifier elimination, interpolation and unsat-core extraction
- Write ad-hoc simplifiers and operators
- and more...

Please let us know of any problem or possible improvements by opening an issue on github or by writing to pysmt@googlegroups.com .

Where to go from here:

- Getting Started: *Installation* and *Hello World*;
- Tutorials: Simple examples showing how to perform common operations using pySMT.
- Full *API Reference*
- *Extending pySMT*

CHAPTER 2

Table of Contents

2.1 Getting Started

In this section we will see how to install pySMT and how to solve a simple problem using it.

2.1.1 Installation

To run pySMT you need Python 3.5+ or Python 2.7 installed. If no solver is installed, pySMT can still create and dump the SMT problem in SMT-LIB format. pySMT works with any SMT-LIB compatible solver. Moreover, pySMT can leverage the API of the following solvers:

- MathSAT (http://mathsat.fbk.eu/)
- Z3 (https://github.com/Z3Prover/z3/)
- CVC4 (http://cvc4.cs.nyu.edu/web/)
- Yices 2 (http://yices.csl.sri.com/)
- CUDD (http://vlsi.colorado.edu/~fabio/CUDD/)
- PicoSAT (http://fmv.jku.at/picosat/)
- Boolector (http://fmv.jku.at/boolector/)

The Python binding for the SMT Solvers must be installed and accessible from your `PYTHONPATH`.

To check which solvers are visible to pySMT, you can use the command `pysmt-install` (simply `install.py` in the sources):

```
$ pysmt-install --check
```

provides the list of installed solvers (and version). Solvers can be installed with the same script: e.g.,

```
$ pysmt-install --msat
```

installs the MathSAT5 solver. Once the installation is complete, you can use the option `--env` to obtain a string to update your `PYTHONPATH`:

```
$ pysmt-install --env
export PYTHONPATH="/home/pysmt/.smt_solvers/python-bindings-2.7:${PYTHONPATH}"
```

By default the solvers are installed in your home directory in the folder `.smt_solvers`. `pysmt-install` has many options to customize its behavior.

GMPY2

PySMT supports the use of the gmpy2 library (version 2.0.8 or later) to handle multi-precision numbers. This provides an efficient way to perform operations on big numbers, especially when fractions are involved. The gmpy library is used by default if it is installed, otherwise the `fractions` module from Python's standard library is used. The use of the gmpy library can be controlled by setting the system environment variables `PYSMT_GMPY2` to `True` or `False`. If this is set to true and the gmpy library cannot be imported, an exception will be thrown.

2.1.2 Hello World

Any decent tutorial starts with a *Hello World* example. We will encode a problem as an SMT problem, and then invoke a solver to solve it. After digesting this example, you will be able to perform the most common operations using pySMT.

The problem that we are going to solve is the following:

> Lets consider the letters composing the words *HELLO* and *WORLD*, with a possible integer value between 1 and 10 to each of them.
>
> Is there a value for each letter so that H+E+L+L+O = W+O+R+L+D = 25?

The module *pysmt.shortcuts* provides the most used functions of the library. These are simple wrappers around functionalities provided by other objects, therefore, this represents a good starting point if you are interested in learning more about pySMT.

We include the methods to create a new *Symbol()* (i.e., variables), and typing information (the domain of the variable), that is defined in *pysmt.typing*, and we can write the following:

```
from pysmt.shortcuts import Symbol
from pysmt.typing import INT

h = Symbol("H", INT)

domain = (1 <= h) & (10 >= h)
```

When importing *pysmt.shortcuts*, the infix notation is enabled by default. Infix notation makes it very easy to experiment with pySMT expressions (e.g., on the shell), however, it tends to make complex code less clear, since it blurs the line between Python operators and SMT operators. In the rest of the example, we will use the textual operators by importing them from *pysmt.shortcuts*.

```
from pysmt.shortcuts import Symbol, LE, GE, And, Int
from pysmt.typing import INT

h = Symbol("H", INT)

# domain = (1 <= h) & (10 >= h)
domain = And(LE(Int(1), h), GE(Int(10), h))
```

Instead of defining one variable at the time, we will use Python's comprehension to apply the same operation to multiple symbols. Comprehensions are so common in pySMT that n-ary operators (such as *And()*, *Or()*, *Plus()*) can accept am iterable object (e.g. lists or generator).

```
from pysmt.shortcuts import Symbol, LE, GE, Int, And, Equals, Plus, Solver
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]
```

```
letters = set(hello+world)

domains = And(And(LE(Int(1), l),
                  GE(Int(10), l)) for l in letters)

sum_hello = Plus(hello)
sum_world = Plus(world)

problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(36)))

formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)
```

Note: Limited serialization

By default printing of a string is limited in depth. For big formulas, you will see something like `(x & (y | ...))`, where deep levels of nestings are replaced with the ellipses `...`. This generally provides you with an idea of how the structure of the formula looks like, without flooding the output when formulas are huge. If you want to print the whole formula, you need to call the `serialize()` method.

Defaults methods for formula allow for simple printing. Checking satisfiability can also be done with a one-liner.

```
print("Checking Satisfiability:")
print(is_sat(formula))
```

Model extraction is provided by the *get_model()* shortcut: if the formula is unsatisfiable, it will return `None`, otherwise a *Model*, that is a dict-like structure mapping each Symbol to its value.

```
print("Serialization of the formula:")
print(formula)

print("Checking Satisfiability:")
print(is_sat(formula))
```

Shortcuts are very useful for one off operations. In many cases, however, you want to create an instance of a *Solver* and operate on it incrementally. This can be done using the *pysmt.shortcuts.Solver()* factory. This factory can be used within a context (`with` statement) to automatically handle destruction of the solver and associated resources. After creating the solver, we can assert parts of the formula and check their satisfiability. In the following snippet, we first check that the `domain` formula is satisfiable, and if so, we continue to solve the problem.

```
with Solver(name="z3") as solver:
    solver.add_assertion(domains)
    if not solver.solve():
        print("Domain is not SAT!!!")
        exit()
    solver.add_assertion(problem)
    if solver.solve():
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

In the example, we access the value of each symbol (*get_value()*), however, we can also obtain a model object using *get_model()*.

Note: Incrementality and Model Construction

Many solvers can perform aggressive simplifications if incrementality or model construction are not required. Therefore, if you do not need incrementality and model construction, it is better to call *is_sat()*, rather than instantiating a solver. Similarly, if you need only one model, you should use *get_model()*

With pySMT it is possible to run the same code by using different solvers. In our example, we can specify which solver we want to run by changing the way we instantiate it. If any other solver is installed, you can try it by simply changing `name="z3"` to its codename (e.g., `msat`):

Solver	pySMT name
MathSAT	msat
Z3	z3
CVC4	cvc4
Yices	yices
Boolector	btor
Picosat	picosat
CUDD	bdd

You can also not specify the solver, and simply state which Logic must be supported by the solver, this will look into the installed solvers and pick one that supports the logic. This might raise an exception (*NoSolverAvailableError*), if no logic for the logic is available.

Here is the complete example for reference using the logic QF_LIA:

```
from pysmt.shortcuts import Symbol, LE, GE, Int, And, Equals, Plus, Solver
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]

letters = set(hello+world)

domains = And(And(LE(Int(1), l),
                  GE(Int(10), l)) for l in letters)

sum_hello = Plus(hello)
sum_world = Plus(world)

problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(36)))

formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)

with Solver(logic="QF_LIA") as solver:
    solver.add_assertion(domains)
    if not solver.solve():
        print("Domain is not SAT!!!")
        exit()
    solver.add_assertion(problem)
    if solver.solve():
```

```
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

2.1.3 What's Next?

This simple example provides the basic ideas of how to work with pySMT. The best place to understand more about pySMT is the `pysmt.shortcuts` module. All the important functionalities are exported there with a simple to use interface.

To understand more about other functionalities of pySMT, you can take a look at the examples/ folder .

2.2 Tutorials

This page is under-construction. For now, it contains a copy of the files within the examples/ folder of the pySMT repository.

If you are interested in helping us create better tutorials, please let us know at info@pysmt.org .

2.2.1 First example

```
# Checking satisfiability of a formula.
#
# This example shows:
#  1. How to build a formula
#  2. How to perform substitution
#  3. Printing
#  4. Satisfiability checking
from pysmt.shortcuts import Symbol, And, Not, is_sat

varA = Symbol("A") # Default type is Boolean
varB = Symbol("B")
f = And([varA, Not(varB)])
g = f.substitute({varB:varA})

res = is_sat(f)
assert res # SAT
```

```
print("f := %s is SAT? %s" % (f, res))

res = is_sat(g)
print("g := %s is SAT? %s" % (g, res))
assert not res # UNSAT
```

2.2.2 Hello World word puzzle

```
# This is the tutorial example of pySMT.
#
# This example shows how to:
# 1. Deal with Theory atoms
# 2. Specify a solver in the shortcuts (get_model, is_sat etc.)
# 3. Obtain an print a model
#
#
# The goal of the puzzle is to assign a value from 1 to 10 to each letter s.t.
#    H+E+L+L+O = W+O+R+L+D = 25
#
from pysmt.shortcuts import Symbol, And, GE, LT, Plus, Equals, Int, get_model
from pysmt.typing import INT

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]
letters = set(hello+world)
domains = And([And(GE(l, Int(1)),
                   LT(l, Int(10))) for l in letters])

sum_hello = Plus(hello) # n-ary operators can take lists
sum_world = Plus(world) # as arguments
problem = And(Equals(sum_hello, sum_world),
              Equals(sum_hello, Int(25)))
formula = And(domains, problem)

print("Serialization of the formula:")
print(formula)

model = get_model(formula)
if model:
    print(model)
else:
    print("No solution found")
```

2.2.3 Hello World word puzzle using infix-notation

```
# This is a different take on the puzzle.py example
#
# This examples shows how to:
# 1. Enable and use infix notation
# 2. Use a solver context
#
from pysmt.shortcuts import Symbol, And, Plus, Int
from pysmt.shortcuts import Solver
from pysmt.typing import INT
```

```
# Infix-Notation is automatically enabled whenever you import pysmt.shortcuts.
#
# To enable it without using shortcuts, do:
#
#   from pysmt.environment import get_env
#   get_env().enable_infix_notation = True
#
# Similarly, you can disable infix_notation to prevent its accidental use.
#

hello = [Symbol(s, INT) for s in "hello"]
world = [Symbol(s, INT) for s in "world"]
letters = set(hello+world)

# Infix notation for Theory atoms does the overloading of python
# operator. For boolean connectors, we use e.g., x.And(y) This
# increases readability without running into problems of operator
# precedence.
#
# Note how you can mix prefix and infix notation.
domains = And([(Int(1) <= l).And(Int(10) >= l) for l in letters])

sum_hello = Plus(hello) # n-ary operators can take lists
sum_world = Plus(world) # as arguments
problem = (sum_hello.Equals(sum_world)).And(sum_hello.Equals(Int(25)))
formula = domains.And(problem)

print("Serialization of the formula:")
print(formula)

# A context (with-statment) lets python take care of creating and
# destroying the solver.
with Solver() as solver:
    solver.add_assertion(formula)
    if solver.solve():
        for l in letters:
            print("%s = %s" %(l, solver.get_value(l)))
    else:
        print("No solution found")
```

2.2.4 Combine multiple solvers

```
# This example requires Z3 and MathSAT to be installed (but you can
#  replace MathSAT with any other solver for QF_LRA)
#
# This examples shows how to:
# 1. Define Real valued constants using floats and fractions
# 2. Perform quantifier elimination
# 3. Pass results from one solver to another
#
from pysmt.shortcuts import Symbol, Or, ForAll, GE, LT, Real, Plus
from pysmt.shortcuts import qelim, is_sat
from pysmt.typing import REAL

x, y, z = [Symbol(s, REAL) for s in "xyz"]

f = ForAll([x], Or(LT(x, Real(5.0)),
```

```
                    GE(Plus(x, y, z), Real((17,2))))) # (17,2) ~> 17/2
print("f := %s" % f)
#f := (forall x . ((x < 5.0) | (17/2 <= (x + y + z))))

qf_f = qelim(f, solver_name="z3")
print("Quantifier-Free equivalent: %s" % qf_f)
#Quantifier-Free equivalent: (7/2 <= (z + y))

res = is_sat(qf_f, solver_name="msat")
print("SAT check using MathSAT: %s" % res)
#SAT check using MathSAT: True
```

2.2.5 Model-Checking an infinite state system (BMC+K-Induction) in ~150 lines

```
# This example shows a more advance use of pySMT.
#
# It provides a simple implementation of Bounded Model Checking [1]
# and K-Induction [2] and applies it on a simple infinite-state
# transition system.
#
# [1] ...
#
# [2] ...
#
from six.moves import xrange

from pysmt.shortcuts import Symbol, Not, Equals, And, Times, Int, Plus, LE
from pysmt.shortcuts import is_sat, is_unsat
from pysmt.typing import INT


def next_var(v):
    """Returns the 'next' of the given variable"""
    return Symbol("next(%s)" % v.symbol_name(), v.symbol_type())

def at_time(v, t):
    """Builds an SMT variable representing v at time t"""
    return Symbol("%s@%d" % (v.symbol_name(), t), v.symbol_type())


class TransitionSystem(object):
    """Trivial representation of a Transition System."""

    def __init__(self, variables, init, trans):
        self.variables = variables
        self.init = init
        self.trans = trans

# EOC TransitionSystem



def get_subs(system, i):
    """Builds a map from x to x@i and from x' to x@(i+1), for all x in system."""
    subs_i = {}
    for v in system.variables:
```

```
        subs_i[v] = at_time(v, i)
        subs_i[next_var(v)] = at_time(v, i+1)
    return subs_i


def get_unrolling(system, k):
    """Unrolling of the transition relation from 0 to k:

    E.g. T(0,1) & T(1,2) & ... & T(k-1,k)
    """
    res = []
    for i in xrange(k):
        subs_i = get_subs(system, i)
        res.append(system.trans.substitute(subs_i))
    return And(res)


def get_simple_path(system, k):
    """Simple path constraint for k-induction:
    each time encodes a different state
    """
    res = []
    for i in xrange(k):
        subs_i = get_subs(system, i)
        for j in xrange(i+1, k):
            subs_j = get_subs(system, j)
            for v in system.variables:
                v_i = v.substitute(subs_i)
                v_j = v.substitute(subs_j)
                res.append(Not(Equals(v_i, v_j)))
    return And(res)


def get_k_hypothesis(system, prop, k):
    """Hypothesis for k-induction: each state up to k fulfills the property"""
    res = []
    for i in xrange(k):
        subs_i = get_subs(system, i)
        res.append(prop.substitute(subs_i))
    return And(res)


def get_bmc(system, prop, k):
    """Returns the BMC encoding at step k"""
    init_0 = system.init.substitute(get_subs(system, 0))
    prop_k = prop.substitute(get_subs(system, k))
    return And(get_unrolling(system, k), init_0, Not(prop_k))

def get_k_induction(system, prop, k):
    """Returns the K-Induction encoding at step K"""
    subs_k = get_subs(system, k)
    prop_k = prop.substitute(subs_k)
    return And(get_unrolling(system, k),
               get_k_hypothesis(system, prop, k),
               get_simple_path(system, k),
               Not(prop_k))

def check_property(system, prop):
```

```
    """Interleaves BMC and K-Ind to verify the property."""
    print("Checking property %s..." % prop)
    for b in xrange(100):
        f = get_bmc(system, prop, b)
        print("   [BMC]    Checking bound %d..." % (b+1))
        if is_sat(f):
            print("--> Bug found at step %d" % (b+1))
            return

        f = get_k_induction(system, prop, b)
        print("   [K-IND]  Checking bound %d..." % (b+1))
        if is_unsat(f):
            print("--> The system is safe!")
            return


def main():
    # Example Transition System (SMV-like syntax)
    #
    # VAR x: integer;
    #     y: integer;
    #
    # INIT: x = 1 & y = 2;
    #
    # TRANS: next(x) = x + 1;
    # TRANS: next(y) = y + 2;

    x, y = [Symbol(s, INT) for s in "xy"]
    nx, ny = [next_var(Symbol(s, INT)) for s in "xy"]

    example = TransitionSystem(variables = [x, y],
                init = And(Equals(x, Int(1)),
                           Equals(y, Int(2))),
                trans = And(Equals(nx, Plus(x, Int(1))),
                            Equals(ny, Plus(y, Int(2)))))

    # A true invariant property: y = x * 2
    true_prop = Equals(y, Times(x, Int(2)))

    # A false invariant property: x <= 10
    false_prop = LE(x, Int(10))

    for prop in [true_prop, false_prop]:
        check_property(example, prop)
        print("")

if __name__ == "__main__":
    main()
```

2.2.6 How to access functionalities of solvers not currently wrapped by pySMT

```
# This example requires MathSAT to be installed.
#
# This example shows how to use a solver converter to access
# functionalities of the solver that are not wrapped by pySMT.
#
# Our goal is to call the method msat_all_sat from the MathSAT API.
```

```
#
import mathsat
from pysmt.shortcuts import Or, Symbol, Solver, And

def callback(model, converter, result):
    """Callback for msat_all_sat.

    This function is called by the MathSAT API everytime a new model
    is found. If the function returns 1, the search continues,
    otherwise it stops.
    """
    # Elements in model are msat_term .
    # Converter.back() provides the pySMT representation of a solver term.
    py_model = [converter.back(v) for v in model]
    result.append(And(py_model))
    return 1 # go on

x, y = Symbol("x"), Symbol("y")
f = Or(x, y)

msat = Solver(name="msat")
converter = msat.converter # .converter is a property implemented by all solvers
msat.add_assertion(f) # This is still at pySMT level

result = []
# Directly invoke the mathsat API !!!
# The second term is a list of "important variables"
mathsat.msat_all_sat(msat.msat_env(),
      [converter.convert(x)],      # Convert the pySMT term into a MathSAT term
      lambda model : callback(model, converter, result))

print("'exists y . %s' is equivalent to '%s'" %(f, Or(result)))
#exists y . (x | y) is equivalent to ((! x) | x)
```

