

Synthesis of Proven-to-be Correct Programs

Eva Maria Wagner
 TU Wien
 Vienna, Austria

Extended Abstract

Background. Program verification is often the most immediate association when discussing the field of formal methods. The process of checking whether a program is correct according to some specification can be handled by automated theorem provers (ATP), like Vampire [6]. While this process leads to outputs with high levels of trust, the disadvantage is that it requires the additional manual step of writing the program. The idea of deductive program synthesis [7, 8] is to include the construction of the program in the automation process. In other words, give the automated prover only the specification as input and let it construct the correct program in addition to proving the specification. This technique reduces the extra manual step of actually writing the code. It avoids inaccurate implementations in addition to saving time and manual labour.

In Figure 1 the workflows of program verification and deductive program synthesis are depicted. In particular, the boxes highlighted in red signalize the removal of the manual task of writing the program code.

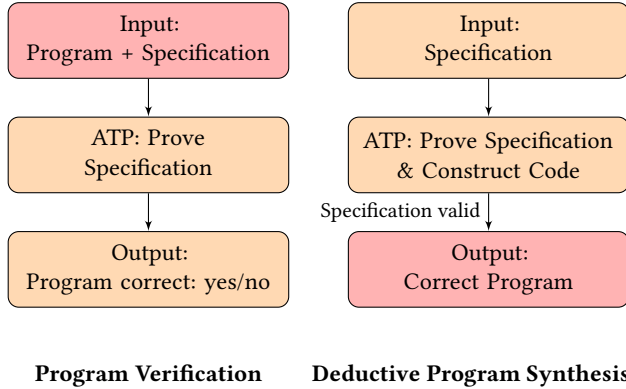


Figure 1: Comparing program verification with deductive program synthesis.

Outline. We explain the underlying framework of our deductive synthesis tool on a high level. Our implementation uses the first-order theorem prover Vampire as its underlying mechanism for proving the specification and is able to synthesize the code conforming to the input specification.

Specification Format. One major characteristic of different synthesis techniques is the format of the specification, which expresses the requirements of the to-be-synthesized code. The specification format can range from a set of input/output pairs, e.g. programming by example [1], to formulas in different types of logic, e.g. propositional logic or temporal logic [2]. Deductive program synthesis uses

first-order logic as specification format, in particular we consider input of the form:

$$\forall \bar{x} \exists y. F[\bar{x}, y]. \quad (1)$$

In (1) the formula F specifies the *task* of the to-be synthesized program, the variables $\bar{x} = x_1, \dots, x_n$ correspond to the *program input* and the variable y corresponds to the *program output*.

Deductive Synthesis. In our implementation, the synthesis task is successful if an automated theorem prover, such as Vampire, is able to prove the formula (1) and can construct a computable term $p(\bar{x})$ that depends on the input variables $\bar{x} = x_1, \dots, x_n$ such that the formula

$$\forall \bar{x}. F[\bar{x}, p(\bar{x})] \quad (2)$$

is valid. The novelty of this approach is that, in addition to *automatically constructing* the program $p(\bar{x})$, there is a proof that certifies the correctness of the program according to the given specification, i.e. the proof that (2) is valid.

In order to explain this process of synthesizing programs from specifications of the form (1) in more detail, I will give an example derivation for the poster presentation. It explains the basics of saturation and showcases how synthesis is possible with the use of answer literals.

Induction. In order to possibly synthesize a recursive program from a given specification, an important proving technique comes into play. Induction enables automated theorem provers to prove more complex formulas, [3]. Due to the immediate relation of induction and recursion, extending the synthesis framework in [5] to inductive reasoning is therefore key for synthesizing programs that use recursion. In particular, this extension helps with reasoning over inductive datatypes, e.g. natural numbers, lists or binary trees. To showcase this, we give the standard induction axiom over the natural numbers for some arbitrary formula G :

$$(G[0] \wedge \forall n. (G[n] \rightarrow G[n+1])) \rightarrow \forall x. G[x]. \quad (3)$$

In order to show that some property holds for all natural numbers, the induction principle can be utilized as follows. If the property holds for the number 0 (the base case) and assuming that if for every n the property holds, it can be shown that it also holds for its successor $n+1$ (the step case), one can conclude that it holds for all natural numbers.

Recursive Synthesis. Coming back to synthesis, assume the goal is to prove a formula of the form (1) with a single input variable x . Successful instantiation of the standard induction axiom over the natural numbers (3) on (1) essentially results in constructing a program $p(x)$, where

$$\begin{aligned} p(0) &= t_0 \\ p(n+1) &= t_s(p(n)) \end{aligned} \quad (4)$$

for some terms t_0 and t_s . Hence, this results in synthesizing a recursive program (4). Note that a valid induction axiom can have varying formats (think, e.g. of starting with a different base case) and may reason over different inductive datatypes, such as lists or binary trees. The synthesized programs will therefore also be of this format and possibly defined over these types.

Automating Deductive Synthesis. We achieved the successful implementation of deductive program synthesis that makes use of induction during an automatic proof and therefore enables synthesis of recursive programs over different inductive data types like natural numbers, lists and binary trees, [4].

Conclusion. The naive idea of "letting computers do all the work for you" seems desirable, but some immediate follow-up questions pop up as well. Can it be that expressing the synthesis task may be as difficult as doing the programming yourself? How can one certify that all possible programs may be expressed in the specification format? How can one specify a synthesis task without any room for ambiguity? How does the synthesis tool certify that the computer's output is correct? While there is definitely a need for the first two questions to be discussed, our research helps to answer the last two. In particular, we introduced our deductive synthesis framework that handles very formal specifications to disallow ambiguity and uses the automated theorem prover Vampire for certifying the correctness of the output.

Keywords

Deductive Program Synthesis, Answer Literals, Saturation, Induction, Recursion, Automated Reasoning

Acknowledgments

This research is funded by the ERC Consolidator Grant ARTIST 101002685. I would also like to thank my collaborators Petra Hozzová, Márton Hajdu and Laura Kovács (supervisor) for valuable discussions and support on this research.

References

- [1] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.* 46, 1 (Jan. 2011), 317–330. doi:10.1145/1925844.1926423
- [2] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. doi:10.1561/25000000010
- [3] Márton Hajdu, Laura Kovács, Michael Rawson, and Andrei Voronkov. 2022. The Vampire Approach to Induction. In *Practical Aspects of Automated Reasoning*.
- [4] Petra Hozzová, Daneshvar Amrollahi, Márton Hajdu, Laura Kovács, Andrei Voronkov, and Eva Maria Wagner. 2024. Synthesis of Recursive Programs in Saturation. In *Automated Reasoning*, Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt (Eds.). Springer Nature Switzerland, Cham, 154–171. doi:10.1007/978-3-031-63498-7_10
- [5] Petra Hozzová, Laura Kovács, Chase Norman, and Andrei Voronkov. 2023. Program Synthesis in Saturation. In *CADE*. 307–324. doi:10.1007/978-3-031-38499-8_18
- [6] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *CAV*, 1–35.
- [7] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (1980), 90–121. doi:10.1145/357084.357090
- [8] Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (March 1971), 151–165. doi:10.1145/362566.362568