SHEDDING LIGHT ON TASK DECOMPOSITION IN PROGRAM SYNTHESIS: THE DRIVING FORCE OF THE SYNTHESIZER MODEL

Janis Zenkner, Tobias Sesterhenn, Christian Bartelt
Machine Learning and Cognitive Software
Institute for Software and Systems Engineering, TU Clausthal
janis.zenkner@tu-clausthal.de

ABSTRACT

Task decomposition is a fundamental mechanism in program synthesis, enabling complex problems to be broken down into manageable subtasks. ExeDec, a state-of-the-art program synthesis framework, employs this approach by combining a Subgoal Model for decomposition and a Synthesizer Model for program generation to facilitate compositional generalization. In this work, we develop REGISM, an adaptation of ExeDec that removes decomposition guidance and relies solely on iterative execution-driven synthesis. By comparing these two exemplary approaches—ExeDec, which leverages task decomposition, and REGISM, which does not—we investigate the interplay between task decomposition and program generation. Our findings indicate that ExeDec exhibits significant advantages in length generalization and concept composition tasks, likely due to its explicit decomposition strategies. At the same time, REGISM frequently matches or surpasses ExeDec's performance across various scenarios, with its solutions often aligning more closely with ground truth decompositions. These observations highlight the importance of repeated execution-guided synthesis in driving task-solving performance, even within frameworks that incorporate explicit decomposition strategies. Our analysis suggests that task decomposition approaches like ExeDec hold significant potential for advancing program synthesis, though further work is needed to clarify when and why these strategies are most effective.

1 INTRODUCTION

In 2019, François Chollet introduced the Abstraction and Reasoning Challenge (ARC-AGI) benchmark to advance Artificial General Intelligence (AGI) research by assessing systems on their ability to perform broad reasoning tasks (Chollet, 2019). Among the many approaches to achieving AGI, program synthesis has emerged as a particularly promising avenue (Chollet, 2019). Program synthesis automates the creation of programs that meet specified requirements, such as Input-Output (I/O) examples (Devlin et al., 2017), formal logical descriptions (Hocquette & Cropper, 2023), natural language instructions (Desai et al., 2016), or partial program templates (Alur et al., 2013). Despite recent progress, current systems still fall short of human-level performance (Chollet et al., 2024), making human cognition a key inspiration for generalizable reasoning methods. One such capability is the ability to decompose unfamiliar problems into smaller, manageable components (Marcus, 2003; Frankland & Greene, 2020). Task decomposition leverages this principle by dividing complex tasks into subtasks that require generalizable skills to solve (Gulwani et al., 2017). By solving these subtasks independently and integrating their solutions, systems can tackle complex problems with greater scalability and efficiency. Moreover, the emphasis on developing general-purpose capabilities, rather than narrowly tailored solutions, aligns with the broader goal of fostering universal reasoning abilities (Chollet, 2019).

ExeDec (Shi et al., 2023b) integrates program synthesis with task decomposition, combining transductive and inductive reasoning to enhance synthesis performance (Li et al., 2024). It consists of two interconnected components: the *Subgoal* Model, which iteratively decomposes complex tasks by predicting the next subgoal (high-level transductive reasoning), and the *Synthesis* Model, which

Exemplary ExeDec decomposition

Task specification: $x0 = [42, -48] \rightarrow y = [42, 42]$
Ground Truth: `x1 = Scanl1 (max) x0`

Step 1:

- Predicted Subgoal: $[-48, 42]$
- Subprogram: `x1 = Sort x0`
- Execution: $x1 = [-48, 42]$

Step 2:

- Predicted Subgoal: $[48, 42]$
- Subprogram: `x2 = Zip (max) x0 x1`
- Execution: $x2 = [42, 42]$

Figure 1: ExeDec workflow illustrating the challenges posed by misleading subtasks. The task is taken from the original test set and can be solved by computing the cumulative maximum of the input list. The subgoals are those generated by ExeDec. The decomposition process is shown for a single I/O pair, while the full workflow with all I/O pairs is available in Appendix A.

generates programs to solve these subtasks (low-level inductive reasoning). This structured approach enables the Subgoal Model to acquire broadly transferable skills for handling diverse and complex tasks while allowing the Synthesis Model to focus on program generation. As a result, ExeDec outperforms other approaches including large language models on two program synthesis benchmarks Shi et al. (2023b). ExeDec compares its performance against a No-Subgoal Ablation, a baseline where the Subgoal Model is excluded. Besides the difference in architecture, the key distinction lies in the training of the synthesis model: In ExeDec, the synthesis model focuses purely on program generation, while in the No-Subgoal Ablation, it is tasked with both implicit task decomposition *and* program generation. To distinguish both synthesis models, we refer to ExeDec's synthesis model as Synthesizer Model.

A detailed analysis of ExeDec's internal mechanisms offers insights into refining program synthesis approaches and optimizing task-solving strategies. Our analysis is driven by the hypothesis that while the Subgoal Model contributes to ExeDec's performance, its role is more nuanced than previously assumed. Instead, the primary factor enabling task completion is the repeated application of the Synthesizer Model, rather than the Subgoal Model itself. Figure 1 illustrates this dynamic with a single-step task, i.e., a task solvable by a single function and without decomposition. Here, the Subgoal Model generates a subtask prediction that differs from the ground truth causing the Synthesizer Model to produce a subprogram deviating from the target program. In the next step, despite another Subgoal Model inaccuracy, the Synthesizer Model corrects the error and ultimately solves the task. This highlights the Synthesizer Model's robustness in mitigating decomposition inaccuracies. A key takeaway is that increasing the frequency of task decomposition—even when the predicted decompositions are imperfect—can be advantageous, as it allows the Subgoal Model to leverage the Synthesizer Model's corrective capabilities. The workflow in Figure 1 is a real example generated by ExeDec; additional examples are in Appendix A. The results of the ExeDec study quantitatively support this hypothesis further: Even though, ExeDec achieves a small boost in performance beyond the No-Subgoal Ablation, the No-Subgoal Ablation brings the most significant performance improvement (Shi et al., 2023b). This suggests that the explicit decomposition adds value but may not be critical for all tasks and that the Synthesizer Model has a greater influence on overall task success than previously assumed.

Building on these observations, this paper explores the interaction between task decomposition and synthesis within ExeDec. Our key contributions are as follows:

1. We provide insights into the challenges of learning task decomposition across domains, showing that learning is easier in domains with fewer decomposition possibilities.
2. We show that the Synthesizer Model plays a key role in boosting performance, even in the absence of explicit *and* implicit decomposition guidance.
3. We reveal that the Synthesizer Model, in hindsight, decomposes tasks, even without being trained to do so.