2.2.7 How to use any SMT-LIB complaint SMT solver

```
# This example shows how to define a generic SMT-LIB solver, and use
# it within pySMT. The example looks for mathsat in /tmp, you can
# create a symlink there.
#
# Using this process you can experiment with any SMT-LIB 2 compliant
# solver even if it does not have python bindings, or has not been
# integrated with pySMT.
#
# Note: When using the SMT-LIB wrapper, you can only use logics
# supported by pySMT. If the version of pySMT in use does not support
# Arrays, then you cannot represent arrays.
#

# To define a Generic Solver you need to provide:
#
# - A name to associate to the solver
# - The path to the script + Optional arguments
# - The list of logics supported by the solver
#
# It is usually convenient to wrap the solver in a simple shell script.
```

```
# See examples for Z3, Mathsat and Yices in pysmt/test/smtlib/bin/*.template
#
from pysmt.logics import QF_UFLRA, QF_UFIDL, QF_LRA, QF_IDL, QF_LIA
from pysmt.shortcuts import get_env, GT, Solver, Symbol
from pysmt.typing import REAL, INT
from pysmt.exceptions import NoSolverAvailableError

name = "mathsat" # Note: The API version is called 'msat'
path = ["/tmp/mathsat"] # Path to the solver
logics = [QF_UFLRA, QF_UFIDL] # Some of the supported logics

env = get_env()

# Add the solver to the environment
env.factory.add_generic_solver(name, path, logics)

r, s = Symbol("r", REAL), Symbol("s", REAL)
p, q = Symbol("p", INT), Symbol("q", INT)

f_lra = GT(r, s)
f_idl = GT(p, q)

# PySMT takes care of recognizing that QF_LRA can be solved by a QF_UFLRA solver.
with Solver(name=name, logic=QF_LRA) as s:
    res = s.solve()
    assert res, "Was expecting '%s' to be SAT" % f_lra


with Solver(name=name, logic=QF_IDL) as s:
    s.add_assertion(f_idl)
    res = s.solve()
    assert res, "Was expecting '%s' to be SAT" % f_idl

try:
    with Solver(name=name, logic=QF_LIA) as s:
        pass
except NoSolverAvailableError:
    # If we ask for a logic that is not contained by any of the
    # supported logics an exception is thrown
    print("%s does not support QF_LIA" % name)
```

2.2.8 How to combine two different solvers to solve an Exists Forall problem

```
# EF-SMT solver implementation
#
# This example shows:
#  1. How to combine 2 different solvers
#  2. How to extract information from a model
#
from pysmt.shortcuts import Solver, get_model
from pysmt.shortcuts import Symbol, Bool, Real, Implies, And, Not, Equals
from pysmt.shortcuts import GT, LT, LE, Minus, Times
from pysmt.logics import AUTO, QF_LRA
from pysmt.typing import REAL
from pysmt.exceptions import SolverReturnedUnknownResultError
```

```
def efsmt(y, phi, logic=AUTO, maxloops=None,
          esolver_name=None, fsolver_name=None,
          verbose=False):
    """Solves exists x. forall y. phi(x, y)"""

    y = set(y)
    x = phi.get_free_variables() - y

    with Solver(logic=logic, name=esolver_name) as esolver:

        esolver.add_assertion(Bool(True))
        loops = 0
        while maxloops is None or loops <= maxloops:
            loops += 1

            eres = esolver.solve()
            if not eres:
                return False
            else:
                tau = {v: esolver.get_value(v) for v in x}
                sub_phi = phi.substitute(tau).simplify()
                if verbose: print("%d: Tau = %s" % (loops, tau))

                fmodel = get_model(Not(sub_phi),
                                   logic=logic, solver_name=fsolver_name)
                if fmodel is None:
                    return tau
                else:
                    sigma = {v: fmodel[v] for v in y}
                    sub_phi = phi.substitute(sigma).simplify()
                    if verbose: print("%d: Sigma = %s" % (loops, sigma))
                    esolver.add_assertion(sub_phi)

        raise SolverReturnedUnknownResultError


def run_test(y, f):
    print("Testing " + str(f))
    try:
        res = efsmt(y, f, logic=QF_LRA, maxloops=20, verbose=True)
        if res == False:
            print("unsat")
        else:
            print("sat : %s" % str(res))
    except SolverReturnedUnknownResultError:
        print("unknown")
    print("\n\n")


def main():
    x,y = [Symbol(n, REAL) for n in "xy"]
    f_sat = Implies(And(GT(y, Real(0)), LT(y, Real(10))),
                    LT(Minus(y, Times(x, Real(2))), Real(7)))

    f_incomplete = And(GT(x, Real(0)), LE(x, Real(10)),
                       Implies(And(GT(y, Real(0)), LE(y, Real(10)),
                                   Not(Equals(x, y))),
                               GT(y, x)))
```

```
    run_test([y], f_sat)
    run_test([y], f_incomplete)


if __name__ == "__main__":
    main()
```

2.2.9 How to detect the logic of a formula and perform model enumeration

```
# Perform ALL-SMT on Theory atoms
#
# This example shows:
#  - How to get the logic of a formula
#  - How to enumerate models
#  - How to extract a partial model
#  - How to use the special operator EqualsOrIff
#
from pysmt.shortcuts import Solver, Not, And, Symbol, Or
from pysmt.shortcuts import LE, GE, Int, Plus, Equals, EqualsOrIff
from pysmt.typing import INT
from pysmt.oracles import get_logic

def all_smt(formula, keys):
    target_logic = get_logic(formula)
    print("Target Logic: %s" % target_logic)
    with Solver(logic=target_logic) as solver:
        solver.add_assertion(formula)
        while solver.solve():
            partial_model = [EqualsOrIff(k, solver.get_value(k)) for k in keys]
            print(partial_model)
            solver.add_assertion(Not(And(partial_model)))


A0 = Symbol("A0", INT)
A1 = Symbol("A1", INT)
A2 = Symbol("A2", INT)

f = And(GE(A0, Int(0)), LE(A0, Int(5)),
        GE(A1, Int(0)), LE(A1, Int(5)),
        GE(A2, Int(0)), LE(A2, Int(5)),
        Equals(Plus(A0, A1, A2), Int(8)))

all_smt(f, [A0, A1, A2])

# By using the operator EqualsOrIff, we can mix theory and bool variables
x = Symbol("x")
y = Symbol("y")
f = And(f, Or(x,y))

all_smt(f, [A0, A1, A2, x])
```

2.2.10 Shows how to use multi-processing to perform parallel and asynchronous solving

```
# This example requires Z3
#
# This example shows how to parallelize solving using pySMT, by using
# the multiprocessing library.
#
# All shortcuts (is_sat, is_unsat, get_model, etc.) can be easily
# used. We sho two techniques: map and apply_async.
#
# Map applies a given function to a list of elements, and returns a
# list of results. We show an example using is_sat.
#
# Apply_async provides a way to perform operations in an asynchronous
# way. This allows us to start multiple solving processes in parallel,
# and collect the results whenever they become available.
#
# More information can be found in the Python Standard Library
# documentation for the multiprocessing module.
#
#
# NOTE: When running this example, a warning will appear saying:
#   "Contextualizing formula during is_sat"
#
# Each process will inherit a different formula manager, but the
# formulas that we are solving have been all created in the same
# formula manager. pySMT takes care of moving the formula across
# formula managers. This step is called contextualization, and occurs
# only when using multiprocessing.
# To disable the warning see the python module warnings.
#
from multiprocessing import Pool, TimeoutError
from time import sleep

from pysmt.test.examples import get_example_formulae
from pysmt.shortcuts import is_sat, is_valid, is_unsat
from pysmt.shortcuts import And

# Ignore this for now
def check_validity_and_test(args):
    """Checks expression and compare the outcome against a known value."""

    expr, expected = args # IMPORTANT: Unpack args !!!
    local_res = is_valid(expr)
    return local_res == expected


# Create the Pool with 4 workers.
pool = Pool(4)

# We use the examples formula from the test suite.
# This generator iterates over all the expressions
f_gen = (f.expr for f in get_example_formulae())

# Call the functino is_sat on each expression
res = pool.map(is_sat, f_gen)
```

```
# The result is a list of True/False, in the same order as the input.
print(res)

sleep(1) # Have some time to look at the result

# Notice that all shortcuts (is_sat, qelim, etc) require only one
# mandatory argument. To pass multiple arguments, we need to pack them
# together.

# Lets create a list of (formula, result), where result is the
# expected result of the is_valid query.
f_gen = (f.expr for f in get_example_formulae())
res_gen = (f.is_valid for f in get_example_formulae())
args_gen = zip(f_gen, res_gen)

# We now define a function that check the formula against the expected
# value of is_valid: See check_validity_and_test(...) above.
# Due to the way multiprocessing works, we need to define the function
# _before_ we create the Pool.

# As before, we call the map on the pool
res = pool.map(check_validity_and_test, args_gen)

# We can also apply a map-reduce strategy, by making sure that all
# results are as expected.
if all(res):
    print("Everything is ok!")
else:
    print("Ooops, something is wrong!")
    print(res)

# A different option is to run solvers in an asynchronous way.  This
# does not provide us with any guarantee on the execution order, but
# it is particular convenient, if we want to perform solving together
# with another long operation (e.g., I/O, network etc.)  or if we want
# to run multiple solvers.

# Create a formula
big_f = And(f.expr for f in get_example_formulae() \
            if not f.logic.theory.bit_vectors and \
               not f.logic.theory.arrays and \
                   f.logic.theory.linear)

# Create keyword arguments for the function call.
# This is the simplest way to pass multiple arguments to apply_async.
kwargs = {"formula": big_f, "solver_name": "z3"}
future_res_sat = pool.apply_async(is_sat, kwds=kwargs)
future_res_unsat = pool.apply_async(is_unsat, kwds=kwargs)

# In the background, the solving is taking place... We can do other
# stuff in the meanwhile.
print("This is non-blocking...")

# Get the result with a deadline.
# See multiprocessing.pool.AsyncResult for more options
sat_res = future_res_sat.get(10)  # Get result after 10 seconds or kill
try:
    unsat_res = future_res_unsat.get(0) # No wait
```

```
except TimeoutError:
    print("UNSAT result was not ready!")
    unsat_res = None
print(sat_res, unsat_res)
```

2.2.11 Demonstrates how to perform SMT-LIB parsing, dumping and extension

```
# The last part of the example requires a QF_LIA solver to be installed.
#
#
# This example shows how to interact with files in the SMT-LIB
# format. In particular:
#
# 1. How to read a file in SMT-LIB format
# 2. How to write a file in SMT-LIB format
# 3. Formulas and SMT-LIB script
# 4. How to access annotations from SMT-LIB files
# 5. How to extend the parser with custom commands
#
from six.moves import cStringIO # Py2-Py3 Compatibility

from pysmt.smtlib.parser import SmtLibParser


# To make the example self contained, we store the example SMT-LIB
# script in a string.
DEMO_SMTLIB=\
"""
(set-logic QF_LIA)
(declare-fun p () Int)
(declare-fun q () Int)
(declare-fun x () Bool)
(declare-fun y () Bool)
(define-fun .def_1 () Bool (! (and x y) :cost 1))
(assert (=> x (> p q)))
(check-sat)
(push)
(assert (=> y (> q p)))
(check-sat)
(assert .def_1)
(check-sat)
(pop)
(check-sat)
"""

# We read the SMT-LIB Script by creating a Parser.
# From here we can get the SMT-LIB script.
parser = SmtLibParser()

# The method SmtLibParser.get_script takes a buffer in input. We use
# cStringIO to simulate an open file.
# See SmtLibParser.get_script_fname() if to pass the path of a file.
script = parser.get_script(cStringIO(DEMO_SMTLIB))

# The SmtLibScript provides an iterable representation of the commands
# that are present in the SMT-LIB file.
#
```

```
# Printing a summary of the issued commands
for cmd in script:
    print(cmd.name)
print("*"*50)

# SmtLibScript provides some utilities to perform common operations: e.g,
#
# - Checking if a command is present
assert script.contains_command("check-sat")
# - Counting the occurrences of a command
assert script.count_command_occurrences("assert") == 3
# - Obtain all commands of a particular type
decls = script.filter_by_command_name("declare-fun")
for d in decls:
    print(d)
print("*"*50)

# Most SMT-LIB scripts define a single SAT call. In these cases, the
# result can be obtained by conjoining multiple assertions.  The
# method to do that is SmtLibScript.get_strict_formula() that, raises
# an exception if there are push/pop calls.  To obtain the formula at
# the end of the execution of the Script (accounting for push/pop) we
# use get_last_formula
#
f = script.get_last_formula()
print(f)

# Finally, we serialize the script back into SMT-Lib format.  This can
# be dumped into a file (see SmtLibScript.to_file).  The flag daggify,
# specifies whether the printing is done as a DAG or as a tree.

buf_out = cStringIO()
script.serialize(buf_out, daggify=True)
print(buf_out.getvalue())

print("*"*50)
# Expressions can be annotated in order to provide additional
# information. The semantic of annotations is solver/problem
# dependent. For example, VMT uses annotations to identify two
# expressions as 1) the Transition Relation and 2) Initial Condition
#
# Here we pretend that we make up a ficticious Weighted SMT format
# and label .def1 with cost 1
#
# The class pysmt.smtlib.annotations.Annotations deals with the
# handling of annotations.
#
ann = script.annotations
print(ann.all_annotated_formulae("cost"))

print("*"*50)

# Annotations are part of the SMT-LIB standard, and are the
# recommended way to perform inter-operable operations. However, in
# many cases, we are interested in prototyping some algorithm/idea and
# need to write the input files by hand. In those cases, using an
# extended version of SMT-LIB usually provides a more readable input.
# We provide now an example on how to define a symbolic transition
```

```
# system as an extension of SMT-LIB.
# (A more complete version of this example can be found in :
#    pysmt.tests.smtlib.test_parser_extensibility.py)
#
EXT_SMTLIB="""\
(declare-fun A () Bool)
(declare-fun B () Bool)
(init (and A B))
(trans (=> A (next A)))
(exit)
"""

# We define two new commands (init, trans) and a new
# operator (next). In order to parse this file, we need to create a
# sub-class of the SmtLibParser, and add handlers for he new commands
# and operators.
from pysmt.smtlib.script import SmtLibCommand

class TSSmtLibParser(SmtLibParser):
    def __init__(self, env=None, interactive=False):
        SmtLibParser.__init__(self, env, interactive)

        # Add new commands
        #
        # The mapping function takes care of consuming the command
        # name from the input stream, e.g., '(init' . Therefore,
        # _cmd_init will receive the rest of the stream, in our
        # example, '(and A B)) ...'
        self.commands["init"] = self._cmd_init
        self.commands["trans"] = self._cmd_trans

        # Remove unused commands
        #
        # If some commands are not compatible with the extension, they
        # can be removed from the parser. If found, they will cause
        # the raising of the exception UnknownSmtLibCommandError
        del self.commands["check-sat"]
        del self.commands["get-value"]
        # ...

        # Add 'next' function
        #
        # New operators can be added similarly as done for commands.
        # e.g., 'next'. The helper function _operator_adapter,
        # simplifies the writing of such extensions.  In this example,
        # we will rewrite the content of the next without the need of
        # introducing a new pySMT operator. If you are interested in a
        # simple way of handling new operators in pySMT see
        # pysmt.test.test_dwf.
        self.interpreted["next"] = self._operator_adapter(self._next_var)

    def _cmd_init(self, current, tokens):
        # This cmd performs the parsing of:
        #   <expr> )
        # and returns a new SmtLibCommand
        expr = self.get_expression(tokens)
        self.consume_closing(tokens, current)
        return SmtLibCommand(name="init", args=(expr,))
```

```
    def _cmd_trans(self, current, tokens):
        # This performs the same parsing as _cmd_init, but returns a
        # different object. The parser is not restricted to return
        # SmtLibCommand, but using them makes handling them
        # afterwards easier.
        expr = self.get_expression(tokens)
        self.consume_closing(tokens, current)
        return SmtLibCommand(name="trans", args=(expr,))

    def _next_var(self, symbol):
        # The function is called with the arguments obtained from
        # parsing the rest of the SMT-LIB file. In this case, 'next'
        # is a unary function, thus we have only 1 argument. 'symbol'
        # is an FNode. We require that 'symbol' is _really_ a symbol:
        if symbol.is_symbol():
            name = symbol.symbol_name()
            ty = symbol.symbol_type()
            # The return type MUST be an FNode, because this is part
            # of an expression.
            return self.env.formula_manager.Symbol("next_" + name, ty)
        else:
            raise ValueError("'next' operator can be applied only to symbols")

    def get_ts(self, script):
        # New Top-Level command that takes a script stream in input.
        # We return a pair (Init, Trans) that defines the symbolic
        # transition system.
        init = self.env.formula_manager.TRUE()
        trans = self.env.formula_manager.TRUE()

        for cmd in self.get_command_generator(script):
            if cmd.name=="init":
                init = cmd.args[0]
            elif cmd.name=="trans":
                trans = cmd.args[0]
            else:
                # Ignore other commands
                pass

        return (init, trans)

# Time to try out the parser !!!
#
# First we check that the standard SMT-Lib parser cannot handle the new format.
from pysmt.exceptions import UnknownSmtLibCommandError

try:
    parser.get_script(cStringIO(EXT_SMTLIB))
except UnknownSmtLibCommandError as ex:
    print("Unsupported command: %s" % ex)

# The new parser can parse our example, and returns the (init, trans) pair
ts_parser = TSSmtLibParser()
init, trans = ts_parser.get_ts(cStringIO(EXT_SMTLIB))
print("INIT: %s" % init.serialize())
print("TRANS: %s" % trans.serialize())
```

