Code Synthesis in Self-improving Software Systems

Roberto Rodrigues Filho
Lancaster University
Lancaster, UK
r.rodriguesfilho@lancaster.ac.uk

Alexander Wild
Lancaster University
Lancaster, UK
a.wild3@lancaster.ac.uk

Barry Porter
Lancaster University
Lancaster, UK
b.f.porter@lancaster.ac.uk

***Abstract*—This paper aims at provoking a wide discussion around code synthesis and its importance in creating the next generation of self-improving software systems. As a starting point for the discussion, a machine-to-machine self-improving framework is presented. The framework aims at improving system's performance by integrating two modules: i) a self-improving online module, and ii) a code synthesiser offline module. The online module learns, at runtime, as it handles the system's inputs, how to best compose the system from a pallet of available software components and a user-defined high level goal. The offline code synthesiser generates new components based on the perceived system's input, executing environment and the system's goal provided by the online module. The code synthesiser then provides better component options for the online module to integrate with the system to improve its performance. The paper describes the framework, focusing on its main challenges.**

***Index Terms*—code synthesis, self-improving systems**

I. Introduction

Contemporary systems are becoming more complex, with increasing size, surpassing millions of lines of code running on highly distributed, dynamic and heterogeneous operating environments. These systems are soon to be beyond human capacity to fully understand and to appropriately manage them to maintain the current levels of quality users demand. Self-integrating and self-improving systems aim at addressing these concerns by pushing more responsibility to the system itself, enabling systems to integrate newly added components, and self-improve with a reduced level of human interference. A key limitation of self-improving systems is that the entire system, including the newly added pieces, is written by experts, who put great effort into carefully considering technical details to engineer new components to improve the system's performance. As a step beyond the state-of-the-art of self-improving systems to enable them to properly handle contemporary systems complexity, it is necessary to further push the responsibility of handling frequent and low-level technical tasks, such as the creation of new components to improve systems, to the machine itself, and to transform the role of developers from dealing with **technical details** to defining **systems intent**. This paper presents code synthesis as a way to realise this vision and to create the next generation of self-improving systems.

We centre the discussion around the application of code synthesis to maximise the potential of self-improving systems. Our discussion is mainly on the challenges in creating code synthesisers and integrating them to online self-integrating, self-improving systems. We propose a framework to show how these two elements connect and to illustrate the potential of this resulting machine-to-machine loop in improving systems performance and reducing even further human interference.

This paper brings a discussion on the challenges of implementing a functioning version of the proposed framework. The challenges range from techniques to define which data to collect from the online system to better assist code synthesisers in creating high quality code, to challenges in code synthesis process itself, such as addressing the large search space for possible programs and the lack of a precise way to navigate through the search space of possible programs.

II. Approach

Our approach envisions the integration of two main modules: i) the online self-improving module, and ii) the offline code synthesiser. The online self-improving module has been implemented in previous works [1]–[3] referenced as Emergent Systems framework. This module is responsible to assemble fully functioning systems from a pallet of small software components, and reassemble the system by swapping these components at runtime. The online module enables the system to learn, at runtime, with no predefined domain-specific knowledge and using reinforcement learning techniques, how to best compose the system according to the operating environment and high level system goals. This module has been implemented and extensively experimented with in the context of web-based applications in data centres [3]. The offline code synthesiser, however, is the novel part of the proposed framework. Inspired by recent advances in code synthesis with techniques ranging from genetic improvement [4] to neural networks [5], we envision the code synthesiser as a key element to advance self-improving systems' state-of-the-art.

The code synthesis module contains multiple code synthesisers focused on generating code for different set of components. Depending on the component, different code synthesis strategies should be applied. For example, for components that are generated from improving existing components, such as cache components, which are generated by improving their hash function, techniques such as genetic improvement has been shown effective [4], whereas the synthesis of entirely new components may use neural networks as described in [5].

The integration of two modules and the machine-to-machine loop happens when the online self-improving module feeds data about the running system and its operating environment to the code synthesis module, which in turn uses the provided data to synthesise a set of new components. The generated components are then made available for the self-improving module to use in production and further improve the system's performance. The self-improving module has to collect data from the system to assist the reinforcement learning algorithm to find the most suitable software components, and also to assist the offline synthesisers to generate high quality components that would actually improve the system's performance. The data used to classify the operating environment and the system's input are common for the two modules, but the code synthesisers may require further data, such as the specific input for the target component. As the self-improving system runs and collects more data, the better the quality of code produced by the synthesisers, which moves the system forward towards its optimal composition for the given operating condition in a complete autonomous fashion.

