Biuletyn WAT
Vol. LXVIII, Nr 1, 2019

Artificial intelligence for software development — the present and the challenges for the future

ŁUKASZ KORZENIOWSKI, KRZYSZTOF GOCZYŁA

Gdańsk University of Technology, Faculty of Electronics,
Telecommunications and Informatics, 11/12 Narutowicza Str.,
80-233 Gdańsk, Poland, lkorzeni@gmail.com, kris@eti.pg.edu.pl

Abstract. Since the time when first CASE (Computer-Aided Software Engineering) methods and tools were developed, little has been done in the area of automated creation of code. CASE tools support a software engineer in creation the system structure, in defining interfaces and relationships between software modules and, after the code has been written, in performing testing tasks on different levels of detail. Writing code is still the task of a skilled human, which makes the whole software development a costly and error-prone process. It seems that recent advances in AI area, particularly in deep learning methods, may considerably improve the matters. The paper presents an extensive survey of recent work and achievements in this area reported in the literature, both from the theoretical branch of research and from engineer-oriented approaches. Then, some challenges for the future work are proposed, classified into *Full AI*, *Assisted AI* and *Supplementary AI* research fields.
Keywords: software development, artificial intelligence, machine learning, automated code generation
DOI: 10.5604/01.3001.0013.1464

1. Introduction

For the past 30 years, software engineering industry has been struggling to introduce repeatable processes of software development and to improve their efficiency. The experience of building large scale systems proved that manual software development processes are very error-prone and it is the human being their biggest power but also the weakest link. Although the concept of automated creation of programs was in the area of interest of researchers since 1960's, lack of powerful enough hardware prevented this idea from being truly explored. Introduction of CASE tools

brought the promise of minimizing the number of human mistakes by following well-defined processes decently supported by dedicated software. Despite recent significant loss of focus, mostly due to common adoption of agile processes, CASE has established a firm division of software engineering disciplines which is commonly shared and understood in software development industry today.

Recent improvements in processing capabilities of modern hardware and software platforms like cloud computing, supported by significant advances in development of algorithms, have caused machine learning (ML) to gain its momentum. After establishing its position in areas like image classification or speech recognition, researchers are looking into using machine learning for automated construction of programs. There are several practical reasons for introducing ML (and, more general, artificial intelligence — AI) methods and techniques into software development processes:

- traditional process, regardless of the methodology applied, is highly manual and therefore expensive and error-prone;
- the COTS (Commercial Off-The-Shelf) approach turns out to be equally expensive: additional effort is required to find and acquire appropriate software solutions; ready-to-use software quickly becomes obsolete, which increases maintenance cost, etc.), so companies are reluctant to adapt their internal processes to existing solutions;
- there is a lack of software developers on the job market.

In this paper we present a survey of current achievements in employing AI into software development processes conducted using an exploratory literature review. Early this year, two papers were published that present classification of current research directions in this area. Gottschlich *et al.* [1] identify three main areas of development which focus around the characteristics of learning process: intention, invention, and adaptation. The authors describe the landscape of research from the perspective of enhancements that particular research directions can bring in these essential capabilities of learning. Kant [2] summarizes recent advances in program synthesis putting most attention at different organizations of neural network architectures and corresponding learning algorithms that leverage them. The two papers, however, cover only selected excerpts of the work that has been done so far and do not refer holistically to software engineering processes. Our work extends these surveys with the practical applicability dimension by putting recent research on automated programming in the context of software development process. Recent advances in automated software systems development are presented here from the point of view of classically understood disciplines of software development:

- requirements specification,
- system design,
- system development,
- system verification.

MOST WIEDZY Downloaded from mostwiedzy.pl

The structure of the paper is as follows: In Sections 2, 3, 4, and 5 we present current research directions and summarize reported achievements in introducing AI methods into the abovementioned four disciplines, or stages, of software development. To the best of our knowledge, there exists no such a holistic overview of this research area. In Section 6 we contribute to the field by proposing concrete subjects for further research, formulating them from the software engineering point of view. We classify them into *Full AI*, *Assisted AI*, and *Supplementary AI* research fields which represent different classes of expectations that researchers could express against usage of AI for software development. *Full AI* groups research directions, aimed at gaining the ability to automatically generate any program based on a specification given, *Assisted AI* focuses on cooperation possibilities between a business user and AI towards semi-automated program creation while *Supplementary AI* represents research around using AI to improve performance/quality of manual software development process. These classes of research are presented in decreasing order of complexity and increasing order of feasibility and applicability in a short-time horizon. Section 7 concludes the paper.

2. Requirements specification

Gathering requirements for building software systems has always been a major challenge. Regardless of the level of formalism dictated by the process being used, capturing the intention of business users was a long and rather tedious task. In practice, it is often much easier for business user to verify whether the output of a program is correct or not than to specify the procedure of producing this output to the level of detail necessary for a programmer to develop the software. Utilizing machine learning techniques to create programs automatically leads to interesting considerations regarding the way to specify such programs.