2 BACKGROUND

2.1 PROGRAMMING-BY-EXAMPLE

Programming-by-Example (PBE) enables task specification without requiring programming knowledge, hardware familiarity, or coding skills (Gulwani, 2011), making it accessible and versatile across various domains. A PBE task is defined through I/O pairs that specify a desired behavior (see Figure 1). Formally, the specification of a task is a set $X = (I_1, O_1), \ldots, (I_n, O_n)$, where each pair (I_i, O_i) corresponds to an example of the desired behavior. To formalize the search space for solutions, PBE operates within a Domain-Specific Language (DSL), which defines the set of possible programs P. The DSL encompasses functions, identifiers, constants, and variables, forming the building blocks for program generation. The goal in PBE is to identify a program $p \in P$ that satisfies the specification. In other words, p must transform each input sample I_i into its respective output O_i for all $i \leq n$.

2.2 COMPOSITIONAL GENERALIZATION

Compositional generalization refers to a model's capacity to systematically and predictably generalize to novel combinations of known components, akin to how humans generalize language and concepts (Wiedemer et al., 2024). To evaluate this ability in program synthesis, Shi et al. (2023b) introduced specific task categories. Each task is designed to examine a distinct aspect of generalization, ensuring a comprehensive and diverse evaluation.

Length Generalization Models are trained on tasks requiring n decomposition steps and tested on tasks with $m > n$ decomposition steps. This scenario examines the ability to handle increased program complexity.

Compose Different Concepts This task evaluates whether the model can integrate distinct functional concepts. DSL functions are divided into two concepts. Training tasks involve functions either from Concept A or Concept B, while test tasks require the model to combine both concepts.

Switch Concept Order This scenario explores whether models can generalize to new sequences of operations. Again using a categorization of DSL functions into two categories, training tasks always begin with Concept A and end with Concept B. Test tasks reverse this order, starting with Concept B and ending with Concept A.

Compose New Operations The goal here is to determine if a model can integrate a previously isolated function into larger compositions. During training, the function is either used alone or excluded, while testing combines it with other functions. A relevant scenario involves synthesizing code that integrates a newly introduced function into a broader workflow, based solely on isolated training examples.

Add Operation Functionality This task evaluates whether a model can infer and apply new functionalities for an operation. Training tasks limit an operation to specific functionalities (e.g., `Scanl1` with `min`), while testing introduces variations (e.g., `Scanl1` with `max`, or `*`).

3 METHODS & EVALUATION

3.1 EXEDEC

ExeDec (Shi et al., 2023b) is an approach designed to decompose complex tasks into manageable subtasks and solve them effectively. Figure 2 displays the framework which comprises two primary modules: The Subgoal Model, responsible for decomposing the task into subtasks, and the Synthesizer Model for generating subprograms, i.e., programs solving the subtasks. The generated subprogram is subsequently executed, and its output values are utilized to update the task specification. Both modules are Transformer models which are trained on decomposed data using a teacher-forcing approach. For each subprogram in the ground-truth solution, the training data includes: the updated specification derived from executing the preceding ground-truth subprogram, the execution results of the subprogram on all examples, and the subprogram itself. The Subgoal Model is trained to predict the subprogram's execution result based on the updated specification.

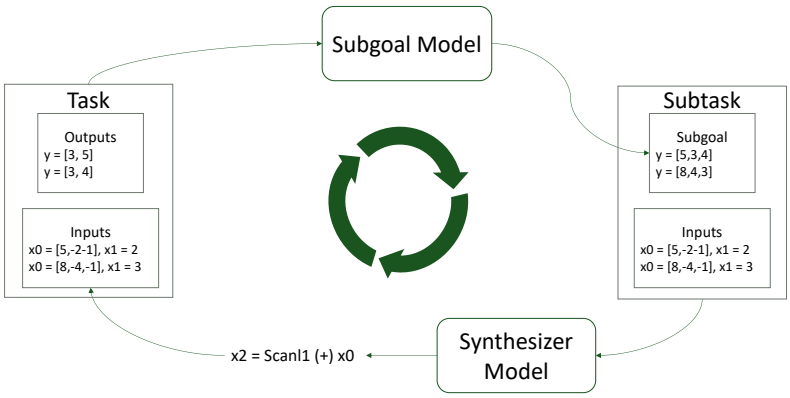


Figure 2: ExeDec Workflow: First, the Subgoal Model predicts the next subgoal based on the current task specifications. Next, the Synthesizer Model generates a program using the subtask specifications, aiming to solve the subtask. Finally, the generated program is executed, and its output is used to update the task specifications.

Given the ground truth *subtask* specifications, the Synthesizer Model is trained to predict the subprogram solving the subtask. The model used in the No-Subgoal Ablation matches the Synthesizer Model architecture. Yet, it is trained to predict the subprogram of the next subtask based on the updated *task* specifications.

3.2 REGISM

To further investigate the role of ExeDec's Subgoal Model, we developed Repeated Execution-Guided Invoking of Synthesis Model (REGISM), a modified version of the ExeDec framework that excludes the Subgoal Model. REGISM invokes the Synthesizer Model to generate a single-step program, i.e., a program consisting of one function only, executes this program, and updates the task specifications using the execution output. At a high level, REGISM shares the same architectural setup and workflow as ExeDec's No-Subgoal Ablation. The key distinction lies in the training paradigm and the role of the synthesis model.

In both REGISM and the No-Subgoal Ablation, the synthesis model generates a single-step program, executes it, and updates the task specification using the execution results. This iterative process creates new subtask inputs for subsequent steps. The primary difference between the two settings lies in how the models are trained. The No-Subgoal Ablation model is trained to predict subprograms using the ground truth subtask inputs and the *task* outputs. This forces the model to implicitly infer both decomposition strategies and program generation, making the learning process more entangled. In contrast, the Synthesizer Model used in REGISM is trained to predict subprograms using ground truth subtask inputs and *subtask* outputs, as it is originally designed for use with ExeDec's decomposition model. Consequently, the Synthesizer Model is trained to focus purely on program generation without the added complexity of decomposition. During testing, when subtask outputs are unknown, both models receive subtask inputs and task outputs as input. This distinction in training objectives enables us to assess the impact of explicitly supervising decomposition strategies, as done in ExeDec, compared to relying purely on program synthesis, as in REGISM.

3.3 Evaluation

3.3.1 Domains

String Manipulation The RobustFill domain, introduced by Devlin et al. (2017), focuses on synthesizing sequences of string manipulation operations. Given I/O string pairs, the goal is to generate programs that transform inputs into corresponding outputs using a DSL (Appendix B). The DSL in-

cludes functions for extracting substrings, modifying strings, and composing operations by applying modifications to prior results. It also supports constant string characters. RobustFill programs are structured as concatenations of largely independent expressions, except for the *Compose* operation, which applies a function to another's result. This weak interdependence among subtasks reduces the combinatorial space, simplifying task decomposition learning. However, it also limits error correction during synthesis, making precise decomposition and implementation critical for correct program generation.

List Manipulation The Deepcoder domain (Balog et al., 2016) addresses tasks involving integer list manipulations. Its DSL includes both first-order and higher-order operations. Details can be found in Appendix B. Tasks in this domain can have multiple inputs, and the outputs are defined as either a list of integer values, an integer, or a boolean value. Programs follow a line-by-line structure, where each expression relies on an input variable and/or the result of a previous expression as its input. Such a framework mirrors the way humans typically construct programs. The domain's line-by-line style enables error correction yielding an expansive combinatorial space and various task decomposition strategies contributing to its inherent complexity.

