Verification of Python Code with a Dynamic Object Model

Master's Thesis Project Report

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

Department of Computer Science, ETH Zurich
Programming Methodology Group

May 2024

Abstract

Formal software verification is a powerful method that can give strong guarantees about software correctness. Python is a popular programming language, and this popularity, among other reasons, is due to the expressiveness provided by its dynamic object model. With the combination of `__dict__`, and magic methods `__getattribute__`, `__getattr__` and `__setattr__`, Python programs offer a lot of flexibility on how object attributes are accessed and modified. However, this makes it challenging to reason about Python programs in a modular way. Nagini is a Python verifier, based on Vipers backend infrastructure, and it does not support Python's dynamic object model. In this thesis we propose a technique for reasoning about Python's dynamic object model in the Nagini verifier. We find that including `__dict__` and the mentioned magic methods in Nagini, with some restrictions, allows us to verify commonly seen code patterns that take advantage of Python's dynamic object model, while keeping specifications simple and verification modular, and we implement and verify this proposed model.

Contents

1 Introduction

Formal software verification is an important tool that can be used to assure that software satisfies requirements. It can provide stronger guarantees of correctness when compared with various forms of testing, such as unit or fuzz testing. One important approach to formal software verification, based on Hoare logic [1], is through the use of permission logics, such as Separation logic [2] and Implicit Dynamic Frames [3], which facilitate reasoning about memory and state.

```
1  class Dynamic:
2      def __getattr__(self, name: str) -> int:
3          return 0
4
5  d = Dynamic()
6  print(d.a)       # prints "0"
7  d.a = 100
8  print(d.a)       # prints "100"
```

Listing 1: `__getattr__` as default value.

One of the keys to Python's popularity 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". Listing 1 shows a simple example where the `__getattr__` defines a default value for instance attributes that have not been defined. However, this flexibility makes it difficult to modularly reason about various properties about Python programs, since something as simple as an attribute access can have arbitrary effects.

Nagini [4] is a Python verifier based on the Viper [5] 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 use of `__getattribute__`, `__getattr__` and `__setattr__`. Instead of implementing attribute access the way it is 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 Nagini does not allow to define `__getattr__`, Listing 1 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, and potentially requires writing much more complex specifications, which can potentially come at a performance cost.

In this thesis we define and implement a more accurate Python object model specification in Nagini that allows some instance attribute customization, enabling the use of common magic method code patterns, while still allowing for modular verification, and keeping specifications simple and performance acceptable. This means allowing Nagini to verify programs that make use of `__dict__`, as well as `__getattribute__`, `__getattr__` and `__setattr__` "magic methods", with some limitations.

In Section 2 we briefly explain permission-based deductive program verification in the context of Implicit Dynamic Frames. In the rest of Section 2 we introduce Python's dynamic object model, taking a closer look at each of `__getattribute__`, `__getattr__`, `__setattr__`, and `__dict__`. In Section 3 we summarize how attribute access magic methods are used in real code examples by identifying three commonly used patterns, that later serve as the basis for motivating examples. In Section 4 we present a model that allows to reason about Python's dynamic object model, and compares it with the old, simple object model. The new model is designed to verify the real code patterns identified in Section 3, and be backwards compatible with the previous object model. We introduce motivating examples and consider each of `__getattribute__`, `__getattr__`, `__setattr__`, and `__dict__` separately, in terms of design and encoding. In Section 5 we cover some alternative design choices from those taken in Section 4. In Section 6 we discuss what parts of the design have been implemented and tested. In Section 7 we discuss two additional contributions that are necessary pre-requisites, which

are, Alternative Dictionary Encoding - Keydict, and the encoding of strings as sequences of integers using Viper's `Seq[Int]`, in order to support the verification of the motivating examples. In Section 8 we evaluate the implementation on the motivating examples as well as some existing Nagini tests, and finally, in Section 9 we summarize this report.

2 Background

2.1 Permission Logics

Permission logics enables reasoning about memory and state by assigning every unit of memory a permission and carefully handling the "chain of custody" of these permissions. What this means is that it is clear which parts of the program hold what permissions at all times, and each function caller hands over all the necessary permissions to the callee, that then returns those permissions back to the caller.

Every operation on each unit of memory requires some permission. The permissions required by each operation denote this operation's footprint. If the footprint of a certain operation is disjoint from the footprint of an expression, then this operation does not change the expression's value. For example, if the operation is a field `x` write, and the expression is a field `x` read, then the footprint of the field `x` write will overlap with the footprint of the expression of field `x` read.

A permission for a value can be split into fractions, where the full permission is needed to change the value, and at least some fraction of the permission is needed to read the value. This means that, for example, two separate parts of the program can hold `1/2` of a permission to a field, which allows each part to read this field, however, in order to changed the field's value these two permissions need to be combined together, to form a full permission.

In Nagini, through the use of the Viper verification infrastructure, this permission functionality is provided by Implicit Dynamic Frames [6] which facilitates local reasoning of heap locations in a concurrent setting.

There are two verifiers for the Viper language - Silicon [7], which is a symbolic-execution-based verifier, and Carbon [5], which is a verficiation-condition-generation based verifier.

2.2 Python's object model and magic methods

In Python, classes can be customized using special methods called "magic methods". These are methods with special names, always starting and ending with __ ("dunder"), listed in the Python documentation [8], that allow to implement operator overloading, customizing attribute access and class creation, and others.

```
1 class Dynamic:
2     def __init__(self) -> None:
3         self.a = 10
4
5 d = Dynamic()
6 print(d.a)       # prints "10"
```

Listing 2: Initialization using `__init__`.

For example, class instances in Python are initialized using the `__init__` magic method, shown in Listing 2, which is called right after instance creation, and is used to initialize a newly created instance.

```
1 class Dynamic:
2     def __init__(self) -> None:
3         print('a' in self.__dict__)      # prints "False"
4         self.a = 10
```

```
5           print('a' in self.__dict__)      # prints "True"
6
7           self.__dict__['b'] = 20
8           print(self.b)                     # prints "20"
9
10  g = Dynamic()
```

Listing 3: Instance attributes are stored in `__dict__`.

Listing 3 shows how instance attributes, created inside `__init__` or elsewhere, are stored in the special namespace attribute `__dict__`. This creation and access goes both ways - all instance attributes can be accessed and modified in `__dict__`, and anything in the `__dict__` can be accessed and modified as an instance attribute using the (dot) operator.

```
1   class Dynamic:
2       def __init__(self) -> None:
3           print('a' in self.__dict__)      # prints "False"
4           print(self.a)                     # AttributeError
5
6   g = Dynamic()
```

Listing 4: `AttributeError` for nonexistant fields.

Trying to access an attribute that does not exist, and is not found in `__dict__`, results in an `AttributeError` exception, as shown in Listing 4.

```
1   #include <iostream>
2
3   class MyClass {
4   public:
5       int x = 0;
6       int my_func() { return x + 10; };
7       // ... various other functions (except constructors)
8       // ...
9   };
10
11  int main() {
12      MyClass c {};
13      std::cout << c.x << "\n";             // prints "0"
14      std::cout << c.my_func() << "\n";     // prints "10"
15      std::cout << c.abc << "\n";           // error: ... no member named 'abc'
16  }
```

Listing 5: C++ example of attribute (member) access

In other languages, just looking at the set of attributes in the class definition might be enough to determine whether an access to a certain attribute will succeed, for example, in Listing 5 no matter the functions defined in (:7), an access to `c.x` will return the instance attribute (in C++ usually referred to as "instance member variable") `x` (:5). This is because C++ does not permit (dot) operator overloading. The same is true for `x.my_func()` call (:14), which will invoke `my_func()` (:6). Trying to access an attribute that has not been defined (:15) will fail, since there is no way to define some default attribute value.

```
1   class MyClass:
2       def __init__(self):
3           self.x = 10
4
5       def __getattr__(self, name: str):
6           return 20
7
8       def __getattribute__(self, name: str):
9           if name == "y":
10              return 30
11          else:
12              return object.__getattribute__(self, name)
13
```

```
14 def hidden_code(self, name):
15     try:
16         if _getattribute := getattr(self, "__getattribute__", False):
17             return _getattribute(name)
18         return object.__getattribute__(self, name)
19     except AttributeError as e:
20         if _getattr := getattr(self, "__getattr__", False):
21             return _getattr(name)
22         raise e
23
24 if __name__ == "__main__":
25     c = MyClass()
26     try:
27         print(hidden_code(c, "x"), end=" ")       # c.x       prints "10"
28     except Exception as e:
29         print(e, end=" ")
30
31     try:
32         print(hidden_code(c, 'abcd'), end=" ")    # c.abcd    prints "20"
33     except Exception as e:
34         print(e, end=" ")
35
36     try:
37         print(hidden_code(c, 'y'), end=" ")       # c.y       prints "30"
38     except Exception as e:
39         print(e, end=" ")
```

Listing 6: `__getattr__` and `__getattribute__` approximate behavior w.r.t. instance attribute access.

However, in Python, an access to an attribute can be intercepted, with potentially arbitrary functionality. To develop a better intuition on how the Python data model works, Listing 6 approximates the behaviour of `__getattr__` and `__getattribute__` as it relates to instance attributes and methods. Note that this is just an approximation, and does not capture the full semantics of `__getattr__` and `__getattribute__` as it relates to other Python functionality besides instance attributes. Listing 6 does not show how class attribute access is resolved, especially when the class attribute is defined in a parent class. For a more comprehensive discussion of `__getattr__` and `__getattribute__` functionality refer to [8].

In Listing 6 `hidden_code()` (:14) models the behavior of (dot) operator accessing instance attributes (:26, :27, :28), and the output to running Listing 6 is `10, 20, 30`. Removing just the definition of `__getattribute__` (:8) would produce the output `10, 20, 20`. Removing just the definition of `__getattr__` would produce the output `10 'MyClass' object has no attribute 'abcd' 30`, and removing the definitions of both `__getattr__` and `__getattribute__` would produce the output of `10 'MyClass' object has no attribute 'abcd' 'MyClass' object has no attribute 'y'`.
Replacing calls to `hidden_code()` with the corresponding attribute accesses with the (dot) operator (as shown in the comments) for (:26, :27, :28) and performing the experiments again yields the same results. Same behavior, with the same results, applies to class method access as well, and is shown in B.1.
A more comprehensive discussion of the Python data model can be found at [8].

The remainder of this chapter discusses each of `__getattribute__`, `__getattr__` and `__setattr__` in a little more detail.

```
1 class Dynamic:
2     def __getattr__(self, name: str) -> object:
3         return f"access to nonexistent field '{name}'"
4
5 d = Dynamic()
6 print(d.a)       # prints "access to nonexistent field a"
7 d.b = 10
8 print(d.b)       # prints "10"
```

Listing 7: `__getattr__` as default value when an attribute is not found.

