Co-evolutionary automatic programming for software development

Arcuri, Andrea; Yao, Xin

DOI:
10.1016/j.ins.2009.12.019

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (Harvard):
Arcuri, A & Yao, X 2014, 'Co-evolutionary automatic programming for software development', *Information Sciences*, vol. 259, pp. 412-432. https://doi.org/10.1016/j.ins.2009.12.019

Information Sciences 259 (2014) 412–432

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Co-evolutionary automatic programming for software development ☆

Andrea Arcuri [a,*], Xin Yao [b]

[a] Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
[b] The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), The School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK

ARTICLE INFO

Article history:
Received 7 March 2008
Received in revised form 17 July 2009
Accepted 21 December 2009
Available online 4 January 2010

Keywords:
Automatic programming
Automatic refinement
Co-evolution
Software testing
Genetic programming

ABSTRACT

Since the 1970s the goal of generating programs in an automatic way (i.e., *Automatic Programming*) has been sought. A user would just define what he expects from the program (i.e., the requirements), and it should be automatically generated by the computer without the help of any programmer. Unfortunately, this task is much harder than expected. Although transformation methods are usually employed to address this problem, they cannot be employed if the gap between the specification and the actual implementation is too wide. In this paper we introduce a novel conceptual framework for evolving programs from their specification. We use genetic programming to evolve the programs, and at the same time we exploit the specification to co-evolve sets of unit tests. Programs are rewarded by how many tests they do not fail, whereas the unit tests are rewarded by how many programs they make to fail. We present and analyse seven different problems on which this novel technique is successfully applied.

1. Introduction

Writing a formal specification (e.g., in Z [48] or JML [31]) before implementing a program helps to identify problems with the system requirements. The requirements might be, for example, incomplete and ambiguous. Fixing these types of errors is very difficult and expensive during the implementation phase of the software development cycle. However, writing a formal specification might be more difficult than implementing the actual code, and that might be one of the reasons why formal specifications are not widely employed in industry [40].

However, if a formal specification is provided, then exploiting the specification for automatic generation of code would be better than employing software developers, because it would have a much lower cost. Since the 1970s the goal of generating programs in an automatic way has been sought [26]. A user would just define what he expects from the program (i.e., the requirements), and it should be automatically generated by the computer without the help of any programmer.

This goal has opened a field of research called *Automatic Programming* (also called *Automatic Refinement*) [43]. Unfortunately, this task is much harder than it was expected. Transformation methods are usually employed to address this problem (e.g. [26,11,47,39,51,37]). The requirements need to be written in a formal specification, and sequences of transformations are used to transform these high-level constructs into low-level implementations. Unfortunately, this process can rarely be automated completely, because the gap between the high-level specification and the target implementation language might be too wide.

☆ A preliminary version of this paper was presented in the IEEE International Conference on Automated Software Engineering 2007 [7].

* Corresponding author. Tel.: +44 47 678 28 244.
E-mail addresses: arcuri@simula.no (A. Arcuri), x.yao@cs.bham.ac.uk (X. Yao).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2009.12.019

In this paper we present a novel conceptual framework for evolving programs from their specification. A population of candidate programs co-evolves with a population of unit tests. We employ genetic programming to evolve the candidate programs, whereas search based software testing techniques are employed to evolve the unit tests. The fitness value of the candidate programs depends on how many tests they pass, whereas the unit tests are rewarded based on how many programs they make to fail. We call this approach *Co-evolutionary Automatic Programming* [7].

This type of co-evolution is similar to what happens in nature between *predators* and *preys*. For example, faster preys escape predators more easily, and hence they have a higher probability of generating offspring. This influences the predators, because they need to evolve as well to get faster if they want to feed and survive. In our context, the evolutionary programs can be considered as preys, whereas the unit tests are predators. The programs need to evolve to fix their faults, and this will make them "escape" from the unit tests. In our analogy, a program with few faults is a "fast" program. If a program manages to escape from the unit tests, it will have a higher probability of reproducing. On the other hand, if it has many faults, then it would be "slow", hence it is likely to be "killed" by the unit tests.

Once the programs evolve to be "fast" enough to escape from the unit tests, new mutations that make them "faster" will not spread, because all the programs will have the same fitness, and the chance of survival will be the same for each of them. Unfortunately for the programs, they cannot rest for long. In fact, the unit tests are evolving as well (i.e., the "slow" ones die, whereas the ones with new good mutations reproduce), and sooner or later they will get "faster". When this event happens, the programs do not have all the same chance of survival, and only the "fastest" among them will survive. Hopefully, this co-evolution will produce an *arms race* in which each of the two populations continually improve its performance, and that would lead to the evolution of a program that satisfies the given formal specification.

The idea of co-evolving programs and test cases is not entirely new [23]. The novelty of this paper lies in its original application in software engineering, i.e. automatic refinement, and in all the related problems that require to be addressed (e.g., how to automatically generate the fitness function and how to sample proper test cases for non-trivial software).

Although there is a wide range of successful techniques that are inspired by nature (especially in optimisation and machine learning), the aim of this paper is not to mimic a natural process. If we can improve the performance of a nature inspired system by using something that is not directly related to its inspiring natural process, we should use it. For example, to improve the performance of our framework, in this paper we also investigate the role of *Automated N-version Programming* [19]. Furthermore, we exploit the formal specification to divide the set of unit tests into sub-sets, each of them specialised in trying to find faults related to different reasons of program failure. To evolve complex software composed of several functions, if there are relations among the functions (e.g., an hierarchy of dependencies), then we exploit these relations. For example, we can use them to choose the order in which the specifications of the single functions are automatically refined, and then we use the programs evolved so far to help the refinement of other functions.

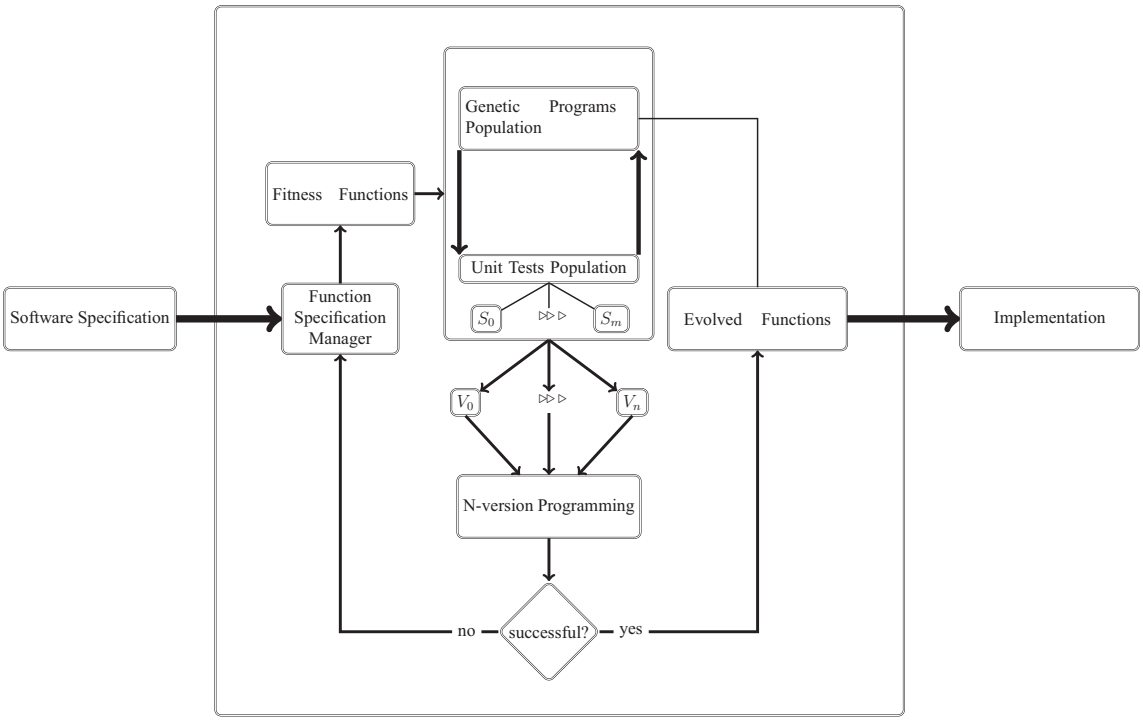

Fig. 1. High level view of the proposed framework for co-evolutionary automatic programming. *S* means "sub-population", whereas *V* means "version".

We have implemented a prototype of our conceptual framework in order to evaluate and analyse it. Seven different problems are used in our empirical study. Fig. 1 shows a high-level view of the framework. A formal specification is taken as input. A set of test cases is automatically generated based on the formal specification. A population of programs is evolved for a fixed number of generations to pass these test cases. At each generation, new test cases are co-evolved to find new faults in the evolving programs. The best individual in the last generation is chosen as an implementation of the formal specification. Because this individual could be faulty, more than one version of the program is evolved with different runs of the co-evolution, and these versions are then used together in an ensemble. The details of each component of the framework and their interactions will be discussed throughout the paper.

The use of the formal specifications to automatically create fitness functions makes it possible to apply the framework to any problem that can be defined with a formal specification (as long as an oracle can be automatically derived). However, evolving correct programs for solving, for example, the *halting problem* [15] is not possible.

The main contribution of this paper is a novel approach to Automatic Programming, in particular the automatic refinement of formal specifications. This approach has a wider range of applications than previous techniques reported in literature, because it makes no particular assumption on the gap between the formal specification and the target implementation.

The paper is organised as follows. Section 2 reviews related work. Section 3 gives short background information on software testing, genetic programming and co-evolution. A description of how programs can evolve to satisfy a formal specification follows in Section 4, whereas how to optimise the training set is explained in Section 5. Section 6 briefly describes what N-version Programming is and how it can be exploited to improve the performance of our framework. A discussion on how complex software might be evolved follows in Section 7. The case study used for validating our novel framework is presented in Section 8. Finally, Section 9 concludes the paper.

2. Related work

The problem we address in this paper, and the way we try to solve it, is similar to *Evolvable Hardware* [22]. The design of electronic circuits is an important and expensive task in industry. Hence, search based techniques have been used to tackle this problem in which, given a specification of a function (for hardware that can be for example a truth table), a solution that implements the function is sought.

Our framework also shares some similarities with the system for evolving secure protocols from a formal specification [13]. A protocol is a sequence of messages, and a genetic algorithm was used to evolve it. The fitness function depends on how well the protocol satisfies its specification. Although neither genetic programming nor co-evolution was used, our work shares the same idea of formally specifying the expected behaviour (what it should be done, but without stating how to do it) of a system (e.g., programs and protocols), and then using natural computation techniques to find a solution that implements that system.