2.2.12 Shows the use of UNSAT Core as debugging tools

```
#
# This example requires MathSAT or Z3
#
# In this example, we will encode a more complex puzzle and see:
#
# - Use of UNSAT cores as a debugging tool
# - Conjunctive partitioning
# - Symbol handling delegation to auxiliary functions
#

# This puzzle is known as Einstein Puzzle
#
# There are five houses in five different colours in a row. In each
# house lives a person with a different nationality. The five owners
# drink a certain type of beverage, smoke a certain brand of cigar and
# keep a certain pet.
#
# No owners have the same pet, smoke the same brand of cigar, or drink
# the same beverage.
#
# The Brit lives in the red house.
# The Swede keeps dogs as pets.
# The Dane drinks tea.
# The green house is on the immediate left of the white house.
# The green house owner drinks coffee.
# The owner who smokes Pall Mall rears birds.
# The owner of the yellow house smokes Dunhill.
# The owner living in the center house drinks milk.
# The Norwegian lives in the first house.
# The owner who smokes Blends lives next to the one who keeps cats.
# The owner who keeps the horse lives next to the one who smokes Dunhill.
# The owner who smokes Bluemasters drinks beer.
# The German smokes Prince.
# The Norwegian lives next to the blue house.
# The owner who smokes Blends lives next to the one who drinks water.
#
# The question is: who owns the fish?

from pysmt.shortcuts import Symbol, ExactlyOne, Or, And, FALSE, Iff
from pysmt.shortcuts import get_model, get_unsat_core, is_sat, is_unsat

#
# Lets start by expliciting all values for all dimensions

Color = "white", "yellow", "blue", "red", "green"
Nat = "german", "swedish", "british", "norwegian", "danish"
Pet =  "birds", "cats", "horses", "fish", "dogs"
Drink = "beer", "water", "tea", "milk", "coffee"
Smoke = "blends", "pall_mall", "prince", "bluemasters", "dunhill"
Houses = range(0,5)
#
# We number the houses from 0 to 4, and create the macros to assert
# properties of the i-th house:
#
#  e.g., color(1, "green")  to indicate that the house 1 is Green
#
```

```
# This is not strictly necessary, but it is a way of making programs
# more readable.
#
def color(number, name):
    assert name in Color
    if number in Houses:
        return Symbol("%d_color_%s" % (number, name))
    return FALSE()

def nat(number, name):
    assert name in Nat
    if number in Houses:
        return Symbol("%d_nat_%s" % (number, name))
    return FALSE()

def pet(number, name):
    assert name in Pet
    if number in Houses:
        return Symbol("%d_pet_%s" % (number, name))
    return FALSE()

def drink(number, name):
    assert name in Drink
    if number in Houses:
        return Symbol("%d_drink_%s" % (number, name))
    return FALSE()

def smoke(number, name):
    assert name in Smoke
    if number in Houses:
        return Symbol("%d_smoke_%s" % (number, name))
    return FALSE()

#
# We can encode the facts
#

facts = And(
    # The Brit lives in the red house.
    And( Iff(nat(i, "british"), color(i, "red")) for i in Houses ),

    # The Swede keeps dogs as pets.
    And( Iff(nat(i, "swedish"), pet(i, "dogs")) for i in Houses ),

    # The Dane drinks tea.
    And( Iff(nat(i, "danish"), drink(i, "tea")) for i in Houses ) ,

    # The green house is on the immediate left of the white house.
    And( Iff(color(i, "green"), color(i+1, "white")) for i in Houses) ,

    # The green house owner drinks coffee.
    And( Iff(color(i, "green"), drink(i, "coffee")) for i in Houses ) ,

    # The owner who smokes Pall Mall rears birds.
    And( Iff(smoke(i, "pall_mall"), pet(i, "birds")) for i in Houses ) ,

    # The owner of the yellow house smokes Dunhill.
    And( Iff(color(i, "yellow"), smoke(i, "dunhill")) for i in Houses ) ,
```

```
    # The owner living in the center house drinks milk.
    And( drink(2, "milk") ) ,

    # The Norwegian lives in the first house.
    And( nat(0, "norwegian") ) ,

    # The owner who smokes Blends lives next to the one who keeps cats.
    And( Iff(smoke(i, "blends"), Or(pet(i-1, "cats"), pet(i+1, "cats"))) for i in Houses ) ,

    # The owner who keeps the horse lives next to the one who smokes Dunhill.
    And( Iff(pet(i, "horses"), Or(smoke(i-1, "dunhill"), smoke(i+1, "dunhill"))) for i in Houses ) ,

    # The owner who smokes Bluemasters drinks beer.
    And( Iff(smoke(i, "bluemasters"), drink(i, "beer")) for i in Houses ) ,

    # The German smokes Prince.
    And( Iff(nat(i, "german"), smoke(i, "prince")) for i in Houses ) ,

    # The Norwegian lives next to the blue house.
    # Careful with this!!!
    And( Iff(nat(i, "norwegian"), Or(color(i-1, "blue"), color(i+1, "blue"))) for i in Houses ) ,

    # The owner who smokes Blends lives next to the one who drinks water.
    And( Iff(smoke(i, "blends"), Or(drink(i-1, "water"), drink(i+1, "water"))) for i in Houses )
    )

domain = And(
    And(ExactlyOne(color(i, c) for i in Houses) for c in Color),
    And(ExactlyOne(nat(i, c) for i in Houses) for c in Nat),
    And(ExactlyOne(pet(i, c) for i in Houses) for c in Pet),
    And(ExactlyOne(drink(i, c) for i in Houses) for c in Drink),
    And(ExactlyOne(smoke(i, c) for i in Houses) for c in Smoke),
    #
    And(ExactlyOne(color(i, c) for c in Color) for i in Houses),
    And(ExactlyOne(nat(i, c) for c in Nat) for i in Houses),
    And(ExactlyOne(pet(i, c) for c in Pet) for i in Houses),
    And(ExactlyOne(drink(i, c) for c in Drink) for i in Houses),
    And(ExactlyOne(smoke(i, c) for c in Smoke) for i in Houses),
    )

problem = And(domain, facts)

model = get_model(problem)

if model is None:
    print("UNSAT")
    # We first check whether the constraints on the domain and problem
    # are satisfiable in isolation.
    assert is_sat(facts)
    assert is_sat(domain)
    assert is_unsat(problem)

    # In isolation they are both fine, rules from both are probably
    # interacting.
    #
    # The problem is given by a nesting of And().
    # conjunctive_partition can be used to obtain a "flat"
    # structure, i.e., a list of conjuncts.
```

```
    #
    from pysmt.rewritings import conjunctive_partition
    conj = conjunctive_partition(problem)
    ucore = get_unsat_core(conj)
    print("UNSAT-Core size '%d'" % len(ucore))
    for f in ucore:
        print(f.serialize())

    # The exact version of the UNSAT-Core depends on the solver in
    # use.  Nevertheless, this represents a starting point for your
    # debugging.  A possible way to approach the result is to look for
    # clauses of size 1 (i.e., unit clauses). In the facts list there
    # are only 2 facts:
    #   2_drink_milk
    #   0_nat_norwegian
    #
    # The clause ("1_color_blue" <-> "0_nat_norwegian")
    # Implies that "1_color_blue"
    # But (("3_color_blue" | "1_color_blue") <-> "2_nat_norwegian")
    # Requires "2_nat_norwegian"
    # The ExactlyOne constraint forbids that both 0 and 2 are nowegian
    # thus, we have a better idea of where the problem might be.
    #
    # Please go back to the comment '# Careful with this!!!' in the
    # facts list, and change the Iff with an Implies.
    #
    # Done?
    #
    # Good, you should be getting a model, now!
else:
    for h in Houses:
        # Extract the relevants bits to get some pretty-printing
        c = [x for x in Color if model[color(h, x)].is_true()][0]
        n = [x for x in Nat if model[nat(h, x)].is_true()][0]
        p = [x for x in Pet if model[pet(h, x)].is_true()][0]
        d = [x for x in Drink if model[drink(h, x)].is_true()][0]
        s = [x for x in Smoke if model[smoke(h, x)].is_true()][0]
        print(h, c, n, p, d, s)
        if p == "fish":
            sol = "The '%s' owns the fish!" % n
    print(sol)
```

2.3 Change Log

2.3.1 0.6.1: 2016-12-02 – Portfolio and Coverage

General:

- Portfolio Solver (PR #284):

  Created Portfolio class that uses multiprocessing to solve the problem using multiple solvers. get_value and get_model work after a SAT query. Other artifacts (unsat-core, interpolants) are not supported. Factory.is_* methods have been extended to include *portfolio* key-word, and exported as is_* shortcuts. The syntax becomes:

  is_sat(f, portfolio=[”s1”, “s2”])

- Coverage has been significantly improved, thus giving raise to some clean-up of the tests and minor bug fixes. Thanks to Coveralls.io for providing free coverage analysis. (PR #353, PR #358, PR #372)
- Introduce PysmtException, from which all exceptions must inherit. This also introduces hybrid exceptions that inherit both from the Standard Library and from PysmtException (i.e., PysmtValueError). Thanks to **Alberto Griggio** for suggesting this change. (PR #365)
- Windows: Add support for installing Z3. Thanks to **Samuele Gallerani** for contributing this patch. (PR #385)
- Arrays: Improved efficiency of array_value_get (PR #357)
- Documentation: Thanks to the **Hacktoberfest** for sponsoring these activities:
  - Every function in shortcuts.py now has a docstring! Thanks to **Vijay Raghavan** for contributing this patch. (PR #363)
  - Contributing information has been moved to the official documentation and prettyfied! Thanks to **Jason Taylor Hodge** for contributing this patch. (PR #339)
  - Add link to Google Group in Readme.md . Thanks to @ankit01ojha for contributing this. (PR #345)
- smtlibscript_from_formula(): Allow the user to specify a custom logic. Thanks to **Alberto Griggio** for contributing this patch. (PR #360)

Solvers:

- MathSAT: Improve back-conversion performance by using MSAT_TAGS (PR #379)
- MathSAT: Add LIA support for Quantifier Elimination
- Removed: Solver.declare_variable and Solver.set_options (PR #369, PR #378)

Bugfix:

- CVC4:
  - Enforce BV Division by 0 to return a known value (0xFF) (PR #351)
  - Force absolute import of CVC4. Thanks to **Alexey Ignatiev** (@2sev) for reporting this issue. (PR #382)
- MathSAT: Thanks to **Alberto Griggio** for contributing these patches
  - Fix assertions about arity of BV sign/zero extend ops. (PR #350, PR #351)
  - Report the error message generated by MathSAT when raising a SolverReturnedUnknownResultError (PR #355)
- Enforce a single call to is_sat in non-incremental mode (PR #368). Thanks to @colinmorris for pointing out this issue.
- Clarified Installation section and added example of call to `` `pysmt-install --env` ``. Thanks to **Marco Roveri** (@marcoroveri) for pointing this out.
- SMT-LIB Parser:
  - Minor fixes highlighted by fuzzer (PR #376)
  - Fixed annotations parsing according to SMTLib rules (PR #374)
- **pysmt-install: Gracefully fail if GIT is not installed (PR #390)** Thanks to **Alberto Griggio** for reporting this.
- Removed dependency from internet connections when checking picosat version (PR #386)

2.3.2 0.6.0: 2016-10-09 – GMPY2 and Goodbye Recursion

BACKWARDS INCOMPATIBLE CHANGES:

- Integer, Fraction and Numerals are now defined in pysmt.constants (see below for details). The breaking changes are:
  - Users should use pysmt.constants.Fraction, if they want to guarantee that the same type is being used (different types are automatically converted);
  - Methods from pysmt.utils moved to pysmt.constants;
  - Numerals class was moved from pysmt.numeral (that does not exist anymore).
- Non-Recursive TreeWalker (PR #322)

  Modified TreeWalker to be non-recursive. The algorithm works by keeping an explicit stack of the walking functions **that are now required to be generators**. See pysmt.printer.HRPrinter for an example. This removes the last piece of recursion in pySMT !
- Times is now an n-ary operator (Issue #297 / PR #304)

  Functions operating on the args of Times (e.g., rewritings) should be adjusted accordingly.
- Simplified module pysmt.parsing into a unique file (PR #301)

  The pysmt.parsing module was originally divided in two files: pratt.py and parser.py. These files were removed and the parser combined into a unique parsing.py file. Code importing those modules directly needs to be updated.
- Use solver_options to specify solver-dependent options (PR #338):
  - MathSAT5Solver option 'debugFile' has been removed. Use the solver option: "debug_api_call_trace_filename".
  - BddSolver used to have the options as keyword arguments (static_ordering, dynamic_reordering etc). This is not supported anymore.
- Removed deprecated methods (PR #332):
  - FNode.get_dependencies (use FNode.get_free_variables)
  - FNode.get_sons (use FNode.get_args)
  - FNode.is_boolean_operator (use FNode.is_bool_op)
  - pysmt.test.skipIfNoSolverAvailable
  - pysmt.randomizer (not used and broken)

General:

- Support for GMPY2 to represent Fractions (PR #309).

  Usage of GMPY2 can be controlled by setting the env variable PYSMT_GMPY to True or False. By default, pySMT tries to use GMPY2 if installed, and fallbacks on Python's Fraction otherwise.
- Constants module: pysmt.constants (PR #309)

  This module provides an abstraction for constants Integer and Fraction, supporting different ways of representing them internally. Additionally, this module provides several utility methods:
  - is_pysmt_fraction
  - is_pysmt_integer
  - is_python_integer

  - is_python_rational
  - is_python_boolean

  Conversion can be achieved via:

  - pysmt_fraction_from_rational
  - pysmt_integer_from_integer
  - to_python_integer (handle long/int py2/py3 mismatch)

- Add Version information (Issue #299 / PR #303)
  - pysmt.VERSION : A tuple containing the version information
  - pysmt.__version__ : String representation of VERSION (following PEP 440)
  - pysmt.git_version : A simple function that returns the version including git information.

  install.py (pysmt-install) and shell.py gain a new –version option that uses git_version to display the version information.

- Shortcuts: read_smtlib() and write_smtlib()
- Docs: Completely Revised the documentation (PR #294)
- Rewritings: TimesDistributor (PR #302)

  Perform distributivity on an N-ary Times across addition and subtraction.

- SizeOracle: Add MEASURE_BOOL_DAG measure (PR #319)

  Measure the Boolean size of the formula. This is equivalent to replacing every theory expression with a fresh boolean variable, and measuring the DAG size of the formula. This can be used to estimate the Boolean complexity of the SMT formula.

- PYSMT_SOLVERS controls available solvers (Issue #266 / PR #316):

  Using the PYSMT_SOLVER system environment option, it is possible to restrict the set of installed solvers that are actually accessible to pySMT. For example, setting PYSMT_SOLVER="msat,z3" will limit the accessible solvers to msat and z3.

- Protect FNodeContent.payload access (Issue #291 / PR 310)

  All methods in FNode that access the payload now check that the FNode instance is of the correct type, e.g.:

  FNode.symbol_name() checks that FNode.is_symbol()

  This prevents from accessing the payload in a spurious way. Since this has an impact on every access to the payload, it has been implemented as an assertion, and can be disabled by running the interpreter with -O.

Solvers:

- Z3 Converter Improvements (PR #321):
  - Optimized Conversion to Z3 Solver Forward conversion is 4x faster, and 20% more memory efficient, because we work at a lower level of the Z3 Python API and do not create intermediate AstRef objects anymore. Back conversion is 2x faster because we use a direct dispatching method based on the Z3 OP type, instead of the big conditional that we were using previously.
  - Add back-conversion via SMT-LIB string buffer. Z3Converter.back_via_smtlib() performs back conversion by printing the formula as an SMT-LIB string, and parsing it back. For formulas of significant size, this can be drastically faster than using the API.
  - Extend back conversion to create new Symbols, if needed. This always raise a warning alerting the user that a new symbol is being implicitly defined.

- OSX: Z3 and MathSAT can be installed with pysmt-install (PR #244)
- MathSAT: Upgrade to 5.3.13 (PR #305)
- Yices: Upgrade to 2.5.1
- Better handling of solver options (PR #338):

  Solver constructor takes the optional dictionary `solver_options` of options that are solver dependent. It is thus possible to directly pass options to the underlying solver.

Bugfix:

- Fixed: Times back conversion in Z3 was binary not n-ary. Thanks to **Ahmed Irfan** for submitting the patch (PR #340, PR #341)
- Fixed: Bug in `array_value_assigned_values_map`, returning the incorrect values for an Array constant value. Thanks to **Daniel Ricardo dos Santos** for pointing this out and submitting the patch.
- Fixed: SMT-LIB define-fun serialization (PR #315)
- Issue #323: Parsing of variables named bvX (PR #326)
- Issue #292: Installers: Make dependency from pip optional (PR #300)
- Fixed: Bug in MathSAT's `get_unsat_core` (PR #331), that could lead to an unbounded mutual recursion. Thanks to **Ahmed Irfan** for reporting this (PR #331)

2.3.3 0.5.1: 2016-08-17 – NIRA and Python 3.5

Theories:

- Non Linear Arithmetic (NRA/NIA): Added support for non-linear, polynomial arithmetic. This thoery is currently supported only by Z3. (PR #282)
  - New operator POW and DIV
  - LIRA Solvers not supporting Non-Linear will raise the NonLinearError exception, while solvers not supporting arithmetics will raise a ConvertExpressionError exception (see test_nlira.py:test_unknownresult)
  - Algebraic solutions (e.g., sqrt(2) are represented using the internal z3 object – This is bound to change in the future.

