pySym Documentation

Release 1

Michael Bann

Apr 30, 2018

Contents

`pySym` is a Symbolic Execution Engine for Python Scripts. It is written in Python and designed for easy adoption for a subset of symbolic execution problems.

CHAPTER 1

Getting Started

1.1 About pySym

1.1.1 Introduction to `pySym`

`pySym` is a python application that utilizes the Microsoft Z3 Theorem Prover to solve it's constraints. The goal for this application is to faithfully execute *basic* python scripts and provide a means to analyze them symbolically. Note that this application will not allow you to test your python apps for python interpreter version dependent bugs. It will attempt to follow what Python *should* be doing and thus would be more appropriate for the following two cases:

1. Finding logic bugs in your application.
2. Discovering possibly unused code segments
3. Generating test cases
4. Rapidly prototyping concepts for symbolic analysis

1.1.2 Python Versions

pySym is written in Python 3 and will parse Python 3 code. It will likely not parse Python 2 code, however small code can simple be auto-upgraded if need be by the 2to3 script.

1.1.3 pySym Weaknesses

`pySym` is not, nor will it ever be, a complete Python Symbolic Execution Engine. Scripted languages are rapidly evolving, and Python is no exception. Rapid evolution aside, even between minor versions of Python there are many small changes that will cause unintended code paths. Likely it is not feasible for any script based Symbolic Execution of Python to be that thorough.

If your goal is faithful Python symbolic exeuction to any given Python major/minor versions, I'd recommend looking at a project called CHEF. This project is a novel approach whereby the authors instrument the source interpreter with the S2E framework thereby causing very thorough code path generation.

1.1.4 pySym Strengths

pySym is an attempt at generalizing Python Symbolic Execution utilizing Python itself. One major downfall that approaches such as CHEF have is that it is unable to make symbolic input aside from int and string types. With pySym, already it has the ability to produce fully symbolic ints, floats, lists, and strings. When more datatypes are added, they will have the ability to be fully symbolic as well. This is important when you want to stress test symbolic Dictionaries or other more complex objects.

It is also easy to use. Simply run the installer, load up the script you want to execute (with or without changes), and tell pySym to explore. It's not dependent on compiling special versions of applications or using strange commands that you're not quite sure what you're doing. You can be up and running in as little as 10 minutes.

As a follow-on, because everything can be symbolic and you can prototype Python code in general quickly, pySym gives you a way to explore concepts without needing to be a symbolic exeuction expert. Simply write your code as you would normally for Python, then run it through pySym.

1.2 Installing pySym

1.2.1 pip install

pySym has been refactored to be a proper python package. This is now the easiest means of installing this library. To do so, simply clone this repo and run pip install:

```
$ git clone https://github.com/bannsec/pySym.git
$ cd pySym
$ pip install .
```

Note that it's recommended to install into a python virtual environment instead of your system environment.

1.2.2 Docker install

You can also install pySym using a pre-build Docker container. This is setup as an auto-build image, so you will always be up-to-date:

```
$ sudo docker pull bannsec/pysym
$ sudo docker run -it --rm bannsec/pysym
```

1.2.3 setup.sh (depreciated)

If you run on Ubuntu (or derivative) this will probably work for you. It completely automates the setup process that is outlined below. Usage is as follows:

```
bash$ ./setup.sh
```

That's it. You're all setup. Note that if you want to remove pySym that was installed this way, you can use the uninstall.sh script to do so.

1.2.4 Manual Install

If the auto install doesn't work for some reason, the following manual steps can be taken to install.

Install Dependencies

Dependencides for `pySym` include:

- `virtualenv` (Python's virtual environment)
- `gcc`
- `make`
- `python3`
- `git`

For Ubuntu, the following line works for me:

```
$ sudo apt-get update
$ sudo apt-get install -y make gcc g++ virtualenv virtualenvwrapper python3 git
```

`virtualenvwrapper` is recommended to make switching between environments easier, but it's not required.

Set up your Virtual Environment

It's not necessary to create a virtual environment, but it's good practice and keeps things clean. Perform the following steps to set up a virtual environment named pySym:

```
$ mkdir -p ${HOME}/.virtualenvs/pySym
$ virtualenv -p $(which python3) ${HOME}/.virtualenvs/pySym
```

Now you should activate it so that you can work inside it. If you have virtualenvwrappers installed, you can do the following:

```
$ workon pySym
```

If you don't have that installed, or for whatever reason it isn't working, you can source the virtualenv directly to activate your environment:

```
$ source "${HOME}/.virtualenvs/pySym/bin/activate"
```

You should see "(pySym)" next to your command prompt to indicate the environment is activated properly.

Install Python Packages

With your virtual environment activated, install the Python dependencies. From the root of the repository, do the following:

```
(pySym)$ pip install -r requirements.txt
```

The following are optional packages for running the py.test unit tests. If you're not planning on running the tests you can omit these:

```
(pySym)$ pip3 install pytest
(pySym)$ pip3 install python-coveralls
(pySym)$ pip3 install coverage
(pySym)$ pip3 install pytest-cov
(pySym)$ pip3 install pytest-xdist
```

Install Microsoft Z3 with Python Bindings

Time to install Microsoft's Z3. This step will take a while to compile everything, so be patient. First thing, create the working directory and git clone the z3 tool:

```
(pySym)$ mkdir -p ${HOME}/opt/pySymZ3
(pySym)$ cd ${HOME}/opt
(pySym)$ git clone https://github.com/Z3Prover/z3.git pySymZ3
(pySym)$ cd pySymZ3
```

Perform the build and install. As noted, this could take a while:

```
(pySym)$ python scripts/mk_make.py --python
(pySym)$ cd build
(pySym)$ make
(pySym)$ make install
```

Performing these steps while having your Python virtual environment activated will cause the install to be performed into that environment. This also means that you do not need root priviledges for any of this install.

You are now ready to use `pySym`.

1.3 pySym Quick-Start

1.3.1 Running Your First Program

Assuming that you have already installed pySym, actiate your virtual environment and load up a source:

```
$ workon pySym
(pySym)$ ipython
```

Automated Loading

Assuming you have a program you want to symbolically execute called *my_test_program.py*, you can do so with the following lines:

```
In [1]: import pySym

In [2]: proj = pySym.Project("my_test_program.py")

In [3]: pg = proj.factory.path_group()
```

You can now run it by simply executing:

```
In [4]: pg.explore()
```

Manually From Strings

You can also load your script directly via python string. The following example loads it from a file:

```
In [1]: from pySym.pyPath import Path

In [2]: import ast

In [3]: from pySym import Colorer

In [4]: from pySym.pyPathGroup import PathGroup

In [5]: with open("test","r") as f:
   ...:         b = ast.parse(source).body
   ...:         p = Path(b,source=source)
   ...:         pg = PathGroup(p)
```

You can now run it by simply executing:

```
In [6]: pg.explore()
```

See the examples page for example programs.

