Verification of Python Code with a Dynamic Object Model

Master's Thesis Project Description

Edgars Vitolins

Supervisors: Dr. Marco Eilers, Prof. Dr. Peter Müller

October 2023

1 Introduction

One of the reasons why Python has become so popular is its expressiveness. Functionality that might be fixed in other languages is often highly customizable in Python. One great example of this is Python's dynamic object model, which allows to redefine basic operations such as attribute access and object creation through the use of "magic methods". However, this flexibility makes it difficult to verify various properties about Python programs, since something as simple as an attribute access can have arbitrary effects.

1.1 Python's object model and magic methods

```python
class Dynamic:
    def __getattr__(self, name):
        return "access to nonexistent field " + name

d = Dynamic()
print(d.hello)
# prints "access to nonexistent field hello"
```

Listing 1: Access to nonexistent field

```python
class Dynamic:
    # ignore setting attr
    def __setattr__(self, name, val):
        print(f"create foo_{name} = {val}")
        self.__dict__[f"foo_{name}"] = val

d = Dynamic()
d.i = 300        # prints "create foo_i = 300"

print(d.foo_i)   # prints "300"
print(d.i)       # AttributeError: 'Dynamic' object has no attribute 'i'
```

Listing 2: Setting a field

In Python, classes can be customized using special methods called "magic methods". These are methods with special names, listed in the Python documentation [1], that allow to implement operator overloading, customizing attribute access and class creation, and others.
Listing 1 shows a simple example of how customizable attribute access in Python is. In other languages, just looking at a class definition might be enough to determine whether an access to a certain attribute will succeed. However, in Python, an access to an attribute can be intercepted, with potentially arbitrary functionality.
The way attribute access customization in Python works is that first Python will try to call the method `__getattribute__` with the name of the attribute as a string argument. If `__getattribute__` raises an `AttributeError`, or if no such method is defined, and the attribute name is not found in `__dict__` (the so-called instance directory, which is the data structure Python uses to store attributes), then Python

will try to call `__getattr__`. In Listing 1, since there is no `__getattribute__` defined and since `hello` has not been created, Python will call `__getattr__`. So even though the attribute `hello` was never created, an access to it did not raise an `AttributeError` exception.

Listing 2 shows how setting an attribute can be customized as well. In the example, `__setattr__` creates an attribute, but with "foo_" prepended to its name.
In principle, `__getattribute__`, `__getattr__` and `__setattr__` allow attribute access and manipulation to execute arbitrary computation and manipulate state since there are no restrictions on what can be done in those methods.

1.2 Viper and Nagini

Nagini [2] is a Python verifier based on the Viper [3] verification infrastructure. It works by encoding Python programs into the Viper intermediate language, and then uses existing Viper infrastructure to verify those programs. Viper is a tool-chain on which several frontends have been built, that takes programs encoded in the Viper intermediate language and verifies them using an SMT solver.
Currently, Nagini only allows the customization of some magic methods, and it does not allow the modification of `__getattribute__`, `__getattr__` and `__setattr__`. Instead of implementing attribute access the way its done in Python with a `__dict__` instance directory lookup, Nagini models instance attribute access as a simple value lookup of a fixed memory location. Since `__getattr__` and `__setattr__` definitions are currently not allowed in Nagini, Listing 1 and 2 cannot be verified.

For verification purposes, correct modeling of instance attribute access in Python is difficult because a field read involves several method invocations with arbitrary computations and state modifications, which adds a lot of complexity, potentially requires writing much more complex specifications, and comes at a significant performance cost.
More information on how Nagini implements the Python object model and the reasons behind it can be found in section `2.2.4` of [2].

2 Thesis Statement

The purpose of this thesis is to define and implement a more accurate Python object model specification in Nagini that would allow some instance attribute customization. More concretely, this will mean allowing Nagini to verify programs that make use of `__getattribute__`, `__getattr__` and `__setattr__` "magic methods".

3 Core goals

3.1 Collect examples

Collect some examples of how magic methods are used in popular Python libraries. Condense these examples into compact code snippets that serve as the focus of the thesis.

3.2 Design and implement prerequisites in Nagini

Add or extend features in Nagini that will be required to encode a more accurate object model. For example, the current implementation of Nagini does not handle strings well. Rework or otherwise adjust the way strings are implemented in Nagini for the purposes of this thesis (in particular, attribute names, stored as strings, are used as keys in instance dictionaries).
Change the way Nagini handles multiple concurrent accesses to `dict` data structures, given how Python implements attribute accesses in instance dictionaries.

3.3 More accurate Python model specification

Define a new Python object model encoding to Viper that more accurately reflects Python's actual semantics, for the purpose of implementing instance attribute access customization using `__getattribute__`, `__getattr__` and `__setattr__` (focusing on attribute accesses and not function calls). This could involve adding more specification constructs if needed, or redesigning currently existing specifications. Define requirements for what magic methods are allowed to do, for example, that they should be pure functions with no side effects.

3.4 Explore switch between current and complex versions

Explore various approaches as to how the new functionality of verifying magic methods could be combined with the existing Nagini model. One way of combining these two verification approaches is by processing classes without magic methods with the current simplified model, and only applying the more complex magic method model when it is necessary, as well as allowing the user to switch between the two using ghost code.
Explore how using the more complex model conflicts with Nagini's reliance on enforcing and assuming behavioral subtyping.
This goal also involves designing annotations in such a way that would allow switching between the more complex magic method model and the current simplified model as easily as possible.
The expected outcome for this goal is either a specification mechanism and an encoding for combining and switching between the existing view and the more complex view from Core Goal 3.3 or a detailed description of what makes such a dynamic switch impossible, impractical, or outside the scope of a Master's Thesis.

3.5 Implementation in Nagini

Implement functionality from Core Goal 3.3 and 3.4 in Nagini, focusing on verifying concrete code snippets from Core Goal 3.1.

4 Extension goals

4.1 Metaclasses

Explore if and how verifying magic methods as discussed in Core Goal 3.3 relates to verifying Python Metaclasses. Discuss what functionality would still need to be implemented or implement said functionality in Nagini to allow it to verify some Python Metaclasses.

4.2 Different simplified models

Core Goal 3.3 will involve defining some simplified Python Object model, but there might be other approaches that could have been taken. Discuss what those other approaches are, and how they compare to the approach taken in Core Goal 3.3.

4.3 Method calls based on dynamic lookups

This goal can be seen as the function side to the attribute overriding from Core Goal 3.3. Like with attribute accesses, Python has a very flexible function calling mechanism. Explore what is necessary in order to implement verification of a more accurate model of this function calling mechanism in Nagini. Consider using the results from [4].

4.4 Other magic methods

Implement verification or more "magic methods" beyond `__getattribute__`, `__getattr__` and `__setattr__` from Core Goal 3.3 and what is already part of Nagini.

References

[1] Python's data model documentation. https://docs.python.org/3/reference/datamodel.html. Last accessed: 15.10.2023.

[2] Marco Eilers. *Modular Specification and Verification of Security Properties for Mainstream Languages*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2022.

[3] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.

[4] Benjamin Weber. Automating modular reasoning about higher-order functions, November 2017. Available at https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Benjamin_Weber_MA_report.pdf.