General:

- Python 3.5: Full support for Python 3.5, all solvers are now tested (and working) on Python 3.5 (PR #287)
- Improved installed solvers check (install.py)
  - install.py –check now takes into account the bindings_dir and prints the version of the installed solver
  - Bindings are installed in different directories depending on the minor version of Python. In this way it is possible to use both Python 2.7 and 3.5.
  - There is a distinction btw installed solvers and solvers in the PYTHONPATH.
  - Qelim, Unsat-Core and Interpolants are also visualized (but not checked)
- Support for reading compressed SMT-LIB files (.bz2)
- Simplified HRPrinter code
- Removed six dependency from type_checker (PR #283)
- BddSimplifier (pysmt.simplifier.BddSimplifier): Uses BDDs to simplify the boolean structure of an SMT formula. (See test_simplify.py:test_bdd_simplify) (PR #286)

Solvers:

- Yices: New wrapper supporting python 3.5 (https://github.com/pysmt/yicespy)
- Yices: Upgrade to 2.4.2
- SMT-LIB Wrapper: Improved interaction with subprocess (#298)

Bugfix:

- Bugfix in Z3Converter.walk_array_value. Thanks to **Alberto Griggio** for contributing this patch
- Bugfix in DL Logic comparison (commit 9e9c8c)

2.3.4 0.5.0: 2016-06-09 – Arrays

BACKWARDS INCOMPATIBLE CHANGES:

- MGSubstituter becomes the new default substitution method (PR #253)

  When performing substitution with a mapping like `{a: b, Not(a), c}`, `Not(a)` is considered before `a`. The previous behavior (MSSubstituter) would have substituted `a` first, and then the rule for `Not(a)` would not have been applied.

- Removed argument `user_options` from Solver()

Theories:

- Added support for the Theory of Arrays.

  In addition to the SMT-LIB definition, we introduce the concept of Constant Array as supported by MathSAT and Z3. The theory is currently implemented for MathSAT, Z3, Boolector, CVC4.

  Thanks to **Alberto Griggio**, **Satya Uppalapati** and **Ahmed Irfan** for contributing through code and discussion to this feature.

General:

- Simplifier: Enable simplification if IFF with constant: e.g., (a <-> False) into !a
- Automatically enable Infix Notation by importing shortcuts.py (PR #267)
- SMT-LIB: support for define-sort commands without arguments
- Improved default options for shortcuts:
  - Factory.is_* sets model generation and incrementality to False;
  - Factory.get_model() sets model generation to True, and incrementality to False.
  - Factory.Solver() sets model generation and incrementality to True;
- Improved handling of options in Solvers (PR #250):

  Solver() takes `**options` as free keyword arguments. These options are checked by the class SolverOptions, in order to validate that these are meaningful options and perform a preliminary validation to catch typos etc. by raising a ValueError exception if the option is unknown.

  It is now possible to do: `Solver(name="bdd", dynamic_reordering=True)`

Solvers:

- rePyCUDD: Upgrade to 75fe055 (PR #262)
- CVC4: Upgrade to c15ff4 (PR #251)
- CVC4: Enabled Quantified logic (PR #252)

Bugfixes:

- Fixed bug in Non-linear theories comparison
- Fixed bug in reset behavior of CVC4
- Fixed bug in BTOR handling of bitwidth in shifts
- Fixed bug in BTOR's get_value function
- Fixed bug in BTOR, when operands did not have the same width after rewriting.

2.3.5 0.4.4: 2016-05-07 – Minor

General:

- BitVectors: Added support for infix notation
- Basic performance optimizations

Solvers:

- Boolector: Upgraded to version 2.2.0

Bugfix:

- Fixed bug in ExactlyOne args unpacking. Thanks to **Martin** @hastyboomalert for reporting this.

2.3.6 0.4.3: 2015-12-28 – Installers and HR Parsing

General:

- pysmt.parsing: Added parser for Human Readable expressions
- pysmt-install: new installer engine
- Most General Substitution: Introduced new Substituter, that performs top-down substitution. This will become the default in version 0.5.
- Improved compliance with SMT-LIB 2 and 2.5
- EagerModel can now take a solver model in input
- Introduce new exception 'UndefinedSymbolError' when trying to access a symbol that is not defined.
- Logic names can now be passed to shortcuts methods (e.g., is_sat) as a string

Solvers:

- MathSAT: Upgraded to version 5.3.9, including support for new detachable model feature. Thanks to **Alberto Griggio** for contributing this code.
- Yices: Upgraded to version 2.4.1
- Shannon: Quantifier Elimination based on shannon expansion (shannon).
- Improved handling of Context ('with' statement), exit and __del__ in Solvers.

Testing:

- Introduced decorator pysmt.test.skipIfNoSMTWrapper
- Tests do note explicitely depend anymore on unittest module. All tests that need to be executable only need to import pysmt.test.main.

Bugfix:

- #184: MathSAT: Handle UF with boolean args Fixed incorrect handling of UF with bool arguments when using MathSAT. The converter now takes care of rewriting the formula.
- #188: Auto-conversion of 0-ary functions to symbols
- #204: Improved quoting in SMT-LIB output
- Yices: Fixed a bug in push() method
- Fixed bug in Logic name dumping for SMT-LIB
- Fixed bug in Simplifier.walk_plus
- Fixed bug in CNF Converter (Thanks to Sergio Mover for pointing this out)

Examples:

- parallel.py: Shows how to use multi-processing to perform parallel and asynchronous solving
- smtlib.py: Demonstrates how to perform SMT-LIB parsing, dumping and extension
- einstein.py: Einstein Puzzle with example of debugging using UNSAT-Cores.

2.3.7 0.4.2: 2015-10-12 – Boolector

Solvers:

- Boolector 2.1.1 is now supported
- MathSAT: Updated to 5.3.8

General:

- EqualsOrIff: Introduced shortcut to handle equality and mismatch between theory and predicates atoms. This simply chooses what to use depending on the operands: Equals if Theory, Iff if predicates. Example usage in examples/all_smt.py
- Environment Extensibility: The global classes defined in the Environment can now be replaced. This makes it much easier for external tools to define new FNode types, and override default services.
- Parser Extensibility: Simplified extensibility of the parser by splitting the special-purpose code in the main loop in separate functions. This also adds support for escaping symbols when dealing with SMT-LIB.
- AUTO Logic: Factory methods default to logics.AUTO, providing a smarter selection of the logic depending on the formula being solved. This impacts all is_* functions, get_model, and qelim.
- Shell: Import BV32 and BVType by default, and enable infix notation
- Simplified HRPrinter
- Added AIG rewriting (rewritings.AIGer)

Bugfix:

- Fixed behavior of CNFizer.cnf_as_set()
- Fixed issue #159: error in parsing let bindings that refer to previous let-bound symbols. Thanks to *Alberto Griggio* for reporting it!

2.3.8 0.4.1: 2015-07-13 – BitVectors Extension

Theories:

- BitVectors: Added Signed operators

Solvers:

- Support for BitVectors added for Z3, CVC4, and Yices

General:

- SmartPrinting: Print expression by replacing sub-expression with custom strings.
- Moved global environment initialization to environment.py. Now internal functions do no need to import shortcuts.py anymore, thus breaking some circular dependencies.

Deprecation:

- Started deprecation of get_dependencies and get_sons
- Depreaced Randomizer and associated functions.

2.3.9 0.4.0: 2015-06-15 – Interpolation and BDDs

General:

- Craig interpolation support through Interpolator class, binary_interpolant and sequence_interpolant shortcuts. Current support is limited to MathSAT and Z3. Thanks to Alberto Griggio for implementing this!
- Rewriting functions: nnf-ization, prenex-normalization and disjunctive/conjunctive partitioning.
- get_implicant(): Returns the implicant of a satisfiable formula.
- Improved support for infix notation.
- Z3Model Iteration bugfix

BDDs:

- Switched from pycudd wrapper to a custom re-entrant version called repycudd (https://github.com/pysmt/repycudd)
- Added BDD-Based quantifier eliminator for BOOL theory
- Added support for static/dynamic variable ordering
- Re-implemented back-conversion avoiding recursion

2.3.10 0.3.0: 2015-05-01 – BitVectors/UnsatCores

Theories:

- Added initial support for BitVectors and QF_BV logic. Current support is limited to MathSAT and unsigned operators.

Solvers:

- Two new quantifier eliminators for LRA using MathSAT API: Fourier-Motzkin (msat_fm) and Loos-Weisspfenning (msat_lw)
- Yices: Improved handling of int/real precision

General:

- Unsat Cores: Unsat core extraction with dedicated shortcut get_unsat_core . Current support is limited to MathSAT and Z3
- Added support for Python 3. The library now works with both Python 2 and Python 3.

- QuantifierEliminator and qelim shortcuts, as well as the respective factory methods can now accept a 'logic' parameter that allows to select a quantifier eliminator instance supporting a given logic (analogously to what happens for solvers).
- Partial Model Support: Return a partial model whenever possible. Current support is limited to MathSAT and Z3.
- FNode.size(): Added method to compute the size of an expression using multiple metrics.

2.3.11 0.2.4: 2015-03-15 – PicoSAT

Solvers:

- PicoSAT solver support

General:

- Iterative implementation of FNode.get_free_variables(). This also deprecates FNode.get_dependencies().

Bugfix:

- Fixed bug (#48) in pypi package, making pysmt-install (and other commands) unavailable. Thanks to Rhishikesh Limaye for reporting this.

2.3.12 0.2.3: 2015-03-12 – Logics Refactoring

General:

- install.py: script to automate the installation of supported solvers.
- get_logic() Oracle: Detects the logic used in a formula. This can now be used in the shortcuts (_is_sat()_, _is_unsat()_, _is_valid()_, and _get_model()_) by choosing the special logic pysmt.logics.AUTO.
- Expressions: Added Min/Max operators.
- SMT-LIB: Substantially improved parser performances. Added explicit Annotations object to deal with SMT-LIB Annotations.
- Improved iteration methods on EagerModel

Backwards Incompatible Changes:

- The default logic for Factory.get_solver() is now the most generic *quantifier free* logic supported by pySMT (currently, QF_UFLIRA). The factory not provides a way to change this default.
- Removed option _quantified_ from all shortcuts.

2.3.13 0.2.2: 2015-02-07 – BDDs

Solvers:

- pyCUDD to perform BDD-based reasoning

General:

- Dynamic Walker Function: Dynamic Handlers for new node types can now be registered through the environment (see Environment.add_dynamic_walker_function).

2.3.14 0.2.1: 2014-11-29 – SMT-LIB

Solvers:

- Yices 2
- Generic Wrapper: enable usage of any SMT-LIB compatible solver.

General:

- SMT-LIB parsing
- Changed internal representation of FNode
- Multiple performance improvements
- Added configuration file

2.3.15 0.2.0: 2014-10-02 – Beta release.

Theories: LIRA Solvers: CVC4 General:

- Type-checking
- Definition of SMT-LIB logics
- Converted the DAGWalker from recursive to iterative
- Better handling of errors during formula creation and solving
- Preferences among available solvers.

Deprecation:

- Option 'quantified' within Solver() and all related methods will be removed in the next release.

Backwards Incompatible Changes:

- Renamed the module pysmt.types into pysmt.typing, to avoid conflicts with the Python Standard Library.

2.3.16 0.1.0: 2014-03-10 – Alpha release.

Theories: LIA, LRA, RDL, EUF Solvers: MathSAT, Z3 General Functionalities:

- Formula Manipulation: Creation, Simplification, Substitution, Printing
- Uniform Solving for QF formulae
- Unified Quantifier Elimination (Z3 support only)

2.3.17 0.0.1: 2014-02-01 – Initial release.

2.4 Developing in pySMT

2.4.1 Licensing

pySMT is distributed under the APACHE License (see LICENSE file). By submitting a contribution, you automatically accept the conditions described in LICENSE. Additionally, we ask you to certify that you have the right to submit such contributions. We adopt the "Developer Certificate of Origin" approach as done by the Linux kernel.

```
Developer Certificate of Origin Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 660
York Street, Suite 102, San Francisco, CA 94110 USA

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I have
    the right to submit it under the open source license indicated in
    the file; or

(b) The contribution is based upon previous work that, to the best of my
    knowledge, is covered under an appropriate open source license and I
    have the right under that license to submit that work with
    modifications, whether created in whole or in part by me, under the
    same open source license (unless I am permitted to submit under a
    different license), as indicated in the file; or

(c) The contribution was provided directly to me by some other person
    who certified (a), (b) or (c) and I have not modified it.

(d) I understand and agree that this project and the contribution are
    public and that a record of the contribution (including all personal
    information I submit with it, including my sign-off) is maintained
    indefinitely and may be redistributed consistent with this project
    or the open source license(s) involved.
```

During a Pull-Request you will be asked to complete the form at CLAHub: https://www.clahub.com/agreements/pysmt/pysmt . You will only have to complete this once, but this applies to **all** your contributions.

If you are doing a drive-by patch (e.g., fixing a typo) and sending directly a patch, you can skip the CLA, by sending a signed patch. A signed patch can be obtained when committing using `git commit -s`.

2.4.2 Tests

Running Tests

Tests in pySMT are developed using python's built-in testing framework *unittest*. Each *TestCase* is stored into a separate file, and it should be possible to launch it by calling the file directly, e.g.: `$ python test_formula.py`.

However, the preferred way is to use nosetests, e.g.: `$ nosetests pysmt/tests/test_formula.py`.

There are two utility scripts to simplify the testing of pysmt: `run_tests.sh` and `run_all_tests.sh`. They both exploit additional options for nosetests, such as parallelism and timeouts. `run_all_tests.sh` includes all the tests that are marked as `slow`, and therefore might take some time to complete.

Finally, tests are run across a wide range of solvers, versions of python and operating systems using Travis CI. This happens automatically when you open a PR. If you want to run this before submitting a PR, create a (free) Travis CI account, fork pySMT, and enable the testing from Travis webinterface.

All tests should pass for a PR to be merged.

Writing Tests

TestCase should inherit from *pysmt.test.TestCase*. This provides a default `SetUp()` for running tests in which the global environment is reset after each test is executed. This is necessary to avoid interaction between tests. Moreover, the class provides some additional assertions:

class `pysmt.test.`**`TestCase`**(*methodName='runTest'*)

Wrapper on the unittest TestCase class.

This class provides setUp and tearDown methods for pySMT in which a fresh environment is provided for each test.

`assertRaisesRegex`(*expected_exception*, *expected_regexp*, *callable_obj=None*, **args*, ***kwargs*)

Asserts that the message in a raised exception matches a regexp.

Args: expected_exception: Exception class expected to be raised. expected_regexp: Regexp (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. args: Extra args. kwargs: Extra kwargs.

`assertValid`(*formula*, *msg=None*, *solver_name=None*, *logic=None*)

Assert that formula is VALID.

`assertSat`(*formula*, *msg=None*, *solver_name=None*, *logic=None*)

Assert that formula is SAT.

`assertUnsat`(*formula*, *msg=None*, *solver_name=None*, *logic=None*)

Assert that formula is UNSAT.

PYSMT_SOLVER

The system environment variable `PYSMT_SOLVER` controls which solvers are actually available to pySMT. When developing it is common to have multiple solvers installed, but wanting to only test on few of them. For this reason `PYSMT_SOLVER` can be set to a list of solvers, e.g., `PYSMT_SOLVER="msat, z3"` will provide access to pySMT only to msat and z3, independently of which other solvers are installed. If the variable is unset or set to `all`, it does not have any effect.

2.4.3 How to add a new Theory within pySMT

In pySMT we are trying to closely follow the SMT-LIB standard. If the theory you want to add is already part of the standard, than many points below should be easy to answer.

1. **Identify the set of operators that need to be added** You need to distinguish between operators that are needed to represent the theory, and operators that are syntactic sugar. For example, in pySMT we have less-than and less-than-equal, as basic operators, and define greater-than and greater-than-equal as syntactic sugar.
2. **Identify which solvers support the theory** For each solver that supports the theory, it is important to identify which sub/super-set of the operators are supported and whether the solver is already integrated in pySMT. The goal of this activity is to identify possible incompatibilities in the way different solvers deal with the theory.
3. **Identify examples in "SMT-LIB" format** This provides a simple way of looking at how the theory is used, and access to cheap tests.

Once these points are clear, please open an issue on github with the answer to these points and a bit of motivation for the theory. In this way we can discuss possible changes and ideas before you start working on the code.

Code for a new Theory

A good example of theory extension is represented by the BitVector theory. In case of doubt, look at how the BitVector case (bv) has been handled.

Adding a Theory to the codebase is done by following these steps:

1. Tests: Add a test file `pysmt/test/test_<theory>.py`, to demonstrate the use for the theory (e.g., `pysmt/test/test_bv.py`).
2. Operators: Add the (basic) operators in `pysmt/operators.py`, create a constant for each operator, and extend the relevant structures.
3. Typing: Extend `pysmt/typing.py` to include the types (sorts) of the new theory.
4. Walker: Extend `pysmt/walkers/generic.py` to include one `walk_` function for each of the basic operators.
5. FNode: Extend `is_*` methods in `pysmt/fnode.py:FNode`. This makes it possible to check the type of an expression, obtaining additional elements (e.g., width of a bitvector constant).
6. Typechecker: Extend `pysmt/type_checker.py:SimpleTypeChecker` to include type-checking rules.
7. FormulaManager: Create constructor for all operators, including syntactic sugar, in `pysmt/formula.py:FormulaManager`.

At this point you are able to build expressions in the new theory. This is a good time to start running your tests.