Another related field of research is *Software Cloning*, in which evolutionary techniques have been used for automatically cloning programs by considering only their external behaviours [42]. The external behaviour of a program can be partially represented by a set of unit tests. In other words, programs are evolved till they are able to pass all the used unit tests. It has been argued that this technique might be helpful for Complexity Evaluation, Software Mutants Generation, Test Fault Tolerance and Software Development in Extreme Programming [42]. Besides the different goals, the main differences from our approach are that no formal specification was employed and the test case sets were fixed (i.e., they did not co-evolve with the programs).

The work of Adamopoulos et al. on Mutation Testing [2] has some similarities to our framework. Its goal is to find test cases that can recognise faulty *mutants* of the tested software, because such a test suite would be good for asserting the reliability of the software. Mutants (generated with a precise set of rules) co-evolve with the test cases, and they are rewarded on how many tests they pass, whereas the test cases are rewarded on how many mutants they identify as semantically different from the original program.

In our previous work, we used genetic programming and co-evolution to also address other software engineering problems. We investigated the task of automatically repairing faulty software [4,5,8] and automatically improving non-functional properties [6].

3. Background

3.1. Software testing

Software testing is used to find the presence of faults in computer programs [36]. Even if no fault is found, testing cannot guarantee that the software is fault-free. However, testing can be used to increase our confidence in the software reliability.

Different types of testing approaches exist. For example, in black box testing [36] the tester has no access to the internal structure of the tested software. Only a specification that describes its functionality is provided. The aim of the tester is to find an input that satisfies the pre-condition of the program specification but fails the post-condition. In that case, it would mean that the program has a fault. In white box testing [36] the tester has access to the source code of the function/class under test. A formal specification is not needed, although it can be useful to automate the phase in which the results are

checked against an oracle. Given a coverage criterion (e.g., branch coverage), the aim is to find a test set that has as high a coverage as possible. Maximising the coverage of a test suite is important because faults could be present in portions of the software that are rarely executed.

Automating the testing phase is an important goal that is keenly sought. Search based techniques have been applied to tackle this task with promising results [33].

3.2. Genetic programming

Genetic programming (GP) [16,29] is a paradigm to evolve programs. Although it was first introduced by Cramer [16], only since Koza [29] GP has been widely known with many successful applications. A genetic program can often be seen as a tree, in which each node is a function whose inputs are the children of that node. A population of programs is held at each generation, where individuals are chosen to fill the next population accordingly to a problem specific fitness function. Crossover and mutation operators are applied on the programs to generate new offspring.

The basic components of a genetic program are called *primitives* [12]. The choice of primitives is very important for the correct evolution of the desired program. Often, the choice of the primitives is optimised for the particular task that is addressed. A set of primitives can represent for example either a complete programming language or, in the case for example of controlling a robot, it can represent operations such as "move forward" and "turn left".

GP has been principally applied to solve real-world learning problems.

3.3. Co-evolution

In co-evolutionary algorithms, one or more populations co-evolve influencing each other. There are two types of influences: *cooperative co-evolution* in which the populations work together to accomplish the same task, and *competitive co-evolution* as predators and preys in nature. In this paper we use competitive co-evolution.

Co-evolutionary algorithms are affected by the *Red Queen* effect [41], because the fitness value of an individual depends on the interactions with other individuals. Because other individuals evolve as well, the fitness function is not static. For example, exactly the same individual can obtain different fitness values in different generations. One consequence is that it is difficult to keep trace of whether a population is actually "improving" or not [14,34].

One of the first applications of competitive co-evolutionary algorithms was the work of Hillis on generating sorting networks [23]. He modelled the task as an optimisation problem, in which the goal is to find a correct sorting network that does as few comparisons of the elements as possible. He used evolutionary techniques to search the space of sorting networks, where the fitness function was based on a finite set of tests (i.e., sequences of elements to sort): the more tests a network was able to correctly pass, the higher fitness value it got. For the first time, Hillis investigated the idea of co-evolving such tests with the networks. The reason for doing so was that random test cases might be too easy, and the networks can learn how to sort a particular set of elements without being able of generalising. The experiments of Hillis showed that shorter networks were found when co-evolution was used.

Ronge and Nordahl used co-evolution of genetic programs and test cases to evolve controllers for a simple "robot-like" simulated vehicle [44]. Similar work has been done by Ashlock et al. [9]. In that work, the test cases are instances of the environment in which the robot moves.

Note that the application area of such work was restricted (i.e., sorting networks and robot controllers), whereas we use co-evolution for a more general and complex task (i.e., automatic refinement).

4. Evolution of the programs

4.1. Basic concepts

Given the specification of a program P, the goal is to evolve a program that satisfies it (i.e., we want to evolve a program that is semantically equivalent to P). To achieve this result, GP is employed. At each step of GP, the fitness of each program is evaluated on a finite set T of unit tests that depends on the specification. The more unit tests a program is able to pass, the better the fitness value it will get rewarded. This set T should be relatively small, otherwise the computational cost of the fitness evaluation would be too high. In fact, for calculating the fitness of a program we need to run it on each unit test in T. What can be defined as "small" or as "big" depends on the available computational resources.

4.2. Training set

The set T is different from a normal training set in machine learning. Let X be the set of all possible inputs for P, and Y be the set of all possible outputs. A program might take as input more than one variable, hence X is a set of all those possible combinations. The element $x \in X$ is a vector of the input variables for P. By $g(x) = g(x[0], \ldots, x[n])$ we mean the execution of the program g with input vector x. The elements in x can be of different types (e.g., integer, double and pointer). If P has an

internal state, then we should generalise x in a way in which how to put the state in the right configuration (e.g., by previous function invocations) is considered.

The output elements $y \in Y$ can be composed of a single value, or of a vector of values with potentially different types. Each of these values would represent a different *assert* statement in the unit test. For example, in checking whether an array is correctly sorted, there could be an assert statement to check whether the length of the array has been changed. There could be an assert statement for each element of the array to see whether they are equal or not to the elements of the expected sorted array. Fig. 2 shows a simple Java-like example of a unit test that is mapped into input x and output y. Note that in many languages the length of an array cannot be changed, and we check the length instead of another property only for simplicity.

Given any input $x \in X$, we do not have the expected value $y^* = g^*(x)$, with g^* being the optimal program that we want to evolve and $y^* \in Y$ being the expected result for the input x. Hence, in our case a unit test $t \in T$ instead of being seen as a pair (x, y^*) (as a typical element of a training set would be), is a pair (x, c), where c is a function $c(x, y) : X, Y \rightarrow \Re$. The function c gives as output a value of 0 if y is equal to y^*, otherwise a real positive value that expresses how different the two results are. A higher value means a bigger difference between the two results. Because the function c is the same for each $t \in T$, we can simplify the notation by considering only the input x for a unit test. In other words, $t \in X$ and $T \subseteq X$. A program $g \in G$, where G is the set of all possible programs, is said to pass a test $t \in T$ iff $c(t, g(t)) = 0$. How to automatically derive the function c will be explained later in this section.

The scenario described in this paper is very different from the normal applications of GP:

- The training set T can be automatically generated with any cardinality. There is no need for any external entity that, for a given set of x, says which should be the corresponding y^*.
- Usually, because there are only a limited number of pairs (x, y^*), all of them are used for training. In our case we have the additional problem of choosing a subset T, because using the entire X is generally not feasible.
- The training set T does not contain any noise.
- We are not looking for a program that on average performs well, but we want a program that always gives the expected results, e.g. $\forall x \in X \bullet g(x) = g^*(x)$. Hence, a program does not need to worry about over-fitting the training set. Even if only one test in T is failed, that means that the specification is not satisfied.
- To prevent GP from learning some wrong patterns in the data, it would be better not to have a fixed T. In other words, it would be better to have different test cases at each generation. Fig. 3 shows a non-trivial example in which an undesired pattern is learnt: it represents a training set for the Triangle Classification problem [36] and an incorrect simple program that is able to pass all these test cases.

4.3. Heuristic based on the specification

The function $c(x, y) : X, Y \rightarrow \Re$ is an heuristic that calculates how far the result $y = g(t)$ is from satisfying the post-condition of the specification for the input $x = t$ when the pre-condition is satisfied. If the pre-condition is not satisfied, c should return 0. Note that c does not calculate the distance between $x = t$ and $y = g(t)$. It calculates the distance from $y = g(t)$ to the expected result $y^* = g^*(t)$.

```
int[] a = new int[]{7,9,1};
sort(a);
assert(a.length==3);
assert(a[0]==1);
assert(a[1]==7);
assert(a[2]==9);
/*
    x =  {7,9,1}
    y =  {3,1,7,9}
*/
```

Fig. 2. Example of a unit test mapped into a pair (x,y).

```
<(5,-2,3), 0> // 0 represents 'not triangle'
<(4,3,6) , 1> // 1 represents 'scalene'
<(9,9,16), 2> // 2 represents 'isosceles'
<(3,3,3) , 3> // 3 represents 'equilateral'

function classifyTriangle(a, b, c)
   return  a + b - c;
```

Fig. 3. An example where an undesired pattern is learnt.

It is important to note that the function c is not required to be able to compute the expected result y^* for an input x. Being able to state whether a particular result k is correct for an input x (i.e., $k = y^*$) is enough. This can be done without knowing the value of y^*. We introduce here an example to clarify this concept. Assume that P is a sorting algorithm that takes as input an array of integers and sorts it. Given as input an arbitrary array $x = (4, 3, 2, 1)$, if the output of g is $y = (1, 4, 2, 3)$, we do not need to know $g^*(x)$ to conclude that $g(x) \neq g^*(x)$, because y is not sorted. We can conclude that an array is not sorted by looking at the specification of the sorting algorithm. In this particular case, we have a 4 before a 2, which is enough to conclude that the array is not sorted. Although a specification can state whether an array is sorted or not, it cannot say how to sort it.

Because the expected results y^* are not known even though a formal specification is employed, the function c cannot calculate any geometric distance between y and y^* in the space Y. However, because it heuristically states how far y is from satisfying the specification, a 0 as a result means that $y = y^*$. Furthermore, if $y \neq y^*$, then c necessarily returns a value strictly bigger than 0. Therefore, c indirectly calculates a particular type of distance between y and y^*, although y^* is not known.

Such a function is inspired by the one employed in Tracey's work on black box testing [50]. In that work, the goal is to find a test case that breaks the specification. This happens in the case of pre-condition Pr that is satisfied but the post-condition Po is not. In searching for such a test case, the pre-condition of the function is conjugated with the negated post-condition, i.e. $Q = Pr \wedge \neg Po$. A test case that satisfies Q does break the specification. The distance function c is hence applied on that predicate Q for guiding the search. A distance value 0 is given if Q is satisfied, and this means that a fault that breaks the specification is present for the input t.

