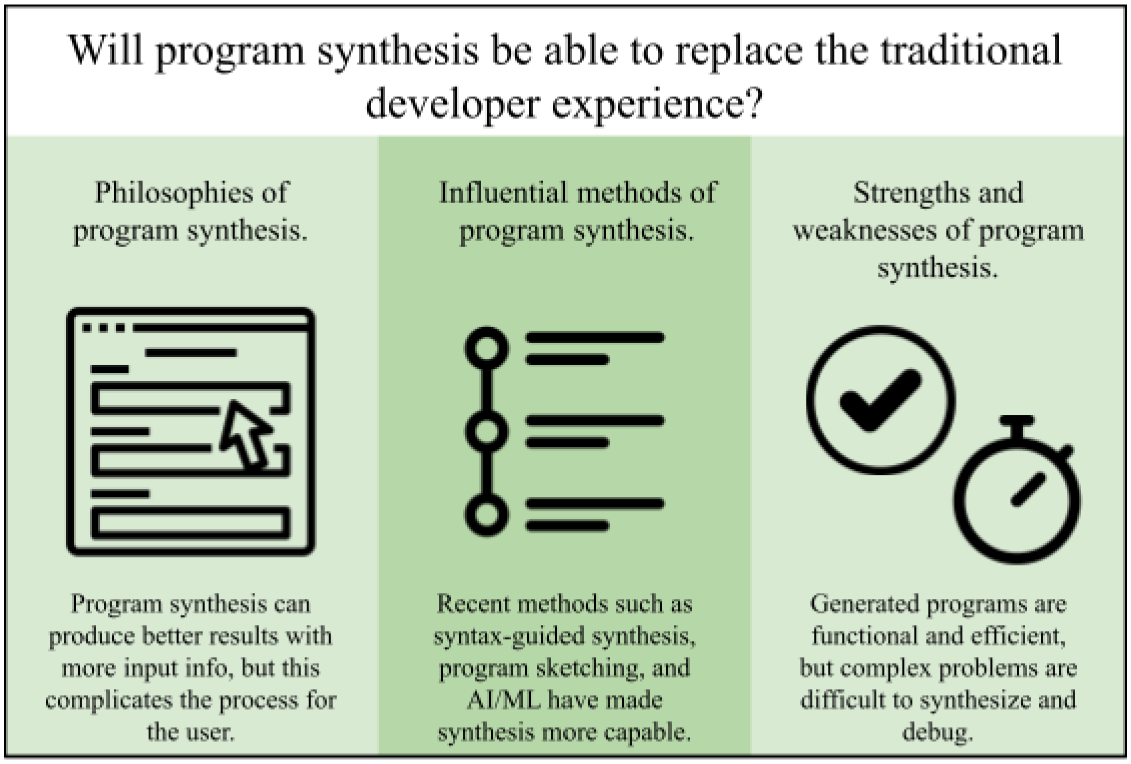
Program Synthesis: The Future of Computer Science?

Wyatt Cowley

Brigham Young University

Abstract

Program Synthesis: The Future of Computer Science?



Will program synthesis be able to replace the traditional developer experience?

Philosophies of program synthesis.

Program synthesis can produce better results with more input info, but this complicates the process for the user.

Influential methods of program synthesis.

Recent methods such as syntax-guided synthesis, program sketching, and AI/ML have made synthesis more capable.

Strengths and weaknesses of program synthesis.

Generated programs are functional and efficient, but complex problems are difficult to synthesize and debug.

Keywords: program synthesis, computer science, generated code, programming by example, automating coding

Program Synthesis: The Future of Computer Science?

One of the historical holy grails in computer science is creating programs that will generate code for other programs, automating the coding process. Program Synthesis is a field of study to achieve this goal. Traditionally, program synthesis has three parts: the user input, program space, and search technique. User input takes many forms—such as input/output examples, textual descriptions, or partially completed programs—but together, they must sufficiently describe the program so the program synthesis model can implement it. Program space is a collection of functions the model can use to formulate its solution. The final part, the search technique, tests various permutations of the functions in the program space until the model finds a solution that matches the user's specifications.

This literature review addresses the question of whether program synthesis will eventually be able to replace the traditional developer experience. Although program synthesis has improved dramatically over recent years, program synthesis will not be able to perform at a level similar to a human developer for the foreseeable future. Thus, program synthesis will probably not replace developers but will instead aid them while coding.

