Feature Model for Imperative Program Synthesis

Aleksandar S. Dimovski
Mother Teresa University
Skopje, MKD
aleksandar.dimovski@unt.edu.mk

Goran Velinov
University Ss. Cyril and Methodius
Skopje, MKD
goran.velinov@finki.ukim.mk

Shpetim Rexhepi
Mother Teresa University
Skopje, MKD
shpetim.rexhepi@unt.edu.mk

***Abstract*—In this paper, we thoroughly explore the space of the state-of-the art in the field of imperative program synthesis. This results in a feature model that identifies the key features and their variation points for designing today's imperative program synthesizers. Furthermore, we establish two major areas of improvement: program expressivity and scalability.**

***Index Terms*—Feature Model, Imperative Program Synthesis, Variation Points**

I. Introduction

Imperative program synthesis is a process of automatically constructing an imperative program from a specification. This has been an idealistic goal in computer science for about a half a century [1]. The apparent benefit is that only a set of desired properties has to be provided, and a synthesizer will automatically return a correct-by-construction program taking into account all desired properties, so no additional verification [2] is needed. All the steps involved in synthesizing the target program should be mechanical, carried out algorithmically by a computer, without any programmer intervention.

Imperative program synthesis represents a *search problem*, where the key challenge is to deal with the enormous size of its search space of candidate programs, which range over arbitrary programs in general-purpose imperative languages. To simplify the synthesis problem, the search is usually limited to programs of particular form. For this aim, program templates (sketches) are employed, which represent partial programs with missing "*holes*" for unknown statements and expressions. On the other hand, specifications are stated in the form of either logical formulas (e.g., assertions) or input-output examples. Hence, inputs to synthesis algorithms are a program template and a specification, while the output is a complete program satisfying the given specification. However, naive algorithms that simply enumerate all candidate programs until they find a correct one are unlikely to scale. Therefore, the synthesis algorithms usually employ various techniques to efficiently search for correct programs, ranging from decision procedures for supported logical theories (SAT and SMT solvers) [3] to static analyses developed in the program debugging field [4], [5].

Software Product Line (SPL) engineering [6], [7], [8] is a known technique applied in industry to meet the need for custom-tailored software via producing a family of related software products. Today SPLs are very popular in various domains, like cars, mobiles, health [9], etc. Products (family members) in a SPL are differentiated by their *features*, where a feature is an increment in program functionality. This way, individual products are specified via features enabled for them, while SPLs are specified via *feature models* [10] that capture the information about common and variant features. The feature models are represented as a tree of hierarchically arranged features with various relationships between features.

In this paper, we introduce a conceptual model for designing imperative program synthesizers given in the form of a *feature diagram*. We construct this model from a review of the state-of-the-art in the field of imperative program synthesis. The model gives insight into the features already present as well as into the under-explored variation points. Moreover, we propose new features that should be added in the synthesizers in the future.

In short, our contributions are the following:

- We systematically explore related work and derive a model in the form of a feature diagram highlighting the essential features of imperative program synthesizers.
- We identify shortcomings in today's imperative program synthesizers, and propose novel features that should be supported by them.

II. Imperative Program Synthesis

We consider a simple general-purpose imperative language á la C that contains statically allocated program variables and the mathematical integers $\mathbb{Z}$ as the data types. The unknown holes are represented by the constructs "`??`", which are placeholders that the synthesizer should replace with concrete statements and expressions, so that the obtained complete program meets the specification. The syntax of this language is defined by context-free grammars:

$$\begin{aligned} s\,(s \in \mathit{Stm}) &::= \texttt{skip} \mid \texttt{??} \mid \texttt{x=}ae \mid s; s \mid \texttt{if}\,(be)\,s\,\texttt{else}\,s \mid \\ &\qquad \texttt{while}\,(be)\,\texttt{do}\,s \mid \texttt{assert}(be) \\ ae\,(ae \in \mathit{AExp}) &::= n \in \mathbb{Z} \mid \texttt{??} \mid [n, n'] \mid \texttt{x} \in \mathit{Var} \mid ae \oplus ae, \\ be\,(be \in \mathit{BExp}) &::= ae \bowtie ae \mid \texttt{??} \mid \neg be \mid be \wedge be \mid be \vee be \end{aligned}$$