3.3.2 Experimental Setup

For a fair comparison with ExeDec, REGISM invokes the Synthesizer Model the same number of times as ExeDec's Subgoal Model decomposes the task, with all approaches using a beam size of ten. Furthermore, we used the same 1,000 test tasks, the pretrained models, and DSLs from the original ExeDec study (Shi et al., 2023b)[1]. Details on the creation of the test tasks can be found in Appendix C. Five different initalizations were used per task category. Results are displayed as averages across those initializations, and error bars denote the 95% confidence interval

In addition to ExeDec and REGISM, we evaluate the Synthesizer Model itself to assess the effect of step-by-step as opposed to single-step generation. In summary, we compare ExeDec against these approaches: First, in the No-Subgoal Ablation setting, the synthesis model is trained to predict the subprogram using the subtask inputs and the task output. This setup requires the model to simultaneously learn both decomposition strategies and general-purpose problem-solving abilities. Second, in REGISM, the Synthesizer Model is trained to predict the subprogram using the in- and outputs of the subtask. This approach focuses exclusively on enabling the model to generate a program based on the provided specifications. Third, we evaluate the Synthesizer Model itself by invoking it only once per task.

4 Results & Discussion

Our analysis is motivated by the hypothesis that the Subgoal Model plays a nuanced role in boosting ExeDec's performance. Specifically, we hypothesize that the repeated invocation of the Synthesizer Model is the major driving factor behind task completion, rather than the Subgoal Model itself.

4.1 Domain Differences and Quality of the Subgoal Model

To evaluate ExeDec's ability to learn task decompositions, we analyze its performance relative to the ground truth. Figure 3 presents a density plot illustrating the distribution of solved tasks. The x-axis represents the subtask accuracy per task, measured as the overlap between predicted subtasks and ground truth subtasks. The y-axis indicates subprogram accuracy, defined as the overlap between predicted programs and the ground truth solutions. Each task's position in the plot reflects its accuracy: for a task requiring four decomposition steps, if ExeDec correctly predicts two subtask specifications and solves three subtasks with ground-truth-matching programs, it appears at $(x = 50\%, y = 75\%)$. Consequently, the density plot highlights distinct solution patterns. In the top-right quadrant, both subtasks and subprograms closely align with the ground truth. The bottom-right region represents cases where subtasks mostly match the ground truth, but subprograms differ due to alternative syntactic implementations. The bottom-left quadrant contains instances where both subtasks and subprograms deviate from the ground truth. Finally, in the top-left region, sub-

[1]https://github.com/google-deepmind/exedec

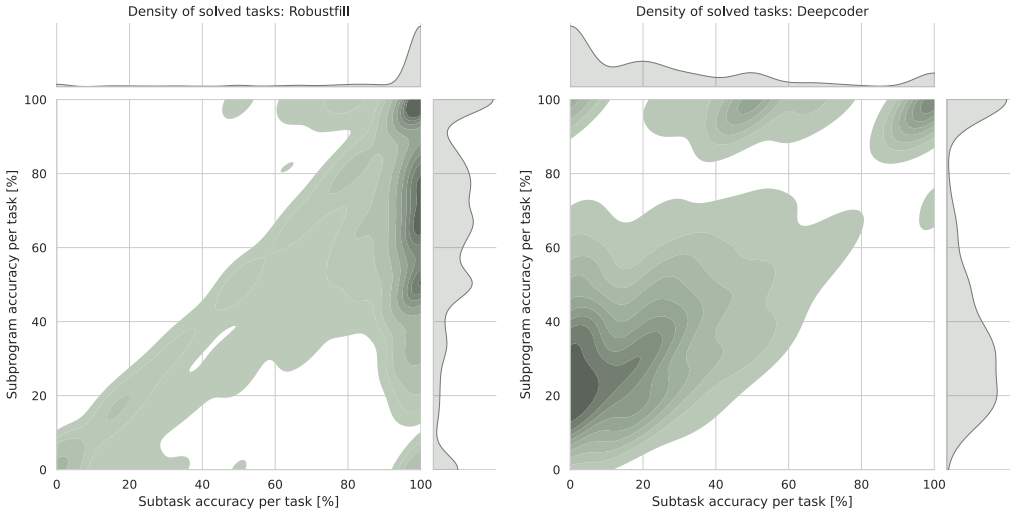


Figure 3: Tasks solved by ExeDec, clustered based on subtask and subprogram accuracy. The x-axis represents subtask accuracy per task (overlap between predicted and ground truth subtasks), while the y-axis indicates subprogram accuracy (overlap between predicted programs and ground truth solutions). Values are averaged across compositional generalization categories, based on 8,900+ solved tasks in DeepCoder and 26,000+ in RobustFill.

programs align closely with the ground truth while subtasks differ. This visualization provides insight into the semantic and syntactic accuracy of ExeDec's solutions.

Ambiguous Performance of the Subgoal Model In the Robustfill domain, the majority of tasks is clustered in the bottom and top right quadrant, i.e., subtasks are predicted as intended by the ground truth but the implementation differs, showcasing the Subgoal Model's accuracy in learning decompositions. A smaller portion of task solutions stretches along the main diagonal. These tasks are solved using - at least partially - alternative decompositions. In the Deepcoder domain, a small cluster exists in the top-right corner, indicating tasks solved as intended by the ground truth. Yet, the majority of tasks is solved by using alternative decompositions and implementations (bottom-left corner). In the ExeDec study, the authors hypothesize that the low subgoal accuracy is caused by slight deviations in the predicted subgoals that the Synthesizer Model compensates for (top-left quadrant). However, as most subtasks are also solved using alternative programs, our results indicate a solution strategy deviating from the ground truth. Although the tasks are ultimately solved, these results question whether the Subgoal Model plays a crucial role in achieving this. Plots displaying each compositional generalization category separately can be found in Appendix D.

More Frequent Invocation of the Synthesizer Model The nuanced role of ExeDec's Subgoal Model is further supported by its tendency to decompose tasks into a greater number of subtasks than those specified by the ground truth in the Deepcoder domain (Figure 4). This increased number

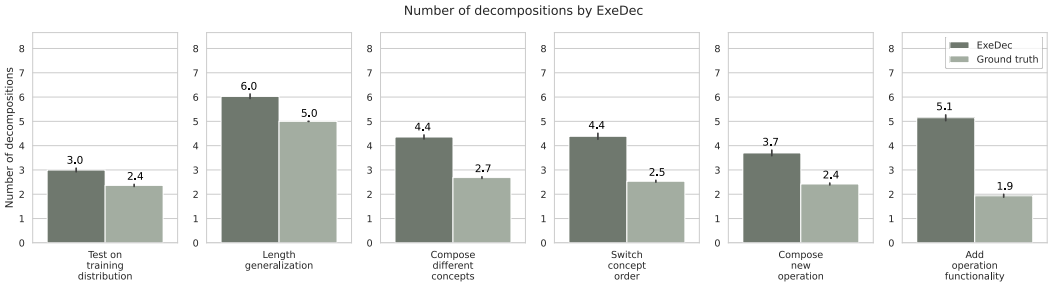


Figure 4: Comparison of the number of decompositions of task solution in the DeepCoder domain using ExeDec.

of decompositions leads to more frequent invocation of the Synthesizer Model, which seems to be the primary factor driving the observed performance improvements.