In Listing 7, when `d.a` is not found in `__dict__` attribute directory, `__getattr__` is called. Note that `__getattr__` gets called only if the attribute is not found in `__dict__`, and, in Listing 7, access to `d.b` is not affected by `__getattr__`.

```
1  class Dynamic:
2      def __init__(self) -> None:
3          self.a = 10
4          self.x = -1
5
6      def foo(self):
7          return -1
8
9      def __getattribute__(self, name: str) -> object:
10         if name == "x" or name == "y":
11             return 20
12         elif name == "foo":
13             return lambda: 30
14         else:
15             return object.__getattribute__(self, name)
16
17 g = Dynamic()
18 print(g.a)       # prints "10"
19 print(g.x)       # prints "20"
20 print(g.foo())   # prints "30"
21 print(g.b)       # AttributeError
```

Listing 8: Access to nonexistent field.

As stated before, unlike `__getattr__`, which is only called when an attribute is not found in `__dict__`, or when `__getattribute__` raises an `AttributeError`, magic method `__getattribute__` gets called on every (dot) operator access, including for attributes that are also found in `__dict__`, and for methods.

In Listing 8, trying to access `g.a` (:18) calls `__getattribute__` (:9) with `name == "a"`, returning `object.__getattribute__` (:15), reverting to the default Python attribute access behavior, which returns the attribute `a` (:3).

The result of accessing `g.x` (:19) is different, and does not revert to the default Python behavior. Instead, `__getattribute__` returns integer `20` (:11), without looking at `__dict__`, which contains an entry for `x` (:4).

Similarly with a call to `g.foo()` (:20), even though there is a function by that name (:6), that function is not called. Instead, `__getattribute__` returns an anonymous function (:13).

Trying to access `g.b` (:21), same as with `g.a` (:18) will fall back to the default Python behavior (:15), however, unlike `g.a` that has an entry in `__dict__`, there is no such entry for `g.b` and an `AttributeError` exception is raised. Had there been a magic method `__getattr__` defined for `class Dynamic` as well, then an `AttributeError` exception would have triggered a call to `__getattr__`, as shown in Listing 7.

```
1  class Dynamic:
2      def __setattr__(self, name: str, val: int) -> None:
3          self.__dict__[f"foo_{name}"] = val
4
5  d = Dynamic()
6  d.i = 10
7  print(d.foo_i)       # prints 10
8  print(d.i)           # AttributeError
```

Listing 9: Setting an attribute.

Attribute creation and modification is customized with `__setattr__`, as shown in in Listing 9.

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.

Attributes can also be accessed and modified with `getattr()` and `setattr()`, which provide the same functionality as trying to access an attribute using the (dot) operator, but additionaly allow to pass in the name of the attribute with an expression. For example, trying to read `g.x` is equivalent to `getattr(g, 'x')`, however, there is no equivalent (dot) operator access to something like `getattr(g, prefix + 'x')`, when `prefix` is only known at runtime. Both `getattr()` and `setattr()` function the same as (dot) operator access with regards to magic methods `__getattribute__`, `__getattr__` and `__setattr__`, as well as `__dict__`.

Note that, in Listing 9, trying to set an attribute with something like `setattr(self, name, val)`, instead of using `__dict__` directly (:3), would result in infinite recursion on `__setattr__`.

3 Real code examples

After looking at a number of Python libraries in search for typical usages of `__getattribute__`, `__getattr__` and `__setattr__`, we have identified three code patterns, which we discuss in this Section. The code patterns from these examples inspired three of the five motivating examples, that showcase the new dynamic object model functionality of Section 4. These examples were taken from the Python libraries of mkdocs [9], SCIONlab [10], matplotlib [11], Spark [12], Scrapy [13], Numpy [14], PyTorch [15], Keras [16], and TensorFlow [17]. The old Nagini object model does not support the use of `__getattr__`, `__setattr__`, and `__getattribute__`, and therefore cannot verify the code patterns below.

Besides the three identified code patterns, we also found some other ways in which `__getattribute__`, `__getattr__` and `__setattr__` are used, which are not supported by our proposed dynamic object model. We cover these additional patterns in Section 3.4.

3.1 Wrapper pattern

In the Listings below, the instance that defines `__getattr__` is a wrapper to some other class instance, which is stored as an attribute, in an OOP composition pattern. When some attribute access to `__dict__` fails, that same attribute is looked up in the wrapped instance as well.

```
1 def __getattr__(self, key):
2     if key in ('option', 'warnings'):
3         raise AttributeError
4     return getattr(self.option, key)
```

Listing 10: `__getattr__` code snippet from mkdocs, showing the wrapper pattern.

```
1 def __getattr__(self, attr):
2     return getattr(self.wrapped, attr)
```

Listing 11: `__getattr__` code snippet from SCIONLab, showing the wrapper pattern.

```
1 def __getattr__(self, name):
2     try:
3         return self._dict[name]
4     except KeyError:
5         raise AttributeError(
6             f"'Spines' object does not contain a '{name}' spine")
```

Listing 12: `__getattr__` code snippet from Matplotlib, showing the wrapper pattern.

Consider Listings 10 (mkdocs[1]), 11 (SCIONLab[2]), and 12 (Matplotlib[3]) which show various definitions of `__getattr__`. Listings 10 and 11 return some version of `self.some_instance_attribute.attribute_name`, and Listing 12 works similarly, but instead of wrapping an instance of a class, it wraps an instance of a dictionary, or some other type that defines the `[]` operator.

3.2 Prefix pattern

What the following Listings have in common is that for all three magic methods `__getattr__`, `__setattr__`, and `__getattribute__` the string value of the attribute name changes the resulting behavior, and that depending on that value, in this case on the prefix, these magic methods will return different results.

```
def __getattr__(key: str) -> Any:
    if key.startswith("__"):
        raise AttributeError(key)
    if hasattr(MissingPandasLikeScalars, key):
        raise getattr(MissingPandasLikeScalars, key)
    if hasattr(MissingPandasLikeGeneralFunctions, key):
        return getattr(MissingPandasLikeGeneralFunctions, key)
    else:
        raise AttributeError("module 'pyspark.pandas' has no attribute '%s'" % (key))
```

Listing 13: `__getattr__` code snippet from Apache Spark, showing the prefix pattern.

```
def __setattr__(self, name: str, value: Any) -> None:
    if not name.startswith("_"):
        raise AttributeError(f"Use item[{name!r}] = {value!r} to set field value")
    super().__setattr__(name, value)
```

Listing 14: `__setattr__` code snippet from Scrapy, showing the prefix pattern.

```
def __getattribute__(self, name):
    if name.startswith('__'):
        return object.__getattribute__(self, name)
    variable = object.__getattribute__(self, 'value')
    if hasattr(variable, name) and name not in self.__class__.__dict__:
        return getattr(variable, name)
    return object.__getattribute__(self, name)
```

Listing 15: `__getattr__` code snippet from Matplotlib, showing the prefix pattern.

Consider Listings 13 (Apache Spark[4]), 14 (Scrapy[5]), and 15(Matplotlib[6]) which show definitions of `__getattr__`, `__setattr__`, and `__getattribute__` from different projects. Listing 13 has a special case for when the attribute name starts with `"__"` (:2), same as Listing 15 (:2). Similarly, Listing 14 (:2) also has a special case, this time for `"_"`.

3.3 Lock pattern

In all three of the following examples, `__setattr__` will create or modify an attribute, but only if another special attribute has a certain value, otherwise the instance is immutable.

[1] https://github.com/mkdocs/mkdocs/blob/30e8142315127bcc2b05ee14ff99f300a7126a87/mkdocs/config/config_options.py#L556

[2] https://github.com/netsec-ethz/scionlab/blob/0e39dc1188a860de570a5b56ac266cea3b6f2f39/scionlab/hostfiles/scionlab-config#L608

[3] https://github.com/matplotlib/matplotlib/blob/d67f08e449a114a08c694c614ac847c565fbb372/lib/matplotlib/spines.py#L558

[4] https://github.com/apache/spark/blob/3b8c0049a5b58f26eb16c2d42070aea31e37a6c3/python/pyspark/pandas/__init__.py#L146

[5] https://github.com/scrapy/scrapy/blob/4300a1d240c7c2c21a4ef0c1c60c3d844493e516/scrapy/item.py#L111

[6] https://github.com/matplotlib/matplotlib/blob/d67f08e449a114a08c694c614ac847c565fbb372/galleries/examples/units/basic_units.py#L141

```
1  def __setattr__(self, attr, value):
2      if not self.__has_singleton():
3          # allow the singleton to be initialized
4          return super().__setattr__(attr, value)
5      elif self is self.__singleton:
6          raise AttributeError(
7              f"attributes of {self!r} are not writeable")
8      else:
9          # duplicate instance - we can end up here from __array_finalize__,
10         # where we set the __class__ attribute
11         return super().__setattr__(attr, value)
```

Listing 16: `__getattr__` code snippet from Numpy, showing the lock pattern.

```
1  def __setattr__(self, key, value):
2      # Disable setting attributes after initialization to prevent false
3      # impression of being able to overwrite a field.
4      # Allowing setting internal states mainly so that _parent can be set
5      # post initialization.
6      if getattr(self, '_frozen', None) and not key.startswith('_'):
7          raise TypeError('Struct.__setattr__() is disabled after __init__()')
8      super().__setattr__(key, value)
```

Listing 17: `__getattr__` code snippet from PyTorch, showing the lock pattern.

```
1  def __setattr__(self, name, value):
2      if not self._mutable:
3          raise ValueError("Mutating `tf.data.Options()` returned by "
4                           "`tf.data.Dataset.options()` has no effect. Use "
5                           "`tf.data.Dataset.with_options(options)` to set or "
6                           "update dataset options.")
7      if hasattr(self, name):
8          object.__setattr__(self, name, value)
9      else:
10         raise AttributeError("Cannot set the property {} on {}.".format(
11             name,
12             type(self).__name__))
```

Listing 18: `__getattr__` code snippet from Tensorflow, showing the lock pattern.

Consider Listings 16 (Numpy[7]), 17 (PyTorch[8]), and 18 (TensorFlow[9]), which show various definitions of `__setattr__` from different projects. Listing 16 checks the value of some attribute `__singleton` (:5) to determine if the `__setattr__` assignment will succeed. Listing 17 does the same with attribute `_frozen` (:6), and similarly Listing 18 checks the value of `_mutable` (:2).

3.4 Additional patterns

Besides the three patterns above, we found some additional code patterns that our proposed dynamic object model cannot support. In one pattern, used in Tensorflow[10] and Keras[11],`__getattr__` is used as a lazy loader of Python modules, where certain modules are only imported as needed through an attribute access. In another pattern, used in Matplotlib[12] and PyTorch[13] magic methods are used to generate new functions, that are defined inside the magic method's body itself. Nagini does not

[7] https://github.com/numpy/numpy/blob/148db6431d4fca8698a8a5da45203cccbbb5eb40/numpy/ma/core.py#L6762

[8] https://github.com/pytorch/pytorch/blob/d59f1da62f6a74d815dede5ac5513b37c49733a3/caffe2/python/schema.py#L551

[9] https://github.com/tensorflow/tensorflow/blob/4df9abf173afc01a63aab72d44551de1cc806ebd/tensorflow/python/data/util/options.py#L52

[10] https://github.com/tensorflow/tensorflow/blob/a371cbe21ce0d02ef7654d3127302850082ce500/tensorflow/python/keras/utils/generic_utils.py#L1179

[11] https://github.com/keras-team/keras/blob/fe85879747d637c54c8fc0a74c56e3b40d8d9be5/keras/src/utils/backend_utils.py#L68

[12] https://github.com/matplotlib/matplotlib/blob/46b39ab5bea6668e56140da568bde60276f6f906/lib/matplotlib/spines.py#L500

[13] https://github.com/pytorch/pytorch/blob/0935b3d794c1b96911084fcc8c7748db5d52b0aa/caffe2/python/functional.py#L30

support the definition of functions inside functions. Lastly, some libraries, for example, PyTorch[14] and TensorFlow[15] show the use of `__getattr__` and `__getattribute__` with clear side-effects, where the call to the magic method generates either some warning of makes a log entry.