where $\bowtie \in \{<, \leq, =, \neq\}$, $\oplus \in \{+, -, *, /\}$, and $[n, n']$ ranges over integer intervals. We say that a program (statement) is *complete* if it does not have any holes, otherwise we say that it is *partial* (template, sketch).

We use a motivating example to illustrate the imperative program synthesis problem. Consider the template `sum` in Fig. 1, which aims to compute the sum of all integers from 0 to a given input integer `n`. The goal is to generate the complete program in Fig. 2 representing a correct solution to the given synthesis problem. Through this example we will illustrate

```
int sum(int n){
  assume(n ≥ 1);
  int res = ??;
  ??
  while(??) do {
    ??}
  return res;
}
```

Fig. 1. *Template* sum

```
int sum(int n){
  assume(n ≥ 1);
  int res = 0;
  int i = 1;
  while(i ≤ n) do {
    res = res+i;
    i = i+1; }
  return res; }
```

Fig. 2. *Solution* sum

various state-of-the-art imperative program synthesizers. To construct the feature model, we first systematically explore the literature on imperative program synthesis.

Sketch: We start with the Sketch tool by Solar-Lezama [11], [12], which is one of the earliest approaches to address this problem. The Sketch solves the synthesis problem symbolically using a SAT solver for propositional logic, by relying on the learnt conflict clauses to prune the search space. The Sketch generates a correct complete program from a partial program (sketch), such that the missing holes ?? are replaced only by integer constants, and the specification is given as an assertion. To do so, Sketch employs the technique of counter-example guided inductive synthesis. In particular, it repetitively constructs a finite set of inputs and executes SAT queries to find integer constants for the holes so that the obtained program satisfies the assertion for the given inputs. It uses additional SAT queries to verify whether the obtained program is correct for all possible inputs. Moreover, Sketch reasons about while-loops by unrolling them b-times ($b > 0$) and replacing them with if-s, so that only b-bounded paths that are going through the loops at most b-times are considered. Hence, Sketch uses bounded validation procedure to check the bounded correctness of candidate programs with respect to the given assertion and bounded loops.

For example, in the motivating example sum the inputs are the sketch: int res = ??;int i = ??;while(i ≤ n)do{res = res+i;i = i+??; }, as well as the assertion: assert(res==n*(n+1)/2). The Sketch successfully resolves this problem by filling the missing holes with integer constants, thus producing the correct program in Fig. 2.

FamilySketcher: Another approach [13], [14] that solves the sketching problem (where holes stand for integer constants), called FamilySketcher, uses lifted (SPL) static analysis based on abstract interpretation. The main point is that all possible sketch realizations (i.e., instantiations of concrete integer constants for holes) constitute a SPL with numerical features. Hence, the effort of performing an efficient search through all sketch realizations is transferred to a static analyzer specifically designed for SPLs, which uses a decision tree-based numerical abstract domain. By performing the static analysis of the so-constructed program family, the tool reports those products of the family that satisfy the final assertion as final solutions to the sketching problem. This approach can also generate a solution that is optimal with respect to the number of execution steps to termination. Unlike the Sketch that reasons about while-loops by unrolling them, the FamilySketcher can handle unbounded while-loops and execution paths via widening. Moreover, the Sketch being based on SAT solver is especially suited for resolving bit-manipulating programs, while the FamilySketcher being based on numerical abstract domains is more suited for numerical programs with integers.