1.4 Examples

1.4.1 Tokyo Westerns CTF 2017: My Simple Cipher

External Writeup: BannSecurity.com

1.4.2 Prime Finder

Let's recreate a simple problem of finding all the primes under 50. Stealing and modifying some code from the internet, we can use the following python code to do just that.

```
# Enable logging if you want. It's faster without.
#import logging
#logging.basicConfig(level=logging.DEBUG,format='%(name)s - %(levelname)s -
↪%(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
from pySym.pyPath import Path
from pySym import ast_parse
from pySym import Colorer
from pySym.pyPathGroup import PathGroup

source = """
def isprime(n):

    # 0 and 1 are not primes
    if n < 2:
        return 0

    # 2 is the only even prime number
    if n == 2:
        return 1

    # all other even numbers are not primes
    if n % 2 == 0:
        return 0
```

```
    # range starts with 3 and only needs to go up
    # the square root of n for all odd numbers
    for x in range(3, int(n**0.5) + 1, 2):
        if n % x == 0:
            return 0

    return 1

x = [x for x in range(50) if isprime(x) == 1]
"""

b = ast_parse.parse(source).body
p = Path(b,source=source)
pg = PathGroup(p,discardFailures=True)
```

The first two lines are just to add more logging. If you're not interested in watching scrolling text, just leave those two out. Aside from that, the rest is self explanatory. As of writing, I have not implemented Bools yet, so this is why I'm returning integer 1 or 0. The end effect is the same, however.

As written, this code will symbolically execute without special changes. To do so, just execute:

```
pg.explore()
```

This will cause *pySym* to start exploring this state and finding valid paths. Since we're only dealing with concrete variables, there will be one valid path through. However, there will be many deadended paths since *pySym* will take every branch along the way.

Also note, I used "discardFailures=True" in PathGroup. This is because with this option enabled, you won't be wasting memory of your computer with deadended paths. When there are many such paths, or if the paths are complicated, the total memory used by paths you're not interested in can become high. This option allows you to immediately forget those paths and thus reduce your memory profile.

Once done, we can confirm that the expected results were attained:

```
In [7]: pg
Out[7]: <PathGroup with 1 completed>

In [8]: x = pg.completed[0].state.getVar('x')

In [9]: print(x)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

The first command simply shows us that our path group has 1 completed path. As mentioned above, there would be many more deadended paths if we hadn't set discardFailures to True. The second command reaches into our only completed path (index 0) and asks the state to give us the pyObjectManager object for variable named 'x'. We can do many things with this variable at this point, however for the purposes of this example, we simply print out the variable. It will pause for a few moments as z3 solves constraints for us and *pySym* puts it into the proper representation (a list).

1.5 What is Implemented

1.5.1 Overview

The idea here is to document the knowns and unknowns with *pySym*. Generally speaking, the basics of Python execution have mostly been implemented. By this I mean, while loops, for loops, integers, arithmetic, bit operations, etc. I'll focus mainly here on big picture structures.

1.5.2 Known Limitations

- importing isn't implemented
- function scoping isn't implemented (meaning, if you declare a function inside another function, scope will be messed up)
- In some cases, call nesting with chaining doesn't work. Not sure exactly when this is, but for instance (test(test2()).rstrip("x")) i believe would fail.
- Case mixing is always a bit of a gamble (real/int/bitvec)

1.5.3 pyState functions

- **pyState.BVV(i,size=ast.Num(Z3_DEFAULT_BITVEC_SIZE))**
  - Declares a BitVec Value of value "i" with optional BitVec size size
- **pyState.BVS(size=ast.Num(Z3_DEFAULT_BITVEC_SIZE))**
  - Declares a symbolic BitVec of max bits "size"
- **pyState.Int()**
  - Declares a symbolic Integer
- **pyState.Real()**
  - Declares a symbolic Real value
- **pyState.String(length=ast.Num(Z3_MAX_STRING_LENGTH))**
  - Declares a symbolic String value of length "length"

Example:

- **If we want to declare a variable to be a Symbolic String of length 10, this would go in the python source code script that w**
  - s = pyState.String(10)
  - Note that you assign it like you would if you were executing it. Also, you do not need to from pySym import pyState to call this.

1.5.4 Python Built-in

- abs (Fully Implemented)
- all (Not Implemented)
- any (Not Implemented)
- ascii (Not Implemented)
- bin (Partially Implemented)
- bool (Not Implemented)
- bytearray (Not implemented)
- bytes (Not Implemented)
- callable (Not Implemented)
- chr (Fully Implemented)

- classmethod (Not Implemented)
- compile (Not Implemented)
- complex (Not Implemented)
- delattr (Not Implemented)
- dict (Not Implemented)
- dir (Not Implemented)
- divmod (Not Implemented)
- enumerate (Not Implemented)
- eval (Not Implemented)
- exec (Not Implemented)
- filter (Not Implemented)
- float (Not Implemented)
- format (Not Implemented)
- frozenset (Not Implemented)
- getattr (Not Implemented)
- globals (Not Implemented)
- hasattr (Not Implemented)
- hash (Not Implemented)
- help (Not Implemented)
- hex (Partially Implemented)
- id (Not Implemented)
- input (Not Implemented)
- int (Partially Implemented)
- isinstance (Not Implemented)
- iter (Not Implemented)
- len (Fully Implemented)
- list (Not Implemented)
- locals (Not Implemented)
- map (Not Implemented)
- max (Not Implemented)
- memoryview (Not Implemented)
- min (Not Implemented)
- next (Not Implemented)
- object (Not Implemented)
- oct (Not Implemented)
- open (Not Implemented)

- ord (Fully Implemented)
- pow (Not Implemented)
- print (Partially Implemented)
- property (Not Implemented)
- range (Partially Implemented)
- repr (Not Implemented)
- reversed (Not Implemented)
- round (Not Implemented)
- set (Not Implemented)
- setattr (Not Implemented)
- slice (Not Implemented)
- sorted (Not Implemented)
- staticmethod (Not Implemented)
- str (Partially Implemented)
- sum (Not Implemented)
- super (Not Implemented)
- tuple (Not Implemented)
- type (Not Implemented)
- vars (Not Implemented)
- **zip (Partially Implemented)**
  - zip(list1,list2) works. 3 or more lists doesn't work at the moment
- __import__ (Not Implemented)

1.5.5 Numbers

- Real/Int and implicit BitVecs are implemented
- **Integer Methods**
  - bit_length (Not Implemented)
  - conjugate (Not Implemented)
  - denominator (Not Implemented)
  - from_bytes (Not Implemented)
  - imag (Not Implemented)
  - numerator (Not Implemented)
  - real (Not Implemented)
  - to_bytes (Not Implemented)
- **Float Methods**
  - as_integer_ratio (Not Implemented)

  - conjugate (Not Implemented)
  - fromhex (Not Implemented)
  - hex (Not Implemented)
  - imag (Not Implemented)
  - is_integer (Not Implemented)
  - real (Not Implemented)

1.5.6 Strings

- **methods**
  - capitalize (Not Implemented)
  - casefold (Not Implemented)
  - center (Not Implemented)
  - count (Not Implemented)
  - encode (Not Implemented)
  - endswith (Not Implemented)
  - epandtabs (Not Implemented)
  - find (Not Implemented)
  - format (Not Implemented)
  - format_map (Not Implemented)
  - index (Partially Implemented)
  - isalnum (Not Implemented)
  - isalpha (Not Implemented)
  - isdecimal (Not Implemented)
  - isdigit (Not Implemented)
  - isidentifier (Not Implemented)
  - islower (Not Implemented)
  - isnumeric (Not Implemented)
  - isprintable (Not Implemented)
  - isspace (Not Implemented)
  - istitle (Not Implemented)
  - isupper (Not Implemented)
  - join (Partially Implemented)
  - ljust (Not Implemented)
  - lower (Not Implemented)
  - lstrip (Not Implemented)
  - maketrans (Not Implemented)

  - partition (Not Implemented)
  - replace (Not Implemented)
  - rfind (Not Implemented)
  - rindex (Not Implemented)
  - rjust (Not Implemented)
  - rpartition (Not Implemented)
  - rsplit (Not Implemented)
  - rstrip (Fully Implemented)
  - split (Not Implemented)
  - splitlines (Not Implemented)
  - startswith (Not Implemented)
  - strip (Not Implemented)
  - swapcase (Not Implemented)
  - title (Not Implemented)
  - translate (Not Implemented)
  - upper (Not Implemented)
  - zfill (Partially Implemented)

1.5.7 Lists

- **methods**
  - append (Fully Implemented)
  - clear (Fully Implemented)
  - copy (Not Implemented)
  - count (Not Implemented)
  - extend (Not Implemented)
  - index (Not Implemented)
  - insert (Partially Implemented – No symbolic index)
  - pop (Not Implemented)
  - remove (Not Implemented)
  - reverse (Not Implemented)
  - sort (Not Implemented)

1.5.8 Python Common Libraries

- **random**
  - randint (Partially Implemented)

1.5.9 Dictionaries

Not implemented

1.5.10 Tuples

Not Implemented

1.5.11 Files

Not Implemented

1.5.12 Sets

Not Implemented

1.5.13 Booleans

Not Implemented

1.5.14 Bytes

Not Implemented

1.5.15 ByteArray

Not Implemented

1.5.16 Class

Not Implemented

1.5.17 Functions

Mostly implemented. Arbitrary function declaration. Keyword arguments, positional arguments, default arugments are implemented.

Some nested call limitations at the moment. If unsure if it'll work, just try it and let me know.

1.5.18 Symbolic Hooking

There often times is a need to hook symbolic execution. This is most often used to flatten a group of statements into a single expression or otherwise change the execution flow. See hooking for more information.

1.6 Symbolic Hooking

1.6.1 What is Hooking

Hooking is a means to interject your own commands into the symbolic execution of the application. For instance, a common reason to hook a function or part of a function is to provide your own symbolic summary for it. In this way, you can jump out of the symbolically executed script, back into the engine and tell *pySym* how to keep that section symbolic.

One quick example is, if you consider an if statement nested inside a while loop, as follows:

