Nagini: A Static Verifier for Python

Marco Eilers and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{marco.eilers, peter.mueller}@inf.ethz.ch

Abstract. We present Nagini, an automated, modular verifier for statically-typed, concurrent Python 3 programs, built on the Viper verification infrastructure. Combining established concepts with new ideas, Nagini can verify memory safety, functional properties, termination, deadlock freedom, and input/output behavior. Our experiments show that Nagini is able to verify non-trivial properties of real-world Python code.

1 Introduction

Dynamic languages have become widely used because of their expressiveness and ease of use. The Python language in particular is popular in domains like teaching, prototyping, and more recently data science. Python's lack of safety guarantees can be problematic when, as is increasingly the case, it is used for critical applications with high correctness demands. The Python community has reacted to this trend by integrating type annotations and optional static type checking into the language [20]. However, there is currently virtually no tool support for reasoning about Python programs beyond type safety.

We present Nagini, a sound verifier for statically-typed, concurrent Python programs. Nagini can prove memory safety, data race freedom, and user-supplied assertions. Nagini performs *modular* verification, which is important for verification to scale and to be able to verify libraries, and *automates* the verification process for programs annotated with specifications.

Nagini builds on many techniques established in existing tools: (1) Like VeriFast [10] and other tools [4,19,22], it uses separation logic style permissions [16] in order to locally reason about concurrent programs. (2) Like .NET Code Contracts [7], it uses a contract library to enable users to write code-level specifications. (3) Like many verification tools [2, 6, 11, 13], it verifies programs by encoding the program and its specification into an intermediate verification language [1, 8], namely Viper [14], for which automatic verifiers already exist.

Nagini combines these techniques with new ideas in order to verify advanced properties and handle the dynamic aspects of Python. In particular, Nagini implements a comprehensive system for verifying finite blocking [5] and input/output behavior [18], and builds on Mypy [12] to verify safety while also supporting important dynamic language features. Nagini is intended for verifying substantial, real-world code, and is currently used to verify the Python implementation of the SCION internet architecture [3]. To our knowledge, it is the first tool to enable automatic verification of Python code. Existing tools for JavaScript [21, 24]

also target a dynamic language, but focus on faithfully modeling JavaScript's complex semantics rather than practical verification of high-level properties.

Due to its wide range of verifiable properties, Nagini has applications in many domains: In addition to memory safety, programmers can choose to prove that a server implementation will stay responsive, that data science code has desired functional properties, or that algorithms terminate and preserve certain invariants, for example in a teaching context. Nagini is open-source and available online[1], and can be used from the popular PyCharm IDE via a prototype plugin.

In this paper, we describe Nagini's supported Python subset and specification language, give an overview of its implementation and the encoding from Python to Viper, and provide an experimental evaluation of Nagini on real-world code.

2 Language and Specifications

Python Subset: Nagini requires input programs to comply to the static, nominal type system defined in PEP 484 [20] as implemented in the Mypy type checker [12], which requires type annotations for function parameters and return types, but can normally infer types of local variables. Nagini fully supports the non-gradual part of Mypy's type system, including generics and union types.

The Python subset accepted by Mypy and Nagini can accomodate most real Python programs, potentially via some workarounds like using union types instead of structural typing. While our subset is statically typed, it includes many features and potential pitfalls not found in static languages, such as dynamic addition and removal fields from objects. Some other features like reflection and dynamic code generation are not supported.

Where compromises are necessary, Nagini aims for modularity, performance, and completeness for features typically found in user code over general support for all language features. As an example, Nagini works with a simplified model of Python's object attribute lookup behavior: A simple attribute access in Python leads to the invocation of several "magic" methods, which, if modelled correctly, would result in an overhead that would likely make automatic verification intractable. Nagini exploits the fact that these methods are mostly used to implement decorators, metaclasses, and system libraries, but rarely in user code. It assumes the default behavior of those methods, and implements direct support for frequently-used decorators and metaclasses that change their behavior. Importantly, Nagini flags an error if verified programs override these methods or are otherwise outside the supported subset, and is therefore sound.