In the past year, the public has adopted various forms of AI for use in areas like search and content generation. Similar to program synthesis, recent products like OpenAI's ChatGPT and Codex promise the possibility of revolutionizing the programming developer experience. However, before pivoting away from program synthesis to these new technologies, it is crucial to fully understand program synthesis and its various approaches to generating code. This paper discusses some of the common philosophies found in program synthesis. Then, it describes influential methods used to enhance program synthesis and concludes whether program synthesis can feasibly change or replace the standard developer experience by laying out the general strengths and weaknesses of the field.

Philosophies of Program Synthesis

Program synthesis is a relatively new discipline. It has been used in several viable products (like Excel's FlashFill) but is primarily in the research stage. There are many different philosophies and competing viewpoints on the goals of program synthesis and how it should assist in software development. These philosophies include how much user input should be required, how much the user will understand the synthesis process, and how the synthesizer will be used. These philosophies are represented in three different viewpoints: synthesis-assisted development, synthesis-driven development, and synthesis-based development.

The **synthesis-assisted development** vision is for synthesis to be a copilot in the coding process, assisting in implementation details and providing ideas for the code that a developer writes. In this vision, the user is highly involved in providing specification details and guiding the synthesis process. For example, Solar-Lezama (2013) discussed what he termed program sketching. Program sketching is a synthesis technique where the user inputs an incomplete program and input/output examples. The synthesizer takes the almost complete program and finishes it using the input/output examples as guides. Building on this idea, Ferdowsifard et al. (2021) introduced a synthesizer, LooPy, that runs within a standard code editor, which synthesizes and autocompletes parts of a program when prompted by input/output examples. Anecdotal evidence gathered after testing LooPy with amateur programmers suggests it helps users solve coding challenges that use basic control structures. Ferdowsifard et al. (2021) show that the model of program synthesis introduced by Solar-Lezama (2013) is feasible to implement and provides advantages over traditional coding.

Synthesis-assisted development can take other forms. Zhang et al. (2021) show how continuous human involvement in the synthesis process can make it more effective. Often, program synthesis is done in a black box, meaning that users do not see the internal implementation or process the synthesizer takes to formulate a program. Zhang et al. (2021) built a different type of synthesizer that relies on human interaction to manually prune the search field and guide it to a solution that satisfies the input/output examples provided. Both Ferdowsifard et al. (2021) and Zhang et al.'s (2021) programs help complete standard coding programs; however, the user needs to have a general idea of the solution to their program to take advantage of their approaches. Thus, while synthesis-assisted development is helpful in software development, it is not currently accessible to non-coders.

Synthesis-driven development also requires technical knowledge from the user, but the synthesizer plays a more independent role than in synthesis-assisted development. Synthesis-driven development requires additional context beyond input/output examples to synthesize a solution. For example, Jain et al. (2022) recently created a synthesizer dubbed Jigsaw that relies on large language models to generate code and synthesis techniques to validate that code. The large language model they used required a natural language description of the task at hand, and the synthesis techniques required input/output examples. Syntax-guided synthesis, or SyGuS, is another example of this vision. In SyGuS, the synthesizer receives input/output examples describing the task and specific functional grammar that it can use in a solution (Alur et al., 2013). Both of these methods require two different forms of input to make their synthesis more efficient. Because of the added information they require from the user, these models can generally synthesize more complicated problems, making them more able to replace traditional coding.

The final vision of program synthesis, **synthesis-based development**, requires minimal information from the user. This vision calls for purely inductive synthesis, where the user only gives the synthesizer input/output examples for the desired program. Feser et al. (2015) developed a synthesizer that works this way. Feser et al.'s (2015) system analyzes the programming type (such as integers, arrays, or booleans) of the inputs and outputs to hypothesize about the program's structure. Feser et al.'s (2015) system uses this hypothesis to speed up searching for a program that satisfies the given input/output examples. Synthesis-based development is the vision most accessible to non-coders; however, it is the least sophisticated and cannot currently generate complex programs.

To conclude, synthesis-assisted, synthesis-driven, and synthesis-based development have different aims and capabilities. Both synthesis-assisted and synthesis-driven development require technical knowledge of computer science to use effectively. Users with no technical background can only easily use synthesis-based methods, though synthesis-based development is the least capable of the three visions discussed. Taken together, these philosophies suggest that program synthesis may revolutionize the coding experience by supporting the programmer, but to aid in any complicated program, a high degree of understanding is still required of the user.

