**Department of Electrical, Computer, and Software Engineering**

**Part IV Research Project**

Literature Review and

Statement of Research Intent

Project Number: 77

# Are Machines Intelligent Enough to Create Programs Themselves?

Nathan Cairns

Buster Major

Jing Sun, Gill Dobbie

15/04/2019

# Declaration of Originality

This report is my own unaided work and was not copied from nor written in collaboration with any other person.

Name: Nathan Cairns

**ABSTRACT:** The ability to automate parts of software development through code generation would help increase programmer productivity and reduce software defects. In this paper, an analysis of the current literature in the domain of automatic software generation is presented. This includes an identification of popular approaches, followed by gaps in the existing research, and finally a statement of research intent based on these findings.

## 1. Introduction

Product automation is a common practice in many engineering disciplines, such as mechanical and electrical engineering. Software engineering is an industry which lacks a comprehensive set of automation processes and product quality suffers accordingly. Many software defects arise from human error and this results in programmers spending an average of 70-80% of their time doing testing and debugging [1]. Automatic code generation would hopefully reduce the amount of time spent fixing code and reorient developers to be focused on designing and developing products. This paper presents a comprehensive review on the current literature surrounding the field of automatic code generation and summarizes with a statement of research intent. It starts by outlining the current approaches and tools described in academic literature and compares them. Following this, any gaps that were identified in the research domain are brought to attention. Finally, a statement of research intent is declared from the findings and gaps identified in the previous sections.

## 2. Approaches

In this section several approaches are outlined as described in the existing automatic software generation academic literature.

### 2.1. Machine learning

Machine learning is a popular method for generating code from some an input specification. This is generally done by using supervised learning to output a program line by line based on I/O examples and their corresponding programs [2]. In machine learning a large amount of labelled data is required. This means that data should have both what is expected to be inputted to the system and the corresponding output which is expected. There are two main approaches to automatic program learning. These are program synthesis and program induction [3].

#### *2.1.1. Program synthesis*

Program synthesis is done by teaching a neural network to generate a program based on I/O training data [3]. This is a difficult program generation technique especially with regards to generating complex programs. For example, in [4] a lot of success was seen when generating very simple programs from I/O snippets taken from programing competition websites. This was done through combining neural network architectures with search-based techniques, rather than trying

to replace them. i.e. using gradient descent to search for the correct program. [5] Does something similar using machine learning techniques to train a program synthesizer to understand I/O examples (encoding) and to search a program space (decoding). Search approaches are further discussed in 2.2.

#### *2.1.2. Program induction*

Program induction is done by training a neural network to be a latent program representation that provides program output based on I/O training data [3]. This is a popular type of generation which has seen a lot of success. For example, in [6] they proposed a well performing generative model for code generation that worked to create code that generates representations of trading cards. However, since this form of program generation doesn't generate a program but a latent representation of a program this is most likely something we will not pursue further.

#### *2.1.3. Reinforcement learning*

Reinforcement learning is another type of machine learning where a mapping from situations to actions is learnt so that a scalar reward or reinforcement signal is maximized. The learner is not told which action to take, but instead must discover which actions yield the highest reward by trying them [7]. Reinforcement learning is not used much in program generation, however, based on how it is used in other applications it may be possible to be appropriated to work for generating programs. For example, in [8] a set of instructions is taken and transformed into executable instructions. This is done by using reinforcement learning on Microsoft Windows troubleshooting guides and generating the valid actions to complete the steps in the guide. Whilst this is not the express generation of a program it may be possible to generate a program using a similar method. A program is after all simply a sequence of executable instructions.

#### *2.1.4. Datasets*

A very important part of any application of machine learning is the dataset. Some important things to consider is where the data has come from, how the data will be represented, and how the data is labelled. The most common form of data used in machine learning approaches for code generation is program I/O coupled with the corresponding programs [3], [4]. However, some papers use more novel datasets such as trading cards [6]. Obtaining datasets was generally done through defining a Domain Specific Language (DSL) and then using this DSL to generate valid programs and their corresponding I/O [3], [4], [9]. For the reinforcement learning approach windows troubleshooting guides and puzzle game tutorials were used as a dataset [8].

