Lecture Notes in Networks and Systems 418

Ajith Abraham · Niketa Gandhi ·
Thomas Hanne · Tzung-Pei Hong ·
Tatiane Nogueira Rios ·
Weiping Ding *Editors*

Intelligent Systems Design and Applications

21st International Conference on Intelligent Systems Design and Applications (ISDA 2021) Held During December 13–15, 2021

Springer

Lecture Notes in Networks and Systems

Volume 418

Ajith Abraham · Niketa Gandhi ·
Thomas Hanne · Tzung-Pei Hong ·
Tatiane Nogueira Rios ·
Weiping Ding
Editors

Intelligent Systems Design and Applications

21st International Conference on Intelligent Systems Design and Applications (ISDA 2021) Held During December 13–15, 2021

Springer

Editors

Ajith Abraham
Scientific Network for Innovation
and Research Excellence
Machine Intelligence Research
Labs (MIR Labs)
Auburn, WA, USA

Niketa Gandhi
Scientific Network for Innovation
and Research Excellence
Machine Intelligence Research
Labs (MIR Labs)
Auburn, WA, USA

Thomas Hanne
Institut für Wirtschaftsinformatik
Fachhochschule Nordwestschweiz
Olten, Switzerland

Tzung-Pei Hong
Department of Computer Science
and information Engineering
National University of Kaohsiung
Kaohsiung, Taiwan

Tatiane Nogueira Rios
Federal University of Bahia
Ondina, Brazil

Weiping Ding
Nantong University
Nantong Shi, Jiangsu, China

ISSN 2367-3370 ISSN 2367-3389 (electronic)
Lecture Notes in Networks and Systems
ISBN 978-3-030-96307-1 ISBN 978-3-030-96308-8 (eBook)
https://doi.org/10.1007/978-3-030-96308-8

Preface

Welcome to the 21st International Conference on Intelligent Systems Design and Applications (ISDA'21) held in the World Wide Web. ISDA'21 is hosted and sponsored by the Machine Intelligence Research Labs (MIR Labs), USA. ISDA'21 brings together researchers, engineers, developers and practitioners from academia and industry working in all interdisciplinary areas of computational intelligence and system engineering to share their experience, and to exchange and cross-fertilize their ideas. The aim of ISDA'21 is to serve as a forum for the dissemination of state-of-the-art research, development and implementations of intelligent systems, intelligent technologies and useful applications in these two fields.

ISDA'21 received submissions from 34 countries, and each paper was reviewed by at least five or more reviewers, and based on the outcome of the review process, 132 papers were accepted for inclusion in the conference proceedings (36% acceptance rate). First, we would like to thank all the authors for submitting their papers to the conference and for their presentations and discussions during the conference. Our thanks go to program committee members and reviewers, who carried out the most difficult work by carefully evaluating the submitted papers. Our special thanks to the following plenary speakers, for their exciting plenary talks:

- Yukio Ohsawa, The University of Tokyo, Japan
- Juergen Branke, University of Warwick, UK
- Cengiz Toklu, Beykent University, Turkey
- Günther Raidl, Technische Universität Wien, Austria
- Kalyanmoy Deb, Michigan State University, USA
- Oscar Cordon, University of Granada, Spain
- Andries Engelbrecht, University of Stellenbosch, South Africa
- Antônio de Padua Braga, Federal University of Minas Gerais, Brazil
- Frédéric Guinand, Le Havre Normandy University, France
- Marco Dorigo, Université Libre de Bruxelles, Belgium

We express our sincere thanks to the organizing committee chairs for helping us to formulate a rich technical program. Enjoy reading the articles!