Different Methods of Program Synthesis

The differing philosophies of program synthesis give a baseline for speed and efficiency based on the vision used, and many papers are released each year detailing new ways to improve the synthesis process across the board. Some impactful methods to improve program synthesis include syntax-guided synthesis, program sketching, and AI/ML.

One of the main barriers to program synthesis is that the available program space (or the different functions the synthesizer can use) is incredibly vast, making it challenging to synthesize complex programs with many components. In 2010, Gulwani (2010) discussed restricting the available functions to domains such as bit-vector manipulations or regex expressions to limit the program space. Alur et al. (2013) expanded upon this work by defining a standard way to provide a synthesizer with a limited set of grammar or operations to formulate its solution. Alur and fellow researchers termed their approach **Syntax-guided Synthesis** or SyGuS and instituted a yearly competition for synthesizers implementing SyGuS. This yearly competition allowed researchers to show off their state-of-the-art systems without writing a whole paper and inspired competition in the program synthesis space.

The method Alur et al. (2013) introduced quickly became a guiding force in program synthesis. In years since, many researchers have built their synthesizers based on SyGuS, including Alur et al. (2017) and Lee et al. (2018). Alur et al. (2017) increased the functionality of Alur et al.'s (2013) system by introducing a divide-and-conquer approach that attempted to solve multiple smaller problems, eventually combining them into a final program that satisfied the specifications. They called this system EUsolver. Lee et al. (2018) built directly on top of EUSolver, further optimizing it to create the synthesizer Euphony, which is faster and more successful than EUSolver and the original implementation from Alur et al. (2013). These three studies show how guiding syntax helped to spur innovation in the field of program synthesis.

Program sketching also helps to guide the synthesis process, albeit in a different way. Researchers introduced program sketching to complete partial programs that the user gave the system, but it has since become an intermediate step in many different synthesizers. As discussed earlier, Solar-Lezama's (2013) paper introduced program sketching, where the user inputs an incomplete program and relies on the synthesizer to implement some of the details. This idea has been incorporated into other synthesizers to speed them up. Feser et al. (2015) detail a way to use sketches to speed up the synthesis project. Although the user only enters input and output examples into the program, the synthesizer uses the type and format of the input and output to create a sketch—or incomplete program—of the solution. Then it completes that sketch through methods discussed in Solar-Lezama's (2013) paper. Similarly, Liu et al. (2019) describe another way to generate a sketch to expedite program generation. By mining popular code repositories like Github, researchers developed a probability map based on what APIs (Application Programming Interfaces) or functions developers use next to each other. From this usage data, Liu et al.'s (2019) model generated a sketch of what they expected the final program to look like and then finished that sketch using the input/output data provided by the user. These two studies—Feser et al. (2015) and Liu et al. (2019)—used the ideas put forth by Solar-Lezama's paper to further enhance the speed of their synthesis.

SyGuS and program sketching show how methods introduced into the program synthesis field have impacted research efforts. These methods are often not mutually exclusive and can be used in tandem to produce better results. FrAngel, built in 2019, is a component-based synthesizer, meaning it receives various functions and grammar to generate a solution, similar to syntax-guided synthesis (Shi et al., 2019). To create complex control structures such as for-loops, FrAngel creates angelic structures, similar to program sketches, that test the general structure of the solution. After finding a promising general structure, FrAngel completes the angelic structure by adding in all the missing components. By using principles of both syntax-guided synthesis program sketching, FrAngel can synthesize more than these methods individually. Shi et al. (2019) report that FrAngel solved 91.6% of provided problems involving control structures, whereas implementations only using SyGuS solved around 65% and those only using program sketching solved around 50%. This data signifies that as more advances are made in the program synthesis space, they will continue to build on each other, likely leading to exponential growth in synthesis capability.

In recent years, using **artificial intelligence and machine learning** has become increasingly more viable for program synthesis applications, overcoming weaknesses of non-AI/ML-based models. Many state-of-the-art synthesizers rely on the user to provide functions and an explanation of what each function does. While these explanations help the synthesizer make more educated guesses of how to formulate a solution, they make it more difficult for the user to use a synthesizer quickly. Lee et al. (2022) use machine learning to eliminate this need. Before trying to generate a complete solution, their machine learning model differentiates between the functions provided and learns their purpose. The ML model then passes this knowledge into the synthesizer, which uses this information to efficiently generate a valid program. Another common weakness of program synthesis is that solutions cannot be long or complex, as the time to search for solutions grows dramatically with the solutions' size. To address this issue, Jain et al. (2016) implement the actual code generation with a large-language model, a form of artificial intelligence. Large-language models, such as OpenAI's ChatGPT or Codex, can quickly generate code given a natural language prompt, but they often make errors in logic or syntax. Jain et al.'s (2016) method fixes these errors by using synthesis validation techniques on top of the large-language model to fix issues in the generated code and end with a valid program that satisfies all test cases. Both of these synthesizers addressed one of the critical weaknesses of program synthesis by using AI/ML. It is likely that further implementing AI/ML into the synthesis process will help address other weaknesses of program synthesis.