III. Challenges

There are several challenges that have not yet been fully investigated in the self-improving module. In [2] there is a list of challenges to implement this module, with problems ranging from classifying operating environments to rapidly navigating through a large search space of possible software compositions. These challenges are listed and discussed in [2] and [3]. This section, however, focuses on the challenges of integrating the existing self-improving module with a code synthesiser, and of the code synthesis process itself.

The main challenge related to the self-improving module is the collection of data from the live system to be used in the code synthesiser. This challenge unfolds in two sub-challenges: i) controlling the volume of data to be collected, and ii) the definition of which data to collect. There is a trade off between impacting the system performance, and the quality of code synthesised when controlling the volume of collected data. Large volumes of data often lead to better quality of code produced by the synthesiser, and at the same time, the collection of high volumes of data from a live system highly impacts the system's performance [7]. Balancing this is extremely challenging, especially when synthesising components for multiple parts of the system.

The definition of what data to collect has a direct impact on the quality of synthesised components, mainly because the data dictates how the synthesiser optimises the generated code. If any important feature of the environment or the input being handled by the component is not collected, the produced components will not perform as desired. In fact, the fabricated component is likely to perform even worse than previously available ones which instead of improving the system, adds more useless possible compositions for the online module to learn. Furthermore, defining the data to be collected also impacts the volume of data collected. For a big system with many components, it is imperative to only focus on a subset of specific components that are more likely to have more impact on the system's performance, instead of collecting information from the entire system. Autonomously detecting such components is also a very challenging and important task.

For the code synthesis module, considering code synthesis of entire new component, presents the following main challenges: i) a large search space, and ii) that the search space has no natural fitness/loss landscapes. Firstly, even with constraints put on the language to fabricate new components, in terms of limiting program length and limiting cyclomatic complexity, the search space grows exponentially with each line, with the number of valid program implementations representing a small fraction of the total space size. Finally, [6] suggests that program space has no natural fitness/loss landscape. Unlike in neural network training, where a gradient can be followed from an initial condition to a low-loss point in parameter space, no program synthesis technique has found a followable gradient. Genetic programming relies on there being a way to move from regions of low fitness to regions which contain the maximal fitness solution, but there is no guarantee this is possible for every component to be synthesised.

IV. Conclusion

This paper presented a framework connecting a code synthesis module with self-improving and self-integrating module as a possibility for the next generation of self-integrating and self-improving systems. The framework consists of a machine-to-machine loop between the online self-improving system, which learns which systems composition maximises the satisfaction of the systems goals, and the offline code synthesis module that produces new components for the online module to integrate to the system to improve its performance. We also presented some challenges in implementing such framework that ranges from the difficulty in defining which data to collect from the system to the innate difficulties in synthesising software code. Finally, we hope the community to find the discussed challenges important and to join us in the effort of advancing code synthesis techniques to be applied in self-integrating and self-improving software.

Acknowledgement

This work was supported by the Leverhulme Trust Research Project Grant *The Emergent Data Centre*, RPG-2017-166.

References

[1] B. Porter, M. Grieves, R. Rodrigues Filho, D. Leslie, "REx: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems", Symposium on Operating Systems Design and Implementation, November 2016.

[2] B. Porter, R. Rodrigues Filho, "Losing Control: The Case for Emergent Software Systems using Autonomous Assembly, Perception and Learning", International Conference on Self-Adaptive and Self-Organizing Systems, September 2016.

[3] R. Rodrigues Filho, B. Porter, "Defining Emergent Software using Continuous Self-Assembly, Perception, and Learning", Transactions on Autonomous and Adaptive Systems, September 2017.

[4] C. McGowan, A. Wild, B. Porter, "Experiments in Genetic Divergence for Emergent Systems", Int. GI Workshop, June 2018.

[5] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow, "Deepcoder: Learning to write programs", arXiv:1611.01989, 2016.

[6] J. Renzullo, W. Weimer, M. Moses, and S. Forrest. "Neutrality and epistasis in program space", Int. GI Workshop, June 2018.

[7] D. Ardelean, A. Diwan, C. Erdman, "Performance analysis of cloud applications", In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2018.