Ajith Abraham
Thomas Hanne
Weiping Ding
General Chairs

Tzung-Pei Hong
Tatiane Nogueira Rios
Program Chairs

ISDA 2021—Organization

General Chairs

Ajith Abraham	Machine Intelligence Research Labs (MIR Labs), USA
Thomas Hanne	University of Applied Sciences and Arts Northwestern Switzerland, Switzerland
Weiping Ding	Nantong University, China

Program Chairs

Tzung-Pei Hong	National University of Kaohsiung, Taiwan
Tatiane Nogueira Rios	Universidade Federal da Bahia, Brazil

Publication Chairs

Niketa Gandhi	Machine Intelligence Research Labs, USA
Kun Ma	University of Jinan, China

Special Session Chair

Gabriella Casalino	University of Bari Aldo Moro, Italy

Publicity Chairs

Aswathy S. U.	Jyothi Engineering College, Kerala, India
Pooja Manghirmalani Mishra	University of Mumbai, Maharashtra, India
Mahendra Kanojia	MVLU College, Maharashtra, India
Anu Bajaj	Machine Intelligence Research Labs (MIR Labs), Washington, USA

International Publicity Team

Mabrouka Salmi	National School of Statistics and Applied Economics (ENSSEA), Kolea, Tipaza, Algeria
Phoebe E. Knight	UTeM, Malaysia
Marco A. C. Simões	Bahia State University, Brazil
Hsiu-Wei Chiu	National University of Kaohsiung, Taiwan
Serena Gandhi	Media Specialist, CA, USA

International Program Committee

Abdul Syukor Mohamad Jaya	Universiti Teknikal Malaysia Melaka, Malaysia
Alfonso Guarino	University of Foggia, Italy
Alzira Mota	Polytechnic Institute of Porto, School of Engineering, Portugal
André Borges Guimarães Serra e Santos	Polytechnic Institute of Porto, Portugal
Angelo Genovese	Università degli Studi di Milano, Italy
Anoop Sreekumar R. S.	Manonmaniam Sundaranar University, India
Antonella Falini	University of Bari Aldo Moro, Italy
Antonella Guzzo	University of Calabria, Italy
Anu Bajaj	Machine Intelligence Research Labs, USA
Arun B Mathews	MTHSS Pathanamthitta, India
Aswathy S. U.	Jyothi Engineering College, India
Bay Vo	Ho Chi Minh City University of Technology (HUTECH), Vietnam
Biju C. V.	Jyothi Engineering College, India
Blerina Spahiu	Università degli Studi di Milano Bicocca, Italy
Carlos Pereira	ISEC, Portugal
Christian Veenhuis	Technische Universität Berlin, Germany
Claudio Savaglio	Institute for High-Performance Computing and Networking (ICAR-CNR), Italy
Daniele Schicchi	Institute for Educational Technology, National Research Council of Italy, Italy
Devi Priya Rangasamy	Kongu Engineering College, India
Dìpanwita Thakur	Banasthali University, India
E. M. Roopa Devi	Kongu Engineering College, India
Ela Pustulka	FHNW Olten, Switzerland
Elizabeth Goldbarg	Federal University of Rio Grande do Norte, Brazil
Fabio Scotti	Università degli Studi di Milano, Italy
Fariba Goodarzian	Machine Intelligence Research Labs, USA
Federico Divina	Pablo de Olavide University, Spain
Gabriella Casalino	University of Bari Aldo Moro, Italy
Gautami Tripathi	Jamia Hamdard, India