Since our model does not support lazy importing Python modules, defining functions within functions, and defining `__getattr__` and `__getattribute__` with side-effects, these code patterns are not supported by the proposed new dynamic object model.

3.5 Determining function call target at runtime

```
1  def __getattribute__(self, name):
2    if name == '__dict__':
3      return super().__getattribute__(name)
4    return self.__dict__['__wrapped'].__getattribute__(name)
```

Listing 19: `__getattribute__` code snippet from Tensorflow, showing an example where function calls are resolved at runtime.

Similar to the wrapper pattern in Section 3.1, Listing 19 (TensorFlow[16]) shows a definition of `__getattribute__`, where the result is retrieved from some instance attribute `__wrapped`, however, unlike before, this is a definition of `__getattribute__`, which handles both attribute accesses as well as function calls, and in Listing 19 (:4) function calls are re-routed to `__wrapped`, instead of being handled by the instance itself with `object.__getattribute__`. This will become an important distinction, and is discussed more in Section 4.4.2.

4 Design

4.1 Overview

4.1.1 Previous (simple) object model in Nagini

The old (simple) object model in Nagini does not support `__dict__`, `__getattr__`, `__setattr__`, and `__getattribute__`. Instance attributes are implemented as Viper fields, meaning they are fixed memory locations, determined at Nagini's compile-time, meaning they are created while the program is encoded into Viper code, as opposed to run-time, when the encoded Viper program is verified. Mypy [18] is a static type checker for Python, which Nagini uses to determine instance attribute types. As a static type checker, mypy imposes certain limitations, for example, instance attributes using the (dot) operator must be defined inside the `__init__` magic method. Nagini handles the type information of instance attributes automatically, and this type information is determined by mypy. Permission assertions `Acc(x.f)`, `MayCreate(x, 'f')`, and `MaySet(x, 'f')`, discussed in more detail in Section 4.3.2, are used to handle the permissions of instance attributes. Additionally, `Acc` requires that the instance attribute exists, `MayCreate` requires that it does not exist. `MaySet` just requires a permission to an instance attribute, and does not specify whether it exists. Attributes are accessed using the (dot) operator, and as previously discussed, `__dict__` access is not supported.

4.1.2 New (complex) Nagini's dynamic object model

The new (complex) Nagini's dynamic object model supports `__dict__`, `__getattr__`, `__setattr__`, and `__getattribute__`, with some limitations. The new model supports all of the same functionality, including the same permission assertions, `Acc`, `MayCreate`, and `MaySet`, as the old model. Both models can be used in the same program, and the default is the simple object model. To use the complex model, a user defined class (or one of its parent classes) must have the `Complex` decorator, discussed in Section 4.3.1.

[14] https://github.com/pytorch/pytorch/blob/0935b3d794c1b96911084fcc8c7748db5d52b0aa/torch/distributed/distributed_c10d.py#L365

[15] https://github.com/tensorflow/tensorflow/blob/a371cbe21ce0d02ef7654d3127302850082ce500/tensorflow/python/keras/engine/training_v1.py#L2873

[16] https://github.com/tensorflow/tensorflow/blob/e9476d0fd503f95377b76b47798bf0829085709b/tensorflow/python/platform/flags.py#L70

The new dynamic object model supports the same (dot) attribute access, with the same type information from mypy, and in addition supports attribute access using `__dict__`, for example, with `self.__dict__['x']`, discussed in Section 4.2.1. Using this method will not retrieve type information from mypy, which means type information has to be handled manually. This is a deliberate design choice, because mypy does not know about some attributes inside `__dict__`, and mypy can also misidentify the types of the attributes that it is aware of. For more information on the interaction between mypy and the new object model refer to 4.2.5. The new object model still uses mypy, meaning that the code still has to pass mypy's static type checking, which can be a problem when use of (dot) operator access is mixed together with `__dict__` access.

The new complex model supports the use of `__getattr__`, however, it must be pure. Similarly, `__getattribute__` is also assumed to be pure. The reason is that by defining these function to be pure they can be used in specifications. For example, if `__getattr__` were permitted to be a method with side-effects, then something as simple as accessing a field using the (dot) operator `expr.attr_name`, when `attr_name` is not in `__dict__`, could not be used in specifications.

Nagini does not support throwing exceptions inside pure functions. This is important because `__getattr__` and `__getattribute__` must be defined as pure functions and because of how Python handles cases where both `__getattr__` and `__getattribute__` are defined, shown in Listing 6, where the exception `AttributeError` plays an important role. Nagini also does not support determining function call targets at runtime. Both of these limitations force the new dynamic object model to place certain restrictions on what can be defined inside `__getattribute__`.

Firstly, all calls to `__getattribute__` with non-instance attribute names must return the default `object.__getattribute__` call, which means, for example, `__getattribute__` called with function name `foo` must return `object.__getattribute__(self, 'foo')`. As just discussed, this is because Nagini does not support determining function call targets at runtime.

Secondly, inside `__getattribute__`, calls to `object.__getattribute__` can only occur as the return statement. As already stated, this is because Nagini does not support throwing exceptions inside pure functions, but if `object.__getattribute__` occurs as a return statement, and `__getattr__` is defined, when a call to `object.__getattribute__` will raise an `AttributeError`, the continued execution of `__getattribute__` will not need to be preempted, as it occurs as the last return statement.

With the new complex model, the use of the (dot) operator can be mixed with the use of `__dict__` access, including for the same attributes, as long as mypy static type checker does not report this as an error, which can happen when, for example, an instance attribute is defined using the `__dict__`, which mypy does not recognize, and then accessed using the (dot) operator, which mypy does recognize.

The encoding does not significantly increase the number of annotations generated, and the number of annotations generated by the new model's encoding, when compared to the encoding of the old model, increases by a constant factor.

4.2 New Python functionality

This Section will look at all of the new Python functionality that is now available in Nagini, namely, `__dict__`, `__getattr__`, `__setattr__`, and `__getattribute__`, as well as the limitations on their use. This Section will also introduce Listings 21, 22, 23, 24, and 25, to serve as motivating examples, showcasing the new functionality.

4.2.1 `__dict__` design

In the simple object model, instance attributes can only be accessed with the member access (dot) operator. In the new complex model they can be accessed with `__dict__` as well, shown in Listing 2. The class has to be set as `Complex`, discussed in Section 4.3.1, and `__dict__` cannot contain names already defined elsewhere, thus overriding their access. For example, `__dict__` cannot contain function, class variable and magic method names, as well as names with special meaning in Python, such as

__dict__ itself, and __slots__. All keys have to be of type `string`. Handling attribute types is left to the user, and Nagini does not do this automatically.

Each field in the __dict__ has its own permission, and the __dict__ instance itself has none, and can be accessed from separate threads. Information about the encoding can be found in Section 4.4.1.

```
1  from nagini_contracts.contracts import *
2
3  @Complex
4  class Dynamic:
5      def __init__(self) -> None:
6          self.__dict__['x'] = 10
7          Ensures(Acc(self.__dict__['x']))
8          Ensures(self.__dict__['x'] == 10)
9
10 def simple_dict() -> None:
11     d = Dynamic()
12     Assert(d.__dict__['x'] == 10)
```

Listing 20: Simple example of __dict__.

Listing 20 shows how attributes can be accessed using __dict__. An attribute `'x'` is created and assigned to in (:6), and later accessed in (:12).

```
1  from nagini_contracts.contracts import *
2
3  class SimpleParent:
4      def __init__(self) -> None:
5          self.x = "15"
6          Ensures(Acc(self.x))
7          Ensures(self.x == "15")
8
9  @Complex
10 class ComplexChild(SimpleParent):
11     def __init__(self) -> None:
12         super().__init__()
13         Ensures(Acc(self.x))
14         Ensures(self.x == "15")
15
16 def simple_parent_test() -> None:
17     p = SimpleParent()
18     c = ComplexChild()
19
20     Assert(p.x == "15")
21     Assert(c.x == "15")
22
23     Assert(c.__dict__['x'] == "15")
24     # Assert(p.__dict__['x'] == "15")   # will not work, SimpleParent is non-complex.
```

Listing 21: Non-complex parent class instance attributes accesed from a complex child class instance using a __dict__.

Listing 21 shows how the new dynamic object model functionality is integrated into the Python type hierarchy using a new Nagini class decorator `Complex`, discussed in Section 4.3.1. `class ComplexChild` is a subclass of `class SimpleParent`. This means that an instance of `ComplexChild` will provide the new dynamic object model functionality, while an instance of `SimpleParent` will not. With `SimpleParent` instance `p` (:17) and `ComplexChild` instance `c` (:18) accessing attributes functions the same (:20 and :21), however, only `c` can use the new functionality, for example, by accessing attributes with `c.__dict__['x']` (:23). Note that this applies even for attributes that are defined in `SimpleParent`, such as `self.x` (:5), so long as this value is passed on to `ComplexChild` (:12, :13, :14).

Pre-existing Nagini permission assertions `Acc`, `MayCreate`, and `MaySet` retain their meaning w.r.t. attributes stored in __dict__, explained further in Section 4.3.2.

4.2.2 `__getattr__` design

Complex classes can define the `__getattr__` magic method. This function is assumed to be pure, as if it had the `@Pure` decorator, meaning it cannot have side-effects. This is done so that (dot) attribute access with a `__getattr__` could be used in specifications. If `__getattr__` were defined as a method, then accessing `self.x` when `x` is not set, meaning `__getattr__` is called, could not be used in specifications. An example of `__getattr__` in action can be seen in Listing 7. When writing pre and posconditions, attributes in `__dict__` need to be accessed using the `__dict__[name]` method, as opposed to using the (dot) operator, as shown in Listing 23 For an encoding of `__getattr__`, refer to Section 4.4.2.

```
1  from nagini_contracts.contracts import *
2
3  @Complex
4  class PrefixStuff:
5      def __init__(self) -> None:
6          Ensures(MayCreate(self, 'x'))
7          Ensures(MayCreate(self, '__x'))
8
9      def __getattr__(self, name: str) -> object:
10         Requires(Implies(len(name) >= 2, name[:2] != "__"))
11         return 10
12
13 def prefix_test() -> None:
14     p = PrefixStuff()
15     Assert(p.x == 10)
16     # Assert(p.__x == 10)   # this will fail. see desc below
```

Listing 22: Different behavior depending on the name of the attribute.