GenSketching: The GenSketching tool [15] solves the so-called generalized program sketching problem where the unknown holes can be replaced with arbitrary expressions, not only with integer constants as in the case of Sketch and FamilySketcher. GenSketching consists of two phases: constraint (i.e., logical formula) generation and solving. In the first phase, it uses a forward static analysis to infer the invariant before the hole, and a backward static analysis to infer the sufficient condition that ensures the validity of the assertion after the hole. Then, in the second phase, it uses logical abduction to compute unknown hypothesis in a logical inference query. Since it is based on numerical abstract domains, GenSketching can handle unbounded while-loops and is more suited for resolving programs with integers.

For example, in the motivating example sum the inputs to the tool are the sketch: res = ??;i = ??;while(??)do{res = ??;i = ??; }, as well as the assertion: assert(res==n*(n+1)/2). The GenSketching outputs the correct program in Fig. 2.

SIMPL: The SIMPL synthesizer [16] combines the enumerative search with static analysis to synthesize imperative programs. That is, SIMPL enumerates every candidate program in increasing size until it finds a solution. To speed up the search process, it performs static analysis of partial programs, so that it identifies and prunes out those partial programs that will certainly fail to be a solution. The inputs are a partial program with statement holes and a set of input-output examples specifying the desired behaviours. The output is a complete program matching the given input-output examples. SIMPL can handle numerical programs with integer variables, arrays, and unbounded loops. The full correctness of a candidate program is established by checking all input-output examples against it.

SemGus: The semantic-guided synthesis (SemGus) [17] provides a framework for inferring imperative programs that may contain unbounded loops. The tool is capable of both synthesizing imperative programs and proving that a synthesis problem is unrealizable, i.e. it has no solution. The entire synthesis problem is encoded using Constrained Horn Clauses (CHCs), which are a class of first-order logical formulas that can be solved with a CHC solver, such as Z3. This way, the synthesis problem is effectively reduced to a proof-search problem over CHCs. The inputs are a partial program with statement holes and a set of input-output examples, while the output is a complete program that meets the specification.

For example, in the motivating example sum the inputs of SIMPL and SemGus are the partial program in Fig. 1, and the set of input-output examples: $\{2 \to 3, 3 \to 6, 5 \to 15, 6 \to 21\}$. The output is the complete program in Fig. 2.