Rohit Anand	DSEU, G.B. Pant Okhla-1 Campus, New Delhi, India
Ruggero Donida Labati	Università degli Studi di Milano, Italy
Sabri Pllana	Center for Smart Computing Continuum, Forschung Burgenland, Austria
Sidemar Fideles Cezario	Federal University of Rio Grande do Norte, Brazil
Sílvia Maria Diniz Monteiro Maia	Federal University of Rio Grande do Norte, Brazil
Sindhu P. M.	Nagindas Khandwala College, India
Subodh Deolekar	Welingkar Institute of Management Development and Research, India
Sulaima Lebbe Abdul Haleem	South Eastern University of Sri Lanka, Sri Lanka
Susana Cláudia Nicola de Araújo	Instituto Superior de Engenharia do Porto (ISEP), Portugal
Syariffanor Hisham	Universiti Teknikal Malaysia Melaka, Malaysia
Thatiana C. Navarro Diniz	Federal Rural University of the Semi-arid Region, Brazil
Thiago Soares Marques	Federal University of Rio Grande do Norte, Brazil
Thomas Hanne	University of Applied Sciences and Arts Northwestern Switzerland, Switzerland
Tzung-Pei Hong	National University of Kaohsiung, Taiwan
Wen-Yang Lin	National University of Kaohsiung, Taiwan
Wenbin Pei	Lanzhou University, China
Yibao Zhang	Lanzhou University, China
Youssef Ghanou	Moulay Ismail University of Meknes, Morocco
Zurina Saaya	Universiti Teknikal Malaysia Melaka, Malaysia

Contents

Open-Ended Automatic Programming Through Combinatorial Evolution

Sebastian Fix, Thomas Probst, Oliver Ruggli, Thomas Hanne, and Patrik Christen(✉)

FHNW University of Applied Sciences and Arts Northwestern Switzerland, 4600 Olten, Switzerland
patrik.christen@fhnw.ch

Abstract. Combinatorial evolution – the creation of new things through the combination of existing things – can be a powerful way to evolve rather than design technical objects such as electronic circuits. Intriguingly, this seems to be an ongoing and thus open-ended process creating novelty with increasing complexity. Here, we employ combinatorial evolution in software development. While current approaches such as genetic programming are efficient in solving particular problems, they all converge towards a solution and do not create anything new anymore afterwards. Combinatorial evolution of complex systems such as languages and technology are considered open-ended. Therefore, open-ended automatic programming might be possible through combinatorial evolution. We implemented a computer program simulating combinatorial evolution of code blocks stored in a database to make them available for combining. Automatic programming in the sense of algorithm-based code generation is achieved by evaluating regular expressions. We found that reserved keywords of a programming language are suitable for defining the basic code blocks at the beginning of the simulation. We also found that placeholders can be used to combine code blocks and that code complexity can be described in terms of the importance to the programming language. As in a previous combinatorial evolution simulation of electronic circuits, complexity increased from simple keywords and special characters to more complex variable declarations, class definitions, methods, and classes containing methods and variable declarations. Combinatorial evolution, therefore, seems to be a promising approach for open-ended automatic programming.

Keywords: Automatic programming · Combinatorial evolution · Open-endedness

1 Introduction

Genetic algorithms and evolutionary computation in general are widely used for solving optimisation problems [6]. Such algorithms follow the paradigm of biological evolution. They consist of a collection of virtual organisms, where every organism represents a possible solution to a given problem. Some fitness measure is then calculated for each organism in an iterative process and it tries to find improved solutions by forming random mutations and crossovers on them.

In contrast to such evolutionary computation, *combinatorial evolution* as proposed by W. Brian Arthur [1,2], makes no modifications to the organisms themselves. New solutions are formed through the combination of existing components which then form new solutions in later iterations with the goal of satisfying certain needs. The more useful a combination is, the higher is its need rating. Combining existing components to construct new components can be observed in the evolution of technology [1,2]. For instance, the invention of radar was only possible through combining simpler electronic parts fulfilling functions like amplification and wave generation [3]. In order to investigate combinatorial evolution, Arthur and Polak [3] created a simple computer simulation, where electronic circuits were evolved in a combinatorial manner. Their simulation started by randomly combining primitive elementary logic gates and then used these simpler combinations for more complicated combinations in later iterations. Over time, a small number of simple building blocks was transformed into many complicated ones, where some of them might be useful for future applications. It was concluded that combinatorial evolution allows building some kind of library of building blocks for the creation of future and more complicated building blocks.

Intriguingly, combinatorial evolution is a key ingredient to achieve open-ended evolution [27,29], that is the ongoing creation of novelty [4,26]. This contrasts classical computational approaches where the aim is to converge towards a solution as fast as possible. Computational approaches according to open-ended evolution are therefore not more efficient but they are more creative since they generate ongoing novelty.