Listing 22, inspired by the prefix pattern in Section 3.2, shows how a class can change the result of trying to access a field not found in `__dict__` depending on the contents of the attribute name. Permission assertion `MayCreate` (:6 and :7) returns permissions to non-existing fields `x` and `__x`, which are then accessed in (:15) and (:16). Since they are non-exsiting, `__getattr__` is called, which checks the contents of the attribute name (:10), in this case, checking that the name does not begin with `__`, and returns a value. As a result, accessing `x` returns `10`, however, trying to access `__x` fails (:16), because of `__getattr__` precondition (:10).

```
1  from nagini_contracts.contracts import *
2
3  @Complex
4  class MyClass:
5      def __init__(self) -> None:
6          self.x = 10
7          Ensures(Acc(self.x))
8          Ensures(self.x == 10)
9
10 @Complex
11 class WrapperMyClass:
12     def __init__(self, wraps: MyClass) -> None:
13         self.c = wraps
14         Ensures(Acc(self.c))
15         Ensures(self.c is wraps)
16         Ensures(MayCreate(self, 'x'))
17
18     def __getattr__(self, name: str) -> object:
19         Requires(Acc(self.__dict__['c']))
20         Requires(Acc(self.__dict__['c'].__dict__[name]))
21         return self.c.__dict__[name]
22
23 def wrapper_test() -> None:
24     c = MyClass()
25     wc = WrapperMyClass(c)
26     Assert(wc.x == 10)
```

Listing 23: `WrapperMyClass`, where instance attributes not found it its own `__dict__` are looked for in the `__dict__` of a completely different class `MyClass`. Note that (:19) cannot be self.c because that

would recurse via precondition.

Listing 22, inspired by the wrapper pattern in Section 3.1, shows how one class (`WrapperMyClass`) can wrap around an instance of a completely different class (`MyClass`) so that trying to access a non-existent attribute in the wrapper class will also try to access an attribute by that same name in the wrapped classed.
Instance `c` of `MyClass` is created (:24) and passed into the initializer for the `WrapperMyClass` instance `wc` on line (:25). `c` contains an attribute `x` (:6), however, `wc` does not (:16). Trying to access an attribute by the name `x` with `wc` (:26) results in a call to `__getattr__` (:18), which tries to access an attribute by that same name `x` in the wrapped instance (:21), and in this example, returns `10`.
Note how `__getattr__` precondition uses `self.__dict__['c']` to access the wrapped instance, instead of something like `self.c`. This is because `self.c` itself would result in a recursive conditional call on `__getattr__` in the precondition, which is not supported in Nagini.

4.2.3 `__setattr__` design

Complex classes can define `__setattr__`, which will then handle instance attribute creation and modification. Given the intended purpose of modifying the `__dict__`, this `__setattr__` is not assumed to be pure, unlike `__getattr__` and `__getattribute__` which are assumed pure. For a simple example of how `__setattr__` can be used in combination with `__dict__`, look at Listing 9.

```
1  from nagini_contracts.contracts import *
2
3  @Complex
4  class LockClass:
5      def __init__(self) -> None:
6          self.__dict__['lock'] = False
7          self.foo = 10
8          self.__dict__['lock'] = True
9          Ensures(Acc(self.foo))
10         Ensures(self.foo == 10)
11         Ensures(Acc(self.__dict__['lock']))
12         Ensures(self.__dict__['lock'] == True)
13
14     def __getattr__(self, item: str) -> object:
15         return None
16
17     def __setattr__(self, name: str, value: int) -> None:
18         Requires(MaySet(self, name))
19         Requires(Acc(self.lock))
20         Requires(self.lock == False)
21
22         self.__dict__[name] = value
23
24         Ensures(Acc(self.lock))
25         Ensures(Acc(self.__dict__[name]))
26         Ensures(self.__dict__[name] is value)
27
28 def lock_test() -> None:
29     l = LockClass()
30     Assert(l.foo == 10)
31
32     l.__dict__['lock'] = False  # without this, lines below will fail
33     l.foo = 31
34     Assert(l.foo == 31)
```

Listing 24: `LockClass().lock` that determines if instance attributes can be modified with the (dot) operator. `__getattr__` needed for mypy.

Listing 24, inspired by the lock pattern in Section 3.3, shows how an instance can prevent attribute creation and modification at run-time, depending on the value of some special attribute. Any instance attribute creation or modification using the (dot) operator for an instance of `LockClass` will call `__setattr__` (:17), which requires that a special attribute `lock` is set to `False` (:20). When `LockClass` instance `l` is created (:29) and initialized (:5), a special attribute `lock` is created by first setting it to `False` (:6), continuing with some other initialization work (:7) and eventually set to `True` (:8),

thus locking the instance from making new attribute modifications with the (dot) operator. Note that creating and modifying attributes using `__dict__[name]` is still allowed, since it does not invoke `__setattr__`.

When an instance of `LockClass` is locked, attributes can still be accessed (:30), however trying to modify an attribute (:33) is not allowed unless if the special `lock` attribute is set back to `False` (:32). More on the encoding and limitations of `__setattr__` can be found in Section 4.4.3.

4.2.4 `__getattribute__` design

Complex classes can also define `__getattribute__`, shown in Listing 8. Same as `__getattr__` this function is assumed to be pure, as if it had the `@Pure` decorator, meaning it cannot have side-effects.

When `__getattribute__` is defined, attributes stored in `__dict__` are commonly accessed using `object.__getattribute__`. Due to limitations discussed in Section 4.4.4, `object.__getattribute__` can only be used as the return statement. Outside of `__getattribute__`, the call to `object.__getattribute__` does not have the same restriction.

Unlike `__getattr__` and `__setattr__`, we do not allow calling `__getattribute__` directly, however, `__getattribute__` is called automatically when using the member access (dot) operator.

As previously discussed in Section 4.2.1, `__getattribute__` can only be used to modify the behavior attribute names that are not already defined elsewhere. What this means is that when `__getattribute__` is called with `name` equal to, for example, `"__dict__"` or some function name, `__getattribute__` must return `object.__getattribute__(self, name)`. This is because we do not allow a function call to be intercepted with `__getattribute__`, and made to return a reference to some different function, or otherwise altered from the default behavior of `object.__getattribute__(self, name)`. A real world example of a `__getattribute__` definition that would come up against this limitation is shown in Listing 19. For an encoding of `__getattribute__` refer to Section 4.4.4.

```
1  from nagini_contracts.contracts import *
2
3  @Complex
4  class GetattributeStuff:
5      def __init__(self) -> None:
6          self.x = 1
7          Ensures(Acc(self.x))
8          Ensures(self.x == 1)
9          Ensures(MayCreate(self, 'y'))
10         Ensures(MayCreate(self, 'z'))
11
12     def foo(self) -> object:
13         Ensures(Result() == 50)
14         return 50
15
16     def __getattr__(self, name: str) -> object:
17         Ensures(Result() == 10)
18         return 10
19
20     def __getattribute__(self, name: str) -> object:
21         Requires(MaySet(self, name))
22
23         # Auto generated:
24         # Ensures(
25         #     Implies(
26         #         name in PSet("foo", "__dict__", ...),
27         #         Result() == object.__getattribute__(self, name))
28         # )
29
30         if name == "z":
31             return 100
32         else:
33             return object.__getattribute__(self, name)
```

```
34
35
36  def getattribute_example() -> None:
37      g = GetattributeStuff()
38
39      Assert(g.x == 1)        # g.__getattribute__('x') == 1     WITH __getattr__
40      Assert(g.y == 10)       # g.__getattribute__('y') == 10    WITH __getattr__
41      Assert(g.z == 100)      # g.__getattribute__('z') == 100   WITH __getattr__
42
43      Assert(g.foo() == 50)          # does not call __getattribute__
44      Assert(g.__dict__['x'] == 1)   # does not call __getattribute__
45
46      Assert(object.__getattribute__(g, "x") == 1)       # same as g.__dict__['x']
47      # Assert(object.__getattribute__(g, "y") == 10)    # will not work. see desc below
48      # Assert(object.__getattribute__(g, "z") == 100)   # will not work. see desc below
```

Listing 25: Interaction between __getattribute__ and __getattr__.

Listing 25 shows how __getattribute__ and __getattr__ can be used together. `class GetattributeStuff` contains a definition for both __getattr__ (:16) and __getattribute__ (:20), and gives access to three attributes - `x` which exists, and `y` and `z` which do not. The class also contains a function `foo()`.

After initializing `class GetattributeStuff` instance `g`, accessing an attribute `g.x` (:39) will result in a call to __getattribute__ with `name == 'x'`, which returns `object.__getattribute__(self, 'x')` (:33). Since field `x` exists (:6), `g.x` returns that value `1`. Trying to access `g.y` (:40) follows a similar pattern, however, the field `y` does not exist (:9), and instead of `object.__getattribute__(self, 'y')` retrieving a value from __dict__, __getattr__ is called, and ultimately `g.y` returns value `10` (:40). An access to `g.z` (:41) does not result in a call to `object.__getattribute__()` (:31), and instead just returns `100` (:41).

Accessing attributes with certain names (:43 and :44) does not result in a call to __getattribute__ at all. The reason this is ok is because Nagini automatically generates a postcondition for __getattribute__, which roughly corresponds to the comment on (:24). Writing a function body for __getattribute__ that violates this autogenerated postcondition will not verify.

Lastly, `object.__getattribute__(self, name)` can also be used outside the body of __getattribute__, as can be seen on (:46), and is equivalent to `self.__dict__[name]`. Because `object.__getattribute__(self, name)` in this example is equivalent to `self.__dict__[name]`, trying to access `y` (:47) or `z` (:48), which are not in the __dict__, will result in errors.

4.2.5 Interaction with mypy

```
1   class Dynamic1:
2       pass
3
4
5   class Dynamic2:
6       def __getattr__(self, name: str) -> str:
7           return "abcd"
8
9
10  d1 = Dynamic1()
11  d1.__dict__["hello"] = "world"
12  print(d1.hello)          # mypy error: "Dynamic1" has no attribute "hello"
13
14  d2 = Dynamic2()
15  d2.__dict__["hello"] = "world"
16  print(d2.hello)          # No mypy error
```

Listing 26: mypy attribute customization limitations.

Mypy is a static type checker for Python. Since Nagini uses mypy to determine attribute types, all of __getattr__, __setattr__ and __getattribute__ need to have their attribute and return types specified. Note that Nagini does not support `Any` type, therefore the built-in __dict__ is assumed to be

of type `Dict[str, object]`. All of the mypy limitations on the use of attribute access customization and the use of `__dict__` remain, for example, in Listing 26 even though both `Dynamic1` and `Dynamic2` function the same, mypy will complain about an access to `d1.hello`. Nagini does not verify code that returns mypy errors.

Refer to the documentation in [18] to work around mypy's limitations on the mixed use of both (dot) member access, together with accessing attributes through `__dict__` directly, and the implications of creating different combinations of `__getattr__`, `__setattr__` and `__getattribute__`. For example, in Listing 24, without a definition of `__getattr__` (:14), trying to access `self.lock` (:19) without having defined `lock` with (dot) access in `__init__` (:5) will result in a mypy error. However, since `__getattr__` is defined (even though it is unused in the example), mypy no longer complains about an access to `self.lock` (:19). Alternatively, Listing 24 could also have replaced calls to `self.lock` with `self.__dict__['lock']`, and then there wouldn't be a need to define `__getattr__`, because mypy does not complain about accesses to items in `__dict__`.