Typically, machine learning employs some sort of neural network that is trained using a set of examples expressing the desired behaviour of the program. It is then expected that through learning the neural network would generalize the learned behaviour and be able to correctly respond to an input that differs from examples used during training. The set of training examples forms partial specification of the problem and it is the role of learning algorithm to determine the rest. This approach has proven to be very successful in many areas like image or speech recognition, especially with usage of deep neural networks fed with large set of training examples. On the other hand, trying to employ such approach for automated creation of programs would meet a number of obstacles. Assuming that it is a business user that needs to provide this partial specification, is she able to prepare a big enough training set for the learning algorithm to discover her intention? If that is not possible or impractical then is there any other way of providing the intention? This

MOST WIEDZY Downloaded from mostwiedzy.pl

section explores several research directions in this area and presents related recent work (see Figure 1).

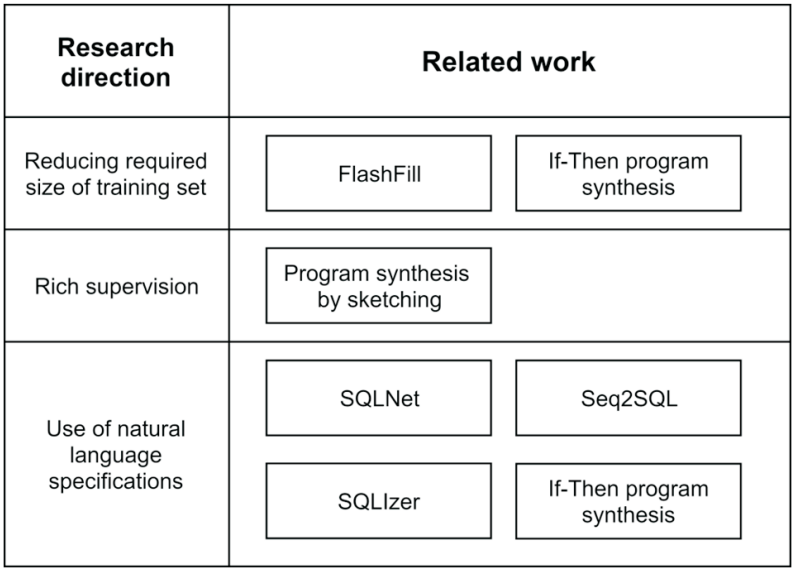

Fig. 1. Research directions in the requirements specification area

2.1. Reducing required size of training set

This problem is often referred to as one-shot learning or more general *k*-shot learning where learning algorithm is expected to provide reasonable results after being trained with only *k* input-output examples.

A notable work in this field was described by Gulwani in [3], which presents the FlashFill algorithm for automated creation of string manipulation programs in Excel spreadsheet (see an example in Figure 2). The authors present a solution that is able to determine a program using only a very limited set of input-output examples (typically up to 4). What is especially interesting from the point of view of automated software development is the interactive approach that is taken to synthesize programs. The algorithm asks a business user for a pair of input-output strings and tries to determine a suitable program. User tries to use the program but if she finds it to generate wrong output for some other input she provides another training example, and the cycle repeats. The underlying mechanism used in the algorithm is the program synthesis from basic string manipulation functions and using dedicated data structure to efficiently search the space of all possible solutions and find the one that fits best into the set of given input-output examples. While

MOST WIEDZY Downloaded from mostwiedzy.pl

the mentioned paper describes the program synthesis in a fairly simple domain of string manipulation, the approach of interaction between a business user and a learning algorithm is a dream-come-true in the area of expressing requirements. The low number of required examples, together with ease of expression, make it very simple for business users to utilize. Of course, as the source domain becomes more complicated, the difficulty of such synthesis increases dramatically.

User data	ZIP code
Jan Kowalski, 84-307 Gdynia, Morska 12	84-307
Zenon Iksiński, 32-200 Elbląg, Wawelska 15	32-200
Anna Kowalska, 12-456 Kraków, Studzienna 75	12-456

Fig. 2. A FlashFill example — the algorithm learns the string manipulation formula to extract ZIP code embedded in user data

Chen *et al.* [4] describe an approach to create programs composed from a set of predefined triggers and actions based on a natural language specification. The work focuses on the If-Then program domain which consists of simple programs expressed with set of conditions and corresponding actions. The goal of learning is to find the correct program based on a natural language description. Instead of using an individual training set for each such problem, the authors assume usage of a shared repository of natural language descriptions (e.g. IFTTT.com) which effectively serves as a common training set. Based on analysis of existing descriptions, the learning algorithm applies the attention approach (further explained in section 4.1) to determine keywords in the natural language description that would represent conditions and actions in the best way and then translates such a description into a program. An interesting scenario is described when the already trained system is extended with a newly available trigger or an action. In such a scenario, the repository initially lacks natural language descriptions using the new features. The goal for the learning algorithm is to gain good level of prediction for new features while not deteriorating prediction accuracy for the existing ones. This effectively represents the maintenance scenario in classical software development processes where after having already a working software a change in its specification takes place. Although prediction accuracy for new features achieved by the authors is over 80%, it uncovers the difficulty of introducing a change into an automatically created program — a special way of training is required and the initial results for new features are much worse that for existing ones.