```
def my_function(my_list):
  output = []
  for element in my_list:
     if element == 0:
        output.append("zero")
     else:
        output.append("one")
  return output
```

In the above example, that `if` statement inside the `for` loop would actually cause pySym to state split for each element. Depending on the size of the input list and how symbolic the input actually is, this could cause a path explosion issue. One way around that is to hook `my_function` and create a summary for it.

1.6.2 How to Hook

At present, hooking in *pySym* is accomplished via the *pySym.Project.hook* method.

See the method documentation for more details:

pySym.Project.Project.hook()

1.7 API

What follows is `pySym` core API documentation.

Note: This documentation is a work in progress. Not everything is documented yet.

1.7.1 Project

class pySym.Project.**Project**(*file*, *debug=False*)

Bases: object

factory

file_name

str – Name of the file that's being symbolically executed.

hook(*address*, *callback*)

Registers pySym to hook address and call the callback when hit.

Callback function will be called with the current state as the first parameter.

Args: address (int): Line number of the source code to hook this callback to. callback (types.FunctionType): Function to call when line number is hit.

Example:

```
>>> def my_callback(state):
        print(state)
>>> project.hook(12, my_callback)
```

1.7.2 pyObjectManager

Submodules

pyObjectManager.BitVec module

class pySym.pyObjectManager.BitVec.**BitVec**(*varName*, *ctx*, *size*, *count=None*, *state=None*, *increment=False*, *value=None*, *uuid=None*, *clone=None*)

Bases: object

Define a BitVec

canBe(**args*, ***kwargs*)

copy()

count

ctx

getValue(**args*, ***kwargs*)

getZ3Object(**args*, ***kwargs*)

increment()
Increment the counter

isStatic(**args*, ***kwargs*)

is_constrained

is_unconstrained

mustBe(**args*, ***kwargs*)

parent

setTo(*var*, **args*, ***kwargs*)
Sets this BitVec object to be equal/copy of another. Type can be int, or BitVec

size

state
Returns the state assigned to this object.

uuid

value

varName

pyObjectManager.Char module

class pySym.pyObjectManager.Char.**Char**(*varName*, *ctx*, *count=None*, *variable=None*, *state=None*, *increment=False*, *uuid=None*, *clone=None*)

Bases: object

Define a Char (Character)

canBe(**args*, ***kwargs*)

copy()

count

ctx

getValue(**args*, ***kwargs*)

getZ3Object(**args*, ***kwargs*)

increment()

isStatic(**args*, ***kwargs*)

is_constrained

is_unconstrained

mustBe(**args*, ***kwargs*)

parent

setTo(*var*, **args*, ***kwargs*)
Sets this Char to the variable. Raises exception on failure.

state
Returns the state assigned to this object.

uuid

varName

variable

pyObjectManager.Ctx module

class pySym.pyObjectManager.Ctx.**Ctx**(*ctx*, *variables=None*)
Bases: object

Define a Ctx Object

__getitem__(*index*)
We want to be able to do "list[x]", so we define this.

__setitem__(*key*, *value*)
Sets value at index key. Checks for variable type, updates counter according, similar to getVar call

copy()

ctx

index(*elm*)
Returns "index" of the given element. Raises exception if it's not found For a pseudo dict class, this is just the key for the key,val pair

items()

state
Returns the state assigned to this object.

variables

variables_need_copy

pyObjectManager.Int module

class pySym.pyObjectManager.Int.**Int**(*varName*, *ctx*, *count=None*, *value=None*, *state=None*, *increment=False*, *uuid=None*, *clone=None*)

Bases: object

Define an Int

canBe(**args*, ***kwargs*)

copy()

count

ctx

getValue(**args*, ***kwargs*)

getZ3Object(**args*, ***kwargs*)

increment()

isStatic(**args*, ***kwargs*)

is_constrained

is_unconstrained

mustBe(**args*, ***kwargs*)

parent

setTo(*var*, **args*, ***kwargs*)
Sets this Int object to be equal/copy of another. Type can be int or Int

state
Returns the state assigned to this object.

uuid

value

varName

pyObjectManager.List module

class pySym.pyObjectManager.List.**List**(*varName*, *ctx*, *count=None*, *variables=None*, *state=None*, *increment=False*, *uuid=None*)

Bases: object

Define a List

__getitem__(*index*)
We want to be able to do "list[x]", so we define this.

__setitem__(*key*, *value*)
Sets value at index key. Checks for variable type, updates counter according, similar to getVar call

append(*var*, *kwargs=None*)

Input: var = pyObjectManager oject to append (i.e.: Int/Real/etc) (optional) kwargs = optional keyword args needed to instantiate type

Action: Resolves object, creates variable if needed

Returns: Nothing

canBe(*var*)
Test if this List can be equal to another List Returns True or False

copy()

count

ctx

getValue()
Return a possible value. You probably want to check isStatic before calling this.

increment()

index(*elm*)
Returns index of the given element. Raises exception if it's not found

insert(*index*, *object*, *kwargs=None*)
Emulate the list insert method, just on this object.

isStatic()
Checks if this list can only have one possible value overall (including all elements). Returns True/False

mustBe(*var*)
Test if this List must be equal to another

parent

pop(*i*)

setTo(*otherList*, *clear=False*)
Sets this list to another of type List (optional) clear = Should we clear the current variables and set, or set the current variables in place retaining their constraints?

state
Returns the state assigned to this object.

uuid

varName

variables

variables_need_copy

pyObjectManager.Real module

class pySym.pyObjectManager.Real.**Real**(*varName*, *ctx*, *count=None*, *value=None*, *state=None*, *increment=False*, *uuid=None*)

Bases: object

Define a Real

__str__()
str will change this object into a possible representation by calling state.any_real

canBe(*var*)
Test if this Real can be equal to the given variable Returns True or False

copy()

count

ctx

getValue()
Resolves the value of this real. Assumes that isStatic method is called before this is called to ensure the value is not symbolic

getZ3Object(*increment=False*)
Returns the z3 object for this variable

increment()

isStatic()
Returns True if this object is a static variety (i.e.: RealVal(12))

mustBe(*var*)
Test if this Real must be equal to another variable Returns True or False

parent

setTo(*var*, **args*, ***kwargs*)
Sets this Real object to be equal/copy of another. Type can be float, Real, Int, or int

state
Returns the state assigned to this object.

uuid

value

varName

pyObjectManager.String module

class pySym.pyObjectManager.String.**String**(*varName*, *ctx*, *count=None*, *string=None*, *variables=None*, *state=None*, *length=None*, *increment=False*, *uuid=None*)

Bases: object

Define a String

__getitem__(*index*)
We want to be able to do "string[x]", so we define this.

__setitem__(*key*, *value*)
String doesn't support setitem

__str__()
str will change this object into a possible representation by calling state.any_str

canBe(*var*)
Test if this string can be equal to the given variable Returns True or False

copy()

count

ctx

getValue()
Resolves the value of this String. Assumes that isStatic method is called before this is called to ensure the value is not symbolic

getZ3Object()
Convenience function. Will return z3 object for Chr if this is a string of length 1, else error.

increment()

index(*elm*)
Returns index of the given element. Raises exception if it's not found

isStatic()
Returns True if this object is a static variety (i.e.: "test"). Also returns True if object has only one possibility