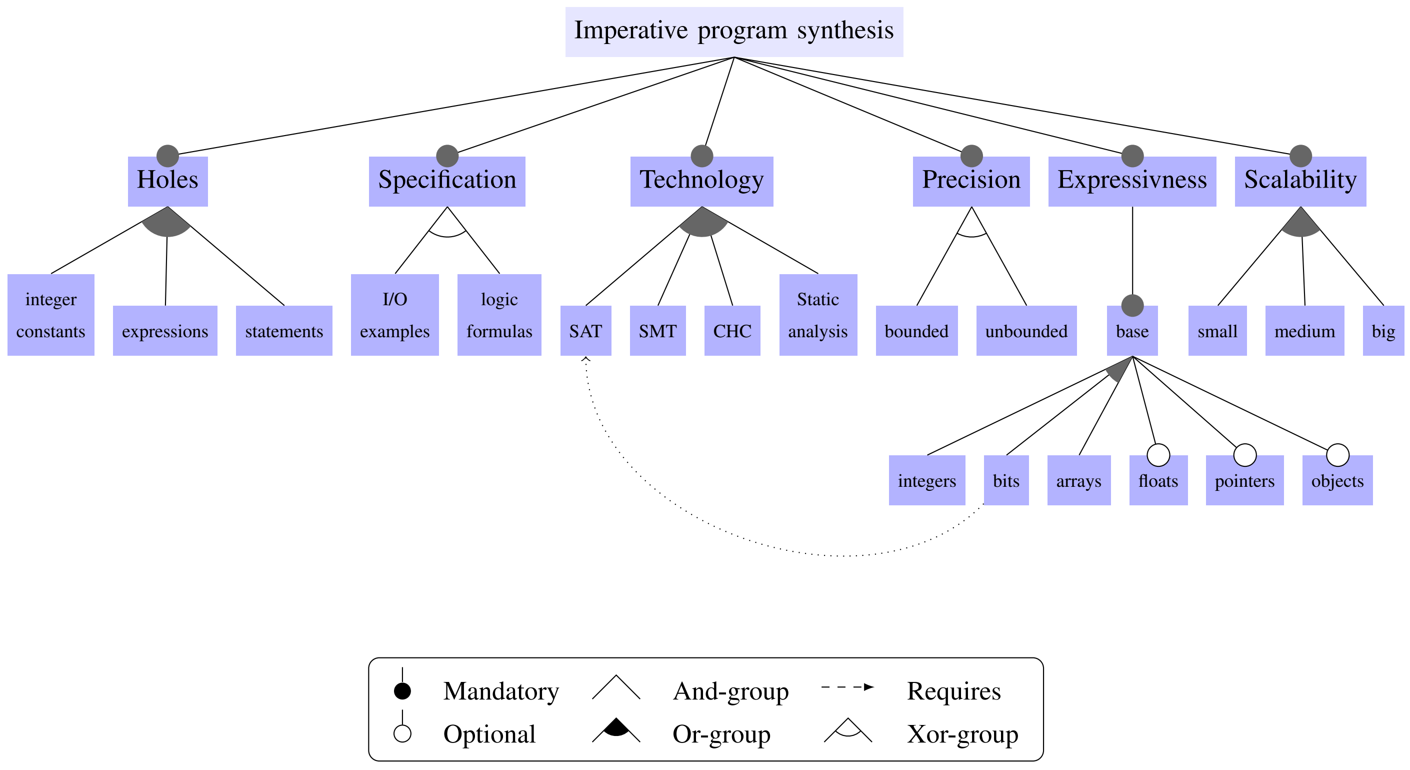

Fig. 3. Feature model for imperative program synthesis.

IMPSynth: The IMPSynth synthesizer [18] is obtained by combining the techniques used in [16] and [19]. It takes as inputs a partial program with statement and expression holes and a formal specification given as a logical formula (assertion), while it outputs a correct-by-construction complete program satisfying the given assertion. IMPSynth consists of three parts: statement-directed enumerative search, static analysis based on abstract interpretation, and expression-directed enumerative search. The first part performs a global coarse-grained search via enumerating all possible candidate programs, thus filling in the statement holes. As a result, partial programs with only missing expression holes, called semi-partial programs, are obtained. The second part uses static analysis to identify and prune out of the search those semi-partial programs that certainly will fail to be solutions. The third part performs an expression-directed enumerative search via substituting concrete expressions in increasing size order for the expression holes in the given semi-partial program, and then by attempting to repair the so-obtained program via application of syntactic mutations [19]. To check the bounded correctness of candidate programs, the tool translates the programs (where `while`-loops are unrolled a finite number of times and replaced with `if`-s) into SMT formulas, whose satisfiability is checked by using SMT solvers. To efficiently explore the big space of mutants, SAT and SMT solvers are called in an incremental way.

For example, in the motivating example `sum` the inputs are the partial program in Fig. 1, and the assertion: `assert(res==n*(n+1)/2)`. The output is the complete program in Fig. 2.

III. Feature Model

By using our analysis of the related works, we construct a feature model (FM) for imperative program synthesizers shown in Fig. 3. We identify common features and their variation points in the current state-of-the art synthesizers, as well as features and variation points that remain under-explored or missing, thus motivating their design in the future.

The feature model consists of six main features: type of Holes, Specification, Technology used in the implementation, Precision, Expressiveness of the language, and Scalability. In what follows, we detail these features and discuss how existing imperative program synthesizers handle them.

a) Supported Features and Variation Points: Every feature in the FM in Fig. 3 has one parent, excluding the root feature (Imperative program synthesis). Connections between a feature and its group of children can be as follows: *And-groups* (no arc, e.g., children of the Imperative program synthesis feature); *Or-groups* (filled arc, e.g., children of the Technology feature); and *Xor-groups* (unfilled arc, e.g., children of the Precision feature). The children of And-groups can be either mandatory (filled circle) or optional (unfilled circle).