Specification Language: Nagini includes a library of specification functions similar to .NET Code Contracts [7] to express pre- and postconditions, loop invariants, and other assertions. Calls to these functions are interpreted as specifications by Nagini, but can be automatically removed before execution. Users can annotate Mypy-style type stub files for external libraries with specifications; the

[1] https://github.com/marcoeilers/nagini

```
1  from nagini_contracts.contracts import *
2  from typing import List
3  import db
4
5  class Ticket:
6      def __init__(self, show: int, row: int, seat: int) -> None:
7          self.show_id = show
8          self.row, self.seat = row, seat
9          Fold(self.state())
10         Ensures(self.state() and MayCreate(self, 'discount_code'))
11
12     @Predicate
13     def state(self) -> bool:
14         return Acc(self.show_id) and Acc(self.row) and Acc(self.seat)
15
16 def order_tickets(num: int, show_id: int, code: str=None) -> List[Ticket]:
17     Requires(num > 0)
18     Exsures(SoldoutException, True)
19     seats = db.get_seats(show_id, num)
20     res = []  # type: List[Ticket]
21     for row, seat in seats:
22         Invariant(list_pred(res))
23         Invariant(Forall(res, lambda t: t.state() and
24                                 Implies(code is not None, Acc(t.discount_code))))
25         Invariant(MustTerminate(len(seats) - len(res)))
26         ticket = Ticket(show_id, row, seat)
27         if code:
28             ticket.discount_code = code
29         res.append(ticket)
30     return res
```

Fig. 1. Example program demonstrating Nagini's specification language. Contract functions are highlighted in italics. Note that functional specifications and postconditions are largely omitted to highlight the different specification constructs.

program will then be verified assuming they are correct. A detailed explanation of the specification language can be found in Nagini's Wiki[2].

An example of an annotated program is shown in Fig. 1. The first two lines import the contract library and Python's library for type annotations. Pre- and postconditions are declared via calls to the contract functions Requires and Ensures in lines 17 and 10, respectively. The arguments of these functions are interpreted as assertions, which can be side-effect free boolean Python expressions or calls to other contract functions. Similarly, loops must be annotated with invariants (line 22), and special *exceptional* postconditions specify which exceptions a method may raise, and what postconditions must hold in this case. The Exsures annotation in line 18 states that a SoldoutException may be raised and makes no guarantees in this case. The invariant MustTerminate in line 25 specifies that the loop terminates; the argument represents a ranking function [5].

Like the underlying Viper language, Nagini uses Implicit Dynamic Frames (IDF) [23], a variation of separation logic [16], to achieve framing and allow local reasoning in the presence of concurrency. IDF establishes a system of *permissions* for heap locations that roughly corresponds to separation logic's points-to predicates. Methods may only read or write heap locations they currently hold

[2] https://github.com/marcoeilers/nagini/wiki

a permission for, and can specify which permissions they require from and give back to their caller in their pre- and postconditions. Since there is only ever a single permission per heap location, holding a permission guarantees that neither other threads nor called methods can modify the respective location.

In Nagini, a permission is created when a field is assigned to for the first time; e.g., when executing line 9, the `__init__` method will have permission to three fields. Permission assertions are expressed using the `Acc` function (line 14). Assertions can be abstracted over using predicates [17], declared in Nagini by using annotated functions (line 12). In the example, the constructor of `Ticket` bundles all available permissions in the predicate `state` using the ghost statement `Fold` in line 9 and subsequently returns this predicate to its caller via its postcondition.

In addition, Nagini offers a second kind of permission that allows *creating* a field that does not currently exist, but cannot be used for reading (since that would cause a runtime error). Constructors implicitly get this kind of permission for every field mentioned in a class; in the example, such a permissions is returned to the caller (line 10) and used in line 28. The loop invariant contains the permission to modify the `res` list using one of several built-in predicates for Python's standard data types (line 22) as well as permissions to the fields of all objects in the list (line 23). This kind of *quantified permission* [15], corresponding to separation logic's iterated separating conjunction, is one of two supported ways to express permissions over unbounded numbers of heap locations.

Other contract functions allow specifying, e.g., I/O behavior, and some have variations for advanced users, e.g., the `Forall` function can take trigger expressions to specify when the underlying SMT solver should instantiate the quantifier.

Verified properties: Nagini verifies some safety properties by default: Verified programs will not raise runtime errors or undeclared exceptions. The permission system guarantees that verified code is memory safe and free of data races. Nagini also verifies some properties that Mypy only checks optimistically, e.g., that referenced names are defined before they are used. As an example, if the `Ticket` class were defined after the `order_tickets` function, Nagini would not allow calls to the function *before* the class definition, because of the call in line 26.