8. Printers: Extend `pysmt/printers.py:HRPrinter` to be able to print expressions in the new theory (you might need to do this earlier, if you need to debug your tests output).
9. Examples: Extend `pysmt/test/examples.py` with at least one example formula for each new operator defined in FormulaManager. These examples are used in many tests, and will help you identify parts of the system that still need to be extended (e.g., Simplifier).
10. Theories and Logics: Extend `pysmt/logics.py` to include the new Theory and possible logics that derive from this Theory. In particular, define logics for each theory combination that makes sense.
11. SMT-LIB: Extend `pysmt/smtlib/parser.py:SmtLibParser` and `pysmt/smtlib/printers.py` to support the new operators.
12. Shortcuts: All methods that were added in FormulaManager need to be available in `pysmt/shortcuts.py`.

At this point all pySMT tests should pass. This might require extending other walkers to support the new operators.

13. Solver: Extend at least one solver to support the Logic. This is done by extending the associated Converter (e.g., `pysmt/solvers/msat.py:MSatConverter`) and adding at least one logic to its `LOGICS` field. As a bare-minimum, this will require a way of converting solvers-constants back into pySMT constants (`Converter.back()`).

2.4.4 Packaging and Distributing PySMT

The setup.py script can be used to create packages. The command

```
python setup.py bdist --format=gztar
```

will produce a tar.gz file inside the `dist/` directory.

For convenience the script `make_distrib.sh` is provided, this builds both the binary and source distributions within `dist/`.

2.4.5 Building Documentation

pySMT uses Sphinx for documentation. To build the documentation you will need Sphinx installed, this can be done via pip.

A Makefile in the `docs/` directory allows to build the documentation in many formats. Among them, we usually consider html and latex.

2.4.6 Preparing a Release (Check-List)

In order to make a release, the master branch must pass all tests on the CI (Travis and Appveyor). The release process is broken into the following steps:

- OSX Testing
- Release branch creation
- Changelog update
- Version change
- Package creation and local testing
- Merge and Tag
- PyPi update
- Version Bumping
- Announcement

OSX Testing

The `master` branch is merge within `travix/macosx`. Upon pushing this branch, Travis CI will run the tests on OSX platform. In this way, we know that pySMT works on all supported platforms.

Release Branch Creation

As all other activities, also the creation of a release requires working on a separate branch. This makes it possible to interrupt, share, and resume the release creation, if bugs are discovered during this process. The branch must be called `rc/a.b.c`, where a.b.c is the version number of the target release.

Changelog Update (docs/CHANGES.rst)

Use `git log` to obtain the full list of commits from the latest tag. We use merge commits to structure the Changelog, however, sometimes additional and useful information is described in intermediate commits, and it is thus useful to have them.

The format of the header is `<version>: <year> -- <Title>`, where version has the format Major.Minor.Patch (e.g., 0.6.1) and year is in ISO format: YYYY-MM-DD (e.g., 2016-11-28). The title should be brief and possible include the highlights of the release.

The body of the changelog should start with the backwards incompatible changes with a prominent header. The other sections (optional if nothing changed) are:

- General: For new features of pySMT
- Solvers: For upgrades or improvements to the solvers

- Theories: For new or improved Theories
- Bugfix: For all the fixes that do not constitute a new feature

Each item in the lists ends with reference to the Github issue or Pull request. If an item deserves more explanation and it is not associated with an issue or PR, it is acceptable to point to the exact commit id). Items should also acknowledge contributors for submitting patches, opening tickets or simply discussing a problem.

Version change

The variable `VERSION` in `pysmt/__init__.py` must be modified to show the correct version number: e.g., `VERSION = (0, 6, 1)`.

Package creation and local testing

The utility script `make_distrib.sh` to create a distribution package is located in the root directory of the project. This will create various formats, and download the latest version of six.

After running this script, the package `dist/PySMT-a.b.c.tar.gz` (where a.b.c are the release number), needs to be uploaded to pypi. Before doing so, however, we test it locally, to make sure that everything works. The most common mistake in this phase is the omission of a file in the package.

To test the package, we create a new hardcopy of the tests of pySMT:

0. `mkdir -p test_pkg/pysmt`
1. `cp -a github/pysmt/test test_pkg/pysmt/; cd test_pkg`
2. This should fail: `nosetests -v pysmt`
3. `pip install --user github/dist/PySMT-a.b.c.tar.gz`
4. `nosetests -v pysmt`
5. `pip uninstall pysmt`

All tests should pass in order to make the release. Note: It is enough to have one solver installed, in order to test the package. The type of issues that might occur during package creation are usually independent of the solver.

Merge and Tag

At this point we have created and tested the release, we can merge the `rc/` branch back into master, and tag the release with: `git tag -a va.b.c` (note the `v` before the major version number), and finally push the tag to github `git push origin va.b.c`.

Now on github, it is possible to create the release associated with this tag. The description of the release is the copy-paste of the Changelog. Additionally, we include the wheel file (remember to include six!) and the tar.gz .

Immediately after tagging, make a commit on master bumping the version. By default we use `(a, b, c+1, "dev", 1)`.

PyPi update

```
twine upload PySMT-a.b.c.tar.gz
```

TODO: Figure out how to have shared credentials for pypi. Currently, only marcogario has upload privileges.

Announcement

- Mailing list: https://groups.google.com/forum/#!forum/pysmt
- Make sure the Github Release has been created

2.5 API Reference

2.5.1 Shortcuts

Provides the most used functions in a nicely wrapped API.

This module defines a global environment, so that most methods can be called without the need to specify an environment or a FormulaManager. Functions trying to access the global environment should use the method get_env(). Keep in mind that the global state of the environment might lead to inconsistency and unexpected bugs. This is particularly true for tests. For tests it is recommended to perform an environment reset in the setUp phase, to be guaranteed that a fresh environment is used (this is the default behavior of `pysmt.test.TestCase`).

pysmt.shortcuts.**get_env**()
Returns the global environment.

Returns The global environment

Return type *Environment*

pysmt.shortcuts.**reset_env**()
Resets the global environment, and returns the new one.

Returns A new environment after resetting the global environment

Return type *Environment*

pysmt.shortcuts.**get_type**(*formula*)
Returns the type of the formula.

Parameters **formula** (FNode) – The target formula

Returns The type of the formula

pysmt.shortcuts.**simplify**(*formula*)
Returns the simplified version of the formula.

Parameters **formula** (FNode) – The target formula

Returns The simplified version of the formula

Return type Fnode

pysmt.shortcuts.**substitute**(*formula*, *subs*)
Applies the substitutions defined in the dictionary to the formula.

Parameters

- **formula** (FNode) – The target formula
- **subs** (*A dictionary from FNode to FNode*) – Specify the substitutions to apply to the formula

Returns Formula after applying the substitutions

Return type Fnode

pysmt.shortcuts.**serialize**(*formula*, *threshold=None*)
Provides a string representing the formula.

Parameters

- **formula** (*Integer*) – The target formula
- **threshold** – Specify the threshold

Returns A string representing the formula

Return type string

pysmt.shortcuts.**get_free_variables**(*formula*)
Returns the free variables of the formula.

Parameters **formula** (FNode) – The target formula

Returns Free variables in the formula

pysmt.shortcuts.**get_atoms**(*formula*)
Returns the set of atoms of the formula.

Parameters **formula** (FNode) – The target formula

Returns the set of atoms of the formula

pysmt.shortcuts.**get_formula_size**(*formula*, *measure=None*)
Returns the size of the formula as measured by the given counting type.

See pysmt.oracles.SizeOracle for details.

Parameters

- **formula** (FNode) – The target formula
- **measure** – Specify the measure/counting type

Returns The size of the formula as measured by the given counting type.

pysmt.shortcuts.**ForAll**(*variables*, *formula*)

$$\forall v_1, \cdots, v_n . \varphi(v_1, \cdots, v_n)$$

pysmt.shortcuts.**Exists**(*variables*, *formula*)

$$\exists v_1, \cdots, v_n . \varphi(v_1, \cdots, v_n)$$

pysmt.shortcuts.**Function**(*vname*, *params*)

$$vname(p_1, \cdots, p_n)$$

pysmt.shortcuts.**Not**(*formula*)

$$\neg\varphi$$

pysmt.shortcuts.**Implies**(*left*, *right*)

$$l \rightarrow r$$

pysmt.shortcuts.**Iff**(*left*, *right*)

$$l \leftrightarrow r$$

pysmt.shortcuts.**GE**(*left*, *right*)

$$l \geq r$$

pysmt.shortcuts.**Minus**(*left*, *right*)

$$l - r$$

pysmt.shortcuts.**Times**(**args*)

$$x_1 \times x_2 \cdots \times x_n$$

pysmt.shortcuts.**Pow**(*left*, *right*)

$$l^r$$

pysmt.shortcuts.**Div**(*left*, *right*)

$$\frac{l}{r}$$

pysmt.shortcuts.**Equals**(*left*, *right*)

$$l = r$$

pysmt.shortcuts.**GT**(*left*, *right*)

$$l > r$$

pysmt.shortcuts.**LE**(*left*, *right*)

$$l \leq r$$

pysmt.shortcuts.**LT**(*left*, *right*)

$$l < r$$

pysmt.shortcuts.**Ite**(*iff*, *left*, *right*)

$$\text{If } i \text{ Then } l \text{ Else } r$$

pysmt.shortcuts.**Symbol**(*name*, *typename=BooleanType*)
Returns a symbol with the given name and type.

Parameters

- **name** – Specify the name
- **typename** – Specify the typename

Returns A symbol with the given name and type

pysmt.shortcuts.**FreshSymbol**(*typename=BooleanType*, *template=None*)
Returns a symbol with a fresh name and given type.

Parameters

- **typename** – Specify the typename
- **template** – Specify the template

Returns A symbol with a fresh name and a given type

pysmt.shortcuts.**Int**(*value*)
Returns an Integer constant with the given value.

Parameters **value** – Specify the value

Returns An Integer constant with the given value

pysmt.shortcuts.**Bool**(*value*)
Returns a Boolean constant with the given value.

Parameters **value** – Specify the value

Returns A Boolean constant with the given value

pysmt.shortcuts.**Real**(*value*)
Returns a Real constant with the given value.

Parameters **value** – Specify the value

Returns A Real constant with the given value

pysmt.shortcuts.**TRUE**()
Returns the Boolean constant TRUE.

returns: The Boolean constant TRUE

pysmt.shortcuts.**FALSE**()
Returns the Boolean constant FALSE.

returns: The Boolean constant FALSE

pysmt.shortcuts.**And**(**args*)

$$\varphi_0 \wedge \cdots \wedge \varphi_n$$

pysmt.shortcuts.**Or**(**args*)

$$\varphi_0 \vee \cdots \vee \varphi_n$$

pysmt.shortcuts.**Plus**(**args*)

$$\varphi_0 + \cdots + \varphi_n$$

pysmt.shortcuts.**ToReal**(*formula*)
Explicit cast of a term into a Real term.

pysmt.shortcuts.**AtMostOne**(**args*)
At most one can be true at anytime.

Cardinality constraint over a set of boolean expressions.

pysmt.shortcuts.**ExactlyOne**(**args*)
Given a set of boolean expressions requires that exactly one holds.

pysmt.shortcuts.**AllDifferent**(**args*)
Given a set of non-boolean expressions, requires that each of them has value different from all the others

pysmt.shortcuts.**Xor**(*left*, *right*)
Returns the XOR of left and right

Parameters

- **left** (FNode) – Specify the left BV
- **right** (FNode) – Specify the right BV

Returns The XOR of left and right

pysmt.shortcuts.**Min**(**args*)
Minimum over a set of real or integer terms.

pysmt.shortcuts.**Max**(**args*)
Maximum over a set of real or integer terms

pysmt.shortcuts.**EqualsOrIff**(*left*, *right*)
Returns Equals() or Iff() depending on the type of the arguments.

This can be used to deal with ambiguous cases where we might be dealing with both Theory and Boolean atoms.

pysmt.shortcuts.**BV**(*value*, *width=None*)
Returns a constant of type BitVector.

value can be either: - a string of 0s and 1s - a string starting with "#b" followed by a sequence of 0s and 1s - an integer number s.t. 0 <= value < 2**width

In order to create the BV representation of a signed integer, the SBV() method shall be used.

Parameters

- **value** – Specify the value
- **width** – Specify the width

Returns A constant of type BitVector

Return type *FNode*

pysmt.shortcuts.**SBV**(*value*, *width=None*)
Returns a constant of type BitVector interpreting the sign.

If the specified value is an integer, it is converted in the 2-complement representation of the given number, otherwise the behavior is the same as BV().

Parameters

- **value** – Specify the value
- **width** – Specify the width of the BV

Returns A constant of type BitVector interpreting the sign.

Return type *FNode*

pysmt.shortcuts.**BVOne**(*width=None*)
Returns the unsigned one constant BitVector.

Parameters **width** – Specify the width of the BitVector

Returns The unsigned one constant BitVector

Return type *FNode*

pysmt.shortcuts.**BVZero**(*width=None*)
Returns the zero constant BitVector.

Parameters **width** – Specify the width of the BitVector

Returns The unsigned zero constant BitVector

Return type *FNode*

pysmt.shortcuts.**BVNot**(*formula*)
Returns the bitwise negation of the bitvector

Parameters **formula** – The target formula

Returns The bitvector Not(bv)

Return type *FNode*

pysmt.shortcuts.**BVAnd**(*left*, *right*)
Returns the Bit-wise AND of two bitvectors of the same size.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The bit-wise AND of left and right

Return type *FNode*

pysmt.shortcuts.**BVOr**(*left*, *right*)
Returns the Bit-wise OR of two bitvectors of the same size.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The bit-wise OR of left and right

Return type *FNode*

pysmt.shortcuts.**BVXor**(*left*, *right*)
Returns the Bit-wise XOR of two bitvectors of the same size.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The bit-wise XOR of left and right

Return type *FNode*

pysmt.shortcuts.**BVConcat**(*left*, *right*)
Returns the Concatenation of the two BVs

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The concatenation of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVExtract**(*formula*, *start=0*, *end=None*)
Returns the slice of formula from start to end (inclusive).

Parameters

- **formula** – The target formula
- **start** – Specify the start index
- **end** – Specify the end index

Returns The slice of formula from start to end (inclusive)

Return type Fnode

pysmt.shortcuts.**BVULT**(*left*, *right*)
Returns the Unsigned Less-Than comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned Less-Than comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVUGT**(*left*, *right*)
Returns the Unsigned Greater-Than comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned Greater-Than comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVULE**(*left*, *right*)
Returns the Unsigned Less-Equal comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned Less-Equal comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVUGE**(*left*, *right*)
Returns the Unsigned Greater-Equal comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned Greater-Equal comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVNeg**(*formula*)
Returns the arithmetic negation of the BV.

Parameters **formula** – The target formula

Returns The arithmetic negation of the formula

Return type *FNode*

pysmt.shortcuts.**BVAdd**(*left*, *right*)
Returns the sum of two BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The sum of the two BVs.

Return type *FNode*

pysmt.shortcuts.**BVSub**(*left*, *right*)
Returns the difference of two BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The difference of the two BV

Return type *FNode*

pysmt.shortcuts.**BVMul**(*left*, *right*)
Returns the product of two BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The product of the two BV

Return type *FNode*

pysmt.shortcuts.**BVUDiv**(*left*, *right*)
Returns the Unsigned division of the two BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned division of the two BV

Return type *FNode*

pysmt.shortcuts.**BVURem**(*left*, *right*)
Returns the Unsigned remainder of the two BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Unsigned remainder of the two BV

Return type *FNode*

pysmt.shortcuts.**BVLShl**(*left*, *right*)
Returns the logical left shift the BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The logical left shift the BV

Return type *FNode*

pysmt.shortcuts.**BVLShr**(*left*, *right*)
Returns the logical right shift the BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The logical right shift the BV

Return type *FNode*

pysmt.shortcuts.**BVAShr**(*left*, *right*)

Returns the RIGHT arithmetic rotation of the left BV by the number of steps specified by the right BV.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The RIGHT arithmetic rotation of the left BV by the number of steps specified by the right BV

Return type *FNode*

pysmt.shortcuts.**BVRol**(*formula*, *steps*)
Returns the LEFT rotation of the BV by the number of steps.

Parameters

- **formula** – The target formula
- **steps** – Specify the number of steps.

Returns The LEFT rotation of the BV by the number of steps

Return type *FNode*

pysmt.shortcuts.**BVRor**(*formula*, *steps*)
Returns the RIGHT rotation of the BV by the number of steps.

Parameters

- **formula** – The target formula
- **steps** – Specify the number of steps.

Returns The RIGHT rotation of the BV by the number of steps

Return type *FNode*

pysmt.shortcuts.**BVZExt**(*formula*, *increase*)
Returns the zero-extension of the BV.

New bits are set to zero.

Parameters

- **formula** – The target formula
- **increase** – Specify the increase

Returns The extension of the BV

Return type *FNode*

pysmt.shortcuts.**BVSExt**(*formula*, *increase*)
Returns the signed-extension of the BV.

New bits are set according to the most-significant-bit.

Parameters

- **formula** – The target formula
- **increase** – Specify the 'increase' value

Returns The signed-extension of the BV.

Return type *FNode*

pysmt.shortcuts.**BVSLT**(*left*, *right*)
Returns the Signed Less-Than comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Signed Less-Than comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVSLE**(*left*, *right*)
Returns the Signed Less-Equal comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Signed Less-Than-Equal comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVSGT**(*left*, *right*)
Returns the Signed Greater-Than comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Signed Greater-Than comparison of the two BVs

Return type *FNode*

pysmt.shortcuts.**BVSGE**(*left*, *right*)
Returns the Signed Greater-Equal comparison of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Signed Greater-Equal comparison of the two BVs.

Return type *FNode*

pysmt.shortcuts.**BVSDiv**(*left*, *right*)
Returns the Signed division of the two BVs.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The the Signed division of left by right

Return type *FNode*

pysmt.shortcuts.**BVSRem**(*left*, *right*)
Returns the Signed remainder of the two BVs

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns The Signed remainder of left divided by right

Return type *FNode*

pysmt.shortcuts.**BVComp**(*left*, *right*)

Returns a BV of size 1 equal to 0 if left is equal to right, otherwise equal to 1.