MOST WIEDZY Downloaded from mostwiedzy.pl

2.2. Rich supervision

A lot of current research efforts in the area of automated construction of programs achieve some notable improvements but require large training sets. Taken the current results of efforts to reduce the required size of training set, it seems that achieving the state when the learning process can be supervised by a human is rather far ahead of researchers. One might also conclude that it is unrealistic to expect a learning algorithm to find correct solution based only on input-output examples.

An alternative solution could be to provide more information to a learning algorithm (so called *rich supervision*). Solar-Lezama [5] describes a technique called sketching where a user presents a partial program with a number of placeholders to be filled in during the learning process (see Figure 3). Apart from the program sketch, the learning algorithm is presented a set of test cases that serve to validate concrete program candidates discovered during learning. The author proves the hypothesis that well-formed generic sketch, supported by a relevantly small number of test cases, allows to clearly define the intention of the user.

```
list reverseEfficient(list l){
        list nl = new list();
        while( □ ){ □ }
}
```

Fig. 3. Example of program sketch from Solar-Lezama [5] — a hint is provided to the learning algorithm that some iteration over list will be required

Looking at this approach from the point of view of classical software development processes, it can be observed that it is of potentially huge practical value. It utilizes common techniques like Test Driven Development and the skill set of today's programmers which would specify their intention with programming. As it can be seen from the mentioned paper, this approach allows also to synthesize quite complex programs, e.g. AES encryption algorithm. On the downsides, it can be pointed out that this approach places a software developer as the one to provide partial specification of the program which still requires additional step of gathering system requirements from business users. It shows, however, that using rich supervision can be an interesting research direction.

MOST WIEDZY Downloaded from mostwiedzy.pl

2.3. Use of natural language specifications

The previous subsection describes some proposal for expressing partial specification of a program to be built utilizing programming itself as a form of expression. But instead of trying to introduce a formal way of providing partial specification, one could provide a full specification, but stated less formally. One example of such an approach was already described by Chen *et al.* [4] for simple If-Then programs. Another recent attempts are described by *Xu et al.* [6], Zhong *et al.* [7], and Yaghmazadeh *et al.* [8] where the authors synthesize SQL queries based on a natural language description (see Figure 4). Although generating SQL queries from a predefined database schema is much easier task than creating full-fledged programs with complex syntax running over unknown data model, it is still a promising direction. Capability of natural language to describe a problem might give valuable hints for algorithms to direct their learning. A huge benefit is that such specification allows for a direct communication between business user and learning algorithms without need to learn any additional skills or tools to facilitate this communication. At the same time, the biggest challenge would be the inherent ambiguity of natural language.

```
SELECT $AGG $COLUMN
WHERE $COLUMN $OP $VALUE
(AND $COLUMN $OP $VALUE)*
```

Fig. 4. SQL query sketch used by Xu *et al.* [6]. Tokens starting with $ are placeholders to be filled in by a neural network

3. System design

In classic software development processes there is a phase (formal or informal) when a big system is decomposed into smaller subsystems and those are further divided into modules with clear responsibilities. At this point, the core architecture of the system is usually formed with well-defined interfaces, contracts, and communication styles between constituent modules. Agile processes like Scrum or eXtreme Programming lack an explicit phase of system design but during each iteration of the process development team makes concrete decisions on software architecture, responsibility of individual services and their cooperation patterns. Regardless of the software development process or architecture style used, the ultimate goal of these activities is to reduce the impact of future changes on the whole system. By introducing clear separation of responsibility between components, it is hoped that any changes would remain local.

MOST WIEDZY Downloaded from mostwiedzy.pl

If we look at automated creation of programs, especially with the usage of neural networks, then modularization is neither obvious nor easy to achieve. Neural networks are usually trained as a whole and there is no notion of components. This would effectively prevent such solutions from widespread use as the impact of introducing even a little change would be very difficult to predict and the risk hard to accept. There are, however, some promising research directions, described below in this section, that have the potential of addressing this problem (see Figure 5).

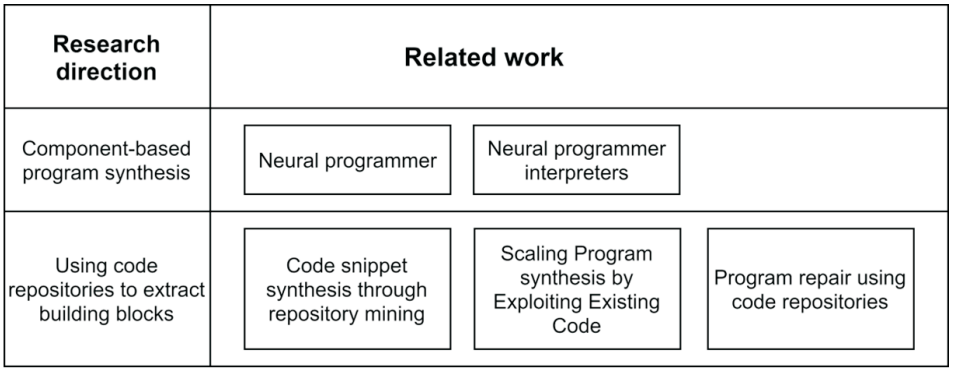

Fig. 5. Research directions in the system design area

3.1. Component-based program synthesis

In order to achieve modularity, each of the components constituting a bigger program should be trained separately. But what then remains is the problem of learning how to synthesize the program from those components.