Domain differences In contrast, in the Robustfill domain, the number of decompositions closely aligns with the ground truth (Figure 13b). This distinct difference can be attributed to the varying difficulty of learning decomposition across the two domains. Given that the Deepcoder domain better aligns with how humans approach coding and the Subgoal Model performs significantly worse in this domain, we focus on the Deepcoder domain in the subsequent experiments to better understand the reasons behind its performance.

4.2 REGISM AS A DRIVING FORCE IN BOOSTING PERFORMANCE

REGISM performs close to ExeDec on average To assess the impact of repeated Synthesizer Model invocation, we evaluate REGISM on the same tasks as ExeDec. As shown in Figure 5, REGISM outperforms the standalone Synthesizer Model, highlighting the benefits of execution guidance and repeated invocation. ExeDec achieves higher accuracy on the training distribution and in tasks requiring the composition of different concepts. REGISM struggles with length generalization, likely due to its single-step training paradigm, which makes deriving correct subprograms for multi-step tasks more challenging. However, when excluding length generalization tasks, REGISM performs only slightly worse than ExeDec, suggesting that repeated invocation is a key factor in solving tasks. While ExeDec's decomposition strategies improve performance in some categories, they offer limited or no advantage in others and can even hinder performance in certain cases. The comparable performance of REGISM and ExeDec indicates that task-solving effectiveness primarily stems from repeated invocation, independent of explicit or implicit decomposition guidance.

REGISM solves tasks closer to the ground truth To gain a better understanding of how REGISM solves tasks, we again perform a qualitative analysis of the task solutions. As shown in Figure 6, REGISM closely approximates the ground truth number of decompositions. Comparing REGISM's results to ExeDec (Figure 4), the tasks solved by REGISM proof to be of similar difficulty to those solved by ExeDec because the number of ground truth decompositions is comparable. Moreover, REGISM solves tasks in about as many iterations as decompositions are intended by the ground truth, i.e., in significantly fewer steps than ExeDec. This indicates that REGISM solves tasks closer

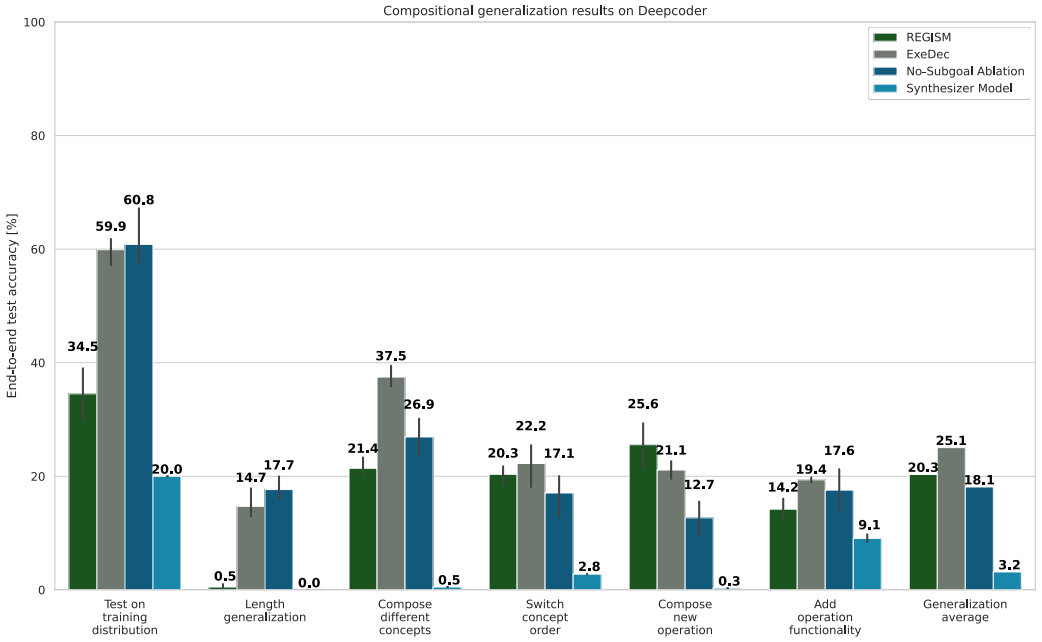


Figure 5: Compositional generalization results using a beam size of 10. End-to-end test accuracy resembles the relative number of solved test tasks.

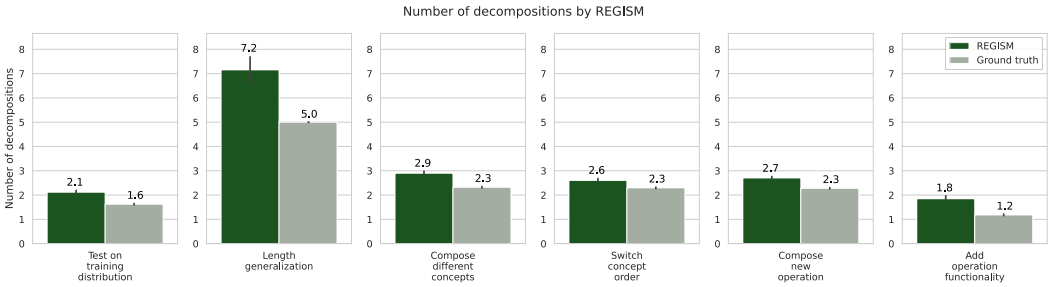


Figure 6: Frequency of invoking the Synthesizer Model in REGISM relative to the number of ground truth decompositions in the Deepcoder domain.

to the ground truth solution thus raising the hypothesis that REGISM's inherent decompositions are closer to the ground truth decomposition than the explicit ones of ExeDec. To investigate this, Figure 7 depicts the subprogram accuracy against the overlap with the ground truth decomposition after executing the subprograms generated by REGISM. This analysis is further split by tasks solved by both approaches or exclusively by either approach. Our findings reveal several notable patterns: REGISM's task solutions tend to spread along the main diagonal. There is one prominent cluster at the right edge of each plot, where all subtasks align with the ground truth decomposition. To a similar but lesser extent than in ExeDec, REGISM's largest cluster is located in the bottom-left quadrant. However, REGISM's solutions exhibit a more uniform distribution along the x-axis compared to ExeDec, whose distribution is distinctly shifted toward lower subtask accuracies. This effect is slightly more pronounced for tasks solved exclusively by REGISM than for those solved by both approaches. These findings indicate that the Synthesizer Model within REGISM generates solutions closer to the ground truth than ExeDec despite not being explicitly or implicitly trained for this purpose. However, for both approaches, the quality of decompositions does not differ significantly between tasks solved exclusively and those solved by both methods. This finding leaves open the question of why ExeDec solves certain tasks that REGISM does not, especially when the predicted subgoals in those tasks are not more precise than in tasks solved by REGISM, and vice versa.