The main difference with our work is on what c is applied to. In this paper we seek a program that is correct. For guiding the search, we use this distance directly on the post-condition when the pre-condition is true (i.e, $\neg Pr \vee Po$). A distance value 0 means that the post-condition is satisfied, so the program is correct for that particular input t. Programs closer to be correct are rewarded with a lower fitness value. On the other hand, in Tracey's work, the fitness is calculated on $Pr \wedge \neg Po = \neg(\neg Pr \vee Po)$ to reward test cases that expose faults in the tested software.

To calculate $c(t, g(t))$ the program g is executed with the input data contained in the unit test t, and then, if the pre-condition is satisfied, the result $y = g(t)$ is compared against the post-condition of the function to see whether the result is correct or not for that particular input $x = t$.

How to calculate the function c? Table 1 is inspired by Tracey's work [50] and shows how to calculate a support function d_θ, which is used to calculate how far a particular predicate θ is from being evaluated as true. In Table 1, K can be any arbitrary positive constant value. A and B can be any arbitrary expression, whereas a and b are the actual values of these expressions based on the values in the input set I. W can be any arbitrary expression.

The function d_θ takes as input a set of values I, and it evaluates the expressions in the predicate θ based on the actual values in I. The function c is recursively built on d_θ, in which θ is the disjunction of the negated pre-condition with the post-condition of the employed specification (i.e, $\neg Pr \vee Po$). In other words, c is equivalent to $d_{\neg Pr \vee Po}$. Of course, for different specifications there will be built different functions c. The set I is hence composed of the constants in the specification and of variables that depend on the input of c (i.e., x and y).

This function d_θ works fine for expressions involving numbers (e.g., integer, float and double) and boolean values. For other types of expressions, such as an equality comparison of pointers/objects, we need to define the semantics of the sub-

Table 1
Example of how to apply the function d_θ on some types of predicates.

Predicate θ	Function $d_\theta(I)$
A	if a is TRUE then 0 else K
$A = B$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$A \neq B$	if $abs(a - b) \neq 0$ then 0 else K
$A < B$	if $a - b < 0$ then 0 else $(a - b) + K$
$A \leqslant B$	if $a - b \leqslant 0$ then 0 else $(a - b) + K$
$A > B$	$d_{B<A}(I)$
$A \geqslant B$	$d_{B\leqslant A}(I)$
$\neg A$	negation is moved inward and propagated over A
$A \wedge B$	$d_A(I) + d_B(I)$
$A \vee B$	$min(d_A(I), d_B(I))$
$A \Rightarrow B$	$min(d_{\neg A}(I), d_B(I))$
$A \Longleftrightarrow B$	$min((d_A(I) + d_B(I)), (d_{\neg A}(I) + d_{\neg B}(I)))$
A xor B	$min((d_A(I) + d_{\neg B}(I))), (d_{\neg A}(I) + d_B(I))$
$\forall x \in X \bullet W$	if X is empty then 0 else $\sum_{v \in X} d_W(I[v/x])$
$\exists x \in X \bullet W$	if X is empty then K else $min(d_W(I[v/x]))$ where $v \in X$

traction operator $-$. For example, it can return 1 if the two values are different and 0 otherwise. This type of distance d_θ is similar to the *branch distance* used in white box testing [33].

Unfortunately, the predicates involving $\forall$ and $\exists$ can be computationally expensive if the set of values on which they depend on is large. In our case study, these sets were arrays of relatively small length, hence the computational cost was not so high. However, the case of expressions such as $\forall x \in \Re \bullet W$ (where W is a predicate) cannot be handled in this way. We will address this problem in the future. One way could be to calculate the predicate on a restricted set of values, and then evolve it at regular intervals for finding at least a value for which the predicate is false (if such a value exists). Unfortunately, a restricted set could lead to the situation in which a predicate is evaluated as true when it is actually false.

4.4. Fitness function for the programs

At each generation i of the evolution, the current population of genetic programs G_i is executed on the current test set T_i. The fitness of each $g \in G_i$ is based on its ability to pass the unit tests in T_i. It is a minimisation problem, in which the heuristic function c is to be minimised over all the test cases in T_i. If the specification is satisfied, the contribution of c to the fitness function is equal to 0.

In evolved programs, it is easy to have code that generates exceptions, such as divisions by zero or accessing arrays out of their bounds. In such cases, no exception is thrown by our framework. If the GP node that generated that error is supposed to return a numerical value, then it just returns a 0 (and hence it operates as a protected operation). However, the framework is informed of each exception, and their number $E(g, T_i)$ is used in the fitness function with the aim of penalising programs that do operations that should be forbidden. The function $E(g, T_i)$ is calculated by counting each generated error when the program is applied on each test case in T_i.

In GP, there is the problem of *bloat* [30], i.e., the increasing size of the programs with redundant or non useful code. Parsimony controls [32] might be used to deal with this problem (e.g., including in the fitness function a penalty term based on the size of the program). In our particular case, parsimony can be deceptive. g^* may be lost if parsimony is imposed. For example, if a g_z is not optimal, smaller than g^* and able to pass all the tests in the current population of tests, then g_z will be preferred to g^* although it is not optimal. It is noteworthy that such a behaviour would not happen in a conventional co-evolutionary algorithm, in which the fitness value of an individual is solely based on interactions with other individuals. Nevertheless, we still have to employ a parsimony control in order to achieve computational efficiency. We penalise the number of nodes $N(g)$ in the program.

The employed fitness function f for each program g is:

$$f(g) = \frac{N(g)}{N(g)+1} + \frac{E(g,T_i)}{E(g,T_i)+1} + \sum_{t \in T_i} c(t, g(t)). \tag{1}$$

5. Optimisation of the training set

Choosing a good training set T is not easy. An ideal set T^* would be one that, if a program $g \in G$ passes all the unit tests $t \in T^*$, then it is guaranteed that it will pass all the tests in X. This means that if g completely fits the data in the training set, then it is guaranteed that g is correct. A trivial set that satisfies such a constraint is $T = X$. However, the set should be relatively small, otherwise the computational cost of fitness evaluation would be too high.

Finding the optimal $T^* \neq X$ is impossible, because in the set G of all possible programs there is always at least one program that fits all the data in this hypothetical T^*, and can have, for example, a special "if" statement based on the value of a $t \in X \setminus T^*$ that makes the program fail on that test t. Although the use of a limited set of primitives (e.g., without "if") might not lead to this problem, this is not true in the general case. Even if an optimal T^* existed, there is no guarantee that GP will be able to evolve a program that fits all the data in T^*. A good strategy for choosing an appropriate training set T is needed, because the performance of GP depends on it. In the following, we describe the use of *co-evolution* to find T.

Because an element t of a training set can be automatically generated, we can use a different training set T_i at each different generation i of GP, and that would also help to prevent misleading cases such as the one in Fig. 3. However, care needs to be taken for assuring that the overhead of generating new unit tests is not too high.

To simplify the following discussions, we assume that GP maintains a population with only one program g_i during generation i. A unit test on which g_i fails gives more information to assist the evolution of g_i than a passed unit test. In fact, if a test t is failed, then the program g is evolved until it passes t (or until GP has been stopped). In the case that all the unit tests are passed, g_i is not able to evolve any further. If $T \neq T^*$, the program is not necessarily correct and there is no more guidance to its evolution. Hence, the idea is to let the test set T_i *co-evolve* with the program g_i, with the aim of giving to g_i at each generation a set of tests that it is not able to pass yet. Let N_i be the set of all the possible unit tests at generation i for which g_i passes, and F_i be the set of tests for which it fails. We have $N_i \subseteq X, F_i \subseteq X$ and $N_i \cup F_i = X$. A strategy ρ is used to choose the next T_{i+1} from T_i and g_i. A reasonable strategy would be one that generates new unit tests that the current g_i will fail. Given:

$$v(i,t) = \begin{cases} 0 & \text{if } t \in N_{i-1}, \\ 1 & \text{if } t \in F_{i-1}, \end{cases}$$

we can formulate the choice of the strategy as an optimisation problem, in which the fitness to maximise is:

$$f(\rho) = \sum_{i=0}^{i=H} \sum_{t \in U_i} v(i,t), \tag{2}$$

where H is the number of allowed generations for GP, and $U_i \subseteq T_i$ is the largest subset of T_i in which all the elements are unique. For the sake of clarity, the cardinality of the training sets T_i does not change during the search, and there can be redundant unit tests in a T_i (although this latter case should be avoided if possible because it would not be of any help). We employ the *co-evolution* of programs and their unit tests to solve this optimisation problem. However, it is important to remember that the goal is to evolve a program g^*, and finding the best sequence of T_i is a secondary task that is addressed because it will help to solve the principal problem.

Even if the best strategy ρ^* is used, none of the T_i will be the optimal T^* (that usually does not exist). Using ρ^* does not imply that g^* will be evolved, and g^* might be evolved even with a sub-optimal strategy, because GP is a randomised algorithm. It can even happen that ρ^* actually makes the evolution of g^* more difficult. An example will help to clarify this statement: consider a boolean function that takes as input one integer and says whether it is positive or not. The program g^t always returns true, whereas g^f always returns false. For simplicity we consider a training set of size 2. If all elements in T_i are positive, GP might evolve g^t (i.e., $g_i = g^t$), and that completely fits the data. At the next generation, the set T_{i+1} in which all the elements are negative values makes g_i fail all the tests, hence it optimises function (2). Then, GP might evolve g^f (i.e., $g_{i+1} = g^f$), and that, again, completely fits the data. The system can go on in this way, switching the training set from all positive to all negative, with GP that always evolves either g^t or g^f. This strategy is optimal regarding Function (2), but it is obviously not a good strategy. A strategy that at each generation would generate T_i with both positive and negative values is likely to give much better results, because it would not fall in that cyclic behaviour (although other types of cycles may potentially arise).

The choice of a strategy depends on the particular *solution concept* [20] that we want to optimise. However, usually we do not know the underlying objectives of a problem. Moreover, although a g_i can pass all the unit tests in T_i, usually there is no guarantee that any g_j with $j > i$ will be able to do the same. Hence, Function (2) captures some desired properties that we want from a strategy, but it is not sufficient.

This problem of continuous changing of the fitness landscape and memory loss is named *mediocre stable states* [21]. Archive methods (e.g., [45]) have been proposed to handle it, with a lot of effort spent on designing algorithms that guarantee a monotonic improvement of the results [27]. When archives are employed, at each generation some individuals are stored in a separated archive (which often is implemented as a queue of finite size). The fitness of current generation individuals is also based on the interaction with those old individuals. It is important to note that these archives are conceptually similar to *regression test* [36] suites.