Note that Nagini uses mypy in order to determine attribute types, and will generate Viper code accordingly, however, since mypy can make mistakes on what the actual type of an attribute is, in order to remain sound, Nagini will also generate code that checks the actual type at runtime.

```
1  class A:
2      def foo(self) -> str:
3          return "foo"
4
5  class B:
6      def foo(self) -> int:
7          return 100
8
9  class MyClass:
10   def _init_(self) -> None:
11     self.f = A()
12
13 c = MyClass()
14 setattr(c, "f", B())
15 print(c.f.foo() + "bar")  # TypeError: ... for +: 'int' and 'str'
```

Listing 27: mypy fails to detect a `TypeError`.

Consider Listing 27, that produces a `TypeError` (:15). Unfortunately, mypy will not produce an error, and the `setattr()` assignment to `f` (:14) is ignored.

```
1  class A:
2      def foo(self) -> str:
3          return "foo"
4
5  class B:
6      def foo(self) -> int:
7          return 100
8
9  class MyClass:
10   def _init_(self) -> None:
11     self.f = A()
12
13 c = MyClass()
14 setattr(c, "f", B())
15 print(c.f.foo() + 100)  # prints 200
```

Listing 28: mypy incorrectly detects a `TypeError`.

In Listing 28 the situation is reversed, and there is no `TypeError`, however, mypy will signal `error: Unsupported operand types for + ("str" and "int")`.
In both cases mypy misidentifies the type returned by `c.f.foo()`. As previously discussed, Nagini creates preconditions on method calls that check the runtime type of the caller. This means that even if mypy misidentifies a type of an attribute, which causes Nagini to generate a call to the wrong

function, this call would still contain a runtime caller type check which would cause the verification to fail, and Nagini verification would remain sound.

```
1 function PrefixStuff___getattr__(self: Ref, name: Ref): Ref
2   requires issubtype(typeof(self), PrefixStuff())
3   ...
```

Listing 29: Nagini soundess check.

To give a concrete example, Listing 29 shows what Nagini generates for `__getattr__` definition in Listing 22 (:9). The runtime type is confirmed in (:2).

4.3 Specification Constructs

This section looks at two new Nagini contracts - `Complex` class decorator, and permission assertions `Acc`, `MayCreate` and `MaySet` when used on fields stored in `__dict__`.

4.3.1 Complex class decorator

All classes in Nagini are either simple (non-complex) or complex. Simple classes use the old object model, meaning, the use of `__dict__`, and magic methods `__getattr__`, `__setattr__` and `__getattribute__` is not supported, as opposed to complex classes, where their use is supported. All classes are simple, unless if they or any of their parent classes have the class decorator `Complex`. Refer to Listing 21 for what that looks like in action.

4.3.2 `__dict__` Permission assertions

Each key in `__dict__` has its own permission, and these permissions can be expressed with the same permission assertions as for simple classes, namely `Acc`, `MayCreate` and `MaySet`. `Acc` gives permission to a field , and this field exists in `__dict__`, `MayCreate` gives permission to a field and this field does not exist, and `MaySet` just gives the permission to the field, but says nothing on whether the field exists. Having either `Acc` or `MayCreate` field permission implies having `MaySet` field permission.

With `MayCreate` and `MaySet` the attribute name is provided as an expression. Refer to Listing 25 and [19] on how to use both `MayCreate` and `MaySet`. However, with `Acc` the attribute name can be provided either using the (dot) operator, as is the case for simple classes, for example `Acc(self.c)`, or by using the functionality provided in complex classes, for example `Acc(self.__dict__['c'])`. Refer to Listing 23 for an example.

Note that these permission assertions, except for type information, function the same for simple classes as for complex classes, meaning that taking a simple class and making it complex does not change the semantics of attribute permissions, but would additionally offer more functionality that comes with complex classes. As previously discussed, type information is not retained when attributes are accessed using `__dict__`, and this type informations has to be handled manually. When using the (dot) operator, this type information is provided by mypy automatically.

As previously discussed in 4.2.1, `__dict__` keys have to be of type `str`, and permissions are only available for names not otherwise defined, for example, permissions to function names in `__dict__` are not available. Section 4.4.1 describes how this is encoded and enforced.

4.4 Encoding

4.4.1 `__dict__` implementation

`__dict__` is encoded using Keydict. For the definition of Keydict refer to 7.1.

Figure 1 shows how attribute access, deletion and modification is encoded using calls to Keydict functions and methods. In the first line of Figure 1 a Python attribute statement is encoded into Viper with a call to `keydict___getitem__`, on the second line a Python attribute deletion is encoded with a call to `keydict___delitem__`, and on the third line a Python attribute modification or creation

$$
\begin{array}{lcl}
enc(\textbf{expr.attr}) & = & keydict\_\_getitem\_\_(\textbf{expr}, enc(\textbf{attr})) \\
enc(del\ \textbf{expr.attr}) & = & keydict\_\_delitem\_\_(\textbf{expr}, enc(\textbf{attr})) \\
enc(\textbf{expr1.attr} = \textbf{expr2}) & = & keydict\_\_setitem\_\_(\textbf{expr1}, enc(\textbf{attr}), \textbf{expr2})
\end{array}
$$

Figure 1: Encoding of (dot) operator attribute access and modification, from Python to Viper. `enc` takes a Python expression or statement and transforms it into Viper code.

is encoded with a call to `keydict__setitem__`. For more information on how these functions and methods themselves are encoded, refer to Section 7.1.

```
1 keydict___init__(expr)
2 exhale acc(keydict___item__(expr, str___create__(8, Seq(95, 95, 105, 110, 105, 116,
      95, 95))).keydict_val, wildcard)
3   ...
```

Listing 30: `__dict__` initialization inside `__init__`. (:2) shows how a permission to key `"__init__"` is exhaled.

Listing 30 shows how `__dict__` is initialized inside of `__init__`. `keydict___init__` (:1) creates a `__dict__` instance and assigns it to `expr`, and returns permissions to all keys, and sets all values to `None`. Right after that permissions to already defined names are exhaled, which in this example is `str___create__(8, Seq(95, 95, 105, 110, 105, 116, 95, 95))` (:2), which is the string encoding for the string `__init__`. As previously described, this includes all function names, class attribute names, special names such as `__dict__` itself, and others.

$$
\begin{array}{lcl}
enc(Acc(\textbf{expr.attr})) & = & acc(keydict\_\_item\_\_(\textbf{expr}, enc(\textbf{attr}).keydict\_val, write) \\
 & & \&\& \ keydict\_\_contains\_\_(\textbf{expr}, enc(\textbf{attr})) \\
enc(Acc(\textbf{expr}.\_\_dict\_\_[\textbf{attr\_expr}])) & = & acc(keydict\_\_item\_\_(\textbf{expr}, enc(\textbf{attr\_expr})).keydict\_val, write) \\
 & & \&\& \ keydict\_\_contains\_\_(\textbf{expr}, enc(\textbf{attr\_expr})) \\
enc(MayCreate(\textbf{expr}, \textbf{attr\_expr})) & = & acc(keydict\_\_item\_\_(\textbf{expr}, enc(\textbf{attr\_expr})).keydict\_val, write) \\
 & & \&\& \ !keydict\_\_contains\_\_(\textbf{expr}, enc(\textbf{attr\_expr})) \\
enc(MaySet(\textbf{expr}, \textbf{attr\_expr})) & & acc(keydict\_\_item\_\_(\textbf{expr}, enc(\textbf{attr\_expr})).keydict\_val, write)
\end{array}
$$

Figure 2: Encoding of `__dict__` permission assertions. `enc` takes a Python expression or statement and transforms it into Viper code.

Figure 2 shows how permission assertions are encoded. Note that having either `Acc` or `MayCreate` permission implies having `MaySet`. The first line of Figure 2 with a Nagini permission assertion `Acc` to a field accessed using the (dot) operator is encoded to the `keydict_val` field of the reference returned by `keydict___item__`, as well as an assertion that `keydict___contains__` is true, meaning that the field exists. The second line to a Nagini permission assertion `Acc` to a field accessed using the `___dict__` method is encoded to the same thing as above, which is the field access using the (dot) operator, and they are equivalent on the encoding side. The third line to a Nagini permission assertion `MayCreate` is encoded to the `keydict_val` field of the reference returned by `keydict___item__`, as well as an assertion that `keydict___contains__` is false, meaning that the field does not exist. The fourth line to a Nagini permission assertion `MaySet` is encoded to the `keydict_val` field of the reference returned by `keydict___item__`, and makes no assertions on whether the field exists. For more information on how these keydict functions and methods themselves are encoded, refer to Section 7.1.

4.4.2 `__getattr__` implementation

$$\begin{aligned} enc(\textbf{expr.attr}) \quad = \quad & keydict\_\_\_contains\_\_(\textbf{expr}, enc(\textbf{attr})) \ ? \\ & keydict\_\_\_getitem\_\_(\textbf{expr}, enc(\textbf{attr})) \ : \\ & cls(\textbf{expr})\_\_\_getattr\_\_(\textbf{expr}, enc(\textbf{attr})) \end{aligned}$$

Figure 3: Encoding of (dot) operator attribute access when `__getattr__` is defined. `enc` takes a Python expression or statement and transforms it into Viper code. `cls(expr)` refers to the static class name of `expr`.

```
1 keydict___contains__(expr, str___create__(1, Seq(120))) ?
2       keydict___getitem__(expr, str___create__(1, Seq(120))) :
3       MyClass___getattr__(expr, str___create__(1, Seq(120)))
```

Listing 31: Code example of (dot) operator attribute access when `__getattr__` is defined, equivalent to `expr.x`, and `expr` is of type `MyClass`. Compare with Figure 3.

When `__getattr__` is defined, accessing an attribute using the (dot) operator results in a conditional expression, as show in Listing 31, modeling the `__getattr__` functionality shown in Listing 6. If the attribute name is found in `self.__dict__` (:1) that attribute is returned (:2), otherwise `__getattr__` (:3) is called.

It is important to be mindful of what this means for `__getattr__` preconditions. If they are written using the (dot) operator, for example `self.x`, and the class of `self` has a `__getattr__` defined, then this postcondition would contain a recursive call to `__getattr__`. This is why Listing 23 uses the `self.__dict__['c']` method of accessing attribute `c`, instead of just using `self.c`.

4.4.3 `__setattr__` implementation

$$enc(\textbf{expr1.attr} = \textbf{expr2}) \quad = \quad cls(\textbf{expr1})\_\_\_setattr\_\_(\textbf{expr1}, enc(\textbf{attr})), \textbf{expr2})$$

Figure 4: Encoding of assignment when `__setattr__` is defined. `enc` takes a Python expression or statement and transforms it into Viper code. `cls(expr)` refers to the static class name of `expr`.

```
1 MyClass___setattr__(expr1, str___create__(3, Seq(102, 111, 111)), expr2)
```

Listing 32: Code example of (dot) operator attribute modification with a defined `__setattr__`, equivalent to `expr1.foo = expr2`, where `expr1` is of type `MyClass`. Compare with Figure 4.