Parameters

- **left** – Specify the left bitvector
- **right** – Specify the right bitvector

Returns A BV of size 1 equal to 0 if left is equal to right, otherwise 1

Return type *FNode*

pysmt.shortcuts.**Select**(*array*, *index*)
Returns a SELECT application on the array at the given index

Parameters

- **array** – Specify the array
- **index** – Specify the index

Returns A SELECT application on array at index

Return type *FNode*

pysmt.shortcuts.**Store**(*array*, *index*, *value*)
Returns a STORE application with given value on array at the given index

Parameters

- **array** – Specify the array
- **index** – Specify the index

Returns A STORE on the array at the given index with the given value

Return type *FNode*

pysmt.shortcuts.**Array**(*idx_type*, *default*, *assigned_values=None*)
Returns an Array with the given index type and initialization.

If assigned_values is specified, then it must be a map from constants of type idx_type to values of the same type as default and the array is initialized correspondingly.

Parameters

- **idx_type** – Specify the index type
- **default** – Specify the default values
- **assigned_values** – Specify the assigned values

Returns A node representing an array having index type equal to idx_type, initialized with default values. If assigned_values is specified, then it must be a map from constants of type idx_type to values of the same type as default and the array is initialized correspondingly.

Return type *FNode*

pysmt.shortcuts.**Solver**(*quantified=False*, *name=None*, *logic=None*, ***kwargs*)
Returns a solver.

Parameters

- **quantified** (*bool*) – Specify if the solver is quantified
- **name** – Specify the name of the solver
- **logic** – Specify the logic that is going to be used.

Return type *Solver*

pysmt.shortcuts.**UnsatCoreSolver**(*quantified=False*, *name=None*, *logic=None*, *unsat_cores_mode='all'*)

Returns a solver supporting unsat core extraction.

Parameters

- **quantified** – Specify if the solver is quantified.
- **name** – Specify the name of the solver
- **logic** – Specify the logic that is going to be used.
- **unsat_cores_mode** – Specify the unsat cores mode.

Returns A solver supporting unsat core extraction.

Return type *Solver*

pysmt.shortcuts.**QuantifierEliminator**(*name=None*, *logic=None*)

Returns a quantifier eliminator.

Parameters

- **name** – Specify the name of the solver
- **logic** – Specify the logic that is going to be used.

Returns A quantifier eliminator with the specified name and logic

Return type *QuantifierEliminator*

pysmt.shortcuts.**Interpolator**(*name=None*, *logic=None*)

Returns an interpolator

Parameters

- **name** – Specify the name of the solver
- **logic** – Specify the logic that is going to be used.

Returns An interpolator

Return type *Interpolator*

pysmt.shortcuts.**is_sat**(*formula*, *solver_name=None*, *logic=None*, *portfolio=None*)

Returns whether a formula is satisfiable.

Parameters

- **formula** (FNode) – The formula to check satisfiability
- **solver_name** (*string*) – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used
- **portfolio** (*An iterable of solver names*) – A list of solver names to perform portfolio solving.

Returns Whether the formula is SAT or UNSAT.

Return type bool

pysmt.shortcuts.**get_model**(*formula*, *solver_name=None*, *logic=None*)

Similar to *is_sat()* but returns a model if the formula is satisfiable, otherwise None

Parameters

- **formula** – The target formula
- **solver_name** – Specify the name of the solver

Param logic: Specify the logic that is going to be used

Returns A model if the formula is satisfiable

Return type *Model*

pysmt.shortcuts.**get_implicant**(*formula*, *solver_name=None*, *logic=None*)
Returns a formula f_i such that Implies(f_i, formula) is valid or None if formula is unsatisfiable.

if complete is set to true, all the variables appearing in the formula are forced to appear in f_i. :param formula: The target formula :param solver_name: Specify the name of the solver :param: logic: Specify the logic that is going to be used :returns: A formula f_i such that Implies(f_i, formula) is valid or None

if formula is unsatisfiable.

Return type *FNode*

pysmt.shortcuts.**get_unsat_core**(*clauses*, *solver_name=None*, *logic=None*)
Similar to *get_model()* but returns the unsat core of the conjunction of the input clauses

Parameters

- **clauses** – Specify the list of input clauses
- **solver_name** – Specify the name of the solver_name
- **logic** – Specify the logic that is going to be used

Returns The unsat core of the conjunction of the input clauses

pysmt.shortcuts.**is_valid**(*formula*, *solver_name=None*, *logic=None*, *portfolio=None*)
Similar to *is_sat()* but checks validity.

Parameters

- **formula** (FNode) – The target formula
- **solver_name** – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used
- **portfolio** – A list of solver names to perform portfolio solving.

Returns Whether the formula is SAT or UNSAT but checks validity

Return type bool

pysmt.shortcuts.**is_unsat**(*formula*, *solver_name=None*, *logic=None*, *portfolio=None*)
Similar to *is_sat()* but checks unsatisfiability.

Parameters

- **formula** (FNode) – The target formula
- **solver_name** – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used
- **portfolio** – A list of solver names to perform portfolio solving.

Returns Whether the formula is UNSAT or not

Return type bool

pysmt.shortcuts.**qelim**(*formula*, *solver_name=None*, *logic=None*)
Performs quantifier elimination of the given formula.

Parameters

- **formula** – The target formula
- **solver_name** – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used

Returns A formula after performing quantifier elimination

Return type *FNode*

pysmt.shortcuts.**binary_interpolant**(*formula_a*, *formula_b*, *solver_name=None*, *logic=None*)
Computes an interpolant of (formula_a, formula_b).

Returns None if the conjunction is satisfiable

Parameters

- **formula_a** – Specify formula_a
- **formula_b** – Specify formula_b
- **solver_name** – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used

Returns An interpolant of (formula_a, formula_b); None if the conjunction is satisfiable

Return type FNode or None

pysmt.shortcuts.**sequence_interpolant**(*formulas*, *solver_name=None*, *logic=None*)
Computes a sequence interpolant of the formulas.

Returns None if the conjunction is satisfiable.

Parameters

- **formulas** – The target formulas
- **solver_name** – Specify the name of the solver to be used
- **logic** – Specify the logic that is going to be used

Returns A sequence intepolant of the formulas; None if the conjunction is satisfiable

Return type FNode or None

pysmt.shortcuts.**read_configuration**(*config_filename*, *environment=None*)
Reads the pysmt configuration of the given file path and applies it on the specified environment. If no environment is specified, the top-level environment will be used.

Parameters

- **config_filename** – Specify the name of the config file
- **environment** – Specify the environment

pysmt.shortcuts.**write_configuration**(*config_filename*, *environment=None*)
Dumps the current pysmt configuration to the specified file path

Parameters

- **config_filename** – Specify the name of the config file
- **environment** – Specify the environment

pysmt.shortcuts.**read_smtlib**(*fname*)
Reads the SMT formula from the given file.

This supports compressed files, if the fname ends in .bz2 .

Parameters **fname** – Specify the filename

Returns An SMT formula

Return type *FNode*

pysmt.shortcuts.**write_smtlib**(*formula*, *fname*)
Reads the SMT formula from the given file.

Parameters

- **formula** – Specify the SMT formula to look for
- **fname** – Specify the filename

2.5.2 Solver, Model, QuantifierEliminator, Interpolator, and UnsatCoreSolver

class pysmt.solvers.solver.**Solver**(*environment*, *logic*, ***options*)
Represents a generic SMT Solver.

OptionsClass
alias of SolverOptions

get_model()
Returns an instance of Model that survives the solver instance.

Restrictions: Requires option generate_models to be set to true (default) and can be called only after solve() (or one of the derived methods) returned sat or unknown, if no change to the assertion set occurred.

is_sat(*formula*)
Checks satisfiability of the formula w.r.t. the current state of the solver.

Previous assertions are taken into account.

Returns Whether formula is satisfiable

Return type bool

is_valid(*formula*)
Checks validity of the formula w.r.t. the current state of the solver.

Previous assertions are taken into account. See *is_sat()*

Returns Whether formula is valid

Return type bool

is_unsat(*formula*)
Checks unsatisfiability of the formula w.r.t. the current state of the solver.

Previous assertions are taken into account. See *is_sat()*

Returns Whether formula is unsatisfiable

Return type bool

get_values(*formulae*)
Returns the value of the expressions if a model was found.

Requires option generate_models to be set to true (default) and can be called only after solve() (or to one of the derived methods) returned sat or unknown, if no change to the assertion set occurred.

Returns A dictionary associating to each expr a value

Return type dict

push(*levels=1*)
Push the current context of the given number of levels.

pop(*levels=1*)
Pop the context of the given number of levels.

exit()
Exits from the solver and closes associated resources.

reset_assertions()
Removes all defined assertions.

add_assertion(*formula*, *named=None*)
Add assertion to the solver.

solve(*assumptions=None*)
Returns the satisfiability value of the asserted formulas.

Assumptions is a list of Boolean variables or negations of boolean variables. If assumptions is specified, the satisfiability result is computed assuming that all the specified literals are True.

A call to solve([a1, ..., an]) is functionally equivalent to:

push() add_assertion(And(a1, ..., an)) res = solve() pop() return res

but is in general more efficient.

print_model(*name_filter=None*)
Prints the model (if one exists).

An optional function can be passed, that will be called on each symbol to decide whether to print it.

get_value(*formula*)
Returns the value of formula in the current model (if one exists).

This is a simplified version of the SMT-LIB function get_values

get_py_value(*formula*)
Returns the value of formula as a python type.

E.g., Bool(True) is translated into True. This simplifies writing code that branches on values in the model.

get_py_values(*formulae*)
Returns the values of the formulae as python types.

Returns a dictionary mapping each formula to its python value.

class pysmt.solvers.solver.**Model**(*environment*)
An abstract Model for a Solver.

This class provides basic services to operate on a model returned by a solver. This class is used as superclass for more specific Models, that are solver dependent or by the EagerModel class.

get_value(*formula*, *model_completion=True*)
Returns the value of formula in the current model (if one exists).

If model_completion is True, then variables not appearing in the assignment are given a default value, otherwise an error is generated.

This is a simplified version of the SMT-LIB funtion get_values .

get_values(*formulae*, *model_completion=True*)
Evaluates the values of the formulae in the current model.

Evaluates the values of the formulae in the current model returning a dictionary.

get_py_value(*formula*, *model_completion=True*)
Returns the value of formula as a python type.

E.g., Bool(True) is translated into True. This simplifies writing code that branches on values in the model.

get_py_values(*formulae*, *model_completion=True*)
Returns the values of the formulae as python types.

Returns the values of the formulae as python types. in the current model returning a dictionary.

converter
Get the Converter associated with the Solver.

class pysmt.solvers.qelim.**QuantifierEliminator**

eliminate_quantifiers(*formula*)
Returns a quantifier-free equivalent formula of the given formula

If explicit_vars is specified, an explicit enumeration of all the possible models for such variables is computed and quantifier elimination is performed on each disjunct separately.

exit()
Destroys the solver and closes associated resources.

class pysmt.solvers.interpolation.**Interpolator**

binary_interpolant(*a*, *b*)
Returns a binary interpolant for the pair (a, b), if And(a, b) is unsatisfaiable, or None if And(a, b) is satisfiable.

sequence_interpolant(*formulas*)
Returns a sequence interpolant for the conjunction of formulas, or None if the problem is satisfiable.

exit()
Destroys the solver and closes associated resources.

class pysmt.solvers.solver.**UnsatCoreSolver**
A solver supporting unsat core extraction

get_unsat_core()
Returns the unsat core as a set of formulae.

After a call to solve() yielding UNSAT, returns the unsat core as a set of formulae

get_named_unsat_core()
Returns the unsat core as a dict of names to formulae.

After a call to solve() yielding UNSAT, returns the unsat core as a dict of names to formulae

2.5.3 Environment

The Environment is a key structure in pySMT. It contains multiple singleton objects that are used throughout the system, such as the FormulaManager, Simplifier, HRSerializer, SimpleTypeChecker.

class pysmt.environment.**Environment**

The Environment provides global singleton instances of various objects.

FormulaManager and the TypeChecker are among the most commonly used ones.

Subclasses of Environment should take care of adjusting the list of classes for the different services, by changing the class attributes.

FormulaManagerClass
alias of FormulaManager

SimplifierClass
alias of Simplifier

SubstituterClass
alias of MGSubstituter

HRSerializerClass
alias of HRSerializer

SizeOracleClass
alias of SizeOracle

AtomsOracleClass
alias of AtomsOracle

stc
Get the Simple Type Checker

qfo
Get the Quantifier Oracle

ao
Get the Atoms Oracle

theoryo
Get the Theory Oracle

fvo
Get the FreeVars Oracle

sizeo
Get the Size Oracle

add_dynamic_walker_function(*nodetype*, *walker*, *function*)
Dynamically bind the given function to the walker for the nodetype.

This function enables the extension of walkers for new nodetypes. When introducing a new nodetype, we link a new function to a given walker, so that the walker will be able to handle the new nodetype.

See pysmt.walkers.generic.Walker.walk_error() for more information.

pysmt.environment.**get_env**()
Returns the Environment at the head of the stack.

pysmt.environment.**push_env**(*env=None*)
Push a env in the stack. If env is None, a new Environment is created.

pysmt.environment.**pop_env**()
Pop an env from the stack.

pysmt.environment.**reset_env**()
Destroys and recreate the head environment.

2.5.4 Exceptions

This module contains all custom exceptions of pySMT.

exception pysmt.exceptions.**PysmtException**
Base class for all custom exceptions of pySMT

exception pysmt.exceptions.**UnknownSmtLibCommandError**
Raised when the parser finds an unknown command.

exception pysmt.exceptions.**SolverReturnedUnknownResultError**
This exception is raised if a solver returns 'unknown' as a result

exception pysmt.exceptions.**UnknownSolverAnswerError**
Raised when the a solver returns an invalid response.

exception pysmt.exceptions.**NoSolverAvailableError**
No solver is available for the selected Logic.

exception pysmt.exceptions.**NonLinearError**
The provided expression is not linear.

exception pysmt.exceptions.**UndefinedLogicError**
This exception is raised if an undefined Logic is attempted to be used.

exception pysmt.exceptions.**InternalSolverError**
Generic exception to capture errors provided by a solver.

exception pysmt.exceptions.**NoLogicAvailableError**
Generic exception to capture errors caused by missing support for logics.

exception pysmt.exceptions.**SolverRedefinitionError**
Exception representing errors caused by multiple defintion of solvers having the same name.

exception pysmt.exceptions.**SolverNotConfiguredForUnsatCoresError**
Exception raised if a solver not configured for generating unsat cores is required to produce a core.

exception pysmt.exceptions.**SolverStatusError**
Exception raised if a method requiring a specific solver status is incorrectly called in the wrong status.

exception pysmt.exceptions.**ConvertExpressionError**(*message=None*, *expression=None*)
Exception raised if the converter cannot convert an expression.

exception pysmt.exceptions.**UnsupportedOperatorError**(*message=None*, *node_type=None*, *expression=None*)
The expression contains an operator that is not supported.

The argument node_type contains the unsupported operator id.

exception pysmt.exceptions.**SolverAPINotFound**
The Python API of the selected solver cannot be found.

exception pysmt.exceptions.**UndefinedSymbolError**(*name*)
The given Symbol is not in the FormulaManager.

exception pysmt.exceptions.**PysmtModeError**
The current mode is not supported for this operation

2.5.5 Factory

Factories are used to build new Solvers or Quantifier Eliminators without the need of specifying them. For example, the user can simply require a Solver that is able to deal with quantified theories, and the factory will return one such

solver among the available ones. This makes it possible to write algorithms that do not depend on a particular solver.

class pysmt.factory.**Factory**(*environment*, *solver_preference_list=None*, *qelim_preference_list=None*, *interpolation_preference_list=None*)

Factory used to build Solver, QuantifierEliminators, Interpolators etc.

This class contains the logic to magically select the correct solver. Moreover, this is the class providing the shortcuts is_sat, is_unsat etc.

set_solver_preference_list(*preference_list*)

Defines the order in which to pick the solvers.

The list is not required to contain all the solvers. It is possible to define a subsets of the solvers, or even just one. The impact of this, is that the solver will never be selected automatically. Note, however, that the solver can still be selected by calling it by name.

set_qelim_preference_list(*preference_list*)

Defines the order in which to pick the solvers.

set_interpolation_preference_list(*preference_list*)

Defines the order in which to pick the solvers.

all_solvers(*logic=None*)

Returns a dict <solver_name, solver_class> including all and only the solvers directly or indirectly supporting the given logic. A solver supports a logic if either the given logic is declared in the LOGICS class field or if a logic subsuming the given logic is declared in the LOGICS class field.

If logic is None, the map will contain all the known solvers

has_solvers(*logic=None*)

Returns true if self.all_solvers(logic) is non-empty

all_quantifier_eliminators(*logic=None*)

Returns a dict <qelim_name, qelim_class> including all and only the quantifier eliminators directly or indirectly supporting the given logic. A qelim supports a logic if either the given logic is declared in the LOGICS class field or if a logic subsuming the given logic is declared in the LOGICS class field.

If logic is None, the map will contain all the known quantifier eliminators

all_unsat_core_solvers(*logic=None*)

Returns a dict <solver_name, solver_class> including all and only the solvers supporting unsat core extraction and directly or indirectly supporting the given logic. A solver supports a logic if either the given logic is declared in the LOGICS class field or if a logic subsuming the given logic is declared in the LOGICS class field.

If logic is None, the map will contain all the known solvers

all_interpolators(*logic=None*)

Returns a dict <solver_name, solver_class> including all and only the solvers supporting interpolation and directly or indirectly supporting the given logic. A solver supports a logic if either the given logic is declared in the LOGICS class field or if a logic subsuming the given logic is declared in the LOGICS class field.