When co-evolution is employed, each $t \in T_i$ has a fitness value as well. A fitness function is used to reward unit tests that the programs in the current population G_i have difficulties in passing. In contrast to Eq. (1), this is a maximisation problem. A value of 0 means that all the programs in G_i pass the test t. The fitness function is:

$$f(t) = \sum_{g \in G_i} c(t, g(t)). \tag{3}$$

It is important to outline that the employed $c(t, g(t))$ values are the same as those used for Eq. (1). Hence, they are calculated only once, and then they are used for both fitness functions. Fig. 4 shows the relations between the programs and the unit tests with respect to how their fitness values are calculated. G is the population of programs, whereas T is the population of unit tests. For simplicity, sets of cardinality 3 are displayed. In Fig. 4a it is shown on which unit tests the fitness of the first program g_0 is calculated. On the other hand, Fig. 4b shows on which programs the fitness for the first unit test t_0 is calculated. Note that the arc between the first program and the first unit test is used in both fitness calculations. Finally, Fig. 4c presents all the possible $|G| \cdot |T|$ connections.

Once the fitness function is calculated for each test in T_i, we use search algorithms to sample the new test cases [33]. In particular, we employ genetic algorithms [24] to evolve the next test set T_{i+1}.

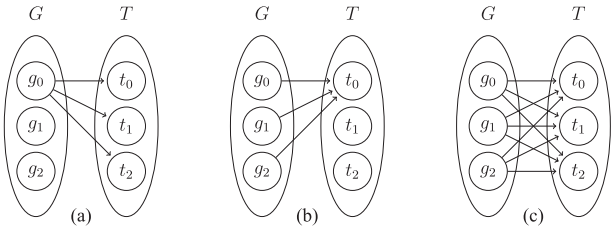

Fig. 4. Relations between the programs and the unit tests regarding their fitness values.

(1) Evolve the GP population for one generation.
(2) Calculate the fitness value for each test case $t \in T_i$ based on the GP programs.
(3) Based on these fitness values, use a search algorithm to generate new test cases for T_{i+1}.
(4) If current generation i is chosen for intensive evolution, then:
  (a) Calculate the fitness value for each test case $t \in T_{i+1}$ based only on the best (i.e., highest fitness value) GP program in the current GP population.
  (b) Use a search algorithm to generate new test cases and replace the old ones in T_{i+1}.
  (c) If the termination criterion for the intensive evolution is not satisfied, go back to Step $4 \triangleright a$
(5) If the termination criterion is not satisfied, go back to Step 1.

Fig. 5. Pseudo-code of the algorithm used to choose test cases at each GP generation.

However, we cannot expect to have good (in the sense of being able of finding faults) unit tests already in the first generations of a genetic algorithm. In that case, trying to evolve programs on simple unit tests for many generations would not be very appropriate. Moreover, when the programs are able to pass all the unit tests in the current generation, time would be needed to evolve more challenging test cases.

One solution is to allow more generations for the evolution of the unit tests. For example, every k generations of the co-evolution, a special intensive evolution of the unit tests can be done. During such intensive evolution phases, no program evolves (and that helps to reduce the computational overhead of the intensive evolution). Moreover, for fitness evaluation of the unit tests, we can just use the best current program (and that helps as well in reducing the overhead). In fact, because we do not need to evolve all programs, there is no need to execute all of them on the unit test population (that would have been done if we needed to calculate their fitness values), because the best program is already an appropriate candidate for evaluating the quality of the unit tests. Fig. 5 shows the pseudo-code of the algorithm used to choose test cases at each GP generation.

5.1. Specialised sub-populations

In this section we describe a co-evolutionary algorithm that is built on Tracey's work on black box testing [49]. We call this algorithm *Specialised Sub-Populations* (SSP). The aims of this algorithm are to provide more challenging test cases and to increase diversity among them.

The pre-condition of the function is conjugated with the negated post-condition. Then, this predicate is transformed into a disjunctive normal form. For each disjunction element, an independent sub-population $S_{j,i}$ in T_i is used, where $\bigcup S_{j,i} \subseteq T_i = T_i$. In other words, the test set T_i is partitioned into non-overlapping sets $S_{j,i}$. Each subset uses a different heuristic function for evaluating its test cases, and the resulted fitness function to minimise depends on the disjunction element related to the subset. For example, if the pre-condition Pr of the program is true and the post-condition Po is $A \wedge B$, then it will be $Pr \wedge \neg Po = \neg A \vee \neg B$. Hence, T_i will be divided into two independent sub-populations, one that exploits $d_{\neg A}(I)$ for its fitness function, and the other that uses $d_{\neg B}(I)$.

The original idea [49] was to evolve a different unit test for each possible reason for which the program can break the specification. It is a minimisation problem, where a value of 0 is the optimum and it means that the considered program failed that particular test. However, this is insufficient for our problem. We want not only to evolve a test population that the current population of programs is not able to pass, but we also want it as hard as possible. Hence, we need to exploit the function d_θ to reward the unit tests that make the post-condition as far as possible from being evaluated as true. Thus, we have to transform the minimisation problem (that does not give any more information once a test is failed) to a maximisation one.

Given W a component of the disjunction we want to satisfy, the goal is to get a test case t that maximises:

$$h_W(t, g(t)) = \begin{cases} k + d_{\neg W}(I) & \text{if } W \text{ is true,} \\ \frac{k}{1+d_W(I)} & \text{otherwise,} \end{cases} \tag{4}$$

where k is any arbitrary constant (e.g., $k = 1$), and the set value I is based on the variables in t, $g(t)$ and the constants in W. The function $h_W(t, g(t))$ is never less than 0, and, for each different sub-population, it can replace $c(t, g(t))$ in Eq. (3).

The use of $h_W(t, g(t))$ also solves another important problem. If a test is passed by all the programs, by using only c we will get a fitness value of 0, that gives no gradient information until at least one program that fails the test case is evolved. On the other hand, h could give gradient information even when a test is passed.

The next sub-populations for T_{i+1} are still generated with a search algorithm (i.e., genetic algorithms). However, there is no communication among the sub-populations.

6. N-version programming

After applying our framework, we get as output a program that hopefully satisfies the wanted formal specification. However, testing cannot prove the correctness of a program, hence our output program might contain faults. Therefore, in this section we discuss how we can further improve the reliability of the programs.

Fault tolerance is the ability of software to be able to behave correctly even in the cases in which faults occur. One technique to achieve this goal is *N-version Programming* [10]. Briefly, the idea is to implement the same software as several independent versions (at least 2), and then use all of them in parallel for the computation. The result will be chosen by a simple voting criterion among the different versions.

If the software versions are independent (e.g, implemented by different software developers without any communication between them), it is possible that an error in one version would not appear in the other versions. However, true independence can be hard to achieve in practise. There might be other ideas in hardware fault tolerance [46] that we could borrow in our future work.

One problem with N-version programming is that it is expensive. The same software needs to be implemented many times in independent ways. Hence, work on how to automate N-version programming with GP has been proposed [19].

It is useful to note that N-version programming is conceptually the same idea as *ensembles* in machine learning. An example of GP ensemble is the work of Imamura et al. [25]. N-version programming can be used in our framework to improve its performance. For the same software specification, different runs of the framework can be done (e.g., with different random seeds and/or different parameter setting). Hence, the final output programs of each run can be put together in an ensemble, although different versions in this case would not be independent from each other.

Let σ be a system that can generate a correct program p^* from a formal specification with probability $C(\sigma) = \delta$. The program p^* will always give the right answer for all the inputs with probability δ. The programs generated with probability $1 - \delta$ give a wrong output at least for one input (but no assumption on the input is made, e.g., whether for each different incorrect program these inputs are either dependent or not). Let E^n_σ be an ensemble of n programs generated with σ. It follows that:

$$C(\sigma) > 0.5 \Rightarrow \lim_{n\to\infty} C(E^n_\sigma) = 1, \tag{5}$$

assuming that these n programs are independent of each other.

If $\delta \leqslant 0.5$, then it is still possible that the probability of having a correct ensemble increases. What happens in these other cases is directly dependent on whether the faults in the programs generated by σ are independent or not. If they are independent, the faults can be covered. In fact, even if with a low $C(\sigma)$, a high $C(E^n_\sigma)$ might still be obtained.

7. Evolving complex software

The framework described so far can be used to evolve single functions. However, unless software is developed in a monolithic way, there are several functions that interact and depend on each other. In this section we describe a possible way to enable our framework to deal with these cases. Although software architectures can be very complex, we describe a simple approach here to illustrate how our conceptual framework can cope with complex software.

Let us call Z the set of function specifications of the software we want to automatically implement. We could use our framework directly on each $z \in Z$, but that would not take into account the dependencies among the functions. For example, let's say that a method A needs to call a method B. If we first evolve B, then we can use it when we try to evolve A. However, if we try to evolve directly A that would be more difficult, because for each position in which B needs to be called inside A we need to evolve a separate copy of B. Moreover, B can be seen as a sub-problem of A, and the specification of B is very useful for the fitness function of A. If we do not exploit this specification, the fitness function generated by A might give no direct indication on how to solve this sub-problem.

We can use the following simple strategy to tackle this problem:

(1) Choose a function specification z from Z.
(2) Use our framework to evolve an implementation of z. If that is successful, then remove z from Z, and add its implementation to the set of used primitives. Otherwise, keep z in Z to try to evolve it again later.
(3) If $Z \neq \{\}$, then go to Step 1, otherwise the algorithm terminates.

The choice of z from Z might be guided from architectural information of the system we want to implement. For example, simple functions would be preferred to functions that depend on other functions. Hence, we would start to consider these more complex functions only when the functions they rely on have already been evolved by the system. The component of the framework that is responsible for doing this choice is called *Function Specification Manager* (see Fig. 1).

If a function is "considered" correctly evolved, it will be added to the set of GP primitives. Hence, the other functions that will be evolved afterwards will be able to use it.

If the framework is not able to evolve a particular function, it will try do it again later on (i.e., it is not removed from Z). In fact, that function might need some other functions that have not been evolved yet. One possible implementation of Z could be a queue that has been ordered by exploiting the function dependencies. Therefore, if a specification z is not correctly implemented, then it will be pushed back at the end of the queue.

Another optimisation could be that, instead of adding an evolved function to the general set of GP primitives, only the functions that could need that evolved function would have it added to their set of used primitives. However, this type of dependence is not always known.

Once all the functions have been evolved, the final complete implementation of the software is given as output of our framework (see Fig. 1).

8. Case study

In this section a set of GP primitives that defines a simple programming language is presented. Then, seven different problems that use this set are described: *MaxValue*, *AllEqual*, *TriangleClassification*, *Swap*, *Order*, *Sorting* and *Median*. Different types of experiments are discussed and their results are shown. Finally, the results are discussed.