Without a `__setattr__` definition, attribute modification using the (dot) operator is handled by `keydict___setitem__` as shown in Figure 1. When `__setattr__` is defined, it replaces the Keydict call, as shown in Figure 4 and Listing 32.

It is important to note what this means for the body of `__setattr__`. Using the (dot) operator to modify a value inside `__setattr__` can result in a recursive call on `__setattr__` itself. If this is not the desired behavior, instead of using the (dot) operator, `self.__dict__` can be accessed directly with `self.__dict__[name]`, which is why in Listing 24 `__setattr__` modifies an attribute using `self.__dict__[name] = value` (:22).

4.4.4 `__getattribute__` implementation

$$enc(\textbf{expr.attr}) \quad = \quad cls(\textbf{expr})\_\_getattribute\_\_(\textbf{expr}, enc(\textbf{attr}))$$

Figure 5: Encoding of (dot) operator attribute access when `__getattribute__` is defined. `enc` takes a Python expression or statement and transforms it into Viper code. `cls(expr)` refers to the static class name of `expr`.

Just like `__getattr__` in Section 4.4.2, `__getattribute__` implementations are assumed to be pure in order to allow their use in specifications.

In Python, `__getattribute__` is called on every attribute access, including functions and built-ins like `__dict__`. This behavior is not supported in Nagini, so in order to allow `__getattribute__` to be used for instance attributes Nagini generates a postcondition in `__getattribute__` which corresponds to Listing 25 (:24).

```
1 function GetattributeStuff___getattribute__(self_0: Ref, name_0: Ref): Ref
2   ...
3   ensures (name_0 in
4     Set(str___create__(3, Seq(102, 111, 111)), ...)) ==>
5     result ==
6     (keydict___contains__(self_0, name_0) ?
7       keydict___getitem__(self_0, name_0) :
8       GetattributeStuff___getattr__(self_0, name_0))
9   ...
```

Listing 33: Autogenerated postcondition for `__getattribute__`.

Listing 33 shows a concrete example of what that looks like in Viper. Every name that is not allowed in `self.__dict__` (:3 and :4) must exhibit the normal (dot) access behavior (:6, :7, and :8). Note that these names are not in `self.__dict__`, and the permissions to those names do not exist, and (:6, :7, and :8) simply represents the default (dot) access as if `__getattribute__` had not been defined.

With the autogenerated postcondition in place, `__getattribute__` is only called for instance attribute names, and for all others, for example function names, the function itself can be called, since that is the result of `__getattribute__` anyway. In other words, only `__getattribute__` definitions that do not change the behavior of function name, and other special name accesses can be verified.

```
1 GetattributeStuff___getattribute__(self_1, str___create__(1, Seq(120)))
```

Listing 34: (dot) operator attribute access when `__getattribute__` is defined. Compare with 5.

Listing 34 shows what Listing 31 is replaced with when `__getattribute__` is defined. In order to still use Figure 1 functionality, `self.x` can be accessed with `self.__dict__['x']` (since `__dict__` is a special name and does not result in a call to `__getattribute__`). In order write `__getattribute__` that has the same functionality as Listing 31 for `self.x`, the attribute name `x` inside `__getattribute__` needs to result in a call to `object.__getattribute__`.

With the behavior of `__getattribute__` in Listing 6 in mind, consider what would happen in Listing 25 if `object.__getattribute__` inside `__getattribute__` (:33) were placed somewhere inside the body of the `__getattribute__` and not right at the `return` statement, and this call to `object.__getattribute__` raised an `AttributeError`. In Python this `AttributeError` would be caught outside of `__getattribute__`, and if `__getattr__` is defined, call it. However, with Section 4.4.1, and 4.4.2 definitions, that is not what Nagini would generate, and `__getattribute__` execution would not be preempted by the `AttributeError`. This is why inside `__getattribute__` calls to `object.__getattribute__` are only allowed as the return statement, as this matches the pre-emptive behavior in Python. This is a design decision, and other approaches are discussed in Section 5.

```
1  def __getattr__(self, name: str) -> object:
2      return 10
3
4  def __getattribute__(self, name: str) -> object:
5      ...
6      if name == "z":
7          a: int  = object.__getattribute__(self, name)
8          a += 1
9          return a
10     else:
11         ...
```

Listing 35: Incorrect definition of `__getattribute__`, that would fail to verify.

Listing 35 shows an incorrect `__getattribute__` definition. Consider what would happen with the proposed encoding, when `object.__getattribute__` is allowed to be called somewhere besides the return statement. Nagini would generate a conditional expression on `name` (:7) and if it is not in `__dict__` then it would call `__getattr__` (:1), increment the result by 1 (:8) and return (:9). As explained before, this is not how `__getattribute__` works, hence the need for `object.__getattribute__` to only be allowed as the return statement.

5 Different design choices

When designing the new dynamic object model we made certain design choices. This Section will go over some of those choices, and contrast them to some obvious alternatives.

5.1 Calls to `object.__getattribute__` inside `__getattribute__`

Section 4.4.4 goes over the reasons why `object.__getattribute__` has to be handled with care given the proposed model. One alternative solution is to not allow both `__getattr__` and `__getattribute__` to be defined at the same time. Since the proposed solution allows for more functionality with minimal effort, this alternative was not chosen.

Another alternative approach is to create a more sophisticated mechanism for exception handling, that would more closely model the behavior of Listing 6. This approach is not optimal, because it would require considerable effort, and would likely fit better with the rest of the codebase when implemented together with some unified, automatic way of handling exceptions inside pure functions, which currently is not supported in Nagini.

5.2 Exhaling permissions to function names

Nagini does not allow `__getattribute__` to handle function calls, as discussed in Section 4.4.4. In order to sidestep the issue of `__getattribute__` changing the behavior of function calls, a postcondition is autogenerated inside `__getattribute__`. This postcondition allows to ignore `__getattribute__` for function calls, and exhale permissions to the names of those functions inside `__dict__`.

An alternative approach could be to allow to have permissions to function names in `__dict__`, and for every function call to require the permission to that function's name in `__dict__`, and for the value to remain unchanged. This would allow to temporarily place some value inside `__dict__` under the name of a function, but then restore it to the default value before a function is called. This approach would require to juggle around permissions to function names, and in return enjoy the ability to store items in `__dict__` under those function names. We do not consider this trade-off to be worthwhile.

5.3 Defining `Acc`, `MayCreate`, and `MaySet`

Section 4.3.2 goes over the functionality of `Acc`, `MayCreate`, and `MaySet` permission assertions, and how they are reused for use with complex classes with `__dict__`, effectively supporting the same functionality so that a simple class made complex (without changing anything else) would continue to function the same.

An alternative approach could be to redefine `Acc`, `MayCreate`, and `MaySet` in some way, or to introduce new permission assertions. For example, redefining `Acc`, which takes a single argument and does not allow the attribute name to be passed in as an expression, and to redefine it to something that takes two arguments, to better fit the pattern followed by `MayCreate` and `MaySet`. However, the proposed solution seems to fit the target use-cases from Listings 21, 22, 23, 24, and 25 well enough. In particular, the issue with `Acc` is addressed by allowing `Acc(self.x)` to be replaced with `Acc(self.__dict__['x'])` which still only takes a single argument, but allows the attribute name to be passed in as an expression.

5.4 Assuming `__getattr__` and `__getattribute__` to be pure

As already discussed, `__getattr__` and `__getattribute__` are assumed to be pure, in order to allow their functionality to be used in specifications.

An obvious alternative is to allow to select if and when `__getattr__` and `__getattribute__` are pure. There are two issues with this. Firstly, it is unlikely that `__getattr__` or `__getattribute__` that is defined as a method could be useful in the context of verification, since that would mean that attributes can no longer be used in specifications. Secondly, this would likely require considerable effort to implement, and as mentioned before, would likely be of little use. Another approach is to allow to define two versions of `__getattr__` and `__getattribute__` each. One version that is a pure function, and another version that is not. In specifications, and other contexts that only allow pure function calls, the pure version would be used, otherwise attribute accesses would result in a non-pure function call. This is a promising approach, however, we decided that it is outside the scope of this thesis.

5.5 `Complex` class decorator

As discussed in Section 4.3.1, all classes are non-complex by default, and only those with the `Complex` class decorator, or those that are subclasses of such classes, are considered complex. There are a couple of different approaches that could have been taken instead.

One alternative solution is to make all classes complex, removing the need for a `Complex` class decorator. However, this would result in slightly more complicated verification files, increasing verification runtime and likely complicate the process of debugging verification errors. Another alternative approach is to automatically detect when a class should be complex, when that class uses some functionality only available to complex classes. This solution is also not optimal, since accidentally using complex functionality at some base class would turn all instances of this class and its subclasses complex, with the aforementioned downsides.

6 Implementation

We have implemented the new dynamic object model presented in Section 4 in Nagini, however, some functionality is still missing. Support for all of `__dict__`, `__getattr__`, `__setattr__`, and `__getattribute__` is available, as shown in Listings 21, 22, 23, 24, and 25. Attributes can still be used in the same ways that were supported in the old object model, for example, multiple assignment (unpacking) is still available. Three examples of functionality that is still missing and that could relatively easily be implemented, without requiring significant restructuring, are `getattr()` and `setattr()` built-ins discussed in Section 2.2, field deletion using `del`, and a general purpose implementation of the Alternative Dictionary Encoding (Keydict), discussed in Section 7.1, besides `__dict__`.

7 Additional Contributions

This Section discusses two additional contributions to Nagini that enabled the implementation of the motivating examples shown in Listings 21, 22, 23, 24, and 25.

7.1 Alternative Dictionary Encoding - Keydict

Currently, there is no support in Nagini for a map data structure that could separate permissions on a per key basis, which is something that is required for an implementation of `__dict__`, in order to allow separate threads to access different subsets of instance attributes. This need for separate permissions for each key in `__dict__` motivates the creation of `Keydict`.

```
1  field keydict_val: Option[Ref]
2
3  method keydict___init__(self: Ref)
4      ensures forall key: Ref :: {keydict___item__(self, key)} acc(keydict___item__(self
       , key).keydict_val) && !keydict___contains__(self, key)
5
6  function keydict___item__(self: Ref, key: Ref): Ref
7      ensures keydict___item__inv(self, result) == key
8
9  function keydict___item__inv(self: Ref, val_ref: Ref): Ref
10
11 function keydict___contains__(self: Ref, key: Ref) : Bool
12     requires acc(keydict___item__(self, key).keydict_val, wildcard)
13 {
14   keydict___item__(self, key).keydict_val.isSome
15 }
16
17 function keydict___getitem__(self: Ref, key: Ref) : Ref
18     requires acc(keydict___item__(self, key).keydict_val, wildcard)
19     requires keydict___contains__(self, key)
20 {
21   keydict___item__(self, key).keydict_val.value
22 }
23
24 method keydict___setitem__(self: Ref, key: Ref, item: Ref)
25     requires acc(keydict___item__(self, key).keydict_val)
26     ensures acc(keydict___item__(self, key).keydict_val)
27     ensures keydict___item__(self, key).keydict_val.value == item
28     ensures keydict___contains__(self, key)
29     ensures keydict___getitem__(self, key) == item
30 {
31     keydict___item__(self, key).keydict_val := Some(item)
32 }
```

Listing 36: Keydict definition.