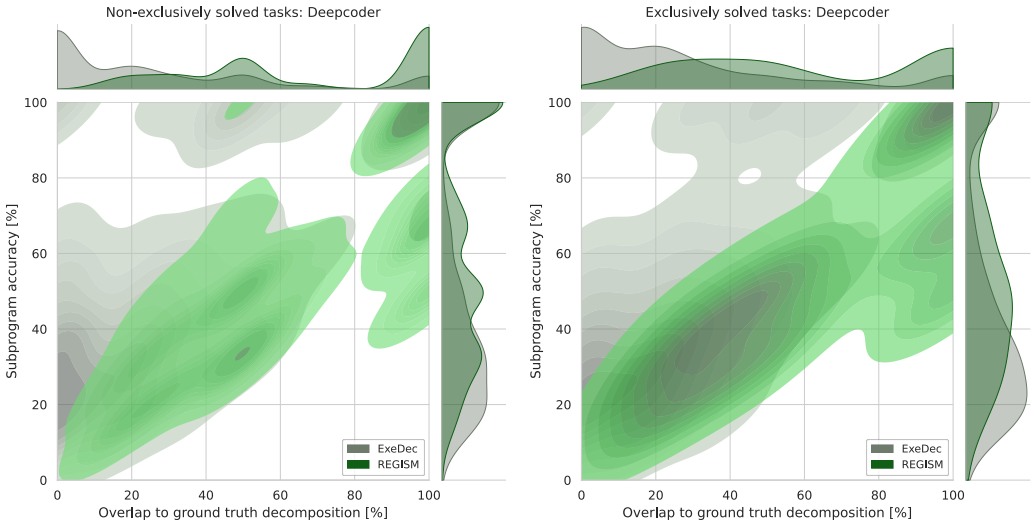


Figure 7: Average densities of ExeDec and REGISM across all compositional generalization categories and initializations. No clear differences between exclusively and non-exclusively solved tasks can be seen.

4.3 LIMITATIONS

While our findings highlight REGISM as a key component of ExeDec, two limitations warrant consideration. First, our evaluation is restricted to a single domain, which limits the generalizability of the results. Second, only one decomposition approach was tested, leaving open the question of how alternative methods might influence performance.

5 RELATED WORK

Programming-by-Example Neural networks play a crucial role in guiding program synthesis for PBE (Balog et al., 2016; Yin & Neubig, 2017; Lee et al., 2018). Many approaches divide synthesis into multiple steps, enabling iterative refinement or task planning. Search-based methods include top-down approaches that recursively expand partial solutions (Nye et al., 2019) and bottom-up methods that solve smaller subproblems before combining them into complete programs (Shi et al., 2022; 2023a; Odena et al., 2020). Planning-based techniques first generate abstract program structures, which are later refined into executable code (Murali et al., 2017; Nye et al., 2019; Chen et al., 2020; Hong et al., 2021; Klinger et al., 2023; Prasad et al., 2023; Zhang et al., 2023; Cano et al., 2023). Execution-guided approaches iteratively refine programs using intermediate feedback, such as partial executions or state traces (Ellis et al., 2019; Chen et al., 2018; Shrivastava et al., 2021). Recent advances in Large Language Models have further improved program synthesis by leveraging pre-training on large-scale corpora of code and natural language (Li & Ellis, 2024; Li et al., 2024; Tang et al., 2024). These models enhance synthesis frameworks by capturing semantic and structural patterns of code, aiding in both program generation and refinement.

Decomposition-based Program Synthesis Multi-step synthesis methods inherently break tasks into subtasks as part of their iterative process. In contrast, decomposition approaches explicitly partition a task *before* leveraging this structure to improve synthesis. Some methods model relationships between subprograms, enabling relational reasoning to identify dependencies and shared structures across subtasks (Hocquette & Cropper, 2023; Hocquette et al., 2024; Hocquette & Cropper, 2024). Similarly, hierarchical composition techniques systematically construct programs by progressively assembling higher-level abstractions from lower-level components (Liu et al., 2023).

6 CONCLUSION

Our results highlight ExeDec's strong performance in the Robustfill domain, where learning decompositions is more straightforward. At the same time, we demonstrate that the Synthesizer Model plays a critical role in enhancing program synthesis performance in the Deepcoder domain, even in the absence of both explicit and implicit decomposition guidance. Our findings further reveal that the Synthesizer Model inherently decomposes tasks, despite not being explicitly trained for this purpose. Thus, traditional performance metrics, such as the number of solved tasks, are insufficient for isolating the contributions of the decomposition model from those of the synthesis model. To more effectively attribute performance gains to each component, our work proposes using REGISM as an ablation method.

Hybrid approaches that integrate transductive and inductive strategies have shown promise in improving program synthesis (Li et al., 2024). ExeDec exemplifies such a hybrid model, where the Subgoal Model represents the transductive component and the Synthesizer Model embodies the inductive aspect, working together to enable compositional generalization. Our results show that both REGISM and ExeDec successfully solve a diverse range of tasks, suggesting potential benefits of framing ExeDec as a hybrid approach. However, identifying the specific conditions under which this combination is most effective remains an open question for future research.

ACKNOWLEDGMENTS

Janis Zenkner is supported by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) and the German Federal Ministry of Education and Research (BMBF).

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Leonardo Hernandez Cano, Yewen Pu, Robert D Hawkins, Josh Tenenbaum, and Armando Solar-Lezama. Learning a hierarchical planner from humans in multiple generations. *arXiv preprint arXiv:2310.11614*, 2023.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. *Advances in Neural Information Processing Systems*, 33:1690–1701, 2020.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Francois Chollet, Mike Knoop, Gregory Kamradt, and Bryan Landers. Arc prize 2024: Technical report. *arXiv preprint arXiv:2412.04604*, 2024.

Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pp. 345–356, 2016.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pp. 990–998. PMLR, 2017.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Steven M Frankland and Joshua D Greene. Concepts and compositionality: in search of the brain's language of thought. *Annual review of psychology*, 71(1):273–303, 2020.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Céline Hocquette and Andrew Cropper. Relational program synthesis with numerical reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 6425–6433, 2023.

Céline Hocquette and Andrew Cropper. Relational decomposition for program synthesis. *arXiv preprint arXiv:2408.12212*, 2024.

Céline Hocquette, Andreas Niskanen, Rolf Morel, Matti Järvisalo, and Andrew Cropper. Learning big logical rules by joining small rules. *arXiv preprint arXiv:2401.16215*, 2024.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, pp. 4308–4318. PMLR, 2021.

Tim Klinger, Luke Liu, Soham Dan, Maxwell Crouse, Parikshit Ram, and Alexander Gray. Compositional program generation for systematic generalization. *arXiv preprint arXiv:2309.16467*, 2023.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *arXiv preprint arXiv:2406.08316*, 2024.

Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, et al. Combining induction and transduction for abstract reasoning. *arXiv preprint arXiv:2411.02272*, 2024.

Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *International Conference on Machine Learning*, pp. 21672–21697. PMLR, 2023.