8.1. Primitives

In the following, we give details of a set of primitives that defines a non-trivial programming language. It is important to note that the set was chosen without any particular bias towards the programs in our case study. The set used in this paper is more general and complex than the one we used in our preliminary work [7]. This is the reason why the results of the experiments are different. Extending our prototype to support an entire real-world language (e.g., C and Java) is an interesting and important goal that we are currently pursuing.

Each node in our GP engine has a type, and constraints exist on which types of children a node can have. Our language considers three types: *statement*, *integer* and *boolean*. The root node of a tree should be of statement type. To guarantee that no evolutionary operator breaks any constraint, we use Strongly Typed Genetic Programming [35].

The name of the primitives is consistent with their semantics: x and y represent sub-trees of integer type; a and b are boolean type sub-trees; and w and z are statement sub-trees. Based on the context, we use the same symbol for representing either a sub-tree or its output. The primitives are:

Constants: Five integer constants with value from 0 to 4. Two boolean constants: true and false.
Arithmetic functions: (add x y), (sub x y), (mul x y) and (div x y).
Boolean functions: (bigger x y), (bigger_or_equal x y), (equal x y), (and a b), (or a b) and (neg a).
Base statements: (skip) is the empty statement, that does nothing when executed. The concatenation of statement executions is done with (seq w z), whereas conditional statements are done with (if a w z). In that case, w is executed only if a is true, otherwise z will be executed. Finally, for loops we use (loop x y w), in which w is executed $y - x$ times, and index is an integer terminal (i.e., a leaf node) that gives either the value of the iterator variable of the closest loop or zero if it is used outside a loop.
Variable: For simplicity, only one variable called *result* is supported in the language, with a statement (write_result x) and an integer terminal read_result to manipulate it. If the program that is tried to be evolved should return an integer value, then the value inside *result* at the end of the computation will be given as output. Otherwise, *result* can be just used as a variable for temporally storing computed data.
Integer inputs: For any integer variable as input to the program, a terminal input_i is included to the set of primitives, where i is a constant that is different for each input variable.
Array input: If the program takes as input an array, then the primitives to handle it will be added to the used set. We use (array x) to get the value in position x, whereas (write_array x y) writes y in position x. Finally, the terminal length returns the length of the array. Note that only one array as input is supported.

8.2. Programs to evolve

To evaluate our novel framework, seven different problems have been chosen. Some of them take as input an array A of integers. The array is passed by reference. The length of A is represented by l. The state of the array A after a function is executed is represented by A'. Comparisons between arrays (e.g., $A' = A$) are not made on the pointers, but on the status of those arrays (i.e., the internal elements).

Each program has a global variable named *result*. All the problems described in the following use the same set of primitives that were described in the previous section, with some differences based on the type of input. For each problem a specification is presented. Instead of using a specific language such as Z [48] or JML [31], we prefer to use a simple first order logic specification for the short examples that we present. In fact, our conceptual framework could be applied to any formal specification. Hence, the choice of a specification language is not essential. We believe that a first order logic could be more easily understood by readers not familiar with real-world specification languages (e.g., Z [48]). Nevertheless, although the following specifications could be re-written in another language, the derived fitness functions would be the same. Therefore, the final results would not change.

Extending our framework to support a specific language would just require to implement the tool to derive the fitness function. A good fitness function would be able to provide gradient even in the cases in which the specification is not satisfied. Possible problems related only to a particular language are not addressed in this paper.

In the following, *Pr* and *Po* are the abbreviations for pre-condition and post-condition respectively.

```
public static int getDistanceForMaxValue(int[] a, int[] a1, int r,  int K)
{
  int sum = 0;
  int min = Integer.MAX_VALUE;

  for(int i=0; i<a.length; i++)
    if(r < a[i])
      sum += (a[i]-r)+K;

  for(int i=0; i<a.length; i++)
  {
    int tmp = 0;

    if(r != a[i])
    tmp = Math.abs(r-a[i])+K;

    if(tmp<min)
      min = tmp;
  }

  sum += min;

  for(int i=0; i<a.length; i++)
    if(a[i] != a1[i])
      sum += Math.abs(a[i]-a1[i]) + K;

  return sum;
}
```

Fig. 6. Example of fitness function for *MaxValue*.

The first program we consider is to find the maximum value inside A. Such a value has to be stored in *result*. The specification of this problem *MaxValue* is:

$$\begin{aligned} &int\ MaxValue(int[]\ A)\\ &Pr : l \geqslant 1\\ &Po : \forall i \in [0, l-1] \bullet result \geqslant A[i] \wedge\\ &\qquad \exists i \in [0, l-1] \bullet result = A[i] \ \wedge A' = A \end{aligned} \tag{6}$$

In Fig. 6 there is an example of a distance function that can be automatically derived from this specification of *MaxValue*. That distance function can be automatically derived based on the rules in Table 1. Once a test case is executed, the array a represents the state of the input array before executing *MaxValue*, whereas $a1$ represents its state after the execution. The variable r is the result we obtain from executing *MaxValue*, whereas K is a constant as defined in Table 1.

In the *AllEqual* problem, given A as input, we want to evolve a program that is able to say whether all the elements in A are equal to each other or not. If they are not all equal, the variable *result* should store the value 0. Otherwise, it should store any value different from 0. The specification is:

$$\begin{aligned} &int\ AllEqual(int[]\ A)\\ &Pr : l \geqslant 1\\ &Po : ((result = 0 \ \wedge \ \exists i \in [0, l-2] \bullet A[i] \neq A[i+1]) \vee\\ &\qquad (result \neq 0 \ \wedge \ \forall i \in [0, l-2] \bullet A[i] = A[i+1])) \ \wedge A' = A \end{aligned} \tag{7}$$

A very famous program in software testing is *TriangleClassification* [36]. Given three integers as input, the program should classify whether they represent the edges of an incorrect triangle (in that case the program should return for example a 1), a scalene triangle (return value 2), an isosceles one (return value 3) or an equilateral triangle (return value 4). The specification is:

$$\begin{aligned} &int\ TriangleClassification(int\ a,\ int\ b,\ int\ c)\\ &Pr : true\\ &Po : NT \ \vee \ (V \ \wedge \ SC) \ \vee \ (V \ \wedge \ IS) \ \vee \ (V \ \wedge \ EQ)\\ &NT : (a \geqslant b + c \ \vee \ b \geqslant a + c \ \vee \ c \geqslant a + b) \wedge result = 1\\ &V : a < b + c \ \wedge \ b < a + c \ \wedge \ c < a + b\\ &SC : a \neq b \ \wedge \ b \neq c \ \wedge \ a \neq c \ \wedge \ result = 2\\ &IS : ((a = b \ \wedge \ b \neq c) \ \vee \ (a = c \ \wedge \ b \neq a) \ \vee \ (b = c \ \wedge \ a \neq b)) \ \wedge \ result = 3\\ &EQ : a = b \ \wedge \ b = c \ \wedge \ result = 4 \end{aligned} \tag{8}$$

The program *Swap* takes as input an array and two indexes. The values at those positions are then swapped

$$
\begin{aligned}
&int\ Swap(int[]\ A,\ int\ i,\ int\ j)\\
&Pr : A{\neq}null\ \wedge\ i \geqslant 0\ \wedge\ i < A\cdot I\ \wedge\ j \geqslant 0\ \wedge\ j < A\cdot I\\
&Po : (\forall x \in [0, l-1] \bullet (x{\neq}i\ \wedge\ x{\neq}j) \Rightarrow A'[x] = A[x])\ \wedge\\
&\qquad A'\cdot I = A\cdot I\ \wedge\ A'[i] = A[j]\ \wedge\ A'[j] = A[i]
\end{aligned}
\tag{9}
$$

Similarly to *Swap*, the program *Order* swaps the two indexed values, but only if the value in position i is bigger than the value in position j.

$$
\begin{aligned}
&int\ Order(int[]\ A,\ int\ i,\ int\ j)\\
&Pr : A{\neq}null\ \wedge\ i \geqslant 0\ \wedge\ i < A\cdot I\ \wedge\ j \geqslant 0\ \wedge\ j < A\cdot I\\
&Po : (\forall x \in [0, l-1] \bullet (x{\neq}i\ \wedge\ x{\neq}j) \Rightarrow A\prime[x] = A[x])\ \wedge\\
&\qquad A'\cdot I = A\cdot I\ \wedge\ ((A[i] > A[j]\ \wedge\ A'[i] = A[j]\ \wedge\ A'[j] = A[i])\ \vee\\
&\qquad (A[i] \leqslant A[j]\ \wedge\ A'[i] = A[i]\ \wedge\ A'[j] = A[j]))
\end{aligned}
\tag{10}
$$

The *Sorting* problem has been widely studied in computer science [15]. Given an array as input, the program should order the elements in the array such that the values in each position should be equal to or smaller than the following elements. Although the evolution of a *Sorting* algorithm has already been attempted in the past (e.g. [28,3]), those works were specialised in this task. In their cases, they used biased primitives and ad hoc fitness functions. In contrast, our framework is *general*, and does not make any particular assumption on the program to be evolved. Moreover, in the case of sorting algorithms the problem of automatically choosing the most appropriate training set has not been addressed in literature

$$
\begin{aligned}
&int\ Sorting(int[]\ A)\\
&Pr : true\\
&Po : \forall i \in [0, l-2] \bullet A'[i] \leqslant A'[i+1]\ \wedge\ A'\cdot I = A\cdot I\ \wedge\\
&\quad \forall i \in [0, A'\cdot I - 1] \bullet \exists t \in [0, A\cdot I - 1] \bullet (A'[i] = A[t]\ \wedge\\
&\quad |\{j \in [0, A\cdot I - 1] | A'[i] = A[j]\}| = |\{j \in [0, A'\cdot I - 1] | A'[j] = A[t]\}|)
\end{aligned}
\tag{11}
$$

Finally, we consider the problem of evolving a *Median* program:

$$
\begin{aligned}
&int\ Median(int[]\ A)\\
&Pr : l \geqslant 1\\
&Po : (\forall x \in A \bullet (L(x) \geqslant \lceil (A\cdot I/2) \rceil\ \wedge\ G(x) \geqslant \lceil (A\cdot I/2) \rceil) \Rightarrow result \leqslant x)\ \wedge\\
&\qquad L(result) \geqslant \lceil (A\cdot I/2) \rceil\ \wedge\ G(result) \geqslant \lceil (A\cdot I/2) \rceil\ \wedge\\
&\qquad \exists x \in A \bullet result = x\\
&L(x) : |\{t \in A | t \leqslant x\}|\\
&G(x) : |\{t \in A | t \geqslant x\}|
\end{aligned}
\tag{12}
$$

Modifying the array given as input does not break the above specification. We could have added $A' = A$ to that specification, but this would have significantly increased the complexity of the possible solutions.