Authors of Neelkantan *et al.* [9] describe their attempt to utilize basic arithmetic and logic functions in the process of learning a neural network. The specification for the program is given by a query expressed in natural language (in a similar fashion that the one described by Chen et al. [4]). This time, however, the logic of the program may be significantly more complex. Learning program takes place in a fixed number of steps. Initially, the program is constructed with predefined functions and in each subsequent step it is enhanced. Experiments done by the authors show that their approach outperforms Long Short-Term Memory (LSTM, Hochreiter-Schmidhuber [19]) based methods.

Another attempt of this kind is described by Reed *et al.* [10]. Here, also a program being learned is composed from lower-level programs, but the authors focus on learning only the composition part while the constituent programs are not even executed. The authors have also decided to use rich supervision for learning. Instead of presenting input-output examples, they use pairs of input and corresponding

MOST WIEDZY Downloaded from mostwiedzy.pl

program execution trace (equivalent of call stack in programming languages). The learning algorithm is not focused on converging to particular output but to particular execution trace. This approach has been proved to possess some interesting properties. In the first place, it is generic: the same target program may use different sets of subprograms, which is the essence of modularity — one implementation can be exchanged with another one as long as its contract remains the same. Secondly, the training goal of an underlying neural network is only to learn how to compose lower-level programs, which abstracts from their complexity. This allows the network to be much simpler and learning process much more efficient as compared to models where a big program is trained as a whole. Moreover, this idea opens the possibility of automatic composition of programs from subprograms that are also automatically generated and the level of nesting does not influence the learning efficiency. Additionally, the authors discuss an approach towards embracing change. They suggest a procedure of introducing a new "feature" (a new subprogram) into the learning algorithm in a way that would not cause deterioration of neural network's capabilities in solving previously trained set. Last but not least, the paper considers a multi-task program which is a single program having the functionality of multiple individual programs. The authors prove that accuracy of the multi-task program remains at the same level as that of individual programs trained separately. This shows a great potential of automatic synthesizing not only simple algorithms but also complex programs solving real-world problems.

3.2. Using code repositories to extract building blocks

While the previous direction presents a top-down approach to automated program construction, also a bottom-up approach can be considered. In this scenario, higher-level programs would be discovered by using the source code of lower level subprograms. The extremely fast growing code repositories like GitHub could serve as a great source of such subprograms.

One example of this approach is described by Pavlinovic *et al.* [11]. The authors present a method for suggesting code snippets to a programmer based on the knowledge gathered from code repositories. The process consists of two steps. At the first step, repository mining takes place to extract the most commonly used snippets. The second step is triggered on user demand; it analyzes the code that was already manually created and tries to find suitable snippets that would fit best. The decision on which snippet to apply is left to the programmer.

Although this is a very basic example, it reveals the potential of exploring existing code repositories and extracting valuable information from them. Another potential application is described by Bornholt *et al.* [13] where the authors propose usage of code repositories to reveal sketches that would serve as a basis for learning programs following the Programming by Sketching approach described

MOST WIEDZY Downloaded from mostwiedzy.pl

by Solar-Lezama [5]. Xin *et al.* [12], in turn, explore the potential of source code gathered in repositories to automatically detect issues in existing programs. Although all of these attempts currently manage to deal only with problems of limited scale, usage of existing large code bases is definitely a direction that is worth of exploring. It is also the direction that can bring artificial intelligence to the next level as it opens the possibility of learning new capabilities by exploring the existing knowledge.

4. System development

Programming is the part of software development process that still remains purely manual. Over the last years, significant progress has been made in the area of tools aiding software developers (e.g. automated generation of code snippets or hints for improving code quality) but all these attempts place a human in the center of programming activity. The consequence of the assumption that programming is a manual task is conclusion that the code written must be understandable for humans. This leads to the whole big area of code quality, design patterns, test driven development (TDD) approach or general rules of writing maintainable programs which have been intensively explored by researchers over last 20 years. On the other hand, if we assume that it is not a human that is writing the code, then some of these rules may no longer apply. For example, automatically generated code potentially does not have to be easily readable for humans as it can be also maintained automatically. Although automated programming is a very young discipline and currently far away from being able to write/maintain big programs, this shows that we should not necessarily think of automatically generated programs as we do of traditionally developed software.

Machine learning algorithms have proven over recent years to work very well in the areas like image or speech recognition. These are usually some form of classification problems where the boundaries between different classes are blurred. In these areas, soft boundaries between classes are probably expected as there are image or voice samples that are hard to qualify into certain class even for a human. However, this may be a major challenge in adoption of machine learning approach to automated programming. The nature of program creation requires usage of sharp rather than fuzzy logic to implement many real-world business requirements.