### 2.2. Search

Search is probably the most popular method of generating programs. In a search approach a program space is searched until the most appropriate program is found [2]. There are 4 main types of search [5]:

1. Enumerative: Conducts a search using a tree pruning approach.
2. Stochastic: Conducts a search using a probabilistic function approach.
3. Constraint Based: Conducts a search by encoding a problem in SAT/SMT solvers.
4. Version Space Algebra: Conducts a search by using DSLs and performing a divide-and-conquer search.

In the following sub-sections enumerative search and constraint-based search are discussed.

#### *2.2.1. Enumerative search*

Enumerative search involves enumerating an entire problem space and then using tree-based searching and pruning to narrow down the most appropriate result to a query. In terms of program generation this means searching over all the possible programs and choosing which one best fits an inputted specification. One popular application of enumerative search in program generation is called Sketching. Sketching is where a programmer provides a partial implementation of a program and the model synthesizes the remainder [10]. Sketching is a good method of searching as it helps significantly reduce the search space [11]. SKETCH is a popular language used for implementing sketching [4], [10], [11]. Another commonly used tool for enumerative search-based program synthesis is $\lambda^2$ [4], [12] . $\lambda^2$ combines enumerative search with deduction to prune the search space. It an be used to infer small functional programs for data structure manipulation from I/O examples, by combining functions from a provided library [4]. Enumerative search is an approach that works well but is somewhat limited in scope. It is better kept to specific domains and therefore smaller search spaces. For example, [13] uses enumerative search to generate search algorithms with the best combination of domain specific heuristics. Whilst [13] presents a very useful application it shows the limits of an enumerative search approach with regards to scope.

#### *2.2.2. Constraint based / machine translation*

Machine Translation (MT) tools are computer programs which are capable of automatically producing in a target language the translation of a text in a source language [14]. If we consider a programming language to be like a natural language with its own grammar and rules, then we can see the possible application in translating natural language to code. The development of a MT tool able to transfer natural language to code is extremely desirable as it would allow less experienced programmers to easily generate code from specifications.

Using MT for program synthesis is a complicated method with little successful study contributed. In [15] a Statistical Machine Translation (SMT) framework is used to translate source code into pseudo code. From this study the following question arises: Can a similar SMT framework be used to translate pseudo code into source code? If this avenue were to be explored the extensive dataset of manually annotated source code produced in [15] could potentially be used. The open source SMT toolkit Moses [16] could be used to develop such a tool. In [17] a Java IDE extension was developed using SMT and natural language processing tools which accepts as input free-form queries containing a mixture of English and Java, and synthesizes a list of ranked (possibly partial) Java code expressions. A basic search method was used that simply matched keywords to the API database stored in the IDE. However, this simple approach led to the tool described in [17] only being able to produce expressions excluding local variable declarations. i.e. unable to generate loops or conditionals. This problem was addressed in [18] by adding a required unit test to the input specification. However, this reduces the ease of use by requiring additional specialized input.

The use of a Long Short-Term Memory (LSTM) was used in [19] to solve general sequence problems. They found that large deep LSTMs with a limited vocabulary can outperform a standard SMT-based approach whose vocabulary is unlimited on a large-scale MT task. This suggests an LSTM architecture could be appropriate if we choose to use a MT approach.

### 2.3. Rule based

Rule based program generation is a naive yet effective approach to generating programs. This approach involves defining a set of rules and then simply generating programs from a well-defined specification language. These rules can be learnt via machine learning [20] or can be manually defined [9]. Defining a set of rules can be very time consuming. Therefore, machine learning provides an alternative to generating a rule set for generating code. However, the only successful implementations of this approach require a very formal specification language. For example, in [20] Real-Time Process Algebra (RTPA) is used as a source language. RTPA is a very formal language which requires a high-level of understanding. This creates a barrier to usage and diminishes the utility of a code generator. Using machine learning to generate rules ties in closely to Machine Translation as described in 2.2.2.