mustBe(*var*)
Test if this string must be equal to the given variable. This means there's no other options and it's not symbolic

parent

pop(*index=None*)
Not exactly something you can do on a string, but helpful for our symbolic execution

setTo(*var*, *clear=None*)
Sets this String object to be equal/copy of another. Type can be str or String. clear = Boolean if this variable should be cleared before setting (default False)

state
Returns the state assigned to this object.

uuid

varName

variables

Module contents

class pySym.pyObjectManager.**ObjectManager**(*variables=None*, *returnObjects=None*, *state=None*)

Bases: object

Object Manager will keep track of objects. Generally, Objects will be variables such as ints, lists, strings, etc.

copy()
Return a copy of the Object Manager

getParent(*key*, *haystack=None*)
Returns the parent object for any given object by recursively searching.

getVar(*varName*, *ctx*, *varType=None*, *kwargs=None*, *softFail=None*)

Input: varName = name of variable to get ctx = Context for variable (optional) varType = Class type of variable (ex: pyObjectManager.Int) (optional) kwargs = args needed to instantiate variable (optional) softFail = True/False, should raise an exception if getVar fails. Default is False

Action: Find appropriate variable object, creating one if necessary

Returns: pyObjectManager object for given variable (i.e.: pyObjectManager.Int)

newCtx(*ctx*)
Sets up a new context (ctx)

returnObjects

setVar(*varName*, *ctx*, *var*)

Input: varName = variable name (i.e.: 'x') ctx = Context to set for var = variable object of type pyObjectManager.X

Action: Sets variable to the input (var) object

Returns: Nothing

state
Returns the state assigned to this object.

variables

1.7.3 pyPath

class pySym.pyPath.**Path**(*path=None*, *backtrace=None*, *state=None*, *source=None*, *project=None*)
Bases: object

Defines a path of execution.

backtrace

copy(*state=None*)

Input: (optional) state == pyState object to use instead of copying the current state.

Action: Create a copy of the current Path object

Returns: Copy of the path

error

printBacktrace()
Convinence function to print out what we've executed so far

source

state

step()
Move the current path forward by one step Note, this actually makes a copy/s and returns them. The initial path isn't modified. Returns: A list of paths or empty list if the path is done

pySym.pyPath.**random**() → x in the interval [0, 1).

1.7.4 pyPathGroup

class pySym.pyPathGroup.**PathGroup**(*path=None*, *ignore_groups=None*, *search_strategy=None*, *project=None*)
Bases: object

__str__()
Pretty print status

active

completed

deadended

errored

explore(*find=None*)

Input: (optional) find = input line number to explore to

Action: Step through script until line is found

Returns: True if found, False if not

found

ignore_groups

search_strategy
str – Strategy for searching the paths.

Valid options are:

- Breadth (default): Traditional searching. Step each path in order.
- Depth: Drill one path down as far as possible.
- Random: Randomize what paths get stepped and what order.

step()
Step all active paths one step.

unstash(*path=None*, *from_stash=None*, *to_stash=None*)
Simply moving around paths for book keeping.

1.7.5 pyState

Subpackages

pyState.functions package

Submodules

pyState.functions.abs module

pySym.pyState.functions.abs.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate abs funcion

pyState.functions.bin module

pySym.pyState.functions.bin.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate bin funcion

pyState.functions.hex module

pySym.pyState.functions.hex.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate hex funcion

pyState.functions.int module

pySym.pyState.functions.int.**handle**(*state*, *call*, *obj*, *base=10*, *ctx=None*)
Simulate int funcion

pyState.functions.len module

pySym.pyState.functions.len.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate len funcion

pyState.functions.ord module

pySym.pyState.functions.ord.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate ord funcion

pyState.functions.print module

pySym.pyState.functions.print.**handle**(*state*, *call*, *s*, *ctx=None*)
Pretend to print stuff

pyState.functions.range module

pySym.pyState.functions.range.**handle**(*state*, *call*, *a*, *b=None*, *c=None*, *ctx=None*)
Simulate range funcion

pyState.functions.str module

pySym.pyState.functions.str.**handle**(*state*, *call*, *obj*, *ctx=None*)
Simulate str funcion

pyState.functions.zip module

pySym.pyState.functions.zip.**handle**(*state*, *call*, *left*, *right*, *ctx=None*)
Simulate zip funcion

Module contents

pyState.Assign

pySym.pyState.Assign.**handle**(*state*, *element*)
Attempt to handle the Python Assign element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Assign*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to Assign. It is not meant to be called manually via a user.

Example

Example of ast.Assign is: x = 1

pyState.AugAssign

pySym.pyState.AugAssign.**handle**(*state*, *element*)

Attempt to handle the Python AugAssign element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.AugAssign*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to AugAssign. It is not meant to be called manually via a user.

Example

Example of ast.Assign is: x += 1

pyState.BinOp

pySym.pyState.BinOp.**handle**(*state*, *element*, *ctx=None*)

Attempt to handle the Python BinOp element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.BinOp*) – element from source to be handled
- **ctx** (int, *optional*) – context to resolve BinOp in if not current

Returns list contains pyObjectManager variables (Int/Real/etc)

Return type list

This function handles calls to BinOp. It is not meant to be called manually via a user.

Example

Example of ast.BinOp is: x + 1

pyState.BoolOp

pySym.pyState.BoolOp.**handle**(*state*, *element*)
Attempt to handle the Python BoolOp element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.BoolOp*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to BoolOp. It is not meant to be called manually via a user.

Example

Example of ast.BoolOp is: x == 1 and y == 2

pyState.Break

pySym.pyState.Break.**handle**(*state*, *element*)
Attempt to handle the Python Break element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Break*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to Break. It is not meant to be called manually via a user. Under the hood, it simply pops off the call stack until a loop change is seen (i.e.: we've left the for loop)

Example

Example of ast.Break is: break

pyState.Call

pySym.pyState.Call.**handle**(*state*, *element*, *retObj=None*)
Attempt to handle the Python Call element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Call*) – element from source to be handled
- **retObj** (pyState.ReturnObject, *optional*) – *retObj* is an optional input to specify a ReturnObject to be used ahead of time.

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.Call. It is not meant to be called manually via a user. A call will cause a context switch, populate variables, and set other internals. Upon return, the state will be inside the function.

Example

Example of ast.Call is: test()

pyState.Compare

pySym.pyState.Compare.**handle**(*state*, *element*, *ctx=None*)

Attempt to handle the Python Compare element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Compare*) – element from source to be handled