Here we want to explore whether combinatorial evolution could be also applied to software development, more specifically to automatic programming to eventually make it open-ended [7]. An early idea of automatic programming was to implement high-level programming languages that are more human readable resulting in compilers, which produce low-level programs – down to machine code – from human readable syntax [8]. However, human input in some form was still needed and the programming task was simply transferred to a higher level. Furthermore, the software solution is limited by the programmer's capabilities and creativity. Language therefore remains a barrier between programmers and computers. A way around this barrier would be to let the computer do the programming (also occasionally denoted as metaprogramming [10]), which might even lead to better programs. Koza [18] addressed this issue through genetic programming, where populations of computer programs are generated by a computer using genetic algorithms. The problem space consists of programs that try to solve (or approximately solve) problems. It has been demonstrated that random mutations and crossovers in source code can effectively contribute in creating new sophisticated programs [24].

Therefore, it seems possible to define a programming task and let a computer do the programming. However, looking at the process of software development, programming seems more comparable to technological rather than biological evolution. Existing libraries or algorithms are often integrated into new software without the necessity of modifying them. Therefore, an automatic programming approach that creates new computer programs by means of combinatorial evolution might be an interesting alternative to genetic programming. Also, due to open-endedness, combinatorial evolution holds the promise to be more creative generating ongoing novelty. In the present study we investigate ways to define a programming task for automatic programming through combinatorial evolution including the evaluation of the generated code with a need rating. Our research question is whether it is possible to generate computer programs of increasing complexity using automatic programming through combinatorial evolution. Specifically, we ask what kind of basic code blocks are needed at the beginning? How are these code blocks implemented to allow them to combine? How can code complexity be measured?

2 Automatic Programming

Since the development of computers, it has been a challenge to optimise and adapt program code to access the potential performance of a computer. While the computational power of computers has been steadily increasing in recent years, program code is still limited by the ability of programmers to create efficient and functioning code. Programming languages have also evolved over the past decades. The development of programming languages has sought to provide programmers with abstractions at higher levels. However, this also led to limitations, especially regarding performance and creativity. It is thus intriguing to shift the programming to the computer itself. Most of the programming is currently done by human programmers, which often leads to a time-intensive and error-prone process of software development. The idea that computers automatically create software programs has been a long-standing goal [5] with the potential to streamline and improve software development.

Automatic programming was first considered in the 1940s describing the automation of a manual process in general and with the goal to maximise efficiency [22]. Later, automatic programming was considered a type of computer programming in which code is generated using tools that allow developers to write code at a higher level of abstraction [22]. There are two main types of automatic programming: *application generators* and *generative programming*. Cleaveland [9] describes the development of application generators as the use of high-level programming models or templates to translate certain components into low-level source code. Generative programming, on the other hand, assists developers in writing programs. This can be achieved, e.g. by providing standard libraries as a form of reusable code [10]. In generative programming it is crucial to have a domain model, which consists of three main parts: a problem space, a solution space, and a configuration knowledge mapping that connects them [11]. The problem space includes the features and concepts used by application engineers to express their needs. These can be textual or graphical programming languages, interactive wizards, or graphical user interfaces. The solution space consists of elementary components with a maximum of combinability and a minimum of redundancy. The configuration knowledge mapping presents a form of generator that translates the objects from the problem space to build components in the solution space [10]. Most recently, automatic programming shifted towards higher level programming languages and incorporating even more abstraction [21].