When a feature is selected, then its parent is selected too. On the other hand, if a parent feature is selected, then we have: (1) all its mandatary children of an And-group are selected; (2) all its optional children of an And-group may be selected; (3) at least one of its children of an Or-group is selected; and (4) exactly one its child of a Xor-group is selected. A feature model may also have cross-tree constraints, expressed as propositional formulas, that show non-hierarchical relationship between features. E.g., bits $\implies$ SAT is such cross-tree constraint in the FM in Fig 3, expressing that SAT solvers must be used in synthesizers to handle bit-manipulating programs.

A *concrete feature* is a leaf in the FM, while an *abstract feature* is an interior node in the FM. A *configuration* of a feature model is a set of concrete features selected from the FM according to its semantics. Note that all ancestors of a concrete feature are implicitly included in a configuration. One configuration corresponds to one individual product from the SPL represented by the given FM. For example, from the FM in Fig. 3 we can derive the following configurations: {integer constants, logic formulas, SAT, bounded, integers, bits, small} corresponding to `Sketch`; {integer constants, logic formulas, static analysis, unbounded, integers, small} corresponding to `FamilySketcher`; {expressions, logic formulas, static analysis, unbounded, integers, small} corresponding to `GenSketching`; {expressions, statements, I/O examples, static analysis, unbounded, integers, arrays, medium} corresponding to `SIMPL`; {expressions, statements, I/O examples, CHC, unbounded, integers, arrays, medium} for `SemGus`; and {expressions, statements, logic formulas, SAT, SMT, static analysis, bounded, integers, medium} for `IMPSynth`.

b) Missing Features and Variation Points: We identify several features that should be supported in the near future: floats (data type), pointers, objects, and big (-sized programs). One of the potential future directions for extending the current synthesizers is by enriching the expressivity of the programming langue they consider. In particular, it would be interesting to handle other language constructs such as pointers, objects, as well as data types other than integers and bits such as floats.

Real-world software is rarely monolithic entity, but is often built from a multitude of components (modules). To synthesize such real-world (big-sized) programs, the algorithms should retain synthesis strategies that are compositional in the structure of programs. Compositionality is the key feature for achieving scalability, i.e. the possibility to break up a larger program into smaller sub-programs that can be synthesized independently. To enable such compositional synthesis, the specification of the full behaviour of the program should be possible to be split into simpler sub-specifications for its individual components (sub-programs). This way, the synthesis would proceed in a bottom-up fashion by first synthesizing the sub-programs, which will then be composed into the resulting big program.

IV. Conclusion

We have presented a feature model for imperative program synthesis that includes features and variation points, which are either supported or under-explored in the current tools. The proposed feature model can help low-code development researchers understand the concept of SPLs and connect the low-code research to program synthesis and domain-specific languages.

References

[1] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1989, pp. 179–190. [Online]. Available: https://doi.org/10.1145/75277.75293

[2] A. S. Dimovski, D. R. Ghica, and R. Lazic, "A counterexample-guided refinement tool for open procedural programs," in *Model Checking Software, 13th International SPIN Workshop, 2006, Proceedings*, ser. LNCS, vol. 3925. Springer, 2006, pp. 288–292. [Online]. Available: https://doi.org/10.1007/11691617_17

[3] A. Dimovski and D. Gligoroski, "Generating highly nonlinear boolean functions using a genetic algorithm," in *6th Int. Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service, TELSIKS 2003*, ser. IEEE, vol. 2, 2003, pp. 604–607. [Online]. Available: https://doi.org/10.1109/TELSKS.2003.1246297