- **ctx** (*int, optional*) – *ctx* is an optional input to specify a context to be used when resolving this ast object

Returns list contains state objects either generated or discovered through handling this ast. – or – list contains True constraints derived from input ast element as z3 elements.

Return type list

This function handles calls to ast.Compare. It is not meant to be called manually via a user.

Example

Example of ast.Compare is: 1 < 2

pyState.Expr

pySym.pyState.Expr.**handle**(*state*, *element*)

Attempt to handle the Python Expr element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Expr*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.Expr. It is not meant to be called manually via a user.

Example

Example of ast.Expr is: test() (Note no assignment for call. This makes it an expression)

pyState.For

pySym.pyState.For.**handle**(*state*, *element*)
Attempt to handle the Python For element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.For*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.For. It is not meant to be called manually via a user.

Example

Example of ast.For is: for x in [1,2,3]

pyState.FunctionDef

pySym.pyState.FunctionDef.**handle**(*state*, *element*)
Attempt to handle the Python FunctionDef element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.FunctionDef*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.FunctionDef. It is not meant to be called manually via a user. Under the hood, it registers this function with the *state* object so that when it's referenced later it can be found.

Example

Example of ast.FunctionDef is: def test():

pyState.GeneratorExp

pySym.pyState.GeneratorExp.**handle**(*state*, *element*, *ctx=None*)
Attempt to handle the Python GeneratorExp element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.GeneratorExp*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.GeneratorExp. It is not meant to be called manually via a user. Under the hood, it converts the generator expression into a list comprehension and calls the handler for list comprehension.

Example

Example of ast.GeneratorExp is: x for x in [1,2,3] (note it's not inside List Comprehension brackets)

pyState.If

pySym.pyState.If.**handle**(*state*, *element*)

Attempt to handle the Python If element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.If*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.If. It is not meant to be called manually via a user. Under the hood, it resolves the conitional arguments, splits it's state, and takes both possibilities as the same time.

Example

Example of ast.If is: if x > 5:

pyState.ListComp

pySym.pyState.ListComp.**handle**(*state*, *element*, *ctx=None*)

Attempt to handle the Python ListComp element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.ListComp*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.ListComp. It is not meant to be called manually via a user. Under the hood, it re-writes the ast into an equivalent functional form, then calls that function symbolically.

Example

Example of ast.ListComp is: [x for x in range(10)]

pyState.Pass

pySym.pyState.Pass.**handle**(*state*, *element*)

Attempt to handle the Python Pass element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under

- **element** (*ast.Pass*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.Pass. It is not meant to be called manually via a user. Under the hood, it very simply pops off the current instruction and returns the updated state object as a list.

Example

Example of ast.Pass is: pass

pyState.Return

pySym.pyState.Return.**handle**(*state*, *element*)

Attempt to handle the Python Return element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Return*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.Return. It is not meant to be called manually via a user. Under the hood, it resolves the return element, sets the ReturnObject, and updates the state.

Example

Example of ast.Return is: return x

pyState.Subscript

pySym.pyState.Subscript.**handle**(*state*, *element*, *ctx=None*)

Attempt to handle the Python Subscript element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.Subscript*) – element from source to be handled
- **ctx** (*int , optional*) – Context to resolve this Subscript in (default is current context)

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.Subscript. It is not meant to be called manually via a user.

Example

Example of ast.Subscript is: x[5] = 2

pyState.UnaryOp

pySym.pyState.UnaryOp.**handle**(*state*, *element*, *ctx=None*)

Attempt to handle the Python UnaryOp element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.UnaryOp*) – element from source to be handled
- **ctx** (*int , optional*) – Context to resolve this UnaryOp in (default is current context)

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.UnaryOp. It is not meant to be called manually via a user.

Example

Example of ast.UnaryOp is: not True

pyState.While

pySym.pyState.While.**handle**(*state*, *element*)

Attempt to handle the Python While element

Parameters

- **state** (pyState.State) – pyState.State object to handle this element under
- **element** (*ast.While*) – element from source to be handled

Returns list contains state objects either generated or discovered through handling this ast.

Return type list

This function handles calls to ast.While. It is not meant to be called manually via a user.

Example

Example of ast.While is: while x < 10:

pyState.z3Helpers

A file to hold my helper items directly relating to z3

pySym.pyState.z3Helpers.**Z3_MAX_STRING_LENGTH = 256**

def varIsUsedInSolver(var,solver,ignore=None) – ” “” Determine if the given var (z3 object) is used in solver. Optionally ignore a list of constraints. ” “” logger.warn(“varIsUsedInSolver is deprecated. Use solver.var_in_solver instead”)

ignore = [] if ignore is None else ignore

If it’s a solo constraint, make it a list if type(ignore) is not None and type(ignore) not in [list, tuple]:

ignore = [ignore]

Sanity check assert isZ3Object(var), "Expected var to be z3 object, got type {} instead".format(type(var)) assert isinstance(solver,z3.Solver), "Solver must be type z3.Solver, got type {}".format(type(z3.Solver))

Remove any ignored assertions assertions = [ass for ass in solver.assertions() if ass not in ignore]

for ass in assertions:

if var.get_id() in [x.get_id() for x in pyState.get_all(ass)]: return True

return False

pySym.pyState.z3Helpers.**bvadd_safe**(*x*, *y*, *signed=False*)

BitVector addition overflow/underflow checks

Parameters

- **x,y** (*z3.BitVecRef or z3.BitVecNumRef*) – These variables are the two BitVecs that will be added together.
- **signed** (*bool, optional*) – Should this addition be treated as signed?

Returns tuple of z3 solver constraints to detect an overflow or underflow

Return type tuple

This function wraps Z3 C API functions to allow for a python interpretation of overflow and underflow checks. The returned tuple does not perform the addition, rather it is constraints that will perform the checks for overflow and underflow.

Example

If you want to verify the addition of x and y will not overflow/underflow:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x,y,z = z3.BitVecs('x y z',32)

In [5]: s.add(pyState.z3Helpers.bvadd_safe(x,y))

In [6]: s.add(x + y == z)

In [7]: s
Out[7]:
[Extract(32, 32, ZeroExt(1, x) + ZeroExt(1, y)) == 0,
 Implies(And(x < 0, y < 0), x + y < 0),
 x + y == z]

In [8]: s.check()
Out[8]: sat
```