Beyond this, Nagini can verify 1) functional properties, 2) input/output properties, i.e., which I/O operations may or must occur, using a generalization of the method by Penninckx et al. [18], and 3) finite blocking [5], i.e., that no thread blocks indefinitely when trying to acquire a lock or join another thread, which includes deadlock freedom and termination. Verification is modular in the sense that adding code to a program only requires verifying the added parts; any code that verified before is guaranteed to still verify. Top level statements are an exception and have to be reverified when any part of the program changes, since Python's import mechanism is inherently non-modular.

3 Implementation

Nagini's verification workflow is depicted in Fig. 2. After parsing, Nagini invokes the Mypy type checker on the input and rejects the program if errors are found.

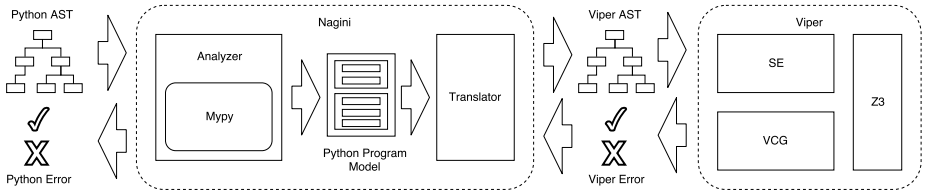

Fig. 2. Nagini verification workflow.

It then analyzes the input program and extracts structural information into an internal model, which is then encoded into a Viper program. The program is verified using one of the two Viper backends, based on either symbolic execution (SE) or verification condition generation (VCG), respectively. Any resulting Viper-level error messages are mapped back to a Python-level error.

Encoding: Nagini encodes Python programs into Viper programs that verify only if the original program was correct. At the top level, Viper programs consist of *methods*, whose bodies contain imperative code, side-effect free *functions*, and the aforementioned *predicates*, as well as *domains*, which can be used to declare and axiomatize custom data types. The structure of a created Viper program roughly follows the structure of the Python program: Each function in the Python program corresponds to either a method, a function, or a predicate in the Viper program, depending on its annotation. Additional Viper methods are generated to check proof obligations like behavioral subtyping and to model the execution of all top level statements.

Nagini maintains various kinds of ghost state, e.g., for verifying finite blocking and to represent which names are currently defined. It models Python's type system using a Viper domain axiomatized to reflect subtype relations. Nagini desugars complex Python language constructs into simple ones that exist in Viper, but subtle language differences often require additional effort in the encoding. As an example, Viper distinguishes references from primitive values whereas Python does not, requiring boxing and unboxing operations in the encoding.

Tool interaction: Nagini is invoked on an annotated Python file, and verifies this file and all (transitive) imports without user interaction. It then outputs either a success message or Python-level error messages that indicate type or verification errors, use of unsupported features, or invalid specifications, along with the source location. As an example, removing the Fold statement in line 9 of Fig. 1 yields the error message "Postcondition of ___init___ might not hold. There might be insufficient permission to access self.state(). (example.py@10.16)".

4 Evaluation

In addition to having a comprehensive test suite of over 12,500 lines of code, we have evaluated Nagini on a set of examples containing (parts of) implementations

	Example	LOC / Spec.	Viper LOC	SF	FC	FB	IO	T_{Seq}	T_{Par}
1	rosetta/quicksort	31 / 10	635	✓	-	✓	-	8.48	8.31
2	interactivepython/bst	145 / 65	947	✓	✓	-	-	57.44	41.80
3	keon/knapsack	33 / 10	864	✓	-	-	-	19.39	14.49
4	wikipedia/duck_typing	19 / 0	486	✓	-	-	-	1.82	1.92
5	scion/path_store	207 / 94	2133	✗	-	-	-	51.37	35.26
6	example	40 / 19	736	✓	-	✓	-	6.11	5.91
7	verifast/brackets_checker	143 / 82	1081	✓	✓	✓	✓	7.66	6.63
8	verifast/putchar_with_buffer	139 / 88	865	✓	-	✓	✓	4.74	4.29
9	chalice2viper/watchdog	66 / 22	769	✓	-	✓	-	3.66	3.41
10	parkinson/recell	46 / 25	561	✓	✓	-	-	2.09	2.07

Fig. 3. Experiments. For each example, we list the lines of code (excluding whitespace and comments), the number of those lines that are used for specifications, the length of the resulting Viper program, properties (SF = safety, FC = functional correctness, FB = finite blocking, IO = input/output behavior) that could be verified (✓), could not be verified (✗) or were not attempted (-), and the verification times with Viper's SE backend, sequential and parallelized, in seconds.

of standard algorithms from the internet[3], the example from Fig. 1, a class from the SCION implementation, as well as examples from other verifiers translated to Python. Fig. 3 shows the examples and which properties were verified; the functional property we proved for the binary search tree implementation is that it maintains a sorted tree. The examples cover language features like inheritance (example 10), comprehensions (3), dynamic field addition (6), operator overloading (3), union types (4), threads and locks (9), as well as specification constructs like quantified permissions (6) and predicate families (10). Nagini correctly finds an error in the SCION example and successfully verifies all other examples.

The runtimes shown in Fig. 3 were measured by averaging over ten runs on a Lenovo Thinkpad T450s running Ubuntu 16.04, Python 3.5 and OpenJDK 8 on a warmed-up JVM. They show that Nagini can effectively verify non-trivial properties of real-life Python programs in reasonable time. Due to modular verification, parts of a program can be verified independently and in parallel (which Nagini does by default), so that larger programs will not inherently lead to performance problems. This is demonstrated by the speedup achieved via parallelization on the two larger examples; for the smaller ones, verification time is dominated by a single complex method. Additionally, the annotation overhead is well within the range of other verification tools [9].

Acknowledgements. Thanks to Vytautas Astrauskas, Samuel Hitz, and Fábio Pakk Selmi-Dei for their contributions to Nagini. We gratefully acknowledge support from the Zurich Information Security and Privacy Center (ZISC).

[3] We chose examples that do not make use of dynamic features or external libraries from rosettacode.org, interactivepython.org and github.com/keon/algorithms.

References

1. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs, pp. 364–387. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
2. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Commun. ACM 54(6), 81–91 (Jun 2011)
3. Barrera, D., Chuat, L., Perrig, A., Reischuk, R.M., Szalachowski, P.: The scion internet architecture. Commun. ACM 60(6), 56–65 (May 2017)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 115–137. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
5. Boström, P., Müller, P.: Modular Verification of Finite Blocking in Non-terminating Programs. In: Boyland, J.T. (ed.) European Conference on Object-Oriented Programming (ECOOP). LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl (2015)
6. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte, W.: Vcc: Contract-based modular verification of concurrent c. In: 2009 31st International Conference on Software Engineering - Companion Volume. pp. 429–430 (May 2009)
7. Fähndrich, M., Barnett, M., Logozzo, F.: Code contracts. http://research.microsoft.com/contracts (2008)
8. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers, pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
9. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: End-to-end security via automated full-system verification. In: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014. pp. 165–181 (2014)
10. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java, pp. 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (May 2015)
12. Lehtosalo, J., et al.: Mypy - optional static typing for python. http://mypy-lang.org (2017)
13. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness, pp. 348–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
14. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016)
15. Müller, P., Schwerhoff, M., Summers, A.J.: Automatic verification of iterated separating conjunctions using symbolic execution. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 405–425. Springer International Publishing, Cham (2016)
16. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings. pp. 1–19 (2001)

17. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–258. POPL '05, ACM, New York, NY, USA (2005)
18. Penninckx, W., Jacobs, B., Piessens, F.: Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs, pp. 158–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
19. Piskac, R., Wies, T., Zufferey, D.: Grasshopper. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 124–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
20. van Rossum, G., Lehtosalo, J., Langa, Ł.: Type Hints. https://www.python.org/dev/peps/pep-0484/ (2014)
21. Santos, J.F., Maksimovic, P., Naudziuniene, D., Wood, T., Gardner, P.: JaVert: JavaScript verification toolchain. PACMPL 2(POPL), 50:1–50:33 (2018)
22. Smans, J., Jacobs, B., Piessens, F.: Vericool: An automatic verifier for a concurrent object-oriented language. In: Barthe, G., de Boer, F.S. (eds.) Formal Methods for Open Object-Based Distributed Systems. pp. 220–239. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
23. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames. ACM Trans. Program. Lang. Syst. 34(1), 2:1–2:58 (May 2012)
24. Stefanescu, A., Park, D., Yuwen, S., Li, Y., Rosu, G.: Semantics-based program verifiers for all languages. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016. pp. 74–91 (2016)