Gary F Marcus. *The algebraic mind: Integrating connectionism and cognitive science*. MIT press, 2003.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pp. 4861–4870. PMLR, 2019.

Augustus Odena, Kensen Shi, David Bieber, Rishabh Singh, Charles Sutton, and Hanjun Dai. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. *arXiv preprint arXiv:2311.05772*, 2023.

Kensen Shi, Hanjun Dai, Kevin Ellis, and Charles Sutton. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, 36:51327–51346, 2023a.

Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*, 2023b.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Learning to combine per-example solutions for neural program synthesis, 2021. *URL https://arxiv. org/abs/2106.07175*, 2021.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *arXiv preprint arXiv:2402.12275*, 2024.

Thaddäus Wiedemer, Prasanna Mayilvahanan, Matthias Bethge, and Wieland Brendel. Compositional generalization from first principles. *Advances in Neural Information Processing Systems*, 36, 2024.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv preprint arXiv:2303.05510*, 2023.

APPENDIX A FURTHER EXEDEC EXAMPLES

Task specification: $\{x0 = [42, -48] \rightarrow y = [42, 42],\ x0 = [-35, -21] \rightarrow y = [-35, -21],\ x0 = [39, 32] \rightarrow y = [39, 39]\}$
Ground Truth: `y = Scanl1 (max) x0`

Step 1:

- Predicted Subgoals: $[-48, 42], [-35, -21], [32, 29]$
- Subprogram: `x1 = Sort x0`
- Update: $x1 = [-48, -42], x1 = [-35, -21], x1 = [32, 39]$

Step 2:

- Predicted Subgoal: $[48, 42], [-35, -21], [39, 39]$
- Subprogram: `x2 = Zip (max) x0 x1`
- Execution: $x2 = [42, 42], x2 = [-35, -21], x2 = [39, 39]$

(a) Task 1

Task specification: $\{x0 = 1|x1 = [-2, -25, 1] \rightarrow y = [-2, -2, 1],\ x0 = 5|x1 = -4 \rightarrow y = -4,\ x0 = 2|x1 = [-28, -15] \rightarrow y = [-28, -15]\}$
Ground Truth: `y = Scanl1 (max) x0`

Step 1:

- Predicted Subgoals: $[-25, -2, 1], -4, [-28, -19]$
- Subprogram: `x2 = Sort x1`
- Update: $x2 = [-25, -2, 1], x2 = -4, x2 = [-28, -19]$

Step 2:

- Predicted Subgoal: $[-25, -23, -24], -4, [-28, -13]$
- Subprogram: `x3 = Scanl1 (-) x2`
- Execution: $x3 = [-25, -23, -24], x2 = -4, x2 = [-28, -13]$

Step 3:

- Predicted Subgoal: $[-25, -2, 22], -4, [-28, -15]$
- Subprogram: `x4 = Scanl1 (-) x3`
- Execution: $x4 = [-25, -2, 22], x2 = -4, x2 = [-28, -15]$

Step 4:

- Predicted Subgoal: $[-25, -25, 1], -4, [-28, -15]$
- Subprogram: `x5 = Zip (min) x1 x4`
- Execution: $x5 = [-25, -25, 1], x2 = -4, x2 = [-28, -15]$

Step 5:

- Predicted Subgoal: $[-2, -25, 1], -4, [-28, -15]$
- Subprogram: `x6 = Zip (max) x1 x5`
- Execution: $x6 = [-2, -25, 1], x6 = -4, x6 = [-28, -15]$

Step 6:

- Predicted Subgoal: $[-2, -2, 1], -4, [-28, -15]$
- Subprogram: `x7 = Zip (max) x2 x6`
- Execution: $x7 = [-2, -2, 1], x7 = -4, x7 = [-28, -15]$

(b) Task 2

Figure 8: Examples from the Deepcoder domain displaying the struggles of misleading subtasks. Both tasks can be solved with a single-step function that computes the cumulative maximum from the input list.

APPENDIX B ROBUSTFILL AND DEEPCODER DSLS

Both DSLs were taken from the original ExeDec study (Shi et al., 2023b).

$$
\begin{aligned}
\text{Program } P &:= \texttt{Concat}(e_1, e_2, \ldots) \\
\text{Expression } e &:= s \mid m \mid o \mid \texttt{ConstStr}(c) \\
\text{Compose } o &:= m_1(m_2) \mid m(s) \\
\text{Substring } s &:= \texttt{SubStr}(k_1, k_2) \mid \texttt{GetSpan}(r_1, i_1, b_1, r_2, i_2, b_2) \\
&\quad \mid \texttt{GetUpto}(r, i) \mid \texttt{GetFrom}(r, i) \mid \texttt{GetToken}(r, i) \\
\text{Modification } m &:= \texttt{ToCase}(a) \mid \texttt{Replace}(c_1, c_2) \mid \texttt{Trim()} \\
&\quad \mid \texttt{GetFirst}(r, i) \mid \texttt{GetAll}(r) \\
&\quad \mid \texttt{Substitute}(r, i, c) \mid \texttt{SubstituteAll}(r, c) \\
&\quad \mid \texttt{Remove}(r, i) \mid \texttt{RemoveAll}(r) \\
\text{Regex } r &:= \texttt{NUMBER} \mid \texttt{WORD} \mid \texttt{ALPHANUM} \mid \texttt{ALL_CAPS} \mid \texttt{PROPER_CASE} \\
&\quad \mid \texttt{LOWER} \mid \texttt{DIGIT} \mid \texttt{CHAR} \mid \delta \\
\text{Case } a &:= \texttt{ALL_CAPS} \mid \texttt{PROPER_CASE} \mid \texttt{LOWER} \\
\text{Position } k &:= -100 \mid -99 \mid \ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \mid 100 \\
\text{Index } i &:= -5 \mid -4 \mid \ldots \mid -1 \mid 1 \mid 2 \mid \ldots \mid 5 \\
\text{Boundary } b &:= \texttt{START} \mid \texttt{END} \\
\text{Character } c &:= A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid 0 \mid \ldots \mid 9 \mid \delta \\
\text{Delimiter } \delta &:= \texttt{\&}\ \texttt{,}\ \texttt{.}\ \texttt{?}\ \texttt{!}\ \texttt{@}\ \texttt{()}\ \texttt{[]}\ \texttt{\%}\ \texttt{\#}\ \texttt{\$}\ \texttt{"}\ \texttt{'}
\end{aligned}
$$

Figure 9: String manipulation functions that can be used to solve tasks from the Robustfill domain.