While these kinds of automatic programming heavily depend on human interaction and thus the capabilities and creativity of programmers, genetic programming can be regarded an attempts to reduce this dependency and shift the focus to automation done by the computer itself. Koza [18] describes genetic programming as a type of programming in which programs are regarded as genes that can be evolved using genetic algorithms [15,16]. It aims to improve the performance of a program to perform a predefined task. According to Becker et al. [5], a genetic algorithm takes, as an input, a set of instructions or actions that are regarded as genes. A random set of these instructions is then selected to form an initial sequence of DNA. The whole genome is then executed as a program and the results are scored in terms of how well the program solves a predefined task. Afterwards, the top scorers are used to create offspring, which are rated again until the desired program is produced. To find new solutions, evolutionary techniques such as crossover, mutation, and replication are used [23]. Crossover children are created by picking two parents and switching certain components. Another technique is mutation, which uses only one individual parent and randomly modifies its parts to create a new child. Sometimes parents with great fitness will be transferred to the next iteration without any mutation or crossover because they might do well in later steps as well.

3 Combinatorial Evolution

With combinatorial evolution, new solutions build on combinations of previously discovered solutions. Every evolution starts with some primitive, existing building blocks and uses them to build combinations. Those combinations are then stored in an active repertoire. If the output satisfies a need better than an earlier solution, it replaces the old one and will be used as the building block in later iterations. Building blocks are thus not modified, they are combined together creating new building blocks. The result is a library of functionalities that may be useful for a solution in the future [1,2].

As Ogburn [20] suggested, the more equipment there is within a material culture, the greater the number of inventions are. This is known as the Ogburn's Claim. It can therefore be inferred that the number and diversity of developed components as well as their technological developments matters because next generation components build upon the technological level of the previous, existing components. To investigate this, Arthur and Polak [3] created a simple computer simulation to 'discover' new electronic circuits. In their simulation, they used a predefined list of truth tables of basic logic functions such as full adders or n-bit adders. Every randomly created combination represented a potential satisfaction of a need, which was then tested against this list. If the truth table of a newly created circuit matched one from the predefined list, it is added to the active repertoire as it fulfils the pre-specified functionality. Sometimes, it also replaced one that was found earlier, if it used fewer parts and therefore would cost less. New technologies in the real world are not usually found by randomly combining existing ones nor do they exist in a pre-specified list to be compared against. Nevertheless, their needs are generally clearly visible in economics and current technologies [3].

Combinatorial evolution is in general an important element of evolutionary systems. Stefan Thurner and his colleagues developed a general model of evolutionary dynamics in which the combination of existing entities to create new entities plays a central role [27–29]. They were able to validate this model using world trade data [17], therefore underlining the importance of evolutionary dynamics in economic modelling in general and combinatorial interactions in particular. The model shows punctuated equilibria that are typical for open-ended evolutionary systems [27–29].

4 Code Complexity

Genetic algorithms have been used for automatic programming already, however, a large number of iterations are required to significantly increase code complexity in order to solve more complex problems [14]. It therefore seems beneficial to use combinatorial evolution in which complexity seems to increase in fewer steps and thus less time.

Code complexity has been measured in this context with different approaches. The cyclomatic complexity of a code is the number of linearly independent paths within it [12]. For instance, if the code contains no control flow elements (conditionals), the complexity would be 1, since there would be only a single path through the code [30]. If the code has one single-condition IF statement, the complexity would be 2 because there would be two paths through the code – one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE [30]. Two nested single-condition IFs (or one IF with two conditions) would produce a complexity of 3 [19,30]. According to Garg [13], cyclomatic complexity is one of the most used and renowned software metrics together with other proposed and researched metrics, such as the number of lines of code and the Halstead measure. Although cyclomatic complexity is very popular, it is difficult to calculate for object-oriented code [25].

5 Methods

5.1 Development Setup and Environment

We used the programming language Java though other programming languages would have been feasible as well. The development environment was installed on VirtualBox – an open source virtualisation environment from Oracle. Oracle Java SE Development Kit 11 was used with Apache Maven as build automation tool. To map the existing code with a database, Hibernate ORM was used. It allows mapping object-oriented Java code to a relational database. Furthermore, code versioning with GitHub was used.

