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.

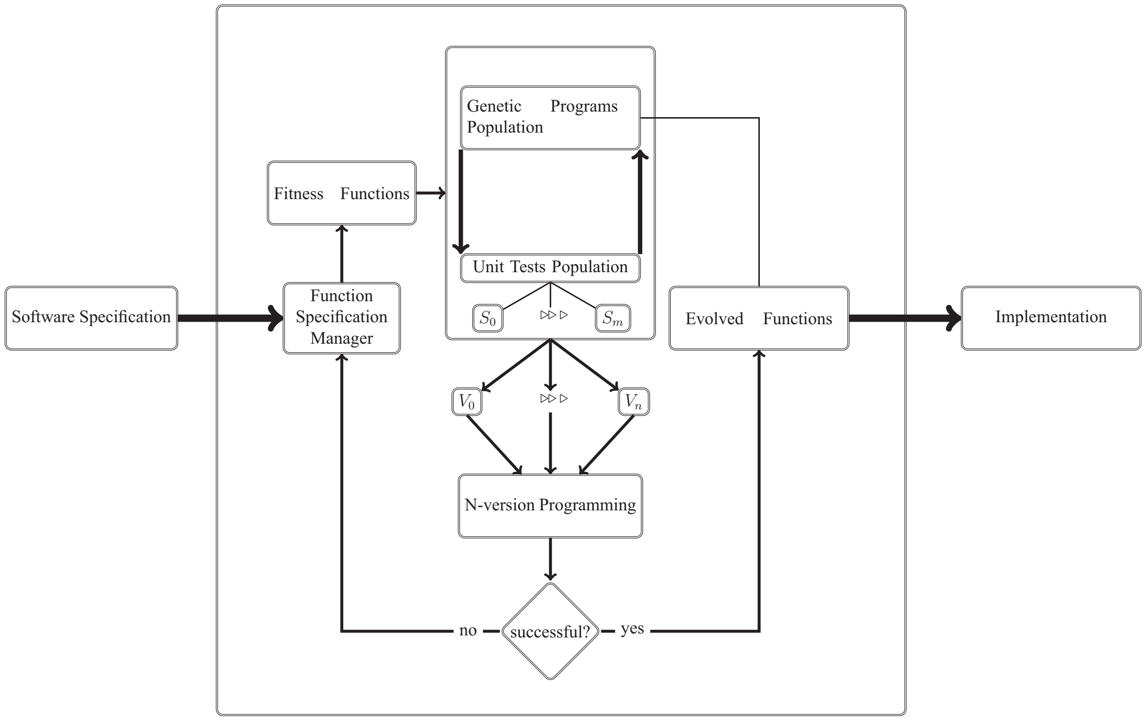

Fig. 1. High level view of the proposed framework for co-evolutionary automatic programming. *S* means "sub-population", whereas *V* means "version".