## 3. Gaps in research domain

The gaps in the research domain are as follows.

### 3.1. Input specification

One big gap in the research domain is using different input specifications for the generate program. Currently a lot of work in the field uses I/O as an input specification, regardless of the approach [3], [4], [21]. This is most likely so prevalent

as writing a formal specification is often harder than writing a program itself and because it allows us to validate the semantic correctness of our generated program [3]. Regardless of input specification used program I/O is likely to be necessary to validate the produced program. Whilst there are definite advantages to using program I/O there are also several problems. For example, as described in [4], in practice there is often only a small amount of I/O examples and they often have values which place a high information burden on them. This can restrict I/O models to creating only very simple programs. For these reasons it would be good to explore this gap in the research domain. More novel approaches could be taken such as that described in [15] where source code is translated to pseudo code (the inverse of this would be done to achieve program synthesis) or that described in [9] where machine learning models are generated from graphs in academic research papers. An even more comprehensive approach could be taken like that described in [22] where a task set specification is used to generate programs. A task set specification is a diagram which show the flow of data between functions and algorithms. It is important to note however, that a balance must be found between the simplicity and effectiveness of an input specification. For example, [20] produced a very effective model for code generation, however, the input specification used a very complicated syntax. This negatively affects the usability of the final product and hence diminishes its real-world effectiveness. The gap here is to identify and use an input specification which is an alternative to program I/O and more usable than the more formal existing alternatives.

### 3.2. Program simplicity

Another significant gap in this research domain is the simplicity of the programs produced by existing models. Currently most models are only effective at generating the simple programs [4], [17], [18]. Hence a gap to be filled would be the development of a model able to produce more complex programs. It is important to note that this is an extremely hard problem to solve and it is unlikely to be achievable with the current resources and time frame we have.

## 4. Statement of research intent

The intent of research as gathered from the relevant literature reviewed in the previous sections is as follows: My project partner and I intend to research the feasibility of using a program synthesis machine learning approach to create a model which can generate simple functional programs from an alternative input specification to program I/O. We believe through doing this we can make a reasonable contribution to the field of program generation.

The high-level approach we will take start by identifying an input specification language and mapping it to an output programming language. At this point we have decided to use Python for our output language and are yet to decide on our input language. We will then programmatically generate a dataset from our input specification which can be used to train a machine learning model which will be able to transform our input specification to Python. We will then evaluate

the generated Python code by checking semantic equivalence. This high-level design can be seen in Fig. 1. A week-by-week plan of how we will accomplish this implementation can be seen in Fig. 2.