Historically, SyGuS and program sketching have increased the rate of improvement in program synthesis. Today, new AI/ML methods show significant promise in growing the complexity of solvable problems. That said, these new methods are unlikely to solve the complexity for the user but shift it to other areas—strengthening the thesis that program synthesis will require informed developers.

Strengths and Weaknesses of Program Synthesis

The methods and philosophies of program synthesis discussed above have different capabilities and advantages, but many implementations share similar essential traits. In this section, the general strengths and weaknesses of program synthesis will be discussed to help evaluate how it might impact the field of software development.

Program synthesis has two clear strengths over other systems that generate code: solutions can be verifiably proven and are often short and simple. Assuming user input is accurate, programs generated by program synthesis are guaranteed to function as expected. Program synthesis relies on basic specifications, like input/output examples, to validate a program it creates. The synthesizer will continue to run and attempt to generate different programs until the solution performs as expected on all specifications (David & Kroening, 2017). If a solution can be reached with program synthesis methods, it is guaranteed to act as expected, which cannot be said for code generated solely by large language models or code written by human developers. As stated in Jain et al.'s (2022) paper, "...PTLMs [pre-trained large language models]… treat code as text. Consequently, the code produced by such models has no guarantees of correctness or quality." Clearly, an advantage of program synthesis is that code generated is guaranteed to run correctly.

Program synthesizers often find short and simple solutions because of their search method. Alur et al. (2013) discuss how when their synthesizer searches for a solution, it attempts every solution at size one before trying every solution at size two. This method means that a solution is guaranteed to be the shortest possible. Unfortunately, the size of a solution does not directly correlate to speed, depending on how long certain lines in the program take to complete. For instance, when using APIs to synthesize solutions, as Liu et al. (2019) did, it is difficult to prove that a solution is the most efficient as there is no feasible way of determining the time complexity of the APIs used. Despite this, having a short and simple solution is generally easier to understand and maintain.

The strengths of maintainability and provability show why program synthesis is so exciting to many in the computer science discipline. However, program synthesis struggles to generate solutions because of the difficulty of efficiently searching the program space and the possibility of silent errors (errors that are difficult to detect). It is difficult to efficiently search the program space because the size of possible solutions increases exponentially as the program size increases linearly. When attempting to solve long problems, many synthesizers will eventually stall out without finding a solution as there are far too many functions to possibly test. Researchers employ many strategies to increase the speed of synthesis, such as syntax-guided synthesis or representing the program space with more efficient data structures (Alur et al., 2013; Koppel et al., 2022). However, these strategies need to do more to overcome the exponential growth of the possible function field.

Users of program synthesis also need to grapple with silent errors in synthesis, especially if they do not pay close attention to the input they give the synthesizer. Program synthesis is generally a black-box process, meaning the user cannot track what the synthesizer does internally. Some synthesizers are not black boxes, like the one developed by Zhang et al. (2021), but even open-box synthesizers are extremely difficult to track. Zhang and fellow researchers remark: "Figuring out how exactly these synthesizers solve a task may even perplex the synthesis designers themselves." Because of this complexity, when errors occur during synthesis, users will struggle to identify the cause and fix it.

Ambiguous or incomplete user input commonly causes silent errors. For example, if a syntax-guided synthesizer does not receive all of the functions it needs to solve a task, it will fail (Alur et al., 2013). If a synthesizer that requires API usage receives usage details that are inaccurate or incomplete, it will also likely fail (Liu et al., 2019). If a synthesizer that uses natural language and input/output examples receives conflicting information in these two formats, it will not be able to find a solution that matches both (Jain et al., 2022). If any synthesizer that uses input/output examples does not receive enough examples, or any of the given examples is incorrect, the synthesizer will likely either fail or output an incorrect solution (David & Kroening, 2017). Clearly, many errors can arise with incomplete or incorrect specifications, and these errors are difficult to debug because of the black-box, complicated nature of program synthesis.