The recent research in the field of automated programming focuses around three major approaches:

— program synthesis,
— program induction,
— hybrid of the two above.

Program synthesis, discussed in [1, 14], is the problem of finding the source code of a program (in some arbitrary programming language) that meets a set

MOST WIEDZY Downloaded from mostwiedzy.pl

of given constraints. The constraints may form either a full specification (deductive program synthesis, Manna-Waldinger [15, 16]) or only a partial one. Such an approach usually utilizes some form of a solver based on Satisfiability Modulo Theories (SMT, Barret *et al.* [17]) and Boolean satisfiability (SAT) algorithms which search the space of possible programs that would meet given criteria. The search space usually grows exponentially with the size of the problem. The outcome of this method is the source code which can be interpreted by humans, so its correctness may be manually verified.

Program induction tries to use machine learning methods to search for weights in a neural network such that the network provides expected output for a given training set. Usually, variants of recurrent neural networks (RNN) are used for this purpose. Such a trained neural network forms the program for which the source code is not available, so its correctness cannot be easily verified.

The abovementioned approaches have both benefits and drawbacks. Program synthesis requires searching over very big space of possible programs which may be infeasible, but produces a program whose correctness can be proven. Program induction, in turn, can be much more effective in finding a solution but there is always uncertainty whether it is correct or not. This leads to recent interest of researchers in hybrid approaches. Balog *et al.* [18] for example use machine learning to drive the search using SMT-based algorithm.

This section explores recent work related to automated construction of programs (see Figure 6).

Research direction	**Related work**		
Improve efficiency of program synthesis	Convolutional neural networks	Neural GPUs	DeepCoder
Broaden the range of possible programs that can be automatically synthesized	Long Short-Term Memory	Pointer Networks	Neural Turing Machines

Fig. 6. Research directions in the system development area.

4.1. Improve efficiency of program synthesis

Regardless the approach taken, one of the important directions of research is to make automated programming more effective. This can be achieved in a number

MOST WIEDZY Downloaded from mostwiedzy.pl

of ways from requiring less training effort (less input-output examples), through limiting the domain size to being able to search program space more accurately.

In basic RNN, the network layers are fully connected — each node in a given layer is connected to every node in the next layer. For large input sizes, the learning process of such a network proves to be inefficient. Lecun *et al.* [20] present a multi--layer neural network architecture called convolutional neural network (CNN) that reduces the number of connections and therefore allows much faster learning. It is based on the idea that particular features of the input that neural network needs to discover are local, so there is no need to connect neurons representing the parts of the input that are "far away" from each other. This allows to reduce the number of network weights to be learned significantly. Additional mechanisms for dropping nodes and connections randomly (with given probability) during network training increase training speed even more. This approach has been originally used for image classification or handwriting recognition, but Kaiser-Sutskever [21] demonstrates its usage to learn arithmetic operations on long binary representations of numbers.

An extension of the idea of local connections in neural network is the concept of attention which helps finding the parts of the input that are valuable for the learning process. With the attention-based approach, during the learning process probability distribution over the input is generated which represents the probability of information encoded in input to be valuable. This allows to increase the efficiency of learning process by focusing mostly on meaningful information. Some examples of using attention-based approach to program synthesis are described by Chen *et al.* [4] and Graves *et al.* [22].

Program 0:	**Input-output example:**	*Description*:

`k ← int`
`b ← [int]`
`c ← SORT b`
`d ← TAKE k c`
`e ← SUM d`
	Input:
	`2, [3 5 4 7 5]`
	Output:
	`[7]`
		A new shop near you is selling n paintings. You have $k < n$ friends and you would like to buy each of your friends a painting from the shop. Return the minimal amount of money you will need to spend.

Fig. 7. Example program synthesized by DeepCoder (from Balog *et al.* [18]). From the right: program description, input-output example and synthesized program in DeepCoder language DSL

Another approach to increasing efficiency of learning is to constrain the domain of programs being searched during the learning process. Balog *et al.* [18] present the DeepCoder framework targeted at automated finding programs that solve typical programming puzzles (see Figure 7). The authors assume that such puzzles are defined by a set of input-output sequences of integers and the goal is to find a correct program that transforms inputs to corresponding outputs. They introduce a domain specific language consisting of typical collection transformations

MOST WIEDZY Downloaded from mostwiedzy.pl

like sorting, extraction of first/last element, applying a mapping function to each element, etc. Then, they use neural network that determines the probability of each of the functions to appear in the target program based on the input-output pairs. This probability distribution is used to drive program synthesis using an SMT-based solver. The results show that by reducing the domain size and using the hybrid approach (program synthesis + program induction) an order of magnitude speedup in the learning process can be achieved.

4.2. Broaden the range of possible programs that can be automatically synthesized

One way of increasing accuracy of program space search could be to introduce a special structure over recurrent neural network that would make it easier for the network to discover "constructs" that are typical for traditional programming, like usage of memory, loops or conditional statements.