$$
\begin{aligned}
\text{Program } P &:= i_1;\ i_2;\ \ldots;\ a_1;\ a_2;\ \ldots \\
\text{Initialization } i &:= v \leftarrow \texttt{INPUT} \\
\text{Assignment } a &:= v \leftarrow f \mid v \leftarrow h \\
\text{First-Order Operation } f &:= \texttt{Head}(l) \mid \texttt{Last}(l) \mid \texttt{Access}(n, l) \mid \texttt{Minimum}(l) \mid \texttt{Maximum}(l) \\
&\quad \mid \texttt{Sum}(l) \mid \texttt{Take}(n, l) \mid \texttt{Drop}(n, l) \mid \texttt{Reverse}(l) \mid \texttt{Sort}(l) \\
\text{Higher-Order Operation } h &:= \texttt{Map}(\lambda, l) \mid \texttt{Filter}(\beta, l) \mid \texttt{Count}(\beta, l) \mid \texttt{Zip}(\Sigma, l, l) \\
&\quad \mid \texttt{Scanl1}(\Sigma, l) \\
\text{int} \rightarrow \text{int Lambda } \lambda &:= (+1) \mid (-1) \mid (*2) \mid (/2) \mid (*(-1)) \mid (**2) \mid (*3) \mid (/3) \mid (*4) \mid (/4) \\
\text{int} \rightarrow \text{bool Lambda } \beta &:= (>0) \mid (<0) \mid (\%2 == 0) \mid (\%2 == 1) \\
(\text{int}, \text{int}) \rightarrow \text{int Lambda } \Sigma &:= (+) \mid (-) \mid (*) \mid (\texttt{min}) \mid (\texttt{max}) \\
\text{Integer Variable } n &:= v \\
\text{List Variable } l &:= v \\
\text{Variable Name } v &:= x_1 \mid x_2 \mid \ldots
\end{aligned}
$$

Figure 10: First and Higher-Order Functions contained in the DSL for the Deepcoder domain.

APPENDIX C BENCHMARK CREATION

Shi et al. (2023b) define tasks based on compositional generalization categories by structuring programs according to their length and the concepts they employ.

C.1 ROBUSTFILL

In the Robustfill domain, each program is composed of concatenated subprograms, and program length is determined by the number of these subprograms.

- Length Generalization: Training tasks consist of programs with lengths ranging from 1 to 6, while testing tasks feature longer programs with lengths between 7 and 10.
- Compose Different Concepts: The operations are categorized into two concepts: substring-related operations form the "substring concept," while modification operations and constant strings belong to the "non-substring concept" (excluding the Compose operation). Both training and testing programs have lengths between 2 and 6.
- Switch Concept Order: Training tasks enforce a structure where the first half of the subprograms belong to the substring concept, and the latter half to the non-substring concept. In testing tasks, this order is reversed. Program lengths remain between 2 and 6.
- Compose-New-Operation: A quarter of the training tasks are single-subprogram programs that exclusively contain the `Compose` operation. The remaining training tasks consist of programs of length 2-6 without Compose. Testing tasks include length 2-6 programs that incorporate the `Compose` operation.
- Add Operation Functionality: Training tasks consist of programs of length 1-6 where substring operations are never used within the `Compose` operation. Testing tasks maintain the same length range but include cases where substring operations appear within `Compose`.

C.2 DEEPCODER

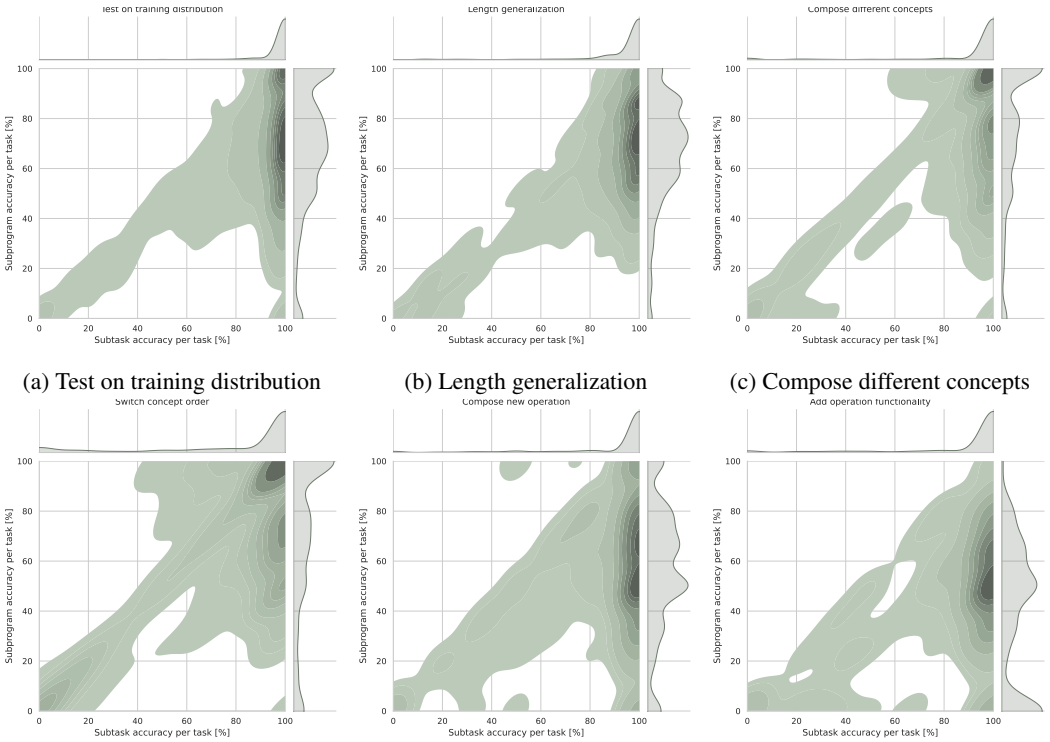
In the Deepcoder domain, programs are line-by-line and task length is defined as the number of non-input lines in the program.

- Length Generalization: Training tasks consist of programs with lengths between 1 and 4, while testing tasks feature programs of length 5.
- Compose Different Concepts: Operations are divided into two categories: the first concept includes all first-order operations and the `Map` operation, while the second concept consists of all higher-order operations. Both training and testing programs have lengths between 1 and 4.
- Switch Concept Order: Training tasks follow a structure where the first half of the subprograms belong to the first-order operations and `Map` concept, while the latter half use higher-order operations. In testing tasks, this order is reversed. Program lengths remain between 1 and 4.
- Compose New Operation: A quarter of the training tasks consist of length 1 programs that exclusively contain the `Scanl1` operation. The remaining training tasks include length 2-4 programs that do not use `Scanl1`. Testing tasks include length 2-4 programs that incorporate the `Scanl1` operation.
- Add Operation Functionality: Training tasks consist of programs of length 1-4, where `Scanl1` is only applied with the lambda functions `(-)` and `(min)`. In testing tasks, `Scanl1` is used with additional lambda functions `(+)`, `(*)`, and `(max)`.

APPENDIX D COMPOSITIONAL GENERALIZATION ANALYSIS

In the Robustfill domain, the majority of tasks is concentrated along $x = 100\%$, representing tasks where subtasks are predicted as intended by the ground truth but the implementation differs, showcasing the Subgoal Model's accuracy in learning decompositions (Figure 11). The densities decrease as you move towards the bottom-left corner, indicating fewer cases that deviate semantically and syntactically from the ground truth. Yet, these tasks are solved using - at least partially - alternative decompositions. Interestingly, the densities do not differ clearly between the compositional generalization tasks, even though the performance varies significantly.