[4] A. S. Dimovski, "Error invariants for fault localization via abstract interpretation," in *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, ser. LNCS, vol. 14284. Springer, 2023, pp. 190–211. [Online]. Available: https://doi.org/10.1007/978-3-031-44245-2_10

[5] A. Dimovski, "Fault localization by abstract interpretation and its applications," *J. Comput. Lang.*, vol. 80, p. 101288, 2024. [Online]. Available: https://doi.org/10.1016/j.cola.2024.101288

[6] A. S. Dimovski, S. Apel, and A. Legay, "A decision tree lifted domain for analyzing program families with numerical features," in *24th Int. Conf. FASE 2021, Proceedings*, ser. LNCS, vol. 12649. Springer, 2021, pp. 67–86. [Online]. Available: https://arxiv.org/abs/2012.05863

[7] A. S. Dimovski, C. Brabrand, and A. Wasowski, "Finding suitable variability abstractions for family-based analysis," in *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, ser. LNCS, vol. 9995. Springer, 2016, pp. 217–234. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6_14

[8] A. S. Dimovski and S. Apel, "Lifted static analysis of dynamic program families by abstract interpretation," in *35th Conference ECOOP 2021*, ser. LIPIcs, vol. 194. Schloss Dagstuhl, 2021, pp. 14:1–14:28. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2021.14

[9] B. Atanasovski, M. Bogdanovic, G. Velinov, L. Stoimenov, A. S. Dimovski, B. Koteska, D. Jankovic, I. Skrceska, M. Kon-Popovska, and B. Jakimovski, "On defining a model driven architecture for an enterprise e-health system," *Enterp. Inf. Syst.*, vol. 12, no. 8-9, pp. 915–941, 2018. [Online]. Available: https://doi.org/10.1080/17517575.2018.1521996

[10] D. Batory, "Feature models, grammars, and propositional formulas," in *9th International Software Product Lines Conference, SPLC '05*, ser. LNCS, vol. 3714. Springer-Verlag, 2005, pp. 7–20.

[11] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 281–294. [Online]. Available: https://doi.org/10.1145/1065010.1065045

[12] A. Solar-Lezama, "Program sketching," *STTT*, vol. 15, no. 5-6, pp. 475–495, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0249-7

[13] A. Dimovski, S. Apel, and A. Legay, "Program sketching using lifted analysis for numerical program families," in *NASA Formal Methods, NFM 2021*, ser. LNCS, vol. 12673. Springer, 2021, pp. 95–112. [Online]. Available: https://doi.org/10.1007/978-3-030-76384-8_7

[14] A. Dimovski, "Quantitative program sketching using decision tree-based lifted analysis," *J. Comput. Lang.*, vol. 75, p. 101206, 2023. [Online]. Available: https://doi.org/10.1016/j.cola.2023.101206

[15] A. S. Dimovski, "Generalized program sketching by abstract interpretation and logical abduction," in *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, ser. LNCS, vol. 14284. Springer, 2023, pp. 212–230. [Online]. Available: https://doi.org/10.1007/978-3-031-44245-2_11

[16] S. So and H. Oh, "Synthesizing imperative programs from examples guided by static analysis," in *Static Analysis - 24th International Symposium, SAS 2017, Proceedings*, ser. LNCS, vol. 10422. Springer, 2017, pp. 364–381. [Online]. Available: https://doi.org/10.1007/978-3-319-66706-5_18

[17] J. Kim, Q. Hu, L. D'Antoni, and T. W. Reps, "Semantics-guided synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–32, 2021. [Online]. Available: https://doi.org/10.1145/3434311

[18] A. S. Dimovski, "Imperative program synthesis by abstract static analysis and smt mutations," in *GPCE '25*. ACM, 2025. [Online]. Available: https://doi.org/10.1145/3742876.3742884

[19] A. Dimovski, "Mutation-based lifted repair of software product lines," in *38th European Conference on Object-Oriented Programming, ECOOP 2024*, ser. LIPIcs, vol. 313. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 12:1–12:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2024.12