5.2 Simulation

Simulations are initialised by adding some basic code building blocks into a repository. The first simulation iteration then starts by randomly selecting code blocks from this repository. Selected blocks are then combined into a new code block, which subsequently gets analysed for its usefulness and complexity. Based on this analysis, the code block is assigned a value. Nonsense code, which is the most common result when randomly combining keywords of a programming language, are assigned a value of 0 and not used any further. Only code blocks with a value greater than 0 are added to the repository and consequently have a chance of being selected in a later iteration.

5.3 Code Building Blocks

Preliminary experiments in which code snippets with placeholders were predefined showed that this approach would limit the creativity and complexity of the automatic programming solution by the predefined snippets. The simulation would only create program logic that is already given by the basic set of code blocks.

To overcome this limitation, we defined basic code building blocks according to keywords and special characters of the Java programming language, e.g. the keywords `int`, `for`, `class`, and `String` as well as the special characters `&`, `=`, `;`, and `{`. Additionally, we defined three more extra code blocks: First, `PLACEHOLDER` to define where blocks allow other code blocks to be combined and integrated. This is particularly important for nesting certain code elements, such as methods that must be nested into a class construct to be valid Java code. Second, `NAME` to name something, e.g. classes, methods, and variables. And third, the special keyword `main` in the main method definition.

5.4 Selecting and Combining Code Blocks

During the selection process, new source code is generated based on combinations of existing code blocks from the repository. The chance that a particular code block is selected depends on its classification value (see next section). In a first step, a helper function defines a random value of how many code blocks are taken into consideration in the current iteration. There is a minimum of two code blocks required to generate a new code block. The maximum number can be predefined in the program. Arthur and Polak [3] combined up to 12 building blocks. To reduce the number of iterations needed for receiving valid Java code, a maximum of eight blocks turned out to be a good limit. After randomly defining the number of code blocks to be combined, the weighted random selection of code blocks based on their classification value follows. Instead of simply chaining all selected code blocks together, there is also the possibility to nest them into a placeholder if available. A random function decides whether a code block is nested into the placeholder, or simply added to the whole code block. This procedure is important because program code usually exhibits such nested structures.

5.5 Code Analysis and Building Block Classification

After the selection and combination process, the newly generated source code is passed into the classification function where it gets analysed. The classification process is required to weight the different code blocks according to their relevance in the Java programming language and to see whether the code evolved with respect to complexity. This is achieved with regular expression patterns, which allow identifying relevant Java code structures such as classes and methods that can be weighted with predefined classification values for these code structures. Basic structures such as variable declarations are assigned a value of 1. More elaborate structures such as classes have a value of 2 and even more complicated structures such as methods have a value of 3. If a structure contains several of these substructures, their classification values is added. An important structure in many programming languages is the declaration of a variable. With the following regular expression, any declaration of the value types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short` are detected:

```
(PLACEHOLDER(?!PLACEHOLDER))?
(boolean|byte|char|double|float|int|long|short) NAME;
(PLACEHOLDER(?!PLACEHOLDER))?
```

Other important elements are brackets. E.g. they are used in methods and classes specifying the body. The syntax is given by the programming language. Placeholders inside brackets are important, they allow new code to be injected into existing code blocks in future combinations. We therefore created the following regular expression:

```
^(\{PLACEHOLDER\}|\(PLACEHOLDER\))$
```

As already shown in the simple simulation with electronic circuits [3], one needs a minimal complexity of the initial building blocks to be able to generate useful and more complex future combinations. Classes and methods are essential to build anything complex in Java. Therefore, regular expressions were implemented to identify valid classes and methods. Valid means, the element is closed and it successfully compiles. Variable declarations and methods are allowed to be nested in the class structure. The following regular expression to detect classes was developed:

```
(protected|private|public) class NAME \{
((boolean|byte|char|double|float|int|long|short) NAME;
|(protected|private|public) void NAME\(
((boolean|byte|char|double|float|int|long|short) NAME)?\) \{
((boolean|byte|char|double|float|int|long|short) NAME;
```