In the Deepcoder domain (Figure 12), four distinct clusters can be observed. While small clusters exist in the top-right corner, indicating tasks solved as intended by the ground truth, most tasks are



(a) Test on training distribution (b) Length generalization (c) Compose different concepts

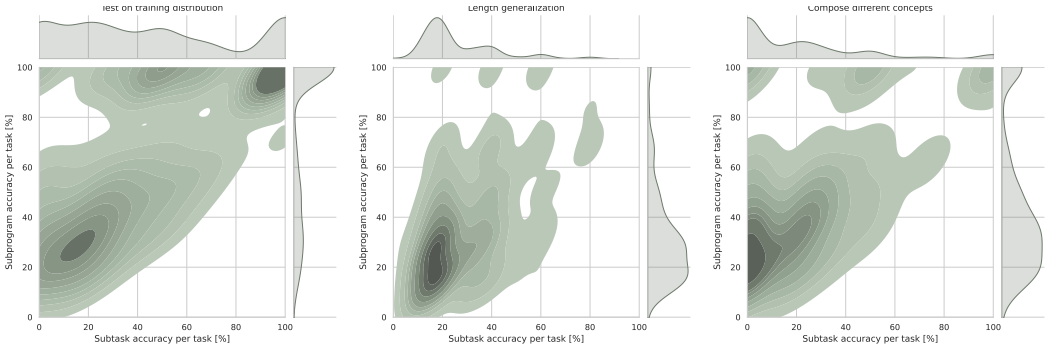
(d) Switch concept order (e) Compose new operation (f) Add operation functionality

Figure 11: Tasks solved in the Robustfill domain, clustered across subtask and subprogram accuracy.

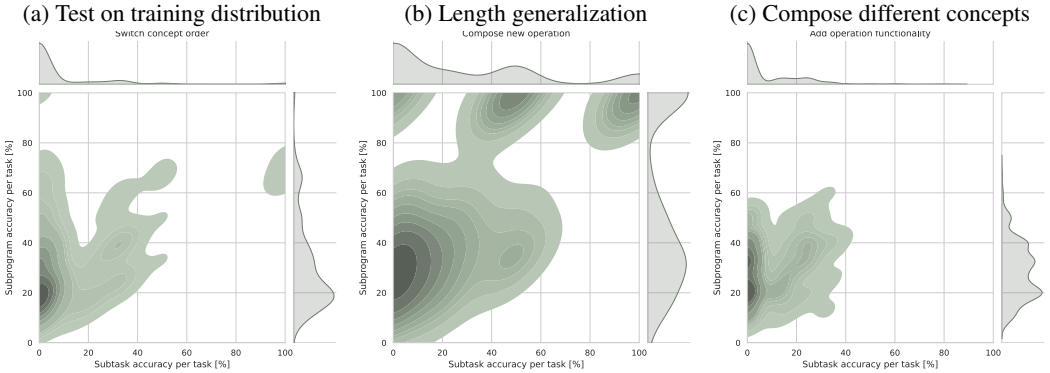
solved by using alternative decompositions and implementations (bottom-left corner). Again, no clear differences between the compositional generalization tasks exist. Yet, the subgoal accuracy distribution is more uniform when tested on the training distribution. This indicates that the Subgoal Model can learn and apply decomposition patterns in this task category more easily.

The differences between the Robustfill and Deepcoder domains are evident in the decomposition process. This becomes again clear when comparing the number of decompositions to the ground truth. In the Robustfill domain, the number of predicted decompositions aligns closely with the ground truth, where the predicted solutions almost exactly match the number of decompositions required by the target solution (Figure 13b). This suggests that the solutions closely match the target program, likely due to the relatively straightforward nature of decomposition learning in the Robustfill domain. In contrast, ExeDec's Subgoal Model in the Deepcoder domain decomposes tasks into significantly more subtasks than specified by the ground truth (Figure 4). This leads to more frequent invocations of the Synthesizer Model, which appears to be a key driver of the observed performance improvements. The increased number of decompositions reflects the greater challenge of learning decomposition in the Deepcoder domain, where tasks often require a more nuanced understanding and finer granularity of subgoal generation.

Furthermore, the simpler structure of the Robustfill domain, where intermediate states remain constant, contrasts with Deepcoder's dynamic intermediate states and complex task structures. This dynamic nature better mirrors how humans approach coding tasks. However, the Subgoal Model performs significantly worse in the Deepcoder domain, highlighting challenges in capturing this human-like reasoning. Due to the ambiguous performance of the Subgoal Model in the Deepcoder domain, we focused on this domain in subsequent experiments to better understand the factors influencing its behavior and to explore potential enhancements.

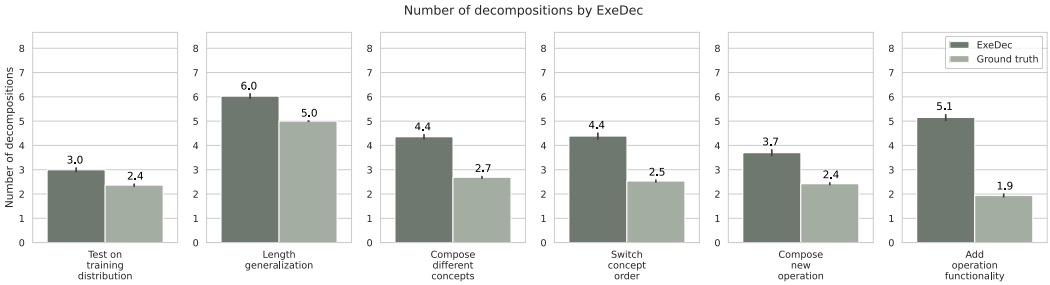


(a) Test on training distribution (b) Length generalization (c) Compose different concepts

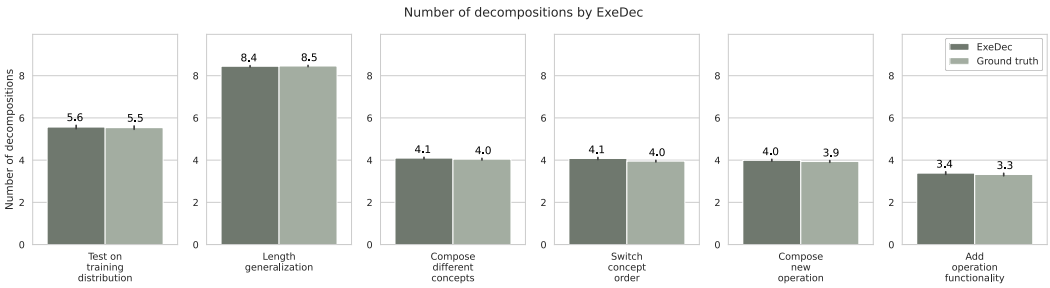


(d) Switch concept order (e) Compose new operation (f) Add operation functionality

Figure 12: Tasks successfully solved in the Deepcoder domain, clustered according to subtask- and subprogram-level accuracy.



(a) Deepcoder Domain



(b) Robustfill Domain

Figure 13: Comparison of the number of decompositions performed by ExeDec across different domains.