By using our set of primitives, possible implementations for the different specifications are shown in Figs. 7–13. To our best knowledge, the specifications for Sorting and Median algorithms we provide cannot be refined at the moment using transformation techniques.

8.3. Experiments

We first describe how the conceptual framework has been implemented, then we discuss its parameters and how we set them. Experiments in which we compare random testing versus co-evolution and versus co-evolution with SSP follow. The performance of the testing engine is also shown. We then used the best configuration to carry out experiments with N-version programming.

For implementing the framework we used the Java language, and for the GP engine we used the open source library ECJ [1]. However, using faster languages as *C* and compiling the evolutionary programs in native code, instead of executing them

```
(seq (write_result (array 0 ))
     (loop 0 length
           (if (bigger_or_equal read_result (array index))
               skip
               (write_result (array index )))))
```

Fig. 7. Implementation of *MaxValue* defined in (6).

```
(seq (write_result 1 )
     (loop 0 length
             (if (equal (array 0 )(array index ))
                 skip
                 (write_result 0 ))))
```

Fig. 8. Implementation of *AllEqual* defined in (7).

```
(if (bigger_or_equal input_0 (add input_1 input_2))
    (write_result 1)
    (if (bigger_or_equal input_1 (add input_0 input_2))
        (write_result 1)
        (if (bigger_or_equal input_2
                             (add input_0 input_1))
            (write_result 1)
            (if (and (equal input_0 input_1)
                     (equal input_1 input_2))
                (write_result 4)
                (if (or (or (equal input_0 input_1)
                            (equal input_1 input_2))
                            (equal input_0 input_2))
                    (write_result 3)
                    (write_result 2))))))
```

Fig. 9. Implementation of *TriangleClassification* defined in (8).

```
(seq (seq (write_result (array input_0))
          (write_array input_0  (array input_1)))
          (write_array input_1  read_result))
```

Fig. 10. Implementation of *Swap* defined in (9).

```
(if (bigger (array input_0)  (array input_1))
    (seq (seq (write_result (array input_0))
              (write_array input_0  (array input_1)))
              (write_array input_1  read_result))  skip )
```

Fig. 11. Implementation of *Order* defined in (10).

```
(loop 0 length
      (loop 0 (sub length 1)
            (if (bigger (array input_0)  (array input_1))
                (seq (seq (write_result (array input_0))
                          (write_array input_0  (array input_1)))
                          (write_array input_1  read_result))
                skip )))
```

Fig. 12. Implementation of *Sorting* defined in (11).

```
(seq  (loop 0 length
            (loop 0 (sub length 1)
                  (if (bigger (array input_0)  (array input_1))
                      (seq (seq (write_result (array input_0))
                                (write_array input_0
                                             (array input_1)))
                                (write_array input_1
                                             read_result))
                      skip )))
      (write_result (array (div (sub length 1 )2 ))))
```

Fig. 13. Implementation of *Median* defined in (12).

by interpretation, would dramatically speed up the framework [38]. The reason for using Java was that it is a very easy and powerful language to use, that made possible to implement the framework in less time.

In our framework there is a very large number of parameters that need to be set. Trying to optimise all of them was not possible. Hence, we chose settings that are common in the literature, and then we did several experiments (not presented in this paper for reason of space) to tune the ones that we think are the most important. The following settings are what we finally chose. However, no guarantee on their optimality can be given. All the settings are the same for all experiments. Note that we only show the most important settings, because a detailed description of all parameters would take several pages.

We used a population size of 5000 individuals that are evolved for 200 generations. The maximum depth allowed for a tree is 12. For generating the next population, pairs of parents are chosen for reproduction. A tournament selection with size 7 is employed. In other words, for choosing each parent, seven individuals are randomly taken from the population, and the best among them is chosen as the parent.

For each pair of parents, one of the following actions is taken for generating two new offspring:

- With probability 0.1, the selected individuals are directly copied into the next generation without any modification.
- Crossover is done with probability 0.3.
- Mutation is done with probability 0.6.

Crossover takes as input two parents, and it randomly chooses one node in each of them. The offspring are copies of the parents with the sub-trees rooted at those two nodes swapped.

Mutation is used to copy the parents with a little change in their trees for generating slightly different offspring. In case of a mutation event, one of the following different mutations provided by ECJ is randomly chosen with a uniform probability, where n is a randomly chosen node in the program tree. If a particular mutation cannot be applied to n, then a new n is randomly chosen, and that is done at most 100 times. If after 100 times no mutable n is found, no mutation is done, and the new offspring will be just a copy of its parent.

Note that in our case neither crossover nor mutation generates syntactically incorrect offspring.

Point mutation: the sub-tree rooted at n is replaced with a new random sub-tree with depth 5.
OneNode mutation: n is replaced by a random node with the same constraints and arity.
AllNodes mutation: each node of the sub-tree of n is randomly replaced with a new node, but with the same type constraints and arity.
Demote mutation: a new node m is inserted between n and the parent of n. Hence, n becomes a child of m. The other children of m will be random terminals.
Promote mutation: the sub-tree rooted at the parent of n will be replaced by the sub-tree rooted in n.
Swap mutation: two children of n are randomly chosen. The sub-trees rooted at these two nodes are swapped.

At each generation, the best individual is copied to the next generation without any modification, i.e., elitism rate is set to 1 individual. All the other settings that are not listed here are used with their default values in ECJ.

To prevent the execution of very expensive programs, each program is stopped after executing at most 20 loop iterations. However, if the program takes as input an array, then the allowed loop iterations will be increased to $l^2 + 1$, where l is the length of the array given as input. A more appropriate automatic way to choose that limit will be studied in our future work.

For choosing the test cases for T_i, we report experiments with three different techniques: random sampling, co-evolution and co-evolution with SSP. Note that in all of these algorithms we enforce the constraint that all the test cases in T_i should be unique. This is done because there would be no particular benefit in testing a program on the same input more than once at each generation.

When random sampling is employed, for each T_i we sample random test cases until we get 100 that are different. Sampling random test cases is not straightforward. We need to put constraints on the size and range of the input variables. For example, sampling extremely long input arrays would make the testing phase too long. Hence, we decided that the length of the arrays is randomly chosen with 16 as the maximum length. Values inside an array A are randomly constrained in either $[-128, 127]$ or in $[-l, l]$, and integer inputs take values in $[-128, 127]$. Although constraining the variables is a common technique in software testing, the choice of these constraints is fairly arbitrary, and it might happen that faults could be discovered only with values outside those fixed ranges. More detailed analysis is needed in future work.

When co-evolution is employed, the size of each T_i is 60, and the archive has size 40. To evolve T_{i+1} from T_i a simple genetic algorithm is used. The chromosome is represented by a list of the integer inputs (if any) of the program, and the input array (if any).

A single point crossover is applied with probability 0.75. It is used separately on the list on integer inputs and on the input array. In other words, two different crossovers are employed, one that combines the lists of integer inputs, the other that combines the arrays. In the case of the input array, if the two parent arrays have different lengths, then the offspring will have a length that is the average of them.

The integer inputs are mutated with probability $1/n$, where n is the number of integer inputs. In the same way, each element in the array has probability $1/l$ of being mutated, where l is the length of the array. A mutation consists of adding a discretised Gaussian noise (mean 0 and variance 1).

Rank selection [52] is used with a bias of 1.5. Elitism rate is set to 1 individual for generation. At each new generation, a completely new individual is added to the population to prevent that the length of the arrays degenerates to a particular low value (e.g., 0). Doing that was easier than defining a new mutation operator that increases/decreases the length of arrays. Moreover, in this way we do not need to handle the problem of a computationally expensive growth of those lengths.

When random sampling is not employed, we also use a special intensive evolution of the training set for every five steps of the co-evolution. In such cases, the best program g_i^b in G_i is chosen and used to evolve the unit tests in T_i. In other words, the tests in T_i evolve using g_i^b to calculate their fitness values, but no program evolves at this time. The next test set T_{i+1} for the programs is evolved in such a way after 1024 generations are allowed to evolve T_i. It is important to note that the number of these generations is lower than the size of the population G_i. Hence, because only one program (g_i^b in particular) is used for the fitness function calculations, the computational cost of this special evolution should be cheaper than the cost of one step of the co-evolution (given the reasonable assumption that the cost of evaluating the fitness function is what contributes most to the total computational cost). However, if no speciation [17] is used, then the test case population can easily converge to very similar individuals. Although they might be high fitness individuals, the loss of diversity in the test case population might decrease the performance of the evolution of the programs.

Based on the idea presented in Section 7, for different specifications we also considered variants of the primitive set. In other words, we add to it some of the already evolved programs. We use numbers for identifying the different versions, e.g., *foo_i* means that specification of the program *foo* is evolved with a new enlarged set of primitives of index *i*. Number 1 uses the basic primitive set. Table 2 shows these configurations.

For each program specification and GP primitive configuration (i.e., a total of 13 program versions), we ran our framework with three different ways of selecting the test cases: random sampling (RS), co-evolution (COE), and co-evolution with SSP (SSP). Each of these 39 configurations has been run 100 times with different random seeds. All the following data presented in the different tables were collected from these 3900 experiments.

Table 3 shows how many times, out of 100, it was possible to evolve a correct implementation. To see whether there is any statistically significant difference between the performance of random sampling, co-evolution and SSP, we also carried out statistical tests to compare these three algorithms on each program version. The performance of such algorithms can be described as a binomial random variable, representing whether a correct program can be evolved (value 1) or not (value 0). To calculate whether there is any difference among the three random variables, we used a two-tailed Fisher's Exact Test with a 0.05 significance level on the results for each of the 13 program versions. A *p*-value lower than 0.05 means that the two random variables are statistically different (for that significance level).

To assess whether a program is correct, for each specification we have manually designed and tailored large sets T^k of 10,000 test cases each. Because no test can prove the correctness of a program, we also manually inspected the code of

Table 2
Configurations in which extra functions were added to the base set of GP primitives. These functions are correct implementations of the specifications we address in our case study.

Configurations	Added primitives
Order_2	*Swap*
Sorting_2	*Swap*
Sorting_3	*Order*
Median_2	*Swap*
Median_3	*Order*
Median_4	*Sorting*

Table 3
Number of correct evolved programs out of 100 independent runs for each program version. When a *p*-value is lower than 0.05, its cell is colored in grey.