pySym.pyState.z3Helpers.**bvdiv_safe**(*x*, *y*, *signed=False*)

BitVector division overflow check

Parameters

- **x,y** (*z3.BitVecRef or z3.BitVecNumRef*) – These variables are the two BitVecs that will be divided

- **signed** (*bool, optional*) – Should this division be treated as signed?

Returns tuple of z3 solver constraints to detect an overflow

Return type tuple

This function wraps Z3 C API functions to allow for a python interpretation of overflow checks. The returned tuple does not perform the division, rather it is constraints that will perform the checks for overflow.

Example

If you want to verify the division of x and y will not overflow:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x,y,z = z3.BitVecs('x y z',32)

In [5]: s.add(pyState.z3Helpers.bvdiv_safe(x,y))

In [6]: s.add(x / y == z)

In [7]: s
Out[7]: [Not(And(x == 1 << 31, y == 4294967295)), x/y == z]

In [8]: s.check()
Out[8]: sat
```

pySym.pyState.z3Helpers.**bvmul_safe**(*x*, *y*, *signed=False*)

BitVector multiplication overflow/underflow checks

Parameters

- **x,y** (*z3.BitVecRef or z3.BitVecNumRef*) – These variables are the two BitVecs that will be multiplied together.
- **signed** (*bool, optional*) – Should this multiplication be treated as signed?

Returns tuple of z3 solver constraints to detect an overflow or underflow

Return type tuple

This function wraps Z3 C API functions to allow for a python interpretation of overflow and underflow checks. The returned tuple does not perform the multiplication, rather it is constraints that will perform the checks for overflow and underflow.

Example

If you want to verify the multiplication of x and y will not overflow/underflow:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()
```

```
In [4]: x,y,z = z3.BitVecs('x y z',32)

In [5]: s.add(pyState.z3Helpers.bvmul_safe(x,y))

In [6]: s.add(x * y == z)

In [7]: s
Out[7]: [bvumul_noovfl(x, y), bvsmul_noudfl(x, y), x*y == z]

In [8]: s.check()
Out[8]: sat
```

pySym.pyState.z3Helpers.**bvsub_safe**(*x*, *y*, *signed=False*)

BitVector subtraction overflow/underflow checks

Parameters

- **x,y** (*z3.BitVecRef or z3.BitVecNumRef*) – These variables are the two BitVecs that will be subtracted.
- **signed** (*bool, optional*) – Should this subtraction be treated as signed?

Returns tuple of z3 solver constraints to detect an overflow or underflow

Return type tuple

This function wraps Z3 C API functions to allow for a python interpretation of overflow and underflow checks. The returned tuple does not perform the subtraction, rather it is constraints that will perform the checks for overflow and underflow.

Example

If you want to verify the subtraction of x and y will not overflow/underflow:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x,y,z = z3.BitVecs('x y z',32)

In [5]: s.add(pyState.z3Helpers.bvsub_safe(x,y))

In [6]: s.add(x - y == z)

In [7]: s
Out[7]:
[If(y == 1 << 31,
    x < 0,
    Implies(And(0 < x, 0 < -y), 0 < x + -y)),
 ULE(y, x),
 x - y == z]

In [8]: s.check()
Out[8]: sat
```

pySym.pyState.z3Helpers.**isInt**(*x*)

Wraps Z3 C API to perform isInt check on Real object x

Parameters **x** (*z3.ArithRef or z3.RatNumRef*) – Real variable from calls like z3.Real('x')

Returns z3.BoolRef asserting type x is Int (i.e.: ends in .0)

Return type z3.BoolRef

This function wraps Z3 C API functions to allow for a python interpretation of isInt. The returned value is a boolean that the input Real type x is an integer (i.e.: ends in .0). This call is the C API that performs the check at solve time, rather than entry time.

Example

If you want to verify that Real type x is an Int:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x = z3.Real('x')

In [5]: s.add(pyState.z3Helpers.isInt(x))

In [6]: s.add(x == 5.0)

In [7]: s
Out[7]: [IsInt(x), x == 5]

In [8]: s.check()
Out[8]: sat
```

pySym.pyState.z3Helpers.**isZ3Object**(*obj*)

Determine if the object given is a z3 type object

Parameters **x** (*any*) – Object can be any type

Returns True if object is known z3 type, False otherwise

Return type bool

This function is helpful if you want to verify that a given input object is a z3 type object. Under the cover, it runs the type operator against the object and compares it to known z3 types.