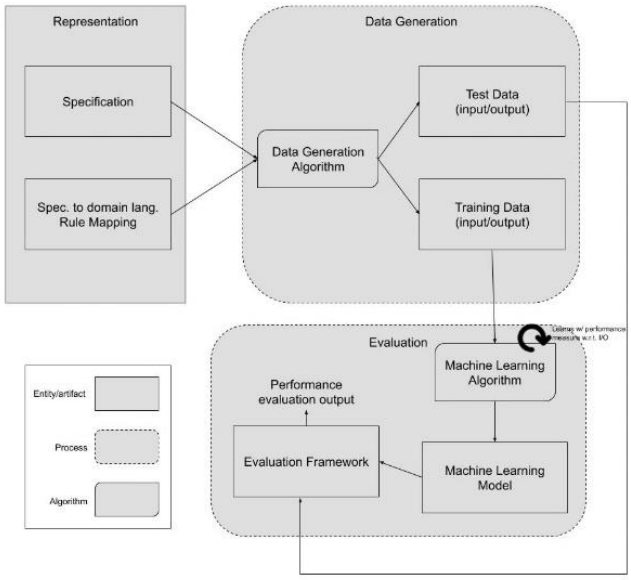


Fig. 1. Graph showing a high-level design of the system

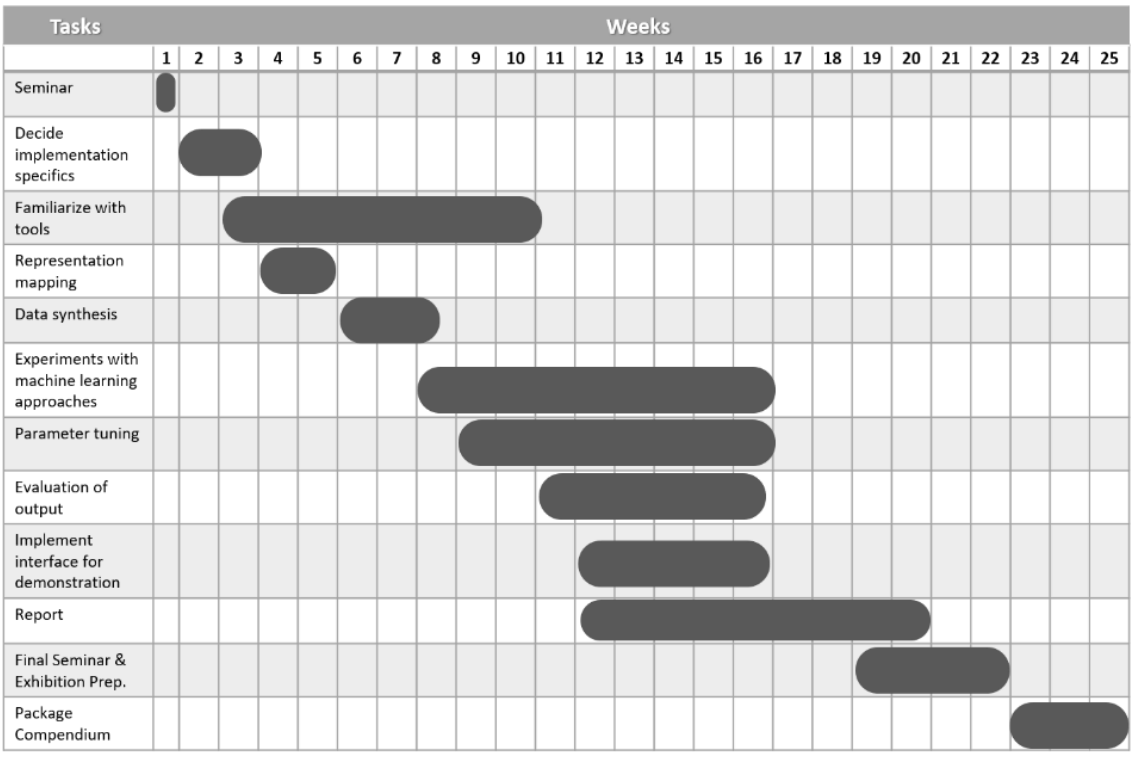


Fig. 2. Gant chart showing project plan

## 5. Conclusions

In this work, a comprehensive literature review of the field of automatic code generation was undertaken. Firstly, several common approaches to code generation were identified. This included approaches using machine learning, searching, and rules. Next, two major gaps in the research were identified. These were a lack of diversity in input specification and the simplicity of programs that can be generated. Finally, from the approaches and gaps identified a statement of research intent was declared.


## Acknowledgements

I would like to thank the following for their contributions to this paper:

- Buster Major for being a great project partner and helping me analyze and digest the research domain reviewed in this document.
- Jing Sun for being a supportive supervisor providing invaluable guidance with regards to academic research.
- Chenghao Cai for providing great explanations and advice with regards to machine learning techniques.


## References


[1] F. Huang, B. Liu and B. Huang, "A taxonomy system to identify human error causes for software defects," in 2012, .