If logic is None, the map will contain all the known solvers

2.5.6 FNode

FNode are the building blocks of formulae.

class pysmt.fnode.**FNodeContent**(*node_type*, *args*, *payload*)

args
Alias for field number 1

node_type
Alias for field number 0

payload
Alias for field number 2

class pysmt.fnode.**FNode**(*content*, *node_id*)
FNode represent the basic structure for representing a formula.

FNodes are built using the FormulaManager, and should not be explicitely instantiated, since the FormulaManager takes care of memoization, thus guaranteeing that equivalent are represented by the same object.

An FNode is a wrapper to the structure FNodeContent. FNodeContent defines the type of the node (see operators.py), its arguments (e.g., for the formula A /B, args=(A,B)) and its payload, content of the node that is not an FNode (e.g., for an integer constant, the payload might be the python value 1).

The node_id is an integer uniquely identifying the node within the FormulaManager it belongs.

args()
Returns the subformulae.

arg(*idx*)
Return the given subformula at the given position.

get_free_variables()
Return the set of Symbols that are free in the formula.

get_atoms()
Return the set of atoms appearing in the formula.

simplify()
Return a simplified version of the formula.

substitute(*subs*)
Return a formula in which subformula have been substituted.

subs is a dictionary mapping terms to be subtituted with their substitution.

size(*measure=None*)
Return the size of the formula according to the given metric.

See SizeOracle

get_type()
Return the type of the formula by calling the Type-Checker.

See SimpleTypeChecker

is_constant(*_type=None*, *value=None*)
Test whether the formula is a constant.

Optionally, check that the constant is of the given type and value.

is_bool_constant(*value=None*)
Test whether the formula is a Boolean constant.

Optionally, check that the constant has the given value.

is_real_constant(*value=None*)
Test whether the formula is a Real constant.

Optionally, check that the constant has the given value.

is_int_constant(*value=None*)
Test whether the formula is an Integer constant.

Optionally, check that the constant has the given value.

is_bv_constant(*value=None*, *width=None*)
Test whether the formula is a BitVector constant.

Optionally, check that the constant has the given value.

is_algebraic_constant()
Test whether the formula is an Algebraic Constant

is_symbol(*type_=None*)
Test whether the formula is a Symbol.

Optionally, check that the symbol has the given type.

is_literal()
Test whether the formula is a literal.

A literal is a positive or negative Boolean symbol.

is_true()
Test whether the formula is the True Boolean constant.

is_false()
Test whether the formula is the False Boolean constant.

is_toreal()
Test whether the node is the ToReal operator.

is_forall()
Test whether the node is the ForAll operator.

is_exists()
Test whether the node is the Exists operator.

is_quantifier()
Test whether the node is a Quantifier.

is_and()
Test whether the node is the And operator.

is_or()
Test whether the node is the Or operator.

is_not()
Test whether the node is the Not operator.

is_plus()
Test whether the node is the Plus operator.

is_minus()
Test whether the node is the Minus operator.

is_times()
Test whether the node is the Times operator.

is_implies()
Test whether the node is the Implies operator.

is_iff()
Test whether the node is the Iff operator.

is_ite()
Test whether the node is the Ite operator.

is_equals()
Test whether the node is the Equals operator.

is_le()
Test whether the node is the LE (less than equal) relation.

is_lt()
Test whether the node is the LT (less than) relation.

is_bool_op()
Test whether the node is a Boolean operator.

is_theory_relation()
Test whether the node is a theory relation.

is_theory_op()
Test whether the node is a theory operator.

is_ira_op()
Test whether the node is an Int or Real Arithmetic operator.

is_lira_op(**args*, ***kwargs*)
Test whether the node is a IRA operator.

is_bv_op()
Test whether the node is a BitVector operator.

is_array_op()
Test whether the node is an array operator.

is_bv_not()
Test whether the node is the BVNot operator.

is_bv_and()
Test whether the node is the BVAnd operator.

is_bv_or()
Test whether the node is the BVOr operator.

is_bv_xor()
Test whether the node is the BVXor operator.

is_bv_concat()
Test whether the node is the BVConcat operator.

is_bv_extract()
Test whether the node is the BVConcat operator.

is_bv_ult()
Test whether the node is the BVULT (unsigned less than) relation.

is_bv_ule()
Test whether the node is the BVULE (unsigned less than) relation.

is_bv_neg()
Test whether the node is the BVNeg operator.

is_bv_add()
Test whether the node is the BVAdd operator.

is_bv_mul()
Test whether the node is the BVMul operator.

is_bv_udiv()
Test whether the node is the BVUDiv operator.

is_bv_urem()
Test whether the node is the BVURem operator.

is_bv_lshl()
Test whether the node is the BVLShl (logical shift left) operator.

is_bv_lshr()
Test whether the node is the BVLShr (logical shift right) operator.

is_bv_rol()
Test whether the node is the BVRol (rotate left) operator.

is_bv_ror()
Test whether the node is the BVRor (rotate right) operator.

is_bv_zext()
Test whether the node is the BVZext (zero extension) operator.

is_bv_sext()
Test whether the node is the BVSext (signed extension) operator.

is_bv_sub()
Test whether the node is the BVSub (subtraction) operator.

is_bv_slt()
Test whether the node is the BVSLT (signed less-than) operator.

is_bv_sle()
Test whether the node is the BVSLE (signed less-than-or-equal-to) operator.

is_bv_comp()
Test whether the node is the BVComp (comparison) operator.

is_bv_sdiv()
Test whether the node is the BVSDiv (signed division) operator.

is_bv_srem()
Test whether the node is the BVSRem (signed reminder) operator.

is_bv_ashr()
Test whether the node is the BVAshr (arithmetic shift right) operator.

is_select()
Test whether the node is the SELECT (array select) operator.

is_store()
Test whether the node is the STORE (array store) operator.

is_array_value()
Test whether the node is an array value operator.

bv_width()
Return the BV width of the formula.

bv_extract_start()
Return the starting index for BVExtract.

bv_extract_end()
Return the ending index for BVExtract.

bv_rotation_step()
Return the rotation step for BVRor and BVRol.

bv_extend_step()
Return the extension step for BVZext and BVSext.

serialize(*threshold=None*)
Returns a human readable representation of the formula.

The threshold parameter can be used to limit the amount of the formula that will be printed. See `HRSerializer`

is_function_application()
Test whether the node is a Function application.

is_term()
Test whether the node is a term.

All nodes are terms, except for function definitions.

symbol_type()
Return the type of the Symbol.

symbol_name()
Return the name of the Symbol.

constant_value()
Return the value of the Constant.

constant_type()
Return the type of the Constant.

bv2nat()
Return the unsigned value encoded by the BitVector.

bv_unsigned_value()
Return the unsigned value encoded by the BitVector.

bv_signed_value()
Return the signed value encoded by the BitVector.

bv_bin_str(*reverse=False*)
Return the binary representation of the BitVector as string.

The reverse option is provided to deal with MSB/LSB.

array_value_get(*index*)
Returns the value of this Array Value at the given index. The index must be a constant of the correct type.

This function is equivalent (but possibly faster) than the following code:

```
m = self.array_value_assigned_values_map()
try:
    return m[index]
except KeyError:
    return self.array_value_default()
```

function_name()
Return the Function name.

quantifier_vars()
Return the list of quantified variables.

2.5.7 Formula

The FormulaManager is used to create formulae.

All objects are memoized so that two syntactically equivalent formulae are represented by the same object.

The FormulaManager provides many more constructors than the operators defined (operators.py). This is because many operators are rewritten, and therefore are only virtual. Common examples are GE, GT that are rewritten as LE and LT. Similarly, the operator Xor is rewritten using its definition.

class pysmt.formula.**FormulaManager**(*env=None*)
FormulaManager is responsible for the creation of all formulae.

ForAll(*variables*, *formula*)

Creates an expression of the form: Forall variables. formula(variables)

Restrictions:

- Formula must be of boolean type
- Variables must be BOOL, REAL or INT

Exists(*variables*, *formula*)

Creates an expression of the form: Exists variables. formula(variables)

Restrictions:

- Formula must be of boolean type
- Variables must be BOOL, REAL or INT

Function(*vname*, *params*)
Returns the function application of vname to params.

Note: Applying a 0-arity function returns the function itself.

Not(*formula*)

Creates an expression of the form: not formula

Restriction: Formula must be of boolean type

Implies(*left*, *right*)

Creates an expression of the form: left -> right

Restriction: Left and Right must be of boolean type

Iff(*left*, *right*)

Creates an expression of the form: left <-> right

Restriction: Left and Right must be of boolean type

Minus(*left*, *right*)

Creates an expression of the form: left - right

Restriction: Left and Right must be both INT or REAL type

Times(**args*)
Creates a multiplication of terms

This function has polimorphic n-arguments:

- Times(a,b,c)
- Times([a,b,c])

Restriction:

- Arguments must be all of the same type
- Arguments must be INT or REAL

Pow(*base*, *exponent*)
Creates the n-th power of the base.

The exponent must be a constant.

Div(*left*, *right*)
Creates an expression of the form: left / right

Equals(*left*, *right*)
Creates an expression of the form: left = right

Restriction: Left and Right must be both REAL or INT type

GE(*left*, *right*)

Creates an expression of the form: left >= right

Restriction: Left and Right must be both REAL or INT type

GT(*left*, *right*)

Creates an expression of the form: left > right

Restriction: Left and Right must be both REAL or INT type

LE(*left*, *right*)

Creates an expression of the form: left <= right

Restriction: Left and Right must be both REAL or INT type

LT(*left*, *right*)

Creates an expression of the form: left < right

Restriction: Left and Right must be both REAL or INT type

Ite(*iff*, *left*, *right*)

Creates an expression of the form: if(iff) then left else right

Restriction:

- Iff must be BOOL
- Left and Right must be both of the same type

Real(*value*)
Returns a Real-type constant of the given value.

value can be:

- A Fraction(n,d)
- A tuple (n,d)
- A long or int n

- A float
- (Optionally) a mpq or mpz object

Int(*value*)
Return a constant of type INT.

TRUE()
Return the boolean constant True.

FALSE()
Return the boolean constant False.

And(**args*)
Returns a conjunction of terms.

This function has polimorphic arguments:

- And(a,b,c)
- And([a,b,c])

Restriction: Arguments must be boolean

Or(**args*)
Returns an disjunction of terms.

This function has polimorphic n-arguments:

- Or(a,b,c)
- Or([a,b,c])

Restriction: Arguments must be boolean

Plus(**args*)
Returns an sum of terms.

This function has polimorphic n-arguments:

- Plus(a,b,c)
- Plus([a,b,c])

Restriction:

- Arguments must be all of the same type
- Arguments must be INT or REAL

ToReal(*formula*)
Cast a formula to real type.

AtMostOne(**args*)
At most one of the bool expressions can be true at anytime.

This using a quadratic encoding: A -> !(B / C) B -> !(C)

ExactlyOne(**args*)
Encodes an exactly-one constraint on the boolean symbols.

This using a quadratic encoding: A / B / C A -> !(B / C) B -> !(C)

AllDifferent(**args*)
Encodes the 'all-different' constraint using two possible encodings.

AllDifferent(x, y, z) := (x != y) & (x != z) & (y != z)

Xor(*left*, *right*)
Returns the xor of left and right: left XOR right

Min(**args*)
Returns the encoding of the minimum expression within args

Max(**args*)
Returns the encoding of the maximum expression within args

EqualsOrIff(*left*, *right*)
Returns Equals() or Iff() depending on the type of the arguments.

This can be used to deal with ambiguous cases where we might be dealing with both Theory and Boolean atoms.

BV(*value*, *width=None*)
Return a constant of type BitVector.

value can be either: - a string of 0s and 1s - a string starting with "#b" followed by a sequence of 0s and 1s - an integer number s.t. 0 <= value < 2**width

In order to create the BV representation of a signed integer, the SBV() method shall be used.

SBV(*value*, *width=None*)
Returns a constant of type BitVector interpreting the sign.

If the specified value is an integer, it is converted in the 2-complement representation of the given number, otherwise the behavior is the same as BV().

BVOne(*width*)
Returns the bit-vector representing the unsigned one.

BVZero(*width*)
Returns the bit-vector with all bits set to zero.

BVNot(*formula*)
Returns the bitvector Not(bv)

BVAnd(*left*, *right*)
Returns the Bit-wise AND of two bitvectors of the same size.

BVOr(*left*, *right*)
Returns the Bit-wise OR of two bitvectors of the same size.

BVXor(*left*, *right*)
Returns the Bit-wise XOR of two bitvectors of the same size.

BVConcat(*left*, *right*)
Returns the Concatenation of the two BVs

BVExtract(*formula*, *start=0*, *end=None*)
Returns the slice of formula from start to end (inclusive).

BVULT(*left*, *right*)
Returns the formula left < right.

BVUGT(*left*, *right*)
Returns the formula left > right.

BVULE(*left*, *right*)
Returns the formula left <= right.

BVUGE(*left*, *right*)
Returns the formula left >= right.

BVNeg(*formula*)
Returns the arithmetic negation of the BV.

BVAdd(*left*, *right*)
Returns the sum of two BV.

BVSub(*left*, *right*)
Returns the difference of two BV.

BVMul(*left*, *right*)
Returns the product of two BV.

BVUDiv(*left*, *right*)
Returns the division of the two BV.

BVURem(*left*, *right*)
Returns the reminder of the two BV.

BVLShl(*left*, *right*)
Returns the logical left shift the BV.

BVLShr(*left*, *right*)
Returns the logical right shift the BV.

BVRol(*formula*, *steps*)
Returns the LEFT rotation of the BV by the number of steps.

BVRor(*formula*, *steps*)
Returns the RIGHT rotation of the BV by the number of steps.

BVZExt(*formula*, *increase*)
Returns the extension of the BV with 'increase' additional bits

New bits are set to zero.

BVSExt(*formula*, *increase*)
Returns the signed extension of the BV with 'increase' additional bits

New bits are set according to the most-significant-bit.

BVSLT(*left*, *right*)
Returns the SIGNED LOWER-THAN comparison for BV.

BVSLE(*left*, *right*)
Returns the SIGNED LOWER-THAN-OR-EQUAL-TO comparison for BV.

BVComp(*left*, *right*)
Returns a BV of size 1 equal to 0 if left is equal to right, otherwise 1 is returned.

BVSDiv(*left*, *right*)
Returns the SIGNED DIVISION of left by right

BVSRem(*left*, *right*)
Returns the SIGNED REMAINDER of left divided by right

BVAShr(*left*, *right*)
Returns the RIGHT arithmetic rotation of the left BV by the number of steps specified by the right BV.

BVNand(*left*, *right*)
Returns the NAND composition of left and right.

BVNor(*left*, *right*)
Returns the NOR composition of left and right.

BVXnor(*left*, *right*)
Returns the XNOR composition of left and right.

BVSGT(*left*, *right*)
Returns the SIGNED GREATER-THAN comparison for BV.

BVSGE(*left*, *right*)
Returns the SIGNED GREATER-THAN-OR-EQUAL-TO comparison for BV.

BVSMod(*left*, *right*)
Returns the SIGNED MODULUS of left divided by right.

BVRepeat(*formula*, *count=1*)
Returns the concatenation of count copies of formula.

Select(*arr*, *idx*)
Creates a node representing an array selection.

Store(*arr*, *idx*, *val*)
Creates a node representing an array update.

Array(*idx_type*, *default*, *assigned_values=None*)
Creates a node representing an array having index type equal to idx_type, initialized with default values.

If assigned_values is specified, then it must be a map from constants of type idx_type to values of the same type as default and the array is initialized correspondingly.

normalize(*formula*)
Returns the formula normalized to the current Formula Manager.

This method is useful to contextualize a formula coming from another formula manager.

E.g., f_a is defined with the FormulaManager a, and we want to obtain f_b that is the formula f_a expressed on the FormulaManager b : f_b = b.normalize(f_a)

2.5.8 Logics

Describe all logics supported by pySMT and other logics defined in the SMTLIB and provides methods to compare and search for particular logics.

class pysmt.logics.**Theory**(*arrays=False*, *arrays_const=False*, *bit_vectors=False*, *floating_point=False*, *integer_arithmetic=False*, *real_arithmetic=False*, *integer_difference=False*, *real_difference=False*, *linear=True*, *uninterpreted=False*)
Describes a theory similarly to the SMTLIB 2.0.

class pysmt.logics.**Logic**(*name*, *description*, *quantifier_free=False*, *theory=None*, *arrays=False*, *arrays_const=False*, *bit_vectors=False*, *floating_point=False*, *integer_arithmetic=False*, *real_arithmetic=False*, *integer_difference=False*, *real_difference=False*, *linear=True*, *uninterpreted=False*)
Describes a Logic similarly to the way they are defined in the SMTLIB 2.0

Note: We define more Logics than the ones defined in the SMTLib 2.0. See LOGICS for a list of all the logics and SMTLIB2_LOGICS for the restriction to the ones defined in SMTLIB2.0

get_quantified_version()
Returns the quantified version of logic.

is_quantified()
Return whether the logic supports quantifiers.

pysmt.logics.**get_logic_by_name**(*name*)
Returns the Logic that matches the provided name.

pysmt.logics.**convert_logic_from_string**(*name*)
Helper function to parse function arguments.

This takes a logic or a string or None, and returns a logic or None.

pysmt.logics.**get_logic_name**(*quantifier_free=False*, *arrays=False*, *arrays_const=False*, *bit_vectors=False*, *floating_point=False*, *integer_arithmetic=False*, *real_arithmetic=False*, *integer_difference=False*, *real_difference=False*, *linear=True*, *uninterpreted=False*)
Returns the name of the Logic that matches the given properties.

pysmt.logics.**get_logic**(*quantifier_free=False*, *arrays=False*, *arrays_const=False*, *bit_vectors=False*, *floating_point=False*, *integer_arithmetic=False*, *real_arithmetic=False*, *integer_difference=False*, *real_difference=False*, *linear=True*, *uninterpreted=False*)
Returns the Logic that matches the given properties.