Programs	RS	COE	SSP	Fisher's exact test *p*-values		
				RS vs. COE	RS vs. SSP	COE vs. SSP
MaxValue	8	12	15	0.480	0.182	0.679
AllEqual	0	3	3	0.246	1.000	1.000
TriangleClassification	0	0	13	1.000	0.000	0.000
Swap	0	12	18	0.000	0.000	0.322
Order_1	0	0	0	1.000	1.000	1.000
Order_2	0	18	91	0.000	0.000	0.000
Sorting_1	0	0	0	1.000	1.000	1.000
Sorting_2	2	0	0	0.497	0.497	1.000
Sorting_3	97	97	86	1.000	0.009	0.009
Median_1	0	0	0	1.000	1.000	1.000
Median_2	0	0	0	1.000	1.000	1.000
Median_3	17	8	16	0.085	1.000	0.084
Median_4	99	96	99	0.368	0.368	1.000

the programs that are able to pass all the tests in T^k. Although these two combined actions (intensive testing and inspection) give strong support to state whether a program might be correct, they do not constitute a proof. However, for the sake of simplicity, we will use the term *correct* although we do it inappropriately.

The population of test cases at the last generation can give feedback on the quality of the testing engine. If the best program in the last generation z is able to pass all the tests in T_z, we call it *valid*. If a program is correct, then it is necessarily also valid. The opposite is of course not necessarily true. If a valid program is incorrect, this would show a poor quality of the testing component of our framework. Table 4 shows the number of times in which an incorrect program was not recognised by our framework (i.e., the number of valid programs that are not correct out of 100 runs for each program version).

Finally, we investigated the use of N-version programming. For each program version, we built ensembles of size up to 10. We did it by randomly picking up the best programs in the final generation from the 100 runs. For each program version and ensemble size, we repeat this ensemble creation 100 times with different random seeds, and we check whether the ensembles are valid and correct. Results are presented in Table 5, whereas Table 6 presents the same type of experiment, but with the programs that are picked up only if they are valid. For generating the test cases, we used co-evolution with SSP. We do not show the same type of experiment with the other configurations because the results are very similar. Two-tailed Fisher's Exact Tests with a 0.05 significance level were carried out to state whether using ensembles gives different performance compared to using only a single program. In Tables 5 and 6, cells of ensembles that give statistically better performance than non-ensemble are colored in dark-grey, whereas if they give worse performance the used color is light-grey.

8.4. Discussion

Table 3 shows that the formal specifications in our case study can be automatically implemented by our framework. In fact, for all the formal specifications it was possible to evolve a correct program at least once out of 100 independent runs of the framework. However, for some configurations (4 out of 13) of the used primitives it was not possible to evolve a correct program (i.e., Order_1, Sorting_1, Median_1 and Median_2).

Table 4
For each tested configuration, it is shown the number of valid programs that are incorrect out of 100 independent runs of the framework.

Programs	RS	COE	SSP
MaxValue	0	0	0
AllEqual	34	78	71
TriangleClassification	0	72	36
Swap	85	1	0
Order_1	94	0	0
Order_2	94	66	7
Sorting_1	0	0	0
Sorting_2	0	0	0
Sorting_3	2	1	6
Median_1	0	0	0
Median_2	0	0	0
Median_3	31	11	30
Median_4	0	4	1

Table 5
For each program version, it is shown the percents of correct evolved ensembles out of 100 independent ensemble creation processes. Both valid and invalid programs were used for generating the ensembles.

Programs	Correct %	Correct ensemble								
		s2	s3	s4	s5	s6	s7	s8	s9	s10
MaxValue	15	2	10	3	2	0	0	1	0	0
AllEqual	3	0	0	2	0	0	0	0	0	0
TriangleClassification	13	2	14	4	11	6	13	7	17	11
Swap	18	6	11	8	6	4	5	7	6	9
Order_1	0	0	0	0	0	0	0	0	0	0
Order_2	91	81	98	95	100	99	100	100	100	100
Sorting_1	0	0	0	0	0	0	0	0	0	0
Sorting_2	0	0	0	0	0	0	0	0	0	0
Sorting_3	86	77	94	95	99	98	100	100	100	100
Median_1	0	0	0	0	0	0	0	0	0	0
Median_2	0	0	0	0	0	0	0	0	0	0
Median_3	16	1	6	3	6	4	6	7	6	5
Median_4	99	95	100	100	100	100	100	100	100	100

Table 6
For each program version, it is shown the percents of correct evolved ensembles out of 100 independent ensemble creation processes. Only valid programs were used for generating the ensembles.

Programs	Correct %	Correct ensemble								
		2	3	4	5	6	7	8	9	10
MaxValue	100	100	100	100	100	100	100	100	100	100
AllEqual	4	0	4	2	1	0	0	1	1	0
TriangleClassification	27	6	30	15	30	26	54	32	53	46
Swap	100	100	100	100	100	100	100	100	100	100
Order_1	0	0	0	0	0	0	0	0	0	0
Order_2	93	92	100	97	98	100	100	100	100	100
Sorting_1	0	0	0	0	0	0	0	0	0	0
Sorting_2	0	0	0	0	0	0	0	0	0	0
Sorting_3	93	90	98	99	100	100	100	100	100	100
Median_1	0	0	0	0	0	0	0	0	0	0
Median_2	0	0	0	0	0	0	0	0	0	0
Median_3	35	12	26	25	21	26	34	39	45	46
Median_4	99	99	100	100	100	100	100	100	100	100

Regarding the comparison of the three different algorithms for generating the test cases, the results in Table 3 show that SSP has statistically better performance in two problems (TriangleClassification and Order_2), but worse in one (Sorting_3). However, in this latter case it is still able to achieve a good performance of 86%. This leads us to consider SSP as the best algorithm among the ones we analysed.

Unfortunately, the performances in general are not very satisfactory (only in three cases out of 13 the performance was higher than 50%). The main reason seems to be the presence of large regions of the search space with gradient toward local optima, where a small program perfectly satisfies a single predicate of the specification but not the others. For example, in the case of *Sorting*, an empty program p_1 (i.e., a program that does not do any computation) would perfectly satisfy the constraint that the output array should be a permutation of the input one. In the same way, a program p_2, that in a single loop writes the same constant in each position of the array, would satisfy the constraint that each element should be less than or equal to the next ones. Of course, in this latter case the output array would not be a permutation of the input one.

Mutation operators do not seem to be suitable for escaping from the basin of attraction of local optima in our experiments. In this example, neighbourhood solutions (i.e., solutions with very similar tree structures) around p_1 and p_2 would fail both predicates (unless of course they are semantically equivalent to either p_1 or p_2). Effective crossover requires the presence of *building blocks* [24]. There should be blocks of the genome that would positively contribute to the fitness value even if they are present in non-optimal individuals. Crossover can be used to gather these building blocks and to spread them in the next generations (a detailed description of the building blocks theory is outside the scope of this paper, please see for example [24] for more details). Unfortunately, neither p_1 nor p_2 contains any important building block, hence an offspring generated by a crossover of p_1 and p_2 is likely to have a worse fitness value. Depending on the weights that the two predicates have in the fitness function, most of the time the population quickly converges to either p_1 or p_2, and our experiments confirmed it.

Simply increasing the population size and the number of generations would not particularly help to address the above problem. In fact, it is the fitness function that plays the main role in this problem. A way to address it would be to formulate the function c (defined in Section 4.3) as a multi-objective function [18], in which each predicate is a separate objective. By doing that, solutions that partially satisfy more than one predicate (and hence they are likely to have some good building blocks in them) will not be disadvantaged against very small simple programs that completely satisfy a single predicate but behave poorly on the other predicates.

For three program configurations (i.e., Order_2, Sorting_3 and Median_4) the performance was strangely high (>90%) compared to the other ten configurations (with the highest performance of 18% for Swap). This means that the fitness function that is automatically derived for those problems was particularly appropriate. We hence conjecture that, regarding the performance of our framework, the quality of the automatically derived fitness functions (measured in their ability of providing gradient toward the optima) is more important than the complexity of the target software. In other words, more complex software can be addressed with a good fitness function, whereas simple software can fail to be refined if the fitness function is not appropriate.

From Table 4 we can see that our testing engine was not appropriate for some problems (i.e., AllEqual, TriangleClassification and Median_3). Note that a low value in that table does not necessarily mean a good performance of the testing engine. It depends on how many correct programs were evolved. For example, if no correct program was evolved, then having no valid program that is incorrect does not give any hint on the performance of the testing engine. If the unit tests are not able to find faults, there is little hope to expect that the GP engine will evolve faultless programs. Therefore, the testing engine has a drastic impact on the final outcome, and Table 4 shows the consequences of it. In particular, the very poor performances of our framework applied to the AllEqual problem shown in Table 3 are likely due to the testing engine.

Table 5 empirically confirms Eq. (5). In other words, if the probability $C(\sigma)$ of having a correct problem is greater than 50%, then an ensemble will increase the performance. Because in the case of $C(\sigma) \leqslant 0.5$ we can observe some statistically significant degradations of the performance (for MaxValue, TriangleClassification, Swap and Median_3 in Table 5), we could hypothesise that the evolved incorrect programs have the same types of faults. This would mean that there is a large area of the search space of GP with a gradient towards a particular subset of local optima representing programs with at least one wrong behaviour in common (i.e., a particular input exists that is wrongly computed in the same way by all the evolved incorrect programs).

The performances of the TriangleClassification shown in Table 6 are quite interesting. Although $C(\sigma) < 0.5$, relative large ensembles give statistically better performance. This would mean that, although incorrect programs might be used to form an ensemble, if they are valid then there are groups of them in which each group has unrelated faults regarding the other groups, hence the performance is increased. Median_3 shows a similar trend. Although no statistically better performance is obtained, it might be possible to obtain that with larger ensemble sizes. It is worth noting that ensemble individuals generated in our experiments are not strictly independent of each other. Therefore, it is possible to produce more reliable ensembles for less reliable individuals.

From Tables 5 and 6 we can learn that, for generating an ensemble, invalid programs should not be used. A valid program is not necessarily correct, but an ensemble of them seems to give better performance. It is important to remember that a correct ensemble might be composed of only incorrect programs, as long as their faults are not positively correlated. For the same reason, theoretically it is not even necessary that the programs should be valid.

By analysing the results in Tables 5 and 6, the strategy that we suggest to use is to run the framework k times (with k depending on how much computational resource is available), and then combine the output programs of these runs in an ensemble. However, only valid programs should be used in the ensemble, that will hence have a size less than or equal to k. If no valid program is generated, we can say that the search has failed.