Keydict is a map, where each key has a separate permission. It is not based on Viper's built-it map type `Map[T, V]`, which stores all map information inside a single type, but instead relies on a function (:6) to provide a map functionality. There is a permission for every possible key, and all keys map to an `Option Type` value. Creating a map gives a permission to every key, and sets every field value to `None()`. Keys are considered present in the map when their corresponding values are of type `Some(Ref)`.

In Listing 47, this corresponds to the `Option Type` field on (:1) and initialization that uses a `forall` quantifier, with a trigger on item access, and creates permissions to all of the keys, and sets all values to `None()` on (:4). New value creation and modification is done on (:24), and lastly, values are accessed on (:17). A value can only be accessed if it is present in the dictionary, meaning that it is not `None()`, which is checked on (:14). To delete an existing value in the dictionary, it is set back to `None()`. More helper methods, such as Listing all the keys or values in the map, are possible beyond the ones in Listing 47, and are included in B.2.

Unlike with a single filed of Viper's built-in type `Map[T, V]`, which only has a permission to that one field, with a Keydict, since each key has its own permission, it is possible to use one subset of keys in one thread in Nagini, and a different subset of keys in another without any conflicts. This property makes Keydict a good candidate for a `__dict__` implementation, discussed in Section 4.2.1.

7.2 Improved String Encoding - Strings as sequences of integers

Previously, Nagini encoded strings as objects consisting of two values - the length of the string, and the hash of the string as an `Int` literal. This worked well for checking equality, however, it did not allow for more complicated specifications, for example, checking a sub-sequence of a string, or reasoning about the contents of a string built up in parts.

The fact that `__dict__`, `__getattr__`, `__setattr__`, and `__getattribute__` operate on strings, and the previous string encoding offered limited functionality on their use in Viper, motivates a new string encoding using sequences of integers.

```
1 x = "abcd"
2 y = x + "efg"
3 Assert(y == "abcdefg")       # assertion might not hold
4 Assert(y[2:6] == "cdef")     # not supported with previous string encoding
```

Listing 37: String example.

```
1 x := str___create__(4, 1684234849)
2 y := str___add__(x, str___create__(3, 6776421))
3 assert str___eq__(y, str___create__(7, 29104508263162465))
```

Listing 38: Previous Nagini string encoding of Listing 37

Listing 38 shows how the example in Listing 37 is encoded using the previous Nagini string encoding. Building up a string in parts and verifying the contents is not supported, meaning that Listing 37 (:3) cannot be verified, and neither can taking a slice be verified (:4), because the contents of the string have already been hashed, as seen in Listing 38 (:1).

```
1  function str___len__(self: Ref): Int
2      decreases _
3    ensures result >= 0
4
5  function str___val__(self: Ref): Seq[Int]
6      decreases _
7
8  function str___create__(len: Int, value: Seq[Int]) : Ref
9      decreases _
10   ensures str___len__(result) == len
11   ensures str___val__(result) == value
12   ensures typeof(result) == str()
13
14 function str___eq__(self: Ref, other: Ref): Bool
15     decreases _
16     requires issubtype(typeof(self), str())
17     ensures (str___val__(self) == str___val__(other)) == result
18     ensures result ==> (str___len__(self) == str___len__(other))
19     ensures result == object___eq__(self, other)
20
21 function str___add__(self: Ref, other: Ref): Ref
22     decreases _
23     requires issubtype(typeof(self), str())
24     ensures str___len__(result) == str___len__(self) + str___len__(other)
25     ensures typeof(result) == str()
26     ensures result == str___create__(str___len__(self) + str___len__(other),
       str___val__(self) ++ str___val__(other))
```

Listing 39: New encoding strings as `Seq[Int]`.

Listing 39 proposes a new encoding, where strings are encoded using Viper's `Seq[Int]` (:5), meaning that strings are encoded as objects associated with a sequence of integers, and each string character is encoded as an `Int` literal of the character's ascii code. This allows to use Viper's built-in `Seq[T]` functionality, such as sequence concatenation (:21), sub-sequence operations and others. Note that even though all the information is now present on the Viper level, Viper's reasoning about strings may not be complete, for example, when building up strings with concatenation it might be necessary to explicitly state that `"q" + "we" == "qw" + "e"` for the purposes of checking for equality (:14).

```
x := str___create__(4, Seq(97, 98, 99, 100))
y := str___add__(x, str___create__(3, Seq(101, 102, 103)))
assert str___eq__(y, str___create__(7, Seq(97, 98, 99, 100, 101, 102, 103)))
assert str___eq__(str___getitem_slice__(y, slice___create__(2, 6)), str___create__(4,
    Seq(99, 100, 101, 102)))
```

Listing 40: New Nagini string encoding of Listing 37.

Listing 40 shows how the example in Listing 37 is encoded using the new Nagini string encoding. Building up a string in parts and verifying the contents is supported (:3) and it is possible to take a slice of a string as well (:4).

8 Evaluation

As already discussed in Section 6, there is an implementation of the proposed new dynamic object model. This implementation can verify the motivating examples and, with a few exceptions, reports the same results as the old model when run on pre-existing Nagini tests, when the user defined classes in those tests are defined as complex.

In this Section we evaluate the implementation on some Nagini tests, and find that using the complex object model increases the number of generated lines of Viper code by about 10%. The runtime difference is often more significant, with some examples taking multiple times longer to verify with the new dynamic object model than with the old simple object model, without using any of the new functionality. The tables in this section also include the motivating examples from Listings 21, 22, 23, 24, and 25, as a reference.

The motivating examples use complex functionality, so there are no simple model runs for those examples to compare with. The pre-existing Nagini tests only use simple functionality, and for those the "Complex case" is measured by giving each user defined class the `Complex` decorator.

File	Simple #lines	Complex #lines	#lines factor increase
simple_parent		1164	
getattribute_and_getattr		1236	
lock		1240	
prefix		1159	
wrapper		1235	
cav_example	1630	1726	1.06
iap_bst	1878	2071	1.10
keon_knapsack	1822	1945	1.07
parkinson_recell	1268	1356	1.07
test_student_enroll_preds	1781	1901	1.07
test_constructor	1164	1287	1.11
test_dynamic_field_creation	1856	1933	1.04
test_havoced_types	1768	1900	1.07
test_isinstance	981	1085	1.11
test_operators	1783	1893	1.06
test_predicate	1254	1370	1.09

Table 1: "Simple #lines" shows the number of Viper lines of code generated when the test file uses the old object model, and "Complex #lines" - the new dynamic object model. "#lines factor increase" shows by what factor the number of lines generated has increased. The first five examples are of Listings 21, 22, 23, 24, and 25, which showcase complex object model functionality, so there is no "Simple #lines" to compare them with. The remaining examples are taken from Nagini test files. For those "Complex runtime" was created by giving each user defined class the `Complex` decorator.

Table 1 shows how the number of lines of generated Viper code changes when using the old object model versus the new dynamic object model, with most test cases showing an increase of about 10% for the new model. This is consistent with our observation that the new encoding should not significantly increase the number of generated Viper lines of code when compared to the encoding of the old object model.

File	Simple runtime (Silicon)	Complex runtime (Silicon)	Runtime factor increase (Silicon)	Simple runtime (Carbon)	Complex runtime (Carbon)	Runtime factor increase (Carbon)
simple_parent		10.53			9.46	
getattribute_and_getattr		11.33			11.43	
lock		11.66			11.39	
prefix		10.09			9.58	
wrapper		10.64			10.01	
cav_example	16.30	21.22	1.30	19.46	30.47	1.57
iap_bst	15.25	90.12	5.91	20.78	85.30	4.10
keon_knapsack	17.63	38.28	2.17	27.46	41.57	1.51
parkinson_recell	9.54	11.21	1.17	9.35	10.67	1.14
test_student_enroll_preds	12.79	14.61	1.14	13.04	14.74	1.13
test_constructor	9.68	13.25	1.37	9.42	10.59	1.12
test_dynamic_field_creation	10.12	11.92	1.18	10.72	11.65	1.09
test_havoced_types	12.18	13.97	1.15	12.53	12.96	1.03
test_isinstance	9.01	9.90	1.10	8.46	9.10	1.08
test_operators	11.08	12.88	1.16	10.89	11.71	1.07
test_predicate	9.60	11.91	1.24	9.64	10.72	1.11

Table 2: "Simple runtime" measured using the old object model functionality, using both Silicon and Carbon verifiers, which is compared against "Complex runtime" where user defined classes use the new dynamic object model. First five examples are of Listings 21, 22, 23, 24, and 25, which showcase complex object model functionality, so there is no "Simple runtime" to compare them with. The remaining examples are taken from Nagini test files. For those "Complex runtime" was created by giving each user defined class the `Complex` decorator.

As shown in Table 2, the runtime difference between simple and complex models is not as consistent as it is for the difference between the generated lines of Viper code, shown in Table 1. Some runtimes, for example `test_isinstance`, only increase by about 10%, however, other tests see a much more significant runtime increase, for example, `iap_bst` taking as many as four or five times as long to verify. There is no major difference between the runtimes of `Silicon` versus `Carbon` verifier backends, and the runtime increase between the simple and complex model's is smaller when using the `Carbon` backend.

Since both the number of lines of generated Viper code as well as the total runtime increases when using the complex object model, these performance tests justify the design choice made in Section 5.5, keeping classes simple by default, and only using complex classes with the new dynamic object model when it is necessary.

9 Conclusion

The expressiveness offered by Python's `__dict__`, and magic methods `__getattribute__`, `__getattr__` and `__setattr__` gives Python programs a lot of ability to customize attribute access. This expressiveness and flexibility also makes it challenging to verify meaningful properties about those Python programs that take advantage of them. With the combination of the Alternative Dictionary Encoding (Keydict), as well as some restrictions on the functionality of the mentioned magic methods, we propose an encoding of Pythons dynamic object model that is capable of verifying various magic method usage patterns, inspired by real use-cases. We have explained the reasons behind some of the design choices made, and identified other possible approaches. Our proposed design allows to switch between the old static object model and the proposed new dynamic object model, and the runtime difference between the two suggests that the new dynamic object model should be used sparingly.

Future work could include setting up a comprehensive test suite to identify and fix any remaining issues with the implementation of the new encoding, as well as implementing any functionality that has

not been implemented yet, for example `getattr` and `setattr` built-ins. Different approaches to some of our design choices could be explored, in particular defining two versions of each of `__getattribute__` and `__getattr__`, one as a pure function, and the other as a method with side-effects. This might offer more flexibility without significantly complicating the design. Lastly, our proposed encoding places significant restrictions on the use of `__getattribute__`, in particular w.r.t. function calls. Perhaps some of those restriction could be relaxed.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[2] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[3] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), may 2012.

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

[5] 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.

[6] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.

[7] Malte H. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Doctoral thesis, ETH Zurich, Zürich, 2016.