Example

If you want to verify x is a valid z3 object:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: x = z3.Real('x')
```

```
In [4]: assert pyState.z3Helpers.isZ3Object(x)
```

pySym.pyState.z3Helpers.**z3_bv_to_int**(*x*)

BitVector to Integer Z3 conversion

Parameters **x** (*z3.BitVecRef or z3.BitVecNumRef*) – BitVector variable to be converted to Int

Returns z3.ArithRef is an expression that will convert the BitVec into Integer inside Z3 rather than before insertion into the solver.

Return type z3.ArithRef

This function wraps Z3 C API functions to allow for a python interpretation of BitVec to Int conversions. The returned object is an expression that Z3 will evaluate as an Int rather than BitVec during solving.

Example

If you want to convert a BitVec into an Int:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x = z3.BitVec('x',32)

In [5]: y = z3.Int('y')

In [6]: x = pyState.z3Helpers.z3_bv_to_int(x)

In [7]: s.add(x == y)

In [8]: s
Out[8]: [BV2Int(x) == y]

In [9]: s.check()
Out[9]: sat
```

pySym.pyState.z3Helpers.**z3_int_to_bv**(*x*, *size=64*)

Integer to BitVector Z3 conversion

Parameters

- **x** (*z3.ArithRef or z3.IntNumRef*) – Int variable to be converted to BitVec
- **size** (*int, optional*) – BitVec bit size. If not specified, defaults to pyState.z3Helpers.Z3_DEFAULT_BITVEC_SIZE

Returns This is the BitVec reference for the associated Int

Return type z3.BitVecRef

This function wraps Z3 C API functions to allow for a python interpretation of Int to BitVec conversions. The returned object is an expression that Z3 will evaluate as an BitVec rather than Int during solving.

Example

If you want to convert an Int int into a BitVec:

```
In [1]: import z3

In [2]: from pySym import pyState.z3Helpers

In [3]: s = z3.Solver()

In [4]: x = z3.BitVec('x',8)

In [5]: y = z3.Int('y')

In [6]: y = pyState.z3Helpers.z3_int_to_bv(y,8)

In [7]: s.add(x == y)

In [8]: s
Out[8]: [x == int2bv(y)]

In [9]: s.check()
Out[9]: sat
```

pySym.pyState.z3Helpers.**z3_matchLeftAndRight**(*left*, *right*, *op*)

Appropriately change the two variables so that they can be used in an expression

Parameters

- **left,right** (pyObjectManager.Int.Int *or* pyObjectManager.Real.Real *or* pyObjectManager.BitVec.BitVec *or* pyObjectManager.Char.Char) – Objects to be matched
- **op** (*ast.**) – Operation that will be performed

Returns (z3ObjectLeft,z3ObjectRight) tuple of z3 objects that can be used in an expression

Return type tuple

The purpose of this function is to match two pyObjectManager.* variables to a given ast operation element. Z3 needs to have matched types, and this call will not only match the objects, but also attempt to concretize input wherever possible.

Example

If you want to auto-match BitVector sizes:

```
In [1]: import z3, pyState.z3Helpers, ast

In [2]: from pySym.pyObjectManager.BitVec import BitVec

In [3]: from pySym.pyState import State

In [4]: state = State()

In [5]: x = BitVec("x",0,16,state=state)

In [6]: y = BitVec("y",0,32,state=state)
```

```
In [7]: l,r = pyState.z3Helpers.z3_matchLeftAndRight(x,y,ast.Add())

In [8]: s = z3.Solver()

In [9]: s.add(l + r == 12)

In [10]: s
Out[10]: [SignExt(16, 0x@0) + 0y@0 == 12]

In [11]: s.check()
Out[11]: sat
```

Module contents

class pySym.pyState.**ReturnObject**(*retID*, *state=None*)

Bases: object

copy()

Copies ReturnObject into an identical instance

Returns ReturnObject with the same ID and State as the previous one

Return type *pySym.pyState.ReturnObject*

retID

state

class pySym.pyState.**State**(*path=None*, *solver=None*, *ctx=None*, *functions=None*, *simFunctions=None*, *retVar=None*, *callStack=None*, *backtrace=None*, *retID=None*, *loop=None*, *maxRetID=None*, *maxCtx=None*, *objectManager=None*, *vars_in_solver=None*, *project=None*)

Bases: object

Defines the state of execution at any given point.

Call(*call*, *func=None*, *retObj=None*, *ctx=None*)

Input: call = ast.Call object (optional) func = resolved function for Call (i.e.: state.resolveCall(call)). This is here to remove duplicate calls to resolveCall from resolveObject (optional) ctx = Context to execute under. If left blank, new Context will be created.

Action: Modify state in accordance w/ call

Returns: ReturnObject the describes this functions return var

Return(*retElement*)

Input: retElement = ast.Return element

Action: Set return variable appropriately and remove the rest of the instructions in the queue

Returns: Nothing for now

addConstraint(**constraints*)

Input: constraints = Any number of z3 expressions to use as a constraint

Action: Add constraint given

Returns: Nothing

any_char(*var*, *ctx=None*)

Input: var == variable name. i.e.: "x" –or– ObjectManager object (i.e.: Char) (optional) ctx = context if not current one

Action: Resolve a possible value for this variable

Return: Discovered variable or None if none found

any_int(*var*, *ctx=None*, *extra_constraints=None*)

Input: var == variable name. i.e.: "x" –or– ObjectManager object (i.e.: Int) (optional) ctx = context if not current one (optional) extra_constraints = tuple of extra constraints to temporarily place on the solve.