Taken together, the general strengths and weaknesses of program synthesis show a methodology that is powerful but only in controlled circumstances. When program synthesis yields a valid program, this program has many advantages over programs produced by human coders, as it is likely to be efficient and bug-free (assuming proper user input). However, this program is only produced when the user takes special care to devise descriptive and detailed specifications, and the program is short.

Conclusion

Discussing the philosophies, methods, strengths, and weaknesses of program synthesis helps to develop a more solid understanding of program synthesis's current capabilities. The three different philosophies of program synthesis show how synthesis will be more capable in the hands of someone experienced with writing code. The methods of program synthesis show that the field can quickly develop when a new idea is adopted, which has the possibility of happening with new AI/ML integrations. The strengths and weaknesses of program synthesis show that when a program is generated successfully, it will have many advantages over human-written code, although there are limits to what can be generated.

Program synthesis is a rapidly changing field. While researching this topic, there seem to be gaps and areas in the field that need to be researched further. Researchers should continue to investigate artificial intelligence and machine learning techniques. Using program synthesis as an additional validator for AI-generated code has already shown the potential to solve some problems. As AI/ML rapidly improves, program synthesis should continue integrating promising AI/ML techniques. Additionally, researchers should conduct more studies on synthesis-assisted development. The systems introduced by Ferdowsifard et al. (2021) and Zhang et al. (2021) show promise but were both only tested by around five participants. Large-scale user testing should be performed for systems such as these to better understand how programmers can use and benefit from synthesis-assisted development.

To conclude, based on observations made in this review, it is unlikely that program synthesis will be able to replace the traditional developer experience in the foreseeable future. Currently, program synthesis is best used by someone who understands coding and knows how to solve the inputted problem. Program synthesis will be able to help developers write more clean and efficient code, but without significant changes in approach, it will not completely replace a human developer.

References

Alur, R., Bodik, R., Juniwal, G., Martin, M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., & Udupa, A. (2013). Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design,* 1-8. https://doi.org/10.1109/FMCAD.2013.6679385.

Alur, R., Radhakrishna, A., & Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. *Tools and Algorithms for the Construction and Analysis of Systems,* 319–336. https://doi.org/10.1007/978-3-662-54577-5_18.

David, C., & Kroening, D. (2017). Program synthesis: challenges and opportunities. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, *375*(2104), 20150403. https://doi.org/10.1098/rsta.2015.0403.

Ferdowsifard, K., Barke, S., Peleg, H., Lerner, S., & Polikarpova, N. (2021). LooPy: Interactive program synthesis with control structures. *Proc.ACM Program.Lang., 5*(OOPSLA). https://doi.org/10.1145/3485530.

Feser, J. K., Chaudhuri, S., & Dillig, I. (2015). Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices, 50*(6), 229–239. https://doi.org/10.1145/2813885.2737977.

Gulwani, S. (2010). Dimensions in program synthesis. *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 13–24. https://doi.org/10.1145/1836089.1836091.

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., & Sharma, R. (2022). Jigsaw: Large language models meet program synthesis. *International*

Conference on Software Engineering, 1219–1231. https://doi.org/10.1145/3510003.3510203.

Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., & Polikarpova, N. (2022). Searching entangled program spaces. *Proc.ACM Program.Lang., 6*(ICFP). https://doi.org/10.1145/3547622.

Lee, S., Nam, S. Y., & Kim, J. (2022). Program synthesis through learning the input-output behavior of commands. *IEEE Access, 10*, 63508-63521. https://doi.org/10.1109/ACCESS.2022.3183091.

Lee, W., Heo, K., Alur, R., & Naik, M. (2018). Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices, 53*(4), 436–449. https://doi.org/10.1145/3296979.3192410.

Liu, B., Dong, W., & Zhang, Y. (2019). Accelerating API-based program synthesis via API usage pattern mining. *IEEE Access, 7*, 159162–159176. https://doi.org/10.1109/ACCESS.2019.2950232.

Shi, K., Steinhardt, J., & Liang, P. (2019). FrAngel: Component-based synthesis with control structures. *Proc.ACM Program.Lang., 3*(POPL). https://doi.org/10.1145/3290386.

Solar-Lezama, A. (2013). Program sketching. *International Journal on Software Tools for Technology Transfer, 15*(5), 475-495. https://doi.org/10.1007/s10009-012-0249-7.

Zhang, T., Chen, Z., Zhu, Y., Vaithilingam, P., Wang, X., & Glassman, E. (2021). Interpretable program synthesis. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3411764.3445646.