Hochreiter-Schmidhuber [19] describes a special kind of recurrent neural network architecture called Long Short-Term Memory (LSTM) that approaches the problem of error propagation over a series of time steps. It introduces the concept of a memory cells which constitute the hidden layer of the network and allow for much faster learning of temporal dependencies between network's inputs and outputs, even for those that occur over long periods of time. Although LSTM was introduced already in 1997, its recent successful applications in the area of speech recognition have brought it into interest of researchers and many of recently proposed network architectures used in program induction derive from these concepts.

Vinyals *et al.* [23] introduce an architecture of neural network, called a pointer network, which addresses the problem of input of variable size. In their approach, the authors focus on a couple of well-known problems involving geometry: finding a convex hull, Deluanay triangulation, and Travelling Salesman Problem (TSP). In each of those problems, the input is a set of coordinates in the Cartesian coordinate system. The novelty of this approach is that instead of representing the output, as a set of edges in the Cartesian space, they rather represent it as an ordered list of pointers to the input coordinates. Throughout learning, they use attention to pick next pointer with the highest probability. Such strategy proves to be very successful in solving problems involving path manipulation. The pointer network can provide very good heuristics for the TSP problem for input sizes up to 20 vertices, but fails both in terms of accuracy and validity of solutions for larger inputs. Still, this is an interesting approach that strives for exploiting an automated approach to programming for solving combinatorial problems.

Neural Turing Machine, described by Graves *et al.* [22], is a proposal of an architecture where a neural network is coupled with an external memory, which resembles the Turing Machine or the Von Neumann architecture. At each step,

MOST WIEDZY Downloaded from mostwiedzy.pl

the neural network reads and writes data into the memory. The attention process determines probability of certain data to be read or written. This allows the neural network to learn how to use the memory rather than simply to apply transformation over the input and opens up the research for solving problems employing the notion of state.

5. System verification

Current software development processes usually include a phase when the overall correctness of the system is verified. Most often this is done by some form of automated or manual testing. The purpose of this approach is to reveal defects that could be introduced by the software developer while encoding requirements into a program. However, in the domain of automated creation of programs' defects are of a different nature. If automatically created program does not perform as expected, then it might be caused either by learning algorithm not being trained to the necessary extent or by the neural network not being able to generalize well enough. This creates the need for humans to be able to better understand the internal mechanics of an automatically created program, which is not an easy task when neural networks are involved. Figure 8 presents some work in the field of visualizing internals of neural networks.

Research direction	Related work		
Explainable artificial intelligence	Visualizing and Understanding Convolutional Networks	Pixel-Wise Explanations for Non-Linear Classifier Decisions	Deep Taylor Decomposition

Fig. 8. Research directions in the system verification area

5.1. Explainable artificial intelligence

Automated programming can come in two flavours — in the one, the output of learning is a source code of synthesized program; in the other, the output program is only encoded in neural network's weights. At the first glance, it may seem that the source code representation gives significant advantage for being able to reason about the correctness of learned program, but in reality with large code this can easily become infeasible. Best practices for traditional programming provide guidelines for software developers to create easily understandable code, but even if we look at the

MOST WIEDZY Downloaded from mostwiedzy.pl

high quality source code of medium-sized systems, it is often very hard for a human to understand it fully. It can be expected that the code produced automatically will be even less understandable because it is not aimed at human's inspection.

An interesting research direction is the explainable artificial intelligence (XAI) which aims at providing some form of visual representation of neural network that would be relatively easy to understand for humans and would allow to reason about its correctness. Recently, a number of research papers have been published that propose a visualization of neural network internals for image classification problems (Zeiler-Frgus [24], Bach *et al.* [25], Montavon *et al.* [26]). They show how each pixel of the input image contributes to the overall image classification. Although hard to interpret, this visualization gives some insight into the neural network internals and allows a human for a coarse-grained assessment of whether the network operates as expected. It can be anticipated that this direction will gain on importance also in the domain of automated programming as it enters the areas related to human life, health, privacy or security on a larger scale.

6. Challenges

This paper presents categorization of research directions in the area of automated software development. One can, however, take a different perspective — from the point of view of the envisioned role of artificial intelligence in software development:

- *Full AI* — artificial intelligence will be capable of creating on its own any program specified by business user or by artificial intelligence itself;
- *Assisted AI* — artificial intelligence will not be able to create programs on its own; it will require human to cooperate in this process and contribute with her domain expertise;
- *Supplementary AI* — artificial intelligence will play supplementary role in the software development process; it will be able to assist in improving software quality.

Each of these visions can guide the further research and present different challenges. *Full AI* is where currently the majority of research takes place. Further development can be achieved with novel architectures of neural networks that are dedicated for solving specific classes of problems. Efficiency of learning process and accuracy of automatically created programs is another challenge. As current research focuses on rather narrow domains and synthesized programs are far less complicated than those written manually, nowadays, this vision remains rather a long term goal and it is rather unlikely that industry will widely benefit from it in near future.