Given a valid program, it might be argued why we want to generate an ensemble instead of using that computational time to test the valid program more intensively. In fact, if a more intensive test phase does not show any fault in the valid program, then we can have a better confidence in its correctness. Otherwise, we can use our framework again to generate a new program, and then carry out a new intensive testing phase on it. Unfortunately, this approach has a non-trivial flaw. The problem is that the programs have already been intensively tested during the co-evolution. If no fault was found during the co-evolution, it is unlikely to find one with a new testing phase. Using a *different* testing engine for a final intensive testing phase could be a good strategy. We will investigate this in the future.

9. Conclusions

Evolving programs from their formal specifications is a novel field that this paper addresses for the first time. A new conceptual framework is introduced, where the goal of automatically refining specifications is achieved by co-evolution of programs and unit tests. The work presented in this paper shows that nature inspired computation can be used to tackle very challenging software engineering problems. These software engineering problems also pose new challenges to natural computation, for which new techniques need to be developed. This paper is a significant step forward in automatic programming using natural computation techniques.

However, the performance of our framework still needs to be further improved. Although it is recognised that the proposal of a new technology does not always give immediate industrial results, it is required to show possible ways in which stronger results can be obtained. In particular, we identify four promising ways that deserve to be investigated further:

- The reformulation of the fitness function of the programs as a multi-objective function [18] is likely to help the algorithm to prevent the convergence toward simple programs that satisfy only a single predicate of the formal specification.
- Besides being exploited for the construction of the fitness function of the programs, a formal specification might also be used to directly create useful sub-trees, which can be added to the set of employed primitives. Several different heuristics can be designed and studied to accomplish this task. For example, an expression such as $\forall x \in A$ could add the sub-tree (loop 0 length skip). This would be an example in which transformation techniques could be heuristically exploited in our framework. Of course, if a direct mapping between a formal specification and its implementation exists, then instead of using our framework it will be better to directly use transformation techniques.
- The same software specification can be written in different ways, and each of these versions will generate a different fitness function. Some of these fitness functions might have a smooth landscape, others may have many local optima. Formal techniques for transforming a specification into an equivalent one that might generate a better fitness function needs further investigation.
- Our framework is composed of complex components, and for each component several different variants exist in the literature. For example, there are several variants of GP operators and different co-evolutionary algorithms. Because our choice of the component variants was fairly arbitrary, research on how to properly do that choice, and on how to effectively tune all the components and their interactions, is likely to lead to significantly better performance for our framework.

At any rate, our conceptual framework has the following limitations:

- Even if better fitness functions might be designed, there is no guarantee that there will always be an easy search landscape. Hence, there might exist real-world classes of specifications for which our framework might not perform well.
- GP is computationally expensive, because in non-trivial applications there is the need of generating and executing millions of individuals before obtaining a final program. If the execution of the application is not too expensive, testing millions of programs is still feasible. For complex software systems, fitness evaluation is usually very expensive. Hence, even if our framework could search that space of solutions in an efficient way, the cost of each fitness function evaluation would be too high.
- In contrast to formal methods, there is no guarantee that the output of our framework is a correct implementation of the target specification.

Acknowledgements

The authors are grateful to Rami Bahsoon, David Robert White, Muhammad Zohaib Iqbal, Thomas Miconi, Damien Jade Duff, Per Kristian Lehre and Ramón Sagarna for insightful discussions. This work was initially supported by an EPSRC Grant (EP/D052785/1).

References

[1] ECJ: a Java-based Evolutionary Computation Research System, <http://www.cs.gmu.edu/eclab/projects/ecj/>.
[2] K. Adamopoulos, M. Harman, R.M. Hierons, How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, in: Genetic and Evolutionary Computation Conference (GECCO), 2004, pp. 1338–1349.
[3] A. Agapitos, S.M. Lucas, Evolving modular recursive sorting algorithms, in: Proceedings of the European Conference on Genetic Programming (EuroGP), 2007, pp. 301–310.
[4] A. Arcuri, On the automation of fixing software bugs, in: Doctoral Symposium of the IEEE International Conference on Software Engineering (ICSE), 2008, pp. 1003–1006.
[5] A. Arcuri, Evolutionary repair of faulty software, Tech. Rep. CSR-09-02, University of Birmingham, 2009.
[6] A. Arcuri, D.R. White, J. Clark, X. Yao, Multi-objective improvement of software using co-evolution and smart seeding, in: International Conference on Simulated Evolution And Learning (SEAL), 2008, pp. 61–70.
[7] A. Arcuri, X. Yao, Coevolving programs and unit tests from their specification, in: IEEE International Conference on Automated Software Engineering (ASE), 2007, pp. 397–400.
[8] A. Arcuri, X. Yao, A novel co-evolutionary approach to automatic software bug fixing, in: IEEE Congress on Evolutionary Computation (CEC), 2008, pp. 162–168.
[9] D. Ashlockand, S. Willson, N. Leahy, Coevolution and tartarus, in: IEEE Congress on Evolutionary Computation (CEC), 2004, pp. 1618–624.
[10] A. Avizienis, The n-version approach to fault-tolerant software, IEEE Transactions on Software Engineering 11 (12) (1985) 1491–1501.
[11] R. Balzer, A 15 year perspective on automatic programming, IEEE Transactions on Software Engineering 11 (11) (1985) 1257–1268.
[12] W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone, Genetic Programming: An Introduction. On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann Publishers, 1997.
[13] J.A. Clark, J.L. Jacob, Protocols are programs too: the meta-heuristic search for security protocols, Information and Software Technology 43 (14) (2001) 891–904.
[14] D. Cliff, G.F. Miller, Tracking the red queen: measurements of adaptive progress in co-evolutionary simulations, in: European Conference on Artificial Life, 1995, pp. 200–218.
[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press and McGraw-Hill, 2001.
[16] N.L. Cramer, A representation for the adaptive generation of simple sequential programs, in: Proceedings of an International Conference on Genetic Algorithms and the Applications, 1985, pp. 183–187.
[17] P.J. Darwen, X. Yao, Speciation as automatic categorical modularization, IEEE Transactions on Evolutionary Computation 1 (2) (1997) 101–108.
[18] K. Deb, Multi-Objective Optimization Using Evolutionary Algorithms, John Wiley and Sons, 2001.
[19] R. Feldt, Generating multiple diverse software versions with genetic programming, in: Proceedings of the Conference on EUROMICRO, 1998, pp. 387–394.
[20] S.G. Ficici, Solution concepts in coevolutionary algorithms, Ph.D. Thesis, Brandeis University, 2004.
[21] S.G. Ficici, J.B. Pollack, Challenges in coevolutionary learning: arms-race dynamics, open-endedness, and mediocre stable states, in: Artificial life VI, 1998, pp. 238–247.
[22] T. Higuchi, Y. Liu, X. Yao, Evolvable Hardware, Springer, 2006.
[23] W.D. Hillis, Co-evolving parasites improve simulated evolution as an optimization procedure, Physica D 42 (1–3) (1990) 228–234.
[24] J.H. Holland, Adaptation in Natural and Artificial Systems, second ed., MIT Press, Cambridge, 1992.
[25] K. Imamura, T. Soule, R.B. Heckendorn, J.A. Foster, Behavioral diversity and a probabilistically optimal gp ensemble, Genetic Programming and Evolvable Machines 4 (3) (2003) 235–253.
[26] M. Jazayeri, Formal specification and automatic programming, in: IEEE International Conference on Software Engineering (ICSE), 1976, pp. 293–296.
[27] E.D. Jong, J. Pollack, Ideal evaluation from coevolution, Evolutionary Computation 12 (2) (2004) 159–192.
[28] K.E. Kinnear, Jr., Generality and difficulty in genetic programming: Evolving a sort, in: Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, 1993, pp. 287–294.
[29] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, The MIT Press, 1992.
[30] W.B. Langdon, R. Poli, Foundations of Genetic Programming, Springer, 2002.
[31] G.T. Leavens, A.L. Baker, C. Ruby, JML: a notation for detailed design, in: Behavioral Specifications of Businesses and Systems, 1999, pp. 175–188.
[32] S. Luke, L. Panait, A comparison of bloat control methods for genetic programming, Evolutionary Computation 14 (3) (2006) 309–344.
[33] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.
[34] T. Miconi, The road to everywhere, evolution,coevolution and progress in nature and in computers, Ph.D. Thesis, University of Birmingham, 2007.
[35] D.J. Montana, Strongly typed genetic programming, Evolutionary Computation 3 (2) (1995) 199–230.
[36] G. Myers, The Art of Software Testing, Wiley, New York, 1979.
[37] C. Ngolah, Y. Wang, Exploring java code generation based on formal specifications in rtpa, in: Canadian Conference on Electrical and Computer Engineering, 2004, pp. 1533–1536.

[38] P. Nordin, W. Banzhaf, Evolving turing-complete programs for a register machine with self-modifying code, in: Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), 1995, pp. 318–325.
[39] R. Olsson, Inductive functional programming using incremental program transformation, Artificial Intelligence 74 (1) (1995) 55–81.
[40] G. Palshikar, Applying formal specifications to real-world software development, IEEE Software 18 (6) (2001) 89–97.
[41] J. Paredis, Coevolving cellular automata: be aware of the red queen, in: Proceedings of the International Conference on Genetic Algorithms (ICGA), 1997, pp. 393–400.
[42] M. Reformat, C. Xinwei, J. Miller, On the possibilities of (pseudo-) software cloning from external interactions, Soft Computing 12 (1) (2007) 29–49.
[43] C. Rich, R.C. Waters, Automatic programming: myths and prospects, Computer 21 (8) (1988) 40–51.
[44] A. Ronge, M.G. Nordahl, Genetic programs and co-evolution, developing robust general purpose controllers using local mating in two dimensional populations, in: Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation, 1996, pp. 81–90.
[45] C.D. Rosin, R.K. Belew, New methods for competitive coevolution, Evolutionary Computation 5 (1) (1997) 1–29.
[46] T. Schnier, X. Yao, Using negative correlation to evolve fault-tolerant circuits, in: Proceedings of the International Conference on Evolvable Systems: From Biology to Hardware, 2003, pp. 35–46.
[47] Y.S. Sook, A translator description language TDL for specification languages and automatic generation of their translators, Journal of Information Processing 13 (3) (1990) 339–346.
[48] J.M. Spivey, The Z Notation, second ed., A Reference Manual, Prentice Hall, 1992.
[49] N. Tracey, J. Clark, K. Mander, Automated program flaw finding using simulated annealing, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 1998, pp. 73–81.
[50] N.J. Tracey, A search-based automated test data generation framework for safety-critical software, Ph.D. Thesis, University of York, 2000.
[51] M.W. Whalen, M.P.E. Heimdahl, An approach to automatic code generation for safety-critical systems, in: IEEE International Conference on Automated Software Engineering (ASE), 1999, pp. 315–318.
[52] D. Whitley, The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best, in: Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89), 1989, pp. 116–121.