Equivalent (but better) to executing get_logic_by_name(get_logic_name(...))

pysmt.logics.**most_generic_logic**(*logics*)
Given a set of logics, return the most generic one.

If a unique most generic logic does not exists, throw an error.

pysmt.logics.**get_closer_logic**(*supported_logics*, *logic*)
Returns the smaller supported logic that is greater or equal to the given logic. Raises NoLogicAvailableError if the solver does not support the given logic.

pysmt.logics.**get_closer_pysmt_logic**(*target_logic*)
Returns the closer logic supported by PYSMT.

pysmt.logics.**get_closer_smtlib_logic**(*target_logic*)
Returns the closer logic supported by SMT-LIB 2.0.

2.5.9 Operators

This module defines all the operators used internally by pySMT.

Note that other expressions can be built in the FormulaManager, but they will be rewritten (during construction) in order to only use these operators.

pysmt.operators.**new_node_type**(*new_node_id=None*)
Adds a new node type to the list of custom node types and returns the ID.

pysmt.operators.**op_to_str**(*node_id*)
Returns a string representation of the given node.

pysmt.operators.**all_types**()
Returns an iterator over all base and custom types.

2.5.10 Oracles

This module provides classes used to analyze and determine properties of formulae.

- QuantifierOracle says whether a formula is quantifier free
- TheoryOracle says which logic is used in the formula.

- FreeVarsOracle says which variables are free in the formula

class pysmt.oracles.**SizeOracle**(*env=None*)
Evaluates the size of a formula

get_size(*formula*, *measure=None*)
Return the size of the formula according to the specified measure.

The default measure is MEASURE_TREE_NODES.

class pysmt.oracles.**AtomsOracle**(*env=None*)
This class returns the set of Boolean atoms involved in a formula A boolean atom is either a boolean variable or a theory atom

get_atoms(*formula*)
Returns the set of atoms appearing in the formula.

2.5.11 Parsing

pysmt.parsing.**parse**(*string*)
Parse an hr-string.

pysmt.parsing.**HRParser**(*env=None*)
Parser for HR format of pySMT.

2.5.12 Printers

class pysmt.printers.**HRPrinter**(*stream*, *env=None*)
Performs serialization of a formula in a human-readable way.

E.g., Implies(And(Symbol(x), Symbol(y)), Symbol(z)) ~> '(x * y) -> z'

printer(*f*, *threshold=None*)
Performs the serialization of 'f'.

Thresholding can be used to define how deep in the formula to go. After reaching the thresholded value, "..." will be printed instead. This is mainly used for debugging.

class pysmt.printers.**HRSerializer**(*environment=None*)
Return the serialized version of the formula as a string.

serialize(*formula*, *printer=None*, *threshold=None*)
Returns a string with the human-readable version of the formula.

'printer' is the printer to call to perform the serialization. 'threshold' is the thresholding value for the printing function.

class pysmt.printers.**SmartPrinter**(*stream*, *subs=None*)
Better serialization allowing special printing of subformula.

The formula is serialized according to the format defined in the HRPrinter. However, everytime a formula that is present in 'subs' is found, this is replaced.

E.g., subs = {And(a,b): "ab"}

Everytime that the subformula And(a,b) is found, "ab" will be printed instead of "a & b". This makes it possible to rename big subformulae, and provide better human-readable representation.

pysmt.printers.**smart_serialize**(*formula*, *subs=None*, *threshold=None*)
Creates and calls a SmartPrinter to perform smart serialization.

2.5.13 Simplifier

class pysmt.simplifier.**Simplifier**(*env=None*)
Perform basic simplifications of the input formula.

simplify(*formula*)
Performs simplification of the given formula.

class pysmt.simplifier.**BddSimplifier**(*env=None*, *static_ordering=None*, *bool_abstraction=False*)
A simplifier relying on BDDs.

The formula is translated into a BDD and then translated back into a pySMT formula. This is a much more expensive simplification process, and might not work with formulas with thousands of boolean variables.

The option `static_ordering` can be used to provide a variable ordering for the underlying bdd.

The option `bool_abstraction` controls how to behave if the input formula contains Theory terms (i.e., is not purely boolean). If this option is False (default) an exception will be thrown when a Theory atom is found. If it is set to True, the Theory part is abstracted, and the simplification is performed only on the boolean structure of the formula.

2.5.14 SMT-LIB

pysmt.smtlib.parser.**open_**(*fname*)
Transparently handle .bz2 files.

pysmt.smtlib.parser.**get_formula**(*script_stream*, *environment=None*)
Returns the formula asserted at the end of the given script

script_stream is a file descriptor.

pysmt.smtlib.parser.**get_formula_strict**(*script_stream*, *environment=None*)
Returns the formula defined in the SMTScript.

This function assumes that only one formula is defined in the SMTScript. It will raise an exception if commands such as pop and push are present in the script, or if check-sat is called more than once.

pysmt.smtlib.parser.**get_formula_fname**(*script_fname*, *environment=None*, *strict=True*)
Returns the formula asserted at the end of the given script.

class pysmt.smtlib.parser.**SmtLibExecutionCache**
Execution environment for SMT2 script execution

bind(*name*, *value*)
Binds a symbol in this environment

unbind(*name*)
Unbinds the last binding of this symbol

get(*name*)
Returns the last binding for 'name'

update(*value_map*)
Binds all the symbols in 'value_map'

unbind_all(*values*)
UnBinds all the symbols in 'values'

class pysmt.smtlib.parser.**Tokenizer**(*handle*, *interactive=False*)
Takes a file-like object and produces a stream of tokens following the LISP rules.

If interative is True, the file reading proceeds char-by-char with no buffering. This is useful for interactive use for example with a SMT-Lib2-compliant solver

The method add_extra_token allows to "push-back" a token, so that it will be returned by the next call to consume_token, instead of reading from the actual generator.

static create_generator(*reader*)
Takes a file-like object and produces a stream of tokens following the LISP rules.

This is the method doing the heavy-lifting of tokenization.

class pysmt.smtlib.parser.**SmtLibParser**(*environment=None*, *interactive=False*)
Parse an SmtLib file and builds an SmtLibScript object.

The main function is get_script (and its wrapper get_script_fname). This function relies on the tokenizer function (to split the inputs in token) that is consumed by the get_command function that returns a SmtLibCommand for each command in the original file.

If the interactive flag is True, the file reading proceeds char-by-char with no buffering. This is useful for interactive use for example with a SMT-Lib2-compliant solver

atom(*token*, *mgr*)
Given a token and a FormulaManager, returns the pysmt representation of the token

get_expression(*tokens*)
Returns the pysmt representation of the given parsed expression

get_script(*script*)
Takes a file object and returns a SmtLibScript object representing the file

get_command_generator(*script*)
Returns a python generator of SmtLibCommand's given a file object to read from

This function can be used interactively, and blocks until a whole command is read from the script.

get_script_fname(*script_fname*)
Given a filename and a Solver, executes the solver on the file.

parse_atoms(*tokens*, *command*, *min_size*, *max_size=None*)
Parses a sequence of N atoms (min_size <= N <= max_size) consuming the tokens

parse_type(*tokens*, *command*, *additional_token=None*)
Parses a single type name from the tokens

parse_atom(*tokens*, *command*)
Parses a single name from the tokens

parse_params(*tokens*, *command*)
Parses a list of types from the tokens

parse_named_params(*tokens*, *command*)
Parses a list of names and type from the tokens

parse_expr_list(*tokens*, *command*)
Parses a list of expressions form the tokens

consume_opening(*tokens*, *command*)
Consumes a single '('

consume_closing(*tokens*, *command*)
Consumes a single ')'

get_assignment_list(*script*)
Parse an assignment list produced by get-model and get-value commands in SmtLib

get_command(*tokens*)
Builds an SmtLibCommand instance out of a parsed term.

class pysmt.smtlib.parser.**SmtLib20Parser**(*environment=None*, *interactive=False*)
Parser for SMT-LIB 2.0.

class pysmt.smtlib.parser.**SmtLibZ3Parser**(*environment=None*, *interactive=False*)
Parses extended Z3 SmtLib Syntax

pysmt.smtlib.script.**check_sat_filter**(*log*)
Returns the result of the check-sat command from a log.

Raises errors in case a unique check-sat command cannot be located.

class pysmt.smtlib.solver.**SmtLibOptions**(***base_options*)
Options for the SmtLib Solver.

•debug_interaction: True, False Print the communication between pySMT and the wrapped executable

class pysmt.smtlib.solver.**SmtLibSolver**(*args*, *environment*, *logic*, *LOGICS=None*, ***options*)
Wrapper for using a solver via textual SMT-LIB interface.

The solver is launched in a subprocess using args as arguments of the executable. Interaction with the solver occurs via pipe.

OptionsClass
alias of *SmtLibOptions*

Defines constants for the commands of the SMT-LIB

class pysmt.smtlib.annotations.**Annotations**(*initial_annotations=None*)
Handles and stores (key,value) annotations for formulae

add(*formula*, *annotation*, *value=None*)
Adds an annotation for the given formula, possibly with the specified value

remove(*formula*)
Removes all the annotations for the given formula

remove_annotation(*formula*, *annotation*)
Removes the given annotation for the given formula

remove_value(*formula*, *annotation*, *value*)
Removes the given annotation for the given formula

has_annotation(*formula*, *annotation*, *value=None*)
Returns True iff the given formula has the given annotation. If Value is specified, True is returned only if the value is matching.

annotations(*formula*)
Returns a dictionary containing all the annotations for the given formula as keys and the respective values. None is returned if formula has no annotations.

all_annotated_formulae(*annotation*, *value=None*)
Returns the set of all the formulae having the given annotation key. If Value is specified, only the formula having the specified value are returned.

2.5.15 Substituter

class pysmt.substituter.**Substituter**(*env*)
Performs substitution of a set of terms within a formula.

Let f be a formula ans subs be a map from formula to formula. Substitution returns a formula f' in which all occurrences of the keys of the map have been replaced by their value.

There are a few considerations to take into account:

- In which order to apply the substitution
- How to deal with quantified subformulas

The order in which we apply the substitutions gives raise to two different approaches: Most General Substitution and Most Specific Substitution. Lets consider the example:

f = (a & b) subs = {a -> c, (c & b) -> d, (a & b) -> c}

With the Most General Substitution (MGS) we obtain: f' = c

with the Most Specific Substitution (MSS) we obtain: f' = d

The default behavior before version 0.5 was MSS. However, this leads to unexpected results when dealing with literals, i.e., substitutions in which both x and Not(x) appear, do not work as expected. In case of doubt, it is recommended to issue two separate calls to the substitution procedure.

substitute(*formula*, *subs*)
Replaces any subformula in formula with the definition in subs.

class pysmt.substituter.**MGSubstituter**(*env*)
Performs Most Specific Substitution.

This is the default behavior since version 0.5

walk_identity_or_replace(*formula*, *args*, ***kwargs*)
If the formula appears in the substitution, return the substitution. Otherwise, rebuild the formula by calling the IdentityWalker.

class pysmt.substituter.**MSSubstituter**(*env*)
Performs Most Specific Substitution.

This was the default beahvior before version 0.5

2.5.16 Type-Checker

This module provides basic services to perform type checking and reasoning about the type of formulae.

- SimpleTypeChecker provides the pysmt.typing type of a formula
- The functions assert_*_args are useful for testing the type of arguments of a given function.

pysmt.type_checker.**assert_no_boolean_in_args**(*args*)
Enforces that the elements in args are not of BOOL type.

pysmt.type_checker.**assert_boolean_args**(*args*)
Enforces that the elements in args are of BOOL type.

pysmt.type_checker.**assert_same_type_args**(*args*)
Enforces that all elements in args have the same type.

pysmt.type_checker.**assert_args_type_in**(*args*, *allowed_types*)
Enforces that the type of the arguments is an allowed type

2.5.17 Typing

This module defines the types of the formulae handled by pySMT.

In the current version these are:

- Bool
- Int
- Real
- BVType
- FunctionType
- ArrayType

Types are represented by singletons. Basic types (Bool, Int and Real) are constructed here by default, while BVType and FunctionType relies on a factory service. Each BitVector width is represented by a different instance of BVType.

class pysmt.typing.**PySMTType**(*type_id=-1*)
Abstract class for representing a type within pySMT.

pysmt.typing.**BVType**(*width=32*)
Returns the singleton associated to the BV type for the given width.

This function takes care of building and registering the type whenever needed. To see the functions provided by the type look at _BVType.

pysmt.typing.**FunctionType**(*return_type*, *param_types*)
Returns the singleton of the Function type with the given arguments.

This function takes care of building and registering the type whenever needed. To see the functions provided by the type look at _FunctionType

Note: If the list of parameters is empty, the function is equivalent to the return type.

pysmt.typing.**ArrayType**(*index_type*, *elem_type*)
Returns the singleton of the Array type with the given arguments.

This function takes care of building and registering the type whenever needed. To see the functions provided by the type look at _ArrayType

2.5.18 Walkers

Provides walkers to navigate formulas.

Two types of walkers are provided: DagWalker and TreeWalker.

Internally, the Walkers have a dictionary that maps each FNode type to the appropriate function to be called. When subclassing a Walker remember to specify an action for the nodes of interest. Nodes for which a behavior has not been specified will raise a NotImplementedError exception.

Finally, an *experimental* meta class is provided called CombinerWalker. This class takes a list of walkers and returns a new walker that applies all the walkers to the formula. The idea is that multiple information can be extracted from the formula by navigating it only once.

class pysmt.walkers.**DagWalker**(*env=None*, *invalidate_memoization=False*)
DagWalker treats the formula as a DAG and performs memoization of the intermediate results.

This should be used when the result of applying the function to a formula is always the same, independently of where the formula has been found; examples include substitution and solving.

Due to memoization, a few more things need to be taken into account when using the DagWalker.

:func _get_key needs to be defined if additional arguments via keywords need to be shared. This function should return the key to be used in memoization. See substituter for an example.

iter_walk (*formula*, ***kwargs*)
Performs an iterative walk of the DAG

walk_all (*formula*, *args*, ***kwargs*)
Returns True if all the children returned True.

walk_any (*formula*, *args*, ***kwargs*)
Returns True if any of the children returned True.

walk_false (*formula*, *args*, ***kwargs*)
Returns False, independently from the children's value.

walk_identity (*formula*, ***kwargs*)
Returns formula, independently from the childrens's value.

walk_none (*formula*, *args*, ***kwargs*)
Returns None, independently from the children's value.

walk_true (*formula*, *args*, ***kwargs*)
Returns True, independently from the children's value.

class pysmt.walkers.**TreeWalker** (*env=None*)
TreeWalker treats the formula as a Tree and does not perform memoization.

This should be used when applying a the function to the same formula is expected to yield different results, for example, serialization. If the operations are functions, consider using the DagWalker instead.

The recursion within **walk_** methods is obtained by using the 'yield' keyword. In practice, each **walk_** method is a generator that yields its arguments. If the generator returns None, no recursion will be performed.

walk (*formula*, *threshold=None*)
Generic walk method, will apply the function defined by the map self.functions.

If threshold parameter is specified, the walk_threshold function will be called for all nodes with depth >= threshold.

walk_skip (*formula*)
Default function to skip a node and process the children

class pysmt.walkers.**IdentityDagWalker** (*env=None*, *invalidate_memoization=None*)
This class traverses a formula and rebuilds it recursively identically.

This could be useful when only some nodes needs to be rewritten but the structure of the formula has to be kept.

class pysmt.walkers.**DagWalker** (*env=None*, *invalidate_memoization=False*)
DagWalker treats the formula as a DAG and performs memoization of the intermediate results.

This should be used when the result of applying the function to a formula is always the same, independently of where the formula has been found; examples include substitution and solving.

Due to memoization, a few more things need to be taken into account when using the DagWalker.

:func _get_key needs to be defined if additional arguments via keywords need to be shared. This function should return the key to be used in memoization. See substituter for an example.

iter_walk (*formula*, ***kwargs*)
Performs an iterative walk of the DAG

walk_true (*formula*, *args*, ***kwargs*)
Returns True, independently from the children's value.

walk_false (*formula*, *args*, ***kwargs*)
Returns False, independently from the children's value.

walk_none (*formula*, *args*, ***kwargs*)
Returns None, independently from the children's value.

walk_identity (*formula*, ***kwargs*)
Returns formula, independently from the childrens's value.

walk_any (*formula*, *args*, ***kwargs*)
Returns True if any of the children returned True.

walk_all (*formula*, *args*, ***kwargs*)
Returns True if all the children returned True.

class pysmt.walkers.**TreeWalker** (*env=None*)
TreeWalker treats the formula as a Tree and does not perform memoization.

This should be used when applying a the function to the same formula is expected to yield different results, for example, serialization. If the operations are functions, consider using the DagWalker instead.

The recursion within **walk_** methods is obtained by using the 'yield' keyword. In practice, each **walk_** method is a generator that yields its arguments. If the generator returns None, no recursion will be performed.

walk (*formula*, *threshold=None*)
Generic walk method, will apply the function defined by the map self.functions.

If threshold parameter is specified, the walk_threshold function will be called for all nodes with depth >= threshold.

walk_skip (*formula*)
Default function to skip a node and process the children

class pysmt.walkers.**IdentityDagWalker** (*env=None*, *invalidate_memoization=None*)
This class traverses a formula and rebuilds it recursively identically.

This could be useful when only some nodes needs to be rewritten but the structure of the formula has to be kept.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

X