[2] R. Simmons-Edler, A. Miltner and S. Seung, "Program Synthesis Through Reinforcement Learning Guided Tree Search," 2018. Available: https://www.openaire.eu/search/publication?articleId=od_____18::7ba76007ea35a4ae2c68f0ba74eee539.

[3] J. Devlin *et al*, "RobustFill: Neural Program Learning under Noisy I/O," 2017. Available: https://www.openaire.eu/search/publication?articleId=od_____18::65ae497a4954be82d8ab7708ae0972dc.

[4] M. Balog *et al*, "DeepCoder: Learning to Write Programs," 2016. Available: https://www.openaire.eu/search/publication?articleId=dedup_wf_001::120bc62d3d89ad268c05e4667afd0f28.

[5] R. Singh and P. Kohli, "AP: Artificial Programming," 2017.

[6] W. Ling *et al*, "Latent Predictor Networks for Code Generation," 2016. Available: https://www.openaire.eu/search/publication?articleId=od_____18::1c92d601c8b8ac76d46a09c2d1132d6a.

[7] R. Sutton, *Reinforcement Learning.* 1992.

[8] S. R. Branavan *et al*, "Reinforcement Learning for Mapping Instructions to Actions," 2009. Available: https://www.openaire.eu/search/publication?articleId=od_____88::2945bae4ce6bd3029eea7167a72109e4.

[9] A. Sethi *et al*, "DLPaper2Code: Auto-generation of Code from Deep Learning Research Papers," 2017. Available: https://arxiv.org/abs/1711.03543.

[10] A. Solar-Lezama *et al*, "Combinatorial sketching for finite programs," *ACM SIGARCH Computer Architecture News*, vol. 34, *(5)*, pp. 404, 2006. . DOI: 10.1145/1168919.1168907.

[11] R. Alur *et al*, "Syntax-guided synthesis," in Oct 2013, Available: https://ieeexplore.ieee.org/document/6679385. DOI: 10.1109/FMCAD.2013.6679385.

[12] J. K. Feser, S. Chaudhuri and I. Dillig, "Synthesizing data structure transformations from input-output examples," *ACM SIGPLAN Notices*, vol. 50, *(6)*, pp. 229-239, 2015. . DOI: 10.1145/2813885.2737977.

[13] S. Minton and S. R. Wolfe, "Using machine learning to synthesize search programs," in 1994, Available: https://ieeexplore.ieee.org/document/342680. DOI: 10.1109/KBSE.1994.342680.

[14] T. Poibeau, *Machine Translation.* MIT Press, 2017.

[15] Y. Oda *et al*, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in Nov 2015, Available: https://ieeexplore.ieee.org/document/7372045. DOI: 10.1109/ASE.2015.36.

[16] P. Koehn *et al*, "Moses: Open Source Toolkit for Statistical Machine Translation," 2007. Available: https://www.openaire.eu/search/publication?articleId=od_____3094::d619d791a9c5f08ec8937e1a1b36e33b.

[17] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," *ACM SIGPLAN Notices*, vol. 50, *(10)*, pp. 416-432, 2015. . DOI: 10.1145/2858965.2814295.

[18] A. Cozzie and S. T. King, "Macho: Writing Programs with Natural Language and Examples," Urbana-Champaign, 2012.

[19] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," 2014. Available: https://www.openaire.eu/search/publication?articleId=od_____18::ada95bfac7eb090942649e38bdabed8c.

[20] J. Y. Xu and Y. Wang, "Towards a methodology for RTPA-MATLAB code generation based on machine learning rules," in Jul 2018, pp. 117-123.

[21] J. K. Feser, S. Chaudhuri and I. Dillig, "Synthesizing data structure transformations from input-output examples," *ACM SIGPLAN Notices*, vol. 50, *(6)*, pp. 229-239, 2015. . DOI: 10.1145/2813885.2737977.

[22] T. E. Smith and D. E. Setliff, "Towards an automatic synthesis system for real-time software," in 1991, pp. 34-42.