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

[9] Tom Christie. mkdocs. https://github.com/mkdocs/mkdocs, 2014.

[10] Jonghoon Kwon, Juan A. García-Pardo, Markus Legner, François Wirz, Matthias Frei, David Hausheer, and Adrian Perrig. SCIONLab: A next-generation Internet testbed. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2020. Best Paper Award.

[11] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[12] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, nov 2016.

[13] Martin Olveyra Gabriel García Michael Cetrulo Artem Bogomyagkov Damian Canabal Andres Moreira Ismael Carnales Matías Aguirre German Hoffmann Anibal Pacheco Bruno Deferrari Shane Evans Ezequiel Rivero Patrick Mezard Rolando Espinoza Ping Yin Lucian Ursu Shuaib Khan Didier Deshommes Vikas Dhiman Jochen Maes Darian Moody Jordi Lonch Zuhao Wan Steven Almeroth Tom Mortimer-Jones Chris Tilden Alexandr N Zamaraev Emanuel Schorsch Michal Danilak Natan Lao Hasnain Lakhani Pedro Faustino Alex Cepoi Ilya Baryshev Libor Nenadál Jae-Myoung Yu Vladislav Poluhin Marc Abramowitz Valentin-Costel Hăloiu Jason Yeo Simon Ratne Julien Duponchelle Jochen Maes Vikas Dhiman Juan Picca Nicolás Ramírez Pablo Hoffman, Daniel Graña. Scrapy. https://github.com/scrapy/scrapy, 2008.

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[16] Francois Chollet et al. Keras, 2015.

[17] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[18] Jukka Lehtosalo and contributors. mypy documentation. https://mypy.readthedocs.io/en/stable/, 2012. Last accessed: 05.05.2024.

[19] Nagini's wiki page. https://github.com/marcoeilers/nagini/wiki. Last accessed: 19.04.2024.

Appendices

A Motivating examples

A.1 Bad Prefix

```
from nagini_contracts.contracts import *

@Complex
class PrefixStuff:
    def __init__(self) -> None:
        Ensures(MayCreate(self, 'x'))
        Ensures(MayCreate(self, '__x'))

    def __getattr__(self, name: str) -> object:
        Requires(Implies(len(name) >= 2, name[:2] != "__"))
        return 10

def prefix_test() -> None:
    p = PrefixStuff()
    Assert(p.x == 10)
    # Assert(p.__x == 10)   # this will fail
```

Listing 41: Different behavior depending on the name of the attribute

A.2 Wrapper Class

```
from nagini_contracts.contracts import *

@Complex
class MyClass:
    def __init__(self) -> None:
        self.x = 10
        Ensures(Acc(self.x))
        Ensures(self.x == 10)

@Complex
class WrapperMyClass:
    def __init__(self, wraps: MyClass) -> None:
        self.c = wraps
        Ensures(Acc(self.c))
        Ensures(self.c is wraps)
        Ensures(MayCreate(self, 'x'))

    def __getattr__(self, name: str) -> object:
        Requires(Acc(self.__dict__['c']))
        Requires(Acc(self.__dict__['c'].__dict__[name]))
        return self.c.__dict__[name]

def wrapper_test() -> None:
    c = MyClass()
    wc = WrapperMyClass(c)
    Assert(wc.x == 10)
```

Listing 42: `WrapperMyClass`, where instance attributes not found it its own `__dict__` are looked for in the `__dict__` of a completely different class `MyClass`. Note that (:19) cannot be `self.c` because that would recurse via precondition

A.3 Assignment Lock

```
from nagini_contracts.contracts import *

@Complex
class LockClass:
    def __init__(self) -> None:
        self.__dict__['lock'] = False
```

```
        self.foo = 10
        self.__dict__['lock'] = True
        Ensures(Acc(self.foo))
        Ensures(self.foo == 10)
        Ensures(Acc(self.__dict__['lock']))
        Ensures(self.__dict__['lock'] == True)

    def __getattr__(self, item: str) -> object:
        return None

    def __setattr__(self, name: str, value: int) -> None:
        Requires(MaySet(self, name))
        Requires(Acc(self.lock))
        Requires(self.lock == False)

        self.__dict__[name] = value

        Ensures(Acc(self.lock))
        Ensures(Acc(self.__dict__[name]))
        Ensures(self.__dict__[name] is value)

def lock_test() -> None:
    l = LockClass()
    Assert(l.foo == 10)

    l.__dict__['lock'] = False  # without this, lines below will fail
    l.foo = 31
    Assert(l.foo == 31)
```

Listing 43: `LockClass().lock` that determines if instance attributes can be modified with the (dot) operator. `__getattr__` needed for mypy

A.4 `__getattribute__` and `__getattr__` example

```
from nagini_contracts.contracts import *

@Complex
class GetattributeStuff:
    def __init__(self) -> None:
        self.x = 1
        Ensures(Acc(self.x))
        Ensures(self.x == 1)
        Ensures(MayCreate(self, 'y'))
        Ensures(MayCreate(self, 'z'))

    def foo(self) -> object:
        Ensures(Result() == 50)
        return 50

    def __getattr__(self, name: str) -> object:
        Ensures(Result() == 10)
        return 10

    def __getattribute__(self, name: str) -> object:
        Requires(MaySet(self, name))

        # Auto generated:
        # Ensures(
        #     Implies(
        #         name in PSet("foo", "__dict__", ...),
        #         Result() == object.__getattribute__(self, name))
        # )

        if name == "z":
            return 100
        else:
            return object.__getattribute__(self, name)


```

```
def getattribute_example() -> None:
    g = GetattributeStuff()

    Assert(g.x == 1)        # g.__getattribute__('x') == 1    WITH __getattr__
    Assert(g.y == 10)       # g.__getattribute__('y') == 10   WITH __getattr__
    Assert(g.z == 100)      # g.__getattribute__('z') == 100  WITH __getattr__

    Assert(g.foo() == 50)          # does not call __getattribute__
    Assert(g.__dict__['x'] == 1)   # does not call __getattribute__

    Assert(object.__getattribute__(g, "x") == 1)          # same as g.__dict__['x']
    # Assert(object.__getattribute__(g, "y") == 10)       # will not work
    # Assert(object.__getattribute__(g, "z") == 100)      # will not work
```

Listing 44: Interaction between __getattribute__ and __getattr__

A.5 Simple Parent

```
from nagini_contracts.contracts import *

class SimpleParent:
    def __init__(self) -> None:
        self.x = "15"
        Ensures(Acc(self.x))
        Ensures(self.x == "15")

@Complex
class ComplexChild(SimpleParent):
    def __init__(self) -> None:
        super().__init__()
        Ensures(Acc(self.x))
        Ensures(self.x == "15")

def simple_parent_test() -> None:
    p = SimpleParent()
    c = ComplexChild()

    Assert(p.x == "15")
    Assert(c.x == "15")

    Assert(c.__dict__['x'] == "15")
    # Assert(p.__dict__['x'] == "15")   # will not work, SimpleParent is non-complex.
```

Listing 45: Non-complex Parent class instance attributes accesed from a complex Child class instance using a __dict__

B Additional code examples

B.1 __getattr__ and __getattribute__ method access

```
class MyClass:
    def x(self):
        return 10

    def __getattr__(self, name: str):
        return lambda: 20

    def __getattribute__(self, name: str):
        if name == "y":
            return lambda: 30
        else:
            return object.__getattribute__(self, name)

def hidden_code(self, name):
    try:
        if _getattribute := getattr(self, "__getattribute__", False):
            return _getattribute(name)
```

```
        return object.__getattribute__(self, name)
    except AttributeError as e:
        if _getattr := getattr(self, "__getattr__", False):
            return _getattr(name)
        raise e

if __name__ == "__main__":
    c = MyClass()
    try:
        print(hidden_code(c, "x")(), end=" ")       # c.x()       prints "10"
    except Exception as e:
        print(e, end=" ")

    try:
        print(hidden_code(c, 'abcd')(), end=" ")    # c.abcd()    prints "20"
    except Exception as e:
        print(e, end=" ")

    try:
        print(hidden_code(c, 'y')(), end=" ")       # c.y()       prints "30"
    except Exception as e:
        print(e, end=" ")
```

Listing 46: `__getattr__` and `__getattribute__` approximate behavior w.r.t. class method access

B.2 Keydict definition

```
field keydict_val: Option[Ref]

method keydict___init__(self: Ref)
    ensures forall key: Ref :: {keydict___item__(self, key)} acc(keydict___item__(self
    , key).keydict_val) && !keydict___contains__(self, key)

function keydict___item__(self: Ref, key: Ref): Ref
    ensures keydict___item__inv(self, result) == key

function keydict___item__inv(self: Ref, val_ref: Ref): Ref

function keydict___contains__(self: Ref, key: Ref) : Bool
    requires acc(keydict___item__(self, key).keydict_val, wildcard)
{
  keydict___item__(self, key).keydict_val.isSome
}

function keydict___getitem__(self: Ref, key: Ref) : Ref
    requires acc(keydict___item__(self, key).keydict_val, wildcard)
    requires keydict___contains__(self, key)
{
  keydict___item__(self, key).keydict_val.value
}

method keydict___setitem__(self: Ref, key: Ref, item: Ref)
    requires acc(keydict___item__(self, key).keydict_val)
    ensures acc(keydict___item__(self, key).keydict_val)
    ensures keydict___item__(self, key).keydict_val.value == item
    ensures keydict___contains__(self, key)
    ensures keydict___getitem__(self, key) == item
{
    keydict___item__(self, key).keydict_val := Some(item)
    // assert 10 == 15   // sanity check
}

// Of all possible keys, which have been set in the keydict
function keydict_keys(self:Ref ): Set[Ref]
    requires forall key: Ref :: {keydict___item__(self, key)} acc(keydict___item__(
    self, key).keydict_val, wildcard)
    ensures forall key: Ref :: {keydict___contains__(self, key)} {key in result}
    keydict___contains__(self, key) == (key in result)

```

```
40 // Of the keys in the Set[Ref], which have been set in the keydict
41 function keydict_keys_specific(self: Ref, keys: Set[Ref]): Set[Ref]
42     requires forall key: Ref :: {key in keys} key in keys ==> acc(keydict___item__(
       self, key).keydict_val, wildcard)
43     ensures forall key: Ref :: {key in keys} key in keys ==> keydict___contains__(self
       , key) == (key in result)
44
45 function keydict_values(self: Ref): Set[Ref]
46     requires forall key: Ref :: {keydict___item__(self, key)} acc(keydict___item__(
       self, key).keydict_val, wildcard)
47     ensures forall value: Ref :: {value in result}
48     (exists key:Ref :: {keydict___contains__(self, key)} (keydict___contains__(self,
       key) && (keydict___getitem__(self, key) == value))) == value in result
49
50
51 function keydict___len__(self: Ref): Int
52     requires forall key: Ref :: {keydict___item__(self, key)} acc(keydict___item__(
       self, key).keydict_val, wildcard)
53     ensures result == |keydict_keys(self)|
```

Listing 47: Full definition of Keydict, with helper methods

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

(●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Verification of Python Code with a Dynamic Object Model

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Vitolins	Edgars

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 14.05.2024	Edgars Vitolins

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard