A Python-Based Design-by-Contract Evolutionary Algorithm Framework with Augmented Diagnostic Capabilities

Ashwin Panchapakesan*, *Graduate Student Member*, Rami Abielmona†, *Senior Member, IEEE*, and Emil Petriu‡, *Fellow, IEEE*
*School of EECS, University of Ottawa, Ottawa, Canada
†Research & Engineering, Larus Technologies Corporation, Ottawa, Canada
‡School of EECS, University of Ottawa, Ottawa, Canada
*apanc006@uottawa.ca, †rami.abielmona@larus.com, ‡petriu@eecs.uottawa.ca

Abstract—**Evolutionary algorithms are a class of algorithms that try to mimic natural, biological evolution *a la* Darwinian natural selection, to compute solutions to a given problem. They are especially useful when no well known strategies for computing solutions to such a problem exist. Evolutionary algorithms begin by creating a collection (population) of candidate solutions to the problem at hand; and through repeated application of genetic operators such as crossover and mutation, they iterate over multiple generations of this population, until they eventually converge onto an attractive solution. One important problem facing code implementing Evolutionary Algorithms is that due to the dynamic nature of the individual chromosomes in a population, simple coding errors lead to complex bugs that are difficult to both diagnose and debug. This problem is only exacerbated when attempting to develop the algorithms in a dynamically typed language such as Python. This paper presents a novel Evolutionary Algorithm framework for the Python programming language that implements design-by-contract, a paradigm in which each function and class must follow a contractual set of pre-conditions and post-conditions. Failure to follow the contract causes an error condition identifying the violated clause, thereby catching bugs earlier in the development process and in a more descriptive manner.**

***Keywords*—evolutionary algorithm, design-by-contract, python programming language**

I. Introduction

EVOLUTIONARY algorithms (EAs) are a class of algorithms that try to mimic natural, biological evolution *a la* Darwinian natural selection, to compute solutions to a given problem. They are especially useful when no well known strategies for computing solutions to such a problem exist. Though these algorithms have been studied since the 1950s [1], it wasn't until John Holland's work in the early 1970s that they gained popularity [2].

EAs begin by creating a collection (population) of random potential solutions (chromosomes). Over time, chromosomes in the population are selected in a manner proportional to their "fitness"[1] for crossover and mutation operations. These operations have the effect of combining the best solutions with each other in order to find (i.e. generate) better solutions over time. It is through the application of these operators that the population evolves into the next generation.

A recurring problem facing code implementing EAs is that due to the dynamic nature of the individual chromosomes in a population, it is far too simple to introduce bugs into source code. This problem is only exacerbated when attempting to code EAs in a dynamically typed language such as Python, whose design philosophy is, that as many expressions in code as possible should compute to some meaningful result or other. This is to say that one of Python's design decisions is to try to interpret the developer's code in some meaningful way, which may or may not be what the developer intended. Still, these design decisions do not take anything away from the deterministic nature of Python, keeping the language and its behaviors deterministic. [3].

As a result, most of the errors that a programmer sees when building or using a genetic framework in Python are far too cryptic[2], despite Python's excellent error reporting features. The reason for this is that any error that Python reports is due to an expression that it is unable to compute. This does not give the programmer much insight as to which part of the framework caused such an error. Debugging such an error requires spending time analyzing different components of the population as they undergo genetic operations and examining the output - a task that becomes intractable when population sizes increase. Further, since EAs rely heavily on chance[3], problems are not easily reproduced.

Thus, a more thorough debugging tool is required. This is solved with the use of the design-by-contract (DbC) paradigm. This is done by applying an existing DbC model checker (called PyContract [4]) to an existing, open-source EA framework [5] in order to improve the usability of the frame-

[1]a measure calculated by an objective function, reflecting how well the chromosome would solve the problem at hand

[2]The error message `AttributeError: 'list' object has no attribute 'chromosomes'` is perfectly reasonable and well written. However, in the context of an EA, it is very unclear which part of the code is causing this error.

[3]Typically, crossover occurs with high probability and mutation occurs with low probability

work. The framework allows programmers to write their own functions for genetic operators, population initialization, etc. Applying DbC to this framework would ensure that problems in the design of an EA (that are to be implemented using this framework) such as improper mutation or crossover operators would be caught before the evolution proceeds any further. Thus, any unfavorable outcome evolutionary results can be isolated down to the evolutionary parameters, declaring the source code bug-free (assuming that the contracts are written correctly).

The rest of this paper is organized as follows:

Sec. II provides a broad overview of some existing Python based frameworks for the development of various types of evolutionary algorithms. Sec. III introduces a problem and a solution to that problem that will be used to demonstrate the herein developed software package. Further, Sec. IV introduces the concept of design-by-contract and Sec. V presents two frameworks that implement it in Python. Finally, some results are shown and discussed in Sec. VI and some improvements are suggested in Sec. VII.

II. Previous Work

There are several frameworks geared towards the development of EAs in Python [6], [7], [8]. However, many of these frameworks suffer from not being very extensible, i.e. it is not always easy to solve a member of a new family of problems with an EA using that framework. Further, all of these frameworks suffer from the problem discussed in section I, namely that the framework provides no easy methodology to debug any evolutionary algorithm developed using it. Design-by-contract is one way of supplementing Python's native error reporting to assist with the debugging process. In this work, a Genetic Algorithm (GA) is used as the EA of choice for implementation within the developed framework.

III. A Genetic Algorithm to solve the Traveling Salesman Problem

A GA to solve the Traveling Salesman Problem (TSP), on the well known Berlin-52 map [9], [10] (which contains 52 cities), is used to demonstrate parts of the herein developed software framework.

The GA begins by generating an initial population of random tours for the traveling salesman (as described in Sec III-A). Once this initial population is created using well-defined initialization parameters (e.g. population size, individual size, etc.), it selects individuals to probabilistically crossover and mutate, thereby making child individuals, which make up the next generation of this population. Repeating this process of selection (which is typically dependent on individual fitness [4]), crossover and mutation over several generations allows the GA to converge on an optimal solution.

[4]The fitness function used in GA is described in Sec III-B

A. An Individual

Within a GA, individuals are made up of one chromosome. This chromosome is made up of a list of 52 integers, each one representing a city on the map. In order for an individual to represent a feasible solution in the solution space, the chromosome is an ordered list, containing a permutation of $\{0, 1, 2, ..., 51\}$, thus making it a valid tour for the TSP.

B. Fitness of an Individual

Since the optimal solution for this problem is an individual whose tour length is minimal, the fitness of each individual could be the length of the tour it encodes. However, since the goal is to maximize the fitness, a better fitness measure of an individual would be the negative of the length of the tour the individual encodes. This can be easily computed under the following assumptions:

1) Each of the 52 cities on the map is represented as a point on the xy plane
2) There is a straight line (i.e. road) connecting every pair of the 52 cities on the map

Thus, the fitness of an individual can be computed as shown in eq. 1

$$fitness = -dist(city_{51}, city_0) - \sum_{c=0}^{50} dist(city_c, city_{c+1}) \tag{1}$$

where
$dist(c_i, c_j) = \sqrt{(city_i.x - city_j.x)^2 + (city_i.y - city_j.y)^2}$

C. Selecting Individuals

In order to create new individuals out of existing individuals, two are selected for crossover and mutation operations. The selection mechanism is fitness proportional, meaning that individuals with higher fitness values are selected more often than individuals with lower fitness values. A tournament selection scheme [11] was used in this GA. In this scheme, *tournSize* many individuals are selected at random from the population and the *numWinners* fittest of these individuals are selected for the crossover operation. The values used for *tournSize* and *numWinners* are detailed in Sec. VI.

D. Crossover

A crossover is an operation that takes two parent individuals and creates a new child individual, whose chromosomes are comprised of parts of the corresponding chromosomes from both parents. In the case of the TSP, one possible crossover function is defined as follows:

1) Select points A and B such that $0 < A < B < 51$
2) Create an empty child chromosome which is intended to hold 52 cities (i.e. a new tour for the traveling salesman)
3) Copy over all the cities between points A and B in the tour represented by $parent_1$ into the child chromosome
4) Copy over all the cities before point A and after point B in the tour represented by $parent_2$ into the

corresponding location in the child chromosome, as long as the city does not already exist between points A and B in the child chromosome.

5) Fill in the remaining cities in the child chromosome based on the order in which they appear in $parent_1$.
6) Insert this child chromosome into a new individual - the child individual of the crossover.

Note that it is imperative that the child individual of a crossover represent a feasible solution so as to ensure that the GA does not create individuals that are outside the solution space.

E. Mutation

A mutation is an operation that slightly changes an individual. One possible mutation is to swap the positions of two cities in the tour. Another possible mutation is to reverse the order of the cities in one contiguous part of the tour.

Again, note that it is imperative that a mutated individual must still represent a feasible solution so as to ensure that the GA does not create individuals that are outside the solution space.

IV. Introducing Design-by-Contract

A. The Design-by-contract Principle

Design-by-contract (DbC) is the principle that interactions, between modules of a software system, should be governed by precise specifications so as to ensure that the code is faithful to those very specifications and does not produce unintended effects (bugs). The contracts will cover mutual obligations (i.e. preconditions), benefits (i.e. postconditions), and consistency constraints (i.e. invariants) [12]. This principle is especially applicable in large modular systems with multiple levels of abstraction, such as a framework for implementing GAs.

B. The Advantage of Using DbC in this Framework

Two of the core design principles of the Python programming language are:

1) Almost all expressions that a programmer tries to compute must be computed in some meaningful way.
2) All error reporting and tracebacks should be meaningful in order to help a programmer better debug their program. In particular, core-dumps and crashes should be avoided.

In most cases, these are very desirable principles in a programming language. However, when working with GAs, where even simple off-by-one errors can cause individual solutions in a population to leave the solution space and where mutation and crossover operations are probabilistic, bugs become difficult to reproduce and traditional step-through debugging becomes infeasible (except in a small subset of the program's functional body). While this error reporting explains why the GA may crash, it does very little to reveal the real source of the error (for example, it may be clear that two variables of very different data types may not be added together, but the bug that causes either variable to be of that different data type is not identified). Thus, whereas the error messages may be well written for most other algorithms, they are rendered far too cryptic to help debug a GA.

Further, due to the amount of data that a GA works with on the stack, traditional print-debugging (or logging) would also be infeasible as the signal-to-noise ratio in the debug logs would be too low to be useful to a developer.

One particularly difficult bug was found to be caused by mistyping `if a>b: a,b = b,a` as `if a<b: a,b = b,a` in the crossover function for the GA solving the TSP. This had the effect of causing the following error, mid-evolution: `IndexError: pop from empty list`. This is because the correctly implemented crossover function, despite implementing its specification accurately, made certain assumptions that were incorrect. These assumptions were incorrect due to the aforementioned mistyping. However, the raised `IndexError` does not provide any information as to the source of this error which causes the developer to be at a loss for where to start the debugging process. Note that such constraint checking goes beyond type checking of variables. For example, it could be argued that using DbC in this framework simply adds strong typing to the GA code written in the python programming language; such an argument is incomplete. The use of Dbc does allow for strong typing of the GA code. However, even with a strongly typed language (such as Java or C), semantics about an object aren't checked by the typing mechanism. For example, in a GA to solve a TSP of 52 cities, a valid chromosome is an array that encodes 52 cities (as mentioned in III-A). While a strongly typed language would enforce a that chromosome is an array, it does not validate that the array contains a permutation of $\{0, 1, 2, ..., 51\}$. Enforcing such constraints set by the specification makes DbC a useful to an evolutionary algorithms framework, as it allows a developer to catch bugs at the point of occurrence, rather than the point of program failure (software crash, core dump, etc).

While other frameworks for creating evolutionary algorithms in python do exist [6], [7], [8], the herein presented software package is the only one that incorporates DbC in order to optimize the development time for a new evoilutionary algorithm.

In general, the application of DbC enforces the constraints for all classes and functions in code. These constraints are typically developed as part of the formal specification for that class or function during the design phase of the software development cycle. If and when a constraint is violated by a section of code, the contract checker (which is part of the DbC framework) raises an error which causes a software crash if left uncaught. This raised error contains a message indicative of which constraint has been violated, so that the developer may inspect the relevant section of code and debug it appropriately. It is important to note that such errors are raised as a result of a violation of the constraints set forth by the software design specifications; these are different from errors raised by the Python interpreter, which are typically due to syntax errors or errors pertaining to the semantics of the data structures used in the code. For example, a feasible chromosome to be used in the GA for the TSP is required to be a permutation of $\{0, 1, 2, ..., 51\}$. If a buggy crossover function returns a chromosome that is not such a permutation, the

Python interpreter will still be able to perform computations on it without raising any errors, but it would lead to erroneous results at the end of a run of the GA. Such a violation would cause the DbC framework to raise an error with a message indicating that the requirement that the product of crossover be a permutation of $\{0, 1, 2, ..., 51\}$ has been violated. This would cause the evolutionary simulation to crash, if the error is not explicitly caught and handled.

Finally, assuming that the GA runs without any errors, if the end result of a run of the GA is unfavorable or unexpected, it is unclear as to whether this divergence in expectations was caused by the stochastic nature of evolution (and suboptimal parameters thereof) or by faulty programming logic. As a result, implementing contracts for each of the functions in this framework allows programmers to catch errors in the programming logic early, trace the error to the buggy function in the program, and eliminate programming errors as the reason for unexpected results at the completion of a run of the GA (assuming that the correct contracts have been implemented).

V. DbC Frameworks for Python

In order to implement DbC to this GA framework, two packages were evaluated. These packages are reviewed in this section.

A. PyContracts

PyContracts is a DbC package that allows a programmer to annotate functions with contract expressions. The syntax of PyContracts closely follows the standard Python ReSTful documentation [13]. This makes the documentation itself serve two purposes:

1) the documentation for the function, which can then be automatically generated into official documentation by software such as Sphinx [14]
2) the contract expressions for DbC.

For example, the contract expressions in algorithm 1 for a function that multiplies two matrices[13] denotes that:

1) The input parameter `a` is a `nested list` of dimensions `M` rows and `N` columns, where `M` and `N` are positive numbers.
2) The input parameter `b` is a `nested list` of dimensions `N` rows and `P` columns, where `P` is a positive number.
    a) This ensures that `a` and `b` are of compatible dimensionalities.
3) The function returns an array of `M` rows and `P` columns.

Note however, that the set of contracts for this function neither mentions anything about the values of the inputs remaining unchanged over the execution of the function; nor provides a way to check this. In fact, PyContracts does not allow developers to write contracts about the values of variables before the function is run, to be checked after the execution of the function [15].

Upon further investigation, it is clear that PyContracts does not save variables on the stack at different points in time during execution. This implies that it is not possible to write contracts for post-conditions that express properties about the values variables before and after the function's execution. For the same reason, it is also impossible to write contracts that express properties of invariants.

As a result of these limitations, PyContracts is not a suitable DbC package for writing contracts for the generic EA framework.

Algorithm 1 Contracts for a Matrix Multiplication Function in PyContracts

```
@contract
def matrix_multiply(a, b):
    ''' Multiplies two matrices
        together.
      :param a: The first matrix.
          Must be a 2D array.
        :type a: array[MxN], M>0, N
            >0
      :param b: The second matrix.
                Must be of
                    compatible
                    dimensions.
        :type b: array[NxP], P>0

          :rtype: array[MxP]
    '''
```

B. PyContract

PyContract is a DbC package that allows a programmer to annotate functions with contract expressions. Although its syntax is unlike PyContracts', it does allow for the development of richer contract expressions, making up for PyContracts' shortcomings.

For example, the contract expressions in algorithm 2 for a function that multiplies two matrices denotes that:

1) The input parameter `a` is a `nested list` of positive row and column dimensions.
2) The input parameter `b` is an `array` of positive row and column dimensions.
3) The number of rows in `b` is equal to the number of columns in `a`.
    a) This ensures that `a` and `b` are of compatible dimensionalities.
4) The function returns an array of `M` rows and `P` columns.
5) The inputs are unchanged

In addition, PyContract also allows for the expression of class invariants, thus overcoming another one of PyContracts' shortcomings. These are expressible with the `inv` declaration in the contract expressions along with referencing `self`. However, there is a limitation to PyContract's ability to express invariants in functions. For example, it is not possible to express contracts about loop invariants whose expressions

Algorithm 2 Contracts for a Matrix Multiplication Function in PyContract

```
def matrix_multiply(a, b):
  ''' Multiplies two matrices
       together.
     pre:
       isinstance(a, array)
       isinstance(b, array)
       len(a) > 0
       len(a[0]) > 0
       len(b) == len(a[0])
     post:
       __old__.a == a
       __old__.b == b
       isinstance(__return__, array
           )
       len(__return__) == len(a)
       len(__return__[0]) == len(b
           [0])
  '''
```

contain variables that are not class variables, but are instead local to the scope of the function itself. In order to express such invariants in this GA framework, a hybrid approach using both PyContract and assert statements native to Python was used.

As previously stated, the DbC implementation for this GA framework is a hybrid of PyContract and `assert` statements native to the Python programming language. For example, the crossover function described in section III-D has the contracts shown in algorithm 3 (explained in table V-B)[5].

Notice that there are no contracts that express invariants in the PyContract syntax. This is because contractual invariant clauses may express invariants that only refer to class variables. This does not include variables that are not bound to a defined class but are still within the local scope of the function for which the contract is written. Therefore, loop invariants that refer to loop counters cannot be checked using PyContract. As a result, the second part of the hybrid implementation of contract checking uses `assert` statements native to Python to enforce invariants which express properties of non-class-variables within the local scope of the function. For example, contractual loop invariants are expressed in algorithm 4.

These loop invariant contract expressions check to ensure that the invariant is True and the hypothesis guard is False in every iteration of the while-loop.

It is important to note that GAs themselves usually have a long runtime. Furthermore, contract checking implies that every time a function is called, the pre-conditions, post-conditions and invariants of that function are verified. In addition, checking post-conditions in this framework, especially against values of variables before the execution of the function requires making a copy of the memory stack before each

[5]For a comprehensive list of contracts used in the GA to solve the TSP, consult the documentation [5]

Algorithm 3 Contracts for the Crossover function for the Traveling Salesman Problem

```
def injectionco(p1, p2):
  """
    pre:
      isinstance(p1, list)
      isinstance(p2, list)
      len(p1) == len(p2)
      sorted(p1) == range(len(p1))
      sorted(p2) == range(len(p2))

    post[p1, p2]:
      p1 == __old__.p1
      p2 == __old__.p2
    post:
      isinstance(__return__, list)
      len(__return__) == len(p1)
      id(__return__) not in [id(p1
          ), id(p2)]
      forall(__return__, lambda
          city: city in p1 and city
           in p2)
      len(set(__return__)) == len(
          __return__)
  """
```

Algorithm 4 Contracts for the Crossover function for the Traveling Salesman Problem

```
def runTSPGA(kwargs):
  ...
  while g < maxGens:
    if testmode:
      assert g < maxGens
      assert best[1] < targetscore
  ...
```

execution of that function. The result is a drastic increase in the runtime of these functions, explicitly because of contract checking. In order to alleviate such effects of DbC on the GA framework, a new configuration parameter was introduced into the framework. This parameter (named `testmode`) is a boolean flag, which when set `True` forces contract checking on all functions for which a contract has been written. When this flag is set `False`, the contracts are not checked, allowing the GA framework to operate at its maximal efficiency without being interrupted by contract checking [5]. Therefore, the ideal usage of a simulation using this framework (now augmented with DbC) would be to run the simulation once for a shorter period of time (evolutionary time, not realtime). Once it is clear that all contracts are being followed, then the simulation may be run for the required (presumably longer) period of time without contract checking, so that it may run at an efficiency

Table I. Explanations of Contracts in Algorithm 3

Contract Expression	Explanation
`pre:`	The next block of indented expressions are pre-conditions of this function
`isinstance(p1, list)`	`p1` is a list
`isinstance(p2, list)`	`p2` is a list
`len(p1) == len(p2)`	`p1` and `p2` have equal number of elements
`sorted(p1) == range(len(p1))`	`p1` is a permutation of $\{0, 1, 2, ..., L - 1\}$ where `L` is the number in elements in `p1`
`sorted(p2) == range(len(p2))`	`p2` is a permutation of $\{0, 1, 2, ..., L - 1\}$ where `L` is the number in elements in `p2`
`post[p1, p2]:`	The next block of indented expressions are post-conditions of this function on the variables `p1` and `p2`
`p1 == __old__.p1`	`p1` remains unchaged as a result of executing this function
`p2 == __old__.p2`	`p2` remains unchaged as a result of executing this function
`post:`	The next block of indented expressions are general post-conditions of this function
`isinstance(__return__, list)`	A `list` is returned
`len(__return__) == len(p1)`	The number of elements in the returned list is equal to the number of elements in `p1`
`id(__return__) not in [id(p1), id(p2)]`	The returned list does not reside in the same memory location as either `p1` or `p2`
`forall(__return__, lambda city: city in p1 and city in p2)`	Every element in the returned list exists in `both p1` and `p2`
`len(set(__return__))== len(__return__)`	Every element in the returned list occurs exactly once

that is not hindered by contract checking. While developing the GA used to demonstrate this software package, the GA was limited to 10 generations before it was forcibly stopped, while running in debug mode. This is in contrast to the 200 generations the GA otherwise requires in order to converge on a desirable solution.

A list of all contracts implemented in the GA for the TSP developed using this framework is available in the official documentation of the Pyvolution package [5].

VI. Results

A GA to solve the TSP on the Berlin-52 map is used to demonstrate the herein developed software framework, as explained in Sec. III. The evolutionary parameters for a run of this GA are listed in Table II. Further, Table III compares the time required for a run of this GA[6] in debug mode and in non-debug mode.

Table II. Basic Evolutionary Parameters for a GA to Solve TSP on the Berlin-52 Map

Parameter	Description	Value
maxGens	Maximum number of generations before evolution is forcibly stopped	200
popSize	The number of chromosomes in every generation of the population	1000
crossProb	Probability of crossover between two selected chromosomes	0.7
mutProb	Probability of mutation between two selected chromosomes	0.05
tournSize	Number of chromosomes competing in a tournament	4
numWinners	Number of winners in each tournament	2

Table III. Elapsed Runtime for the GA to Solve the TSP

Mode	Maximum Elapsed Time	Minimum Elapsed Time	Average Elapsed Time	Standard Deviation
Debug	356.43s	339.78s	348.65s	3.98s
Non-debug	38.37s	34.63s	35.55s	0.73s

These results show that the herein developed software package executes code slower (by a factor of approximately 9.8) when run in debug mode (by setting `testmode` to `True`). This is because each set of constraints for every class, method and function is checked before and after every execution/call to that class, method or function. This allows the framework to alert the developer of a divergence from the formal specification of that piece of code by raising an exception with a message indicating the constraint being violated; the developer may then inspect the relevant section of code to debug. This is a clear advantage over waiting for such a divergence from the specification to cause an error elsewhere in the code, as it would not readily hint at the origin of the bug.

It is important to note that the constraints that must be enforced are to be specified by the developer. This infers that the specification of the constraints themselves may be buggy which introduces a source of potential error within the flow. Nevertheless, the expressions defining these constraints are generally simpler than the algorithms they govern, making it unlikely for a developer to introduce bugs in them.

Furthermore, the number of constraints the developer implements directly increases the run time of the algorithm in non-debug mode as compared to debug mode. This is due to the fact that each of these constraints must be checked every time the function is called or executed. Consider for example, the following constraints implemented in the crossover algorithm:

The constraints on lines 11 and 12 require a `memcpy` to be performed before the function is called, so as to check the

[6]Note that these statistics were gathered over 30 runs of the GA in each mode

Algorithm 5 Constraints in the Crossover Algorithm

```
1  def injectionco(p1, p2):
2     """
3       pre:
4         isinstance(p1, list)
5         isinstance(p2, list)
6         len(p1) == len(p2)
7         sorted(p1) == range(len(p1))
8         sorted(p2) == range(len(p2))
9
10      post[p1, p2]:
11        p1 == __old__.p1
12        p2 == __old__.p2
13      post:
14        isinstance(__return__, list)
15        len(__return__) == len(p1)
16        id(__return__) not in [id(p1
              ), id(p2)]
17        forall(__return__, lambda
              city: city in p1 and city
               in p2)
18        len(set(__return__)) == len(
              __return__)
19    """
```

values of p1and p2 before and after execution. Each of these are $O(n)$ operations for a chromosome enconding a tour of n cities. On the other hand, the constraint on line 17 checks that every city in the tour encoded by the returned child of crossover exists in both parents. This requires $O(n^2)$ time, as the `in` operator performed on a `list` compares every element in the list to check if the required element exists in the list. As this is performed n times (once for each element in the returned list), and each time this is performed, it is performed on two lists (p1 and p2), which takes $O(n)$ time by itself, this constraint requires $O(n^2)$ time to check.

VII. Future Work

Due to the large difference in runtime of the algorithm implemented using this framework in debug mode and non-debug mode (i.e. by setting `testmode` to `True` or `False` respectively), it would be beneficial to implement a static contract checking model as has been done in Java [16] in order for the checking to not influence the runtime of the code. However, this is fairly infeasible without restricting the usage and usability of Python as it is a dynamically typed language. Therefore, the necessity to check the contracts at runtime is not eliminated.

However, with the advent of just-in-time compilers, it might be feasible to perform contract checking only when it is predicted to be necessary , thus offering a reduction in the runtime of an algorithm implemented using this framework.

Table IV. Explanation of Constraints on Algorithm 5

Line	Constraint	Function
4	`isinstance(p1, list)`	The input parameter p1 should be of type list
5	`isinstance(p2, list)`	The input parameter p2 should be of type list
6	`len (p1) == len (p2)`	The two input parameters should contain an equal number of elements
7	`sorted(p1)==range(len(p1))`	p1 should be a permutation of $\{0, 1, 2, ..., 51\}$
8	`sorted(p2)==range(len(p2))`	p2 should be a permutation of $\{0, 1, 2, ..., 51\}$
11	`p1 == __old__.p1`	p1 should remain unchanged by the crossover algorithm (presented in Section III-D)
12	`p2 == __old__.p2`	p2 should remain unchanged by the crossover algorithm (presented in Section III-D)
14	`isinstance(__return__,list)`	An object of type list should be returned
15	`len( __return__ )==len(p1)`	The returned list should contain as many elements as does p1
16	`id(__return__) not in [id(p1 ), id(p2)]`	The returned list is a new object, occupying a space in memory that is not correspondent to p1 or p2
17	`forall ( __return__ , lambda city: city in p1 and city in p2)`	Every element in the returned list is present in both p1 and p2
18	`len(set(__return__)) == len( __return__ )`	There are no repeted elements in the returned list

VIII. Conclusions

It is clear that the herein developed software package, offers a reasonable improvement in the time required to develop evolutionary algorithms due to the augmented diagnostic capabilities that it affords (as explained in Section VI). However, this is at the cost of a slower elapsed time per run of a simulation, as can be seen in the data presented in Table III. Ultimately, if the developer is satisfied that the code is bug-free after having run the simulation in debug mode enough times, then the contract checking may be disabled (by setting `testmode` to `False`) and executing the code at the full speed otherwise afforded by Python, a speedup factor of 9.8 in the case of the GA solving the TSP. Note that this speedup is not constant, but is dependent on the number of constraints being enforced.

References

[1] N. A. Barricelli, “Esempi Numerici di processi di evoluzione,” *Methodos*, 1954.

[2] J. H. Holland, “Adaptation in natural and artificial systems, University of Michigan press,” *Ann Arbor, MI*, vol. 1, no. 97, p. 5, 1975.

[3] G. van Rossum, “Guido van Rossum on the History of Python,” San Francisco, 2011.

[4] J. Webb, “pycontract,” 2010.

[5] A. Panchapakesan, “Genetic 1.1 documentation,” 2012. [Online]. Available: http://packages.python.org/Pyvolution/settings.py.html#fields

[6] F. A. Fortin, “deap.” [Online]. Available: https://pypi.python.org/pypi/deap/0.9.1

[7] C. S. Perone, “Pyevolve,” 2009. [Online]. Available: https://pypi.python.org/pypi/Pyevolve/0.5

[8] M. Khoury, “pySTEP,” 2009. [Online]. Available: https://pypi.python.org/pypi/pySTEP/stable1.20

[9] G. A. Jayalakshmi, S. Sathiamoorthy, and R. Rajaram, “A hybrid genetic algorithmâĂŤa new approach to solve traveling salesman problem,” *International Journal of Computational Engineering Science*, vol. 2, no. 02, pp. 339–355, 2001.

[10] M. Affenzeller and S. Wagner, “SASEGASA: A new generic parallel evolutionary algorithm for achieving highest quality results,” *Journal of Heuristics*, vol. 10, no. 3, pp. 243–267, 2004.

[11] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, “Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications (The Morgan Kaufmann Series in Artificial Intelligence),” 1997.

[12] J.-M. Jazequel and B. Meyer, “Design by contract: The lessons of Ariane,” *Computer*, vol. 30, no. 1, pp. 129–130, 1997.

[13] A. Censi. (2012) PyContracts. [Online]. Available: http://andreacensi.github.com/contracts/

[14] ——. (2012) PyContracts 1.5.0 documentation. [Online]. Available: http://andreacensi.github.com/contracts/#decorating-a-function

[15] ——. Need help with PyContracts. [Accessed: 5 Dec, 2012]. [Online]. Available: http://www.reddit.com/r/Python/comments/11szm3/need_help_with_pycontracts/c6ph64p

[16] Escher Technologies Limited, “Perfect Developer.” [Online]. Available: http://www.eschertech.com/products/perfect_developer.php