Action: Resolve possible value for this variable

Return: Discovered variable or None if none found

any_list(*var*, *ctx=None*)

Input: var == variable name. i.e.: "x" (optional) ctx = context if not current one

Action: Resolve possible value for this variable (list)

Return: Discovered variable or None if none found

any_n_int(*var*, *n*, *ctx=None*)

Input: var = variable name. i.e.: "x" –or– ObjectManager object (i.e.: Int) n = number of viable solutions to find (i.e.: 5) (optional) ctx = context if not current one

Action: Resolve n possible values for this variable

Return: Discovered n values or [] if none found

any_n_real(*var*, *n*, *ctx=None*)

Input: var = variable name. i.e.: "x" –or– ObjectManager object (i.e.: Int) n = number of viable solutions to find (i.e.: 5) (optional) ctx = context if not current one

Action: Resolve n possible values for this variable

Return: Discovered possible values or [] if none found

any_real(*var*, *ctx=None*)

Input: var == variable name. i.e.: "x" –or– ObjectManager Object (i.e.: Int) (optional) ctx = context if not current one

Action: Resolve possible value for this variable

Return: Discovered variable or None if none found Note: this function will cast an Int to a Real implicitly if found

any_str(*var*, *ctx=None*)

Input: var == variable name. i.e.: "x" –or– ObjectManager object (i.e.: String) (optional) ctx = context if not current one

Action: Resolve a possible value for this variable

Return: Discovered variable or None if none found

backtrace

callStack

copy()
Return a copy of the current state

copyCallStack()
Make a copy of the call stack, avoiding deepcopy Returns a copy of the current call stack

ctx

functions

getVar(*varName*, *ctx=None*, *varType=None*, *kwargs=None*, *softFail=None*)
Convinence function that adds current ctx to getVar request

isSat(*extra_constraints=None*)

Input: extra_constraints: Optional list of extra constraints to temporarily add before checking for sat.

Action: Checks if the current state is satisfiable Note, it uses the local and global vars to create the solver on the fly

Returns: Boolean True or False

lineno()
Returns current line number. If returning from a call, returns the return line number Returns None if the program is done

loop

maxCtx

maxRetID

objectManager

path

popCallStack()

Input: Nothing

Action: Pops from the call stack to the run stack. Adds call to completed state

Returns: True if pop succeeded, False if there was nothing left to pop

popConstraint()
Pop last added constraint This doesn't seem to work...

printVars()

Input: Nothing

Action: Resolves current constraints and prints viable variables

Returns: Nothing

pushCallStack(*path=None*, *ctx=None*, *retID=None*, *loop=None*)
Save the call stack with given variables Defaults to current variables if none given

recursiveCopy(*var*, *ctx=None*, *varName=None*)
Create a recursive copy of the given ObjectManager variable. This includes creating the relevant z3 constraints (optional) ctx = Context to copy in. Defaults to ctx 1 (RETURN_CONTEXT). (optional) varName = Specify what to name this variable Returns the copy

registerFunction(*func*, *base=None*, *simFunction=None*)

Input: func = ast func definition (optional) base = base path to import as (i.e.: "telnetlib" if importing "telnetlib.Telnet") (optional) simFunction = Boolean if this should be treated as a simFunction and therefore not handled symbolically. Defaults to False.

Action: Register's this function as being known to this state

Returns: Nothing

remove_constraints(*constraints*)
Removes the given z3 constraints.

Args: constraints (list): List of z3 constraints to remove from the solver.

Returns: int: Returns how many constraints were actually removed.

resolveCall(*call*, *ctx=None*)

Input: call = ast.Call object (optional) ctx = Context to resolve under

Action: Determine correct ast.func object

Returns: ast.func block

resolveObject(*obj*, *parent=None*, *ctx=None*, *varType=None*, *kwargs=None*)

Input:

obj = Some ast object (i.e.: ast.Name, ast.Num, etc) special object "PYSYM_TYPE_RETVAL" (int) will resolve the last return value

(optional) parent = parent node of obj. This is needed for resolving calls (optional) ctx = Context other than current to resolve in (optional) varType = Type of the var to resolve. Needed if resolving a var that doesn't exist yet (optional) kwargs = kwargs for the var, needed if resolving a var that doesn't exist yet

Action: Resolve object into something that can be used in a constraint

Return:

Resolved object ast.Num == int (i.e.: 6) ast.Name == pyObjectManager object (Int, Real, BitVec, etc) ast.BinOp == z3 expression of BinOp (i.e.: x + y)

retID

retVar

setVar(*varName*, *var*, *ctx=None*)
Convinence function that adds current ctx to setVar request

simFunctions

solver

step()
Move the current path forward by one step Note, this actually makes a copy/s and returns them. The initial path isn't modified. Returns: A list of paths or empty list if the path is done

var_in_solver(*var*, *ignore=None*)
Checks if the variable given is in the z3 solver.

pySym.pyState.**duplicateSort**(*obj*)

Input:

obj = z3 object to duplicate kind (i.e.: z3.IntSort()) –or– pyObjectManager type object (i.e.: Int)

Action: Figure out details of the object and make duplicate sort

Return: (class, kwargs) Duplicate pyObjectManager class object for this type (i.e.: Int)

pySym.pyState.**get_all**(*f*, *rs=[]*)

```
>>> x,y = Ints('x y')
>>> a,b = Bools('a b')
>>> get_all(Implies(And(x+y==0,x*2==10),Or(a,Implies(a,b==False))))
[x, y, a, b]
```

pySym.pyState.**hasRealComponent**(*expr*)

Checks for Real component to a z3 expression

Parameters **expr** (*z3 expression object*) –

Returns True if it has real componenet, False otherwise

Return type bool

Checks if expression contains a real/non-int value or variable. This is genereally used in determining proper variable type to create. Z3 will cast Int to Real if you don't select the right type, which add extra complexity to the solving.

Example

Confirm that a Z3 expression has a Real component:

```
In [1]: import z3

In [2]: from pySym import pyState

In [3]: r = z3.Real('r')

In [4]: i = z3.Int('i')

In [5]: pyState.hasRealComponent(r + i == 5)
Out[5]: True
```

pySym.pyState.**replaceObjectWithObject**(*haystack*, *fromObj*, *toObj*, *parent=None*)

Generic search routine to replace an arbitrary object with another

Parameters

- **haystack** (*any*) – Where to search
- **fromObj** (*any*) – What to replace
- **toObj** (*any*) – What to replace with
- **parent** (*any, optional*) – (depreciated) What the parent object is. This option will be removed.

Returns True/False if the object was successfully replaced.

Return type bool

Find instance of fromObj in haystack and replace with toObj. This is used to ensure we know which function return is ours. Also now matches against lineno, col_offset and type. This will likely fail on polymorphic python code

Example

Replace the ast.Compare object with a Return object:

```
In [1]: import ast

In [2]: from pySym import pyState

In [3]: ret = pySym.pyState.ReturnObject(5)

In [4]: s = ast.parse("if 5 > 2:\n\tpass").body[0]

In [5]: print(s.test)
<_ast.Compare object at 0x7f563acb1b70>

In [6]: assert pyState.replaceObjectWithObject(s,s.test,ret)

In [7]: print(s.test)
<pySym.pyState.ReturnObject object at 0x7f563b4c1048>
```

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

U

V

Z