Assisted AI assumes that a human is irreplaceable in the process of software development (at least until *Full AI* matures), but can still benefit from artificial intelligence. As part of the cooperation between a human and AI, AI would have

MOST WIEDZY Downloaded from mostwiedzy.pl

to be engaged in current software development processes at each phase. It could help building ontologies based on natural language specifications, detect components or help integrating them. A human, in turn, would have to supervise AI at each phase and introduce corrections to its decisions. One of the challenges would be creation of visual representation of semi-automatically created software that would help business users to reason about its correctness. Utilizing AI in multiple activities of current software development processes could lead to creation of a new process with artificial intelligence being in the center of it. This vision can be seen as an intermediate step in adoption of artificial intelligence in software development.

Supplementary AI would focus on utilizing artificial intelligence to improve current processes. Example challenges could be: increasing software reuse by detecting similar code fragments in corporate repositories, automated creation of common libraries, providing suggestions on code improvements, detecting defects, or automated creation of documentation (explanation of what program does) based on source code.

The above presented visions should not be seen as alternatives. As current research does not allow us to definitively state to which extent we are able to utilize artificial intelligence in software development, these visions can be treated as different phases of practical implementation of research with *Supplementary AI* being the short-term, *Assisted AI* — mid-term and *Full AI* the long-term and the most advanced one.

7. Conclusions

Automated software development is still a fresh research topic and remains *terra incognita*, tempting to be explored by researchers and software engineers. According to reported results, from among techniques, described in Sections 2-5, only FlashFill has found its industry application. However, the recent years have brought some important progress in this domain. The recent important advances concern novel neural networks architectures, particularly in the class of recurrent neural networks, and learning approaches. In the paper we have presented how neural networks fit into traditional software development processes and pointed out some promising research directions, dividing it into three streams of research directions with different level of hopes that they may bring into software development industry.

The study was financed from funds for the statutory activity of the Faculty of Electronics, Telecommunications and Informatics at the Gdańsk University of Technology.

The article was prepared on the basis of a paper presented at the International Conference of Software Engineering KKIO'18. Pułtusk, September 27-28, 2018.

Received July 17, 2018. Revised October 29, 2018.

Łukasz Korzeniowski https://orcid.org/0000-0001-8458-9825
Krzysztof Goczyła https://orcid.org/0000-0003-3009-8988

MOST WIEDZY Downloaded from mostwiedzy.pl

REFERENCES

[1] Gottschlich J., Solar-Lezama A., Tatbul N., Carbin M., Rinard M., Barzilay R., Amarasinghe S., Tenenbaum J.B. and Mattson T., *The Three Pillars of Machine-Based Programming*, arXiv:1803.07244v1, https://arxiv.org/pdf/1803.07244.pdf, 2018 [accessed: 25.10.2018].

[2] Kant N., *Recent Advances in Neural Program Synthesis*, arXiv:1802.02353v1, https://arxiv.org/pdf/1802.02353.pdf, 2018 [accessed: 25.10.2018].

[3] Gulwani S., *Automating String Processing in Spreadsheets Using Input-Output Examples*, PoPL'11, https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/popl11-synthesis.pdf, 2011 [accessed: 25.10.2018].

[4] Chen X., Liu C., Shin R., Song D., Chen M., *Latent Attention For If-Then Program Synthesis*, NIPS'16 Proceedings of the 30th International Conference on Neural Information Processing Systems, 2016, 4581-4589.

[5] Solar-Lezama A., *Program Synthesis By Sketching*, PhD thesis, EECS Dept., UC Berkeley, 2008.

[6] Xu X., Liu C., Song D., *SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning*, arXiv:1711.04436, https://arxiv.org/pdf/1711.04436, 2017 [accessed: 25.10.2018].

[7] Zhong V., Xiong C., Socher R., *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning*, arXiv:1709.00103, https://arxiv.org/pdf/1709.00103, 2017 [accessed: 25.10.2018].

[8] Yaghmazadeh N., Wang Y., Dillig I., Dillig T., *SQLizer: Query Synthesis from Natural Language*, Proceedings of the ACM on Programming Languages, vol. 1, Issue OOPSLA, Article no. 63, 2017.

[9] Neelakantan A., Le Q.V., Sutskever I., *Neural Programmer: Inducing Latent Programs with Gradient Descent*, arXiv:1511.04834, https://arxiv.org/pdf/1511.04834, 2016 [accessed: 25.10.2018].

[10] Reed S., de Freitas N., *Neural Programmer-Interpreters*, arXiv:1511.06279, https://arxiv.org/pdf/1511.06279, 2016 [accessed: 25.10.2018].

[11] Pavlinovic Z., Babic D., *Interactive Code Snippet Synthesis Through Repository Mining*, https://www.semanticscholar.org/paper/Interactive-Code-Snippet-Synthesis-Through-Mining-Pavlinovic-Babic/1d02c510216726dc31b2b4c9d5e0ed446d7f5c76, 2013 [accessed: 25.10.2018].

[12] Xin Q., Reiss S.P., Krishnamurthi S., *Program Repair Using Code Repositories*, https://www.semanticscholar.org/paper/Program-Repair-Using-Code-Repositories-Xin-Reiss/6c2e280585540773cca5e4e988610647d9e0f4aa, 2016 [accessed: 25.10.2018].

[13] Bornholt J., Torlak E., *Scaling Program Synthesis by Exploiting Existing Code*, https://www.semanticscholar.org/paper/Scaling-Program-Synthesis-by-Exploiting-Existing-Bornholt-Torlak/0a7879e50b69d4146cfa616d0be4bebd7bb47d41, 2015 [accessed: 25.10.2018].

[14] Gulwani S., Polozov O., Singh R., *Program Synthesis. Foundations and Trends® in Programming Languages*, 2017.

[15] Manna Z., Waldinger R., *Synthesis: Dreams => Programs*, IEEE Transactions on Software Engineering, vol. SE-5, no. 4, 1979.

[16] Manna Z., Waldinger R., *A Deductive Approach to Program Synthesis*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 2, issue 1, 90-121, 1980.

[17] Barrett C., Sebastiani R., Seshia S., Tinelli C., *Satisfiability Modulo Theories*, Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications, 825-885, 2009.

MOST WIEDZY Downloaded from mostwiedzy.pl

[18] Balog M., Gaunt A.L., Brockschmidt M., Nowozin S., Tarlow D., *DeepCoder: Learning to Write Programs*, arXiv:1611.01989, https://arxiv.org/abs/1611.01989, 2017 [accessed: 25.10.2018].

[19] Hochreiter S., Schmidhuber J., *Long Short-term Memory*, Neural Computation, 9(8): 1735-1780, 1997.

[20] Lecun Y., Bottou L., Bengio Y., Haffner P., *Gradient-Based Learning Applied to Document Recognition*, Proceedings of the IEEE. 86. 2278-2324, 1998.

[21] Kaiser Ł., Sutskever I., *Neural GPUs Learn Algorithms*, arXiv:1511.08228, https://arxiv.org/pdf/1511.08228, 2015 [accessed: 25.10.2018].

[22] Graves A., Wayne G., Danihelka I., *Neural Turing Machines*, arXiv:1410.5401, https://arxiv.org/pdf/1410.5401, 2014 [accessed: 25.10.2018].

[23] Vinyals O., Fortunato M., Jaitly N., *Pointer Networks*, arXiv:1506.03134, https://arxiv.org/pdf/1506.03134, 2017 [accessed: 25.10.2018].

[24] Zeiler M.D., Fergus R., *Visualizing and Understanding Convolutional Networks*, arXiv:1311.2901, https://arxiv.org/pdf/1311.2901, 2013 [accessed: 25.10.2018].

[25] Bach S., Binder A., Montavon G., Klauschen F., Müller K.R., Sam W., *On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation*, PLoS ONE, 10(7): e0130140. https://doi.org/10.1371/journal.pone.0130140, 2015 [accessed: 25.10.2018].

[26] Montavon G., Bach S., Binder A., Samek W., Müller K.R., *Explaining NonLinear Classification Decisions with Deep Taylor Decomposition*, arXiv:1512.02479, https://arxiv.org/pdf/1512.02479, 2015 [accessed: 25.10.2018].

Ł. KORZENIOWSKI, K. GOCZYŁA

Sztuczna inteligencja w wytwarzaniu oprogramowania — stan aktualny i wyzwania na przyszłość

Streszczenie. Od czasu pojawienia się pierwszych metod i narzędzi CASE niewiele zrobiono w zakresie automatycznego wytwarzania oprogramowania. Narzędzia CASE wspierają deweloperów w tworzeniu struktury systemu, definiowaniu interfejsów i relacji między modułami oprogramowania oraz, po powstaniu kodu, w wykonywaniu zadań testowych na różnych poziomach szczegółowości. Pisanie kodu jest jednak nadal zadaniem wykwalifikowanego specjalisty, co powoduje, że cały proces wytwarzania oprogramowania jest kosztowny i podatny na błędy. Ostatnie postępy w obszarze sztucznej inteligencji, szczególnie w zakresie metod głębokiego uczenia maszynowego, mogą i powinny znacznie poprawić tę sytuację. W artykule przedstawiono przegląd dotychczasowych osiągnięć w tej dziedzinie, znanych z literatury przedmiotu, szczególnie w zakresie czysto teoretycznym, gdyż efekty inżynierskie znajdujące zastosowanie praktyczne są jak dotąd bardzo ograniczone. Następnie zaproponowano i opisano kilka kierunków przyszłych prac w tej dziedzinie, które zaklasyfikowano jako *Full AI*, *Assisted AI* i *Supplementary AI*, w kolejności wynikającej z oczekiwanego stopnia zautomatyzowania procesów wytwarzania oprogramowania.

Słowa kluczowe: wytwarzanie oprogramowania, sztuczna inteligencja, uczenie maszynowe, automatyczne generowanie kodu

DOI: 10.5604/01.3001.0013.1464

MOST WIEDZY Downloaded from mostwiedzy.pl